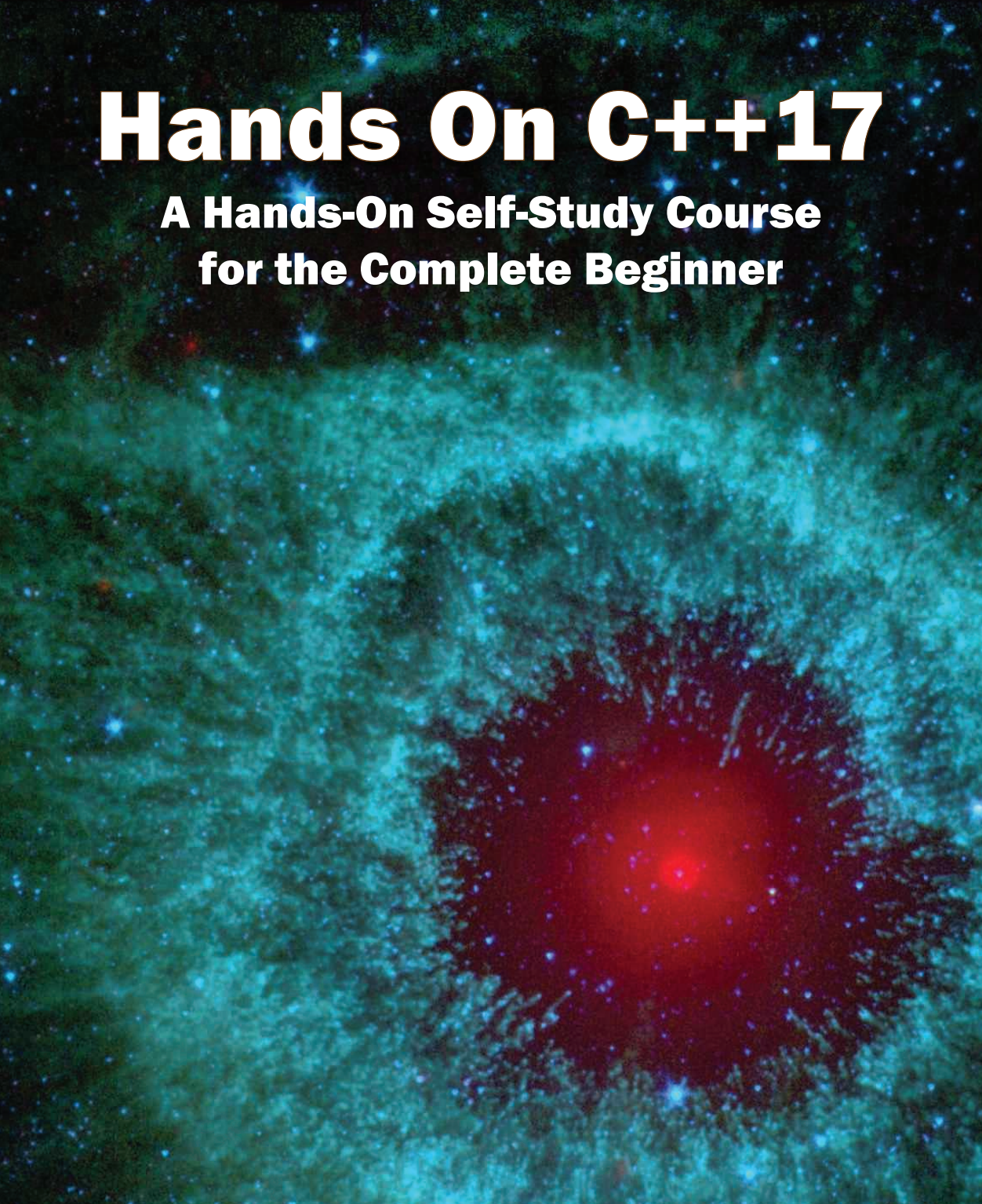
Hands On C++17

A Hands-On Self-Study Course for the Complete Beginner



Alistair Stewart

Digital Skills

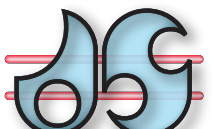
www.digital-skills.co.uk

Hands On C++17

Alistair Stewart

Digital Skills
Milton
Barr
Ayrshire
KA26 9TY
UK

www.Digital-Skills.co.uk



www.digital-skills.co.uk
tel: +44(0)1465 861 638

Copyright © Digital Skills 2020

All rights reserved.

No part of this work may be reproduced or used in any form without written permission from Digital Skills.

Although every effort has been made to ensure accuracy, the author and publisher accept neither liability nor responsibility for any loss or damage arising from the information in this book.

Cover image: NASA

First Published March 2020

Title: Hands On C++17

ISBN: 978-1-874107-26-2

Comments

Please take the time to help improve this book by emailing us any constructive comments you have on its contents.

Any mistakes, omissions or suggests will be processed and where necessary the text will be updated.

Any minor/medium updates will be made available for free to those who have already purchased this ebook.

email: comments@digital-skills.co.uk

Text Style

In this text code is shown in blue monospaced type:

```
in = GetUserInput()
HandleUSerInput(in)
HandleOther()
Sync()
```

as is any reference to C++-defined terms with the descriptive paragraphs.

Function names are always shown with terminating parentheses to distinguish them from variable names. C++ keywords, code, and function names are in `blue monospaced` type while user-defined function names are shown in *italics*.

Important terms, when used for the first time are shown in **bold**.

Lines of code which, because of space restrictions, have had to be spread over two or more lines, start subsequent lines with the ↳ symbol – this symbol is not part of the code.

This text is designed primarily as a tutorial. As such many Activities build on work that has been carried out on previous pages, so jumping straight in to a specific section may sometimes make for a confusing read!

Foreword - Alistair Stewart

Welcome to the first edition of **Hands On C++17**. It's more than 25 years since the original **Hands On C++** book was published. Back then the text was written assuming you were using Borland C++. Unfortunately, Borland are no more, so this version covers only standard C++ and the programs should run with little or no change on any platform. However, the examples have been tested using Microsoft Visual C++ 2019 Community edition but care has been taken not to include any C++ extensions. The only parts of the text that are specific to Visual C++ are the few pages on using the IDE and creating a static library.

The book has been written as an in-depth, hands-on, tutorial for anyone who is either starting out in programming in general or in C++ in particular.

How to Get the Most Out of this Book

Is learning the basics of computer programming difficult? No, but you do have to put in the effort. Despite other publications promising to have you expert in a day, or a week, I'm sure you're smart enough to know that's not going to happen. So, let's get real: you'll learn how to program using C++17 if you put in the work, take your time to make sure you understand something before moving on, and practice, practice, practice.

We've tried to keep things interesting by giving you plenty of practical work to do as you journey through this book, but feel free to try out your own projects as well.

The first two chapters are the only ones in which you won't be programming since they concentrate on the basic concepts behind all computer programming. You can, if you wish, work on the third chapter at the same time as you read through Chapters 1 and 2. That way, you'll be able to start programming right away.

Take your time with each chapter. Make sure you do each of the activities: they are there to give you a deeper understanding as well as to keep you actively involved. Since most activities require you to create a program, the computer will let you know if you've got it right, but you should still take the time to look at the activity's solution given at the end of the chapter. The solution given may differ from your own but it's always of use to see how others tackle the same problem.

Don't be afraid to reread a section or a whole chapter - it's the second or third reading of something new that finally gets the information across to most people.

Support Material

There are a few executable files accompanying this book. These are designed as aids to the learning process but are not required for any of the programming exercises.

You can download a ZIP file containing the support material from our website (*www.digital-skills.co.uk*). Go to the *Downloads* page for C++ to find the link to the file.

Thanks

I'd like to thank **Virginia Marshall** for making sure the English always made sense.

Also, thanks to **Alex Kladitis** for his technical assistance and useful suggestions.

Other Books

Other books by Alistair Stewart available from *www.digital-skills.co.uk*

Hands On AppGameKit Studio Volume 1

Hands On AppGameKit Studio Volume 2

Contents

Algorithms

Designing Algorithms 2
- Following Instructions 2
- Solving Problems 2
- The Nature of Algorithms 5
- Sequence 5
- Selection 6
- Iteration 14
- Complex Conditions 19
- Data 24
- Counts and Totals 26
- Levels of Detail 28
- Checking for Errors 31
- Summary 35

Support Material for this Chapter 37
- Algorithm Constructor (*TriLogic.exe*) 37
- Algorithm Tracer (*CardAlgorithm.exe*) 38
- Reviewer (*Remember01.exe*) 39

Solutions 40

Background Intel

Number Systems 44
- Introduction 44
- Computers and the Binary System 44
- Converting Fractions 47
- Hexadecimal 49
- Octal 52
- Storing Numbers 52
- Negative Numbers in Binary 54
- Floating Point Values in Binary 56
- Character Coding 58
- Summary 59

Support Material for this Chapter 61
- Integer Number Converter (*Numbers.exe*) 61
- Reviewer (*Remember02.exe*) 62

Solutions 63

Starting C++17

Using Visual Studio 66
- Before We Start 66

Creating a Microsoft Account....66
Downloading the Software....67
Start Up....69
A New Project....69
A Second Program....77
Summary....79
Solution....81

Data

Variables and Constants....84
Introduction....84
Memory Allocation....84
Basic Data Types....84
Data Type Variations....85
Naming Variables....88
Declaring Variables....88
Constants....93
The Assignment Statement....97
Type Conversions....104
Type Aliases....109
Lvalues and Rvalues....109
Summary....110
Solutions....114

I/O and Calculation

Input/Output....116
Introduction....116
Input....116
Output....120
About Namespaces....128
Summary....130
More Operators....131
The Arithmetic Assignment Operators....131
The Comma Operator....133
Multiple Assignment....134
The ***sizeof*** Operator....134
Updated Operator Precedence Table....135
Summary....135
Comments....136
A Guessing Game....136
Day of the Week Program....138
Introduction....138
The Calculation....138

Implementing a Solution 140
Solutions 143

Selection

Introduction 146
The ***if*** Statement 146
Variable Scope 149
Testing a Simple ***if*** Statement 153
Compound Conditions 153
Boolean Variables 156
Strange Things about Boolean Values 156
Non-Boolean Expressions 156
if...else 157
Multiple ***if*** Statements 159
Nested ***if*** Statements 160
The Conditional Assignment Statement 163
The ***switch*** Statement 164
Summary 171
Solutions 174

Iteration

Iteration 180
Introduction 180
The ***while*** Statement 180
The ***do...while*** Statement 183
The ***for*** Statement 185
Jump Statements 191
Largest and Smallest Algorithms 193
Nested Loops 195
Summary 198
Solution 200

Arrays

Arrays 206
Problems with Simple Variables 206
One Dimensional Arrays 206
Using One Dimensional Arrays 215
Character Arrays 232
Multi-dimensional Arrays 235
3-Dimensional Arrays and Higher 240
Constant Arrays 241
Summary 241
Solutions 243

Other Data Structures

Composite Data Structures....250
Introduction....250
Setting Up a Record....250
Accessing Record Fields....254
Nested Records....256
Summary....257
Enumerated Types....259
Introduction....259
Using Enumerated Types....259
Using Enumerated Variables....261
Summary....263
Bits....265
Bit Fields....265
Categorising Bits....267
Bit-Manipulation Operators....267
Summary....273
Union....274
Concept....274
Defining a Union....274
Summary....276
Type Aliases....277
Introduction....277
Defining an Alias....277
Pairs and Tuples....278
Introduction....278
Coding Pairs....278
Coding Tuples....281
Summary....283
Structured Binding....285
Introduction....285
Demonstrating Structured Binding....285
Solutions....287

Pointers and References

Raw Pointers....292
Introduction....292
Pointer Variables....293
A Pointer to a Pointer....300
Pointers and Arrays....301
Pointers, Structs and Unions....306
Pointers, Chars and Strings....308

Pointer Arithmetic....310
Pointers and Dynamic Allocation....311
void Pointers....316
Summary....318
Smart Pointers....320
Introduction....320
unique_ptr Pointers....320
shared_ptr Pointers....323
Smart Pointers to Memory Blocks....324
Summary....325
Reference Variables....327
Introduction....327
Using a Reference Variable....327
Aliases for Pointers and Reference Types....330
Summary....330
Solutions....332

User-Defined Functions – 1

User-Defined Functions - 1....336
Introduction....336
Writing our own Functions....336
Value Parameters....341
Pre-conditions and Post-conditions....347
Coding Pre-Conditions....348
Returning a Value from a Function....349
Returning a ***pair*** or ***tuple*** Value....353
Reference Parameters....354
const Parameters....358
Pointer Parameters....360
Array Parameters....361
Returning an Array....364
Returning Reference and Pointer Values....365
Functions and Smart Pointers....370
auto Return....371
Function Attributes....372
Function Overheads....373
Summary....374
Testing Functions....376
Introduction....376
Black Box Testing....376
Boundary Testing....378
White Box Testing....379

Combining Test Requirements 379
Test Drivers 381
Function Complexity 382
Summary 384
Solutions 385

User-Defined Functions - 2

User-Defined Functions - 2 394
Introduction 394
Default Parameter Values 394
Inline Functions 395
Static Variables 397
Lifetime and Scope 399
Function Overloading 401
Operator Overloading 404
Recursive Functions 408
Pointers to Functions 410
main() 414
Summary 417
Data Type Related Functions 419
Introduction 419
Defining the Data Structure 419
Defining the Operations 419
Summary 426
Function Templates 427
Introduction 427
Creating a Function Template 427
Using a Function Template 428
Other Considerations 430
Summary 430
Variadic Functions 432
Introduction 432
Implementation 432
Fold Expressions 436
Summary 438
Solutions 440

Standard Function Library

Standard Functions 452
Introduction 452
Maths Functions 452
Time Functions 457
Random Number Functions 462

System Functions....464
Character Input Functions....465
Summary....466
Solutions....468

Strings and char Functions

C-Style String Functions....472
Introduction....472
Defining Strings....472
String Handling Functions....473
String Conversion Functions....480
Summary....483
Character Functions....485
Introduction....485
char Classification Functions....485
char Modifier Functions....489
Summary....489
Solutions....491

Traditional File Handling in C

Traditional File Handling in C....494
Introduction....494
File Formats....495
Using Text Files....496
Binary Files....505
File Management....514
Printer Output....515
Summary....515
Solutions....517

Starting Classes

Classes and Objects....524
Introduction....524
What is an Object?....524
Object-Oriented Programming....525
Classes in C++....527
Overloading Methods....533
Constructors and Destructors....534
Data Hiding....540
Overloading Operators....542
The Call Operator and Functors....543
Inline Methods....544
The ***this*** Pointer....545

Class Constants 547
Static Properties 548
Overloading << and >> 553
More Friends 556
More on Class Diagrams 556
Visualising An Object 558
Student Class 559
Object Pointers 568
Objects and Arrays 569
Time Class 569
Summary 571
Solutions 574

Creating Static Libraries

Organising Code 594
Introduction 594
Header and Implementation Files for a Class 602
Using Header and Obj Files in Other Projects 604
Summary 608
Creating a Static Library 610
Introduction 610
Creating a Library 610
Using a Static Library 614
Namespace 619
Summary 621
Solutions 623

Error Handling

Introduction 632
Other Options 633
Summary 635
Exception Handling 636
Introduction 636
Exception Handling in C++ 636
Throwing an Exception from a Function 638
Unhandled Exceptions 649
Summary 650
Solutions 652

Container Classes

Creating a Container Class 658
Introduction 658
A Set Container Class 658

Implementing a ***Set*** Class 659
Template Classes 663
Summary 668
STL Container Classes 669
Introduction 669
Iterators 669
The ***array*** Container Class 671
Introduction 671
Declaration 671
Initialising an ***array*** Object 671
Accessing ***array*** Object Elements 672
array Class Methods 672
array Class Iterators 676
Iterator Operators 679
Multi-Dimensional ***array*** Objects 680
Summary 680
The ***vector*** Container Class 683
Introduction 683
Declaration 683
Accessing ***vector*** Object Elements 684
vector Class Methods 684
Iterator-Based Methods 689
vector Objects as Function Parameters 694
Vector Dangers 695
Multi-Dimensional Vectors 697
Summary 698
Solutions 701

Container & Iterator Functions

The Algorithm Package 708
Introduction 708
Creating a Test Data Container 708
Algorithm Package Non-Modifying Functions 708
Algorithm Package Modifying Functions 722
Sorting and Searching 734
Summary 736
Iterator-Based Functions 739
Introduction 739
Functions 739
Summary 742
Solutions 743

The string Class

The string Class 750
Introduction 750
Creating a ***string*** Object 750
I/O Operations 751
Interrogating a String 752
string Class Operators 753
string Class Comparisons 755
Accessing Characters 758
Search and Replace Operations 760
Adding to a String 767
Deleting from a String 769
Converting to a C-Style String 770
Container-Related Methods 770
Iterator Functions and Strings 773
Converting to/from a Numeric Value 774
Other ***String*** Classes 775
Summary 776
Solutions 779

Class Relationships - 1

Inheritance 784
Relationships Between Classes 784
Implementing Derived Classes 786
Hiding Inherited Methods 789
Methods that are not Inherited 794
Constructors in Derived Classes 794
Destructors in Derived Classes 797
The Assignment Operator 797
Overloading Friend Functions 798
The ***USDate*** Class 799
The ***Clock*** Class 801
The ***CountdownClock*** Class 806
Static Attributes in Derived Classes 807
Inheritance Options 807
Multiple Inheritance 808
Summary 813
Solutions 815

Class Relationships - 2

Pointers to Objects 824
Pointer Review 824

Object Pointers....824
Virtual Methods....826
Virtual Classes....831
Summary....833
Abstract Classes....834
Introduction....834
Creating an Abstract Class....834
Coding an Abstract Class....835
Summary....843
Aggregation and Composition....845
Introduction....845
Graphical Representation....845
Programming Composition and Aggregation....845
Summary....849
C++ Exception Classes....850
Introduction....850
C++ Exceptions....850
User-Defined Exceptions....850
Catching Derived Classes....851
Solutions....853

I/O Classes

I/O Classes....864
Introduction....864
The ***ios_base*** Class....864
ios Class....870
istream Class....874
The ***ostream*** Class....883
The ***ofstream*** Class....884
The ***ifstream*** Class....887
Using Binary Files....890
The ***iostream*** Class....894
The ***fstream*** Class....894
Saving Objects....896
The ***ostringstream*** Class....898
The ***istringstream*** Class....900
The ***stringstream*** Class....901
Using String Streams....902
Summary....902
Solutions....908

Program Efficiency

Data Flexibility....914

Introduction 914
decltype and ***declval*** ***914***
variant 917
any 921
optional 924
Summary 927
Program Efficiency 930
Execution Time Efficiency 930
Move Semantics 930
Lvalues and Rvalues Again 931
Rvalue Reference Type 934
Rvalue References and Move Semantics 935
Using a Smart Pointer 940
Summary 941
Solutions 942

Lambdas

Introduction 948
Lambdas in C++ 948
Using a Lambda as a Function Parameter 957
Functions that Return Lambda Pointers 958
Using a Lambda in a Class Method 959
Summary 960
Solutions 961

1

Algorithms

In this Chapter:

- ❑ Understanding Algorithms
- ❑ Creating Algorithms
- ❑ Control Structures
- ❑ Boolean Expressions
- ❑ Data Types
- ❑ Stepwise Refinement
- ❑ The Need for Testing

Designing Algorithms

Following Instructions

Activity 1.1

Carry out the following set of instructions in your head.

Think of a number between 1 and 10
Multiply that number by 9
Add up the individual digits of this new number
Subtract 5 from this total
Think of the letter at that position in the alphabet
Think of a country in Europe that starts with that letter
Think of a mammal that starts with the second letter of the country's name
Think of the colour of that mammal

Congratulations! You've just become a human computer. You were given a set of instructions which you have carried out (by the way, did you think of the colour grey?).

That's exactly what a computer does. We give it a set of instructions, the machine carries out those instructions - and that is ALL a computer does. If some computers seem to be able to do amazing things, that is only because someone has written an amazingly clever set of instructions. A set of instructions designed to perform some specific task (like that in Activity 1.1) is known as an **algorithm**.

A clear and concise algorithm should have the following characteristics:

- One instruction per line
- Each instruction should be clear and unambiguous
- Each instruction should be as brief as possible

Solving Problems

As programmers, we will normally be presented with a problem for which we are expected to come up with an efficient algorithm and computer program.

The first step is to make sure that the problem to be tackled has been stated in a clear and unambiguous way. If it isn't, ask for a clearer, more detailed description of what is required.

The second step is to make sure you understand the problem. If you don't, try reading the problem again or getting help from others.

The third step is to develop some method of tackling the problem. Remember, the first ideas you come up with are often not the best ones!

The final step is to create a solution to the problem - the algorithm.

Let's take a favourite logic problem and see how we can develop a solution:

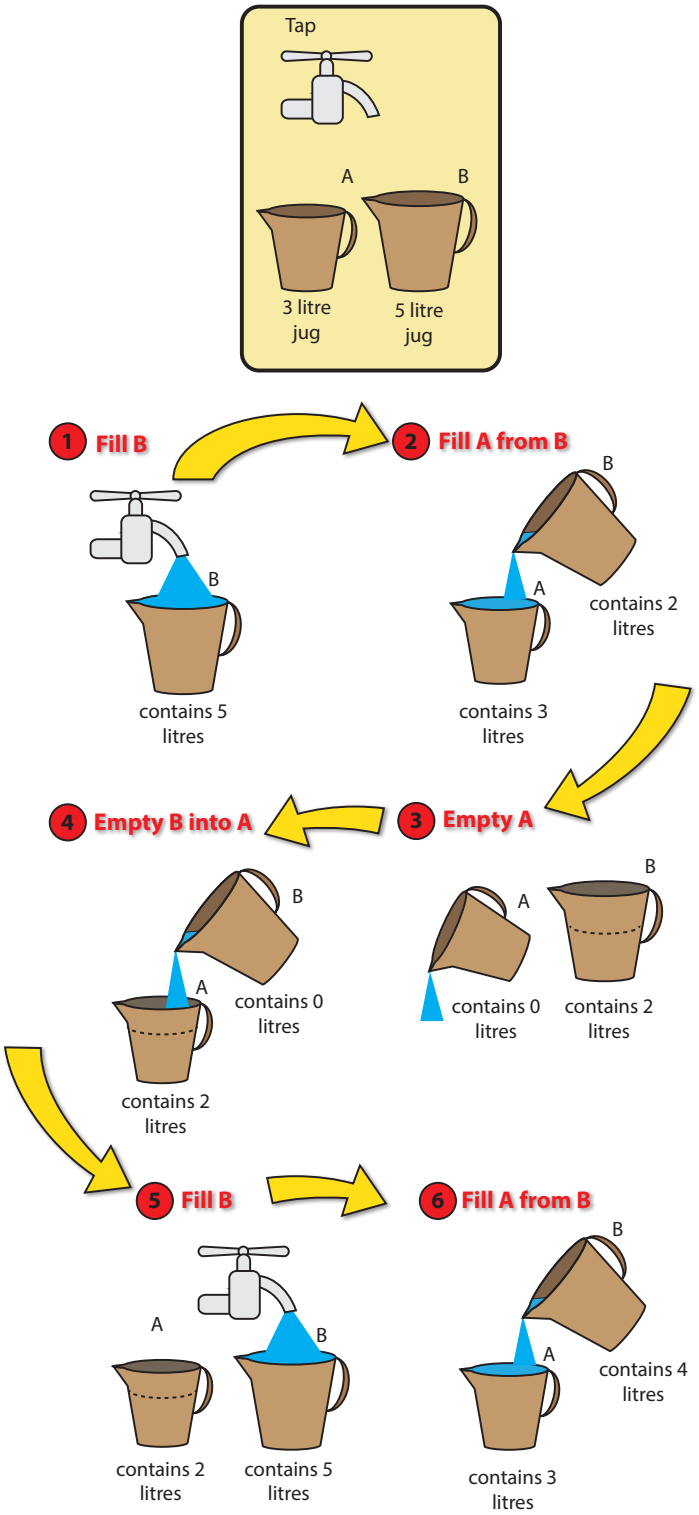
A container (labelled A) holds exactly 3 litres of water when full. A second container

(labelled B) holds exactly 5 litres. With an unlimited water supply, measure out exactly 4 litres of water.

Assuming we understand the problem, FIG-1.1 shows how we might go about tackling the problem.

FIG-1.1

Solving the Four Litre Problem



The solution to the task can now be written as the following algorithm:

```
Fill B
Fill A from B
Empty A
Empty B into A
Fill B
Fill A from B
```

Activity 1.2

Often there will be more than one way to arrive at a solution to a problem. Write an alternative solution to the 4 litre problem by starting with the instruction

```
Fill A
```

As you can see, there are at least two ways to solve the original problem. Is one better than the other? Well, if we start by filling container B, the solution needs less instructions, so that might be a good guideline at this point when choosing which algorithm is best.

Activity 1.3

Write an algorithm for the following problem:

A traveller arrives at a river he must cross with a wolf, goat and cabbage. He has access to a boat to cross the river and reach home. However, the boat can only carry him and one of his belongings. Normally, he would just make several crossings to get the wolf, goat and cabbage to the other side, but, if left alone together, the wolf would eat the goat and the goat would eat the cabbage.

How does the traveller get all three possessions safely to the other side of the river?

Computer Programs

The algorithms that a computer carries out are not written in English like the instructions shown above, but in a more stylised form using a **computer programming language**. C++ is one such language. For example, the code below displays the result of the calculation 12 x 3.

```
num1 = 12;
num2 = 3;
answer = num1 * num2;
cout << answer;
```

The set of program language instructions which make up each algorithm is then known as a **computer program** or **software**.

Just as we may perform a great diversity of tasks by following different sets of instructions, so the computer can be made to carry out any task for which a program exists.

Computer programs are normally **copied** (or **loaded**) from a magnetic disk or flash storage into the computer's memory and then **executed** (or **run**). Execution of a

program involves the computer performing each instruction in the program one after the other. This it does at impressively high rates, possibly exceeding 300,000 million (or 300 billion) instructions per second (usually written as 300,000 **mips**).

Depending on the program being run, the computer may act as a word processor, a database, a spreadsheet, a game, a musical instrument or one of many other possibilities. Of course, as a programmer, you are required to design and write computer programs rather than use them. The coded examples in this text will be mainly short applications designed mainly to highlight some feature of the C++17 language.

The Nature of Algorithms

Although writing algorithms and programming computers can be complicated tasks, there are only a few basic concepts and statements which you need to master before you are ready to start producing software. Luckily, many of these concepts are already familiar to you in everyday situations. If you examine any algorithm, no matter how complex, you will find it consists of only three basic structures:

- **Sequence** where one instruction follows on from another.
- **Selection** where a choice is made between two or more alternative actions.
- **Iteration** where one or more instructions are carried out over and over again.

These structures are explained in detail over the next few pages. All that is needed is for us to move from the rather free-style way we might express these structures in everyday English to the more formalised style used for writing algorithms. This formalisation better matches the structures used within a computer program.

Sequence

A set of instructions designed to be carried out one after another, beginning at the first and continuing, without omitting any, until the final instruction is completed, is known as a **sequence**. For example, the solutions to the 4 litre problem and traveller problems were both examples of a sequence.

Activity 1.4

Download the file containing support material for this book from *www.digital-skills.co.uk* (you'll find a link on the **C++** *Downloads* page) and unzip the file.

From the folder *CPP/Resources/Ch01/TriLogic,* run *TriLogicGame.exe,* press the **Start** button. Click on the tokens to construct the lines of an algorithm that moves the counters from position 1 to position 3.
A typical statement would be

```
MOVE C1 TO P2
```

Note that a larger piece must never be placed on top of a smaller piece.

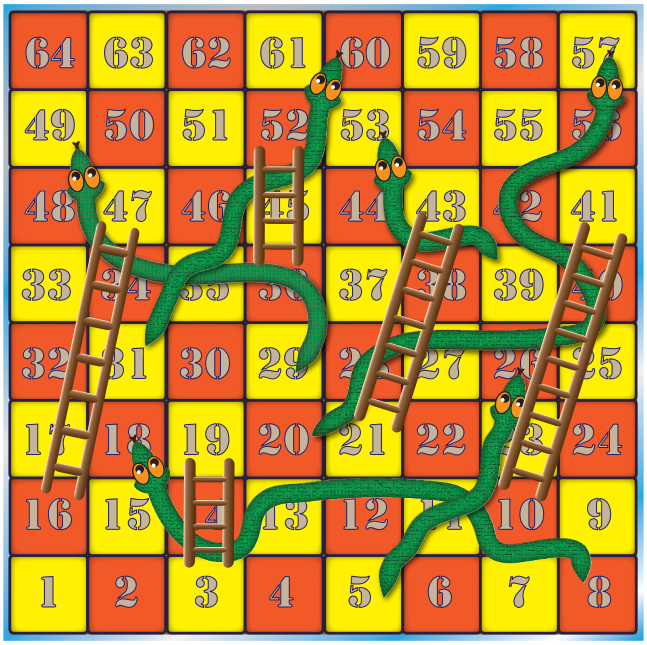
As you can see from the *TriLogic* puzzle, you are expected to construct the instructions in a very specific format. If you deviate from that format, you'll get an error message. This is typical of any true programming language: each statement must be constructed according to strict rules.

Binary Selection

Often a group of instructions in an algorithm should be carried out only when certain circumstances arise. For an example of this, consider the board game of Snakes and Ladders (see FIG-1.2).

FIG-1.2

A Snakes and Ladders Board



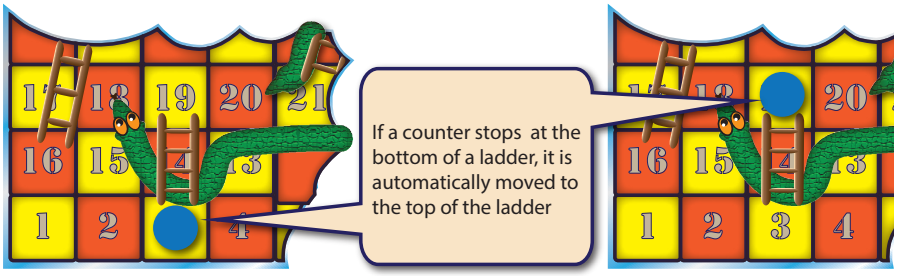
Each player has a counter which moves along the board by an amount determined by the throw of a die. The aim of the game is to be the first to reach the final square (square 64). We could describe a move as

```
Throw die
Move counter forward by the number thrown
Pass die to next player
```

However, when a player's counter stops on a square at the bottom of a ladder, it moves to the square at the top of the ladder (see FIG-1.3).

FIG-1.3

Snakes and Ladders: Moving Up a Ladder



We might explain this rule with an instruction such as:

```
When a counter stops at the bottom of a ladder, move the counter to the top of the ladder
```

Notice that the statement consists of two main components:

a condition : a counter stops at the bottom of a ladder

and

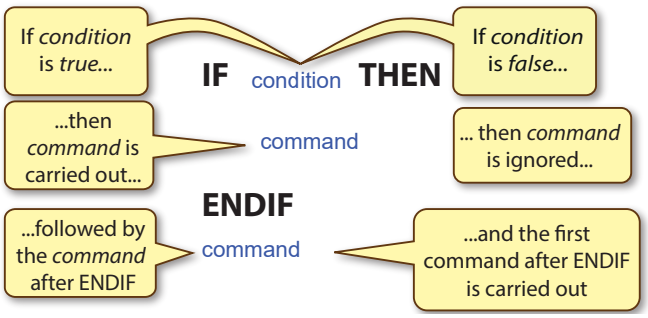
a command : move the counter to the top of the ladder

A **condition** (also known as a **Boolean expression**) is a statement that is either *true* or *false* at a given moment in time. The **command** given in the statement is only carried out if the condition is *true* at that particular moment and hence this type of instruction is known as an **IF statement**.

Although we could rewrite the above instruction in many different ways, when we produce a set of instructions in a formal manner, as we are required to do when writing algorithms, then we use a specific layout as shown in FIG-1.4, always beginning with the word IF.

FIG-1.4

The IF Statement



It is important to realise that there are two alternative options to this structure: to carry out the command or to ignore it. From this we get the formal name for the IF statement - **binary selection**.

Notice that the layout of this instruction makes use of three terms that are always included. These are the words **IF**, which marks the beginning of the instruction; **THEN**, which separates the condition from the command; and finally, **ENDIF** which marks the end of the instruction.

The indentation of the command is important since it helps our eye grasp the structure of the instruction. Appropriate indentation is particularly valuable in aiding readability once an algorithm becomes long and complex. Using this layout, the instruction for our Snakes and Ladders game would be written as:

```
IF counter stops at the bottom of a ladder THEN
    Move counter to top of ladder
ENDIF
```

Sometimes, there will be several commands to be carried out when the condition specified is met. For example, in the game of Scrabble we might describe a turn as:

```
IF you can make a word THEN
    Add the word to the board
    Work out the points gained
    Add the points to your total
    Select more letter tiles
ENDIF
```

Of course, the IF statement will almost certainly appear within a longer set of instructions. For example, we could now write the instruction for a single move in Snakes and Ladders as:

```
Roll die
Move counter forward by the number thrown
IF counter stops at the bottom of a ladder THEN
      Move counter to top of ladder
ENDIF
Pass die to next player
```

This longer list of instructions highlights the usefulness of the term ENDIF in separating the conditional command, Move counter to top of ladder, from subsequent unconditional instructions, in this case, Pass die to next player.

Activity 1.5

A simple game involves two players. *Player 1* thinks of a number between 1 and 100, then *Player 2* makes a single attempt at guessing the number. *Player 1* responds to a correct guess by saying **Correct**. If the guess is incorrect, *Player 1* makes no response. The game is then complete and *Player 1* states the value of the number he thought of.

Write the set of instructions necessary to play the game. In your solution, include the statements:

```
Player 1 says "Correct"
Player 1 thinks of a number
IF guess matches number THEN
```

An algorithm may contain many separate IF statements if the logic requires them. For example, in Snakes and Ladders, another rule is that counters that stop at the head of a snake must move to the tail of the snake.

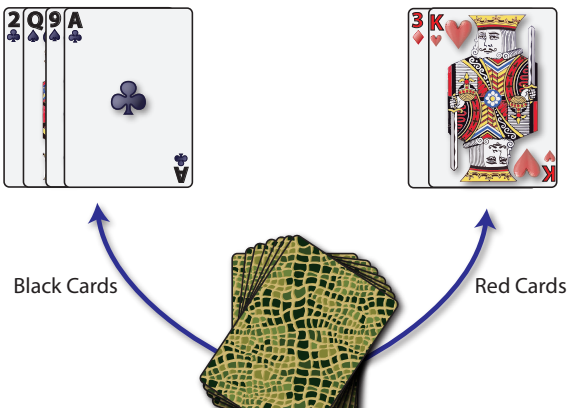
Activity 1.6

Modify the algorithm of Snakes and Ladders, given earlier, to include instructions for landing on the head of a snake.

The IF structure is also used in an extended form to offer a choice between two alternative actions. This expanded form of the IF statement includes another formal term, ELSE, and a second command. If the condition specified in the IF statement is true, then the command following the term THEN is executed, otherwise the command following ELSE is carried out. For instance, lets assume that a card game requires the top card of a face-down deck is to be turned face up and then added to the left or right hand pile as appropriate (see FIG-1.5).

FIG-1.5

Playing Cards



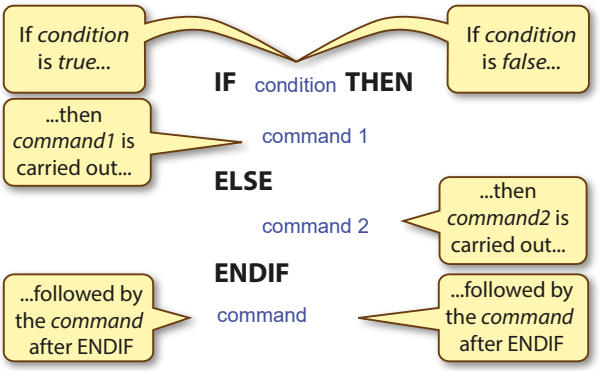
We could write the algorithm for this as:

```
Turn over top card
IF card is black THEN
      Add card to left-hand pile
ELSE
      Add card to right-hand pile
ENDIF
```

The general form of this extended IF statement is shown in FIG-1.6.

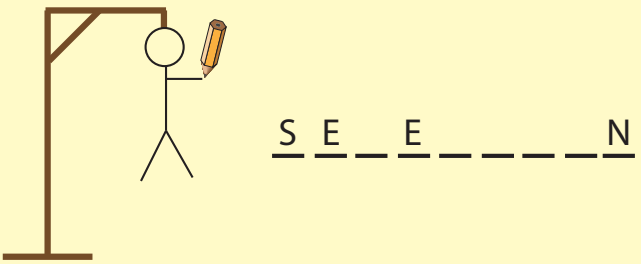
FIG-1.6

The IF..THEN..ELSE Structure



Activity 1.7

In the game of Hangman, *Player 1* has to guess the letters in a word known to *Player 2*. At the start of the game *Player 2* draws one underscore for each letter in the word. When *Player 1* guesses a letter which is in the word, *Player 2* writes the letter above the appropriate underscore. When an incorrect letter is guessed, *Player 2* draws a body part of a hanging man (there are 6 parts in the simple drawing).



Write an IF statement containing an ELSE section which describes the alternative actions to be taken by *Player 2* when *Player 1* guesses a letter.

In the solution include the statements:

```
Add letter at appropriate position(s)
Add part to hanged man
```

Multi-way Selection

Although a simple IF statement can be used to select one of two alternative actions, sometimes we need to choose between more than two alternatives (known as **multi-way selection**). For example, imagine that the rules of the simple guessing game mentioned in Activity 1.5 are changed so that there are three possible responses to *Player 2*'s guess; these being:

- Correct
- Too low
- Too high

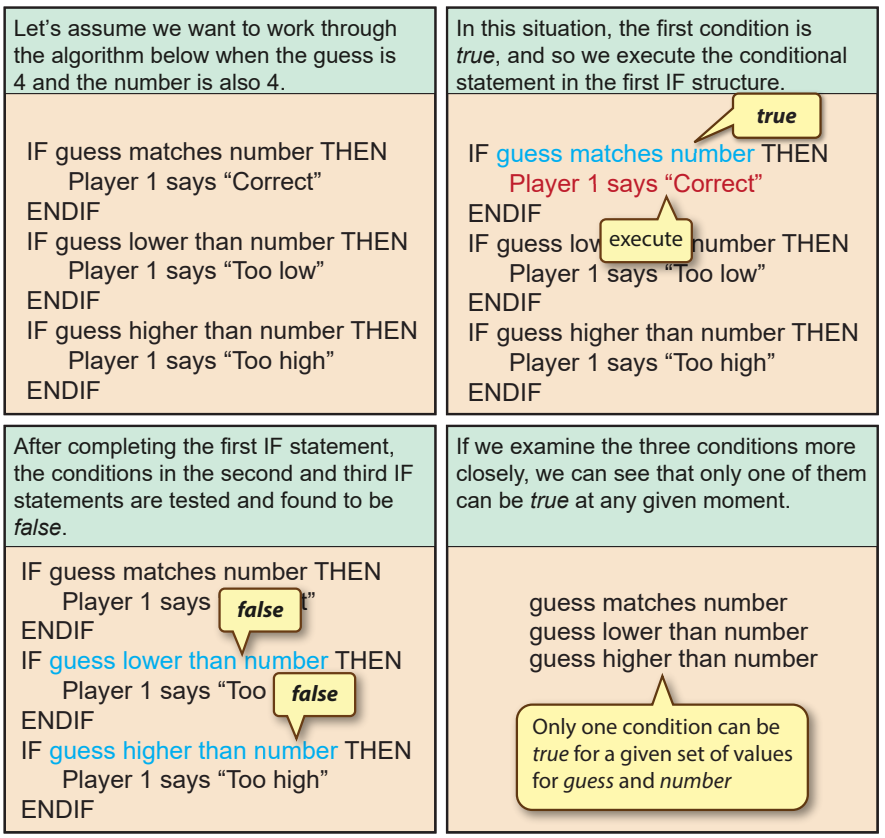
One way to create an algorithm that describes this situation is just to employ three separate IF statements:

```
IF guess matches number THEN
     Player 1 says “Correct”
ENDIF
IF guess is lower than number THEN
     Player 1 says “Too low”
ENDIF
IF guess is higher than number THEN
     Player 1 says “Too high”
ENDIF
```

This will work, but would not be considered a good design for an algorithm since, when the first IF statement is *true*, we still go on and check if the conditions in the second and third IF statements are *true* (see FIG-1.7).

FIG-1.7

Working Through the Algorithm



Checking those last two IF statements is a waste of effort since, if the first condition is true, the others cannot be and therefore testing them serves no purpose.

Where only one of the conditions being considered can be true at a given moment in time, these conditions are known as **mutually exclusive conditions**.

Activity 1.8

Show how the algorithm containing the three IF statements would be dealt with if *guess* was 6 and *number* was 2.

The most effective way to deal with mutually exclusive conditions is to check for one condition, and only if this is false, do we bother to examine the other conditions being tested. So, for example, in Snakes and Ladders, we cannot be at the bottom of a ladder and at the head of a snake at the same time, so we could rewrite our IF statements as

```
IF counter at bottom of ladder THEN
    Move counter to top of ladder
ELSE
    IF counter at head of snake THEN
        Move counter to tail of snake
    ENDIF
ENDIF
```

In the number guessing game, we have three possible outcomes to handle. Taking things slowly, we could start this part of our algorithm with:

```
IF guess matches number THEN
    Player 1 says “Correct”
ELSE
    ***Check the other conditions***
ENDIF
```

Of course a statement like

```
*** Check the other conditions ***
```

is too vague to be much use in an algorithm (hence the asterisks). But what are these other conditions suggested by this statement? They are

```
guess is lower than number
```

and

```
guess is higher than number
```

We already know how to handle a situation where there are only two alternatives: use an IF statement. So selecting between *Too low* and *Too high* requires the statement

```
IF guess is lower than number THEN
    Player 1 says “Too low”
ELSE
    Player 1 says “Too high”
ENDIF
```

Now, by replacing the phrase ***Check the other conditions*** in our original algorithm with our new IF statement we get:

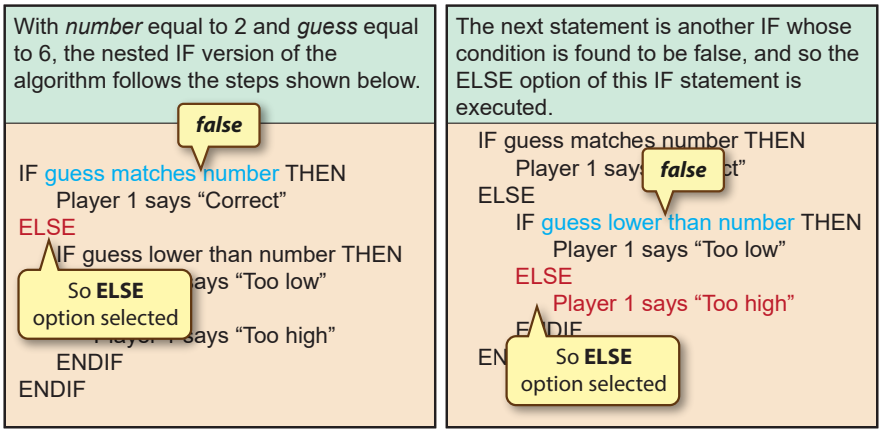
```
IF guess matches number THEN
    Player 1 says “Correct”
ELSE
    IF guess is less than number THEN
        Player 1 says ”Too low”
    ELSE
        Player 1 says “Too high”
    ENDIF
ENDIF
```

Notice that the second IF statement is now totally contained within the ELSE section of the first IF statement. This situation is known as **nested IF statements**.

FIG-1.8 shows how our new nested IF algorithm handles the situation where *number* is 2 and *guess* is 6.

FIG-1.8

Working Through A Nested IF Structure



Activity 1.9

Show what parts of the nested IF algorithm would be performed if *number* was 8 and *guess* 7.

Where there are even more mutually exclusive alternatives, several IF statements may be nested in this way.

Activity 1.10

In a video game, the player controls the movement of a character using the following keys:

Key	Movement
W	Forward
S	Backward
M	Move to the right
N	Move to the left
U	Jump up

The character fires when the space key is pressed.

Only one key can be pressed at a time.

Write a set of nested IF statements which can handle the situation described above.

The solution should contain lines such as

```
IF W key pressed THEN
```

and

```
Move character forward
```

As you can see from the solution to Activity 1.10, although nested IF statements get

the job done, the general structure can be rather difficult to follow. A better method would be to change the format of the IF statement so that several, mutually exclusive conditions can be declared in a single IF statement along with the action required for each of these conditions. This would allow us to rewrite the solution to Activity 1.10 as:

```
IF
    W key pressed:        Move character forward
    S key pressed:        Move character backward
    M key pressed:        Move character to the right
    N key pressed:        Move character to the left
    U key pressed:        Make character jump up
    Space key pressed:    Fire weapon
ENDIF
```

Each option is explicitly named (ending with a colon) and only the one which is *true* will be carried out, the others will be ignored.

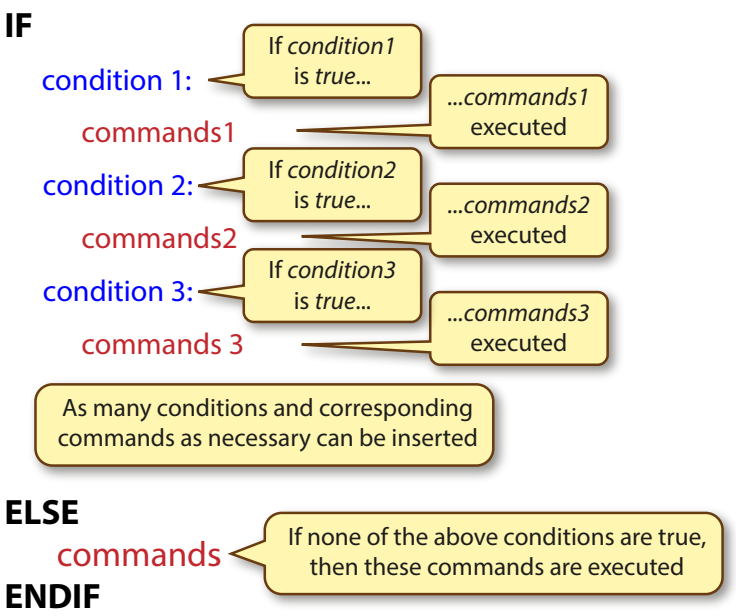
Of course, we are not limited to merely six options; there can be as many as the situation requires. We could add another feature to our character controls by making the game emit a sound when an invalid key (such as X or B) is pressed. To do this we would add an extra ELSE section to our code.

```
IF
    W key pressed:        Move character forward
    S key pressed:        Move character backward
    M key pressed:        Move character to the right
    N key pressed:        Move character to the left
    U key pressed:        Make character jump up
    Space key pressed:    Fire weapon
ELSE
    Play beep noise
ENDIF
```

The additional ELSE option will be chosen only if none of the other options are applicable (that is, if an invalid key is pressed). In other words, it acts like a catch-all, handling all the possibilities not explicitly mentioned in the earlier conditions. This gives us the final form of this style of the IF statement as shown in FIG-1.9.

FIG-1.9

The Multi-Way IF Structure



Activity 1.11

In the TV game Wheel of Fortune (where you have to guess a well-known phrase), you can, on your turn, either guess a consonant, buy a vowel, or make a guess at the whole phrase.

If you think you know the phrase, you should make a guess at what it is; if there are still many unseen letters, you should guess a consonant; as a last resort you can buy a vowel.

Write an IF statement in the style given above describing how to choose from the three options.

Iteration

There are certain circumstances in which it is necessary to perform the same sequence of instructions several times. For example, during a lottery draw, we could describe the basic action as

```
Pick out ball
Call out number on the ball
```

Now, since six balls are drawn, we need to perform these instructions six times. One way to create an algorithm for this task is simply to repeat the statements:

```
Pick out ball
Call out number on the ball
Pick out ball
Call out number on the ball
Pick out ball
Call out number on the ball
Pick out ball
Call out number on the ball
Pick out ball
Call out number on the ball
Pick out ball
Call out number on the ball
```

This would certainly accomplish the task, but it is rather-long winded.

However, not only does it seem rather time-consuming to have to write the same sequence of instructions six times, but it would be even worse if we used the same approach to describe a game of Bingo where many more balls are drawn!

What is required is a way of showing that a section of the instructions is to be repeated a fixed number of times. Carrying out one or more statements over and over again is known as **looping** or **iteration**. The statement or statements that we want to perform over and over again are known as the **loop body**.

Activity 1.12

What statements make up the loop body in the lottery problem given above?

FOR..ENDFOR

When writing a formal algorithm in which we wish to repeat a set of statements a

specific number of times, we use a FOR..ENDFOR structure.

There are two parts to this statement. The first of these is placed just before the loop body and in it we state how often we want the statements in the loop body to be carried out. For the lottery problem our statement would be:

```
FOR 6 times DO
```

Generalising, we can say this statement takes the form

```
FOR value times DO
```

where *value* would be some positive number.

Next come the statements that make up the loop body. These are indented:

```
FOR 6 times DO
    Pick out ball
    Call out number on ball
```

Finally, to mark the fact that we have reached the end of the loop body statements we add the word ENDFOR:

```
FOR 6 times DO
    Pick out ball
    Call out number on ball
ENDFOR
```

The instructions between the terms FOR and ENDFOR are now carried out six times.

Activity 1.13

If we were required to draw out 10 balls rather than 6, what changes would we need to make to the algorithm?

The latest algorithm for our guessing game was:

```
Player 1 thinks of a number between 1 and 100
Player 2 makes an attempt at guessing the number
IF guess matches number THEN
    Player 1 says “Correct”
ELSE
    IF guess is less than number THEN
        Player 1 says “Too low”
    ELSE
        Player 1 says “Too high”
    ENDIF
ENDIF
```

Player 2 would have more chance of winning if he were allowed several chances at guessing *Player 1*’s number. To allow several attempts at guessing the number, some of the statements given above would have to be repeated.

Activity 1.14

What statements in the algorithm above need to be repeated?

To allow for 7 attempts, our new algorithm becomes:

```
Player 1 thinks of a number between 1 and 100
FOR 7 times DO
    Player 2 makes an attempt at guessing the number
    IF guess matches number THEN
        Player 1 says "Correct"
    ELSE
        IF guess is less than number THEN
            Player 1 says "Too low"
        ELSE
            Player 1 says "Too high"
        ENDIF
    ENDIF
ENDFOR
```

Activity 1.15

Can you see a practical problem with the algorithm?

If not, try playing the game a few times, playing exactly according to the instructions in the algorithm.

Occasionally, we may have to use a slightly different version of the FOR loop.

In our *Snakes and Ladders* game we described a player's move with the algorithm

```
Roll die
Move counter forward by the number thrown
IF counter stops at the bottom of a ladder THEN
    Move counter to top of ladder
ENDIF
IF counter stops at the head of a snake THEN
    Move counter to tail of snake
ENDIF
Pass die to next player
```

Although we need to have every player perform these same instructions, we have no way of knowing, when writing the instructions, exactly how many players there will be each time the game is played. To overcome this problem we start our loop with the statement

```
FOR each player DO
```

to give the following algorithm

```
FOR each player DO
    Roll die
    Move counter forward by the number thrown
    IF counter stops at the bottom of a ladder THEN
        Move counter to top of ladder
    ENDIF
    IF counter stops at the head of a snake THEN
        Move counter to tail of snake
    ENDIF
    Pass die to next player
ENDFOR
```

If we had to save the details of a game of chess with the intention of going back to the game later, we might write:

```
FOR each piece on the board DO
    Write down the name and position of the piece
ENDFOR
```

Activity 1.16

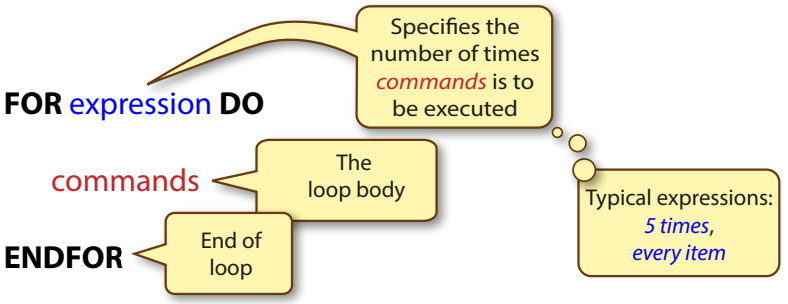
A card game requires only cards showing values between 1 (Ace) and 7, so before beginning the game, all other cards (8 to King) must be removed.

Write an algorithm which places all cards in the range 1 to 7 in a separate pile from those cards showing other values.

The general form of the FOR statement is shown in FIG-1.10.

FIG-1.10

The FOR..ENDFOR Loop



Although the FOR loop allows us to perform a set of statements a specific number of times, this statement is not always suitable for the problem we are trying to solve.

For example, the algorithm created for the guessing game in Activity 1.15 highlighted the problem of having a fixed number of attempts at guessing the value of a number. To solve this problem, we need another way of expressing looping which does not commit us to a specific number of iterations.

REPEAT.. UNTIL

The REPEAT .. UNTIL statement allows us to specify that a set of statements should be repeated until some condition becomes true, at which point iteration should cease.

The word REPEAT is placed at the start of the loop body and, at its end, we add the UNTIL term. The UNTIL term also contains a condition, which, when *true*, causes iteration to stop. This is known as the **terminating** (or exit) **condition**. For example, we could use the REPEAT.. UNTIL structure rather than the FOR loop in our guessing game algorithm. The new version would then be:

```
Player 1 thinks of a number between 1 and 100
REPEAT
    Player 2 makes an attempt at guessing the number
    IF guess matches number THEN
        Player 1 says “Correct”
    ELSE
        IF guess is less than number THEN
            Player 1 says “Too low”
        ELSE
            Player 1 says “Too high”
        ENDIF
    ENDIF
UNTIL Player 2 guesses correctly
```

We could also use the REPEAT..UNTIL loop to describe how a slot machine (one-

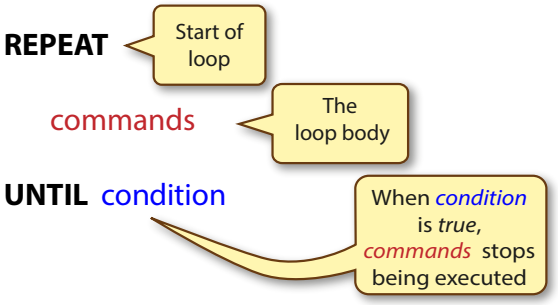
armed bandit) is played:

```
REPEAT
    Put coin in machine
    Pull handle
    IF you win THEN
        Collect winnings
    ENDIF
UNTIL you want to stop
```

The general form of this structure is shown in FIG-1.11.

FIG-1.11

The REPEAT..UNTIL Loop



Activity 1.17

A game requires a player to make use of a shuffled pack of cards lying face-down. The top card is turned over and discarded. This continues until an Ace is turned over.

Using REPEAT..UNTIL, write the logic required for the game.

WHILE.. ENDWHILE

A final method of iteration, differing only subtly from the REPEAT.. UNTIL loop, is the WHILE .. ENDWHILE structure which causes the statements in the loop body to be executed as long as the stated condition is *true*. The condition appears at the start of the structure (beside the term WHILE) and is known as an **entry condition**. The following example illustrates the usefulness of this new structure.

The aim of the card game of Blackjack is to attempt to make the value of your cards add up to 21 without going over that value. Each player is dealt two cards initially but can repeatedly ask for another card by saying "hit". One player is designated the dealer. The dealer must take another card while his cards have a total value of less than 17. So we might attempt to write the rules for the dealer as:

```
Calculate the value of the initial two cards in hand
REPEAT
    Take another card
UNTIL value of cards in hand is greater than or equal to 17
```

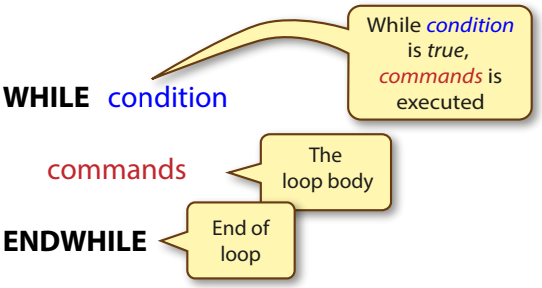
But there's a problem with the solution: if the sum of the first two cards is already 17 or above, we still need to take a third card (just work through the logic, if you can't see why). By using the WHILE..ENDWHILE structure we could rewrite the logic as

```
Calculate the value of the initial two cards in hand
WHILE value of cards in hand is less than 17 DO
    Take another card
ENDWHILE
```

Now determining if the value is less than 17 is performed before the Take another card instruction. If the dealer's two cards already add up to 17 or more, then the *Take another card* instruction will be ignored. The general form of the WHILE.. ENDWHILE statement is shown in FIG-1.12.

FIG-1.12

The WHILE.. ENDWHILE Loop



The differences in operation between the REPEAT..UNTIL and the WHILE.. ENDWHILE structures are shown in FIG-1.13.

FIG-1.13

The Differences Between REPEAT.. UNTIL and WHILE.. ENDWHILE.

REPEAT..UNTIL	WHILE..ENDWHILE
The condition appears after the loop body	The condition appears before the loop body
Looping stops when the condition becomes true	Looping stops when the condition becomes false

The main consequence of these differences is that it is possible to bypass the loop body of a WHILE structure entirely without ever carrying out any of the instructions it contains. On the other hand, the loop body of a REPEAT structure will always be executed at least once.

Activity 1.18

A game involves throwing two dice. If the two values thrown are not the same, then the die showing the lower value must be rolled again. This process is continued until both dice show the same value.

Write a set of instructions to perform this game. Your solution should contain the statements

Roll both dice

and Choose die with lower value

Complex Conditions

We have encountered the use of conditions in the IF, REPEAT..UNTIL, and WHILE.. ENDWHILE structures but so far we have shown only simple conditions in the examples given. More complex conditions can be specified using the same terms we might employ in everyday conversation: AND, OR and NOT.

The AND Operator

In the TV game Family Fortunes, you only win the star prize if you get 200 points and guess the most popular answers to a series of questions. This can be described in our more formal style as:

```
IF at least 200 points gained AND all most popular answers have been guessed
THEN
    winning team get the star prize
ENDIF
```

Note the use of the word AND in the above example. AND (called a **Boolean operator**) is one of the terms used to link simple conditions in order to produce a more complex one (known as a **complex condition**).

The conditions on either side of the AND are the **operands**. Both operands must be true for the overall result to be true. We can generalise this to describe the AND operator as being used in the form:

```
condition 1   AND   condition 2
```

The result of the AND operator is determined using the following rules:

```
Determine the truth of condition 1
Determine the truth of condition 2
IF both conditions are true THEN
    the overall result is true
ELSE
    the overall result is false
ENDIF
```

For example, if a proximity light comes on when it's dark and it detects motion then we can describe the logic of the equipment as:

```
IF it's dark AND motion has been detected THEN
    Switch on light
ENDIF
```

Now, if we assume that at a particular moment in time it's dark but no motion has been detected then condition 1 (*it's dark*) is true but condition 2 (*motion has been detected*) is false. Because one of the conditions is false, the overall result is false and the light does not come on.

You are not limited to just one AND operator in a complex condition; you can have as many as you need. For example, the conditions for foreign national flying to the USA can be written as

```
IF you have a passport AND you have a visa AND you have an airline ticket THEN
    You can fly to the USA
ENDIF
```

All three conditions must be true before you can fly to the USA.

Activity 1.19

A person must meet the following conditions to apply for a job:
age over 21 AND height at least 5 feet 10 inches

Which of the following people can apply for the job:
a) a person who is 18 years old and 6 feet high
b) a person who is 23 years old and 5 feet 9 inches high
c) a person who is 62 years old and 5 feet 10 inches high

When there are two conditions being tested, there are four possible combinations of

results. The first possibility is that both conditions are *false*; another possibility is that condition 1 is *false* but condition 2 is *true*, etc.

Activity 1.20

What are the other possible combinations for the two conditions?

All possibilities of the AND operator are summarised in FIG-1.14.

FIG-1.14

The **AND** Truthtable

Note that the result is *true* only when both conditions are *true*.

condition 1	condition 2	condition 1 AND condition 2
false	false	false
false	true	false
true	false	false
true	true	true

Activity 1.21

In Microsoft Windows applications, the program will request the name of the file to be opened if the **Ctrl** and **O** keys are pressed together.

Write the first line of an IF statement, which includes the term AND, summarising this situation.

The OR Operator

Simple conditions may also be linked by the Boolean OR operator. Using OR, only one of the two conditions specified needs to be true in order to carry out the action that follows. For example, in the game of *Monopoly* you go to jail if you land on the *Go To Jail* square or if you throw three doubles in a row. This can be written as:

```
IF player landed on Go To Jail OR player has thrown 3 pairs in a row THEN
    Move player to jail
ENDIF
```

Like AND, the OR operator works on two operands:

```
condition 1   OR   condition 2
```

Hence the results are determined by the following rules:

```
Determine the truth of condition 1
Determine the truth of condition 2
IF any of the conditions are true THEN
    the overall result is true
ELSE
    the overall result is false
ENDIF
```

For example, if a player in the game of *Monopoly* has not landed on the *Go To Jail* square, but has thrown three consecutive pairs, then the result of the IF statement given above would be:

Condition 1 (*has landed on Go to Jail*) is false
Condition 2 (*has thrown three consecutive doubles*) is true

Because at least one of the conditions is true, the overall result is true, so the player

player moves to Jail.

Using the OR operator, the possible combinations and results are summarised in FIG-1.15.

FIG-1.15

The **OR** Truthtable

condition 1	condition 2	condition 1 OR condition 2
false	false	false
false	true	true
true	false	true
true	true	true

As with AND, you can have as many OR statements as you need in a complex condition. As long as at least one of the conditions given is true, the overall result will be true.

Activity 1.22

In the game of Monopoly, a player can get out of jail if they throw a double (same value on both dice), pay a fine, or hand over a "Get Out of Jail Free" card.

Write an IF statement that reflects this logic.

The NOT Operator

The final Boolean operator which can be used as part of a condition is NOT. This operator is used to reverse the meaning of a condition. In standard English, the opposite of *it's dark* is *it's not dark*; in the structured English we always place the word NOT first. This means that, rather than write *it's not dark,* we would write

```
NOT it's dark
```

In *Monopoly* a player can charge rent on a property as long as that property is not mortgaged. This situation can be described with the statement:

```
IF NOT property mortgaged THEN
    Rent can be charged
ENDIF
```

The NOT operator works on a single operand:

```
NOT condition
```

When NOT is used, the result given by the original condition is reversed. Hence the results are determined by the following rules:

1. Determine the truth of the original condition without the term NOT
2. Complement the result obtained in step 1

For example, if a player lands on a property that is not mortgaged, then the result of the IF statement given above would be calculated as:

Original condition (property mortgaged)	is	false
So, since the complement of *false* is *true*, the result	is	true

This may seem a rather strange way to work out the overall result, but it will prove

to be a useful approach when we examine exactly how C++ operates in a later chapter.

For many conditions you can eliminate the need for NOT by simply changing the condition. So, rather than write NOT it's dark we could write it's light (assuming light or dark are the only two options). But there are situations where using NOT will save a lot of writing. For example, it's easier to write

```
IF NOT it's Monday THEN
```

than

```
IF it's Sunday OR it's Tuesday OR it's Wednesday OR it's Thursday OR it's
↳Friday OR it's Saturday THEN
```

Although both IF statements are equivalent to each other, the first involves a lot less typing!

The results of the NOT operator are summarised in FIG-1.16.

FIG-1.16

The NOT Truthtable

condition	NOT condition
false	true
true	false

Mixing Boolean Operators

Conditions can get as complicated as your head can cope with. You can mix ANDs, ORs and NOTs to your heart's content in order to express the complex condition your logic requires. When you have such a complex condition, the overall result is calculated as follows:

1. Determine the truth of each condition
2. Perform all NOT operations
3. Perform all AND operations
4. Perform all OR operations

There will be some conditions where this order will not give the result you are after. For example, let's assume that to win a game you must first accumulate $100,000 and then either own 25 properties or eliminate all other players. We could write this as:

```
IF player has $100,000 AND player has 25 properties OR all other players
↳eliminated THEN
```

Now, let's assume the following conditions: a player has $80,000, 18 properties and has eliminated all other players. Should this player win the game? No, because he must have at least $100,000. But if we calculate the result according to the rules above, then:

determining the truth of each condition we get:

```
IF false AND false OR true THEN
```

Since, there are no NOT operators, we perform the AND operation giving:

```
IF false OR true THEN
```

And, finally, performing the OR operation leaves us with:

```
IF true THEN
```

So, according to this, the player has won.

In situations like this, where we need to have the operations performed in a different order, we may use parentheses. Any operations within parentheses are always performed first. So, if we rewrite our IF statement as:

```
IF player has $100,000 AND (player has 25 properties OR all other players
↳eliminated) THEN
```

the steps become:

```
    IF false AND (false OR true) THEN
⇒   IF false AND true THEN
⇒   IF false THEN
```

and the player is shown to have not won.

Boolean operator priority is summarised in FIG-1.17.

FIG-1.17

Operator Priority

Priority	Operator
1	( )
2	NOT
3	AND
4	OR

Activity 1.23

A simple card game involves a player turning over cards from the top of a face-down deck until an Ace or a King is uncovered. At this point the game stops.

Write a WHILE statement that describes the logic involved.

Activity 1.24

a) What is another term for *condition*?
b) Give an example of a Boolean operator.
c) If two conditions are linked using the term AND, how many of the conditions must be true before the conditional statement is executed?
d) If a complex condition contains both a NOT and an AND operator, which is performed first?
e) How can we modify a Boolean expression so that an OR operator is performed before an AND operator?

Data

We know we need to retain information. Look at your phone; it is probably packed with names, email addresses, phone numbers, and much more. Even when playing an old-fashioned board game we need to remember things such as the number you threw on the die, where your piece is on the board and so on.

All these examples introduce the need to process facts and figures (known as **data**).

Every item of data has two basic characteristics :

a name
and a value

The name of a data item is a description of the information it represents. Hence, on a form we might see boxes labelled as *Forename*, *Surname*, *Address*, *Phone No*, etc. These are the data names. And when we've completed the form, the boxes contain the values we have written in. These are the data values.

In programming, a data item is often referred to as a **variable**. This term arises from the fact that, although the name assigned to a data item cannot change, its value may vary. For example, the value assigned to a variable called *salary* may rise (or fall) over weeks, months or years.

Types of Data

Most computer programming languages need to be told what type of value is to be held in a variable - for example, it needs to know if a variable will hold a number or a message. Once the variable is set up for one type of value, it can't be used to hold any other type. Four of the basic data types recognised by C++ are:

int	(short for integer) holds whole numbers only (e.g. -12, 0, 92).
float	holds numbers containing fractions (-14.6, 0.005, 176.0). Notice that the fraction part may be .0.
char	(short for character) holds zero, one or more characters.
bool	(short for boolean) holds either *true* or *false*.

Other data types are possible, but we'll look at these in a later chapter.

Operations on Data

There are four basic operations that an algorithm or computer program can do with data. These are:

Input

This involves being given a value for a data item. For example, in our number-guessing game, the player who has thought of the original number is given the value of the guess from the second player. When using a computer, any value entered at the keyboard, or any movement or action dictated by a mouse or joystick would be considered as data entry. This type of action is known as an **input operation**.

Calculation

Most games involve some basic arithmetic. In Monopoly, the banker has to work out how much change to give a player buying a property. If a character in an adventure game is hit, points must be deducted from his strength value. This type of instruction is referred to as a **calculation operation**. When describing a calculation, it is common to use arithmetic operator symbols rather than English. Hence, instead of writing the word *subtract* we use the minus sign (-). A summary of the operators available are

given in FIG-1.18.

FIG-1.18

The Arithmetic Operators

English	Symbol
Multiply	*
Divide	/
Add	+
Subtract	-

Note the standard computing practice of using * as the multiplication operator and / for division

Comparison

Often values have to be compared. For example, we need to compare the two numbers in our guessing game to find out if they are the same. This is known as a **comparison operation**. Rather than use terms such as *is less than*, we use the *less than* symbol (<). A summary of these comparison operators is given in FIG-1.19.

FIG-1.19

The Comparison Operators

English	Symbol
is less than	<
is less than or equal to	<=
is greater than	>
is greater than or equal to	>=
is equal to	=
is not equal to	<>

Note the symbols <> are used to mean *not equal to*. Some languages use !=

Output

The final requirement is to communicate with others to give the result of some calculation or comparison. For example, in the guessing game, player 1 communicates with player 2 by saying either that the guess is *Correct*, *Too high* or *Too low*.

In a computer environment, the equivalent operation would normally involve displaying information on a screen or printing it on paper. For instance, in a racing game your speed and time will be displayed on the screen. This is called an **output operation**.

Activity 1.25

a) What are the two main characteristics of any data item?
b) When data is input, from where is its value obtained?
c) Give an example of a relational operator.

Counts and Totals

Perhaps two of the commonest requirements in programming are keeping counts and keeping totals. Of course, these are operations that we perform often ourselves. If you want to deal 13 cards from a deck, you'll keep a mental count of how many you've dealt so far. If you have several items to pay for in a shop, you'll work out the total price (or maybe you'll just believe what the till tells you!).

To perform count and total operations in a detailed algorithm we need to make use of variables.

Counting

In our guessing game, we might want to know how many guesses it takes to come up with the correct value. We can do this by modifying our previous algorithm as follows:

```
Player 1 thinks of a number between 1 and 100
Set count to zero
REPEAT
    Player 2 makes an attempt at guessing the number
    Add 1 to count
    IF guess matches number THEN
        Player 1 says "Correct"
    ELSE
        IF guess is less than number THEN
            Player 1 says "Too low"
        ELSE
            Player 1 says "Too high"
        ENDIF
    ENDIF
UNTIL player 2 guesses correctly
Player 1 states the value of count
```

The three new statements are typical of any algorithm that performs a count:

- Initialise the count to zero.
- Add 1 to the count each time the operation being counted occurs.
- State the value of the count when counting has ended.

Totalling

In Activity 1.23 we described the logic of a simple card game in which cards were turned over until an Ace or a King was encountered. Let's extend this by counting the total value of the cards turned but excluding the Ace or King which terminates the process. We can perform this using the following changes to the earlier logic:

```
Set total to zero
Turn over card
WHILE NOT card is Ace AND NOT card is KING DO
    Add card value to total
    Turn over card
ENDWHILE
State value of total
```

As we can see, there is very little difference between the logic of totalling and that of counting. In fact, the only difference is the value being added to the variable.

Activity 1.26

Modify the card-totalling algorithm given above so that not only the total but also the number of cards turned (excluding the final card) is calculated and stated.

Activity 1.27

Load and run the app *CardAlgorithm.exe* which is in *CPP/Resources/Ch01/CardAlgorithm* of the support files you downloaded earlier.
Note how the statements are executed and the variables change value.

Levels of Detail

When we start to write an algorithm in English, one of the things we need to consider is exactly how much detail should be included. For example, we might describe how to change a flat tyre on a car as:

Prepare car
Remove wheel which has flat tyre
Fit new wheel
Ensure new tyre is at correct pressure
Store any equipment used

However, this lacks enough detail for anyone unfamiliar with the operation. To help, we could replace the first statement Prepare car with:

Park safely on level ground
Switch off the engine
Put the car in appropriate gear
Pull on the handbrake

More detail could be added to the other original statements in the same way.

This approach of starting with a less detailed sequence of instructions and then, where necessary, replacing each of these with more detailed instructions can be used to good effect when tackling long and complex problems. By using this technique, we are defining the solution to the original problem as an equivalent sequence of tasks before going on to create a set of more detailed instructions on how to handle each of these tasks. This divide-and-conquer strategy is known as **stepwise refinement**.

Now that we've covered the idea behind stepwise refinement, let's have a look at the complete solution to creating an algorithm for changing a flat tyre:

Outline Solution:

1. Prepare car
2. Remove wheel which has flat tyre
3. Fit new wheel
4. Ensure new tyre is at the correct pressure
5. Store any equipment used

This is termed a **LEVEL 1 solution**.

As a guideline, we should aim for a LEVEL 1 solution with between 4 and 12 instructions. Notice that each instruction has been numbered. This is merely to help with identification during the stepwise refinement process. Before going any further, we must assure ourselves that this is a correct and full (though not detailed) description of all the steps required to tackle the original problem. If we are not happy with the solution, then changes must be made before going any further. Next, we examine each statement in turn and determine if it should be described in more detail. Where this is necessary, rewrite the statement to be dealt with, and below it, give the more detailed version. For example, *Prepare car* would be expanded thus:

1. Prepare car
    1.1 Park safely on level ground
    1.2 Switch off the engine
    1.3 Put the car in appropriate gear
    1.4 Pull on handbrake

The numbering of the new statement reflects that they are the detailed instructions pertaining to statement 1. Also note that the number system is not a decimal fraction, so if there were to be many more statements they would be numbered 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, etc.

It is important that these sets of more detailed instructions describe how to perform only the original step being examined - they must achieve no more and no less. Sometimes the detailed instructions will contain control structures such as IFs, WHILEs or FORs. Where this is the case, the whole of that control structure must be included in the detailed instructions for that task. Having satisfied ourselves that the more detailed breakdown is correct, we proceed to the next statement from the original solution.

```
2. Remove wheel which has a flat tyre
    2.1 Remove any hub cap
    2.2 Loosen wheel nuts
    2.3 Jack up the car
    2.4 Remove wheel
```

To fit the new wheel, the extra detail is:

```
3. Fit new wheel
    3.1 Place new wheel on car
    3.2 Replace wheel nuts
    3.3 Lower car
    3.4 Tighten wheel nuts
```

Control structures can be added where necessary. In the next breakdown we use WHILE and IF in the more detailed description:

```
4. Ensure new tyre is at correct pressure
    4.1 Check tyre pressure
    4.2 WHILE pressure not correct DO
    4.3       IF pressure too low THEN
    4.4             Pump some air into the tyre
    4.5       ELSE
    4.6             IF pressure too high THEN
    4.7                   Release some air from the tyre
    4.8             ENDIF
    4.9       ENDIF
    4.10      Check tyre pressure
    4.11 ENDWHILE
```

But not every statement from a level 1 solution needs to be expanded. For example, we may decide that Store any equipment used is sufficient detail for step 5, therefore no further breakdown is required.

Finally, we can describe the solution to the original problem in more detail by substituting the statements in our LEVEL 1 solution by their more detailed equivalent:

```
1.1 Park safely on level ground
1.2 Switch off the engine
1.3 Put the car in appropriate gear
1.4 Pull on handbrake
2.1 Remove any hub cap
2.2 Loosen wheel nuts
2.3 Jack up the car
2.4 Remove wheel
3.1 Place new wheel on car
3.2 Replace wheel nuts
3.3 Lower car
3.4 Tighten wheel nuts
4.1 Check tyre pressure
```

```
4.2 WHILE pressure not correct DO
4.3       IF pressure too low THEN
4.4             Pump some air into the tyre
4.5       ELSE
4.6             IF pressure too high THEN
4.7                   Release some air from the tire
4.8             ENDIF
4.9       ENDIF
4.10      Check tyre pressure
4.11 ENDWHILE
5.    Store any equipment used
```

This is a LEVEL 2 solution. Note that a level 2 solution includes any LEVEL 1 statements which were not given more detail (in this case, Store any equipment used).

For some more complex problems it may be necessary to repeat this process to more levels before sufficient detail is achieved. For example, we might break down the statement 1.1 Park safely on level ground to

```
1.1.1 Stop car out of the way of traffic on level ground
1.1.2 IF you are near passing traffic THEN
1.1.3    Turn on your hazard lights
1.1.4 ENDIF
```

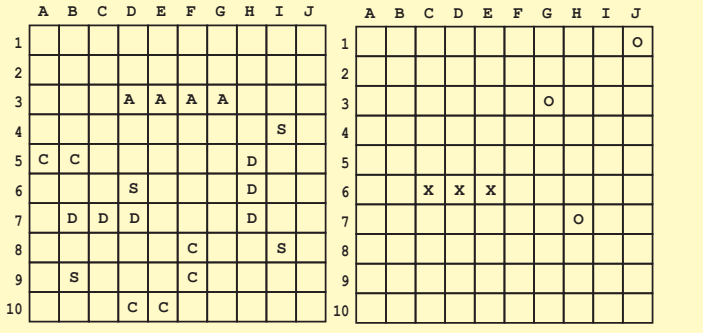
Here a level 2 statement has been broken down into level 3 statements. To create a complete level 3 algorithm, this process would be continued for any other level 2 statements that needed to be expanded to give more detail. When complete, the appropriate statements are collected together, just as we did to create the level 2 description, to produce a level 3 breakdown.

Activity 1.28

The game of battleships involves two players. Each player draws two 10 by 10 grids. Each of these have columns lettered A to J and rows numbered 1 to 10. In the first grid each player marks the position of warships. Ships are added as follows:

1 aircraft carrier	4 squares
2 destroyers	3 squares each
3 cruisers	2 squares each
4 submarines	1 square each

The squares of each ship must be adjacent and must be vertical or horizontal. The first player now calls out a grid reference. The second player responds to the call by saying HIT or MISS. HIT is called if the grid reference corresponds to a position of a ship. The first player then marks this result on his second grid using an O to signify a miss and X for a hit (see diagram below).



Vessels are positioned in the left-hand grid

Results of guesses are placed in the right-hand grid

Activity 1.26 (continued)

If the first player achieves a HIT then he continues to call grid references until MISS is called. In response to a HIT or MISS call the first player marks the second grid at the reference called: O for a MISS, X for a HIT. When the second player responds with MISS the first player's turn is over, and the second player has his turn.

The first player to eliminate all segments of the opponent's ships is the winner. However, each player must have an equal number of turns, and if both sets of ships are eliminated in the same round the game is a draw.

The algorithm describing the task of one player is given in the instructions below. Create a LEVEL 1 algorithm by assembling the lines in the correct order, adding line numbers to the finished description.

```
Add ships to left grid
UNTIL there is a winner
Call grid position(s)
REPEAT
Respond to other player's call(s)
Draw grids
```

To create a LEVEL 2 algorithm, some of the above lines will have to be expanded to give more detail. More detailed instructions are given below for the statements Call grid position(s) and Respond to other player's call(s). By reordering and numbering the lines below create LEVEL 2 details for these two statements.

```
UNTIL other player misses
Mark position in second grid with X
Get other player's call
Get reply
Get reply
ENDIF
Call HIT
Call MISS
Mark position in second grid with O
WHILE reply is HIT DO
Call grid reference
Call grid reference
IF other player's call matches position of ship THEN
ENDWHILE
REPEAT
ELSE
```

Checking for Errors

In this section we have a look at how a paper-based algorithm can be checked for errors. Although now outdated – most programmers get the computer to do this type of work – it shows the type of strategy employed by a compiler's debugger when checking for errors in a program's code.

Once we've created our algorithm we would like to make sure it is correct. Unfortunately, there is no foolproof way to do this! But we can at least try to find any obvious errors or omissions in the set of instructions we have created. This type of error is known as a **logic error**. We do this by going back to the original description of the task our algorithm is attempting to solve and work through the algorithm using imagined values. For example, let's assume we want to check our number guessing game algorithm we created earlier. In the last version of the game we allowed the

second player to make as many guesses as required until he came up with the correct answer. The first player responded to each guess by saying either "*Too low*", "*Too high*" or "*Correct*".

To check our algorithm for errors we must come up with typical values that might be used when carrying out the set of instructions. This set of values should be chosen so that each possible result is achieved at least once. For our game, we have three results possible each time a guess is made. These are "*Too low*", "*Too high*" or "*Correct*". As well as making up values, we need to predict what response our algorithm should give to each value used. Hence, if the first player thinks of the value 42 and the second player guesses 75, then the first player will respond to the guess by saying "*Too high*". Our set of test values must evoke each of the possible results from our algorithm. One possible set of values and the responses for our game are shown in FIG-1.20.

FIG-1.20

Test Data for the Number Guessing Game Algorithm

Test Data	Expected Results
number = 42 guess = 75 guess = 15 guess = 42	 Says "Too high" Says "Too low" Says "Correct"

Once we've created test data, we need to work our way through the algorithm using that test data and checking that we get the expected results. This is known as a **dry run** or **desk checking**.

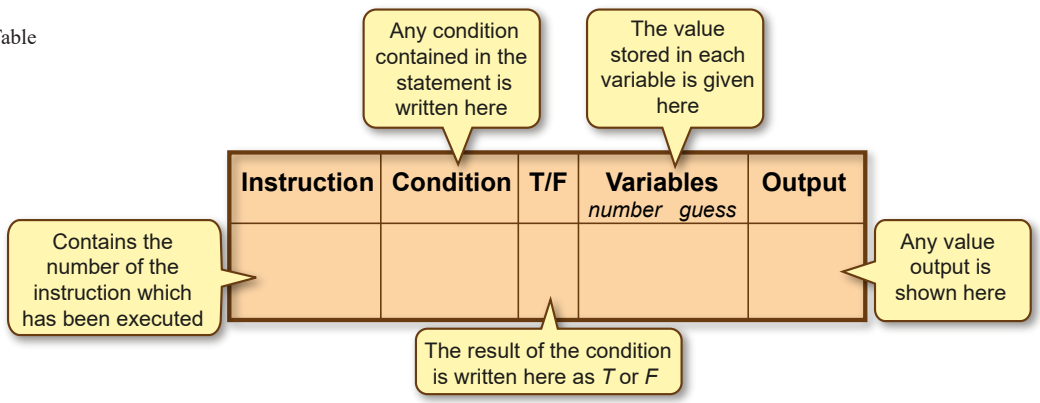
The algorithm for the number game is shown below, this time with instruction numbers added.

```
1. Player 1 thinks of a number between 1 and 100
2. REPEAT
3.    Player 2 makes an attempt at guessing the number
4.    IF guess = number THEN
5.          Player 1 says "Correct"
6.    ELSE
7.          IF guess < number THEN
8.                Player 1 says "Too low"
9.          ELSE
10.               Player 1 says "Too high"
11.         ENDIF
14.   ENDIF
14. UNTIL guess = number
```

Next we create a table (called a **trace table**) with the headings as shown in FIG-1.21.

FIG-1.21

A Trace Table



Now we work our way through the statements in the algorithm filling in a line of the trace table for each instruction.

Instruction 1 is for player 1 to think of a number. Using our test data, that number will be 42, so our trace table starts with the line shown in FIG-1.22.

FIG-1.22

Working through a Trace 1

Instruction	Condition	T/F	Variables *number* *guess*	Output
1			42	

The REPEAT word comes next. Although this does not cause any changes, nevertheless a 2 should be entered in the next line of our trace table. Instruction 3 involves player 2 making a guess at the number (this guess will be 75 according to our test data). After 3 instructions our trace table is as shown in FIG-1.23.

FIG-1.23

Working through a Trace 2

Instruction	Condition	T/F	Variables *number* *guess*	Output
1			42	
2				
3			75	

Instruction 4 is an IF statement containing a condition. This condition and its result are written into columns 2 and 3 as shown in FIG-1.24.

FIG-1.24

Working through a Trace 3

Instruction	Condition	T/F	Variables *number* *guess*	Output
1			42	
2				
3			75	
4	guess = number	F		

Because the condition is false, we now jump to instruction 6 (the ELSE line) and on to 7. This is another IF statement and our table now becomes that shown in FIG-1.25.

FIG-1.25

Working through a Trace 4

Instruction	Condition	T/F	Variables *number* *guess*	Output
1			42	
2				
3			75	
4	guess = number	F		
6				
7	guess < number	F		

Since this second IF statement is also false, we move on to statements 9 and 10. Instruction 10 causes output (speech) and hence we enter this in the final column as shown in FIG-1.26.

FIG-1.26

Working through a Trace 5

Instruction	Condition	T/F	Variables *number*	Variables *guess*	Output
1			42		
2					
3				75	
4	guess = number	F			
6					
7	guess < number	F			
9					
10					Too high

Now we move on to statements 11,12 and 13 as shown in FIG-1.27.

FIG-1.27

Working through a Trace 6

Instruction	Condition	T/F	Variables *number*	Variables *guess*	Output
1			42		
2					
3				75	
4	guess = number	F			
6					
7	guess < number	F			
9					
10					Too high
11					
12					
13	guess = number	F			

Since statement 13 contains a condition which is false, we return to statement 2 and then onto 3 where we enter 15 as our second guess (see FIG-1.28).

FIG-1.28

Working through a Trace 7

Instruction	Condition	T/F	Variables *number*	Variables *guess*	Output
1			42		
2					
3				75	
4	guess = number	F			
6					
7	guess < number	F			
9					
10					Too high
11					
12					
13	guess = number	F			
2					
3				15	

Activity 1.29

Create your own trace table for the number-guessing game and, using the same test data as given in FIG-1.20 complete the testing of the algorithm.

Were the expected results obtained?

Activity 1.30

Load and run *Remember01.exe* (part of the downloaded *Ch01* support files).

Use it to remind you of the points covered in this chapter.

Summary

- Computers can perform many tasks by executing different programs.
- An algorithm is a sequence of instructions which solves a specific problem.
- A program is a sequence of computer instructions which usually manipulates data and produces results.
- Three control structures are used in programs :

```
Sequence
Selection
Iteration
```

- A sequence is a list of instructions which are performed one after the other.
- Selection involves choosing between two or more alternative actions.
- Selection is performed using the IF statement.
- There are three forms of IF statement:

```
IF condition THEN
    instructions
ENDIF

IF condition THEN
    instructions
ELSE
    instructions
ENDIF

IF
    condition 1:
        instructions
    condition 2:
        instructions
    condition x :
        instructions
    ELSE
        instructions
ENDIF
```

- Iteration is the repeated execution of one or more statements.
- Iteration is performed using one of three instructions:

```
FOR number of iterations required DO
    instructions
ENDFOR

REPEAT
    instructions
UNTIL condition

WHILE condition DO
    instructions
ENDWHILE
```

- A condition is an expression which is either ***true*** or ***false***.
- Simple conditions can be linked using AND or OR to produce a complex condition.
- The meaning of a condition can be reversed by adding the word NOT.

- Data items (or variables) hold the information used by the algorithm.
- Data item values may be:

  Input
  Calculated
  Compared
  or Output

- Calculations can be performed using the following arithmetic operators:

Multiplication	*
Addition	+
Division	/
Subtraction	-

- Comparisons can be performed using the relational operators:

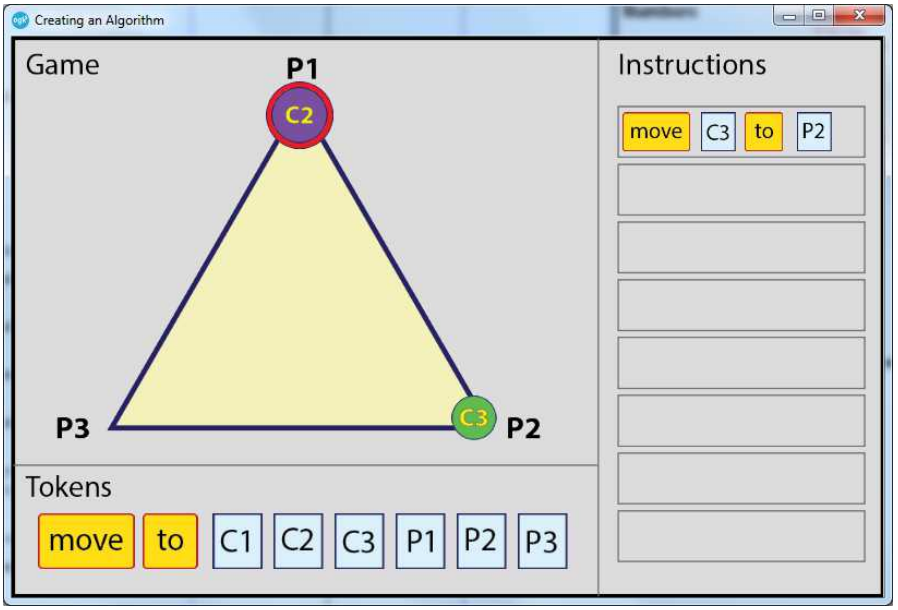
Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
Equal to	=
Not equal to	<>

- In programming, a data item is referred to as a variable.
- Counting involves initialising a variable to zero and incrementing it each time the event being counted occurs.
- Totalling involves initialising a variable to zero and adding a specified value to the total each time a new value is received.
- The divide-and-conquer strategy of stepwise refinement can be used when creating an algorithm.
- LEVEL 1 solution gives an overview of the sub-tasks involved in carrying out the required operation.
- LEVEL 2 gives a more detailed solution by taking each sub-task from LEVEL 1 and, where necessary, giving a more detailed list of instructions required to perform that sub-task.
- Not every statement needs to be broken down into more detail.
- Further levels of detail may be necessary when using stepwise refinement for complex problems.
- An algorithm can be checked for errors or omissions using a trace table.

Support Material for this Chapter

Algorithm Constructor (*TriLogic.exe*)

Screen Shot



Overview

This program allows you to construct and algorithm for moving the three counters (C1, C2 and C3) from position P1 to position P3. The algorithm must obey the rule that a larger counter cannot be placed on a smaller one.

User Instructions

Press the Start button (visible in the top right of the Game area when the program first starts). This moves all three counters to P1 on the playing area .

Click on the appropriate tokens to construct an instruction. All instructions should be of the form

```
MOVE counter TO position
```

and are assembled in the Instructions area in response to the tokens selected.

If the instruction is invalid an error message will appear. Click on the error message to remove it and redo your instruction.

When a correct command is entered, the selected counter will be moved in the Game area.

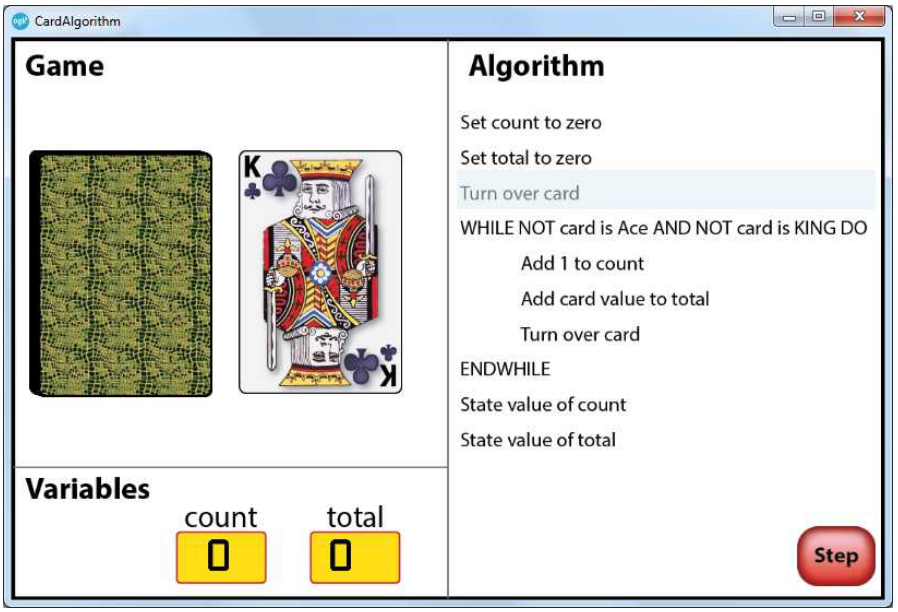
Once a move is complete, continue building instructions until all the counters are at position P3.

Download

The app file is called *TriLogic.exe* and can be found in the *CPP/Resources/Ch01/TriLogic* folder of the download material for this book.

Algorithm Tracer (*CardAlgorithm.exe*)

Screen Shot



Overview

This program shows the step-by-step execution of an algorithm, highlighting the line which has just been executed.

The algorithm (shown in the Algorithm area of the screen) counts the number of playing cards turned over before an Ace or a King is encountered. The card turning is shown in the Game area. The total face value of those cards is also calculated. These two values are displayed in the Variables part of the screen.

The cards are shuffled each time the program is run, so the results will be different every time.

User Instructions

When the program begins, no statement is highlighted.

Press the Step button. This will execute the highlight and execute the first command.

The effect created by executing a command will result in a change in the Variables area, a card being turned in the Game area, a condition being tested, or simply the moving of the highlighter to the next line.

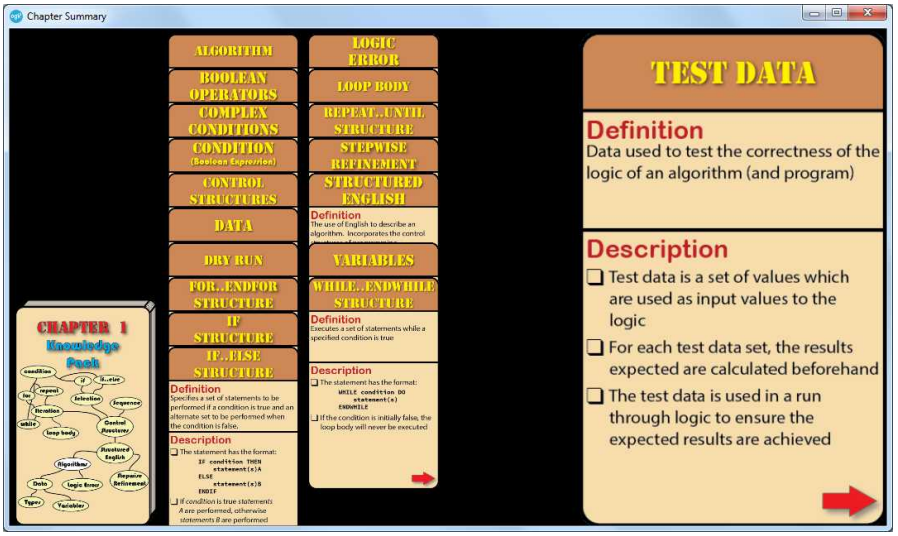
Continue to press the Step button until all the statements in the algorithm have been carried out.

Download

The app file is called *CardAlgorithm.exe* and can be found in the *CPP/Resources/Ch01/CardAlgorithm* folder of the download material for this book.

Reviewer (*Remember01.exe*)

Screen Shot



Overview

This program displays a set of cards summarising various topics covered in this chapter.

User Instructions

When the program starts up a single card is displayed showing the main topics covered in the form of a mindmap. This is the front card of the pack.

Click on the front card. This will move the card pack to the bottom left of the screen and deal all the cards into the centre of the screen.

Click on a card title to have it expand on the right hand side of the screen.

Click on a card's arrow to see the other side of a card.

Click on a card's title to have it return to its position in the dealt pack.

Click on the front card (positioned at the bottom left) to return all the cards to the pack.

Download

The app file is called *Remember01.exe* and can be found in the *CPP/Resources/Ch01/Remember01* folder of the download material for this book.

Solutions

Activity 1.1

No solution required.

Activity 1.2

A second solution is:

```
Fill A                A = 3          B = 0
Pour A into B         A = 0          B = 3
Fill A                A = 3          B = 3
Fill B from A         A = 1          B = 5
Empty B               A = 1          B = 0
Pour A into B         A = 0          B = 1
Fill A                A = 3          B = 1
Pour A into B         A = 0          B = 4
```

Activity 1.3

A possible solution to the river crossing problem:

```
Row to other side with goat
Return to first side
Row over with wolf
Return to first side with goat
Row over with cabbage
Return to first side
Row over with goat
```

Activity 1.4

A possible solution to the TriLogic game is:

```
Move C3 to P3
Move C2 to P2
Move C3 to P2
Move C1 to P3
Move C3 to P1
Move C2 to P3
Move C3 to P3
```

Activity 1.5

```
Player 1 thinks of a number
Player 2 makes a guess at the number
IF guess matches number THEN
   Player 1 says "Correct"
ENDIF
Player 1 states the value of the number
```

Activity 1.6

Updated algorithm for Snakes and Ladders:

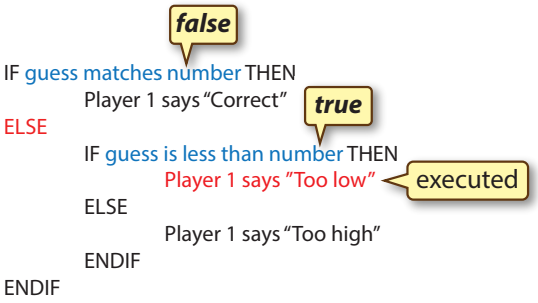
```
Roll die
Move counter forward by the number thrown
IF counter stops at the bottom of a ladder THEN
   Move counter to top of ladder
ENDIF
IF counter stops at the head of a snake THEN
   Move counter to tail of snake
ENDIF
Pass die to next player
```

Activity 1.7

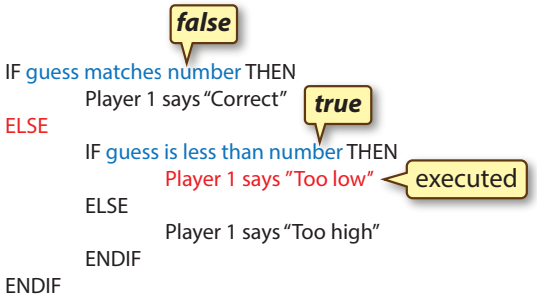
Algorithm for Player 2 response in Hangman:

```
IF letter is in word THEN
   Add letter at appropriate position(s)
ELSE
   Add part to hanged man
ENDIF
```

Activity 1.8



Activity 1.9



Activity 1.10

```
IF W key pressed THEN
  Move character forward
ELSE
  IF S key pressed THEN
    Move character backward
  ELSE
    IF M key pressed THEN
      Move character to the right
    ELSE
      IF N key pressed THEN
        Move character to the left
      ELSE
        IF U key pressed THEN
          Make character jump up
        ELSE
          IF space key pressed THEN
            Fire weapon
          ENDIF
        ENDIF
      ENDIF
    ENDIF
  ENDIF
```

Activity 1.11

```
IF
  you know the phrase:
      Make guess at phrase
  there are many unseen letters:
      Guess a consonant
ELSE
    Buy a vowel
ENDIF
```

Activity 1.12

The two statements which make up the loop body are:

```
Pick out ball
Call out number on the ball
```

Activity 1.13

Only one line, the FOR statement, would need to be changed, the new version being:

```
FOR 10 times DO
```

Activity 1.14

In fact, only the first line of our algorithm is not repeated, so the lines that need to be repeated are:

```
Player 2 makes an attempt at guessing the number
IF guess matches number THEN
     Player 1 says "Correct"
 ELSE
   IF guess is less than number THEN
     Player 1 says "Too low"
   ELSE
     Player 1 says "Too high"
   ENDIF
ENDIF
```

Activity 1.15

The FOR loop forces the loop body to be executed exactly 7 times. If the player guesses the number in less attempts, the algorithm will nevertheless continue to ask for the remainder of the 7 guesses.

Later, we'll see how to solve this problem.

Activity 1.16

```
FOR 52 times DO
  Look at card at top of deck
  IF its value is between 1 and 7 THEN
    Place in left-hand pile
  ELSE
    Place in right-hand pile
  ENDIF
ENDFOR
```

We could have started with the line

```
For each card DO
```

Activity 1.17

```
REPEAT
  Turn over top card
UNTIL card is an Ace
```

Activity 1.18

```
Roll both dice
WHILE dice values don't match DO
  Choose die with lower value
  Throw chosen die
ENDWHILE
```

Activity 1.19

a) Cannot apply. Too young; first condition false.
b) Cannot apply. Too short; second condition false.
c) Can apply. Both conditions true.

Activity 1.20

Other possibilities are:

Both conditions are true
condition 1 is true and condition 2 is false

Activity 1.21

```
IF Ctrl key pressed AND O key pressed THEN
  Request filename
ENDIF
```

Activity 1.22

```
IF double thrown OR fine paid OR used "Get Out of
↳Jail Free" card THEN
  Player gets out of jail
ENDIF
```

Activity 1.23

```
Turn over card
WHILE NOT card is Ace AND NOT card is KING DO
  Turn over card
ENDWHILE
```

Activity 1.24

a) Boolean expression.
b) Boolean operators are: AND, OR, and NOT.
c) Both conditions must be true.
d) NOT has a higher priority and will be performed first.
e) By enclosing the OR operator and its operands in parenthesis, that operation will be performed before the AND.

Activity 1.25

a) Its name and value.
b) From outside the system. In a computerised setup, this is often entered from a keyboard.
c) The relational operators are:

<	(less than)
<=	(less than or equal to)
>	(greater than)
>=	(greater than or equal to)
=	(equal to)
<>	(not equal to)

Activity 1.26

Algorithm with both count and total:

```
Set count to zero
Set total to zero
Turn over card
  WHILE NOT card is Ace AND NOT card is KING DO
    Add card value to total
    Turn over card
  ENDWHILE
  State value of count
  State value of total
```

Activity 1.27

No solution required.

Activity 1.28

The LEVEL 1 is coded as:

```
1. Draw grids
2. Add ships to left grid
3. REPEAT
4.     Call grid position(s)
5.     Respond to other player's call(s)
6. UNTIL there is a winner
```

The expansion of statement 4 would become:

```
4.1 Call grid reference
4.2 Get reply
4.3 WHILE reply is HIT DO
4.4    Mark position in second grid with X
4.5    Call grid reference
4.6  Get reply
4.7 ENDWHILE
4.8 Mark position in second grid with 0
```

The expansion of statement 5 would become:

```
5.1 REPEAT
```

```
5.2    Get other player's call
5.3    IF other player's call matches position of
       ↳ship THEN
5.4      Call HIT
5.5    ELSE
5.6      Call MISS
5.7    ENDIF
5.8 UNTIL other player misses
```

Activity 1.29

Instruction	Condition	T/F	Variables *number*	Variables *guess*	Output
1			42		
2					
3				75	
4	guess = number	F			
6					
7	guess < number	F			
9					
10					Too high
11					
12					
13	guess = number	F			
2					
3				15	
4	guess = number	F			
6					
7	guess < number	T			
8					Too low
11					
12					
13	guess = number	F			
2					
3				42	
4	guess = number	T			
5					Correct
12					
13	guess = number	T			

The expected results (as shown in the Test Data table) were obtained.

Activity 1.30

No solution required.



Background Intel

In this Chapter:

- ❑ How Number Systems Work
- ❑ The Binary System
- ❑ Converting Values between Number Bases
- ❑ Floating Point Formats
- ❑ Hexadecimal
- ❑ Negative Numbers Format
- ❑ Octal
- ❑ Storing Characters

Number Systems

Introduction

The counting system we use today is the decimal or, more correctly, the **denary** system. It uses ten different symbols (0,1,2,3,4,5,6,7,8,9) to represent any numeric value. The number of digits used in a number system relates directly to the **base** (also known as the **radix)** of that system, hence denary is a base 10 number system.

In all modern number systems the position of a digit affects the value represented. Hence, 19 and 91, although containing the same digits, represent two different values. When we start school we are often taught the theory of our positional number system by the use of column headings:

Thousands	Hundreds	Tens	Units

To represent a value we merely write the required numeric symbol in each of the appropriate columns. So seven hundred and twelve is represented by placing the appropriate digit under the correct column:

Thousands	Hundreds	Tens	Units
	7	1	2

If you know a little more mathematics, then the column identities can be changed to powers of 10:

10^3	10^2	10^1	10^0
	7	1	2

Note that the column values are based on the number system radix. So we can generalise to say that for a number system using base *R*, the column values can be represented by

R^3	R^2	R^1	R^0

Computers and the Binary System

Because of the modern computer's memory design, all the information it holds, be it program instructions, numbers, text, images, sounds or video, are stored as a sequence of numeric codes.

The fundamental electronic component of computer memory is the **bit**. A bit acts rather like a light switch - it can be set to one of only two positions. Rather than consider these two positions as *off* and *on*, we treat them as numeric values, 0 and 1.

There's not much scope to store large numbers when you have only a single bit to play with, so computer memory design joins the bits into groups of 8. A group of 8 bits is known as a **byte**. Half a byte (4 bits) is sometimes known as a **nybble**.

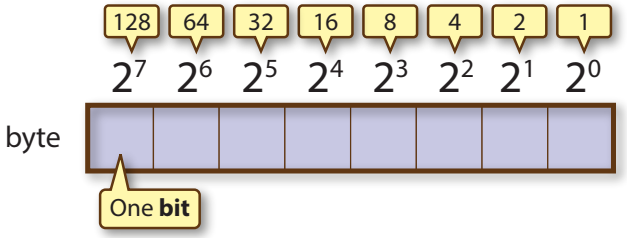
If we think of the 8 bits of a byte as the columns of a number, and knowing that each bit/column can contain only a 0 or a 1, then we can see that the computer has to make use of a base 2 number system. The term **binary** means *consisting of two parts*, and so this number system is known as the **binary number system**.

FIG-2.1 shows an abstract visual representation of a byte and the values assigned to

each column.

FIG-2.1

The Structure of a Byte



From this we can work out that to represent the decimal value 14 in binary, we would write

128	64	32	16	8	4	2	1
				1	1	1	0

Actually, in the computer, all bits must be 0 or 1 so the actual pattern stored within a byte would include leading zeros (00001110) but for the moment we'll omit the leading zeros from our discussions on binary numbers.

Since working in more than one number base at the same time can lead to confusion, the base of a number is often added as a subscript. For example, we can state that 14 in the decimal system is written as 1110 in the binary system with the line

$$14_{10} = 1110_2$$

Converting From Decimal to Binary

If we want to convert a decimal number to binary, we need to work out what numbers from the binary system column values add up to our decimal number. In the case of, for example, the value 23, it can be constructed from 16 + 4 + 2 + 1 (all binary column values). So we can say that 23 is written in binary as

128	64	32	16	8	4	2	1
			1	0	1	1	1

In other words,

$$23_{10} = 10111_2$$

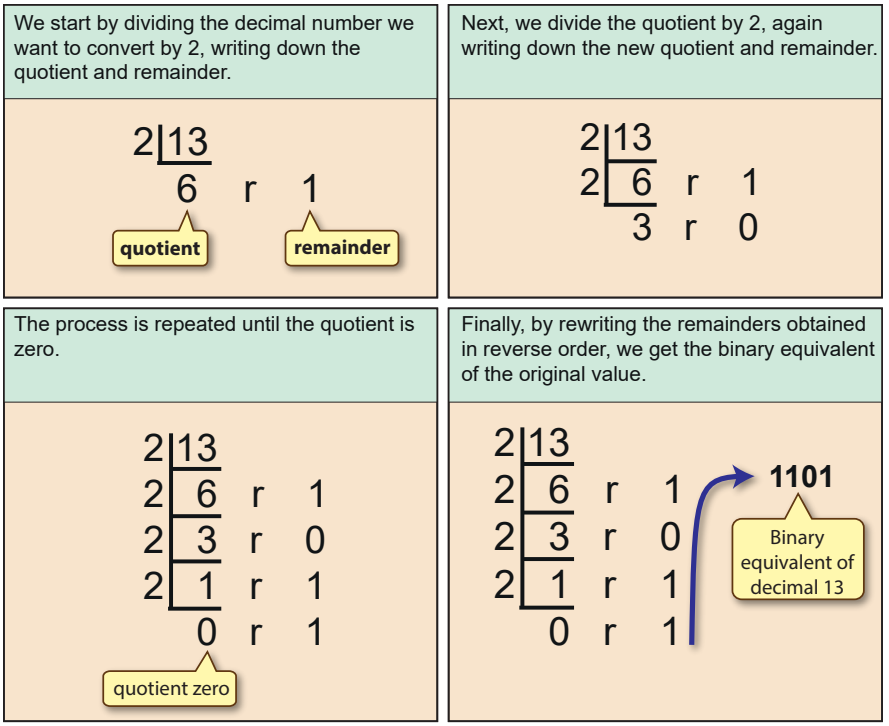
Although this approach is easy enough to use for small numbers, it gets a bit harder with larger values, so a more methodical approach to converting decimal values to binary is required.

That approach involves dividing the decimal number to be converted by 2, writing down the quotient and remainder, then continuing the process, always dividing the quotient by 2. When the quotient reaches zero, the remainders are written in reverse order (latest one first) and this gives us the binary equivalent of the original decimal value.

FIG-2.2 shows how this method is used to convert 13_{10} to binary.

FIG-2.2

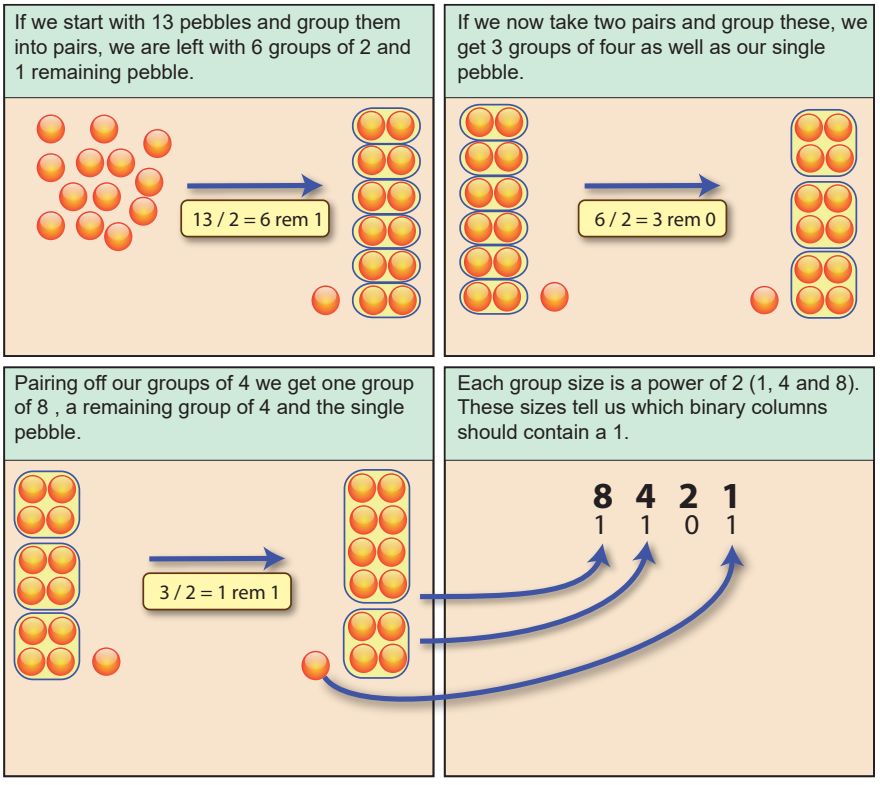
Converting Decimal to Binary



Perhaps it is not blindingly obvious why this approach gives us the result we are after but if we go through the process again using 13 pebbles, then it should become clearer (see FIG-2.3).

FIG-2.3

Why Division by 2 Works



Activity 2.1

Convert the following decimal values to binary using the division by 2 method described earlier.

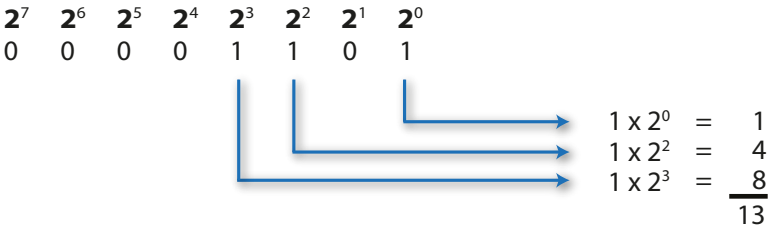
a) 19 b) 72 c) 63

Converting from Binary to Decimal

To convert from binary to decimal, take the value of each column that contains a 1 and add these values together to get the decimal equivalent (see FIG-2.4).

FIG-2.4

Binary to Decimal



Activity 2.2

Convert the following binary values to decimal using the method shown in FIG-2.4.

a) 00101001 b) 11111111 c) 10101010

Activity 2.3

What is the largest value (binary and decimal) that can be stored in a nybble?

Converting Fractions

Decimal Fractions to Binary

So far we have looked at converting only whole numbers but we also need to be able to represent decimal fractions in binary.

In the decimal system, column values to the right of the decimal point are

10^{-1} 10^{-2} 10^{-3} etc.

alternatively written as

$\frac{1}{10}$ $\frac{1}{100}$ $\frac{1}{1000}$

Column values to the right of the binary point, on the other hand, have the values

2^{-1} 2^{-2} 2^{-3}

or

$\frac{1}{2}$ $\frac{1}{4}$ $\frac{1}{8}$

or

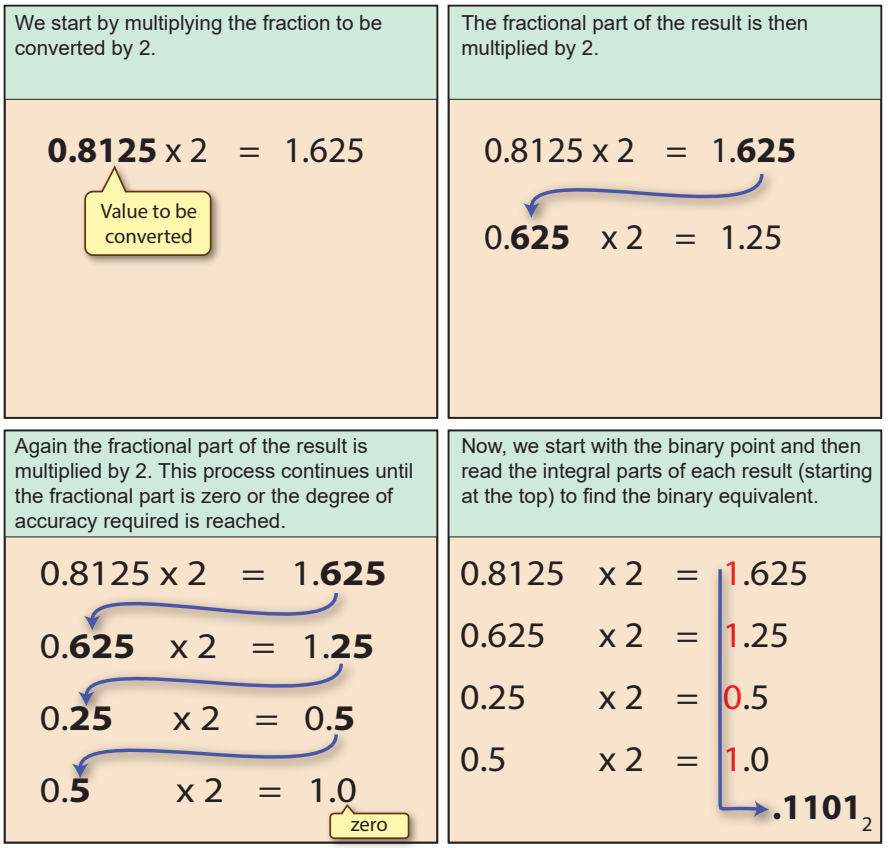
0.5 0.25 0.125

To convert a decimal integer, we had to find how that number could be expressed in terms of values which were exact powers of 2. For example, we saw that 13 could be expressed as $8 + 4 + 1$ ($2^3 + 2^2 + 2^0$). To convert a decimal fraction to a binary fraction we need to find the same thing; powers of 2 that express the original value. The only difference this time is that the power value will be negative since the values involved are less than one. For example, the value 0.625 can be expressed as $0.5 + 0.125$ ($2^{-1} + 2^{-3}$).

This time the standardised conversion process involves continually multiplying fractions by 2 and eliminating the integral part after each multiplication. The steps for converting 0.8125_{10} to binary are shown in FIG-2.5.

FIG-2.5

Converting Decimal Fractions to Binary



When we first multiply by 2, if the number to be converted is 0.5 or greater, we will get a 1 in the integral part of the answer. This tells us that there was a 0.5 component in the original number. By ignoring that integral part in the next calculation we have, in effect, removed the 0.5 value from our calculation. The next multiplication (at this point we have multiplied by 4 — multiplication by 2 twice), causes any 0.25 component in the original value to produce an integral of 1. This integral is then ignored in the next calculation.

Unlike integer values, when we convert a decimal fraction to a binary one, we have no guarantee that the binary fraction we produce will be exactly equivalent to the decimal fraction. In these cases, we must convert to the required number of binary places.

Activity 2.4

Convert the decimal value 0.3125 to a binary fraction.

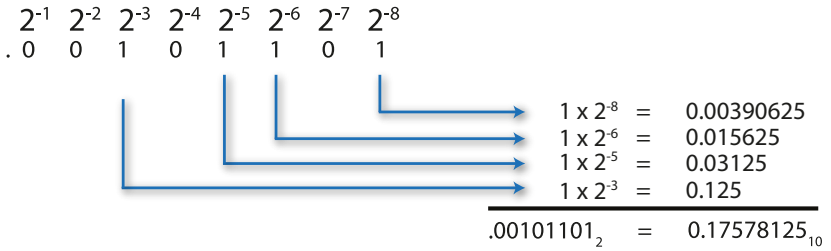
To convert a number which has integral and fractional parts, convert each part separately as if they were two separate values.

Binary Fractions to Decimal

Binary fraction to decimal does not require a new approach. It is handled in exactly the same way as binary whole numbers were converted. The only difference in this case is the column values involved (see FIG-2.6).

FIG-2.6

Converting Binary Fractions to Decimal



Activity 2.5

Convert the binary value 0.100101 to a decimal fraction.

Hexadecimal

Hexadecimal (often shortened to **hex**) is another number system widely used in computing. This number system has the base 16 so it follows that there must be 16 different digits used in the representation of hexadecimal values.

These 16 digits are formed from the 10 digits of the decimal system and the first 6 letters of the alphabet So in hexadecimal 0 to 9 represents the normal decimal values 0 to 9 but A represents 10, B 11, C 12, D 13, E 14 and F 15. With a base 16 number system the column values for this system are:

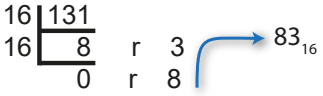
	16^3	16^2	16^1	16^0
or				
	4096	256	16	1

Converting from Decimal to Hexadecimal

Conversion from decimal to hexadecimal is similar to the conversion from decimal to binary. However, this time, instead of dividing by 2, we divide by 16. FIG-2.7 shows how the number 131_{10} is converted to hexadecimal.

FIG-2.7

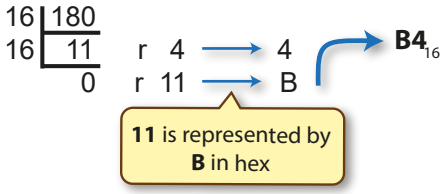
Converting Decimal to Hexadecimal 1



Things are slightly more complicated if the remainder is greater than 9, since we have to remember to convert the remainder to the appropriate hexadecimal letter (see FIG-2.8).

FIG-2.8

Converting Decimal to Hexadecimal 2



Activity 2.6

Convert the following decimal values to hexadecimal:

a) 97 b) 212 c) 255

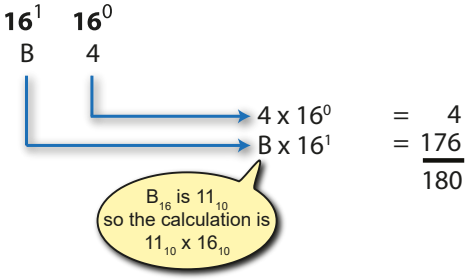
Converting from Hexadecimal to Decimal

Again the conversion from hexadecimal to decimal is similar to that for binary to decimal; the value in a given column is multiplied by the column value. The only change this time is that a wider range of digits may appear in any one column and that the column values are different.

FIG-2.9 shows the value $B4_{16}$ being converted to decimal.

FIG-2.9

Converting Hexadecimal to Decimal



Activity 2.7

Convert the following hexadecimal values to decimal:

a) 2C b) A6 c) DE

Converting from Binary to Hexadecimal

Although the computer uses binary for everything it does, we humans find that number system a bit long-winded since it takes a large number of digits to represent even relatively small values. The other major problem we have with binary is trying to copy out values correctly; with only 0's and 1's, its all too easy to make a mistake when copying so many digits.

Using hexadecimal gives us a way of avoiding writing values in binary and yet, at the same time, making it easy to convert to and from binary when necessary.

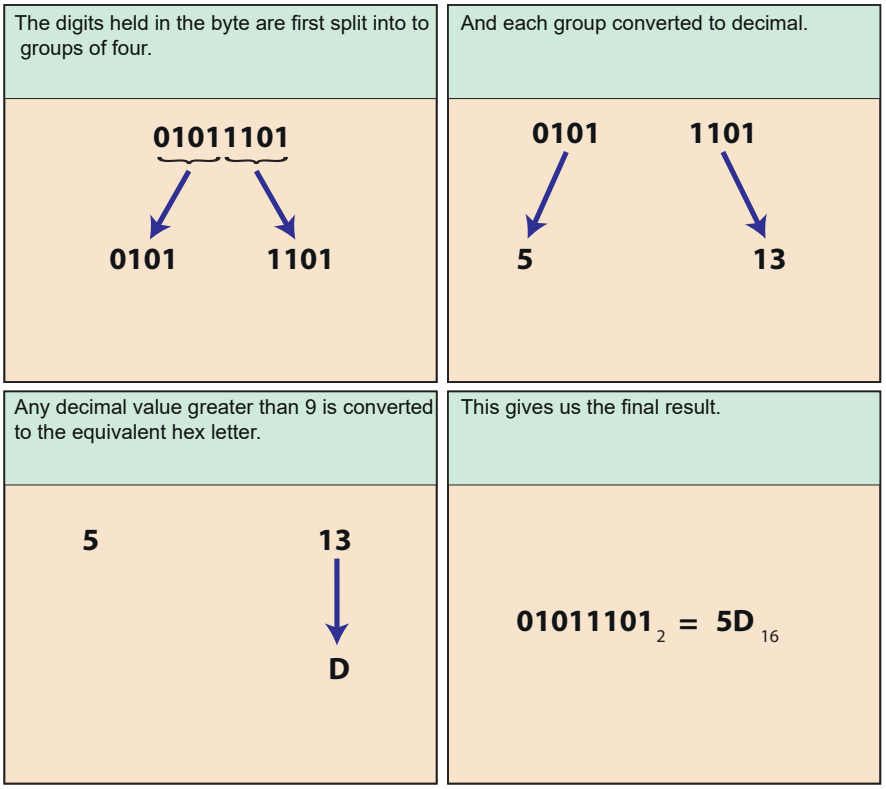
In hexadecimal, **F** is the highest value digit being equal to 15_{10}. Now 15 also happens to be the maximum value that can be represented in the four bits of a nybble. This

means that one hexadecimal digit can be used to represent four bits; two hexadecimal digits can represent a byte.

FIG-2.10 shows the steps involved in converting the 8 bits of a byte to hexadecimal.

FIG-2.10

Converting Binary to Hexadecimal



Activity 2.8

Convert the following binary values to hexadecimal:

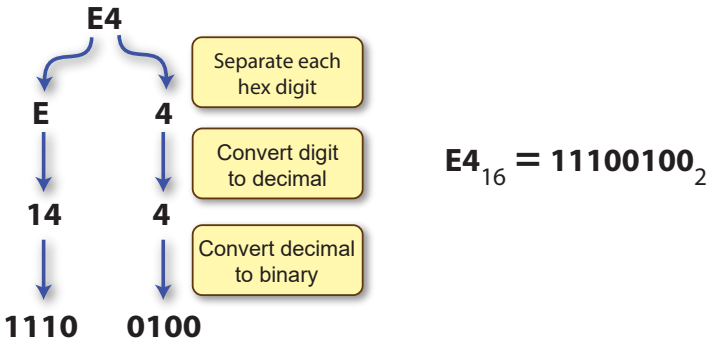
a) 01000111 b) 11111111 c) 11001011

Converting from Hexadecimal to Binary

This is nothing more than a reverse of the previous process. Each hexadecimal digit is converted to decimal and then to a 4 bit value. The process is shown in FIG-2.11.

FIG-2.11

Converting Hexadecimal to Binary



Activity 2.9

Convert the following hexadecimal values to binary:

a) AB b) 8C c) 9A

Octal

Octal is a base 8 number system using the digits 0 to 7. At one time some computers had their memory organised into 6 bit blocks rather than the usual 8. Just as hexadecimal is used as a convenient way of representing 4 bits, so octal was used to represent 3 bits. Two octal digits specifying the contents of 6 bits. Although little used now, octal is included here simply for completeness and because there are options in C++ to allow octal values to be used.

Since octal uses the base 8, column values would be:

8^4 8^3 8^2 8^1 8^0

Activity 2.10

a) Using a division by 8 approach, convert the decimal value 147 to octal.

b) Convert the octal value 75 to decimal.

To convert binary to octal we split the binary value into groups of 3 bits (starting from the right) then convert each group to its octal equivalent. This process is reversed to convert octal to binary.

Activity 2.11

a) Convert the binary value 101110 to octal.

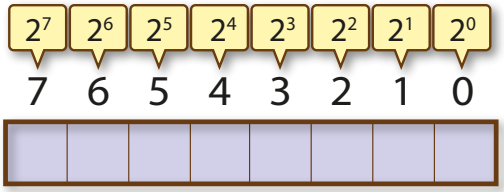
b) Convert 34_8 to binary.

Storing Numbers

The bits within a byte are identified in diagrams by allocating the exponent value of the appropriate number column (see FIG-2.12).

FIG-2.12

Identifying the Bits within a Byte



The right-hand bit (bit 0) is known as the **least-significant bit** since it is of the least numeric value; the left-hand bit is the **most-significant bit**.

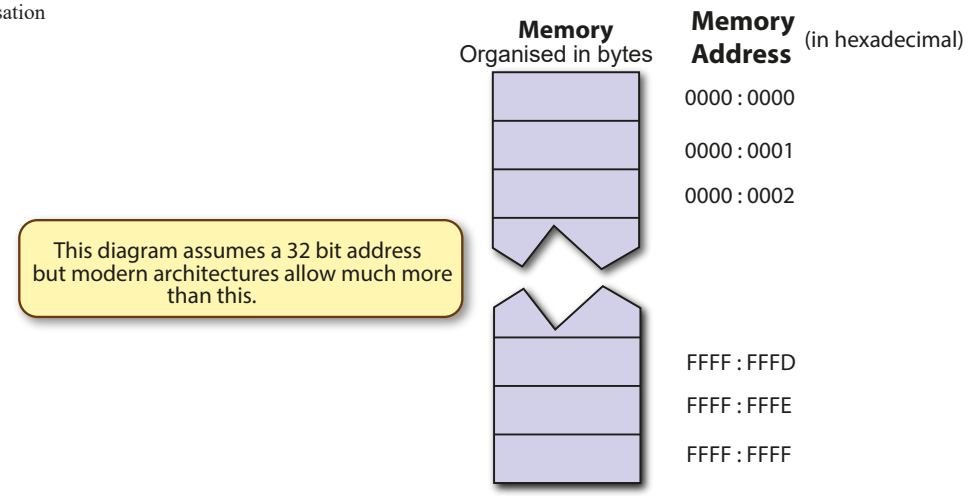
Using a single byte, values in the range 0_{10} (00000000_2) to 255_{10} (11111111_2) can be stored.

When a whole number is stored in a computer's memory, it will normally occupy one, two, four or even eight bytes. The more bytes that are allocated to it, the larger the range of values that can be stored.

Memory is designed in such a way that every byte is allocated a unique address (just like every house in a street has its own unique address). This means that the computer can directly access any byte by specifying the address of that byte (see FIG-2.13).

FIG-2.13

Memory Organisation



The individual bits within a byte cannot be directly accessed but other methods are available to determine the contents of any single bit.

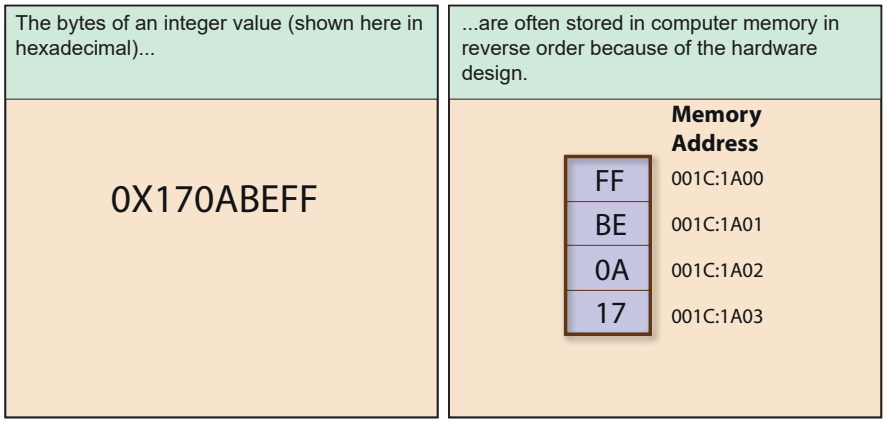
When using multiple bytes to store a value, the right-most byte is known as the **least-significant byte**; the left-hand one, the **most-significant byte**.

Using eight bytes, numbers between 0 to 18,446,744,073,709,551,615 can be stored.

It should be noted that in reality it is common practice to store the bytes of a number in reverse order when written to computer memory (see FIG-2.14).

FIG-2.14

How an Integer is stored in Memory



Despite the apparently strange storage format, the hardware automatically handles the storing and retrieval of the data, ensuring that the original value is always presented unchanged to any program accessing the value.

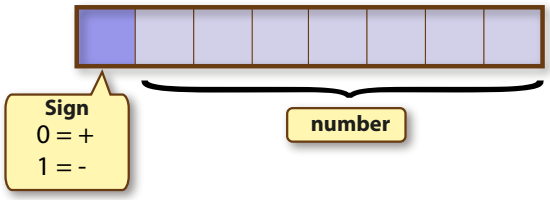
Negative Numbers in Binary

From what we've seen so far, binary can be used to store huge positive values, but how can we store a negative number such as -17?

Remembering that absolutely everything in a computer is stored in 0's and 1's we have to be imaginative with our approach to representing negative values. For example, when storing the number in a single byte, we could reserve one bit to be the sign of the number with a 0 representing + (positive) and a 1 representing - (negative) (see FIG-2.15).

FIG-2.15

Format for Storing Signed Values



At first glance, this might seem to solve our problem. We can represent +7 as 00000111 and -7 as 10000111. Of course, because there are now only 7 bits available for the magnitude of the number, we're limited to a range from -127 (11111111) to +127 (01111111). Another curious thing about this approach is that we have two ways of representing zero: 00000000 (+0) and 10000000 (-0). However, the biggest problem with this approach is that adding a negative number to a positive number doesn't produce the correct results. For example, we know that

```
9 + (-7) = 2
```

But when we attempt the same thing with our binary values

```
00001001 (+9)
10000111 (-7)
------------
10010000 (-16)
```

we arrive at an incorrect result.

One's Complement

Another possible way of representing a negative value is to make the digits exactly the opposite of the positive form. So if +7 is written as

```
00000111
```

then -7 becomes

```
11111000
```

> **Activity 2.12**
>
> Write down the equivalent of the decimal value -248 using the above system.

We have to be careful not to confuse ourselves if we use this approach.

Of course, we cannot allow a situation where a particular binary pattern — in this

case 00000111 — can have an ambiguous value (7 or -248). The solution to this is again to limit the magnitude of the values allowed. If positive value can only go as high as 127 then bit 7 (the left-most bit) will always be zero, so when we complement all the bits to represent a negative value, the left-most bit will always be a one. So, again, the left-most bit turns out to be our sign bit. When it contains a zero we have a positive number; when it contains a 1, we have a negative value.

This way of representing negative values is known as **1's complement form**.

So let's see if 9 + (-7) gives the correct result this time:

```
  00001001 (+9)
  11111000 (-7)
 ------------
1 00000001
```

Notice that the result is 9 bits long, not 8. But this extra, left-hand bit (known as the **overflow bit**) is of no consequence to us since there is no room to store it in a single byte, so it can be eliminated from our result. This leaves us with the value

```
00000001 (+1)
```

This time the result is just 1 out from the correct answer.

Activity 2.13

Using 1's complement form for the binary values, calculate the result obtained when performing the following addition 14 + (-5).

In fact, using 1's complement always gives us a result that is exactly 1 less than the true result. Well, that's easily fixed — we just have to do our calculation and then add 1 to whatever result we get.

Two's Complement

However, another way of dealing with the problem is to add that extra 1 to the negative value before you start the calculation. This is known as **2s complement form**. So -7 is represented in 2's complement form by

```
11111000   (-7 in 1s complement form)
      +1
--------
11111001
```

This time when we do our calculation 9 + (-7) we get

```
  00001001 (+9)

  11111001 (-7)
------------
  00000010 (2)
```

At last we have the correct result.

Activity 2.14

Using 2's complement form, redo the calculation 14 + (-5).

Notice that the left-most bit of the value still acts as a sign bit: 0 when a positive value

is stored, 1 for a negative value.

Two's complement allows values in the range -128 to + 127 to be stored in a single byte.

-128_{10} is stored as 10000000_2.

When 2s complement form is used to store a value over two bytes, it can store values in the range -32,678 to +32,767.

Many programming languages allow numbers to be stored in either **unsigned** (zero and positive numbers only) format or **signed** (negative, zero and positive values) format.

In unsigned format all the bits assigned hold the number's value giving a large range of positive values, but negative numbers cannot be stored. Signed format uses 2's complement, effectively allocating 1 bit for the sign bit with the remaining bits recording the value. Although this allows negative numbers to be recorded, it halves the largest possible value that can be stored.

Floating Point Values in Binary



The E notation is most often seen on a calculator or computer. When written by hand, the E is usually replaced by x10.

In the decimal system, real numbers (those with a decimal point) can be expressed in a different way from whole numbers. On paper these type of values are either written using **fixed point notation** with a fixed number of digits after the decimal point (e.g. 128.3 — one digit after the decimal point) or in **scientific notation** (e.g. 1.283E2).

Scientific notation may look a little strange if you haven't come across it before, but it is really quite easily understood:

> The letter **E** stands for the term **exponent** or **x10 raised to the power**
> The number to the left of the E is called the **significand**.
> The number to the right of the E is the **exponent value**.

The exponent represents the power to which 10 is to be raised. So E01 means 10^1 or simply 10; E02 means 10^2 or 100.

To arrive at the number being represented, we perform the calculation

$$\text{significand} \times 10^{\text{exponent}}$$

so

$$1.283E2$$

$$= 1.283 \times 10^2$$

$$= 1.283 \times 100$$

$$= 128.3$$

Where we have a negative exponent such as 10^{-1} or 10^{-2} then these represent 1/10 (0.1) and 1/100 (0.01) respectively. So a number shown in scientific notation as

1.67E-3

is

1.67 x 10-3
⇒ 1.67 x 0.001
⇒ 0.00167

Of course, we might have written the value 128.3 as .1283E03 or 12.83E1 or 1283E-01 but the convention is to make sure that the integral part of the significand lies in the range 1 to 9. When the significand is within this range, it is termed a **normalised significand**.

For values less than 1, normalising the significand will mean that the exponent will be negative. For example,

0.000013

⇒ 1.3E-5

If the number being represented is negative, then the significand is negative:

-382.19
⇒ -3.8219E2

Activity 2.15

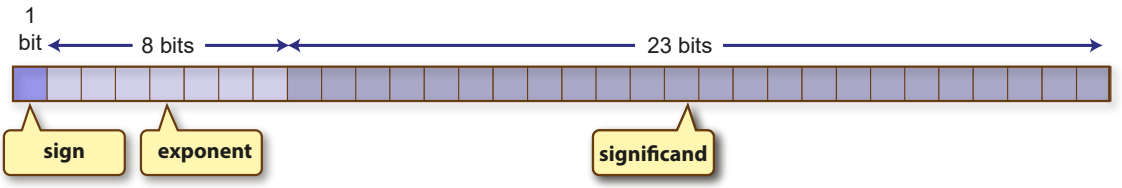
Rewrite the following values in standard notation:

a) 8.7512E3 b) -3.8122E2 c) 6.1937E-2

Most software uses the significand-exponent approach for storing real numbers in a format known as **floating-point** but, of course, the significand and exponent are held as binary values.

The standard layout for a 32 bit floating point value is shown in FIG-2.16.

FIG-2.16 Storing Floating Point Values



The exact formatting on the computer is slightly different from the decimal version.

The exponent is said to have a **bias** of 127.

- The exponent is always 127 greater than its true value. This is simply a method of eliminating the need to use 2's complement when the true exponent is negative.
- The significand is adjusted so its value lies between 1 and 2.
- The sign bit is 0 if the number is positive; 1 if the number is negative.

The significand itself is always stored in positive form even when the value represented is negative; 2's complement is not used. Also, the leading 1 of the significand is not actually stored in memory; instead, its presence is assumed when the computer performs any subsequent calculations.

Let's see how the number 28.75 would be stored in this format:

$28.75_{10} = 11100.11_2$

Normalising the binary mantissa we get:

1.110011 E 100

The leading 1 of the significand is assumed, so only

110011 is stored.

Adding 127 to the exponent gives

10000011

So, when stored in the 32 bit format shown above we get

0 10000011 11001100000000000000000

Activity 2.16

a) Convert the value 0.00872 to binary floating point format.

b) A floating point formatted value is held within the computer as

```
1  10000001  01011010000000000000000
```

Calculate the decimal equivalent of this value.

There are two main situations where a floating point value is interpreted differently by the software:

- When the exponent is zero and the significand is zero, the value held is assumed to be zero.
- When the exponent has the value 11111111_2 (FF_{16}) and the significand is zero, this represents infinite. When the exponent is FF_{16} but the significand is not zero, this represents an error condition and is often shown in program output as NaN (**N**ot **a** **N**umber).

Note that when using 64 bits to store a floating point value, the leading 1 in the significand is actually stored rather than assumed.

Character Coding

As well as numbers, computers need to store characters. Since everything within the machine is stored in binary, this means that we need some sort of coding system to represent characters.

The most universally used coding system in the past has been American Standard Code for Information Interchange (**ASCII** — pronounced *ask - ay*). This uses a single byte to store a character with codes for upper and lower case letters, punctuation marks, numeric digits and a few other symbols.

Only 7 of the 8 bits in a byte are used for the character, the 8th bit originally being

used as a parity bit to help with detection of errors produced during the transmission of data.

In ASCII a capital A is coded as 01000001; a B as 01000010, etc.

The ASCII coding system is quite restrictive with no scope for representing non-European characters.

To cope with the wider range of characters, the **Unicode Standard** was created which has assigned a unique code to every possible character (over 100,000). This coding convention is now used by all modern software.

The most widely used of the Unicode character-coding systems is **UTF-8** (**U**nicode **T**ransformation **F**ormat -8 bit). This uses a variable number of bytes for coding characters. For the original ASCII character set, UTF-8 uses a single byte, employing exactly the same codes as ASCII. When other characters such as those from Greek, Hebrew and Arabic are used, UTF-8 uses two bytes per character. Most other language characters (Chinese, Japanese, etc) require three bytes of coding. More specialised symbols (some mathematical and historic scripts) make use of a fourth byte.

Characters can also be coded using UTF-16 (which codes in characters in either 2 or 4 bytes as required) and UTF-32 (which codes all characters using 4 bytes).

Summary

- A modern number system's base or radix is determined by the number of different symbols used to represent characters.
- A number system's column values are the radix value raised to incrementing powers (increments right to left).
- Binary is a base 2 number system using the digits 0 and 1 to represent all values.
- An individual binary digit is known as a bit.
- A grouping of 4 bits is known as a nybble.
- A grouping of 8 bits (2 nybbles) is known as a byte.
- The base of a number is often included as a subscript where confusion might otherwise arise.
- Integer decimal values are converted to binary by continually dividing by 2 until the quotient is zero and then writing out the remainders in reverse order.
- An integer binary value can be converted to decimal by summing the value of all columns containing a 1.
- Decimal fractions are converted to binary by continually multiplying the fractional part of each result by 2 then listing the integral parts of the results.
- Binary fractions are converted to decimal by summing the values of all columns containing a 1.
- Hexadecimal is a base 16 number system using the digits 0-9,A-F.
- A single hexadecimal digit is a convenient way of representing 4 bits.
- Decimal can be converted to hexadecimal by continual dividing by 16 and

writing out the remainder (last one first). Any remainders of 10 or more must be converted to a hexadecimal letter.

- A hexadecimal value can be converted to decimal by multiplying each hex digit by the value of the column in which it is positioned and then summing the results.
- To convert binary to hexadecimal, split the binary value into groups of four bits, convert the four bits to decimal then convert any values greater than 9 to a hexadecimal letter.
- To convert from hexadecimal to binary, convert each hexadecimal digit to exactly 4 bits.
- Octal is a base 8 number system using the digits 0 - 7.
- A single octal digit is used to represent 3 binary digits.
- Negative integer values are stored in 2s complement form.
- A negative number's 2's complement form is derived by taking the binary form of a positive number, inverting all the digits and adding 1.
- The computer uses floating-point format to store real numbers.
- Floating point format has three components:

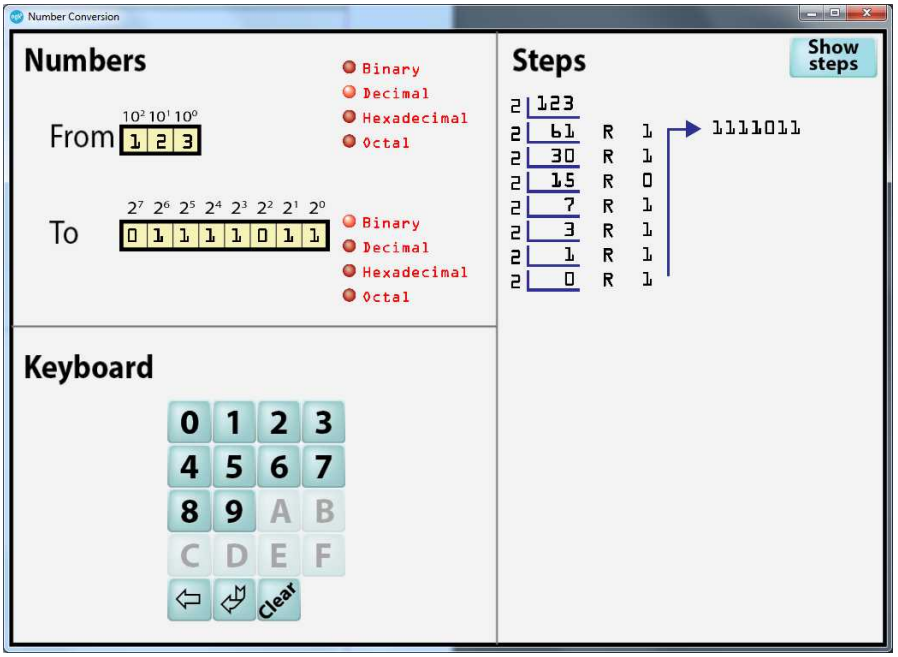
  sign bit
  exponent
  significand

- The sign bit is 0 for positive values; 1 for negative values.
- The significand of a floating point value is always stored in its positive form even when the value represented is negative.
- The exponent has a bias (or offset) of 127 removing the need to store negative values.
- When using 32 bits to store a real value, the significand is normalised to assume a leading 1 which is not stored.
- When using 64 bits to store a real value, the leading 1 in the normalised significand is stored.
- Characters can be coded in a single byte using ASCII format.
- The most widely used character coding system now in use is UTF-8 which uses a variable number of bytes per character.
- UTF-8 uses the same single-byte coding as ASCII for the ASCII character set.

Support Material for this Chapter

Integer Number Converter (*Numbers.exe*)

Screen Shot



Overview

This program allows you to convert a positive integer number from one base to another and shows how the conversion would be achieved manually.

User Instructions

In the Numbers Panel:

Select the *From* number base using the top set of radiobuttons. The keyboard will highlight only the keys appropriate to that number base

Select the *To* number base using the bottom set of radiobuttons.

In the Keyboard Panel:

Type in the value you want to convert. You can use the ⇦ button to delete the last digit or the **Clear** button to delete all digits entered.

Press the ⏎ button to enter your completed value. The value you entered will automatically be converted to the other number base and displayed in the *To* area.

In the Steps Panel:

Press the **Show Steps** button to reveal how the conversion between the selected number bases would be performed manually.

Error Messages

You'll get an error message if you try to enter a value too large to be stored in a single byte.

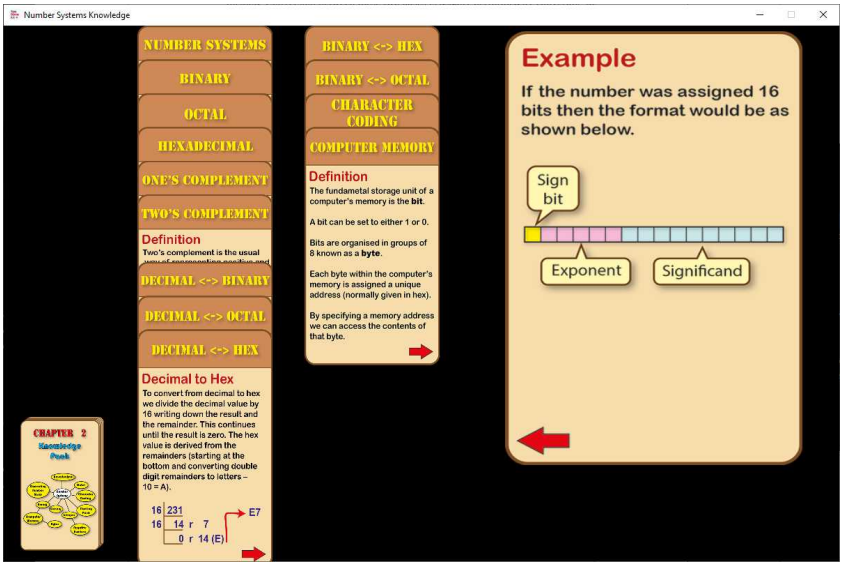
You'll get an error message if you try to display the steps required to convert directly from Octal to Hexadecimal (or vice versa) since there is no direct manual conversion method. Conversion between these two bases is usually performed by converting to binary as an intermediate step.

Download

The app file is called *Numbers.exe* and can be found in the *Cpp/Resources/Ch02/Numbers* folder of the download material for this book.

Reviewer (*Remember02.exe*)

Screen Shot



Overview

This program displays a set of cards summarising various topics covered in this chapter.

User Instructions

The user instructions match those of the **Reviewer** app in Chapter 1.

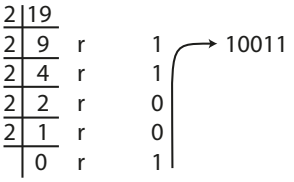
Download

The app file is called *Remember02.exe* and can be found in the *CPP/Resources/Ch02/Remember02* folder of the download material for this book.

Solutions

Activity 2.1

a) Working:



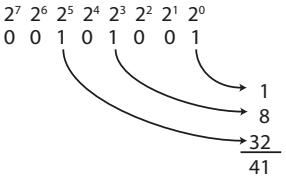
a) $19_{10} = 10011_2$

b) $72_{10} = 1001000_2$

c) $63_{10} = 111111_2$

Activity 2.2

a) Working:



a) $00101001_2 = 41_{10}$

b) $11111111_2 = 255_{10}$

c) $10101010_2 = 170_{10}$

Activity 2.3

The largest value that can be stored in a nibble is $1111_2 = 15_{10}$.

Activity 2.4

0.3125	x 2	=	0.625
0.625	x 2	=	1.25
0.25	x 2	=	0.5
0.5	x 2	=	1.0

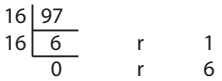
$0.3125_{10} = .0101_2$

Activity 2.5

$0.100101_2 = 0.5 + 0.0625 + 0.015625 = 0.578125_{10}$

Activity 2.6

a) Working:



a) $97_{10} = 61_{16}$

b) $212_{10} = D4_{16}$

c) $255_{10} = FF_{16}$

Activity 2.7

a) $2C_{16} = 44_{10}$

b) $A6_{16} = 166_{10}$

c) $DE_{16} = 222_{10}$

Activity 2.8

a) $01000111_2 = 47_{16}$

b) $11111111_2 = FF_{16}$

c) $11001011_2 = CB_{16}$

Activity 2.9

a) $AB_{16} = 10101011_2$

b) $8C_{16} = 10001100_2$

c) $9A_{16} = 10011010_2$

Activity 2.10

a) $147_{10} = 235_8$

b) $75_8 = 61_{10}$

Activity 2.11

a) $101110_2 = 56_8$

b) $34_8 = 011100_2$

Activity 2.12

We start by converting the positive value 248_{10} to binary. This gives us 11111000_2. Now the bits are complemented, and we get 00000111.

Activity 2.13

`14`	=	`00001110`	
`-5`	=	`11111010`	(1's complement form)
adding gives		`00001000`	(+8)

Activity 2.14

`14`	=	`00001110`	
`-5`	=	`11111011`	(2's complement form)
adding gives		`00001001`	(+9)

Activity 2.15

a) 8751.2

b) -381.22

c) 0.061937

Activity 2.16

a)

$0.00872_{10} =$ `0 01111000 10001110110111100101010`

The significand is calculated as follows:

0.00872 x 2 = 0.01744
0.01744 x 2 = 0.03488
0.03488 x 2 = 0.06976
0.06976 x 2 = 0.13952
0.13952 x 2 = 0.27904
0.27904 x 2 = 0.55808

0.55808 x 2 = 1.11616
0.11616 x 2 = 0.23232
0.23232 x 2 = 0.46464
0.46464 x 2 = 0.92928
0.92928 x 2 = 1.85856
0.85856 x 2 = 1.71712
0.71712 x 2 = 1.43424
0.43424 x 2 = 0.86848

etc.

Hence the significand starts .00000010001110

Normalising this, we get 1.0001110

This means the exponent is -7
Adding a bias of 127 we get an exponent of 120
$120_{10} = 01111000_2$

The leading 1 in the mantissa is assumed, so it is stored
beginning with the digits 0001110.

b)

1 10000001 01011010000000000000000 = 5.40625

The exponent is 129
Subtracting the 127 bias, we get an exponent of 2.

Adding the assumed 1. to the significand, we get

1.0101101

Adjusting for the exponent, we get 101.01101

The integral part of this 101 is 5.

The fractional part is:

0.25 + 0.125 + 0.03125

= 0.40625

Since the sign bit is zero, the number is a positive one and,
expressed in decimal, is:

5.40625

3

Starting C++17

In this Chapter:

- ❑ Installing Visual Studio Community 2019
- ❑ Creating a First C++ Project
- ❑ Basic Code Layout
- ❑ Whitespace in a Program
- ❑ Syntax Errors
- ❑ Running a Program

Using Visual Studio

Before We Start

C++, like most other computer languages, has an exact international standard definition of its elements (ISO/IEC 14882:2014). This latest standard is often referred to as C++17. Currently, most of the features of this standard are implemented by the Microsoft Visual C++ compiler.

Having a standard for the language doesn't stop manufacturers adding features of their own. Microsoft's Visual C++ offers many more features than the defined standard for the language. For example, Visual C++ has the ability to create .NET managed code and graphical user interface (GUI) applications as well as including other, simpler additions to the standard C++17 language.

In this volume we will, for the most part, cover only the standard C++17 language.

There are many C++17 compilers available. Below we show how to download Microsoft's Visual Studio 2019 Community but you may be interested in online compilers (such as the one at **www.OnlineGdb.com)** which give instant access to a compiler without the need to install software on your machine.

Creating a Microsoft Account

Before we can use the free version of Visual Studio, we need to create a Microsoft account. The account requires your name and email address. A section of the form that has to be filled in is shown in FIG-3.1.

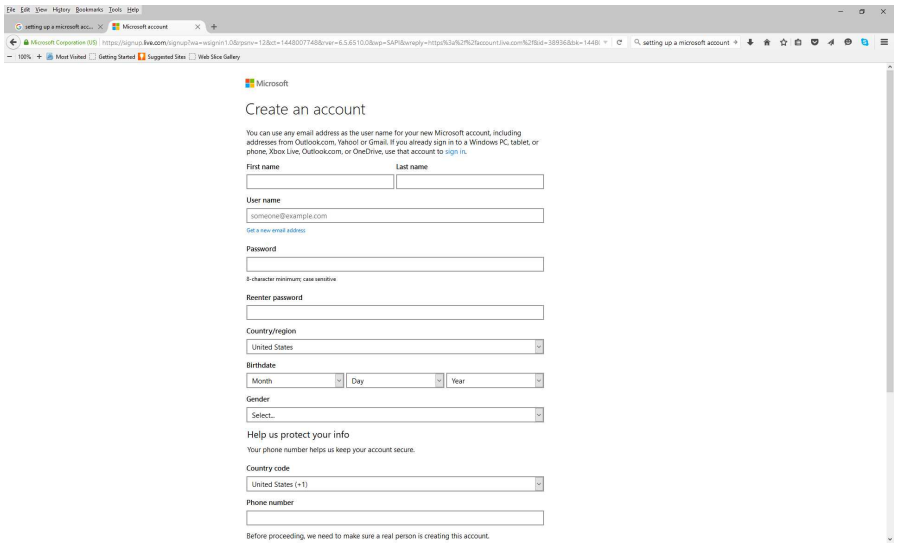


FIG-3.1

Creating a Microsoft Account

The actual web address for this form is currently *https://signup.live.com* but if, by the time you read this the address has changed, just google *"Create a Microsoft account"* and you'll find the necessary page.

Activity 3.1

If you do not already have a Microsoft account, go online and create one.

Downloading the Software

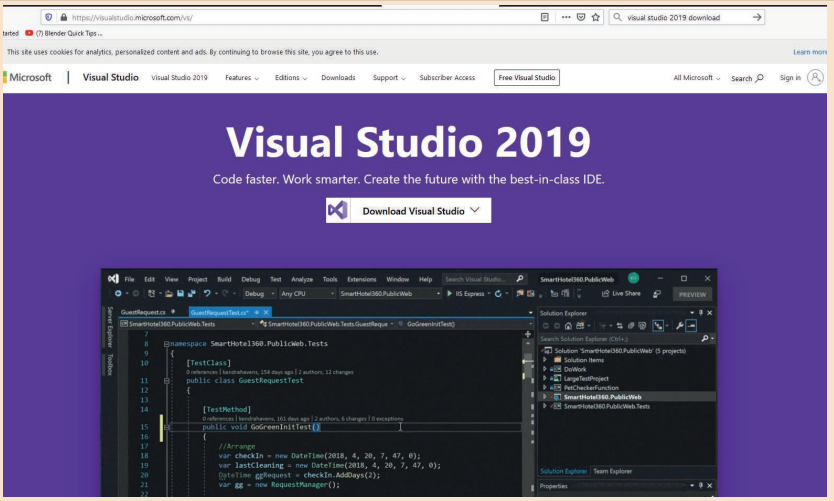
The next stage in the process is to download the free software from the Microsoft site. The package is known as *Visual Studio Community 2019*.

The steps required to install the software are shown in FIG-3.2.

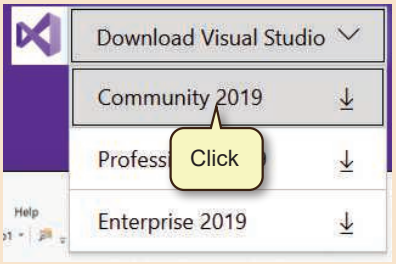
FIG-3.2

Installing Visual Studio 2019 Community

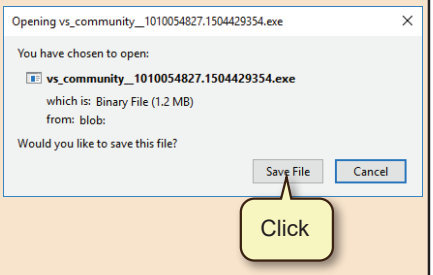
In your browser, enter the address *visualstudio.microsoft.com/vs/*



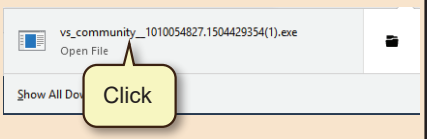
Move over *Download Visual Studio*, and select *Community 2019* from the drop-down menu.



This will produce the following dialog box where you need to click on the **Save File** button.



When the download is complete, click on the filename to start execution.



The package will take a few moments to download.

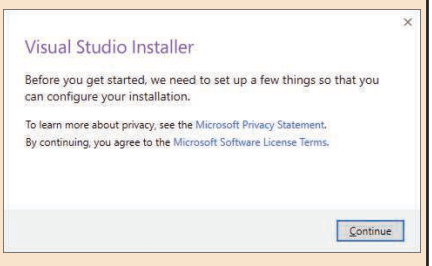
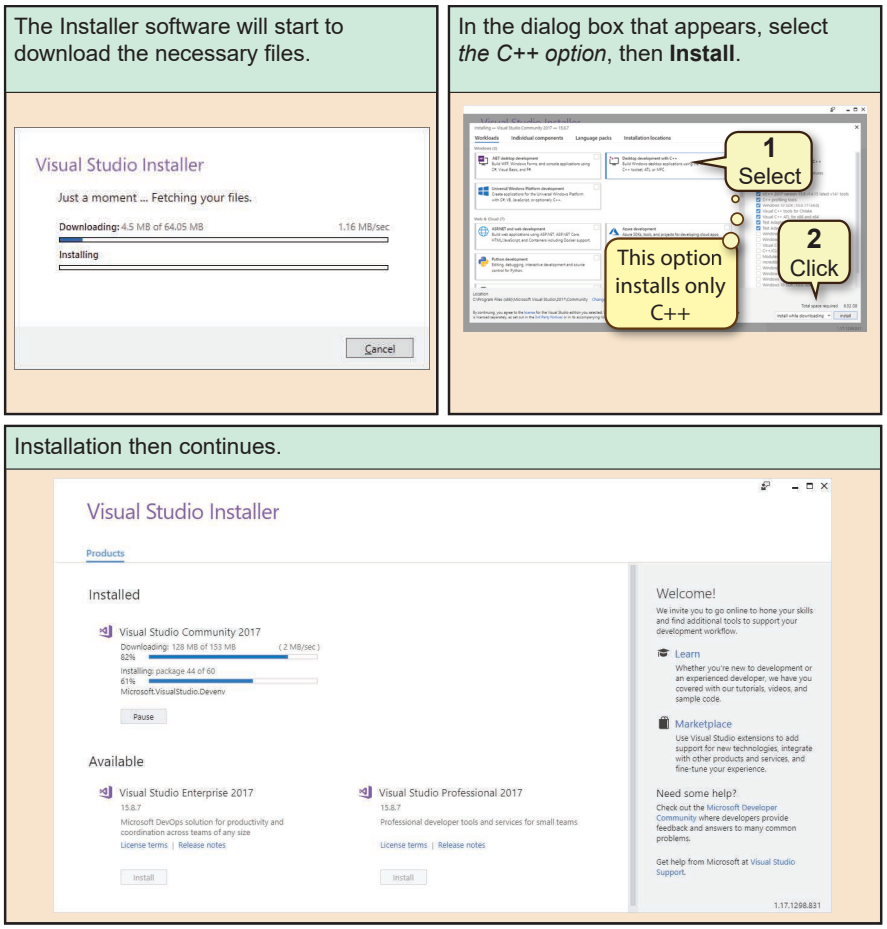


FIG-3.2
(continued)

Installing Visual Studio 2019 Community





Once *Visual Studio Community 2019* is installed, its icon should appear on the desktop.

Activity 3.2

Download and install *Visual Studio Community 2019* on your desktop.

The *Visual Studio Community 2019* software is a complete Integrated Development Environment (IDE) containing all that is required to create, compile, link and run the programs we write. Although it will allow us to create programs using various languages, we will only be looking at its C++ option.

The C++ IDE consists of several components:

The Editor is the most obvious part of the IDE allowing us to type and edit our code in much the same way as a regular text editor. It does offer some useful extra features however. Keywords and other elements in our programs are colour-coded and indenting of the instructions is semi-automatic. It also underlines any term that is invalid as we type.

The Precompiler (also called a **preprocessor**) performs initial preparatory operations on the program code before it is passed to the compiler.

The Compiler checks our code for syntax errors, reporting back on any that it finds. And when all of these errors have been eliminated, it translates our C++ source code into object code, ready to be executed by the computer.

The Linker combines the object code generated with any other existing library modules referenced by our program.

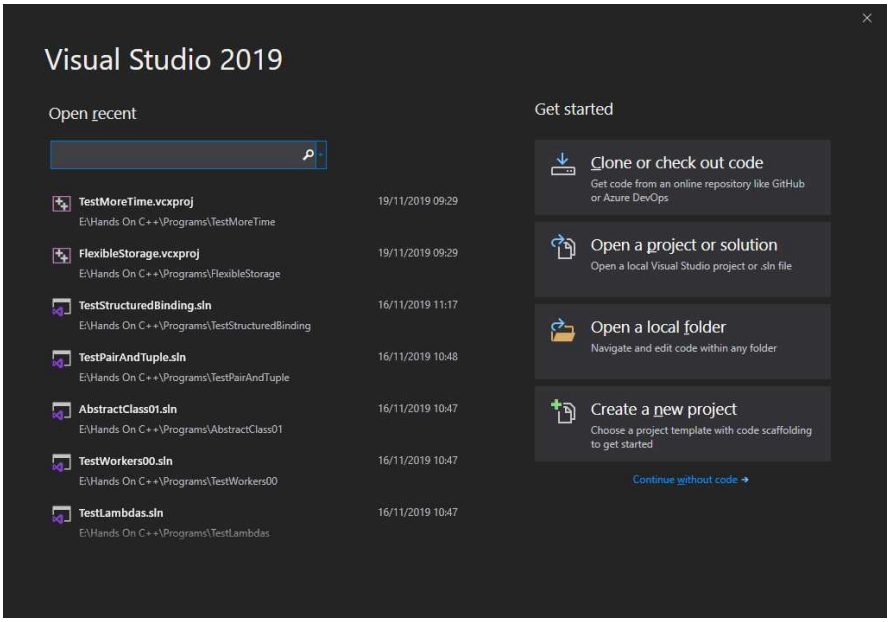
The Libraries are collections of existing object code routines which extend the basic features of the core C++ statements. There are several of these libraries, one of the most important being the standard C++ library. The routines held here allow us to perform various mathematical and string handling operations.

Start Up

Clicking on the *Visual Studio 2019* icon opens up a window similar to that shown in FIG-3.3.

FIG-3.3

The StartUp Screen

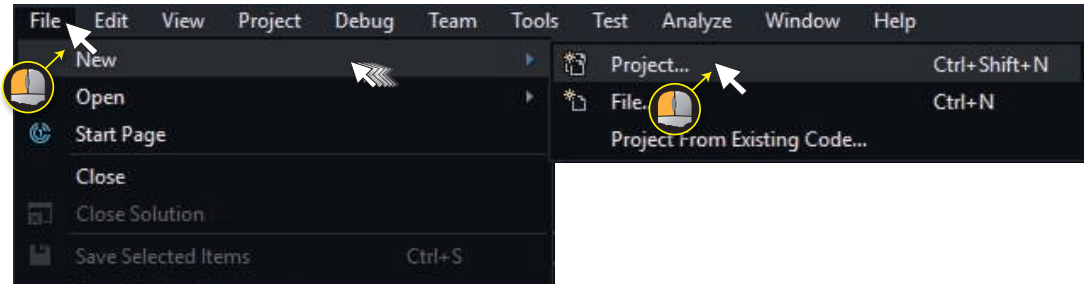


If this is a first project, then clicking on *Continue without code* (shown at the bottom-right of the window) will take us to the main screen.

A New Project

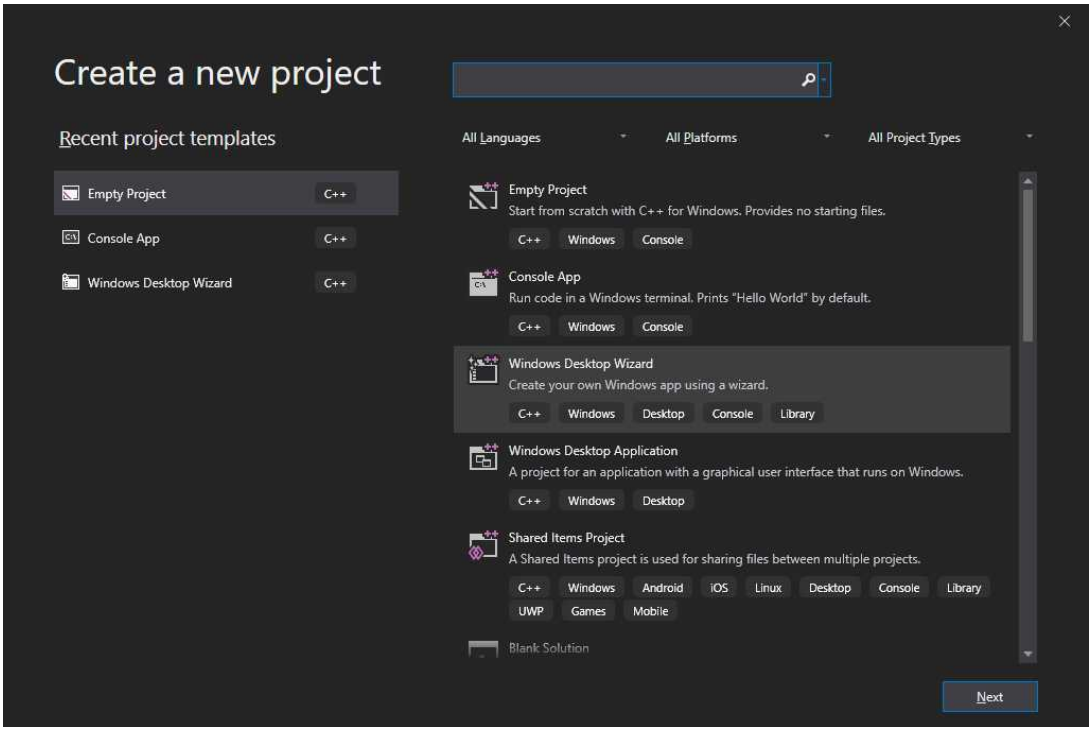
To begin a new C++ project in *Visual Studio Community 2019*, we need to select **File** from the main menu at the top of the IDE, then **New** and **Project** (see FIG-3.4).

FIG-3.4 New Project Option



This takes us to the next screen where we are presented with various options for our new project (see FIG-3.5).

FIG-3.5 Starting a New C++ Project



Since we will be creating basic C++ programs, rather than Windows-based ones, we can select either *Empty Project*, *Console App* or *Windows Desktop Wizard* from the list of options.

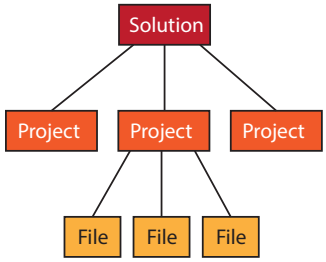
And although *Console App* will actually create the first few lines of our code, the best option here is *Windows Desktop Wizard* since it makes us do all of the hard work ourselves and offers us other options which we will make use of in later chapters.

You may have noticed the terms *project* and *solution* mentioned in some of the options given. It is best that we explain these terms before going further.

In handling various background tasks involved in creating a project, the IDE needs to produce other supporting files, so when we attempt to produce a single C++ program file, we actually end up creating several other files as well. Because of this, Visual C++ creates a new "project" rather than simply a new "program". The *project* is a setup containing the support files that are used to produce a completed C++ application program. Projects themselves are held in "solutions". And a solution may hold more than one project. The overall structure is visualised in FIG-3.6.

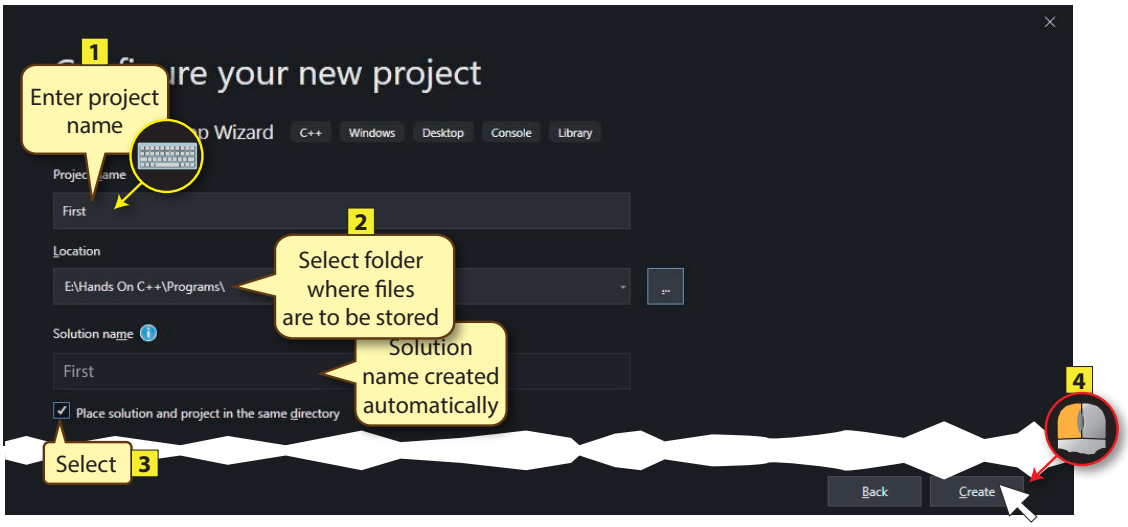
FIG-3.6

Visual Studio: Solutions, Projects and Files



If we take a closer look at the details we need to enter when creating a new project, we can see that the solution name is assumed to have the same name as the project and that a new folder will automatically be created for the solution and its contents (see FIG-3.7).

FIG-3.7 Solution and Project Details



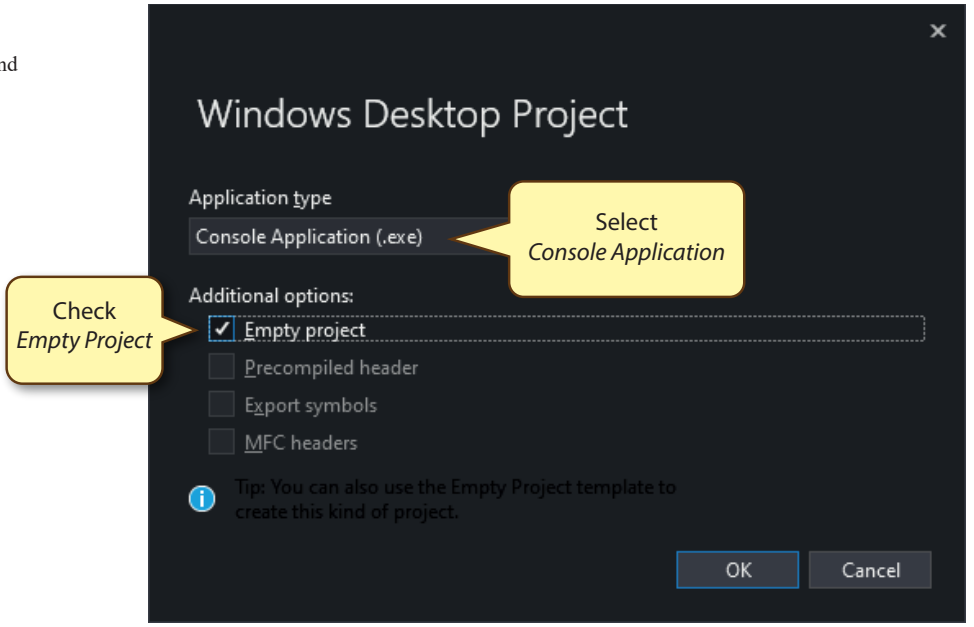
Although the IDE will automatically select a project name and folder for the new project, it is best if we enter our own details.

After clicking on the **Create** button, the *Windows Desktop Project* dialog box appears. This allows us to select what sort of project we wish to create. For the most part we will be creating **Console Applications**.

This means we want to create an empty project, so in the *Additional Options* section we need to check the *Empty Project* option (see FIG-3.8).

FIG-3.8

Application Type and Options



Activity 3.3

Using Windows Explorer, create a folder for your C++ projects (this text assumes that the folder created is *C:\Hands On C++\Programs*).

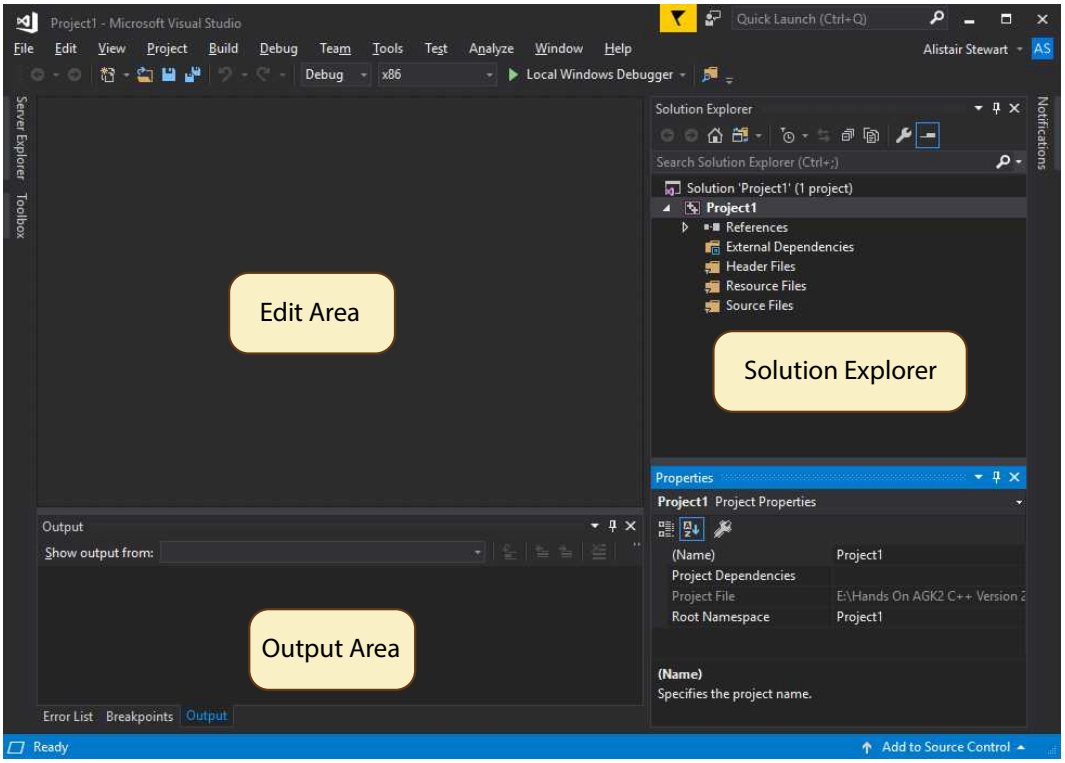
In Visual Studio start a new C++ project and select the *Windows Desktop Wizard*. Name the project and solution *First*.

For the *Location* value, browse to the *C:\Hands On C++\Programs* folder and select it. Now click on **Create**.

In the wizard dialog box, select **Console Application**, check the *Empty Project* checkbox.

Finally, we arrive at the main IDE screen (see FIG-3.9).

FIG-3.9 The Main IDE Window



There are three main areas in the IDE that are important to us:

- The **Edit area** which will be populated when we create a source code file.
- The **Solution Explorer** which details the contents of the current project.
- The **Output area** which displays messages produced by the compiler.

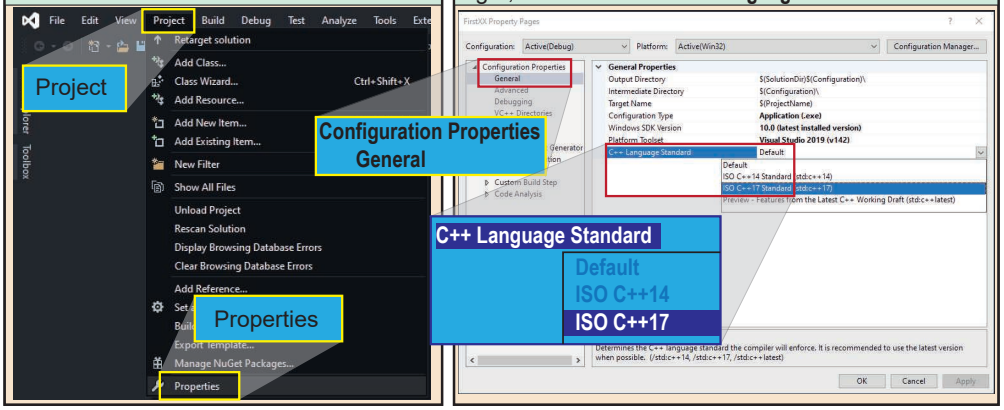
But before we begin creating the code for our new project, we need to make sure that we are using the C++17 version of the compiler – which is not the default setting!

The steps involved are shown in FIG-3.10. Note that this must be set every time we start a new project.

FIG-3.10 Language to C++17

From the main menu at the top of the IDE we must select **Project** and **Properties**.

In the dialog box that appears, we need to select **Configuration Properties > General** and then, on the right, under **C++ Standard Language** choose **C++17**.



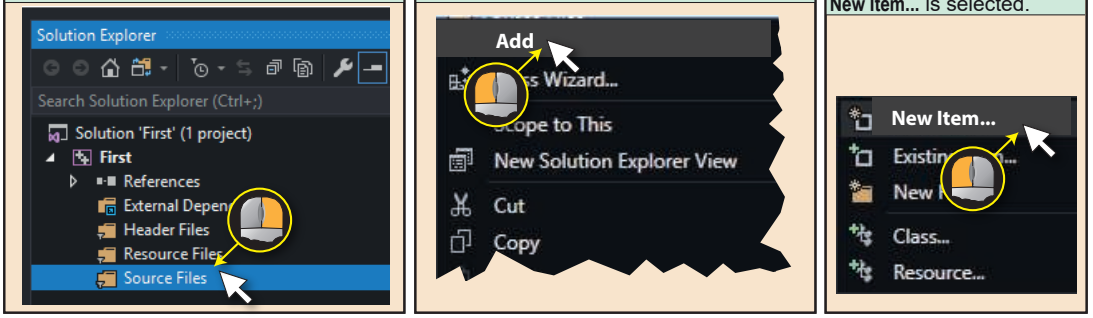
Now we need to add a C++ source file to the project. The steps involved in creating the file are shown in FIG-3.11.

FIG-3.11 Adding a Source Code File to a Project

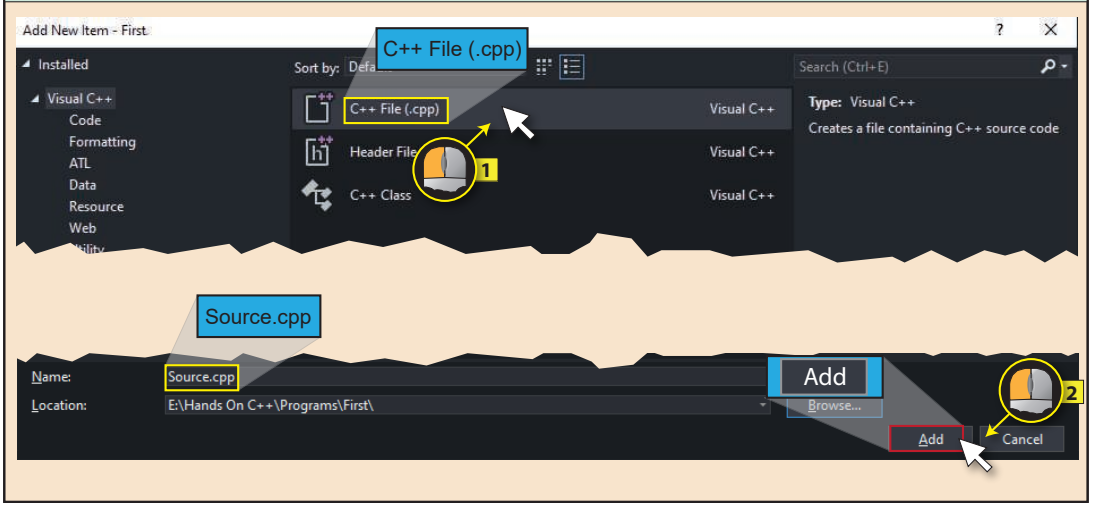
Right click on **Source Files** in *Solution Explorer...*

...to produce a pop-up menu. Move the mouse pointer over **Add** ...

...which then produces a second popup menu where **New Item...** is selected.



In the new dialog box, select **C++ File (.cpp)**. The file about to be created is automatically named *Source.cpp* and for most projects, that will be fine. Finally, we press the **Add** button to add the new file to the project.



This project continues in the next Activity.

Activity 3.4

Follow the steps given in FIG-3.11, add a source file named *Source.cpp* to your project.

Entering the Code

At last, we are ready to enter our program code in the main edit window. The code is shown in FIG-3.12.

FIG-3.12

Program Source Code

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Hello world\n";
   return 0;
}
```

An explanation of the code:

The *iostream* header file defines terms such as *cout* which are not part of the core language.

`#include <iostream>`	This is a command to the pre-compiler to include lines of code held in a file called *iostream* as part of the program. This is known as a **header file**.
`using namespace std;`	This line tells the compiler under which group name (*std*) to look for any unfamiliar terms used within the program code.
`int main()`	This is the first line of the actual program. The program consists of a single function called `main` which returns an integer value (`int`). The function takes no parameters - indicated by the empty parentheses `()`.
`{`	This marks the beginning of the function's code.
`cout << "Hello world\n";`	This is a command to output (`cout <<`) the phrase `Hello world` and then move the cursor to a new line (`\n`).
`return 0;`	Zero is the value to be returned by the function. This value is made available to the operating system.
`}`	This marks the end of the function's code.

We need not concern ourselves with a deeper understanding of the code at this point – that will come later.

This project continues in the next Activity.

Activity 3.5

Click in the edit area of the IDE and enter the code shown in FIG-3.12.

Make sure the characters are in the correct case (usually lower) and that the spacing matches that shown. Take time to make sure you've typed it correctly.

Compiling the Code

The next stage is to have the compiler check that the instructions we have typed in are acceptable. The compiler is a hard task master, demanding that everything should be exactly correct. It will complain bitterly if we misspell a word, use the wrong symbol, or omit part of a line. This type of mistake is known as a **syntax error** since it represents a mistake in the grammar of a statement or of the program as a whole.

Those who are new to programming are almost certain to make such a mistake. As we become used to the structure of the language's statements we'll make less errors - but we'll never eliminate them completely. Luckily, it doesn't matter too much if we make these slight mistakes, because the compiler will point out exactly what the problem is when it has a look at our efforts - but, unfortunately, it won't do the corrections for us!

When we eventually get all the statements in the program correct, the compiler will translate our C++ instructions (source code) into object code. This is the code that the machine will use when executing our program. When we are creating standard C++ programs (as we are here) that object code will, in fact, be machine code - the binary language of the processor that lies at the heart of the computer.

When creating a GUI Windows-based program using the preferred .NET environment, then the object code is something called **Common Language Runtime**. We will not be covering this aspect of Visual C++ in this book.

In the next Activity we're going to deliberately create an error in our program just to see how the compiler reacts and what sort of error message we get under these circumstances.



You can also use the Ctrl+Shift+B key to build your program.

Activity 3.6

Edit your program's code, removing the semicolon (`;`) at the end of the line which starts with the term `cout`.

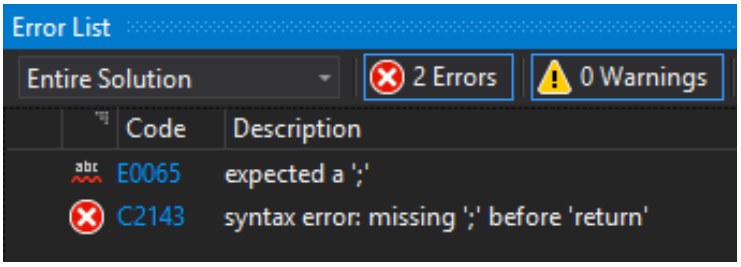
To have the compiler attempt to convert your program to machine code, select **Build|Build Solution** from the main menu.

What result is obtained?

The compiler opens a window titled *Error List* to report on the success - or otherwise - of its attempt to compile the program. In this case, an error is reported (see FIG-3.13).

FIG-3.13

Compiler Output Error Message



The first line of text under the heading *Description* is

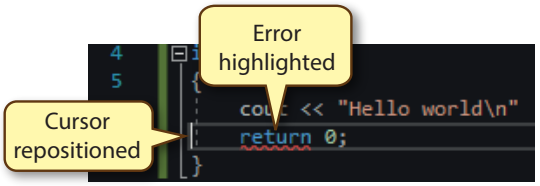
C2143: syntax error : missing ';' before 'return'

which tells us that the compiler has detected the missing semicolon. At the extreme right of this error message (not shown above), we will also see the number of the line on which the error was detected.

By double-clicking on the *Output* window's error message line, the cursor will move to the line containing the error but the problem code is likely to already have a red underline by the area containing the problem (see FIG-3.14).

FIG-3.14

Error Highlighting



Sometimes the compiler isn't quite on the mark when it tries to point out where the errors are. Here, for example, it's indicating the line after the one with the omitted semicolon.

Activity 3.7

Double-click anywhere on the error message in the *Errors List* window and check that the compiler attempts to highlight the position of the error within the edit window.

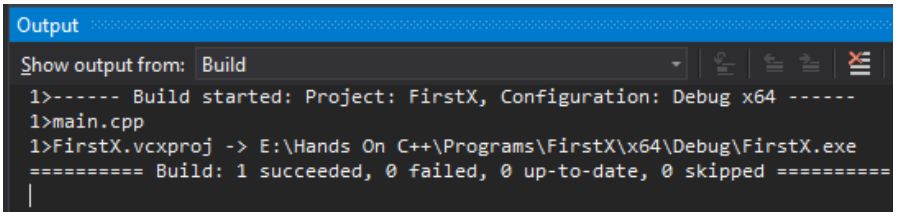
Re-insert the semicolon in your code, close the message window created by the compiler and attempt to compile your program again.

Now the program should have compiled successfully (unless you came up with a few errors of your own which you'll need to correct).

Again, the compiler creates an *Output* window to tell us how things went, but this time it should be similar to the one shown in FIG-3.15.

FIG-3.15

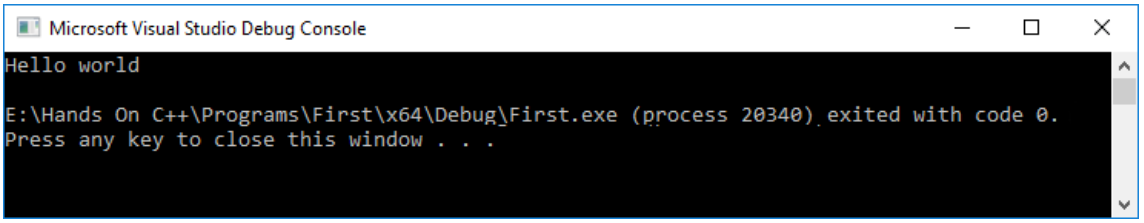
Compiler Output Successful Compilation



Running the Program

With the code successfully compiled, all that is left to do now is to run the program and observe the output it produces (see FIG-3.16).

FIG-3.16 Output Produced by Project First



Activity 3.8

Select **Debug|Start Without Debugging** to run your program.

You should see a command line window containing the program's output, *Hello world*, as well as the additional line *Press any key to continue ...*

Press any key to close down the command line window. Close the *Output* window.

A Second Program

Let's have a look at a second, slightly longer, program. In this one, we'll set up variables where we can store values entered from the keyboard, calculate the sum of the values entered and, finally, display the result of the calculation on the screen. The program code is shown in FIG-3.17.

FIG-3.17

Program Listing

```
// *** Program to add two numbers ***
#include <iostream>
using namespace std;
int main()
{
   int no1, no2, total;

   //*** Read values from keyboard ***
   cout << "Enter first number :";
   cin >> no1;
   cout << "Enter second number :";
   cin >> no2;
   //*** Calculate total ***
   total = no1 + no2;
   //*** Display result ***
   cout << "The sum of " << no1 << " and " << no2 << " is " <<
   ↳total << endl;
}
```

An explanation of new program elements:

`//*** Program to add two numbers ***`

The double slashes represent the start of a comment. The remaining characters on that line are ignored by the compiler. Comments are added solely for the benefit of any humans that happen to be reading the code. As a general rule they are added to explain the purpose of a whole program (as here) or parts of a program (as with the remaining comments in this example). Asterisks are added simply to make the lines of comment stand out.

`int no1, no2, total;`

This line declares the variables used within the program. The term `int` tells us that the variables are to store integer (whole number) values. The remaining text gives us the names of the variables (*no1*, *no2*, and *total*).

`cin >> no1;` This is a command to accept a value typed in at the keyboard (`cin >>`) and to store that value in the variable called *no1*.
Notice that the `cin` statement is preceded by a `cout` statement which displays a message telling the program user what they are expected to do. Such a statement is known as a **prompt**.

`cin >> no2;` This is a command to accept a second value from the keyboard and store the result in *no2*.

`total = no1 + no2;` This statement instructs the computer to calculate the sum of the values in *no1* and *no2* and to store the result in *total*.

```
cout<<"The sum of "<<no1<<" and "<<no2<<" is "<<total<<endl;
```

This is a rather complex output statement which starts by outputting the phrase *The sum of* followed by the contents of *no1* then the word *and*, the contents of *no2*, the word *is* and the contents of *total* before finally moving the cursor to a new line (`endl`). Notice the need to use the symbol `<<` before each item to be output.

No *return* Statement

Notice that the latest program does not end with a `return 0;` statement. Although all other functions which `return` a value must include a return statement, `main()` is a exception to the rule since the C++ compiler will automatically insert the necessary machine code required to return zero into the compiled program's object code.

Whitespaces

It is worth pointing out that, although the compiler demands strict adherence to syntax, it's a lot more forgiving when it comes to adding things like space, tab and return characters (collectively known as **whitespace** characters since they all add blank gaps to a program).

Often we can insert or omit these characters at will between the various terms within our program. So, for example, the compiler is equally happy with the two layouts shown below:

```
total      =     no1 + no2     ;
```

and

```
total=no1+no2;
```

All the compiler needs to know is where one term (also known as a **token**) ends and another begins. The above statement contains six tokens. These are `total`, `=`, `no1`, `+`, `no2` and `;`. Even without the spaces the compiler can identify each of these since variable names can only consist of letters or numbers.

But there are situations where we cannot go adding whitespace characters when the urge comes upon us. For example, we mustn't have spaces in the middle of a term. Split variable names or keywords as in

```
tot   al
```

or

```
c  in
```

would not be recognised by the compiler and would give rise to errors.

On other occasions spaces must be included to help the compiler distinguish between one token and the next. Hence, writing

```
intmain()
```

would be invalid since the compiler would not be able to recognise `int` and `main` as two separate tokens. As a general rule, if two adjacent tokens consist of only letters or numbers, then they must be separated by at least one space character. And to aid readability by a human eye, it is best to add a space between each token.

Activity 3.9

Start a new C++ project (**File|New|Project**). Call the project *InputOutput* and store it in the same folder as before (*C:\HandsOnC++\Programs).*

The project is to be set as *Console Application* with the *Empty Project* option set..

Type in the coding given in FIG-3.17. Take particular care to include spaces at the correct points within the strings given in the final `cout` statement. There should be a space after the word *of* in the first string, a space either side of the word *and* in the second string and either side of the word *is* in the third string.

Build the program (*Ctrl+Shift+B*), eliminating any syntax errors you may have introduced.

Run the program (*Ctrl+F5*). When requested to enter the first number, type in 13 and press the **Enter** key.

When requested to enter the second number, type in 7 and press **Enter**. Examine the output produced.

Run the program again. This time, enter two integer values of your own choice (nothing too extreme!)

Activity 3.10

a) If a semicolon is omitted from a line of code, what type of error is produced?
b) What happens during the compilation process?
c) What output is created when a program is compiled?
d) What term is used when declaring an integer variable in C++?

Summary

- Visual C++ offers features beyond standard C++17.
- Microsoft's Visual Studio Community 2019 package offers an Integrated Development Environment (IDE) which handles all aspects of program creation and execution.
- To create a standard, console C++ project:

Select **File|New|Project**
In the dialog box, under *Visual C++*:
Click **Windows Desktop**
Click **Windows Desktop Wizard**
Type in a name for the program file
Select the folder in which the project is to be created
Make sure *Create directory for solution* is checked
Click **OK**
In the *Windows Desktop Project* dialog:
Under Application type:
Set the application type to **Console Application**
Under *Additional Options*:
Check **Empty Project**
Uncheck **Security Development Lifecycle**
Click **OK**

- To create the file containing the source code:

In the *Solution Explorer*:
Right click over the project name
In the pop-up menu:
Select **Add|New Item**
In the *Add New Item* dialog box:
Select **C++ File (cpp)**
Enter the filename (suggest *main*)
Click **Add**

- To enter program code:

Type the code within the edit window.

- To compile the code:

Select **Build|Build Solution** (shortcut *Ctrl+Shift+B*)
If there are errors,
Click on the error message to highlight line containing error
Edit code

- To execute the program:

Select **Debug|Start Debugging** (*F5*)
(command line window closes at end of program run)
or **Debug|Start Without Debugging** (*Ctrl+F5*)
(command line windows stays open till key press)

- Whitespace characters are any characters that create space on or between lines. For example, space, tab and return characters.
- Whitespace can only be added or removed from code where these changes do not confuse the compiler.

Solution

Activity 3.1

No solution required.

Activity 3.2

No solution required.

Activity 3.3

Create your new folder.
Start up Microsoft Visual Studio Community 2019.
Select **File|New|Project**
Under *Visual C++*, select *Windows Desktop*
On the right, select *Windows Desktop Wizard*
In the **File** edit box, type *First*
In the **Location** combobox select
Hands On C++/Programs
Make sure *Create Directory for Solution* is checked
Click **OK**
In the *Windows Desktop Project* dialog box:
In *Application Type* select *Console Application.*
Check *Empty Project*
Uncheck *Security Development Lifecycle.*
Click **OK**

Details of the folders created will appear in the *Solution Explorer*

Activity 3.4

In *Solution Explorer,* right click on *Source Files*.
In the pop-up menu select **Add** then **New Item**
In the window created, select *C++ File (cpp)*
In the **File** edit box, enter *main*
Click **Add**

Activity 3.5

Click in the Edit Area.
Type in the program code

Activity 3.6

Remove the semicolon (;) at the end of the line beginning with the term `cout`
Select **Build|Build Solution** (or press *Ctrl+Shift+B*)
The compiler will create an output window specifying that an error has been detected in the code

Activity 3.7

Move the cursor over the error message in the *Output* window
Click on the error message
In the edit area, the line of code containing the error will be highlighted
In the case of missing semicolons, it is the line after the faulty line which is indicated

Add back the semicolon

Build the program (*Ctrl+Shift+B*)

The *Output* window will show that the compilation was successful.

Activity 3.8

Press *Ctrl+F5* to run the program
The window will contain the text *Hello world* and *Press any key to continue . . .*
Press a key to close the command window

Activity 3.9

Select **File|New|Project**
Under *Visual C++*, select *Windows Desktop*
On the right, select *Windows Desktop Wizard*
In the **File** edit box, type *First*
In the **Location** combobox select
Hands On C++/Programs
Make sure *Create Directory for Solution* is checked
Click **OK**
In the *Windows Desktop Project* dialog box:
In *Application Type* select *Console Application.*
Check *Empty Project*
Uncheck *Security Development Lifecycle.*
Click **OK**

In *Solution Explorer,* right click on *Source Files*.
In the pop-up menu select **Add|New Item**
In the window created, select *C++ File (cpp)*
In the **File** edit box, enter *main*
Click **Add**

Click in the Edit Area.
Type in the program code.

Build the solution (*Ctrl+Shift+B*)

The *Output* window will show that the compilation was successful.

Press *Ctrl+F5* to run the program
The window will contain the text *Enter first number*

Type in 13 and press **Enter**

After the message, *Enter second number*, type the number 7 and press **Enter**

Press *Ctrl+F5* to run the program and enter two more numbers such as 32 and -7.

Activity 3.10

a) A missing semicolon will create a syntax error.
b) During compilation the source code (the C++ code) is first checked for syntax errors and then converted to the equivalent statements in machine code.
c) The compiler creates object code
d) `int`

4

Data

In this Chapter:

- ❑ Basic Data Types in C++
- ❑ Arithmetic Operators
- ❑ Assigning a Value to a Variable
- ❑ Constants and Named Constants
- ❑ String Constants
- ❑ Converting Values to Other Types
- ❑ Declaring Variables
- ❑ Escape Code Characters
- ❑ How Memory is Allocated to Variables
- ❑ Operator Priority
- ❑ Variable Declaration
- ❑ Variable Naming Conventions

Variables and Constants

Introduction

In the last chapter we saw an example of how variables are set up (declared) within a program. Declaration starts with the data type followed by a comma-separated list of the variables we wish to create. So the line

```
int no1, no2, total;
```

creates three variables each capable of storing a single integer value.

A **constant** (also known as a **literal**) is any fixed value used within a program. These can be numeric (eg. 4.14159, 12, or -7) or non-numeric (eg. *Hello world*).

Memory Allocation

It is probably worth getting some idea of how variables are assigned space within the computer's memory since this will make some aspects of C++ to be covered in later chapters a little clearer.

Every variable defined within a program is allocated its own set of one or more bytes in an area of memory known as the **stack**. The number of bytes allocated is determined by the variable's type. When a variable is assigned a value, that value is stored in the bytes allocated to that variable. And, since every byte in the computer's memory has its own unique address, we can state that every variable we use within a program has four characteristics:

- a name eg. no1
- a type eg. int
- a value eg. 12
- a starting memory address eg. 1D20 : 0F1A

Basic Data Types

If we want to set up variables which can hold numbers that include the decimal point (often termed **real** or **floating-point** numbers), then we use the term `float` to define the data type. For example, we can set up two real variables with the line:

```
float angle, discount;
```

To create a variable capable of storing a single character such as a letter of the alphabet or a symbol, the data type is `char`. For example:

```
char letter;
```

A final basic data type is Boolean. If you remember, back in Chapter 1 we came across the idea of the Boolean expression: statements that were either *true* or *false*. Boolean variables can only take on one of these two values and are used to record the result of Boolean expressions. We'll look at how we use Boolean variables in later chapters, but for now all we need to know is that Boolean variables are declared using the term `bool` as in the line:

```
bool can_drive;
```

We can declare a `char` variable to store a single character, but what if we want to store more than one character in a variable? Perhaps we want to store a message such as

"*Please enter your email address*" in a single variable. How is this to be done? Well, it can be done, of course, but C++ doesn't have a fundamental data type for storing anything more than a single character, so we'll have to do without this type of variable for the moment.

To summarise, we now have four terms we can use when declaring variables : `int`, `float`, `char` and `bool`.

A program can declare as many variables of each type as it requires. Below we see declarations for several variables of each type:

```
int no1, no2;
float x, y, z;
char gender, response;
bool over21, eligible;
```

Data Type Variations

If a variable is designed to hold a numeric value, then the more bytes it has been allocated the larger the number and the more decimal places that can be stored there. On the other hand, it's wasteful of space if we allocate a variable more bytes than it's ever going to need.

To deal with this problem, C++ offers us several more options when declaring variables designed to hold integer values. Each option potentially allocates a different number of bytes to the variable allowing us to pick the option most suited to the range of values we expect our variable to store.

In fact, the number of bytes allocated to the different numeric types is at the discretion of the compiler designers and may vary from compiler to compiler. The storage assigned by Microsoft's Visual C++ is shown in FIG-4.1.

FIG-4.1

Standard Integer Types

Type	Bytes	Range
short	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647
long long	8	$-(2^{63})$ to $2^{63} - 1$



The `long long` type is guaranteed to be at least 64 bits in length on any compiler.

So, if we knew a variable was to contain a numeric value representing a month of the year (so, between 1 and 12) then we might declare it as

```
short month;
```

thereby minimising the number of bytes the variable is assigned.

Notice that `int` and `long` are actually assigned the same number of bytes. In practice this means there is no advantage in declaring a variable as type `long` rather than type `int`. However, since different compilers are free to assign a different number of bytes to each type, if we are using a compiler other than Microsoft Visual C++, check out the number of bytes allocated to each type before deciding which to use.

When declaring `short`, `long` or `long long` variables, it is permissible to include the term `int` just to remind ourselves that these are integer types that are being declared. Hence we can declare variables *a*, *b* and *c* using either

```
short a;
long b;
long long c;
```

or

```
short int a;
long int b;
long long int c;
```

Another option that can be used to optimise the use of the space allocated to an integer variable is to declare the variable as `unsigned`.

When a variable is declared as unsigned, it can no longer store negative values, but the upper limit on positive values is doubled. Unsigned options are shown in FIG-4.2.

FIG-4.2

Unsigned Integer Types

Type	Bytes	Range
unsigned short	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
unsigned long	4	0 to 4,294,967,295
unsigned long long	8	0 to 18,446,744,073,709,551,615

Again we have the option to include `int` when using the terms `unsigned short`, `unsigned long` and `unsigned long long`.

Since we can declare a variable as unsigned, we can also add the optional term `signed` to any of the integer types capable of storing negative values.

All of these optional parts to a declaration mean that a signed 16-bit integer variable can be declared using any one of the following:

```
short
signed short
short int
signed short int
```

It is probably best to go for the longest version since it gives the clearest description of what we are declaring.

Real number types also have some options allowing the number of bytes allocated to be varied. This, in turn, affects the number of decimal places to which a value can be accurately stored. The various options are shown in FIG-4.3.

FIG-4.3

Float Types

Type	Bytes	Range and Accuracy
float	4	$\pm 3.4 \times 10^{+38}$ (about 7 digits)
double	8	$\pm 1.7 \times 10^{+308}$ (about 15 digits)
long double	8	$\pm 1.7 \times 10^{+308}$ (about 15 digits)

Originally, characters were coded using a system known as American Standard Code for Information Interchange (ASCII, for short). Each character occupied a single byte. This was fine if all we needed was a Latin-based character set but was too limited to code other character sets such as Japanese Kanji.

To deal with this shortcoming, a new coding system called **Unicode Transformation**

Format (UTF) was created. This defined codes for every character used by the main languages of the world. However, this coding system needed more than a single byte to represent such a wide range of characters. UTF-32 used four bytes to encode each character, UTF-16 used two bytes while UTF-8 used one, two or four bytes depending on the character being stored. UTF-8 also ensured backward compatibility with ASCII when coding single-byte characters.

C++ offers several basic storage options for characters. For a program storing Latin-based ASCII characters we can use type `char` which assigns a single byte to every character.

A second type (`wchar_t`) uses two bytes. How the character is encoded within those two bytes is at the discretion of the compiler designers, but a limited set of Unicode UTF-16 is the most common.

The full list of character options is shown in FIG-4.4.

FIG-4.4

Character Types

Type	Bytes	Range
signed char	1	character OR integer -127 to 128
unsigned char	1	character OR integer 0 to 255
char	1	defaults to same as `signed char` but can be `unsigned char`
unsigned wchar_t	2	Unicode character OR integer 0 to 65,535
signed wchar_t	2	Unicode character OR integer -32,768 to 32,767
wchar_t	2	defaults to same as `unsigned wchar_t` but can be `signed wchar_t`

As a general rule, the examples given in this text will not make use of two-byte characters other than to show any difference in how they are handled in various situations.

The reason we have some apparently strange options when declaring character variables is because they can, on occasion, be used to store integer values. In such cases we would declare our variables to be either `signed char` or `unsigned char` as appropriate for our needs. In days when main memory space was limited and processors were relatively slow there may have been good reason to make use of this feature. However, it is probably best avoided now. Types `char` and `wchar_t` should be used only to store characters and not integer values.

To accommodate 16 and 32 bit Unicode options, C++ also has character types

```
char16_t
char32_t
```

We are unlikely to require these for Latin-based coding.

Finally, we have the single option for the Boolean variable type (see FIG-4.5).

FIG-4.5

Boolean Type

Type	Bytes	Range
bool	1	*true* or *false*

Remember, it is not the words *true* or *false* that are stored in variables of this type but one of two possible binary values. One of these values represents *true*, the other *false*.

Activity 4.1

A program contains three `long` variables, as well as one `int` and 2 `wchar_t` variables. What is the total number of bytes allocated?

Naming Variables

Once we've decided what type of variable we require, we then need to assign our new variable a name. A principle rule of good programming is that we use meaningful variable names, reflecting the purpose of the value held. For example, names such as *tax*, *wage*, *hours* would be good names when used appropriately. On the other hand, using names such as *a*, *b* or *c* would be unhelpful unless used to represent values in a purely mathematical equation.

Any name we create for use within a program is known as an **identifier**.

C++ itself imposes other restrictions on exactly what term can be used as an identifier. The name must start with a letter or the underscore character and, if there are more characters in the name, these must be letters, numeric digits, or underscore characters. A name cannot contain spaces nor can it be a C++ keyword such as `if` or `for`. From this set of rules we can see that the following terms are valid identifiers:

```
salary
no1
gross_salary
```

but the next set of terms are not:

`net salary`	contains a space
`2ndvalue`	does not start with a letter or underscore
`for`	keyword

Letters can be upper or lower case, but C++ is case-sensitive. This means it treats the upper and lower case of a character as two different symbols. For example, it would treat the terms *Sum* and *sum* as two different identifiers. Although it would hardly be good practice to use two such similar terms deliberately, a mistype can sometimes give rise to this situation.

Activity 4.2

Which of the following variable names are invalid?

a)	x1	b)	first_ans	c)	pic@2
d)	_result	e)	long	f)	final total

Although option *d)* in the Activity above is a valid name, starting variable names with an underscore isn't a good idea since compilers often make use of the leading underscore when naming constants and system variables. For the same reason, names beginning with two underscores should also be avoided.

Declaring Variables

We already know that we declare variables by starting with the type of value that is to be stored in the variable followed by the variable name and ending with a semicolon. This means that statements such as

```
int no1;
unsigned short int no2;
wchar_t letter;
```

are all valid declarations which the compiler would not object to.

We are also free to list several variables in a single declaration as long as they are all of the same type and have their names separated by commas:

```
float net_salary, tax, gross_salary;
```

However, the above approach is not generally considered to be good style since it is easy for any humans reading existing code to miss the declaration of multiple variables. Also by declaring one variable per line, we have the option to add a descriptive comment to a variable. For example,

```
int score;  //Score achieved in current game
```

Perhaps the one time multiple declarations in a single line are acceptable is when the variables are closely related as in the line

```
int hour, minute, second;  //Current time
```

A program may have as many variable declarations as required, but the same variable cannot normally be declared twice (the compiler would flag this as an error):

```
int no;
float no;    //Invalid, variable of this name already exists
```

Although programmers often declare variables at the start of their code (immediately after the opening brace), C++ allows declarations to be made at almost any point in a program a long as the variable is declared before it is used elsewhere in the code.

Assigning an Initial Value

When a program allocates space for a declared variable, it doesn't bother to tidy out that space. So if there is a mess of 1s and 0s lying about in the allocated bytes, they will still be there once allocation is complete. In effect, this means that every variable starts off life containing some unknown value.

Of course, often this won't be a problem since the program will assign a value a few lines later on in the code, but there are times when we'd like to be sure of just exactly what value a variable contains when it has been newly created and, in any case, it is considered good practice to assign an initial value when declaring any variable. To do this we can assign an initial value to a variable as part of the declaration statement.

In earlier versions of C++, variables were normally initialised using the equals symbol as in the line:

```
int no1 = 0;
```

This makes sure that the variable *no1* contains the value zero when it is first created.

Rather than use the equals sign, the start-up value can also be given within a set of parentheses:

```
int no1(0);
```

This method of initialising a variable is known as **constructor initialisation**.

A final initialisation method, added by the C++11 language definition and now the preferred approach, is known as **uniform initialisation** and uses braces rather than parentheses:

```
int no1{0};
```

When initialising numeric variables, we can also use empty braces to signify an initial value of zero, as in the lines:

```
int no1{};
float no2{};
```

Boolean (`bool`) variables can be initialised using the one of the keywords, `true` or `false` as in the lines:

```
bool isOver21 {true};
bool isOK {false};
```

Empty braces assign false to Boolean variables:

```
bool isOK{};   //Assigns false
```

Variables of type `char`, can be assigned a single character which must be enclosed in single quotes:

```
char letter {'A'};
```

When empty braces are used to initialise a `char` variable, the variable is assigned the **null** character:

```
char letter{};
```

The null character has a binary value of 00000000_2 and is not displayable.

Using *auto*

When a variable is declared and also assigned an initial value, we can replace the variable's type keyword with the word `auto`. For example, rather than write

```
int no1{0};
```

we can write

```
auto no1{0};
```

This works because C++ can infer the *no1*'s type from the value being assigned (in this case, an `int` value). This ability is termed **type inference**.

Note that `auto` cannot be used when no initial value is given or empty braces are used since there is then no way for C++ to determine the variable's type. So, the lines

```
auto no1;
auto no2{};
```

would both be flagged as errors.

The Declaration Statement Format

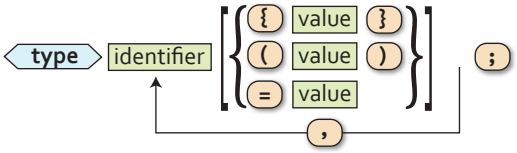
As we saw when creating our first two programs in the previous chapter, the compiler demands that statements within the program conform to an exact format and the declaration statement is no different. It has a definite format that must be complied with.

The easiest way to describe the format (or syntax) of a statement is to use a **syntax diagram**. Although they can look a bit strange to begin with, once you've learned how to read them, they really are quite a neat way of describing the structure of a statement.

In FIG-4.6 we see the syntax diagram for the declaration statement.

FIG-4.6

Declaration Statement Syntax



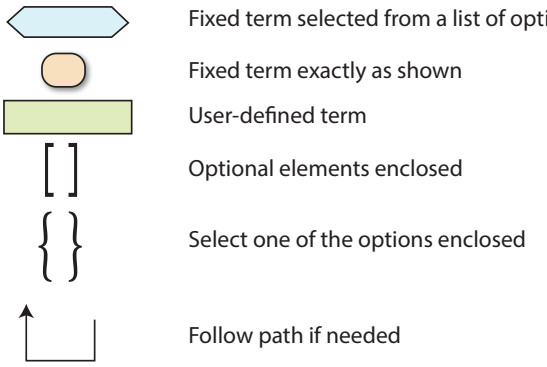
The diagram tells us that we must start with a specific data type (data type options have been described in previous pages) followed by an identifier name.

The next section, which assigns a starting value to the variable, is optional (indicated by the square brackets) and may be omitted from the statement. If an assignment part is to be included, one of three options must be selected (where a choice of options is required, the options are listed within braces).

Another variable can be defined by adding a comma and going back to add in another identifier and its optional start value. By making further use of this loop-back, any number of variables can be included, but eventually the statement must finish with a semicolon. The elements that make up the syntax diagram are defined in FIG-4.7.

FIG-4.7

Elements in a Syntax Diagram



Let's look at a statement and use the syntax diagram to check if it is correct. The statement is:

```
unsigned short no1, no2=4, no3(-2);
```

To check if a variable declaration statement is correct, all we need to do is travel from left to right through the syntax diagram in FIG-4.6, using up the tokens in the statement as we go.

By the time we reach the end of the diagram we need to have used up all the tokens in the statement we are checking. The steps involved are shown in FIG-4.8.

FIG-4.8

Using a Syntax Diagram

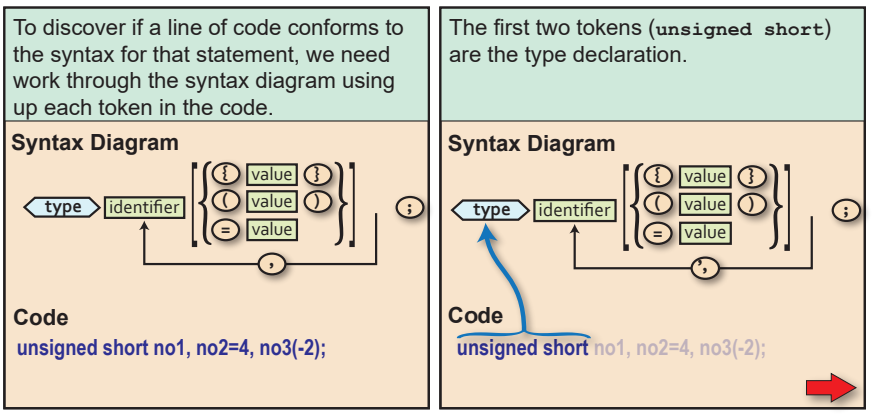
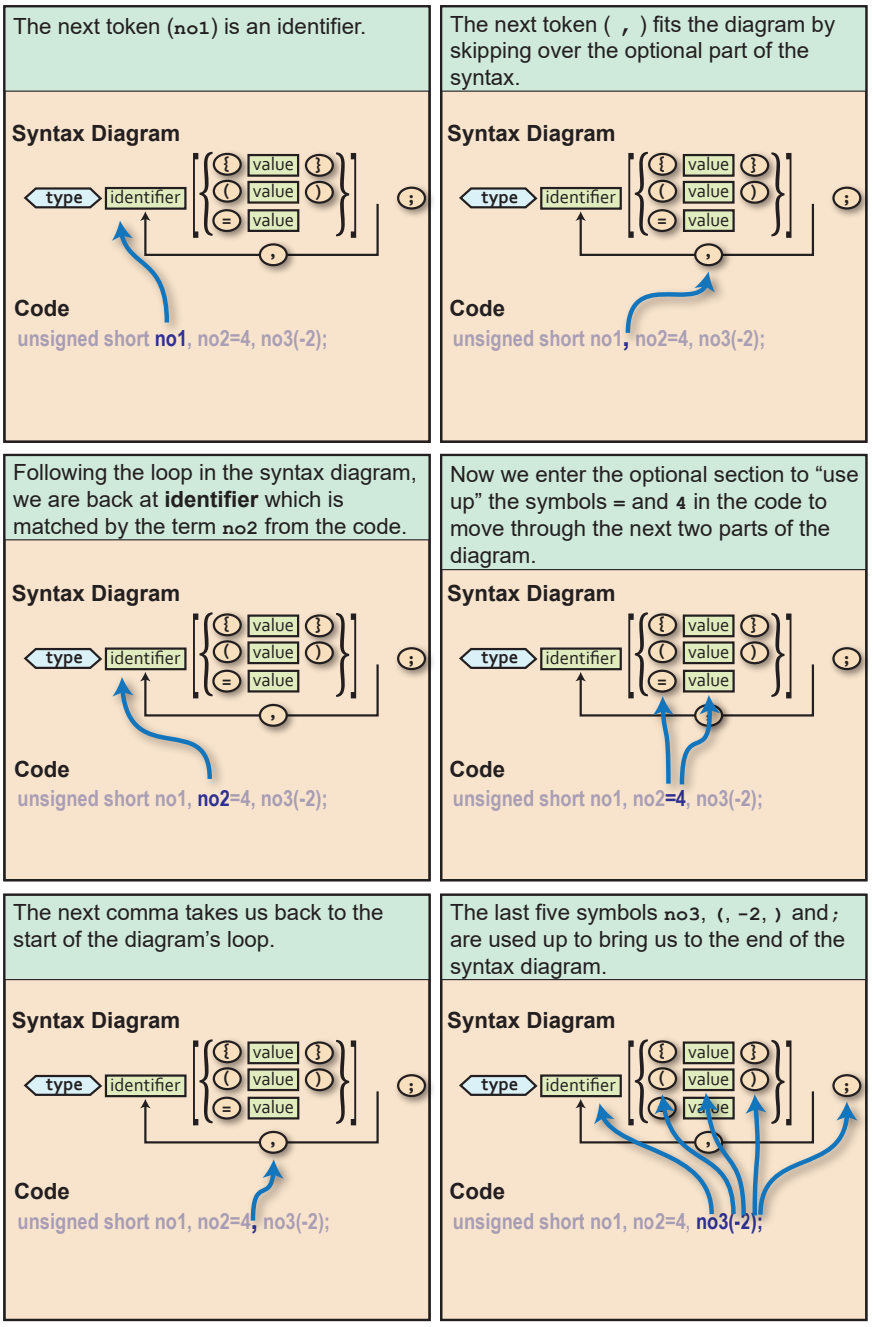


FIG-4.8
(continued)

Using a Syntax Diagram



Since the statement used in the example above fits exactly to the diagram, we know it is a valid statement which the compiler will accept without complaint.

The syntax diagram doesn't tell the whole story of what is acceptable to the compiler, since other things are checked such as ensuring that the type of value being assigned as an initial value is appropriate for the type of variable that is being declared. For example, it would not be appropriate to try assigning a character to an `int` variable.

Activity 4.3

Identify each of the following declarations as valid or invalid. Give a reason for any statement being invalid.

a) `int w;`
b) `short a=6,b(3)`
c) `long t1, t2(0),,t3;`
d) `unsigned a,b,c;`
e) `unsigned long int time-worked = 0;`
f) `long int w(0),x(1),y{-2};`

Constants

Constants (or **literals**) is the name we use for any fixed value mentioned in a program. For example, the statements

```
long int w{0}, x{1}, y{-2};
double no1{14.6};
char symbol{'?'};
```

contain the numeric literals 0, 1, -2, and 14.6 and the character literal ?.

Just like variables, literals are allocated space within the computer's memory. And, like variables they are stored in a specific format. These are just the same format types as used for variables, `int`, `short`, `char`, etc. However, whereas we must explicitly state the type of a variable in its declaration statement, the compiler will automatically assign a type to any literals used within a program.

Constant Types

Integer constants used in a program are automatically allocated the type `int` while real values such as `3.14159` are assumed to be of the type `double`.



It is best not to use lowercase Ls since these are easily confused for a 1 (one).

However, it is possible to force constants to be stored in a different format by adding the appropriate letter at the end of the constant. So by ending an integer value with an `L` (a lower case `l` can also be used) we can force it to be stored in `long int` format (e.g. `12L`); ending with a `U` (or `u`) will result in `unsigned int` format being used (e.g. `4U`); and using the combined `UL` (`ul`, `LU`, or `lu`) ending stores the value in `unsigned long` format (eg. `9UL`). `LL` can be used for `long long` constants and `ULL` for `unsigned long long` constants.

In the case of the Visual C++ compiler, as it uses the same allocation for `int` and `long` values there is little reason to convert from `int` to `long`. Of course, other compilers may allocate different amounts of space making the conversion more useful.

Integer literals are usually given in base 10 format, but it is possible to specify a value in **hexadecimal** by starting the value with a zero followed by the letter X . This gives us the option of writing either



A lowercase x can also be used when specifying a hexadecimal constant.

`255` or `0XFF`

to represent the number 255 in a program. The only reason for wanting to use the second version is that it is a whole lot easier to calculate which bits of a byte will be set to 1 in the space allocated to the number when using hexadecimal. Just why such knowledge might be useful, we'll leave to a later chapter.

As with standard numbers, a hexadecimal value will default to being stored in `int` format but we can use any of the endings described earlier to force the storage used to be of a different integer format.

Integer values can also be given in **octal** format by starting the number with a leading zero as in 017. Just remember not to start decimal values with a zero or they'll be assumed to be octal. And, of course, we can't use the digits 8 or 9 in what is meant to be an octal value.

More recent versions of C++ added the option to specify integer constants in **binary** by adding 0b or 0B to the start of the value. For example,

```
0b01100001
```

When writing out long numbers, English-speaking countries (and others) often use a comma to help identify the value. For instance, it's easier to see that 16,000,000,000 is 16 billion than when we write 16000000000. C++ also allows us to insert separators within an integer constant in order to make it easier for us humans to read the value. However, it's not a comma that is used, but a single quote. So, we might write our previous binary constant as

```
0b0110'0001
```

in order to easily see the contents of each nybble within a byte. The **single quote separator** is ignored by the compiler.

Real values can be held in `float` format by ending the value with an `F` (or `f`) (eg. `3.14159F`) or in `long double` format by ending with an `L` (or `l`) (eg. `3.14159L`).

Character constants are always shown enclosed in single quotes (known as **delimiters**) as in

```
char letter{'A'};
```

The quotes themselves are not stored as part of the constant but ar there only to help the compiler recognise the difference between a variable name and a character constant as shown in the example below:

```
char A, B;
B = A;          means store the contents of variable A in B
B = 'A';        means store the letter A in B
```

Single characters are always stored in `char` format unless they are preceded by a capital `L` in which case they are stored in `wchar_t` format. So, while the term `'X'` will store the letter X in `char` format, `L'X'` stores the letter X in `wchar_t` format.

Multiple character constants such as *Hello world* are always shown enclosed in double quote delimiters (e.g. `"Hello world"`) and are stored in **string** format. Since *string* isn't a basic data type, we'll delay any discussion about string variables until a later chapter.

There are only two Boolean constants; `true` and `false`. These are always stored in `bool` format.

Escape Codes

Certain characters cannot be enclosed in quotes. For example, how do we get the backspace character, which we can see on the keyboard, between a set of single quotes?

C++ solves this problem by having a special code for such troublesome characters. The code always begins with a backslash character (\) followed by a normal character of the alphabet. The backspace character, for example, is represented by the sequence `\b`. So to represent one of these special characters in our program we just enclose its code in single quotes (eg. `'\b'`).

The codes for this small group of special characters are known collectively as **escape codes** or **escape sequences** because the compiler has to escape from its normal method of interpretation to handle these codes. The full list of escape codes is shown in FIG-4.9.

FIG-4.9

Escape Codes

Code	Character
\a	beep sound
\b	backspace
\n	newline
\t	tab
\'	single quote
\"	double quote
\?	question mark
\\	backslash
\0	null



The null character escape code uses a zero - not the letter O.

Escape codes may also appear within strings - something we've already seen in the first program you ran back in Chapter 3. The string `"Hello world\n"` ends with a newline escape code (`\n`) which forces the screen cursor to move to a new line after outputting the words *Hello world*.

The **null** character is worth a mention since it is a character that has absolutely no effect on screen output. It won't show up as a space on the screen or produce any noises through your speakers. But it does play quite an important role in C++ and we'll see it turn up at various places throughout the book.

Hex-Defined Characters

Yet another way to represent any character (standard characters or escape codes) is to start with a backslash and follow this with the hexadecimal value of the character. For example, rather than write 'A' we could write '\x41' ; both represent a capital A (41 is the hexadecimal value of the ASCII code for a capital A). Note that there is no leading zero (as we have when specifying integer values in hexadecimal) and the x must be in lower case.

Activity 4.4

Which format are the following constants stored in:

a) `12` b) `4.8` c) `6.2F`
d) `'7'` e) `L'N'` f) `'\?'`

String Constants

While single character constants must be enclosed in single quotes, string constants must be enclosed in double quotes. For example,

```
"Hello world"
```

String constants may have any number of characters, including zero characters. A

string with zero characters is known as an **empty string** and is represented by a set of double quotes without any gap between them:

```
""
```

Strings can include escape characters as in the line

```
cout << "365 days\nin a year\ndivided into 12 months.";
```

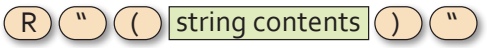
which would display as

```
365 days
in a year
divided into 12 months.
```

If we need to have what appears to be a control character treated as a normal sequence of two separate characters, then we start to define a raw string by adding the letter R immediately before the opening double quote. After the opening double quote we then need an opening parenthesis and a closing parenthesis immediately before the closing double quote. The format for creating a raw string is shown in FIG-4.10.

FIG-4.10

Raw String Constant Format



For example, the line

```
cout << R"(365 days\nin a year\ndivided into 12 months.)";
```

would display as

```
365 days\nin a year\ndivided into 12 months.
```

Actually, the opening and closing sequence at the start of a raw string can be a sequence of characters (as long as the opening sequence ends with a parenthesis and the closing one starts with a parenthesis). For example,

```
R"***(A valid raw string)***"
```

uses `***(` and `)***` as the opening and closing sequence. This would create the output

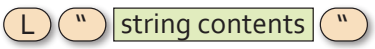
```
A valid raw string
```

This option is only required if we've managed to confuse the compiler with an unlikely sequence of double quotes and parentheses within the string we want to use.

While the individual characters in a string constant are normally stored in `char` format, we can force them to be stored in `wchar_t` format (creating what is termed a **wide string**) by preceding the opening double quote with the letter L (see FIG-4.11).

FIG-4.11

Wide String Constant Format



Using the prefix letter `u` (lowercase) will create a string held in 16-bit Unicode format while `U` (uppercase) creates a 32-bit Unicode string. For example,

```
u"This is stored in 16-bit Unicode"
U"This is stored in 32-bit Unicode"
```

Named Constants

Some programs make use of literals that have a very specific meaning. For example, a program might use the value 3.14159 (an approximation for π (pi)) or the value 212 (the boiling point of water in the old Fahrenheit scale). Rather than just have these number turn up unannounced half way through a program where others might struggle to realise their significance, we have the more organised option of setting

them up as **named constants**. A named constant is declared in the same way as a standard variable, but is preceded by the keyword `const`. Hence we could declare the two values mentioned above with the lines:

```
const double PI{3.14159};
const int BOILING_POINT{212};
```



The use of the underscore in the variable name `BOILING_POINT`

It's a widely used convention to use upper case characters when naming constants.

Although a named constant cannot be assigned a new value in later lines of a program, not all values set up as constants are forever fixed. For example, we might set up a named constant to represent the level of sales tax with the line

```
const double SALES_TAX {0.06};
```

and, if the level of tax changes in subsequent years, only that one line of the source code needs to be modified and the program recompiled to bring it up to date. Contrast this with the effort involved if you'd made reference to the sales tax level at a hundred different places in a program by using the actual value 0.06 – you'd have to find and modify each of those lines when the sales tax level was changed.

Activity 4.5

Write a line of code to declare a named constant called VOWEL which is assigned the value A.

The Assignment Statement

Now we know how to declare variables and the various data types available, it is time to have a look at the assignment statement which allows us to assign a value to a previously declared variable.

We already know that we can assign a value to a variable in the declaration statement, but if we are to change the value stored within a variable later in the program code, then the commonest way to do that is to use the assignment statement. This statement has a very simple syntax (see FIG-4.12).

FIG-4.12

The Assignment Statement Syntax



In the diagram:

identifier	is the name of a variable previously declared. This variable's contents will change to **value** after the statement has been executed.
value	is a constant, other variable name, or expression.

For example, if we had declared a variable using the line

```
int no1{2};
```



It's worth pointing out at this stage that the minus sign, as used above, is known as a **unary operator** since it operates on a single value, 5, changing it from positive value to a negative one.

we could change the value stored in *no1* from 2 to -5 using the assignment statement

```
no1 = -5;
```

Let's have a look at a few more simple assignment examples. Assuming the declarations

```
double x;
char letter;
```

```
bool over21;
```

then the following are all valid assignment statements

```
x = 12.09;
letter = 'X';
over21 = true;
```

The assignment statement can also be used to copy the contents of one variable to another variable. Hence, assuming the declaration

```
int no1{5},  no2;
```

we can copy the contents of *no1* into *no2* using the assignment statement

```
no2 = no1;
```

The contents of *no1* will not be changed by this statement.



Boolean expressions can be used in an assignment statement.

More about this later.

The final option for the assignment statement is for the value to the right of the equals sign to be an arithmetic expression. For example, after the line

```
int no1{13}, no2{7}, result;
```

we can calculate the sum of the contents of *no1* and *no2* and store that answer in *result* using the line

```
result = no1 + no2;
```



All of these arithmetic symbols are known as **binary operators** since they need two values to operate on.

Here we've used the plus sign (+) to indicate that we wish the values to be added together. Other basic arithmetic operators are shown in FIG-4.13.

FIG-4.13

Arithmetic Operators

Operator	Meaning
+	add
-	subtract
*	multiply
/	divide
%	remainder

Using the multiple operator, we could calculate the product of *no1* and *no2* with the line.

```
result = no1 * no2;
```

Arithmetic expressions can contain more than one arithmetic operator and involve literals and/or named constants where required. For example, in calculating the area of a circle, a program might contain the lines

```
const double PI{3.14159265};
double radius, area;
radius = 5.3;
area = PI * radius * radius;
```

Activity 4.6

Write lines of code to perform the following:

a) Create a named constant, *DAYS_IN_YEAR,* and assign it the value 365.
b) Create a variable called *totalDays* which is assigned the total number of days in four consecutive years (allow for a leap year).
c) Create two real variables called *side* and *area*. The variable *side* should start with the value 7 and the contents of *area* should be calculated as *side* multiplied by *side*.

The Remainder Operator

In addition to the regular arithmetic operators, C++ offers a remainder operator using the percent (`%`) symbol. This is used with integer values only to determine the remainder after division has taken place. For example, the expression `9%4` would give the result 1 since 1 remains after 4 is divided into 9. Similarly:

```
10%5        gives 0
7%9         gives 7
```

When negative values are involved, then the result has the same sign as the first value:

```
 7%-3       gives  1
-7%3        gives -1
-7%-3       gives -1
```

Operator Priority

Arithmetic expressions can, on occasion, end up rather long and complex. For example, assuming all the required declarations have been made previously, we might end up with a line such as

```
result = no1 + no2 / 3.0 * no3;
```

The trouble with lines like this are that they are open to different interpretations.

Does it mean add *no1* and *no2* then divide the result by 3.0 then multiply that result by *no3*? Or does it mean divide *no2* by 3.0, then multiply the result by *no3*, then add *no1*? In fact, it's the second of these options.

When calculating the result of an expression, multiplication, division and remainder operations are always carried out before addition and subtraction. That is, multiplication, division and remainder have a higher **priority** than addition and subtraction.

If an expression contains operations of equal priority (as above, where both division and multiplication are used), then calculations involving these operators are performed from left to right. Hence, first the division, then the multiplication and finally the addition are performed to achieve the final result. Exactly how the final value for the expression is calculated is shown in FIG-4.14.

FIG-4.14

How an Expression is Calculated

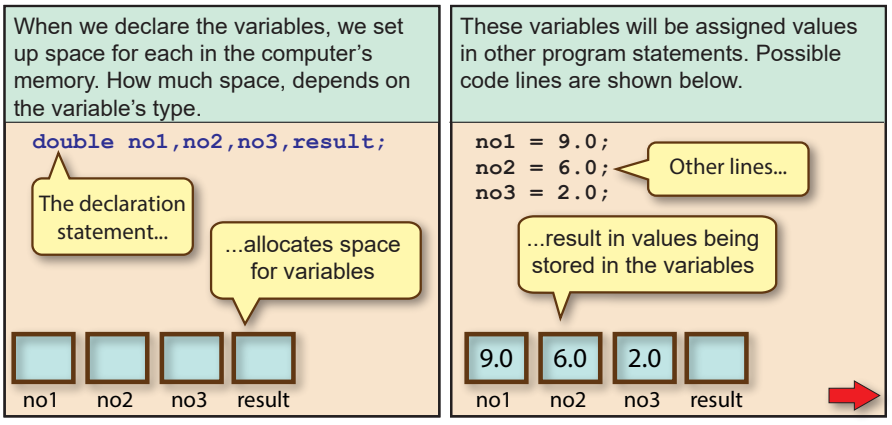
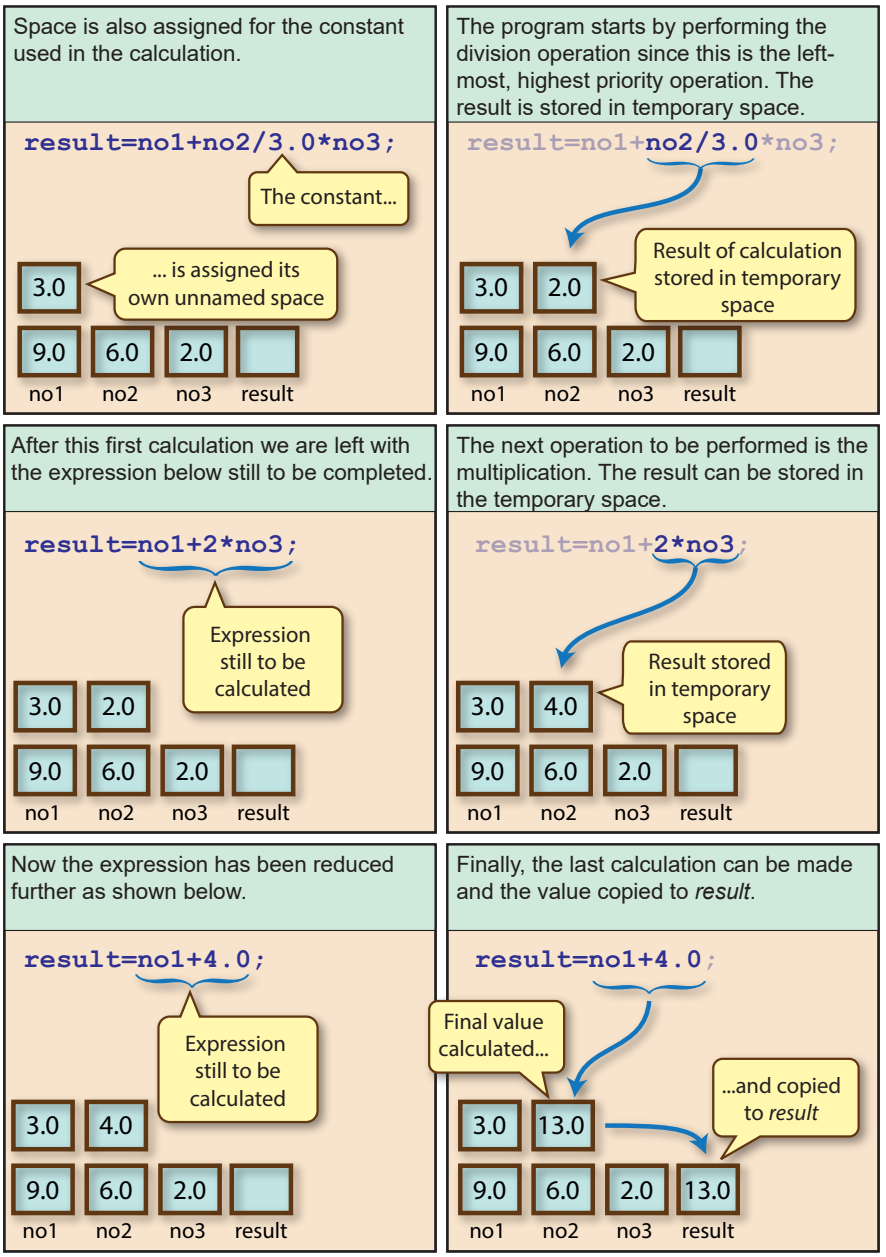


FIG-4.14
(contiued)

How an Expression is Calculated



If we need the operations within an expression to be performed in a different order, we can make use of parentheses. Any calculations within parentheses are always performed first. So in the statement

```
result = (no1 + no2) / 3.0 * no3;
```

no1 and *no2* will be totalled before the division by 3.0 is performed.

With more than one set of parentheses, as in

```
result = (no1 + no2) / (3.0 * no3);
```

the contents of the brackets are calculated on a left-to-right basis. This means `no1 + no2` is calculated first; `3.0 * no3`, second, and finally, the two results are divided.

Parentheses can appear inside other parentheses (called **nested parentheses**). In such cases, values within the inner-most parentheses are calculated first. For example, in the line

```
result = (no1 + (no2 - 5) / no3) * no4;
```

the calculation would be performed using the following steps:

1) `no2 - 5` is calculated first since it is in the inner-most parentheses
2) the result of this is then divided by *no3*
3) *no1* is added to give the overall value within the parentheses
4) this value is multiplied by *no4*
5) the answer is stored in *result*

Note that the division and addition operations are within the same set of parentheses and the normal rules of priority (multiplication and division before addition and subtraction) apply.

We can summarise the order in which the operations within an arithmetic expression are performed with a priority table (see FIG-4.15).

FIG-4.15

Operator Priority

Operator	Description	Priority
( )	parentheses	1
*	multiplication	2
/	division	2
%	remainder	2
+	addition	3
-	subtraction	3

Activity 4.7

What would be the final value of the following expression?

6 -2 * 8 / ((5 + 3) * 2)

Mixed Numeric Types

If the operands of an arithmetic operator are of differing types as in the expression

```
12 * 3L
```

where 12 is stored as an `int` and 3L as a `long`, then the program will automatically convert them all to the same type.

That type will match the value in the expression with the largest range. In this case that would mean both values are converted to a `long` before the expression is evaluated. The result would also be of that same type.

If we look at a longer calculation, say,

```
8.6 + 13 / 3.3F
```

this would be evaluated as follows:

13 is converted to float format (to match 3.3F)
The calculation 13.0F / 3.3F is performed (result is 3.939393F)
The previous result, 3.939393F is converted to a double (to match 8.6)

The calculation 8.6 + 3.939393 is performed (result 12.539393)

When converting numeric values within an expression, values are changed according to the type order shown in FIG-4.16.

FIG-4.16 Automatic Conversion Ordering



If an arithmetic expression involves any `short` or `char` values (`signed` or `unsigned`) they are automatically converted to the equivalent `int` values before any attempt is made to evaluate the expression.

For the most part, during evaluation automatic conversions will not cause problems with the results, but watch out for integer division and the use of `unsigned` results. For example, if we want to get a result of 3.5 from the expression

```
14/4
```

make sure at least one of the values is in floating point format:

```
14.0/4    or    14/4.0
```

Also,

```
int num1{ 12 };
unsigned int num2{ 16 };
cout << (num1 - num2) << endl;
```

will not display -4. Instead, we get

```
4294967292
```

because, when calculating the expression `(num1 - num2)`, both values are held in `unsigned int` format and therefore the result is also an `unsigned int`.

Unusual Assignments

There are a few assignment statements which may seem a little strange the first time you come across them if you're new to programming. For example, the line

```
no1 = no1 + 1;
```

simply means add 1 to the value of *no1*. This is easily understood if we read the line carefully. It contains two instructions:

> Calculate the value obtained by adding 1 to *no1*;
> Store the result in *no1*

Similarly,

```
no1 = no1 + no2;
```

means add the value in *no2* to *no1*.

Another unusual line is

```
no1 = -no1;
```

which is identical in meaning to

```
no1 = no1 * -1;
```

This just means change the sign of the value held in *no1*. So, if *no1* contained the value 12 before the statement was executed, this would change to -12.

On the other hand, if *no1* contained -8 before the statement was executed, it would contain 8 afterwards.

An assignment that is surprisingly useful allows a variable to alternate between two specific values. For example, let's say we want the variable *next* to alternate between the values 14 and 5, then we could do this by first declaring the value to start with one of these values:

```
int next{5};
```

and then executing the line

```
next = 19 - next;
```

to switch value. Each time we want the variable to be assigned the alternate value we execute this same line. Let's take a moment to see how this works. If *next* is 5 when this statement is first executed then we have

```
next = 19 - 5
```

which is 14. If *next* is 14 then the statement evaluates to

```
next = 19 - 14
```

which is 5.

We can get this technique to work for any two values; just subtract the current value of the variable from the sum of the two values you wish to alternate between. So if *next* was to alternate between 2 and 5 the code would be:

```
next = 7 - next;
```

When using division, we might expect the line

```
float answer = 10 / 4;
```

to result in the value 2.5 being stored in the variable *answer*. So it will come as a bit of a shock to find the variable ends up containing the value 2.0.

The reason for this is that when the division operator (/) is working on two integer values (as it is here) then the result is also an integer. Since 4 goes into 10 twice if we ignore any fractions, it is this result that is converted to a `float` value and stored in *answer*. But what if we'd like to get 4.5 as our result when we divide 10 by 4? Easy, just convert at least one of the values to a real value (`float` or `double`) then we'll get the result we are after:

```
float answer = 10 / 4.0;
```

There are other operators that can be used in an assignment operator other than the standard arithmetic ones, but we'll leave these until later chapters.

Activity 4.8

Assuming the following declarations:

```
int no1 = 5, no2 = 10, no3 = -8, result = 2;
```

What value would *result* contain in each of the following:

a) `result = result + no1;`
b) `result = -result % no1;`
c) `result = no3 / no1;`

Invalid Calculations

There are two situations where the result of a calculation is undefined. These are:

a) Division by zero (e.g. `12.6/0.0`)
b) Remainder by zero (e.g. `12 % 0`)

Although these situations are easily avoided when the values involved are constants as in the examples above, these situations are more likely to occur when the expression's second operand is a variable:

```
x = 12.6 / v;     //What if v contains the value zero!
```

Type Conversions

Converting Unnamed Constants

As a general rule, it's a bad idea to try storing a value of one type in a variable of a different type, but C++ will often let us copy some types of values into any type of variable. For example, a program might contain the statement

```
float x{7.93};
```

which, at first glance looks okay, but in fact, *x* and 7.93 are of different types. Since real constants – such as 7.93 – are stored in `double` format, the statement is attempting to copy a `double` value into a `float` variable.

Exactly what the compiler makes of such a statement depends on the actual compiler being used and what warning options have been selected within that compiler. Generally, copying a `double` value to a `float` variable will produce a warning that the value being copied may lose precision since it is being copied into a smaller space than the original (8 bytes for a `double` and only 4 bytes for a `float`) but when the constant involved is within the range that can be stored within a variable, the compiler is likely to accept the statement without comment.

However, even when acceptable to the compiler it is best to ensure both values are of the same type. We can do this for the above statement in one of two ways:

```
double x{7.93}; //Both elements are doubles
```

or

```
float x{7.93F}; //Both elements are floats
```

But sometimes it is not so easy to have the constant match the variable's type. In the line

```
short int a{12};
```

we're attempting to copy an `int` value (12) into a `short int` variable (*a*). C++ doesn't offer any letter that we can place at the end of 12 to convert it to `short int` format. Again, most compilers won't have a problem with this statement since they are smart enough to realise that the value 12 can easily be stored in the smaller space and will perform the conversion for us. But if the value is too large as in the line

```
short int a{12000};
```

we are more likely to get a warning message or even an error. In the case of the current Microsoft C++ compiler, the line above creates the following error message:

```
Error C2397  conversion from 'int' to 'short' requires a narrowing conversion
```

On the other hand, if we write

```
short int a = 120000;
```

the same compiler issues only a warning message:

```
warning C4305: '=': truncation from 'int' to 'short'
```

The same type of response is invoked when trying to store a real value in an integer variable. So,

```
int v1{12.8};
```

causes an error, while

```
int v1 = 12.8;
```

gives only a warning. If we run this second version, *v1* will contain the value 12 with the fractional part of the constant (0.8) being lost.

Converting a value which occupies more bytes than the variable receiving the variables, this is known as **narrowing conversion** (as mentioned in the error message given above).

Converting to a receiving value with more bytes is known as **widening conversion**. However, conversion from a floating-point value (`float`, `double`, `long double`) to any integer type is always considered a narrowing conversion.

Conversion from an integer type to floating-point type will always be handled without comment. The program will simply change the format in which the value is stored meaning that, in decimal terms, it adds .0 to the original value. Hence

```
float v2{18};
```

the integer value 18 is stored in the `float` variable by converting it to the equivalent of 18.0 before being stored in *v1*.

When attempting to convert a character to a number or vice verse with statements such as

```
int v3{'A'};
double v4{'A'};
char letter{65};
```

all three conversions succeed, with *v3* set to 65 (the ASCII code for a capital A), *v4* storing 65.0 and *letter* being set to 'A'. However, if the integer value assigned to a char variable becomes too large,

```
char letter2{512};
```

an error message will result.

We can treat a `char` variable like a numeric value if we require. For example, if we execute the code

```
char letter{ 'A' };
letter = letter + 3;
cout << letter << endl;
```

the value displayed is

```
D
```

The reason for this is that the second line of code converts *letter* to an `int` before adding the two values. Since the ASCII code for A is 65, we end up with the value 68 (65+3). This is then converted back to `char` format for storing in *letter* (68 is the

ASCII code for D).

Statements in which a Boolean value (*true* or *false*) are assigned to numeric variables,

```
int v5{true};
```

uses the same sort of trick to perform the conversion. *true* is stored as 00000001; a binary pattern which converts to 1 (*false* converts to 0).

The Boolean assignment works the other way round too, but only for the integer values 0 (converted to *false*) and 1 (converted to *true*):

```
bool b1{1};      //true
bool b2{0};      //false
```

Other values cause an error:

```
bool b3 {12};  //Error
```

or a warning:

```
bool b3 = 12;
b3 = 12;       //Warning
```

It is this ability to convert between types that means it is quite a simple matter to use `char` variables to store numbers. However, when storing integer values in a `char` variable, with lines such as

```
char no;
no = 18;
```

the compiler will not complain about the type differences (even though 18 is stored as an `int` value and hence has 4 bytes storage compared to the `char`'s single byte). It only complains when the value we are attempting to store is outside the valid range. For example, a program containing the line

```
no = 512;
```

would generate a warning when compiled since 512 is too large to be stored in an `char` variable.

What would happen if we tried to store a negative value in an `unsigned` variable? For example, what value would be displayed if a program executed the following lines:

```
unsigned short s{-17};
```

Although the above version fails, writing

```
unsigned short s = -17;
cout << s << endl;
```

will compile without error or warning!

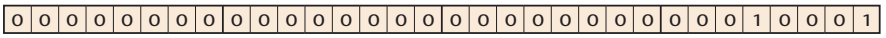
Perhaps, when we run the code given above, we would expect to see the value 17 displayed; the original value with the minus sign missing? If that is the case, then it would come of something of a surprise to discover that the actual value displayed is

```
65519
```

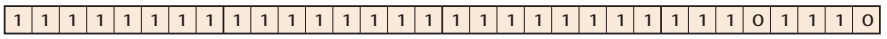
This is where your knowledge of how negative values are stored in binary and how binary is converted to decimal becomes useful. Let's start by understanding how the value -17 is stored within the computer's memory.

As an `int` constant, -17, will be assigned 4 bytes and stored in 2's complement form, so let's remind ourselves how to convert -17 to 2's complement binary. We'll start by

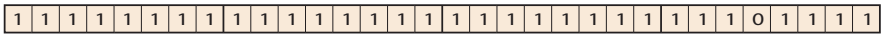
looking at how the value +17 would be stored over 4 bytes as



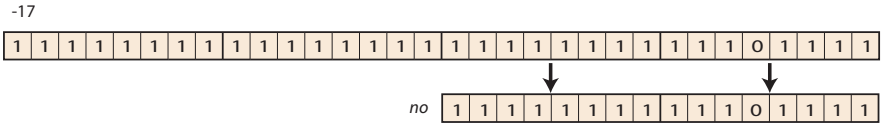
with the bits representing 2^4 (16) and 2^0 (1) set to 1. The next stage is to find the 1's complement by complementing all the bits of the number



and for 2's complement, we add 1.



We are attempting to copy this value to an `unsigned short int`, which is only assigned 2 bytes of space. This means that only the two least-significant bytes of our 4 bytes number can be copied to the variable:



The binary pattern has been moved to an unsigned variable so every bit is assumed to be part of the numeric value (no sign bit). The right-most bit has a value of 2^0, the left-most bit, 2^{15}. Adding the value of each position holding a 1, we get 1+2+4+8+32 +64+128+256+512+1024+2048+4096+8192+16384+32768 which, of course, comes to 65519.

Converting Named Constants, Expressions, and Other Variables

Leaving the computer to do automatic conversions is generally a bad idea because it leaves room for an unintentional conversion which we fail to notice as we create our code. It's much more professional to have explicit conversions wherever possible.

Now, we have seen how to do this with many fixed numeric values - just add the correct letter (F, L, U, etc) to the end of the value. But how do you convert a named constant, variable or arithmetic expression to a different type?

There are several ways to do this in the latest version of C++. For the moment we'll look at only two. Let's assume we want to convert the named constant *PI* from a `double` to a `float` so we can store it in the variable *v1*. In this case, after having declared *PI* as a constant, and *v1* as a `float` variable, we would use the assignment

```
v1 = static_cast<float>(PI);
```

The term `static_cast` is used to tell the compiler that the conversion is to be done during compilation before the program starts running. The same method can be used to convert an expression or normal variable to a required type, as shown in the following lines:

```
area = static_cast<float>(length * breadth);
no1 = static_cast<int>(y);
```

Note that the type we wish to convert to is given in angled brackets < > and that the value being converted must be enclosed in parentheses. If the parentheses enclose an expression – such as `length * breadth` – the result of this calculation will initially be of a type determined by the rules discussed previously (see FIG-4.14).

Always keep in mind that casting to a type with less storage space runs the risk of reducing the accuracy of a numeric value or corrupting it entirely. For example, the following lines

```
int no1;
double x{18.951};
no1 = static_cast<int>(x);
```

will result in the value 18 being stored in *no1* with the .951 being lost.

As we can see, there is no attempt at rounding the value before transfer. If rounding is required then our code must explicitly handle this. This could be achieved in the example above by the following modification:

```
no1 = static_cast<int>(x+0.5);
```

Now the purpose of allowing `unsigned char` and `signed char` as separate types becomes clearer since the range of numeric values which can be held varies between the two types.

Activity 4.9

What value would be displayed by the statement `cout << static_cast<int> (ch);` after executing

a) `signed char ch = -17`
b) `unsigned char ch = -17`

Storing an integer in a `wchar_t` type, with its two bytes, allows for a wider range of values. A variable declaration that specifies the type as simply `wchar_t` creates an unsigned storage space (allowing values in the range 0 to 65,535) but you have the option to explicitly add the term `unsigned` or change the range to -32,768 to 32,767 by declaring the type as `signed wchar_t`.

A second (but now superceded) method of explicit type casting is to specify the type required followed by the value, variable name or arithmetic expression to be converted enclosed in parentheses. For example, we can cast the `double` literal 18.2 to a `float` using the expression

```
float(18.2)
```

A variation on this is to enclose the type in parentheses rather than the value being converted:

```
(float) 18.2
```

If the type required needs two words, then it must be enclosed in parentheses:

```
(long long)23
```

and when the value being converted is an expression we again need parentheses enclosing that value:

```
(long long)(no1 *3)
```

As a general guideline, just remember that C++ will often automatically convert a basic type to any other type, but where that conversion involves moving data from a larger storage area to a smaller storage area we are in danger of losing numeric accuracy (when moving real values) or ending up with a value that bears no apparent relation to the original (moving integers and characters).

Type Aliases

Although we have covered all of the basic types in C++, there are a few other **type aliases** in C++. A type alias is an alternative name for a core or user-defined type. We'll see how to create our own aliases in a later chapter, but for the moment we'll simply mention one of the main aliases we are likely to encounter in existing C++ code.

`size_t` is an alias for an unsigned integer type. Exactly which type of unsigned integer can differ from compiler to compiler, but typically it will be an alias for

```
unsigned int
```

or

```
unsigned long
```

So, if we were to declare a variable to be of type `size_t` as in the line:

```
size_t chars_in_string;
```

This would have exactly the same effect as having written

```
unsigned int chars_in_string;
```



This assumes `size_t` has been declared as an alias to `unsigned int`.

Lvalues and Rvalues

When we write any assignment statement the left-hand side of that assignment must represent a storage space where a value can be stored. For example, in the code

```
int no;
no = 17;
```

the second line is valid because *no* represents an address where a value can be stored. The general term used for such an address is **lvalue**. Think of it as the type of value that must appear on the left side of an assignment.

However, some lvalues can only be assigned a value during initialisation. For example,

```
const float PI = 3.14159;
```

PI is a valid lvalue but it cannot be assigned a new value, hence the line

```
PI = 3.14;
```

would be invalid. Therefore, we can have both **mutable lvalues** (such as *no*) and **immutable lvalues** (such as *PI*).

Returning to our first statements above, the value 17 does not represent a valid address where a value can be stored and is therefore known as a **rvalue**. Should we create a program with the line

```
17 = no;
```

the compiler will issue an error such as

```
Error C2106   '=': left operand must be l-value
```

When an assignment makes use of variable names on both sides of an assignment as in the lines

```
sum = no1 + no2;
```

The compiler will automatically convert any lvalues on the right-hand side of the assignment operator to rvalues.

Looking back at the characteristics of a variable that we used at the start of this chapter, we can say that an lvalue represents a named variable and an rvalue represents a value. Lvalues and rvalues are explained in greater detail in Chapter 25.

Summary

- The four basic data types are : `int`, `float`, `char` and `bool`
- The first three basic types have further options.
- Variables which are to hold integer values can be declared using any of the following terms:

```
short
int
long
long long
unsigned short
unsigned int
unsigned long
unsigned long long
```

- Variables which are to hold real values can be declared using:

```
float
double
long double
```

- Variables designed to store a single character can be declared using:

```
signed char
unsigned char
char
unsigned wchar_t
signed wchar_t
wchar_t
```

- Boolean variables are declared using the term:

```
bool
```

- The space assigned to each variable type can vary between compilers.
- Variable space is assigned within an area of memory known as the stack.
- Variable names should be related to the value held.
- Names must start with a letter or underscore.
- Names of more than one character can use alphanumeric characters and the underscore.
- Case (upper or lower) is relevant when naming variables.
- Names must not be C++ keywords.
- No spaces are allowed within a variable name.
- Variables may be declared with or without a known starting value.
- There are three methods of specifying an initial value when declaring a variable:

using either the assignment symbol (=)	`int v = 12;`
enclosing the value in parentheses.	`int v(12);`

enclosing the value in braces `int v{12};`

- A variable can be declared anywhere in a program as long as it is declared before it is used.
- All variables have four main characteristics:

  A name
  A type
  A value
  A memory address

- A literal is any fixed value within a program.
- A literal will be stored using one of the basic data types.
- Integer numeric constants can be given in decimal, hexadecimal or octal format.
- Hexadecimal constants start with the characters 0X (or 0x).
- Octal constants start with a leading zero.
- Character constants are always enclosed in single quotes.
- Some characters are represented using an escape code.
- Escape code are two digits in length. The first of these being the backslash character.
- The null character is represented using the escape code '\0'
- Normal characters can be represented by using a backslash followed by an x (lowercase only) and the hexadecimal code for that character (e.g. A is '\x41').
- String constants are enclosed in double quotes.
- Any number of characters can be used in a string (including zero characters).
- Escape codes can be included in a string.
- Precede a string with `L` to store its characters in `wchar_t` format.
- Precede a string with `R` and use parentheses as the first and last characters within the quotes to create a raw string where each escape code is treated as two standard characters.
- Where necessary the parentheses of a raw string can be bracketed by additional characters (e.g. `R"??(text of raw string)??"`) to eliminate any ambiguity within the string.
- Constants can be converted to other types by adding a conversion character:

```
12.6F       from double to float
12.6L       from double to long double
12L         from int to long
12U         from int to unsigned int
12UL        from int to unsigned long
L'A'        from char to wchar_t
```

- Named constants are fixed values which are assigned a name.
- The assignment statement is used to assign a value to an existing variable.
- Assigning a value to a variable means that any previous value held in that variable is lost.

- It is good practice to assign a value which is of the same type as the variable in which it is being stored.
- The value being assigned can be given as an arithmetic expression.
- The characters used for standard arithmetic operations are:

*	multiplication
/	division
+	addition
-	subtraction

- When handling two integer values, the division operator returns an integer value.
- The remainder operator (%) is used only on integer values and returns the integer remainder after integer division has taken place.
- When an expression contains several arithmetic operators the order of evaluation is:

  Operations with parentheses are handled first
  Multiplication, division and remainder are handled next
  Addition and subtraction are handled last
  Operators with equal priority are evaluated from left to right

- Using a zero value as the second operand when performing division or remainder operations gives an undefined result.
- When an expression contains a mixture of value types, these are temporarily converted to the "highest" type in that expression before the calculation required is performed.
- Add 1 to a variable using a statement such as

```
no1 = no1 + 1;
```

- Add the contents of one variable to another with a statement such as:

```
no1 = no1 + no2;
```

- Change the sign of a value using a statement such as:

```
no1 = -no1;
```

- For the most part C++ will automatically convert a value to the appropriate type when a value of one type is assigned to a variable of another type.
- When using *uniform initialisation* (the starting value is given within braces) C++ is more likely to reject attempts at assigning values which differ in type from the variable being initialised.
- Real values copied to integer variables have their values truncated, not rounded.
- A character copied to a numeric variable is converted to a number by treating the internal binary representation of the character as a number.
- A number is converted to the character which is represented by the binary pattern of the number.
- Boolean values convert to the numeric values 1 (*true*) or 0 (*false*).
- A numeric value converts to the Boolean value *true*, unless that number is zero, in which case it converts to *false*.

- The contents of a variable can be converted to a different type using `static_cast<`*type*`>`
- `size_t` is an alias for type `unsigned int` or `unsigned long.`
- An lvalue represents an address where a value can be stored.
- An rvalue represents a value that can be assigned.

Solutions

Activity 4.1

```
3 long  variables  = 3 x 4   = 12
1 int              = 1 x 4   =  4
2 wchar_t          = 2 x 2   =  4
total 20 bytes
```

Activity 4.2

c) pic@2 invalid character @
e) long as C++ keyword
f) final total no spaces allowed.

Activity 4.3

b) invalid, semicolon missing
c) invalid, two commas
e) invalid, hyphen not allowed in variable name

Activity 4.4

a) int b) double c) float
d) char e) wchar_t f) char

Activity 4.5

```
const char VOWEL = 'A';
```

Activity 4.6

```
a)   const int DAYS_IN_YEAR = 365;
b)   int totalDays = DAYS_IN_YEAR * 4 + 1;
c)   double side = 7, area = side * side;
```

Activity 4.7

5

Evaluated as

6 - 2 * 8 / (8 *2)
6 - 2 * 8 / 16
6 - 16 / 16
6 - 1
5

Activity 4.8

a) 7
b) -10
c) -1 (integer division)

Activity 4.9

a) -17
b) 239

Since all `char` variable types are assigned a single byte, only the right-most byte of the four bytes allocated to the literal -17 will be copied. This byte contains the value 11101111 which, when stored in a `signed char` variable will be interpreted as -17, in an `unsigned char`, the numeric value 128+64+32+8+4+2+1.

5

I/O and Calculation

In this Chapter:

- ❑ Using cin and scanf()
- ❑ Using cout and printf()
- ❑ Formatting Input and Output
- ❑ Wide Character I/O
- ❑ Arithmetic Assignment Operators
- ❑ The Comma Operator
- ❑ The Increment and Decrement Operators
- ❑ The sizeof Operator
- ❑ Operator Precedence
- ❑ Adding Comments
- ❑ Day of the Week Program

Input/Output

Introduction

It's rather difficult to think of any worthwhile program that doesn't need some input from a keyboard, mouse or other device nor one that doesn't create a result which needs to be output to screen, printer, or a file.

There are many ways of achieving input and output in C++. Some of these the language has inherited from its parent language C, others are new to C++.

Input

Although we can assign a value to a variable while we're writing the program using a statement such as

```
weight = 120.5;
```

there will be times when the programmer isn't going to know in advance exactly what value should be assigned to a variable – we're unlikely to know someone's weight when you are writing a program to calculate their body mass index (bmi). But by using an input statement, we can ask the person using the program to enter the actual value while the program is running.

cin

`cin` (pronounced *see-in*) is one possible term that can be used when a program requires input from the keyboard. Using this statement, we can get the user to enter a value for the variable *weight* using a statement such as:

```
cin >> weight;
```

Note that the term `cin` is followed by two *greater than* symbols (`>>`). These two characters together, when used in conjunction with `cin`, represent the **extraction operator**. This operator must precede the name of the variable for which a value is to be obtained. So, we can interpret the above line as meaning:

Accept a value from the keyboard, and assign that value to the variable weight.

When the computer executes the `cin` statement, it will wait patiently for a value to be entered at the keyboard. Once that value has been keyed in and the **Enter** key pressed, the value is assigned to the variable *weight*. FIG-5.1 shows a complete program for reading in and displaying someone's weight.



You may wonder why an operator that is used in assigning a value to a variable is known as an extraction operator. This is because it is perceived as extracting data required by the variable from the keyboard buffer.

FIG-5.1

Display Salary

```
#include <iostream>
using namespace std;

int main()
{
   int weight{};

   // *** Get value for weight ***
   cout << "Enter your weight (kilos) : ";
   cin >> weight;
   // *** Display weight ***
   cout << "Your weight is " << weight << " kilos" << endl;
}
```

Notice that the `cin` statement is preceded by a `cout` statement. We know from the two programs we ran back in Chapter 3 that the `cout` statement is used to output messages and values to the screen. In this case, it is being used to display a message telling the person running the program exactly what type of value they are expected to enter as well as the units of measurement. Such a message is known as a **prompt**.

Activity 5.1

Start a new project called *Input01*.

Enter the code shown in FIG-5.1.

Compile and run the program, entering a value of 65.2 when prompted for the weight.

It is possible to accept more than one value using the `cin` statement by listing the name of each variable whose value is to be entered. Each name must be preceded by the extraction operator. For example, we could read in the day, month and year of a date with the statement

```
cin >> day >> month >> year;
```

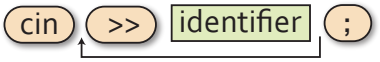
This assumes that *day*, *month* and *year* have be declared as integer-type variables.

When the values are being entered by the user, they should all be entered on a single line separating each value by a space. Once the **Enter** key has been pressed, the first value keyed in is assigned to *day*, the second to *month* and the third to *year*.

The format for the `cin` statement is shown in FIG-5.2.

FIG-5.2

cin Syntax



where:

identifier is a previously declared variable.

Having said that several values can be read in using a single `cin` statement, this is usually not a good idea since it makes it much more likely that the user will make a mistake while entering values. The best policy is to use one `cin` statement for every value that is to be entered.

At this stage in our knowledge we have no way of ensuring that the value entered by the user is valid. For example, if we ask the user to enter the month of the year, we have not created code to make sure that the value entered is in the range 1 to 12 – the user is free to type anything they wish at the keyboard. Later we'll see how to gain much greater control over this.

If the value to be entered is a character, do not add the single quotes which we see in the assignment statement. Hence, in response to the statement

```
cin >> agreed;
```

This assumes *agreed* is declared as a character-type variable.

we would enter *Y*, not *'Y'* (never enter the quotes).

Integer values cannot be entered in hexadecimal, octal or binary format.

Activity 5.2

Which of the following are invalid `cin` statements? You may assume all variables have been appropriately declared.

a) `cin << no1;`
b) `cin >> no1,no2;`
c) `cin >> rno1 >> rno2;`
d) `cin >> letter;`

scanf_s()

scanf_s() is a Microsoft replacement for the original *scanf()* function was used for input but a call to *scanf()* no longer compiles in VS2019 using default settings.

`scanf_s()` (maybe `scanf()` in other compilers) is a function designed to accept input from the keyboard. Its origins are in C but it is also available in C++. Usually, it has little advantage over using `cin`, but it does allow added flexibility if we want to enter values in hexadecimal or octal format.

The `scanf_s()` function takes a variable number of parameters, but the first of these is a string which describes the type of value being entered. For example, when a decimal integer is to be entered, we would use the string `"%d"`. Following the type specifier string we have a comma-separated list of the memory addresses where values being entered are to be stored.

The link between variable names and memory addresses is explained in Chapter 2.

To specify the address of a variable in C++, we precede the variable name by an ampersand (`&`) as in

```
&age
```

Using `scanf_s()` we could read in a value for `int` variable *age* using the statement

```
scanf_s("%d",&age);
```

The following type specifiers (see FIG-5.3) can be used with the `scanf_s()` function.

FIG-5.3

scanf_s() Format Codes

Type Specifier	Meaning
c,C	char/wide char
d	integer
e, E,f, g, G	fixed point float
o	octal
s	string
u	unsigned integer
x,X	hexadecimal

To read in a character value we use the string `"%c"` (lowercase c) for a normal 1 byte ASCII character and `"%C"` (uppercase C) to read in a wide, two-byte character. Examples of relevant lines of code are shown below:

```
char letter;
wchar_t symbol;
scanf_s("%c",&letter);
scanf_s("%C",&symbol);
```

When entering a real value, it makes no difference which of the type specifiers is used, *e, E, f, g,* and *G*, all have the same effect. For example, we could read a value for a `float` variable, *x*, using the line

```
scanf_s("%f",&x);
```

then enter the value in either fixed point notation

```
1.234
```

or in scientific notation

```
123.4E-2
```

When entering an integer value in hexadecimal value at the keyboard, the 0X can be included or omitted - it will make no difference to the interpretation of the value entered as long as we remember to use the string `"%x"` as the first value in `scanf_s()`.

Strings (such as a name or address) can be read using `"%s"`, but we will leave string handling until a later chapter.

There is no code for reading in a Boolean value.

Activity 5.3

Start a new project called *Input02*.

Write a program (based on *Input01*) which uses `scanf_s()` to read a decimal value into a variable named *v1* and then displays the value held in the variable.

Modify the program to read a hexadecimal value.
Modify the program to read an octal value.

Like `cin`, it is possible to use a single call to the `scanf_s()` function to read in more than one value. For example we could read in values for variables *day*, *month* and *year* using the statement

```
scanf_s("%d %d %d", &day, &month, &year);
```

Note that the string contains format information for each of the three values being input. The space between each `%d` in the string tells the program that there will be whitespace characters between the values when they are entered at the keyboard. A typical entry in response to the above line would be

```
26 10 2021
```

In this case the whitespace between each value is a single space, but the user can enter any number of any whitespace characters (space, tab, return) between the values.

Activity 5.4

Start a new project called *GetDate*.

Write a program to read in values for variables *day*, *month* and *year* (remember to include an appropriate prompt). Include the statement

```
cout << day << '/' << month << '/' << year << endl;
```

to display the values that have been entered.

Run the program several times, varying the number and type of whitespace characters between each value.

By changing the space between the `%d` elements within the string, we can actually specify exactly which character(s) appear between the data to be read from the keyboard. For example, by changing the `scanf_s()` statement we used earlier to read

in values for *day*, *month* and *year* to

```
scanf_s("%d/%d/%d", &day, &month, &year);
```

the user can enter a more natural looking

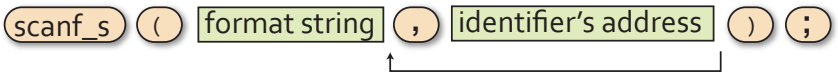
```
26/10/2021
```

and the program will ignore the forward slashes when determining the values to be assigned to the variables.

The format of the `scanf_s()` function is shown in FIG-5.4.

FIG-5.4

scanf_s() Syntax



where:

format string is a string containing details of what type of value(s) are to be entered.

identifier's address is the address in which the value entered is to be stored. The address is created by preceding the variable's name by an ampersand (`&`).

Activity 5.5

Modify your *GetDate* project to allow a forward slash between each value entered.

Does the program still operate correctly if more than one forward slash is entered between the values?

Save your program.

Inputting a Wide Character

If we have declared a `wchar_t` variable and want to assign it a value read from the keyboard, you need to use `wcin` rather than `cin`. For example,

```
wchar_t ch{};
cout << "Enter character: ";
wcin >> ch;
```

Alternatively, `scanf_s()` is quite capable of reading in a `wchar_t` value as long as we remember to use `%C` (uppercase C) in the format string:

```
scanf_s("%C",&ch);
```

Output

We've already used the `cout` statement to create output in earlier programming examples. Now we'll have a more formal look at the statement as well as exploring the `printf()` function which is the main output method in C.

cout

`cout` (pronounced *see-out*) is most often used to output values to a console window (other methods of output exist when dealing with a GUI application). One of the commonest uses for `cout` is as a prompt requesting data from the user:

```
cout << "Please enter your age : ";
```

We can an also use it to display the contents of a variable as in the line

```
cout << salary;
```

Note that the term `cout` is followed by two *less than* signs (`<<`). These two characters together represent the **insertion operator**. This operator must precede the value that is to be output. So, we can interpret the above line as meaning:

Send the value held in the variable salary *to the screen.*

Simple constants can be displayed with statements such as

```
cout << 17;
cout << 15.5;
cout << 'X';
cout << "Hello world";
```

and several values can be displayed using a single `cout` statement by creating a `<<` separated list:

```
cout << 17 << 15.5 << 'X' << "Hello world";
```

The trouble with outputting several values (even if it's done using several `cout` statements) is that no gaps are produced between each of the values being displayed. So both of the examples above will produce the following output:

```
1715.5XHello world
```

To create a gap between each value, we must explicitly display strings of spaces:

```
cout << 17 <<"   "<< 15.5 << "   "<< 'X'<< "   " << "Hello
↳world";
```

The same problem exists when using the four separate `cout` statements to display the result, so the `cout` statements would have to be rewritten as:

```
cout << 17 << "   ";
cout << 15.5 << "   ";
cout << 'X' << "   ";
cout << "Hello world";
```

As we can see, the `cout` statement outputs exactly what we ask of it and nothing more.

If we want to move to a new line after a value is displayed, then we must explicitly display a newline character. For example,

```
cout << 17 << '\n';
cout << 15.5 << '\n';
cout << 'X' << '\n';
cout << "Hello world\n"
```

would create the output

```
17
15.5
X
Hello world
```

An alternative to the newline character is to use the term `endl` (meaning *end line*) which has a similar effect. So we could move the cursor to a new line after outputting the value 17 using the line:

```
cout << 17 << endl;
```

A variable's contents can also be displayed using `cout` as exampled in the lines

```
int no1 {12};
cout << no1 << endl;
```

A final option is to output the result of an expression with a line such as

```
cout << no1 * 5;
```

which, assuming *no1* contains the value 12, would display the value 60.

We should always add a descriptive text when displaying a value. For example,

```
cout << "Salary is $" << salary << endl;
```

Without this, the displayed values will be meaningless to anyone viewing them.

Activity 5.6

Write a program (called *BMI*) to read in your age, weight (kilos) and height (metres) and calculate your body mass index (BMI) using the line

```
bmi = weight / (height + height);
```

Display all of the data entered as well as the body mass index. Also display a message saying that a BMI of more than 25.0 indicates being overweight.

A set of format manipulation functions can be used in conjunction with the `cout` statement to gain more control over how information is displayed on the screen. For example, we can set the number of characters assigned to the display of a value using the function `setw()`. If we wanted the number 17 to be displayed in a space large enough to accommodate a 6 digit value we would write:



The number of character positions assigned to the output of a value is often referred to as a **field size**.

```
cout << setw(6) << 17;
```

Since the number 17 only requires 2 of those 6 spaces, the other 4 positions are left blank so the output from the above statement would be



The symbol Δ is used here to represent a space.

ΔΔΔΔ17

The `setw()` function only affects the value immediately following it, not subsequent values. Hence, the line



The value 17 is right-justified with the field.

```
cout << setw(6) << 17 << -15.5 << endl;
```

will produce the output

ΔΔΔΔ17-15.5

To create the correct spacing we would have to use the line

```
cout << setw(6) << 17 << setw(6) << -15.5 << endl;
```

which produces the output

ΔΔΔΔ17Δ-15.5

Notice that the minus sign and decimal point occupy two of the six positions in the field assigned to the value -15.5.

If we give a field size which is too small for the value being displayed, then the field size information will be ignored and the value will occupy as many positions on the screen as it requires.

In order to use the `setw()` function in a program, we need to include relevant information about the function. This information is held in the header file *iomanip* and can be added to a program using the line

```
#include <iomanip>
```

at the start of the code.

A second manipulator function, `setbase(v)` can be used to show all subsequent integers in the base *v* where *v* can be 8, 10 or 16. For example,

```
cout << setbase(16) << 255 << endl;
```

would display the decimal value 255 as hexadecimal ff.

Unlike `setw()`, `setbase()` affects all further integer values output using `cout`, so it is necessary to revert to the base 10 by subsequently using the term `setbase(10)`.

```
cout << setbase(16) << 255 << setbase(10) << endl;
```

Other format manipulators that can be used with `cout` are listed below (see FIG-5.5);

FIG-5.5

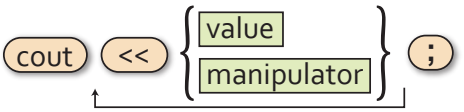
Format Manipulators

Formating Terms	Meaning
dec	all following integers shown in decimal
hex	all following integers shown in hexadecimal
oct	all following integers shown in octal
showbase	shows the number base (0X or 0) for all following hex and oct values
noshowbase	switches off the number base for all following hex and oct values
fixed	displays all following floating-point values in fixed-point format
scientific	displays all following floating-point values in scientific notation
defaultfloat	displays all following values in fixed or scientific format as appropriate
left	left-justifies value within its field
right	right-justifies value within its field
setw(n)	sets the width of field for next value to *n*
setprecision(n)	sets precision of floating-point values to *n*
setfill(ch)	fills unused positions in a field with character *ch*

The format for `cout` is shown in FIG-5.6.

FIG-5.6

cout Syntax



where:

value is a constant, variable, expression.

manipulator is a function designed to manipulate the format used to output data.

Other than `setw()`, all other manipulators operate on all the values that follow them. For example, if our program contains the lines

```
cout << hex;            //Remember to use lowercase!
cout << no1 << endl;
```

```
cout << no2 << endl;
```

the values of both *no1* and *no2* will be displayed in hexadecimal.

Note from the above code, that when we intend to have a format setting (`hex`) last for several `cout` lines it is best to specify that setting on a separate line. This helps make our intentions clear to others who may look at our code. In other words,

```
cout << hex;
cout << no1 << endl;
```

is clearer than

```
cout << hex << no1 << endl;
```

Of course, there is no need to do this if we intend to use the setting for only a single line of output, in which case we might write

```
cout << hex << no1 << dec << endl;
```

If we activate `showbase` for hex and octal values as in

```
cout << showbase;
cout << hex << no1 << endl;
cout << oct << no2 << dec << endl;
cout << noshowbase;
```

then hexadecimal values are output with a leading 0X (e.g. 0X2F) while octal values have a leading zero (e.g. 044).

The `setprecision()` option is used when displaying floating-point values, but has a slightly different effect depending on the formatting being used (`scientific`, `fixed`, `defaultfloat`). To demonstrate this, let's assume we have declared the following

```
double x{123.456789};
```

If we use the following line to output the value of *x*

```
cout < setprecision(4) << x << endl;
```

we might expect to see the value of *x* displayed to 4 decimal places:

```
123.4568
```

but, in fact, what we get is

```
123.4
```

The `setprecision()` manipulator has set the TOTAL number of digits to be displayed and not the number of decimal points. However, if we specify fixed-point notation (by default, the format used is `defaultfloat`), then the effect of `setprecision()` changes to specify the number of decimal places. In other words,

```
cout << fixed;
cout << setprecision(4) << x << endl;
```

outputs

```
123.4568
```

If we specify scientific notation,

```
cout << scientific;
cout << setprecision(5) << x << endl;
```

`setprecision()` determines the number of decimal places displayed in the significand.

When a field size is specified for a value, then where that field size is too large we've

already seen that the surplace positions are space-filled. However, by using `setfill()` we can specify a different character to occupy these positions. for example, in the code

```
int no1{ 24 };
cout << setfill('*');
cout << setw(10) << no1 << endl;
```

the output would be

```
********24
```

By default, values are right-justified within the space they are assigned, but we can change this to left-justified using `left`:

```
int no1{ 24 };
cout << setfill('*') << left;
cout << setw(10) << no1 << endl;
```

This will output

```
24********
```

Outputting Boolean Values

When trying to output the value of a `bool` variable as in the lines

```
bool okay{true};
cout << okay << endl;
```

where the value of the `bool` variable is *true*, the value shown on the screen is

```
1
```

When the variable has the value *false*, the output will be zero.

If we want to see the words *true* or *false* displayed instead, then we can use the manipulator `boolalpha` as in the line

```
cout << boolalpha << okay << endl;
```

which would output

```
true
```

Unlike previously described manipulators, this one isn't defined in the header file *iomanip*, so there are no additional `#include` lines required to make use of it.

Once activated, this manipulator affects all subsequent outputs involving Boolean values. However, we can return to the default 1 and 0 output using `noboolalpha` as a term within a `cout` statement, as in the line:

```
cout << boolalpha << okay << noboolalpha << endl;
```

printf()

Another function inherited from C is `printf()`. At its simplest, we can use `printf()` to display a message:

```
printf("Hello world\n");
```

When we want to display the contents of a variable, then we need to embed formatting information in the output string. These formatting elements within the string always begin with a percent (`%`) character which is followed by the formatting character itself. For example, if we wanted to output the contents of the integer variable *no*, defined as

```
int no = 12;
```

we would write:

```
printf("The value is %d\n",no);
```

The `%d` formatting element specifies where in the text being output a signed integer displayed in base 10 format is to be inserted. The value inserted is held in the named variable, *no*. The output produced would be:

```
The value is 12
```

The formatting character used depends on the type of value being output (integer, real, character, etc) and, in the case of integers, the number base to be used when displaying the value. A complete list of the formatting characters is given in FIG-5.7.

FIG-5.7

printf() Format Codes

Symbol	Meaning
d or **i**	signed integer
u	unsigned integer
f or **F**	fixed point float
e or **E**	scientific notation float (uses e or E)
g or **G**	fixed or scientific notation float as appropriate
x or **X**	hexadecimal (lower or upper case letters)
o	octal
c	character
s	string

For example, the line

```
printf("%f %e %E %X %c",123.4, 123.4, 123.4, 255, 'A');
```

would display

```
123.400000 1.234000e002 1.234000E02  FF A
```

Another option is to include field size details after the `%` symbol. To display an integer value in a field 6 characters long we would use the term

```
%6d
```

If we want unused positions within the field to be zero-filled, then we add a decimal point

```
%.6d
```

Real numbers can be formatted by specifying not only the total size of the field but also the number of decimal places:

```
%8.2f
```

This example specifies the field to be 8 characters in length, two of which are after the decimal point.

So the line

```
printf("%6d %.6d %8.2f",123, 123, 123.4);
```

would produce the outout

```
ΔΔΔ123 000123 ΔΔ123.40
```

We can control other aspects of how numeric values are displayed by adding another character (known as a **flag**) to the formatting information. The options available are

listed in FIG-5.8.

FIG-5.8

printf() Flags

Flag	Effect
+	Positive values preceded by +. Negative values by -
Δ	Positive values preceded by a space. Negative values by -
-	left-aligned output
#	with g and G, trailing zeros not removed with F,f,g,G,e and E, a decimal point is always included with o,x and X, 0 or 0X is included in the output
0	pad fields with zeros instead of spaces

For example, the statement

```
printf("%+5d %#X %08.2f", 12, 255, 9.6732);
```

outputs

```
ΔΔ+12 0XFF 00009.67
```

More than one flag can be included in a single formatting sequence. For example, we could force the +12 output of the above example to be left-justified using the line

```
printf("%-+5d",12);
```

meaning the output produced would be

```
+12ΔΔ
```

And, finally, if you want to output the % symbol as one of the displayed characters, use two % symbols. Hence the line

```
printf("%%");
```

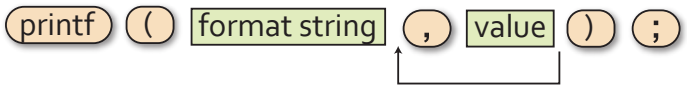
outputs

```
%
```

The format of the `printf()` function is shown in FIG-5.9.

FIG-5.9

printf() Syntax



where:

format string is a string detailing how the specified values are to be displayed, as well as any additional text to be included.

value is a constant, variable, or expression.

Activity 5.7

Create a project called *DecHex* which reads in an integer value and then display it in decimal and hexadecimal.

Activity 5.8

Modify your *BMI* program so that it uses a single `printf()` call to display your height, weight and body mass index, each value being displayed on a separate line. Don't save this version.

Activity 5.9

Start a new project called *Times*. Code the program so that it reads in two times (given in minutes and seconds) and displays their total.

There are couple of ways to tackle this problem but if we were to do it on a piece of paper we would add together the two seconds values and carry any minutes produced to the minutes then total the minutes. The logic of this can be written as

Get the two times (in minutes and seconds)
Add the two seconds values together
Divide this total seconds by 60 to get minutes carried
Calculate remaining seconds after converting some of them to minutes
Add the two minutes values and any carried minutes together
Display the total minutes and the remaining seconds

Outputting a Wide Character

If we attempt to display the contents of a wide character variable using `cout`, we'll see the decimal equivalent of the character rather than the character itself. So, for example,

```
wchar_t ch = L'A';
cout << ch << endl;
```

would display 65.

To have the character itself displayed, use `wcout` in place of `cout`:

```
wcout << ch << endl;
```

A wide character can be displayed using `printf()` as long as `lc` is used in the formatting string:

```
printf("%lc",ch);
```

About Namespaces

At school, a tutor often has to deal with many students who have the same forename, say *John*, for example. That isn't really a problem for the tutor of course, since they can always refer to *John in class C5* or *John in class A2* to make sure others knew which particular *John* they were talking about.

Namespaces are a bit like those school classes. A namespace groups together a set of user-defined elements such as named constants, functions and classes into a named collection. We'll see how to do this in a later chapter.

There can be many separate namespaces as long as each namspace has its own unique group name. And although we will create our own namespaces particularly when creating larger programs, there are also many namespaces already defined within C++.

When we include a line such as

```
using namespace std;
```

we are telling the compiler that any elements that the program refers to which are not part of the C++ core language and not identifiers specifically set up within the

program should be assumed to be in the existing namespace collection known as *std*.

Activity 5.10

To find out exactly which terms in a program the compiler finds in the std namespace, we are going to remove the `using namespace std;` *line from an existing program.*

In your previous project (from Chapter 3), *First*, remove the line `using namespace std;` from the program and recompile the code.

Does the program still compile?

Of course, we can solve the problem we created in the last Activity by adding back the statement we have removed, but an alternative way is to add the namespace's ID, and a double colon to the start of each unknown term. For example, by changing the line

```
cout << "Hello world\n";
```

to

```
std::cout << "Hello world\n";
```

the compiler will again recognise the `cout` statement since the extra term tells the compiler in which namespace to find the definition of the term `cout`.

The double colon is known as the **scope resolution operator**.

Activity 5.11

Modify *First* to include the text

```
std::
```

at the start of the `cout` statement. Does the program compile and run?

The fact that `cout` is not recognised without the namespace specification tells us that the term is not a core part of the C++ language. What it actually is, we will find out in a later chapter.

In fact, we glossed over the need to specify namespace names in all the code we've used up to now and that has been due to our use of the line:

```
using namespace std;
```

Without it, we'd have to add the `std::` qualifier to all of the options listed below:

```
cin
cout
wcout
boolalpha
noboolalpha
```

The code that makes up `main()` and any other code not specifically placed within a namespace are consider to be in the **global namespace**.

NOTE: It is considered bad practice to use a `using` statement within a program since it defeats the reason why namespaces were created in the first place. For example, if we were to create a variable called `cout` with the line

```
int cout;
```

then the compiler would indicate an error when it encountered a line such as

```
cout << "Hello world\n";
```

since it would no longer recognise `cout` as a method of outputting data.

On the other hand, if we had omitted the `using` statement, our two lines would become

```
int cout;
std::cout << "Hello world\n";
```

and the confusion between the two `cout` identities would disappear.

However, for the short programs in this book we will, in general make use of the line

```
using namespace std;
```

simply to shorten the terms within the code.

Summary

- `cin` is an input command, allowing the value being assigned to a variable to be entered from the keyboard.
- `scanf_s()` is a function which allows values to be entered from the keyboard.
- `cout` is an output statement used to display information on a screen.
- `printf()` is a function, inherited from C, which displays information on a screen.
- The line `#include <iostream>` must be placed at the start of any program which uses `cin` or `cout.`
- When using `cout`, display the newline character (`'\n'`) to move the cursor to a new line or, alternatively use the term `endl`.
- Use `setw()` to set the size of the field used when displaying a value.
- `setw()` must be used before each value whose field size is to be set.
- Use `setbase()` to set the number base for displaying integer values.
- The `setbase()` setting remains in use until reset by a different value.
- When using `setw()` or `setbase()`, the line `#include <iomanip>` must be added at the start of the program.
- Other output formatting options allow for different number bases, fixed and scientific numeric notation, justification within a field and specifying a fill character.
- Use `boolalpha` to output `bool` values in text form.
- Use `printf()` as an alternative way of outputting values to the screen.
- Use `wcout` to output wide characters.
- The `using namespace` command tells the compiler which namespace group to search for any term which is not part of the core C++ statements nor defined within the program.

More Operators

We looked at the basic arithmetic operators in the previous chapter, but there are several more arithmetic operators which, although not entirely necessary can allow us to write shorter code.

The Arithmetic Assignment Operators

To add a value *v* to a total, *sum*, we could write

```
sum = sum + v;
```

but in C++ we can shorten this to

```
sum += v;
```

The shorter version is certainly more obscure if we're not familiar with the C++ language but it does make for less typing.

We can do the same sort of thing with subtraction, multiplication, division and remainder:

`no1 = no1 - 6;`	becomes	`no1 -= 6;`
`no2 = no2 * 5;`	becomes	`no2 *= 5;`
`no3 = no3 / no1;`	becomes	`no3 /= no1;`
`no4 = no3 % 5;`	becomes	`no4 %= 5;`

Activity 5.12

Rewrite each of the following lines of code to use arithmetic assignment operators:

a) `sum = sum + no;`
b) `count = count - 6;`
c) `ans = ans * (mark / 7);`
d) `total = total % (previous - current);`

Increment and Decrement Operators (++ and - -)

We now have two ways of adding 1 to a variable:

```
no1 = no1 + 1;
no1 += 1;
```

but the **increment operator** (`++`) offers a third way of achieving the same effect:

```
no1++;
```

The operator can also appear before the variable:

```
++no1;
```

the effect is the same in this example.

But the increment operator is more powerful than it might first appear since it can be used within longer statements. Hence, it is quite valid to write

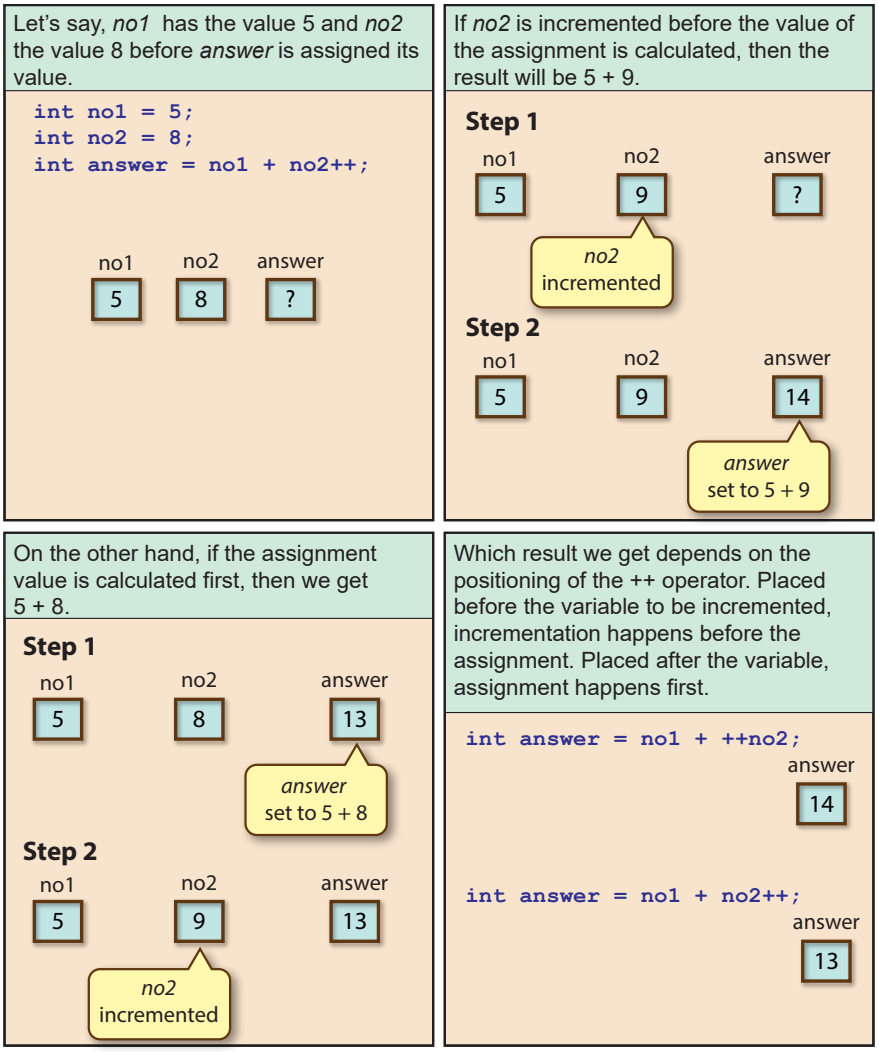
```
int answer = no1 + no2++;
```

This single statement now stores a value in *answer* and adds 1 to the contents of *no2*.

When the contents of two variables are being changed in a single statement, it is important to know just exactly what order things are happening in. Exactly how the task is performed is shown in FIG-5.10.

FIG-5.10

How the Increment Operator Works



Positioning the increment operator before its variable is known as **pre-incrementing**, placing the operator after the variable is known as **post-incrementing**.

In the statement below three + symbols are given in sequence:

```
answer = no1+++no2;
```

Such a situation is ambiguous since it lends itself to two meanings:

```
no1++ + no2
```

or

```
no1 + ++no2
```

However, as long as we add a space at the appropriate point so that the three + signs are split up, the compiler is quite happy to accept the statement.

The increment operator can also be used on a character variable. Incrementing the variable moves the value held to the next character in the ASCII or Unicode table.

For example, in the code snippet

```
char ch = 'C';
ch++;
```

the variable *ch* would contain the character 'D' after the increment operation had been executed.

We also have a **decrement operator** (--) which subtracts 1 from the specified variable and can also be used in either the pre-increment or post-increment position.

Activity 5.13

Write down the value of each variable after the following statements are executed in sequence:

```
int no1(1), no2(10), answer;
answer  = ++no1 * no2--;
answer *= no1++;
```

Activity 5.14

Start a new project called *CharBefore* and create a program which reads a character from the keyboard and displays the preceding character in the coding system (typing in B would display A).

The Comma Operator

The comma operator (,) is, at first glance, perhaps the most useless of things. It is placed between two values and returns the second of these values. So in the line

```
answer = (7 , 12);
```

answer will be assigned the value 12.

Notice the need for parentheses round the two values. These are required because the comma operator has such a low priority it would normally be executed after the assignment operator.

The values either side of the comma can, of course, include variables and arithmetic expressions such as

```
answer = (8 + 2, no1 - 9);
```

in which case the expressions are both calculated before the result of the second expression is assigned to *answer*.

The comma separated terms can even be complete assignment statements as in:

```
answer = (no1 = 12, no2 = no1 + 1);
```

In this case 12 is assigned to *no1*, 13 is assigned to *no2* and the second of these, 13, is assigned to *answer*.

Arithmetic operators and the assignment operator all have a higher priority than the comma operator and so the assignments to variables *no1* and *no2* are completed before the comma operator evaluates its two operands (*no1* and *no2*).

Like other operators, we can use several commas in a statement to create a line such as

```
no1 = (no2 = 12, no3 = 6, no4 = no2 + no3);
```

which contains three commas. It's the value to the right of the final comma that is returned, so in the example above, *no1* would be set to 18.

One final point about the comma operator is that it may be used without making use of the value it returns. This allows us to write a line such as

```
no2 = 12, no3 = 6, no4 = no2 + no3;
```

With the return value unused, the need for parentheses also disappears and we have a line of code that is simply a request to execute three assignment statements. However, use of the comma in this way is normally not done. The preference would be to omit the commas and have each assignment statement on a separate line terminated by a semicolon:

```
no2 = 12;
no3 = 6;
no4 = no2 + no3;
```

At this point you're probably thinking that's a lot of explaining for an operator that you can see no use for. But, in fact, the comma can be used quite extensively in C++ code, but in more complex situations that we will come across in later chapters.

Multiple Assignment

If we want several variables (which have been previously declared) to be assigned the same value, then we are allowed to write statements such as

```
no1 = no2 = no3 = 9;
```

which will assign the value 9 to all three variables named in the statement. The statement is evaluated by the computer from right to left so *no3* is first assigned the value 9 and then the contents of *no3* are copied to *no2* and this is finally copied to *no1*.

The *sizeof* Operator

The `sizeof` operator returns the number of bytes required to store a value of a given type. For example, the expression

```
sizeof (int)
```



Of course, the value returned may vary if you are using a different compiler.

returns the value 4 since `int` values are stored over four bytes.

The operator can also be used on a variable to discover how many bytes it occupies:

```
sizeof (no1)
```

If *no1* is a `double` variable, then the value 8 would be returned.

When specifying the argument for the `sizeof` operator, types must be enclosed in parentheses as in the expressions

```
sizeof (long)
sizeof (double)
sizeof (char)
```

but variable names can be used without the parentheses. So assuming *a*, *b* and *c* have previously been declared as variables, the following expressions are valid:

```
sizeof a
sizeof b
sizeof c
```

The value returned by `sizeof` is in `unsigned int` format.

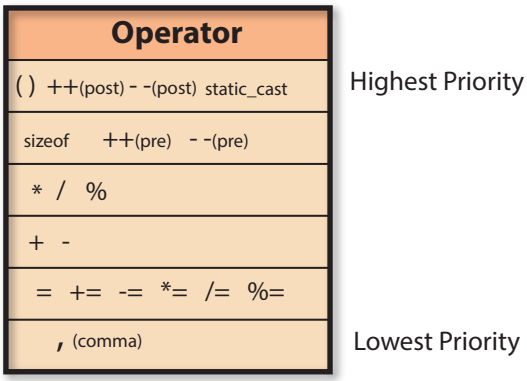
We'll find `sizeof()` very useful later.

Updated Operator Precedence Table

When presented with a piece of complex code we may only understand its purpose if we have a good grounding in operator precedence. We've met a few new operators in this chapter and so it is time we updated the operator precedence table we saw back in Chapter 4. The updated version is shown in FIG-5.11.

FIG-5.11

Operator Priority



Summary

- An arithmetic assignment operator can be used to shorten an assignment statement.
- Arithmetic assignment operators are:
  `+=` `-=` `*=` `/=` `%=`
- The increment operator (`++`) can be used to add 1 to a specified variable.
- The increment operator can be placed before (prefix) or after (postfix) the variable to be incremented.
- If the increment operator is prefixed, then the variable is incremented before any further operations on that line are carried out.
- If the increment operator is postfixed, then the variable is incremented after any further operations on that line are carried out.
- The decrement operator (`--`) subtracts 1 from a variable but otherwise operates in the same fashion as the increment operator.
- The increment and decrement operators can be used on character variables as well as numeric ones.
- The comma operator (`,`) takes two operands and returns the value of the second operand.
- The comma operator has such a low priority, it and its operands must often be enclosed in parenthesis.
- Several variables can be assigned an identical value in a single statement.
- The `sizeof` operator returns the number of bytes assigned to a variable or variable type.

Comments

The importance of comments within a program cannot be overstressed. It is only through the liberal use of comments that we can understand the purpose of each section of code in a program. In the real world, program code will have to be understood by people other than the original programmer who may well have left the organisation for pastures new by the time a program needs to be modified.

Even if we are writing programs simply for our own purposes, comments are still important. Code whose purpose seems obvious on the day we write it will, in a few weeks or a few months time, often have little or no meaning to us if we have not added comments.

C++ allows two methods of adding comments:

`//` All remaining text on the current line following the double forward slash will be treated as comments.

`/* */` All text between the opening comment symbol `/*` and the closing symbol `*/` will be treated as comments. The comments can stretch over several lines.

A Guessing Game

The program in FIG-5.12 implements a simple guessing game. The computer "thinks" of a number between 1 and 10 and then we have to guess what the number is. The program then displays the value it came up with as well as our own guess.

At this stage, it's not a very interesting game, but we will develop it more fully over the next few chapters, inserting new statements to add a little complexity to the game.

The first part of the code makes use of some standard C++ functions which we haven't covered yet. For the moment, just accept that the code does what the comments say. In a later chapter we'll explain the full meaning of the code.

FIG-5.12

The Number Guessing Program

```
#include <iostream>
#include "cstdlib"
#include "ctime"
using namespace std;

int main()
{
   int number;    // the number selected by the computer
   int guess;     // the user's guess at the computer's number

   //*** Generate random number in the range 1 to 10 ***
   srand(static_cast<unsigned>(time(0)));
   number = rand()%10 + 1;
   //*** Get user's guess ***
   cout << "Guess the number (between 1 and 10) : ";
   cin >> guess;

   //*** Display the number and the user's guess ***
   cout << "The number was " << number << endl;
   cout << "Your guess was " << guess << endl;
}
```

Activity 5.15

Start a new project called *GuessingGame* and recreate the program given in FIG-5.12.

Test the program.

Day of the Week Program

Introduction

Although at this point we know only a modest amount of C++, still there are some interesting programs we can create.

For example, there is a mathematical formula (known as **Zeller's Congruence**) which can calculate the day of the week any given date fell/falls on. The formula is based on the Gregorian calender, a calendar system adopted by Great Britain and its Empire (including its US colonies) on the 14th September 1752. So it is only accurate for those territories from that date. However, many other countries adopted the system before Great Britain and some many years later.

The Calculation

The formula for calculating a date's day of the week is:

```
dayofweek = ([2.6M -0.2] + D + Y + [Y/4] + C/4]-2C) modulo 7
```

where:

D	is the day of the month.
M	is a modified month number (Mar = 1.. Dec = 10, Jan = 11, Feb = 12)
modifiedyear	is the 4-digit year (for Jan and Feb months subtract 1 from the actual year)
C	is the 2 most-significant digits of *modifiedyear*
Y	is the 2 least-significant digits of *modifiedyear*
dayofweek	is a number in the range 0 to 6 (0=Sunday, 1=Monday, etc)

Simple, right! Well, maybe a little confusing at first. But let's start by getting the purpose of the variables listed above clear.

If we wanted to find out which day of the week 22 Nov 1963 fell on, then the variables listed above would have the following values:

D = 22
M = 9 (Since March is 1, November is 9)
modifiedyear = 1963
so
C = 19
Y = 63

For the date 5 Jan 2021, the variables' values would be:

D = 5
M = 11
modifiedyear = 2020 (subtracted 1 from actual year because month is Jan)
so

C = 20
Y = 20

Okay, so we've got that bit clear. Now we can have a look at what the actual formula says. Square brackets [ ] indicate that the value is truncated to give an integer result.

For example, using the date 22 Nov 1963

```
        [2.6M - 0.2]
=       [2.6 * 9 - 0.2]
=       [23.2]
=       23
```

No rounding takes places when calculating the integer, meaning that [23.2] still gives a result of 23.

The final bit of the formula that might need a bit of clarification is the term *modulo*. For positive numbers, this really just means the remainder after division (just like the `%` operator). For example,

```
        81 modulo 7
=       4
```

However, for negative values remainder and modulo are different.

```
        -12 % 7
=       -5              (remainder)
```

but

```
        -12 modulo 7
=       2               (-12 is 2 short of an exact division)
```

Implementing Modulo in C++

When you want the `%` operator to act as modulo, we need to add a little to the calculation

If

```
        -12 % 7
=       -5
```

by adding back 7 (the number we divided by)

```
=       -5 + 7
=       2
```

we get the modulo result. Trouble is, if we did this in every case, positive calculations would come out wrong:

```
        81 % 7
=       4
add 7
        4 + 7
=       11              (wrong result)
```

So the last part of the trick is, after adding the 7 to perform the `%7` operation again.

```
        11 % 7
=       4
```

This extra step has no effect if we started out with a negative value:

```
        2 % 7
=       2
```

So, to summarise, to calculate modulo in C++ use the following calculation:

```
(v % b + b )% b
```

where

v is the value on which the modulus operation is to be performed.

b is the number being divided into *v*.

Implementing a Solution

Since this is a slightly longer program than anything we've done before, it will be worth the time to describe the program's algorithm in structured English before attempting to program it in C++:

```
Get the date (day, month and year) in standard form
Calculate M (as given in the formula)
Calculate modifiedyear
Calculate C
Calculate Y
Calculate dayofweek
Display dayofweek
```

Now for the code. The first lines are just the usual routine library details:

```
#include <iostream>
using namespace std;
```

followed by the start of the function *main()*:

```
int main()
{
```

Next we need to declare any variables used by the program.

```
int day,month,year;     // date entered
int M,C,Y,modifiedyear; // calculated values required by formula
int dayofweek;          // day of week on which date falls
```

Now we can implement each part of the structured English solution given above. First we get the date from the program's user:

```
//*** Get date ***
cout << "Enter date in the format dd/mm/yyyy : ";
scanf_s("%d/%d/%d",&day,&month,&year);
```

The next bit is trickier. We need to calculate *M* which for month *Mar* to *Dec* (entered in numeric form as 3 to 12) isn't a problem and can be calculated by the simple statement:

```
M = month - 2;
```

But *Jan* and *Feb* (entered as 1 and 2) present a problem, since *M* needs to be 11 or 12 respectively. But the formula above would give us -1 and 0. To change this to the correct result we would need to add 12 (giving 11 and 12).

Time for some more devious programmer-type thinking. If we take the *month* value and add 9

```
month + 9
```

we'll get a value ranging from 10 (Jan) to 21 (Dec). If we take this value and get the remainder after dividing by 12

```
(month + 9) % 12
```

we'll now get a value as follows: 10 (Jan), 11(Feb), 0 (Mar), 1(Apr), 2 (May)... 9 (Dec).

And if we end by adding 1:

```
(month + 9) % 12 + 1
```

we have exactly the values we need: 11(Jan), 12(Feb), 1(Mar), etc.

So, finally, we have the C++ code for calculating *M*:

```
M = (month + 9) % 12 + 1;
```

Now that we have the correct mindset, calculating the modified year should come a little easier. Normally, the modified year is the same as the year entered:

```
modifiedyear = year
```

but where *M* is 11 or 12, we need to subtract 1 from the entered year to get the modified year. This is easily done if we remember that the division operator (/) gives an integer result when operating on two integer values.

The expression `M/11` will produce a result of 0 if *M* is in the range 1 to 10 but 1 if *M* is 11 or 12. So, the code

```
modifiedyear = year - M / 11;
```

will subtract 1 from *year* only if *M* is 11 or 12 – just what we need for *modifiedyear*.

Now we are about ready to tackle the main calculation. But part of that calculation is

([2.6M -0.2]

so we have to calculate the integer part of 2.6M-0.2 and to do this we can use a `static_cast` conversion:

```
static_cast<int>(2.6 * M - 0.2)
```

Other parts of the expression where integer values are required such as [Y/4] will be handled automatically because the division operator, since it is dealing with two integer values, will return an integer result.

And, finally, we arrive at the line we require for the result:

```
dayofweek = ((static_cast<int>(2.6 * M - 0.2) + day + Y + Y/4 +
↳C/4-2*C)%7 +7)%7;
```

Now we just have to display the result

```
cout << day<<'/'<<month<<'/'<<year<<" fell on a "<<dayofweek
↳<<endl;
```

before finishing the `main()` function with a closing brace. FIG-5.13 lists the complete program with comments added.

FIG-5.13

The DayOfWeek Program

```
//*** Calculate date's day of the week ***
#include <iostream>
using namespace std;

int main()
{
   int day, month, year;   // date entered
   int M,C,Y, modifiedyear; // calculated values required by
                            formula
   int dayofweek;           // day of week on which date falls

   //*** Get date ***
   cout << "Enter date in the format dd/mm/yyyy : ";
   scanf_s("%d/%d/%d",&day,&month,&year);

   //*** Calculate M ***
   M = (month + 9) % 12 + 1;

   //*** Calculate modified year ***
   modifiedyear = year - M / 11;

   //*** Calculate C ***
   C = modifiedyear / 100;

   //*** Calculate Y ***
   Y = modifiedyear % 100;

   //*** Calculate day of week ***
   dayofweek = ((static_cast<int>(2.6 * M -0.2) + day + Y + Y/4 +
   C/4 - 2*C)%7+7)%7;

   //*** Display result ***
   cout << day<<'/'<<month<<'/'<<year<<" fell on a "<<dayofweek
   <<endl;
}
```

Activity 5.16

Start a new project called *DayOfWeek* and recreate the program given in FIG-5.13.

Test the program with the following dates:

22/11/1963 (should give *dayofweek* as 5)
26/10/1961 (*dayofweek* is 4)

If dates are given in a different format in your part of the world (eg. month/day/year) you might like to try changing the program to suit.

The only disappointing part of the final program is that the day of the week is shown as a number rather than given as a word. We'll revisit the program later and correct this shortcoming.

Solutions

Activity 5.1

No solution required.

Activity 5.2

a) << should be >>

b) , should be >>

Activity 5.3

Input02 version 1:

```
#include <iostream>
using namespace std;

int main()
{
  int no;
  //*** Read number ***
  cout << "Please enter an integer value : ";
  scanf_s("%d",&no);

  //*** Display number ***
  cout << "Number entered was " << no << endl;
}
```

Make sure you added comments.

*Input02 v*ersion 2 - `scanf()` line changes to:

```
scanf_s("%x",&no);
```

The number will still display as base 10.

*Input02 v*ersion 3 - `scanf()` changes to:

```
scanf_s("%o",&no);
```

Activity 5.4

Code for *GetDate*::

```
#include <iostream>
using namespace std;

int main()
{
  int day, month, year;

  //*** Get date ***
  cout << "Please enter date (dd mm yyyy):";
  scanf_s("%d %d %d", &day, &month, &year);

  //*** Display date ***
  cout << day << '/' << month << '/' << year <<
endl;
}
```

The program works correctly irrespective of the number of whitespace characters inserted between each value.

Activity 5.5

The `scanf_s()` line of the *GetDate* should be changed to:

```
scanf_s("%d/%d/%d", &day, &month, &year);
```

The program only performs correctly with a single forward slash is entered between values. Otherwise *month* and *year* are assigned apparently random values.

Activity 5.6

Code for *BMI*:

```
#include <iostream>
using namespace std;

int main()
{
  int age;
  float weight, height;
  float bmi;              //body mass index

  //*** Read weight and height ***
  cout << "Enter your weight (kilos) : ";
  cin >> weight;
  cout << "Enter your height (metres) : ";
  cin >> height;

  //*** Calculate body mass index ***
  bmi = weight / (height * height);

  //*** Display results ***
  cout << "Your height : " << height<<endl;
  cout << "Your weight : " << weight<<endl;
  cout << "Your Body Mass Index : "<<bmi<<endl;
  cout << "A BMI of over 25.0 indicates that you are
  overweight\n";
}
```

Activity 5.7

Code for *DecHex*:

```
#include <iostream>
#include "iomanip"
using namespace std;

int main()
{
  int no;

  //*** Read in number ***
  cout << "Enter integer value : ";
  cin >> no;
  //*** Display number in decimal and hexadecimal
  ***
  cout << "Number (decimal)      : " << no << endl;
  cout << "Number (hexadecimal) : " << setbase(16)
    << no << setbase(10) << endl;
}
```

Activity 5.8

In the project *BMI*, the lines

```
cout << "Your height : " << height<<endl;
cout << "Your weight : " << weight<<endl;
cout << "Your Body Mass Index : "<<bmi<<endl;
cout << "A BMI of over 25.0 indicates that you are
overweight\n";
```

should be replaced by the single line:

```
printf("Your height : %.2f\nYour weight : %.2f\nYour
Body Mass Index : %0.1f\n",height,weight,bmi);
```

Activity 5.9

Code for *Times*:

```
#include <iostream>
#include "iomanip"
using namespace std;

int main()
{
  int mins1, secs1;  //First time
  int mins2, secs2;  //Second time
  int anshrs, ansmins, anssecs ; //Total time
  int totalsecs, totalmins, carry ; // Intermediate
                                      results

  //*** Read first time ***
  cout << "Enter first time (mm ss)  : ";
  cin >> mins1 >> secs1;
  //*** Read second time ***
  cout << "Enter second time (mm ss) : ";
  cin >> mins2 >> secs2;
  //*** Add the seconds ***
  totalsecs = secs1 + secs2;
  //*** Calculate carried minutes ***
  carry = totalsecs / 60;
```

```
    //*** Calculate remaining secs ***
    anssecs = totalsecs % 60;
    //*** Calculate total minutes ***
    totalmins = mins1 + mins2 + carry;
    //*** Calculate hours ***
    anshrs = totalmins / 60;
    //*** Calculate remaining minutes ***
    ansmins = totalmins % 60;
    //*** Display times ***
    cout << "Time 1 : "<<setw(4)<<mins1<<":"<<setw(2)
    ↳<<secs1<<endl;
    cout << "Time 2 : "<<setw(4)<<mins2<<":"<<setw(2)
    ↳<<secs2<<endl;
    cout << "Total  : "<<anshrs<<":"<<setw(2)<<ansmins
    ↳<<":"<<setw(2)<<anssecs<<endl;
}
```

Activity 5.10

The program does not compile. The term `cout` is no longer recognised by the compiler.

Activity 5.11

The program now compiles and executes.

Activity 5.12

a) `sum += no;`
b) `count -= 6;`
c) `ans *= (mark / 7);`
d) `total %= (previous - current);`

Activity 5.13

line 1: *no1* = 1, *no2* = 10, *answer* = unknown value

line 2: *no1* = 2, *no2* = 9, *answer* = 20 (calculated before)

Activity 5.14

Code for *CharBefore*:

```
#include <iostream>
using namespace std;

int main()
{
  char letter;

  //*** Read character ***
  cout << "Enter character : ";
  cin >> letter;

  //*** Display "previous" character ***
  cout << "Preceding character is : " << --letter
  ↳<<endl;
}
```

Activity 5.15

No solution required.

Activity 5.16

No solution required.



Selection

In this Chapter:

- ❑ The `if` Statement
- ❑ Boolean Expressions
- ❑ Boolean Operators
- ❑ Variable Lifetime and Scope
- ❑ Nested `if` Statements
- ❑ The Conditional Assignment Statement
- ❑ The `switch` Statement
- ❑ Testing Selection Statements

Introduction

As we saw in the structured English of Chapter 1, many algorithms need to perform an action only when a specified condition is met. The general form for this statement was:

```
IF condition THEN
      action
ENDIF
```

Hence, in our guessing game we described the response to a correct guess as:

```
IF guess = number THEN
     Player 1 says "Correct"
ENDIF
```

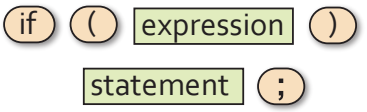
As we'll see, C++ also makes use of an `if` statement to handle such situations.

The *if* Statement

The simplest form of the `if` statement in C++ takes the form shown in FIG-6.1.

FIG-6.1

A Simple `if` Statement



where:

expression is normally a Boolean condition but may be otherwise.

statement is any executable C++ statement. This statement will be executed only if *expression* evaluates to *true* otherwise it will be ignored.

Expression

Generally, *expression* will be one in which the relationship between two quantities is compared. For example, the expression

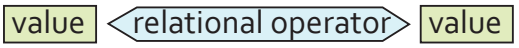
```
no1 < 0
```

will be *true* if the content of the variable *no1* is less than zero (i.e. negative).

When the expression is a simple Boolean condition, it will have the form shown in FIG-6.2.

FIG-6.2

A Boolean Condition



where:

value maybe a constant, variable or an arithmetic expression.

relational operator is one of the operators given in FIG-6.3.

FIG-6.3

Relational Operators

Relational Operators	
Symbol	**Meaning**
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

Notice that the *equal to* operator is constructed from two equal signs, not just a single equals sign. The reason for this is that we are already using a single equals sign in the assignment statement. Also note that *not equal to* makes use of the exclamation sign (`!`). This is the symbol for NOT in C++.

Activity 6.1

Which of the following are not valid Boolean expressions and why?

You may assume all variables have been declared.

a) `no1 < no2` b) `x = y` c) `m > x+2`
d) `8 != no3` e) `no3 =! 12` f) `v1 >= v2`

One of the commonest mistakes a newcomer (and some not so newcomers) can make is to use `=` when they should be using `==`. So watch out for this!

The program in FIG-6.4 reads in an integer value and displays the word *negative* if the value entered is less than zero.

FIG-6.4

Checking for a Negative Value

```
#include <iostream>
using namespace std;

int main()
{
   int no1{};
   // *** Get value ***
   cout << "enter a value ";
   cin >> no1;
   // *** IF no1 negative THEN display "negative" ***
   if (no1 < 0)
      cout << "negative" << endl;
   // *** Display the number entered ***
   cout << "The number is " << no1 << endl;
}
```

Activity 6.2

Create a new project called *Negative* and type in and test the program given in FIG-6.4. Run the program twice, once with a negative value and once with a positive one.

Translating English into Code

It's easy enough to ask a Boolean type question in English (just as we did back in

Chapter 1), but sometimes it takes a little cunning to come up with the C++ equivalent.

For example, it's easy enough to translate the descriptive line

```
IF  the first value is at least twice the second THEN
```

into

```
if (v1 >= v2 * 2)
```



v1 and *v2* hold the values being tested.

but how would we code

```
IF the value is an odd number THEN
```

This is where we have to start thinking with the guile of a programmer. We know we can get the remainder of a division using the `%` operator and that any even number would have a remainder of exactly 0 when divided by 2, so the solution to our problem is



We can't assume that odd numbers always have a remainder of 1, because odd negative numbers have a remainder of -1.

```
if ( v1 % 2 != 0)
```

Activity 6.3

Translate each of the following English conditions into C++ code:

a) IF the first value is no more than 6 greater than the second.
b) IF the integer value is exactly divisible by 3.
(To make things interesting, try this one without using the % operator.)

Multiple Lines

Notice that the syntax of the `if` statement, as described previously, allows for only a single statement after the expression. This is in contrast to the structured English version we saw back in Chapter 1 where we could have as many statements as we wished nested between the terms IF and ENDIF.

Since C++ doesn't use the term ENDIF, we need some other method of telling the compiler that there is to be more than one statement in the `if` structure. This is achieved by using braces to enclose the multiple statements. For example, the structured English statement

```
IF BMI  is too high THEN
    Calculate weight loss required
    Display weight loss required
ENDIF
```

translates into



The braces are aligned vertically and the statements within the braces are indented.

This layout maximises your chances of noticing missing braces.

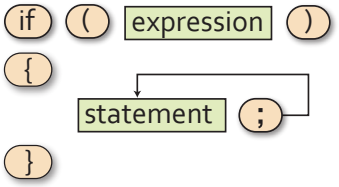
```
if (bmi >= 25)
{
   double newweight = 24.9 * height * height;
   double weightloss = weight - newweight;
   cout << " You need to lose "<<weightloss<<" kilos"<<endl
}
```

Notice how the braces are used to show that all three statements are part of the `if` structure.

So this latest version of the `if` statement has the format shown in FIG-6.5.

FIG-6.5

`if` with Multiple Statements



Activity 6.4

Write a program, *CircleArea*, which reads a value representing the radius of a circle.

The program should, if the number entered is not negative, calculate and display the area of a circle with this radius.

Where a negative value is entered the program should terminate without performing any calculations.

Variable Scope

We are free to declare a variable just about anywhere in a program as long as it is declared before it is used in another statement. But the position at which we make such a declaration has consequences.

In the solution to Activity 6.4, the variable *area* was declared at the start of the program and used within the `if` statement:

```
using namespace std
int main()
{
   const double PI {3.14159265};
   double radius;
   double area;

   //*** Read radius ***
   cout << "Enter radius of circle : ";
   cin >> radius;
   //*** IF not negative, calc and display area ***
   if (radius >= 0)
   {
      area = PI * radius * radius;
      cout << "The area of a circle whose radius is "
      ↳<< radius << " is " << area << endl;
   }
}
```

Activity 6.5

In your current version of *CircleArea*, add the statement

```
cout << area << endl;
```

between the two final closing braces.

Run the program, using first a positive number and then a negative number.

Running this version of *CircleArea* displays a value for the variable *area* even when a negative value is entered (what appears on the screen is the random value initially stored in *area* – this may be the same value created by the previous run).

But since we may declare *area* at any point in the program before it is actually used, we could have declared it within the `if` statement as shown below:

```
int main()
{
   const double PI = 3.14159265;
   double radius;

   //*** Read radius ***
   cout << "Enter radius of circle : ";
   cin >> radius;
   //*** IF not negative, calc and display area ***
   if (radius >= 0)
   {
      double area = PI * radius * radius;
      cout << "The area of a circle whose radius is "
      ↳<< radius << " is " << area << endl;
   }
   cout << area << endl;
}
```

Activity 6.6

Modify *CircleArea* so that the variable area is declared within the `if` statement (as shown above).

What happens when the program is compiled?

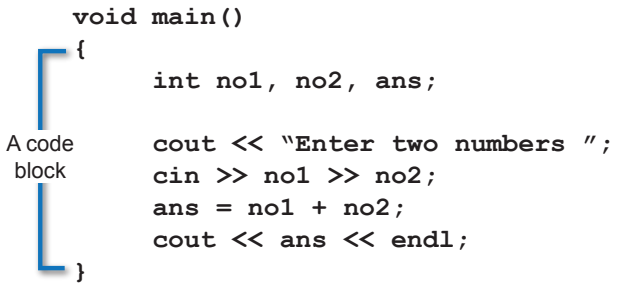
As we saw from Activity 6.6, trying to compile the latest version of *CicleArea* produces an error `area : undeclared identifier`

To explain why the error message has appeared, let's start with the definition of a **block**.

A block is all the code which appears between a set of braces. In a very simple program, all of the code is in a single block marked by the braces at the start and end of `main()` (see FIG-6.6).

FIG-6.6

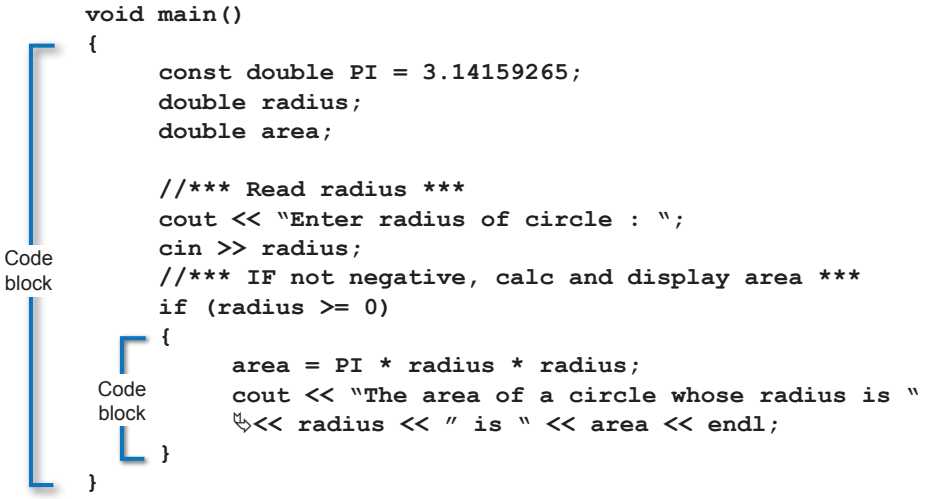
A Code Block



In the case of *CircleArea*, we have two blocks, one being within the `if` structure (see FIG-6.7).

FIG-6.7

An Outer and Inner Block




```
void main()
{
      const double PI = 3.14159265;
      double radius;
      double area;

      //*** Read radius ***
      cout << "Enter radius of circle : ";
      cin >> radius;
      //*** IF not negative, calc and display area ***
      if (radius >= 0)
      {
            area = PI * radius * radius;
            cout << "The area of a circle whose radius is "
            ↳<< radius << " is " << area << endl;
      }
}
```

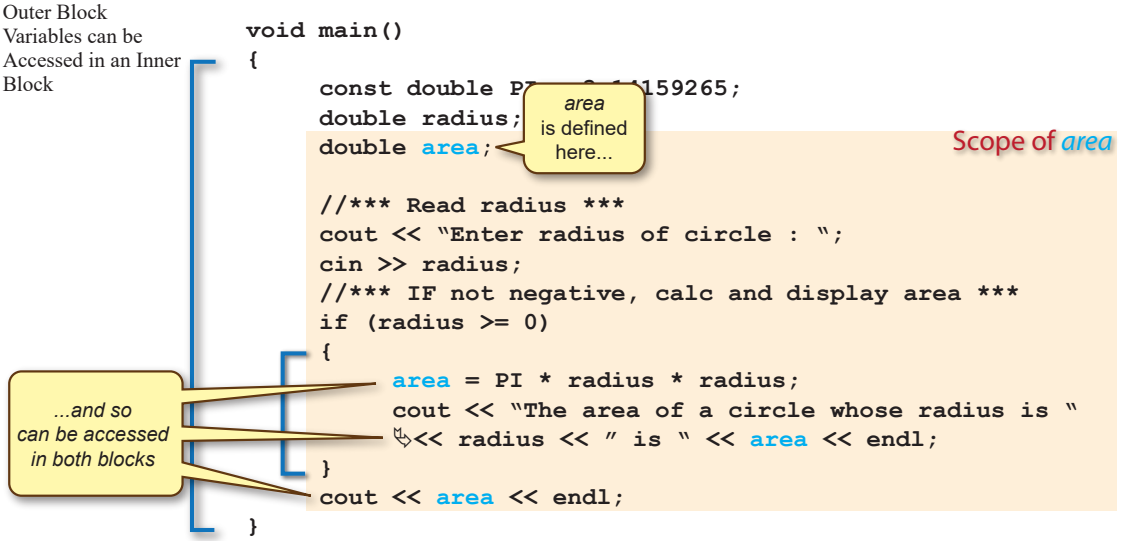
In the code of FIG-6.7, the complete program block encloses the smaller block within the `if` structure. In this situation, the larger, encompassing block is known as the **outer block** and the smaller one, the **inner block**.

When a variable is declared, it can be accessed only in the block in which it is declared and any inner block. This is known as the **scope** of the variable.

This means that when we declare the variable *area* at the start of *CircleArea*, it is available within both blocks of the program (see FIG-6.8).

FIG-6.8

Outer Block Variables can be Accessed in an Inner Block




```
void main()
{
      const double PI = 3.14159265;
      double radius;
      double area;

      //*** Read radius ***
      cout << "Enter radius of circle : ";
      cin >> radius;
      //*** IF not negative, calc and display area ***
      if (radius >= 0)
      {
            area = PI * radius * radius;
            cout << "The area of a circle whose radius is "
            ↳<< radius << " is " << area << endl;
      }
      cout << area << endl;
}
```

However, if we move the declaration of area to the inner block, then our attempt to display its contents at the end of the program becomes invalid because the scope of the variable is now restricted to the inner block (see FIG-6.9).

FIG-6.9

An Inner Block Variable Cannot be Accessed in an Outer Block

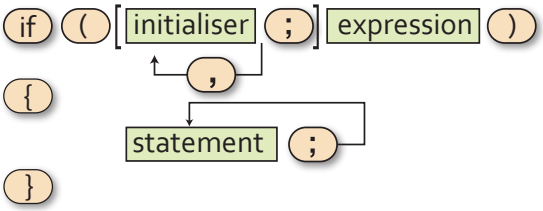
```
void main()
{
      const double PI = 3.14159265;
      double radius;


      //*** Read radius ***
      cout << "Enter radius of circle : ";
      cin >> ra
      //*** IF                ve, calc and display area ***
      if (radiu
      {
            double area = PI * radius * radius;
            cout << "The area of a circle whose radius is "
            << radius << " is " << area << endl;
      }
      cout << area << endl;
}
```


In C++17, the format of the `if` statement has been expanded to allow for an initialisation statement before the expression (see FIG-6.10).

FIG-6.10

`if` with Initialiser Option



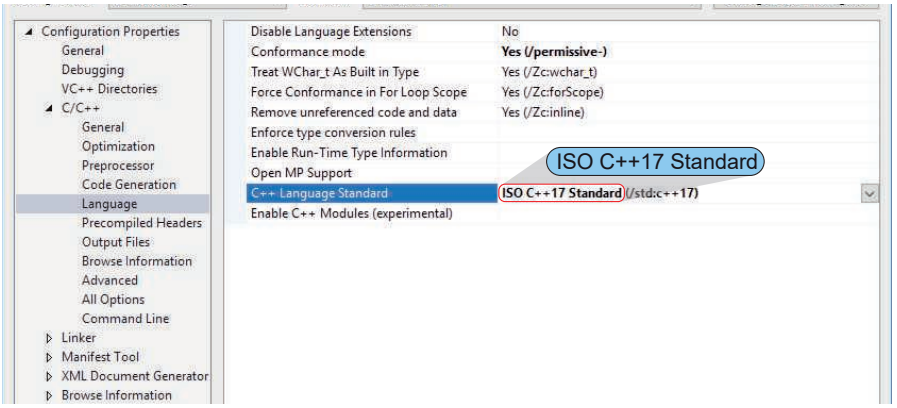
The initialisation section allows us to declare variables that may be used solely within the `if` statement's code block or to set existing variables to some starting value. For example, using our previous example, we might declare the variable *area* in the initialisation section:

```
if(double area{},radius >= 0)
{  ...
```

If this format of the `if` statement does not compile on your hardware, check that the C++17 version of the language is selected. In Visual C++, this requires the project's property page's *C++ language standard* set to C++17 (which, at the time of writing, is not the default setting). The required setting is shown in FIG-6.11.

FIG-6.11

Selecting the C++17 Language Standard



If required, we could use the comma separator to initialise several variables within the initialiser section:

```
if(int {0}, int b{12}, total = 0; x > y)
```



The variable *total* is assumed to have been declared earlier in the code.

Using the initialiser option is down to personal preference. Many might find it clearer to declare variables which are local to the `if` block within the braces of the block itself (as we did earlier with *area*).

Testing a Simple *if* Statement

We already know it is important to test any program we create. When the piece of code being tested contains a single `if` statement, then we need at least two sets of test data. One of these will be designed to ensure that the Boolean condition specified in the `if` statement is *true*; the other test data set will make the condition *false*. For example, if a program contains the lines

```
cin >> no1;
if (no1 < 0 )
   ...
```

then our test data set would consist of a value for *no1* which ensured that the condition `no1 < 0` was *true* ; and the second value for *no1* which ensured `no1 < 0` was *false*. A possible test data table for this code is shown in FIG-6.12.

FIG-6.12

Test Data for the Code

No results are shown since the remainder of the code is unknown.

Test Data	Expected Results
no1 = -5	.....
no1 = 7	.....

A third test that is useful is to make the left and right side of the Boolean expression equal (see FIG-6.13).

FIG-6.13

Test Data Including Equality

Test Data	Expected Results
no1 = -5	.....
no1 = 7	.....
no1 = 0	.....

Of course, sometimes the nature of the Boolean expression is such that the test for equality is part of the original two test values. For example, if we were to test the condition

```
if (no1 == no2)
```

then *no1*'s value being equal to *no2*'s will be required as one of our first two test values.

Activity 6.7

Create a test data table for the program you wrote in Activity 6.4. Remember to include the expected results.

Compound Conditions

Back in Chapter 1 when we discussed the concept of conditions – or Boolean expressions – we soon found the need for quite complex conditions involving the use of the terms AND and OR. For example, we described the logic used in the circuits of a proximity light as

IF it's dark AND motion is detected THEN

and the logic for going to jail in a game of Monopoly as

IF player lands on *Go To Jail* OR player has thrown 3 pairs in a row THEN

Of course, C++ has the same ability to make use of the AND and OR operations to create a compound Boolean expression. However, rather than use the word AND, C++ uses the symbol pair `&&`; OR is represented by `||` . These are known as **Boolean operators**.

For example, let's say we want to check if two variables, *no1* and *no2*, both contain even numbers then we would start our `if` statement with

```
if (no1%2 == 0 && no2%2 == 0)
```

On the other hand, to check if at least one of the two variables held an even value we would use the OR option:

```
if (no1%2 == 0 || no2%2 == 0)
```

Activity 6.8

a) Create a project, *Check01*, which reads in an integer number and, if the number lies between 1 and 12, displays the message "*Value accepted*".

b) Create a project, *Check02*, which reads in an integer value and checks if the value is equal to 9 or 14. If it is, the program should display "*OK*".

We can combine as many simple Boolean expressions as required using the `&&` and `||` operators. For example, we could check that *no1* lies between 1 and 100, but is not equal to 79 using the code

```
if ( no1 >= 1 && no1 <= 100 && no1 != 79)
```

The final Boolean operator available in C++ is NOT (again, we looked at this back in Chapter 1). In C++ NOT is represented by the exclamation mark (`!`). So, while we can check if *no1* is less than zero with the code,

```
if ( no1 < 0)
```

we can check if *no1* is not less than zero using

```
if( !(no1 < 0))
```

Notice the need for the original Boolean expression to be enclosed in parentheses. This is because the `!` operator has a lower priority than the `<` operator.

Of course, for this example, the easiest thing to have done would have been to write

```
if ( no1 >= 0)
```

However, when it comes to more complex conditions, using the `!` operator can simplify finding the inverse of a condition. For example,

```
if ( no1 >= 1 && no1 <= 12)
```

checks that *no1* is in the range 1 to 12 and

```
if ( !( no1 >= 1 && no1 <= 12))
```

checks that *no1* is not in the range 1 to 12. The above expression is possibly easier to arrive at than the alternative:

```
if ( no1 < 1 || no1 > 12)
```

Short-Circuit Evaluation

When a condition includes an `&&` or `||` operator, C++ only evaluates the minimum number of conditions in the expression (working from left to right). For example, let's assume a program contains the following code

```
int num{23};
if (num < 0 && num%2 == 0)
```

As the program executes, it evaluates the condition `num < 0` as *false*, Now, since both `num < 0` and `num%2 == 0` have to be *true* for the complete expression within the parentheses to evaluate to *true*, and since the program has already determined that `num < 0` is *false*, it doesn't bother to evaluate `num%2 == 0`, knowing that the overall expression must be *false*. This is known as **short-circuiting**

Most of the time short-circuiting is a detail that need not worry us. However, should the second expression modify the contents of a variable as in the line

```
if (num < 0 && ++num > -1)
```

then, because of the short-circuiting, the expression `++num > -1` will not be evaluated and hence the variable *num* will not be incremented.

Note that if the compound condition involves the `&&` operator, the second condition will only be tested if the first is found to be *true*. If the `||` operator is used, the second condition will only be tested if the first is *false*.

Testing

When it comes to testing a compound Boolean expression, we need to test each simple condition within that expression with data that creates *true* and *false* results. We also need to ensure that these *true* and *false* results are created in every possible combination within the compound expression. For example, if we were to create test data for the line

```
if (no1 > 6 && no2 < 5)
```

then our data would need to result in at least four combinations so we could create every permutation (see FIG-6.14).

FIG-6.14

Test Condition Combinations

Condition Combinations	
no1 > 6	no2 < 5
false	false
false	true
true	false
true	true

Activity 6.9

Create a set of possible test values for the following `if` statement:

```
if (no1 > 6 && no2 < 5)
```

There will be times when all combinations are not possible simply because the conditions specified in the expression cannot both be *true* or they cannot both be

false. For example, we checked that a number lay outside the range 1 to 12 with the line

```
if ( no1 < 1 || no1 > 12)
```

When we come to testing this statement, it will be impossible for both conditions to be *true* at the same time since the value in *no1* cannot be simultaneously <1 and > 12. In a case such as this the conditions are said to be **mutually exclusive**.

Boolean Variables

As we noted earlier, a Boolean variable is one which takes on a value of either *true* or *false*. We assign a Boolean variable one of these values with a line such as

```
bool maybe = true;
```

but we can also assign it the result of a Boolean expression such as

```
maybe = age > 65;
```

With this last example, the value of the expression `age > 65` will be calculated as *true* or *false* and that value then assigned to *maybe*.

When using a Boolean variable in an `if` statement's condition, although we can write a line such as

```
if (maybe == true)
```

it is equally valid (and more usual) to simply write

```
if(maybe)
```

and, rather than

```
if (maybe == false)
```

we would normally write

```
if(!maybe)
```

Strange Things about Boolean Values

Since C++ will automatically convert a Boolean value to a number (*true* converting to 1 and *false* to 0), this allows us to do some interesting things. For example, the code

```
cin >> no1;
int no2 {no1 == 12};
```

will set *no2* equal to 1 if *no1* has the value 12 and set *no2* equal to 0 if *no1* has any value other than 12.



It will get it wrong for the year 2100! But you could expand the condition to handle that.

A more useful example might be

```
daysinfebruary = 28 + (year % 4 == 0);
```

which will set *daysinfebruary* to 28 in a non leap year, and 29 in leap years.

Non-Boolean Expressions

It is this ability to convert between types that allows the expression given in an `if` statement to be of a non-Boolean form. So, for example, it is legal to write a statement such as:

```
if ( no1 - 6)
```

Instead of a Boolean expression we have an arithmetic expression. But C++ determines how this expression is to be treated by simply reducing it to an integer value and then converting that integer to a `bool` result.

If you recall from Chapter 3 that all non-zero integer values convert to *true* and zero itself converts to *false*, then we can see that the above expression will be deemed to be *true* where *no1* has any value other than 6. Only when *no1* is equal to 6 do we get a *false* result (since 6 - 6 gives us zero).

Even an assignment statement may be used as the `if`'s expression:

```
if (no2 = 12)
```

This will assign the value 12 to *no2* before converting that value to a `bool` result of *true*.

Code like the example above often results from one of the commonest mistakes beginners make when writing a condition: using the single equals symbol (meaning assignment) when they intended to use the double equal symbols (meaning test for equality). Since both are valid expressions, the compiler won't flag an error, but we won't get the results we're expecting from our code!

A `char` value will also convert to a Boolean. This is done by simply converting the character's binary code to an integer value. The only character that is equivalent to *false*, is the null character, '\0', all others equate to *true*.

Activity 6.10

Which of the following expressions will be considered to be *true*?
(assume *no1* = 7 and *no2* = -5)

a) `no1 + no2` b) `no1 - 7` c) `0.6`
d) `no1 - no2` e) `'\0'` f) `'A' - 65`

The Boolean operators (`&&`, `||` and `!`) can also be used when the `if`'s expression does not contain Boolean conditions. For example, we can write

```
if(no1 || no2)
```

which will evaluate to *true* if *no1* has a non-zero value OR *no2* has a non-zero value. When both variables contain zero, the expression is *false*.

Testing

Using non-Boolean expressions in an `if` statement has no effect on the approach to creating test data. We still want to use a value which evaluates to *true* (non-zero) and another which is *false* (zero).

if...else

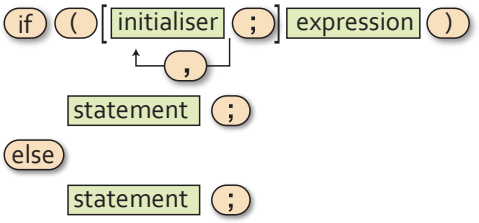
C++'s `if` statement can be extended to include an `else` section - just like the structured English version we saw back in Chapter 1. The default is to allow only a single statement in the `else` section:

```
if (initialiser; expression)
   statement;
else
   statement;
```

With an `else` option included, the simplest syntax is shown in FIG-6.15.

FIG-6.15

Single Statement `if .. else` Structure



If we wanted to display either "Positive" or "Negative" depending on the value of the variable *no1*, we could write

```
if (no1 < 0)
   cout << "Negative";
else
   cout << "Positive"; //Zero treated as positive
```

Activity 6.11

Create a project, *EqualityTest*, which reads in two values and displays either "*Equal*" if the two values are equal, and "*Not equal*" if the values are different.

Test your program with two sets of test data ensuring both messages appear under the correct conditions.

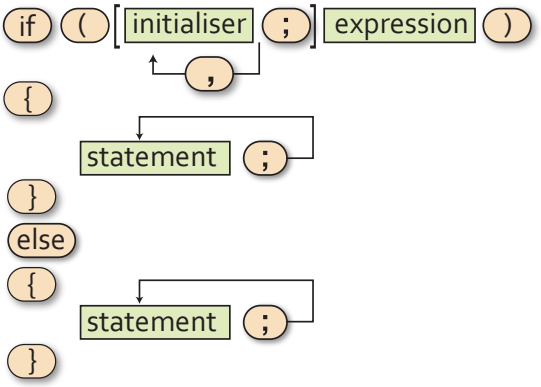
By using parentheses after the word `else`, we can include as many statements as required:

```
if (initialiser;expression)
   statement;
else
{
   statement;
   statement;
      ...
}
```

When we need multiple statements in each section of the statement, then we arrive at the form shown in FIG-6.16.

FIG-6.16

Multiple Statement `if .. else` Structure



Of course, we can combine the two forms, having only a single statement in the first section of the `if` and multiple statements after the `else`, or vice versa.

Activity 6.12

Reload *CircleArea* and modify it so that when a negative value is entered, the message "Invalid radius" is displayed.

Testing

Including an `else` section in an `if` statement makes no difference to the creation of test data. We still need to use values to give both *true* and *false* results.

Multiple *if* Statements

Obviously, we are free to have as many `if` statements in a program as we desire. Each one will have its expression tested and, if found to be *true*, will have the statement(s) it contains executed.

The program in FIG-6.17 reads in a number and does the following:

Displays the message "Negative" if the number is less than zero.
Displays "Even" if the number is exactly divisible by 2.
Displays "Near zero" if the number lies between -3 and +3.

Notice that all three situations could arise for a single number, for example, -2 which is negative, even, and near zero.

FIG-6.17

Using Multiple *if* Statements

```
#include <iostream>
using namespace std;


int main()
{
   int no1;

   //*** Read number ***
   cout << "Enter an integer value : ";
   cin >> no1;
   //***IF negative THEN display message ***
   if ( no1 < 0)
      cout << "Negative\n";
   //*** IF even THEN display message ***
   if ( no1 % 2 == 0)
      cout << "Even\n";
   //*** IF between -3 and 3 THEN display message ***
   if ( no1 >= -3 && no1 <= 3)
      cout << "Near zero\n";
}
```

Activity 6.13

Create a new project, *Categorise*, containing the code given in FIG-6.17.

Create a set of test data for the program.

Run the program with each test data value in your table.

Activity 6.14

Reload your *BMI* program you created in Activity 5.6 and use four `if` statements so that one of the following messages is displayed in place of the final `cout` statement:

"Underweight" is displayed if the *bmi* value is less 18.5;
"Normal" if it is >= 18.5 and < 25;
"Overweight" if it is >=25 and <30;
"Obese" if >= 30.

Are the `if` statements mutually exclusive (can no more than one of them be true at the same time)?

Nested *if* Statements

One criteria of a good program is efficient code. In the solution given for Activity 6.14 there are four `if` statements which, between them, test four mutually exclusive conditions. And yet the code is executed in such a way that when the first condition (`bmi < 18.5`) is *true*, the conditions in the three other `if` statements are still checked even though we know none of them can be *true*.

A better way to deal with a set of mutually exclusive conditions is to create a **nested if** structure. In this setup, the second `if` statement appears in the `else` section of the first. So, whereas before we had written

```
if (bmi < 18.5)
   cout << "Under weight\n";
if (bmi >= 18.5 && bmi < 25)
   cout << "Normal weight\n";
```

we would now rewrite the code as

```
if (bmi < 18.5)
   cout << "Underweight\n";
else
   if (bmi < 25)
      cout << " Normal weight\n";
```

Notice that the second `if` statement, because of its position inside the `else` section of the first `if`, is only executed when the first condition, `bmi < 18.5,` is *false*. Another consequence of the repositioning is that the second `if`'s condition has been simplified. We know that only when *bmi* has a value of 18.5 or above can the second `if` statement be executed, so the lower limit test of this second `if` statement is no longer required.

If we now add an `else` section for the second `if` – a section of code that can only be arrived at when *bmi* >= 25 – it can contain the third `if`:

```
if (bmi < 18.5)
         cout << "Underweight\n";
      else
         if (bmi < 25)
            cout << " Normal weight\n";
         else
            if (bmi < 30)
               cout << "Overweight\n";
```

The fourth `if` from the original code can now be replaced by a third `else` to handle

all *bmi* values of 30 and above:

```
if (bmi < 18.5)
         cout << "Underweight\n";
      else
         if (bmi < 25)
            cout << " Normal weight\n";
         else
            if (bmi < 30)
               cout << "Overweight\n";
            else
               cout << "Obese\n";
```

Finally we have the complete code for displaying the appropriate message. It is shorter than the original code and has less conditions. It will also be more efficiently executed since as soon as one of the conditions is found to be *true* and the associated `cout` statement executed, the remaining parts of the code will be bypassed.

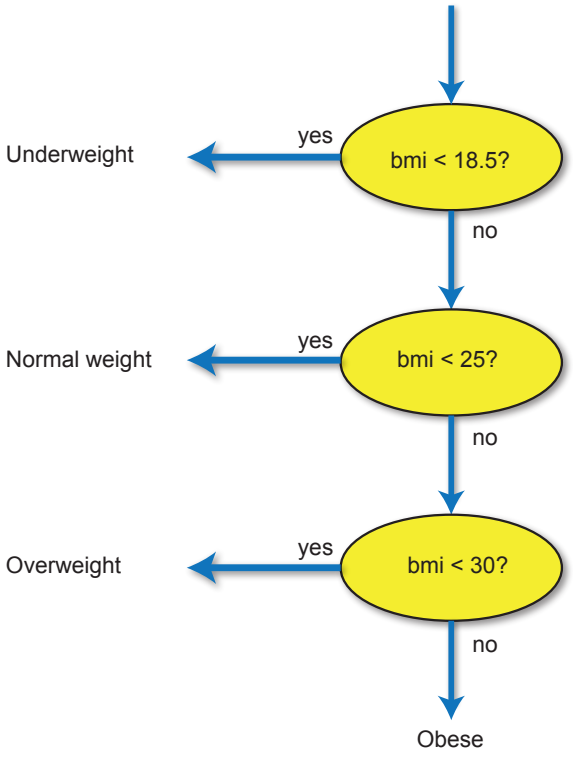
This is one time when the indentation rules are sometimes changed, so instead of using the layout shown above, we modify the code layout to be

```
if (bmi < 18.5)
   cout << "Underweight\n";
else if (bmi < 25)
   cout << " Normal weight\n";
else if (bmi < 30)
   cout << "Overweight\n";
else
   cout << "Obese\n";
```

Sometimes imagining nested `if`s as a road containing a sequence of Y junctions (see FIG-6.18) helps us understand the effect of the code.

FIG-6.18

Nested `if` Statements Visualised



Activity 6.15

Reload your *BMI* program and replace the four separate `if` statements with a nested structure.

Activity 6.16

A bank charges/pays interest on an account's balance at the end of each month. For overdrawn accounts it charges 0.5% interest; for balances which are in credit, but below $2000.00, it pays 0.1% interest and for balances of greater than $2000.00 it pays 0.2% interest.

Create a new project, *BankInterest*, which reads in the balance in an account and displays the interest charged/paid for the month. Use a nested `if` structure in your program.

Activity 6.17

Modify your *GuessingGame* project which you created in Activity 5.15 so that it displays one of the following messages relating to the guess entered

Guess too low
Guess too high
Guess correct

Matching the *else* Keyword to *if*

Since the `else` section of an `if` statement is optional, obviously we are at liberty to omit it. This can cause problems when we use nested `if` statements.

For example, let's assume that we have a program that reads in the gender (entered as *M* or *F*) and age of a company's employees. To display the gender in full we could write

```
if (gender == 'M')
   cout << "Male\n";
else
   cout << "Female\n";
```

Now let's change things slightly and, in the case of males, only display a message if they are over 65 years old:

```
if (gender == 'M')
   if(age > 65)
      cout << "Retired male\n";
else
   cout << "Female\n";
```

This setup creates a problem, because the `else` section will no longer be linked to `if (gender == 'M')` but is now linked to `if (age > 65)` since an `else` clause automatically links to the closest `if` not already associated with an `else`. To reflect this association, the correct indentation for our code would be

```
if (gender == 'M')
   if(age > 65)
      cout << "Retired male\n";
   else
      cout << "Female\n";
```

Left like this, we would get the word *Female* displayed only for a male who was 65 years old or less!

To force the `else` to link to an earlier `if` statement, we need to enclose the second `if` structure in braces:

```
if (gender == 'M')
{
   if(age > 65)
      cout << "Retired male\n";
}
else
   cout << "Female\n";
```

This cuts the second `if` statement off from the `else` and forces the compiler to look for an earlier `if` statement to which the `else` clause can be linked.

Activity 6.18

Create a new project, *Ages*, to read in gender and age values.

Try out the two versions of "Retired male" code shown above, observing the results for males below and above 65 and for females.

The Conditional Assignment Statement

When we need to assign one of two values to a variable, the most obvious way of achieving this is to use an `if` statement:

```
if (x < 0)
   y = 12.9;
else
   y = 0.4;
```

C++ offers an alternative to this approach with its **conditional assignment** statement. The statement starts of with like a normal assignment statement, identifying the variable to be assigned a value and the assignment symbol:

```
y =
```

but then, instead of the value we wish to assign to the variable, a condition is given:

```
y = (x < 0)
```

This condition will be used to decide which of two possible values is to be assigned to the variable *y*. Next we add a question mark:

```
y = (x < 0) ?
```

and the value to be assigned if the specified condition is *true*:

```
y = (x < 0) ? 12.9
```

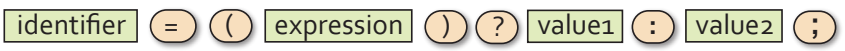
Finally, the statement concludes with a colon and the value to be assigned if the condition is *false*:

```
y = (x < 0) ? 12.9 : 0.4;
```

The overall format of the conditional assignment statement is shown in FIG-6.19.

FIG-6.19

Conditional Assignment



where:

identifier	is a declared variable.
expression	is normally a Boolean expression but may be an arithmetic expression or any other statement that returns a value. This will be short-circuited when a compound condition is used.
value1, value2	are constants, variable names, or arithmetic expressions.

Each of the following conditional assignments are valid:

```
no1 = (no2 > 12) ? 23 : 0;
x = (y != z) ? m : n;
float w = (c == 9) ? m+n : m-n;
```

Activity 6.19

Use a conditional assignment statement to change the sign of the contents of a variable, *x*, if it contains a negative value.

Where *value1* or *value2* are expressions, only the value that is actually to be assigned is calculated, so it is best not to alter the contents of any other variable in these expressions. For example, the line

```
int num2 = (num1 < 0)?-num1: num1++;
```

will not increment the contents of *num1* if `num1 < 0` is *true*.

Testing

Since the conditional assignment statement contains a condition, the same testing rules apply as for the `if` statement with test values which give *true* and *false* results for the condition. If the condition is a compound one, using `&&` or `||` operators, then the test data must ensure all possible *true*, *false* combinations are tested.

The *switch* Statement

The nested `if` statements structure handles multiple, mutually exclusive selection efficiently but the list of conditions can at times make the code look quite cluttered.

Another method of handling the mutually exclusive options is to use the `switch` statement.

Perhaps the easiest way to understand the `switch` statement is to construct one which can be used in the *DayOfWeek* program (which we created back in Chapter 4) to display the day of the week in text form. The statement begins with the keyword

```
switch
```

This is followed by an parentheses-enclosed expression which will be used to determine the next part of the code to be executed. This expression will not normally be a Boolean condition. Instead, it will be a variable or an arithmetic expression which reduces to an integer or character value. In this case, *expression* will be the variable containing the day of the week in an integer form - *dayofweek*:

```
switch (dayofweek)
```

All that follows within the `switch` statement must be enclosed in braces:

```
switch (dayofweek)
{

}
```

Inside the braces we use the keyword **case**, a specific value which might result from the expression being tested, and a colon:

```
switch(dayofweek)
{
   case 0 :
}
```

Beside this **case** label we specify what code we want executed if *dayofweek* has the same value as that specified beside **case**:

```
switch(dayofweek)
{
   case 0 : cout << "Sunday\n";
}
```

We can have as many statements as we require for any given **case**, so we need to mark the end of the statements for this particular option with the keyword **break**:

```
switch(dayofweek)
{
   case 0:  cout << "Sunday\n";
            break;
}
```

Now all we have to do is add in **case** statements for all the other values that we expect the variable *dayofweek* could contain:

```
switch(dayofweek)
{
   case 0 : cout << "Sunday\n";
            break;
   case 1 : cout << "Monday\n";
            break;
   case 2 : cout << "Tuesday\n";
            break;
   case 3 : cout << "Wednesday\n";
            break;
   case 4 : cout << "Thursday\n";
            break;
   case 5 : cout << "Friday\n";
            break;
   case 6 : cout << "Saturday\n";
            break;
}
```

The **switch** statement is executed by first evaluating the *expression* at the start of the statement, converting it to an integer or character value and then executing the **case** statement whose value exactly matches that of *expression*. All other **case** options before and after the one selected are ignored.

A final variation of the **switch** statement is to include an optional **default** section. The statement(s) given in the **default** section will only be executed if none of the **case** options match the value of *expression* at the start of the **switch** statement.

Although we know *dayofweek* should always be in the range 0 to 6, as good programmers we should really prepare for the possibility of some mistake in our code allowing *dayofweek* to go outside that range and add instructions to handle that situation:

```
switch(dayofweek)
{
   case 0 : cout << "Sunday\n";
            break;
   case 1 : cout << "Monday\n";
            break;
   case 2 : cout << "Tuesday\n";
            break;
   case 3 : cout << "Wednesday\n";
            break;
   case 4 : cout << "Thursday\n";
            break;
   case 5 : cout << "Friday\n";
            break;
   case 6 : cout << "Saturday\n";
            break;
   default: cout << "Invalid value in dayofweek.\n";
            cout << " Value is " << dayofweek << endl;
}
```

Multiple `case` Values

If we need more than one value to take us to the same piece of code within a `switch` statement, then we can list two or more `case` labels beside each other. For example, in the code snippet below

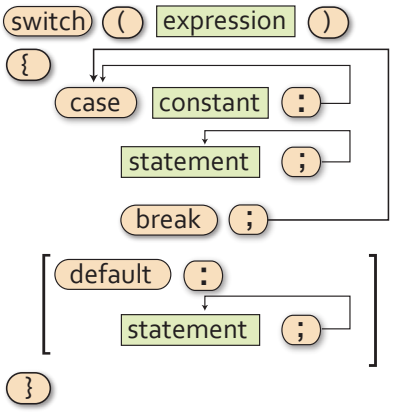
```
switch (character)
{
   case 'a':
   case 'A':   cout << "First letter\n";
               break;
            ...
```

an upper or lower case *A* value in the variable *character* will result in the same piece of code being executed.

We can place as many `case` labels together as we wish, but probably three or four is as many as we would want under most circumstances. The general form of the switch statement is shown in FIG-6.20.

FIG-6.20

switch



where:

expression	is an expression which reduces to an ordinal value (integer, `char`, `bool` or enumerated type).
constant	is a fixed value of the same type as *expression*.

✎ Enumerated types are explained in a later chapter.

statement is any valid C++ statement.

Notice that there is no need for the `break` keyword at the end of the optional `default` section. This is simply because we are already at the end of the `switch` statement. However, the compiler will not complain if we do add the `break` keyword as the final line of this section.

switch Statement Limitations

The `switch` statement does have limitations which means it cannot be used in all the situations that required selection from multiple options.

For example, the `switch` statement cannot be used to display the various categories in our BMI program because each option is based on the variable *bmi* lying within a range of values rather than being a single specific value.

Should the `break` statement at the end of a `case` option be omitted this will not cause a compilation error but when that `case` option is selected for execution, the code of the following `case` option will also be executed. For example, if we had accidentally written

```
case 2 : cout << "Tuesday\n";

case 3 : cout << "Wednesday\n";
         break;
```

without the `break;` line between the two `case` options, then, should *dayofweek* be equal to 2, the words *Tuesday* and *Wednesday* would both be displayed on the screen as the code for both `case 2:` and `case 3:` would be executed.

There are, on rare occasions, reasons to execute the statements associated with two consecutive `case` statements in which case we would deliberately omit the `break` statement. To explicitly show that this omission is deliberate, we can use the term

```
[[fallthrough]]
```

For example,

```
case 'A':
   cout << "Uppercase\n";
   [[fallthrough]]            //No semicolon
case 'a':
   cout << "Vowel\n";
   break;
```

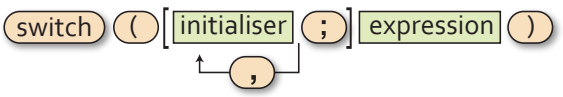
A final restriction of the `switch` statement is that it can only work if we are supplying mutually exclusive `case` options with no requirement to execute more than one of the options supplied (unless these are consecutive options as in the example above).

switch Initialiser

With the introduction of the C++17 standard, the `switch` statement, like `if`, can have an initialiser statement prior to the controlling expression. This means the syntax of the first line becomes that shown in FIG-6.21.

FIG-6.21

switch Format using an Initialiser



where:

initialiser	is any C++ definition, assignment or other statement.
expression	is a variable name or arithmetic expression.

This syntax allows us to create variables which are local to a `switch` control structure such as in the code:

```
switch(int total{}; value)
{
      //total exists only in the switch block
}

//total no longer exists by this point
```

Although the initialiser is normally used to set existing variables or create new ones, it's not restricted to that. For example, it would be quite valid to start with the line

```
switch (cout << "Start switch\n"; letter)
```

which would output the message *Start switch* before determining the value of *letter* and executing the appropriate `case` statement that followed.

Local Variables within a *case* Option

We have to be careful when creating variables within a `switch` control structure. For example, in the code below the variable *x* is declared within `case 2:`

```
switch(value)
{
   case 1:
      // do something
      break;
   case 2:
      int x;
      //do something
      break;
   case 3:
      ...
```

The compiler won't have a problem with this and we can happily access *x* anywhere in the remainder of the `switch` structure:

```
switch(value)
{
   case 1:
      // do something
      break;
   case 2:
      int x;
      //do something
      break;
   case 3:
      //It would be ok to access x here
```

However, if we were to attempt to assign *x* an initial value:

```
case 1:
   // do something
   break;
case 2:
   int x{0};          //Compilation error
   //do something
```

```
        break;
    case 3:
        //do something
        //x would not be set to zero if accessed here
```

the compiler would now produce an error message. The problem here is that in order to set *x* to zero, the code within `case 2` would have to be executed and this will not happen if the control variable, *value*, contains anything other than the value 2.

However, it's okay to initialise our variable if the statement appears in the final option (`case:` or `default:`) within the `switch` control structure:

```
    case 1:
        // do something
        break;
    case 2:
        //do something
        break;
    case 3:
        int x{0};          //OK
        //do something
```

The reason this works is simply because there are no later options where *x* might be accessed without having been initialised.

If we have a variable that we wish to initialise and make use of within a single `case` block, then we can achieve this by enclosing that `case`'s code within its own set of braces:

```
    switch(value)
    {
        case 1:
            ...
            break;
        case 2:
        {
            int x{0};       //OK
            ...
            break;
        }
        case 3:
            ...
    }
```

Activity 6.20

Start a new project called *TestSwitch* and enter the code given below

```
#include <iostream>
using namespace std;
int main()
{
      int value;
      cout << "Enter value ";
      cin >> value;
      switch (value)
      {
            case 1:
```



Activity 6.20 (continued)

```
                        cout << "Case 1 selected\n";
                        break;
                case 2:
                        int x;
                        cout << x << endl;
                        cout << "Case 2 selected\n";
                        break;
                case 3:
                        x = 2;
                        cout << "Case 3 selected\n";
        }
}
```

Check that the program compiles and runs (it doesn't do anything useful).

Change the line

```
int x;
```

to

```
int x{0};
```

What happens when you attempt to compile the new code?

Move the declaration of *x* from `case 2:` to the `switch` code line. How does this affect the compilation?

Remove the line `x = 2;` from `case 3`. *This means that the variable* x *is only used within the* `case 2` *option.*

Return the declaration of *x* to the `case 2` code and enclose all of the `case 2` code in its own set of braces. Does the code now execute as expected?

Testing

A `switch` statement should be tested by creating data that matches every value specified in its `case` labels. In addition, a test value which does not match any of the `case` labels should also be used. This last test will mean that none of the code within the `switch` structure will be executed unless a `default` option has been added.

Sometimes fully testing code when it is embedded in the final program can be difficult or impossible.

For example, testing the `switch` statement that we've added to our *DayOfWeek* project cannot be done by directly entering values for the variable *dayofweek*. Instead, we need to come up with seven dates which after the various calculations have been performed, give the values 0 to 6 for *dayofweek*.

In order to test the `default` option in the `switch` structure we need a value for *dayofweek* that is outside the range 0 to 6. What date can we enter to achieve this?

Because of problems like these, code is often separated from the project for which it is designed and tested separately. For example, we could test our `switch` statement for the days of the week with the program in FIG-6.22 before inserting it in the final program.

FIG-6.22

Using switch

```
#include <iostream>
using namespace std;

int main()
{
   int dayofweek;
   //*** Get value for dayofweek ***
   cout << "Enter value for dayofweek : ";
   cin >> dayofweek;

   //*** Code being tested ***
   switch (dayofweek)
   {
      case 0:  cout << "Sunday\n";
               break;
      case 1 : cout << "Monday\n";
               break;
      case 2 : cout << "Tuesday\n";
               break;
      case 3 : cout << "Wednesday\n";
               break;
      case 4 : cout << "Thursday\n";
               break;
      case 5 : cout << "Friday\n";
               break;
      case 6 : cout << "Saturday\n";
               break;
      default: cout << "Invalid value in dayofweek.\n";
               cout << " Value is " << dayofweek << endl;
   }
}
```

A piece of code created specifically for the purpose of testing some other code, is known as a **test driver**.

Activity 6.21

Start a new project called *SwitchTestDriver* and reproduce the program shown in FIG-6.22.

Use the program to test the `switch` structure. You should perform eight test runs. The first seven of these should use the test values 0 to 6; the final run using any other integer value.

Once the `switch` structure has been tested, copy and paste it into your *DayOfWeek* program so that the day of the week is displayed using the day name rather than a number.

Summary

- Binary selection is implemented in C++ using the `if` statement.
- The expression used within an `if` statement is normally a Boolean expression but can be otherwise.
- Conditional operators used within a Boolean expression are:

`<`	Less than

`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to
`==`	Equal to
`!=`	Not equal to

- The parts of a program in which a variable can be accessed is known as the variable's scope.
- A variable's scope starts at the point in the code where the variable is declared and covers the remainder of that block and any embedded inner blocks.
- C++17 allows an initialiser statement to be included within the parentheses of an `if` statement, prior to the expression.
- A Boolean variable can be used in place of a Boolean expression as the condition in an `if` statement.
- When a non-Boolean expression is converted to Boolean it is first converted to a single integer value. All non-zero values are converted to `true`; zero is converted to `false`.
- Simple Boolean expressions can be linked using the Boolean AND operator (`&&`) or the Boolean OR operator (`||`).
- The meaning of a Boolean expression can be inverted using the NOT operator (`!`).
- `if` statements can be nested.
- When using nested `if` statements an `else` clause automatically links to the last `if` statement which has not yet been linked to an `else`.
- The `else` clause can be forced to link to an earlier `if` by using braces.
- The conditional assignment operator allows one of two values to be assigned to a variable depending on the result of a Boolean expression.
- The `switch` statement can be used to select from multiple alternatives.
- C++17 allows an initialiser statement to be included within the parentheses of a `switch` line, prior to the expression.
- Each `case` label within the `switch` statement must specify a single integer or character value.
- Each of the `case` options within a `switch` statement should normally end with the keyword `break`.
- `break` has the effect of jumping execution to the end of the `switch` control structure.
- If `break` is omitted at the end of the code for a `case` option, execution will continue to the code for the following `case` option.
- To highlight the fact that a `break` statement has been deliberately omitted, the term `[[fallthrough]]` may be inserted in its place.
- Multiple values can pass control to the same section of code within a `switch` statement by placing several `case` labels together.
- The `default` option in a `switch` statement will be executed if none of the

`case` values match the value of the expression given at the start of the `switch` statement. Where no `default` option is given, the whole `switch` statement is bypassed.

- The `switch` statement will execute only the first `case` label whose value matches that of the `switch` statement expression.
- C++17 allows an initialiser statement to be included within the parentheses of a `switch` , prior to the controlling expression.
- Variables may be declared within the `switch` structure but cannot be assigned an initial value unless the variable is declared within its own code block or in the final option (`case` or `default`) of the `switch` structure.

Solutions

Activity 6.1

b) Invalid. Should have double equals symbols

```
x == y
```

e) Invalid. Not equal must be written as `!=` not `=!`

Activity 6.2

When run using a negative value for *no1*, say -3, the output will be

```
negative
The number is -3
```

When run with a positive value, say 10, the output will be

```
The number is 10
```

Activity 6.3

a) `if(no1+6 <= no2)`

b) The easy option is to write

```
if (no % 3 == 0)
```

but if you remember that dividing two integers gives an integer result, then you could also use

```
if (no/3 * 3 == no)
```

The solution to b) works because any number exactly divisible by 3 when divided by 3 and then multiplied by 3 will be equal to the original number. This is not true for values that are not exactly divisible by 3. Remember when both values are integers (*no1* and 3), division will give an integer result. For example 14/3 = 4. Now when we multiply this by 3, we do not arrive at the original value (4 * 3 = 12).

Activity 6.4

Code for *CircleArea*:

```
#include <iostream>
using namespace std;

int main()
{
  const double PI = 3.14159265;
  double radius;
  double area;

  //*** Read radius ***
  cout << "Enter radius of circle : ";
  cin >> radius;
  //*** IF not negative, calc and display area ***
  if (radius >= 0)
  {
    area = PI * radius * radius;
    cout << "The area of a circle whose radius is "
      <<radius<<" is "<<area<<endl;
  }
}
```

Activity 6.5

Although the contents of *area* are undefined when a negative value is entered, the program compiles and runs without any problem.

Activity 6.6

The program no longer compiles since the variable *area* is not recognised in the final `cout` statement..

Activity 6.7

A possible set of test values are:

Test Data	Expected Results
radius = -2	No result
radius = 0	0
radius = 5	78.539

Activity 6.8

a) Code for *Check01*:

```
#include <iostream>
using namespace std;

int main()
{
  int no;

  //*** Read in number ***
  cout << "Enter number between 1 and 12  ";
  cin >> no;

  //*** IF value in range, display message ***
  if(no >= 1 && no <= 12)
    cout << "Value accepted\n";
}
```

b) Code for *Check02*:

```
#include <iostream>
using namespace std;

int main()
{
  int no;

  //*** Read number ***
  cout << "Enter number ";
  cin >> no;

  //*** IF number 9 or 14, display message ***
  if (no == 9 || no == 14)
    cout << "OK\n";
}
```

Activity 6.9

Test Data	Expected Results
no1 = 5, no2 = 7	...
no1 = 6, no2 = 4	...
no1 = 7, no2 = 5	...
no1 = 8, no2 = 3	...

Notice that the test data also includes values that make the variables' values match the literals given in the conditions.

Activity 6.10

a) True (7 -5 = 2)
b) False (7 - 7 = 0)
c) False(0.6 truncates to 0)
d) True (7 + 5 = 12)
e) False (Null char '\0' converts to 0)
f) False ('A' converts to 65 so we get 65-65 = 0)

Activity 6.11

Code for *EqualityTest*:

```
#include<iostream>
using namespace std;

int main()
{
```

```
    int no1, no2;

    //*** Read in two values ***
    cout << "Enter first number ";
    cin >> no1;
    cout << "Enter second number ";
    cin >> no2;
    //*** IF equal, display 'Equal' ELSE display 'Not
          ↳equal' ***
    if (no1 == no2)
      cout << "Equal\n";
    else
      cout << "Not equal\n";
}
```

Activity 6.12

Modified code for *CircleArea*:

```
#include <iostream>
using namespace std;

int main()
{
  const double PI = 3.14159265;
  double radius;
  double area;

  //*** Read radius ***
  cout << "Enter radius of circle : ";
  cin >> radius;
  //*** IF not negative, calc and display area ***
  if (radius >= 0)
  {
    area = PI * radius * radius;
    cout << "The area of a circle whose radius is "
    ↳<<radius<<" is "<<area<<endl;
  }
  else
    cout << "Invalid radius\n";
}
```

Activity 6.13

We need test values to make each `if` statement *true* and *false*. In addition, since the last `if` statement contains two Boolean conditions linked by the AND operator, we need to attempt the combinations *false*, *false*; *false*, *true*; *true*, *false*; and *true*, *true* for that `if` statement. However, since the conditions in this final `if` statement cannot both be *false* at the same time, the *false*, *false* option cannot be achieved and so may be ignored.

Test Data	Expected Results
no1 = 5 (false)(false)(true,false)	*No display*
no1 = 4 (false)(true)(true,false)	Even
no1 = -7 (true)(false)(false,true)	Negative
no1 = -8 (true)(true)(false,true)	Negative, Even
no1 = -2 (true)(true)(true,true)	Negative, Even, Near Zero

Note that the test data shows in parentheses how the Boolean conditions given in each `if` statement will react to that value. For example, the first line of the test data table states that 5 will make the Boolean expression in the first `if` *false*, in the second `if`, *false*, and in the third `if`, the two conditions will be *true, false*.

Activity 6.14

Modified code for *BMI*:

```
#include <iostream>
using namespace std;

int main()
{
  float weight, height, bmi;

  //*** Get weight ***
  cout << "Please enter your weight (kilos):";
  cin >> weight;

  //*** Get height ***
  cout << "Please enter your height (metres):";
  cin >> height;

  //*** Calculate BMI ***
  bmi = weight/(height+height);

  //*** Display details ***
  cout << "Your height : " << height<<endl;
  cout << "Your weight : " << weight<<endl;
  cout << "Your Body Mass Index : "<<bmi<<endl;
  //*** Display BMI classification ***
  if (bmi < 18.5)
    cout << "Underweight\n";
  if (bmi >= 18.5 && bmi < 25)
    cout << "Normal\n";
  if (bmi >= 25 && bmi < 30)
    cout << "Overweight\n";
  if (bmi > 30)
    cout << "Obese\n";
}
```

The `if` statements are mutually exclusive since only one of them can be *true* for any given value of *bmi*.

Activity 6.15

Modified code for *BMI*:

```
#include <iostream>
using namespace std;

int main()
{
  float weight, height, bmi;

  //*** Get weight ***
  cout << "Please enter your weight (kilos):";
  cin >> weight;

  //*** Get height ***
  cout << "Please enter your height (metres):";
  cin >> height;

  //*** Calculate BMI ***
  bmi = weight/(height+height);

  //*** Display details ***
  cout << "Your height : " << height<<endl;
  cout << "Your weight : " << weight<<endl;
  cout << "Your Body Mass Index : "<<bmi<<endl;
  //*** Display BMI classification ***
  if (bmi < 18.5)
    cout << "Underweight\n";
  else
    if (bmi < 25)
      cout << "Normal\n";
    else
      if (bmi < 30)
        cout << "Overweight\n";
      else
        cout << "Obese\n";
}
```

Notice that the complexity of the conditions has been reduced and the last `if` statement has been replaced by an `else`.

An alternative layout for the nested if statements would be:

```
if (bmi < 18.5)
     cout << "Underweight\n";
else if (bmi < 25)
     cout << "Normal\n";
else if (bmi < 30)
     cout << "Overweight\n";
else
     cout << "Obese\n";
```

Activity 6.16

Code for *BankInterest*:

```
#include <iostream>
using namespace std;
```

```
int main()
{
  double balance, interest ;

  //*** Read in balance ***
  cout << "Enter account balance ";
  cin >> balance;

  //*** Calculate interest ***
  if ( balance < 0)
    interest = balance *0.005;
  else if (balance < 2000.0)
    interest = balance * 0.001;
  else
    interest = balance * 0.002;

  //*** Display balance and interest ***
  printf("Your balance is $%0.2f\n",balance);
  printf("Interest on your account (a negative figure
        represents charges) $%0.2f\n",interest);
}
```

Activity 6.17

Modified code for *GuessingGame*:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
  int number, guess;

  //*** Generate random number in range 1 to 10 ***
  srand(static_cast<unsigned>(time(0)));
  number = rand()%10 + 1;

  //*** Read guess ***
  cout << "Guess the number (between 1 and 10) : ";
  cin >> guess;

  //*** Display message ***
  if (guess < number)
    cout << "Guess too low\n";
  else if ( guess > number)
    cout << "Guess too high\n";
  else
    cout << "Guess correct\n";

  //*** Display the number and guess
  cout << "The number was " << number << endl;
  cout << "Your guess was " << guess << endl;
}
```

Activity 6.18

Code for *Ages* (Version 1):

```
#include <iostream>
using namespace std;

int main()
{
  char gender;
  int age;

  //*** Get gender ***
  cout << "Enter gender (M or F) ";
  cin >> gender;
  //*** Get age ***
  cout << "Enter age ";
  cin >> age;
  // *** Check details ***
  if (gender == 'M')
    if (age > 65)
      cout << "Retired male\n";
    else
      cout << "Female\n";
}
```

In this first version, the data M, 21 displays the word *Female*.

The data F, 31 displays nothing.

Code for *Ages* (Version 2):

```
#include <iostream>
using namespace std;

int main()
{
  char gender;
  int age;

  //*** Get gender ***
  cout << "Enter gender (M or F) ";
  cin >> gender;
  //*** Get age ***
  cout << "Enter age ";
  cin >> age;
  // *** Check details ***
  if (gender == 'M')
  {
    if (age > 65)
      cout << "Retired male\n";
  }
  else
    cout << "Female\n";
}
```

In this second version, the data M, 21 displays nothing.

The data F, 31 displays *Female*.

Activity 6.19

```
x = ( x < 0)?-x:x;
```

Activity 6.20

The first version of the program compiles and runs.

When the line

```
int x;
```

is changed to

```
int x{0};
```

the program no longer compiles.

When the declaration of *x* is moved into the `switch` line,

```
switch(int x{};value)
```

the program compiles and runs.

The final version of the switch control structure is coded as:

```
switch (value)
  {
    case 1:
      cout << "Case 1 selected\n";
      break;
    case 2:
    {
      int x{ 0 };
      cout << x << endl;
      cout << "case 2 selected\n" << endl;
      break;
    }
    case 3:
      cout << "Case 3 selected\n";
      break;
  }
```

This executes as expected.

Activity 6.21

The test values 0 to 6 should result in the corresponding day of the week being shown (Sunday to Saturday).

Any value outside that range should produce the *Invalid* message and display the value entered.

Modified code for *DayOfWeek*:

```
#include <iostream>
using namespace std;

int main()
{
```

```
    int day, month, year;// date entered
    int M,C,Y, modifiedyear; // calculated values
                             ↳required by formula
    int dayofweek;// result required

    //*** Get date ***
    cout << "Enter date in the format dd/mm/yyyyy : ";
    scanf_s("%d/%d/%d",&day,&month,&year);

    //*** Calculate M ***
    M = (month + 9) % 12 + 1;

    //*** Calculate modified year ***
    modifiedyear = year - M / 11;

    //*** Calculate C ***
    C = modifiedyear / 100;

    //*** Calculate Y ***
    Y = modifiedyear % 100;

    //*** Calculate day of week ***
    dayofweek = ((static_cast<int>(2.6 * M - 0.2) +
    day + Y + Y/4 + C/4 - 2*C)%7+7)%7;

    //*** Display result ***
    switch (dayofweek)
    {
      case 0:  cout << "Sunday\n";
               break;
      case 1 : cout << "Monday\n";
               break;
      case 2 : cout << "Tuesday\n";
               break;
      case 3 : cout << "Wednesday\n";
               break;
      case 4 : cout << "Thursday\n";
               break;
      case 5 : cout << "Friday\n";
               break;
      case 6 : cout << "Saturday\n";
               break;
      default: cout << "Invalid value in
               ↳dayofweek.\n";
               cout << " Value is "<<dayofweek<<endl;
    }
}
```



Iteration

In this Chapter:

- ❑ The *while* Structure
- ❑ The *do...while* Structure
- ❑ The *for* Structure
- ❑ Using the Comma Operator in Loops
- ❑ Jump Statements
- ❑ Nested Loops
- ❑ Testing Loops

Iteration

Introduction

Iteration (or **looping**) is the term used when one or more statements are carried out repeatedly. As we saw in Chapter 1, structured English has three distinct iterative structures: FOR .. ENDFOR, REPEAT .. UNTIL and WHILE .. ENDWHILE.

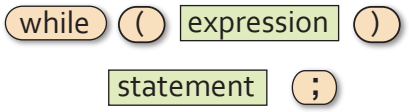
C++ has three similar iterative structures. C++'s `while` statement is identical in operation to structured English's WHILE, but C++' s `do..while` and `for` structures are somewhat different from the REPEAT..UNTIL and FOR..ENDFOR statements of structured English, therefore care should be taken when translating these structured English statements into C++.

The *while* Statement

The `while` statement is probably the easiest of C++'s loop structures to understand, since it is identical in operation, if not syntax, to the WHILE loop in structured English. The general format for the `while` statement is shown in FIG-7.1.

FIG-7.1

`while` Loop - Single Statement



where:

expression is normally a Boolean expression but can be otherwise.

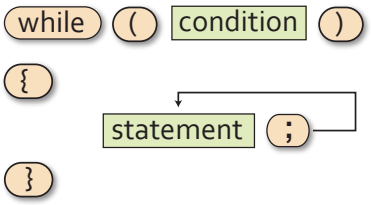
statement is any valid C++ statement. This is known as the **body** of the loop.

If *expression* evaluates to *true*, *statement* is executed; the *expression* is then retested and, if still *true*, *statement* is executed again. This continues until *expression* is found to be *false*, at which stage the program moves on to whatever statement follows the `while` structure.

Almost all loop structures have a loop body which contains more than just a single statement. To achieve that in C++, we need to enclose the statements which form the loop body within a set of braces (see FIG-7.2).

FIG-7.2

`while` Loop - Multiple Statements



The program in FIG-7.3 accepts an integer value from the keyboard that is meant to be in the range 1 to 12. To ensure only a value within that range is accepted, we can use a `while` loop to reject invalid values and force the user to re-enter a value. The program uses the following logic:

```
Read number
WHILE number < 1 OR number > 12 DO
      Display "Invalid. Number must be in the range 1 to 12"
```

Read number
ENDWHILE

FIG-7.3

Using a `while` Loop

```
#include <iostream>

using namespace std;

int main()
{
   int number;
   // *** Read number ***
   cout << "Enter number ";
   cin >> number;
   //*** WHILE number not in range 1 to 12 DO ***
   while ( number < 1 || number > 12)
   {
      // *** Display error message ***
      cout << "Invalid. Number must be in the range 1 to 12\n";
      // *** Read number ***
      cout << "Enter number ";
      cin >> number;
   }
   // *** Display accepted value ***
   cout << "Number accepted was " << number << endl;
}
```

A visual representation of how the `while` loop operates is shown in FIG-7.4.

FIG-7.4

How the `while` Loop Works

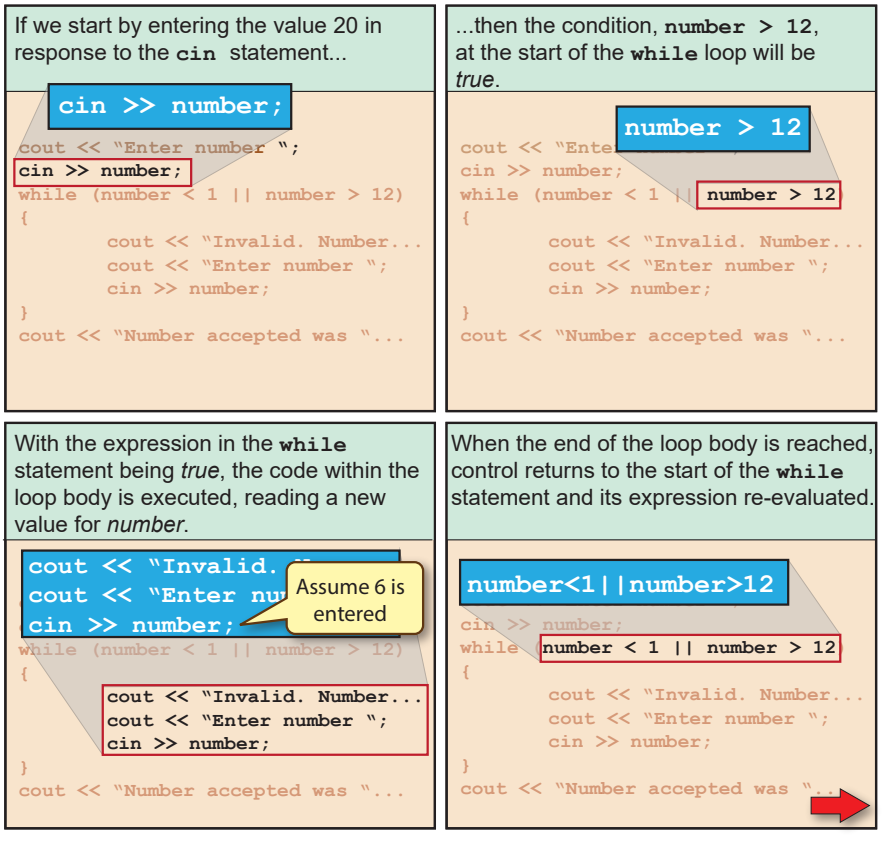
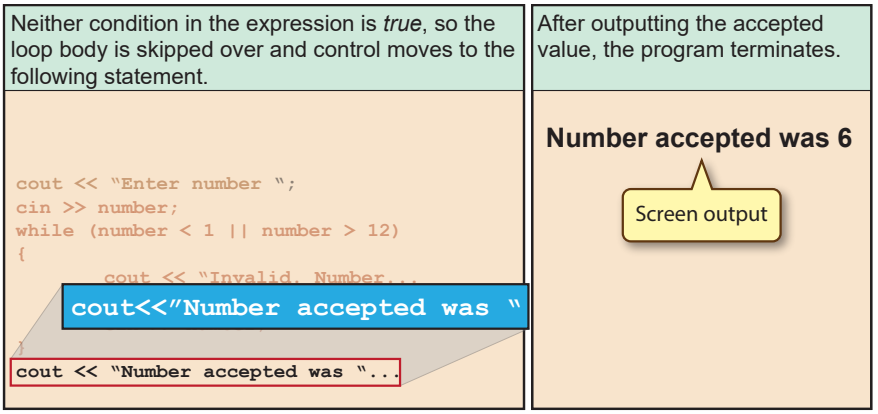


FIG-7.4
(continued)

How the `while` Loop Works



In the above example, the code within the loop body was executed exactly once. But if we had entered two values outside the range 1 to 12 the loop body would have been executed twice. On the other hand, if the very first value entered was, say 6, then the code within the loop body would never have been executed (since neither part of the condition at the start of the `while` loop would have been *true*) and control would have jumped straight to the final `cout` statement.

Activity 7.1

Create a new project called *TestWhile* and type in and test the program given in FIG-7.3.

Run the program three times using the following test data:

Run 1:	8
Run 2:	14, 3
Run 3:	18, 22, 0, 7

How often is the loop body of the `while` statement executed on each run?

Loop counting

We can see that when using a `while` loop it is impossible to tell by just looking at the code how often the loop body will be executed. Should we want to know the number of times a `while` loop is actually executed when the program is run, we can set up an extra variable and use it to count the number of iterations. The variable is incremented each time the loop is executed. FIG-7.5 shows an updated version of the previous program that displays how many attempts were made before valid data was entered.

FIG-7.5

Loop Counting

```
#include <iostream>
using namespace std;

int main()
{
   int number;
   int count = 0;    // Variables used to count iterations.

   cout << "Enter number ";
   cin >> number;
```

FIG-7.5
(continued)

Loop Counting

```
    while ( number < 1 || number > 12)
    {
       cout << "Invalid. Number must be in the range 1 to 12\n";
       cout << "Enter number ";
       cin >> number;
       count++;    //Add 1 to count each time loop executed
    }
    cout << "Number accepted was " << number << endl;
    cout << "You entered " << count << " invalid values\n";
}
```

Activity 7.2

Modify *TestWhile* to include the loop counter as shown in FIG-7.5.

Run the program with the same three sets of test data as before.

Testing

To test a `while` loop, we should create at least three sets of test data. The first of these should ensure that the loop body is executed zero times; the second should execute the loop body exactly once; and the final test data should cause the loop body to be executed more than once. This requirement is reflected in the test data used in the last two Activities.

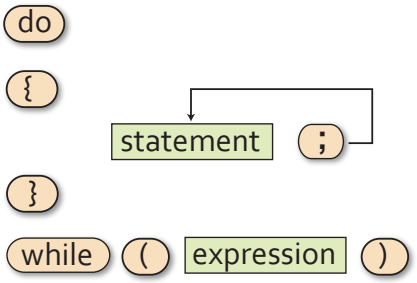
In addition, where the expression given at the start of the `while` loop involves two or more Boolean conditions linked with AND or OR operators, then we should endeavor to test all possible combinations of *true* and *false*, just as we did with the `if` statement. In fact, this additional requirement would not generate any new test data for the last two programs since they already test the combinations *false*, *false* (the value 8 in the first run); *false*, *true* (14 in the second run); *true*, *false* (0 in the third run). The combination *true*, *true*, cannot be created since the two conditions are mutually exclusive.

The *do...while* Statement

The `do..while` loop has the format shown in FIG-7.6.

FIG-7.6

`do..while` Loop



where:

statement	is any valid C++ statement.
expression	is normally a Boolean expression but can be otherwise.

If there is only one statement in the loop body, then the braces can be omitted (just as we've seen already in the `if` and `while` statements).

The statement(s) within the loop body are executed unconditionally. If *expression* is *true*, then control returns to the beginning of the loop body and its statements are executed again. This continues until *expression* is *false*. At that point control moves on to the statement following the `do...while` loop.

The `do..while` statement is as close as C++ comes to structured English's REPEAT.. UNTIL structure. However, the two statements differ in more than syntax. The REPEAT loop is designed to exit when the specified condition becomes *true* but the `do...while` structure exits when its condition is *false*.

For example, an algorithm to sum a set of numbers - the set ending with the value zero - could be written in structured English as:

```
Set sum to zero
REPEAT
    Get a number
    Add number to sum
UNTIL number = 0
Display sum
```

Using a `do...while` loop to implement this in C++ we would get

```
int sum = 0;
int number;
do
{
   cout << "Enter number : ";
   cin >> number;
   sum += number;
}
while (number != 0);
cout << "Total of values entered was " << sum << endl;
```

Notice how the condition in the `do..while` structure – `(number != 0)` – is the opposite of that for the REPEAT...UNTIL structure – number = 0. This is the only change required when implementing a REPEAT...UNTIL structure using C++'s `do...while` statement.

Activity 7.3

Start a new project called *DoLoop* and, using the code given above, write a program to sum a sequence of numbers which ends with the value zero. The program should display the final total of the values entered.

Modify the program so that it displays the running total as well as the final total.

Activity 7.4

Reload your *GuessingGame* project and modify it so that only values in the range 1 to 10 are accepted.

Modify the game again so that the player can keep on guessing until the guess matches the number. The number of guesses required is also displayed at the end of the game.

Notice the it is the nature of a `do...while` loop that the code within the loop body must be executed at least once.

Testing

Test requirements for a `do...while` loop are similar to those for the `while` loop but because the loop body of a `do...while` cannot be executed zero times, then obviously that particular test should be omitted.

The *for* Statement

In structured English, the FOR loop is used to perform an action a specific number of times. For example, we could total the exam marks achieved by a student over 5 exams using the algorithm

```
Set total to zero
FOR 5 times DO
      Get mark
      Add mark to total
ENDFOR
Display total
```

To achieve the same result using C++'s `for` statement, we need to put in a little more effort.

When we want to perform a loop body a specific number of times, we need to set up a variable which will count how often the loop body is executed. We already made use of such a counter in a `while` loop example we saw earlier.

In addition, we need to give the loop counter a starting value - often 1 – when using a `for` loop. And finally, we need to specify what value the loop counter should stop at. So the starting value of the counter and its finishing value together determine how often the loop is to be executed. For example, if we started the counter at 1 and wanted it to finish at 10, the code within the loop body would be executed exactly 10 times. Of course, each time we execute the loop body we need to remember to add 1 to the loop counter otherwise it will always stick at its starting value (1) and we'll never finish executing the loop.

That all seems simple enough, but C++'s `for` statement insists on one little variation on the explanation given above. It requires, not the final value of the loop counter, but the range of values under which looping is to continue. In other words, instead of saying we want to finish when the counter gets to 10, say, we have to specify the looping is to continue while the counter is less than or equal to 10.

All of these requirements are expressed in a single line which starts with the keyword `for` and an opening parenthesis:

```
for(
```

This is followed by the variable being used as a loop counter and the initial value to be assigned to it:



counter needs to be declared as a variable.

```
for(counter = 1
```

Next comes a semicolon followed by the Boolean expression giving the condition under which looping is to continue:

```
for(counter = 1; counter <= 10
```

Finally, we have another semicolon and a statement instructing the counter to be incremented each time the loop is executed, followed by a closing parenthesis:

```
for(counter = 1; counter <= 10; counter++)
```

Now we can write the program (see FIG-7.7) which performs the logic given at the start of this section, calculating the total of a student's five exams.

FIG-7.7

Using a for Loop

```
#include <iostream>
using namespace std;

int main()
{
   int mark ;     //exam mark achieved by student
   int total=0;   //total for all five exams
   int counter;   //for loop counter

   //*** FOR 5 times DO ***
   for(counter = 1; counter <= 5; counter++)
   {
      //*** Get mark ***
      cout << "Enter exam mark : ";
      cin >> mark;
      //*** Add mark to total ***
      total += mark;
   }

   //*** Display total for all five exams ***
   cout << "Total for all five exams is " << total << endl;
}
```

Activity 7.5

Create a new project, *ExamMarks*, and type in and test the program given in FIG-7.7.

Modify the program so that it also gives the student's average mark.

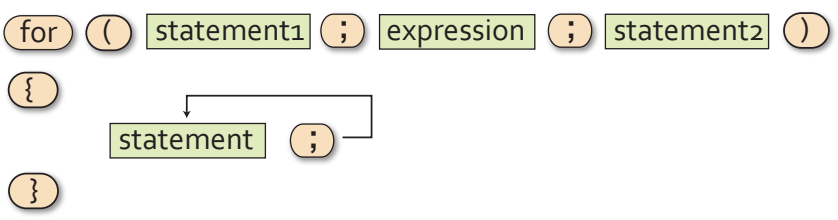
Modify the program again so that the student only sits four exams instead of five.

Modify the program so that a named constant, NOOFEXAMS is used to specify the number of exams sat. Use this constant at appropriate parts in your code. What changes are required to the code if the student were to sit six exams instead of four?

The general form of the `for` statement is shown in FIG-7.8.

FIG-7.8

for Loop



where:

statement1 is any valid C++ statement. Normally, it will be an assignment statement setting the loop counter to its starting value. This statement will be executed only once when control arrives at the `for` statement. This part of the `for` loop is often known as the **initial expression**.

expression is normally a Boolean expression. This expression is evaluated at the start of each iteration. Only if expression is *true* will the loop body be executed. This part of the `for` loop is known as the **conditional expression**.

statement2 any valid C++ statement. Normally, it will be an instruction to increment (or decrement) the loop counter variable. This statement is executed immediately after the statements of the loop body have been executed. This part of the `for` loop is known as the **loop expression**.

We can summarise exactly how a `for` loop operates as follows:

```
The pre-iteration statement (statement1 in the diagram) is executed.
WHILE expression is true DO
    The code in the loop body is executed
    The post-loop statement (statement2) is executed
ENDWHILE
```

Options and Variations in the for Statement

There are various options available when constructing a `for` statement. The simplest of these is the ability to start the loop counter at any value required and to increment (or decrement) by any amount. Some examples are shown below:

```
for(c = -10; c <= 20 ; c+=2)
```

starts the variable *c* at -10 and adds 2 after each iteration.

```
for(k = 10; k >= 0 ; k--)
```

starts *k* at 10 and subtracts 1 after each iteration.

Activity 7.6

Start a new project, *Countdown*, which displays the numbers 10, 9, 8, etc down to 0.

The counter can even be a `char` variable:

```
char ch;
for (ch = 'A'; ch <= 'Z'; ch++)
```

The code above will take the loop counter *ch*, through the letters of the alphabet.

Activity 7.7

Start a new project, *Alphabet*, which displays the letters of the alphabet in lowercase and on a single line.

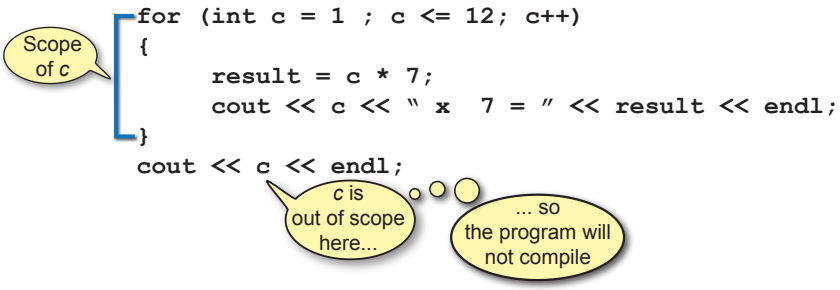
A very commonly used option is to declare the loop counter variable within the first line of the `for` loop. For example,

```
for (int c = 1 ; c <= 10; c++)
```

When this option is used, C++ treats the loop counter variable as one that has been declared within the body of the loop (see FIG-7.9).

FIG-7.9

Loop Counter's Scope



Even when the `for` loop contains only a single line (and hence no braces), the scope of the loop counter declared within the `for` statement is still restricted to within the loop.

Another variation on the `for` statement's structure is to omit, one, two or even all three sections within the parentheses of the statement - but the semicolons must be retained. The following omits the initialising statement:

```
cin >> no;
final = no + 10;
for( ; no <= final; no++)
```

The next snippet omits the post-loop statement:

```
for (total = 0; total <= 100 ; )
{
   cout << "Enter value ";
   cin >> no;
   total += no;
}
```

Omitting the expression from the middle of the `for` statement, although permitted, will cause problems. Without an expression, the result is always assumed to be *true* and so the loop body, once entered, would be repeated forever. By omitting both the first and last statements and retaining only the expression as in

```
cin >> no;
for ( ; no < 0 ; )
{
   cout << "Invalid. Re-enter ";
   cin >> no;
}
```

we have just created another version of a standard `while` loop.

From this example, we can see that C++'s `for` statement is so flexible that it doesn't just have to be used as a method of counting how often the loop is executed; it can be made to terminate under any conditions quite unrelated to the number of iterations that have taken place. The program in FIG-7.10 attempts to demonstrate how to read in an unknown number of values and calculate their average by making use of both

the `for` statement's counting feature and its ability to terminate under an unrelated condition (in this case, the program terminates when a value of zero is entered.

FIG-7.10

`for` Loop Variations

```
#include <iostream>
using namespace std;

int main()
{
   int no = 99;      //Value entered (start with dummy value)
   int total = 0;    //Total of all values entered
   int count;        //Count of the number of values entered
   double average;   //Average of the values entered

   //*** Get a set of values ending with zero ***
   //*** and count how many numbers are entered ***
   for (count = 0 ; no != 0 ; count++)
   {
      //*** Get number ***
      cout << "Enter number ";
      cin >> no;
      //*** Add number to total ***
      total += no;
   }

   //*** Reduce the count by 1 ***
   count--; // Don't count the zero as one of our numbers
   //*** Calculate the average ***
   average = total / count;

   //*** Display the average ***
   cout << "Average of the "<< count << " numbers is " << average
   ↳<< endl;
}
```

Although the `for` loop in the program above is fine, there are a couple of problems with the code elsewhere. Have a look and see if you can spot them (unlikely). We'll investigate further in the next Activity.

Testing

We saw earlier that by removing everything but the expression within a `for` loop, we are left with a structure that acts as a `while` loop. This tells us what sort of testing should be done on a `for` loop.

Some `for` loops will be coded in such a way as to execute a set number of times, and so that will be the only test we can perform, creating enough test data to allow the specified number of iterations.

Other `for` loops are coded with more flexible conditions (as is the one in FIG-7.10). Under these circumstances, we should create test data which will give, zero, one and multiple iterations of the loop. Possible test data for the previous program is shown in FIG-7.11.

FIG-7.11

`for` Loop Test Data

Test Data	Expected Results
no = 9, 10, 4, 0	Average of the 3 numbers is 7.6667
no = 29, 0	Average of the 1 numbers is 29
no1 = 0	No average

Activity 7.8

Start a new project called *ForLoop* and recreate the program shown in FIG-7.10. Test the program using the test data shown in FIG-7.11.

What problems are highlighted by the test runs? Correct the two problems in the code. (HINT: one needs to make use of the `static_cast` keyword, the other requires an `if` statement to be added.)

Using the Comma in a *for* Statement

The `for` statement is where we are most like to come across the use of the comma operator we discussed back in Chapter 4.

It is used, not because we're interested in the value it returns, but simply as a method of fitting more that one instruction into the *initial* and *loop* expressions. Hence it is quite valid to write

```
for (c = 1, total = 0; c <= 10; c++, total += no)
{
 cout << "Enter number ";
 cin >> no;
}
```

Notice that *c* is set to 1 and *total* to 0 before the looping starts; immediately after the loop body has been executed, *c* is incremented and *no* is added to *total*.

Activity 7.9

In the last version of *GuessingGame*, you made use of a `do...while` loop to allow the user multiple chances at guessing the number. The problem with this is that, with unlimited guesses, there is little challenge in the game.

Rewrite your previous program to use a `for` loop limiting the number of guesses to 5. What problems arise using the new version?



We'll return to this problem later.

Using Variables and Expressions in a *for* Statement

The starting value of a loop counter and the final value given in the condition need not simply be a constant as in previous examples. Variable and arithmetic expressions may also be used. For example, in the code snippet

```
cin >> max;
for (c =0; c <= max ; c++)
```

the loop will execute *max* number of times. Since the value of *max* is input from the keyboard then there will be a different number of iterations each time the program is run and the value entered for *max* changes.



Another version of the *for* loop exists which is used when working through every value in a collection of data. This is known as the **range-based for loop** and is covered in a later chapter.

Activity 7.10

Create a new project, *OneTo*, that reads in a positive integer value in the range 0 to 50 (include code to check the value) and then displays all the numbers between 1 and the number entered.

Jump Statements

Statements that transfer the flow of control to some other part of the program are known as **jump** statements.

There are four jump statements in C++: `break`, `continue`, `goto` and `return`. The `return` statement we will look at in a later chapter.

The *break* Statement

We have already come across the `break` statement. It was used at the end of each `case` option within a `switch` statement where executing the `break` instruction caused control to jump to the end of the `switch` statement. The `break` command can also be used within any of the three loop structures to force an early exit from that loop. But it is most likely to be found within a `for` loop.

Executing the `break` statement within a loop causes control to jump to the end of that loop structure. There will be no further iterations of the loop body irrespective of whether the condition given at the start (`while` and `for`) or at the end (`do..while`) of that loop statement is *true* or *false*.

The program in FIG-7.12 uses a `for` loop to read in and total 5 numbers. However, if the number entered is zero, no more values will be read before displaying the total.

FIG-7.12

Reading and Totalling Five Numbers

```
#include <iostream>
using namespace std;

int main()
{
   int no;
   int count;
   int total = 0;
   //*** Read 5 numbers ***
   for(count = 1 ; count <= 5; count++)
   {
      //*** read number ***
      cout << "Enter number ";
      cin >> no;
      //*** If it's zero, exit loop ***
      if ( no == 0)
         break;
      total += no;
   }
   //*** Display the total ***
   cout << "Total of values entered was " << total << endl;
}
```

Notice, that the `break` is positioned within an `if` statement since we only want to exit the loop when a specific condition applies (here it's when zero is entered).

Activity 7.11

Start a new project called *Total* and recreate the program shown in FIG-7.12.

Test your program using the following values:

Run 1: 0 Run 2 : 3, 0 Run 3 : 5,9,1,4,4

The problem with the last version of our *GuessingGame* program was that it made the user have exactly five attempts at guessing the number - even if it had been correctly guessed after the first attempt! By placing a `break` statement within the `for` loop we can allow looping to terminate as soon as the correct value is guessed.

Activity 7.12

Modify your *GuessingGame* program to stop if the correct value is guessed in less than five attempts.

Test your changes making sure that on at least one occasion you do guess the correct value in 4 or less attempts.

The *continue* Statement

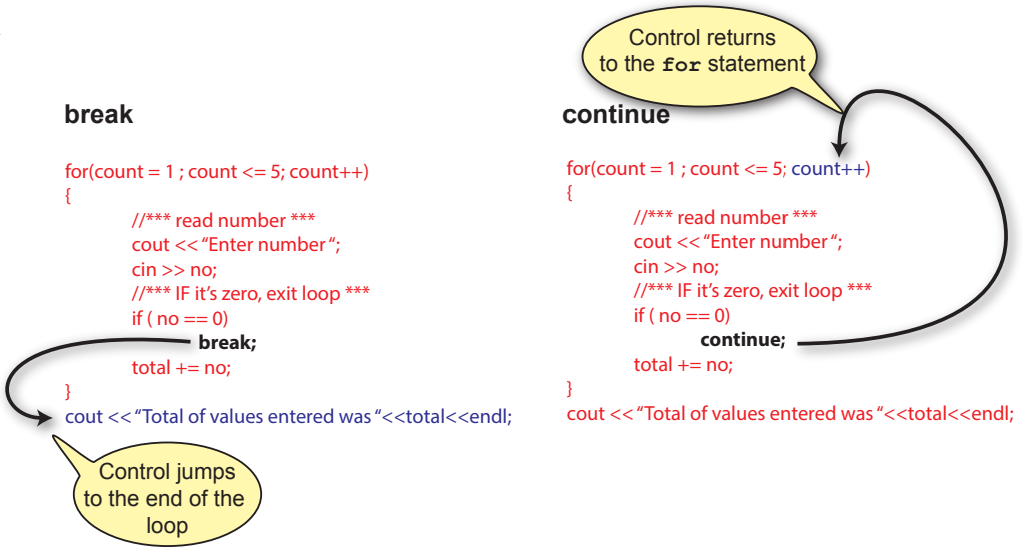
The `continue` statement is designed solely for use within a loop structure. Its effects are less severe than that of `break`; rather than exit the loop structure entirely it exits the current iteration only, ignoring any remaining statements in the loop body and jumping back to the start of the loop where the next iteration proceeds as normal.

In the case of a `for` loop, executing the `continue` statement causes control to exit the current iteration then the loop expression in the `for` statement (*statement2* in FIG-7.8) is executed before the `for` statement's *expression* is tested, and if appropriate, iteration continues as normal.

FIG-7.13 shows the difference between execution of the `break` and `continue` statements of the flow of control.

FIG-7.13

`break` and `continue`



Activity 7.13

In your *GuessingGame* program what would happen if you changed the `break` statement to a `continue` statement?

The *goto* Statement

The `goto` statement is used to transfer control to a specified line of code. The destination line is identified using a **label**. A label is any valid identifier followed by a colon.

The code snippet below shows how the `goto` statement might be used to exit the `for` loop already used in the `break` and `continue` examples in FIG-7.13.

```
for(count = 1 ; count <= 5; count++)
{
    //*** read number ***
    cout << "Enter number ";
    cin >> no;
    //*** IF it's zero, exit loop ***
    if ( no == 0)
       goto next;
    total += no;
}
next:
cout << "Total of values entered was "<<total<<endl;
```

C++ will not allow the `goto` statement to jump past any variable declarations. For example, the following extract is invalid since it would cause the declaration of the variable *total* to be bypassed:

```
goto next:
   .
int total = 0;
   .
next:
   .
```

As well as jumping forward in a program, the `goto` statement can be used to jump back to an earlier statement:

```
back:
   .
   goto back;
```

But since this creates a looping situation it is best handled using one of the loop statements (`while`, `do`, or `for`).

Unlike `break` and `continue`, which can only be used within a control structure, the `goto` statement can be used anywhere within a program. Nevertheless, most guides to good programming practices would suggest that use of the `goto` statement should be avoided under all circumstances since it can make programming code difficult to follow, unstructured and awkward to test.

Largest and Smallest Algorithms

We've already looked at code which uses looping to total a set of values and work out the average of a set of values. These are two very common requirements in programming. Another is finding the largest (or smallest) value in a set of numbers.

Let's imagine we have to read in 8 numbers and then display the largest of those 8 numbers. Seems an easy enough task; something we could achieve in the real world without any difficulty. But writing code to perform the same task isn't quite so simple if we're new to programming.

When we're presented with a problem that needs to be implemented in program code, we need to have a clear idea of how to tackle that problem before we start coding. So let's think about how we might find the smallest value in a set of numbers if we had to perform the task ourselves.

If it was a small enough set of values and they were written down, we would just glance at the list and, as if by magic, we'd be able to pick out the smallest value. On the other hand, asked just exactly how we arrived at our answer, we might struggle to give a logical, step-by-step explanation. And that's the whole problem: if we can't explain how something is done, then we can't program it.

Sometimes it actually helps to make the problem more complicated. Imagine the set of numbers was a very long one and that each number was read out to us. When, after some time, the last number is read out, we need to repeat the smallest value in the list. Since there is no chance that we can memorise every number we hear, we're going to have to come up with a more methodical approach. Of course, all we need is a piece of paper and a pencil; write down the first number read out, then, when a smaller number is read out erase the number on the piece of paper and write down the new one. Follow that same strategy until all the values have been read, and at the end the smallest value in the list will be the last number we wrote down.

This now seems simple enough but it's only once we've gained this sort of insight into solving a problem, and then describe our approach using structured English, that we're ready to start programming. So, the task of finding the smallest of 8 numbers can be described in structured English as:

```
Get first value
Set smallest to first value
FOR 7 times DO
    Get value
    IF value is less than smallest THEN
        Set smallest to value
    ENDIF
ENDFOR
Display smallest
```

Activity 7.14

Create a new project called *Extreme* which implements the algorithm given above. Produce test data for the program and check that it displays the expected results.

Modify your program to find the largest value instead of the smallest.

Modify the program again so that it finds both the largest and smallest values.

Activity 7.15

In ice skating, a small number (n) of judges award marks. The highest and lowest of these are ignored and the others averaged to give a result.

Create a project, *IceScore*, to input a value for n (the number of judges - between 5 and 9), followed by n marks (the mark given by each judge), and display the average score ignoring the highest and lowest mark. (Scores range from 0.0 to 6.0 in increments of 0.1.) The logic required by the program is:



Activity 7.15 (continued)

```
Get value for n
Get value for score
Set highestscore to score
Set lowestscore to score
Set total to score
FOR n-1 times DO
        Get value for score
        Add score to total
        IF score > highestscore THEN
                Set highestscore to score
        ELSE
                IF score < lowestscore THEN
                        Set lowestscore to score
                ENDIF
        ENDIF
ENDFOR
Subtract highestscore and lowestscore from total
Calculate average as total / (n-2)
Display value of average
```

Nested Loops

Many algorithms require **nested loops**. These are situations where one loop control structure appears within another.

In Activity 7.1 we wrote a program using a `while` loop to check that a number was in the range 1 to 12 and in Activity 7.5 we created a program that made use of a `for` loop to read in the 5 exam marks achieved by a student. Now let us imagine a situation where we have to read in 5 exam marks for a student and also make sure that each of those marks was in the range 0 to 100. Obviously, we are going to need to make use of a `while` statement to make sure the marks are in the range 0 to 100 and a `for` loop to read all 5 marks. This results in the following algorithm:

```
Set total to zero
FOR 5 times DO
    Get mark
    WHILE mark < 0 OR mark > 100 DO
        Display error message
        Get mark
    ENDWHILE
    Add mark to total
ENDFOR
Calculate average
Display average
```

In this algorithm we have an example of a nested loop, with the `while` loop embedded within the `for` loop.

Activity 7.16

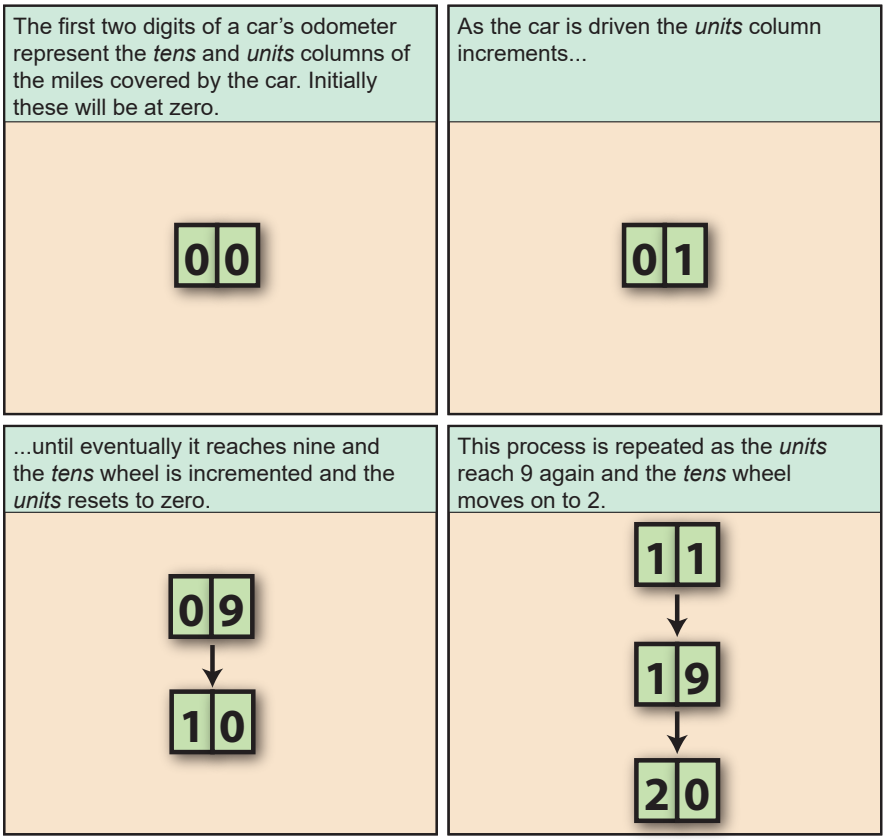
Modify *ExamMarks* so that every mark is in the range 0 to 100.

Nested *for* Loops

Perhaps the most troublesome situation is where `for` loops are nested. The following example demonstrates the characteristics of such a structure. Consider the first two digits of a car's odometer. Initially, they are set to 00. As the car moves, the least-significant digit (*units*) increments while the most significant digit (*tens*) remains unchanged. But when the *units* value reaches 9, the *tens* value increments and the *units* value is reset to zero (see FIG-7.14).

FIG-7.14

How Nested `for` Loops Work



This situation in FIG-7.14 is matched exactly by the following code:

```
for(tens = 0; tens <= 9; tens++)
   for(units = 0; units <= 9; units ++)
      cout << tens << units << endl;
```

The *tens* loop is known as the **outer loop**, while the *units* loop is known as the **inner loop**. A few points to note about nested `for` loops:

- The inner loop increments fastest.
- Only when the inner loop is complete does the outer loop variable increment.
- The inner loop counter is reset to its starting value each time the outer loop counter is incremented.

Activity 7.17

Create a new project, *Counting01*, which implements the nested `for` loop shown in the code snippet above.

Activity 7.18

What would be output by the following code?

```
for (no1 = -2; no1<=1; no1++)
      for (no2 = 0; no2<=3; no2++)
            cout << no1 << "  " << no2 << endl;
```

Activity 7.19

Modify *ExamMarks* so that 3 students each sit 5 exams. Display the average mark for each student.

Modify the program to show the overall average for all three students.

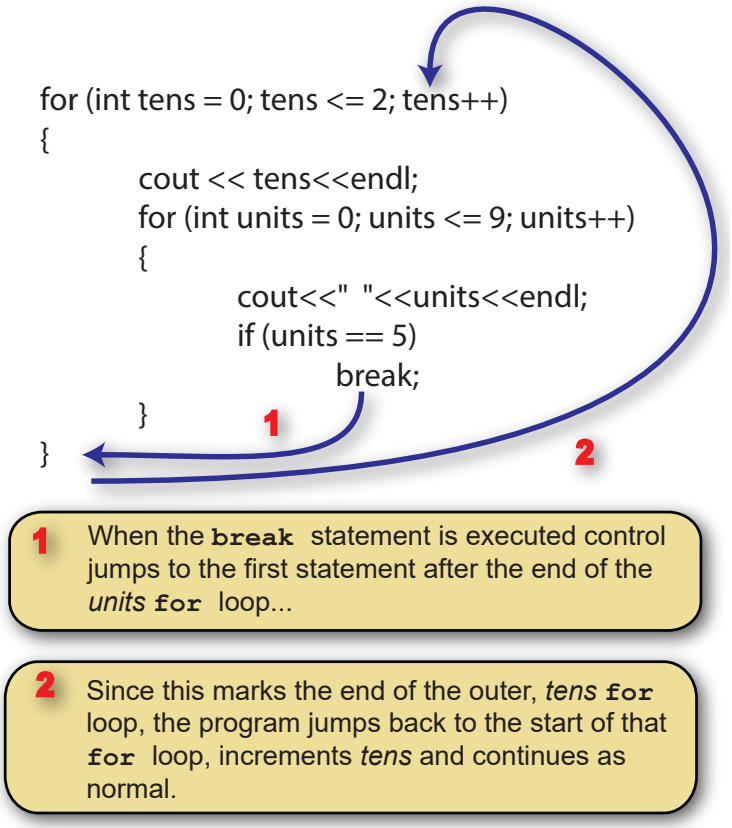
There's no practical limit to how many loops statements can be nested inside each other, but the more deeply nesting occurs, the more work will be involved in testing and the more likely we are to make a mistake.

Using *break* and *continue* within Nested Loops

Where we have a nested loop structure and a `break` or `continue` statement is placed within the loop body of an inner loop, then control jumps to the end of that inner loop only, execution of the outer loop continues as normal (see FIG-7.15).

FIG-7.15

Jump Statements Within Nested Loops



Activity 7.20

Start a new project, *Counting02*, and create a complete program based on the code given in FIG-7.15. Before you run the program, write down what output you think it will produce.

Summary

- Iterative Structures

```
while (expression)
{
   statements;
}
```

```
do
{
   statements;
}
while (expression);
```

```
for (statement1 ; expression; statement2)
{
   statements;
}
```

- In the `while` and `for` statements *expression* is tested before the loop body is executed.
- In the `do...while` statement *expression* is tested after the loop body is executed.
- In the `while` and `for` statements the loop body may be executed a minimum of zero times.
- In the `do...while` statement the loop body must be executed at least once.
- All loops exit when expression is *false*.
- A `for` statement has the structure

```
for ( statement1 ; expression ; statement2)
```

- `statement1`, `expression` and `statement2` can be omitted from a `for` statement but the semicolons must remain.
- Omitting `expression` from a `for` statement will cause an infinite loop.
- `statement1` and `statement2` may contain multiple statements each separated by a comma.
- When used in a loop body, executing the `break` statement transfers control to the first statement after the loop body.
- When used in a loop body, executing the `continue` statement will skip the remaining statements of the loop body before returning to the start of the loop `for` the next iteration. If placed in a `for` loop, `statement2` will be executed as normal.

- The term **nested loop** refers to the situation where a second loop structure is placed within the body of an existing loop.
- In a nested loop, the first loop structure is referred to as the **outer loop**, and the second loop as the **inner loop**.
- When `break` or `continue` is placed within the inner loop of a nested loop structure, the jump is confined to that inner loop.
- The `goto` statement can be used to jump control to a labelled statement.
- Use of the `goto` statement should be avoided.

Solution

Activity 7.1

Run 1: The `while` loop will be executed zero times since the data is within the range 1 to 12. The output will be

Number accepted was 8

Run 2: The `while` loop will be executed once, causing the error message to be displayed and requiring a second value to be entered. The second value is accepted and the output will be

Number accepted was 3

Run 3 : The `while` loop will be executed 3 times since the first three values are invalid. The last value is accepted and the output will be

Number accepted was 7

Activity 7.2

The output will be:

Run 1 : You entered 0 invalid values

Run 2: You entered 1 invalid values

Run 3: You entered 3 invalid values

Activity 7.3

Code for *DoLoop* (Version 1):

```
#include <iostream>
using namespace std;

int main()
{
  int number, sum = 0;

  do
  {
    //*** Get number ***
    cout << "Enter number : ";
    cin >> number;

    //*** Add number to total ***
    sum += number;
  }
  while (number != 0);

  //*** Display total ***
  cout << "Total of values entered was " << sum
  ↳<< endl;
}
```

Code for *DoLoop* (Version 2):

```
#include <iostream>
using namespace std;

int main()
{
  int number, sum = 0;

  do
  {
    //*** Get number ***
    cout << "Enter number : ";
    cin >> number;

    //*** Add number to total ***
    sum += number;

    //*** Display running total ***
    cout << "Total so far " << sum << endl;
  }
  while (number != 0);

  //*** Display total ***
  cout << "Total of values entered was " << sum
       ↳<< endl;
}
```

Activity 7.4

Modified code for *GuessingGame* (Version 1):

```
#include <iostream>
#include "ctime"
using namespace std;

int main()
{
  int number, guess;

  //*** Generate random numb in range 1 to 10 ***
  srand(static_cast<unsigned>(time(0)));
  number = rand()%10 + 1;

  //*** Read guess ***
  cout << "Guess the number (between 1 and 10):";
  cin >> guess;
  while (guess < 1 || guess > 10)
  {
    cout<<"Value must be 1 to 10. Re-enter: ";
    cin >> guess;
  }
  //*** Display message ***
  if (guess < number)
    cout << "Guess too low\n";
  else if ( guess > number)
    cout << "Guess too high\n";
  else
    cout << "Guess correct\n";

  //*** Display the number and guess
  cout << "The number was " << number << endl;
  cout << "Your guess was " << guess << endl;
}
```

Modified code for *GuessingGame* (Version 2):

```
#include <iostream>
#include "ctime"
using namespace std;

int main()
{
  int number, guess;
  int numberofguesses = 0;

  //*** Generate random numb in range 1 to 10 ***
  srand(static_cast<unsigned>(time(0)));
  number = rand()%10 + 1;

  //*** Continue guessing until correct ***
  do
  {
    //*** Read guess ***
    cout << "Guess the number (between 1 and 10):";
    cin >> guess;
    while (guess < 1 || guess > 10)
    {
      cout<<"Value must be 1 to 10. Re-enter: ";
      cin >> guess;
    }
    //*** Increment number of guesses made ***
    numberofguesses ++;

    //*** Display message ***
    if (guess < number)
      cout << "Guess too low\n";
    else if ( guess > number)
      cout << "Guess too high\n";
    else
      cout << "Guess correct\n";
  }
  while ( guess != number);

  //*** Display the number and guess
  cout << "The number was " << number << endl;
  cout << "Your guess was " << guess << endl;
  cout << "You had " << numberofguesses <<
       ↳" guesses\n";
}
```

Activity 7.5

Code for *ExamMarks* (with average):

```
#include <iostream>
using namespace std;

int main()
{
  int mark ;       //exam mark achieved by student
  int total=0;     //total for all exams
  int counter;     //for loop counter
  double average;  //average for all exams

  //*** FOR 5 times DO ***
  for(counter = 1; counter <= 5; counter++)
  {
    //*** Get mark ***
    cout << "Enter exam mark : ";
    cin >> mark;

    //*** Add mark to total ***
    total += mark;
  }

  //*** Calculate average mark ***
  average = total / 5;
  //*** Display total for all exams and average***
  cout << "Total for all 5 exams is " << total
       << endl;
  cout << "Average for all exams is " << average
       << endl;
}
```

Modified code for *ExamMarks* (4 Exams version):

```
#include <iostream>
using namespace std;

int main()
{
     int mark ;                    //exam mark
achieved by student
     int total=0;            //total for all exams
     int counter;            //for loop counter
     double average; //average for all exams

     //*** FOR 4 times DO ***
     for(counter = 1; counter <= 4; counter++)
     {
            //*** Get mark ***
            cout << "Enter exam mark : ";
            cin >> mark;

            //*** Add mark to total ***
            total += mark;
     }

     //*** Calculate average mark ***
     average = total / 4;

     //*** Display total for all exams and average***
     cout << "Total for all 4 exams is " << total
                  << endl;
     cout << "Average for all exams is " << average
                  << endl;
}
```

Modified code for *ExamMarks* (named constant):

```
#include <iostream>
using namespace std;

int main()
{
  const int NOOFEXAMS = 4;

  int mark ;       //exam mark achieved by student
  int total=0;     //total for all exams
  int counter;     //for loop counter
  double average;  //average for all exams

  //*** FOR NOOFEXAMS times DO ***
  for(counter = 1; counter <= NOOFEXAMS; counter++)
  {
    //*** Get mark ***
    cout << "Enter exam mark : ";
    cin >> mark;

    //*** Add mark to total ***
    total += mark;
  }

  //*** Calculate average mark ***
  average = total / NOOFEXAMS;

  //*** Display total for all exams and average***
  cout << "Total for all " << NOOFEXAMS <<
       " exams is " << total << endl;
  cout << "Average for all exams is " << average
       << endl;
}
```

To change the number of exams from 4 to 6, only the line assigning a value to the constant NOOFEXAMS needs to be changed:

```
const int NOOFEXAMS = 6;
```

Activity 7.6

Code for *Countdown*:

```
#include <iostream>
using namespace std;

int main()
{
  int c;

  //*** Display the values 10 down to 0 ***
  for(c=10; c >= 0; c--)
    cout << c << endl;
}
```

Activity 7.7

Code for *Alphabet*:

```
#include <iostream>
using namespace std;

int main()
{
  char ch;

  //*** Display the letters a to z ***
  for(ch='a'; ch <= 'z'; ch++)
    cout << ch << " ";
  cout << endl;
}
```

Activity 7.8

Run 1: The average displayed is 7 not 7.6667.
Run 2: The average displayed is 29 as expected
Run 3: The program fails

The first run highlights a problem with the line

```
average = total / count;
```

because *total* and *count* are both integers, we get an integer result, even though *average* is a `double` variable.

To solve this, we need to cast one of the variables to be of type `double`:

```
average = static_cast<double>(total) / count;
```

The second problem is caused by the same line of code as the first. When no valid numbers have been entered (Run 3) then *count* will be zero. Attempting to divide by zero results in a run-time error causing the program to abort.

To solve this we need to make sure that the average calculation, and its display is only performed when at least one value has been entered.

The revised version of *ForLoop*:

```
#include <iostream>
using namespace std;

int main()
{
```

```
   int no = 99;     //Value entered (start with dummy
                    ↳value)
   int total = 0;   //Total of all values entered
   int count;       //Count of the number of values
                    ↳entered
   double average;  //Average of the values
                    ↳entered

   //*** Get a set of values ending with zero ***
   //*** and count how many numbers are entered ***
   for (count = 0 ; no != 0 ; count++)
   {
      //*** Get number ***
      cout << "Enter number ";
      cin >> no;
      //*** Add number to total ***
      total += no;
   }
   //*** Reduce the count by 1 ***
   count--; // Don't count zero as a number
   if (count > 0)
   {
      //*** Calculate the average ***
      average = static_cast<double>(total) / count;
      //*** Display the average ***
      cout <<"Average of the "<<count<<" numbers is "
      ↳<< average << endl;
   }
   else
      cout << "No values were entered\n";
}
```

Activity 7.9

Modified code for *GuessingGame*:

```
#include <iostream>
#include "ctime"
using namespace std;

int main()
{
   int number, guess;
   int numberofguesses = 0;

   //*** Generate random numb in range 1 to 10 ***
   srand(static_cast<unsigned>(time(0)));
   number = rand()%10 + 1;

   //*** FOR 5 times DO ***
   for (numberofguesses = 1 ; numberofguesses <= 5;
   ↳numberofguesses++)
   {
      //*** Read guess ***
      cout << "Guess the number (between 1 and 10):";
      cin >> guess;
      //*** Display message ***
      if (guess < number)
         cout << "Guess too low\n";
      else if ( guess > number)
         cout << "Guess too high\n";
      else
         cout << "Guess correct\n";
   }

   //*** Display the number and guess ***
   cout << "The number was " << number << endl;
   cout << "Your guess was " << guess << endl;
}
```

The line displaying the number of guesses has been removed since 5 guesses are always given.

The problem is that even when it requires less than five guesses to arrive at the correct number, the player must continue to enter guesses until the five iterations are complete.

Activity 7.10

Code for *OneTo*:

```
#include <iostream>
using namespace std;

int main()
{
   int number; // Value entered
   int c;    //Loop counter

   //*** Read in valid number ***
   cout << "Enter number (0 to 50) ";
   cin >> number;
   while (number < 0 || number > 50)
   {
      cout << "Must be in range 0 to 50. Re-enter: ";
      cin >> number;
   }

   //*** Display values between 1 and number ***
   for(c = 1; c<= number; c++)
      cout << c <<endl;
}
```

Activity 7.11

Run 1 : Only 1 value is entered. The total is given as 0.

Run 2 : Only 2 values are entered. Total 3.

Run 3 : All 5 values are entered. Total 23.

Activity 7.12

Modified code for *GuessingGame*:

```
#include <iostream>
#include "ctime"
using namespace std;

int main()
{
   int number, guess;
   int numberofguesses = 0;

   //*** Generate random numb in range 1 to 10 ***
   srand(static_cast<unsigned>(time(0)));
   number = rand()%10 + 1;

//*** FOR 5 times DO ***
for (numberofguesses = 1 ; numberofguesses <= 5;
↳numberofguesses++)
   {
      //*** Read guess ***
      cout << "Guess the number (between 1 and 10):";
      cin >> guess;
      //*** Display message ***
      if (guess < number)
         cout << "Guess too low\n";
      else if ( guess > number)
         cout << "Guess too high\n";
      else
      {
         cout << "Guess correct\n";
         break;
      }
   }

   //*** Display the number and guess ***
   cout << "The number was " << number << endl;
   cout << "Your guess was " << guess << endl;
   if (numberofguesses < 6)
      cout << "You had " << numberofguesses
      ↳<< " guesses\n";
   else
      cout << "You failed to guess the number\n";
}
```

Activity 7.13

By changing the `break` statement to `continue`, five values are required to be entered on each run of the program even when the correct number is arrived at with fewer guesses.

Activity 7.14

Code for *Extreme* (lowest value):

```
#include <iostream>
using namespace std;

int main()
```

```
{
  int smallest; //Stores smallest value found
  int no;       //Store current number read
  int c;      //Loop counter

  //*** Read in first value ***
  cout << "Enter value ";
  cin >> no;
  //*** Record as smallest value ***
  smallest = no;
  //*** Read in 7 more values ***
  for(c = 1; c <= 7; c++)
  {
    cout << "Enter value ";
    cin >> no;
    //*** If it's smallest so far, record it ***
    if(no < smallest)
      smallest = no;
  }
  //*** Display smallest value found ***
  cout << "Lowest value entered was " << smallest
  ↳<< endl;
}
```

Modified code for *Extreme* (largest):

```
#include <iostream>
using namespace std;

int main()
{
  int largest;  //Stores largest value found
  int no;       //Store current number read
  int c;        //Loop counter

  //*** Read in first value ***
  cout << "Enter value ";
  cin >> no;
  //*** Record as largest value ***
  largest = no;
  //*** Read in 7 more values ***
  for(c = 1; c <= 7; c++)
  {
    cout << "Enter value ";
    cin >> no;
    //*** IF it's the largest so far, record it ***
    if(no > largest)
      largest = no;
  }
  //*** Display largest value found ***
  cout << "Highest value entered was " << largest
  ↳<< endl;
}
```

Modified code for *Extreme* (largest and smallest):

```
#include <iostream>
using namespace std;

int main()
{
  int smallest; //Stores smallest value found
  int largest;  //Stores largest value found
  int no;       //Store current number read
  int c;        //Loop counter

  //*** Read in first value ***
  cout << "Enter value ";
  cin >> no;

  //*** Record as smallest and largest value ***
  smallest = no;
  largest = no;
  //*** Read in 7 more values ***
  for(c = 1; c <= 7; c++)
  {
    cout << "Enter value ";
    cin >> no;
    //*** If it's smallest or largest so far,
    ↳record it ***
    if(no < smallest)
      smallest = no;
    else if (no > largest)
      largest = no;
  }
  //*** Display smallest and largest value found ***
  cout << "Lowest value entered was  " << smallest
  ↳<< endl;
  cout << "Highest value entered was " << largest
  ↳<< endl;
}
```

Activity 7.15

Code for *IceScore*:

```
#include <iostream>
using namespace std;

int main()
{
  int n;                //Number of judges
  double score;         //Score achieved
  double total;         //Total of all scores entered
  double average;       //Average score
  double smallest, largest; //Lowest & highest
  ↳scores
  int c;                //Loop counter

  //*** Get number of judges ***
  cout << "Enter number of judges ";
  cin >> n;
  //*** Get first score ***
  cout << "Enter score for judge 1 : ";
  cin >> score;
  //*** Record score as lowest and highest ***
  smallest = score;
  largest = score;
  //*** Set total to score ***
  total = score;
  //*** Get score for each judge ***
  for (c = 2; c <= n; c++)
  {
    //*** Get valid score ***
    cout << "Enter score for judge " << c << " : ";
    cin >> score;
    //*** Add score to total ***
    total += score;
    //*** IF score smallest or largest, record it
    ↳***
    if (score < smallest)
      smallest = score;
    else if (score > largest)
      largest = score;
  }
  //*** Subtract lowest and highest from total ***
  total -= (smallest + largest);
  //*** Calculate average ***
  average = total / (n-2);
  //*** Display average score ***
  cout << "Skater's average score is : " << average
  ↳<<endl;
}
```

Activity 7.16

Modified code for *ExamMarks:*

```
#include <iostream>
using namespace std;

int main()
{
  const int NOOFEXAMS = 4;

  int mark ;       //exam mark achieved by student
  int total=0;     //total for all exams
  int counter;     //for loop counter
  double average;  //average for all exams

  for(counter = 1; counter <= NOOFEXAMS; counter++)
  {
    //*** Get valid mark ***
    cout << "Enter exam mark : ";
    cin >> mark;
    while (mark < 0 || mark > 100)
    {
      cout << "Mark must be 0 to 100. Re-enter : ";
      cin >> mark;
    }
    //*** Add mark to total ***
    total += mark;
  }
  //*** Calculate average mark ***
  average = total / NOOFEXAMS;
  //*** Display total for all five exams and
  average***
  cout << "Total for all " << NOOFEXAMS <<
```

```
    ↳" 4 exams is " << total << endl;
    cout << "Average for all exams is " << average
    ↳<< endl;
}
```

Activity 7.17

Code for *Counting01*:

```
#include <iostream>
using namespace std;

int main()
{
  int units, tens; //Loop counters

  for (tens = 0; tens <= 9; tens++)
    for (units = 0; units <= 9; units++)
      cout << tens << units << endl;
}
```

Activity 7.18

Output would be:

```
-2 0
-2 1
-2 2
-2 3
-1 0
-1 1
-1 2
-1 3
 0 0
 0 1
 0 2
 0 3
 1 0
 1 1
 1 2
 1 3
```

Activity 7.19

Modified code for *ExamMarks:*

```
#include <iostream>
using namespace std;

int main()
{
  const int NOOFEXAMS = 5;
  const int NOOFSTUDENTS = 3;

  int mark ;       //exam mark achieved by student
  int total=0;  //one student's total for all exams
  int grandtotal=0; //Total of totals
  int counter;     //for loop counter
  int student;     //Outer for loop counter
  double average;  //one student's average
  double overallaverage; //class average

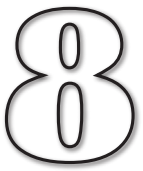
  for (student = 1 ; student <= NOOFSTUDENTS;
  ↳student++)
  {
    total = 0;
    for (counter = 1; counter <= NOOFEXAMS;
    ↳counter++)
    {
      //*** Get valid mark ***
      cout << "Enter exam mark : ";
      cin >> mark;
      while (mark < 0 || mark > 100)
      {
        cout << "Mark must be 0 to 100. Re-enter :
        ↳";
        cin >> mark;
      }
      //*** Add mark to total ***
      total += mark;
    }
    //*** Calculate average mark ***
    average = static_cast<double>(total) /
    ↳NOOFEXAMS;
    //*** Display total and average***
    cout << "Student's total for all " << NOOFEXAMS
    ↳<< " exams is " << total << endl;
    cout << "Student's average for all exams is "
    ↳<< average << endl;
    //*** Add total to grandtotal ***
    grandtotal += total;
  }
  //*** Calculate class average ***
  overallaverage = static_cast<double>(grandtotal)
  ↳/ (NOOFSTUDENTS * NOOFEXAMS);
  //*** Disaplay overall average ***
  cout << "Class's overall average exam score is "
  ↳<< overallaverage << endl;
}
```

Activity 7.20

Code for *Counting02*:

```
#include <iostream>
using namespace std;

int main()
{
  for (int tens = 0; tens <= 2; tens++)
  {
    cout << tens << endl;
    for (int units = 0; units <= 9 ; units++)
    {
      cout << "  " << units << endl;
      if (units == 5)
        break;
    }
  }
}
```



Arrays

In this Chapter:

- ❑ Array Concepts
- ❑ One Dimensional Arrays
- ❑ Initialising Arrays
- ❑ Accessing Array Elements
- ❑ Arrays for Multiple Counts
- ❑ Arrays for Non-Repeating Random Values
- ❑ Arrays for Shuffling
- ❑ Handling Partially Full Arrays
- ❑ Searching an Array
- ❑ Inserting and Deleting within an Array
- ❑ Character Arrays
- ❑ Multi-Dimensional Arrays

Arrays

Problems with Simple Variables

Back in Chapter 4 we had a look at the basic data types available in C++. Although variables of these types served us well enough for the programs we have created so far, they are, in fact, quite limiting in what they allow us to achieve with minimal coding effort. For example, while it is easy enough to read and display five numbers using the code:

```
for (int c = 1; c <= 5; c++)
{
   cout << "Enter number : ";
   cin >> no;
   cout << no << endl;
}
```

how would we write a program which reads in all five numbers and then displays the numbers entered? We would need to start by declaring five variables in which to store the values:

```
int no1, no2, no3, no4, no5;
```

and then add code to read a value into each of these variables:

```
cout << "Enter number : ";
cin >> no1;
cout << "Enter number : ";
cin >> no2;
cout << "Enter number : ";
cin >> no3;
cout << "Enter number : ";
cin >> no4;
cout << "Enter number : ";
cin >> no5;
```

Notice that, because each value is stored in a different variable, we cannot use the `for` loop as we did with the solution to the previous problem.

Finally, we need five `cout` statement to display the five values:

```
cout << no1 << endl;
cout << no2 << endl;
cout << no3 << endl;
cout << no4 << endl;
cout << no5 << endl;
```

The code is a bit long-winded, but just about acceptable. Now imagine that we had the ten values to read in, store, and display. The code for that is going to become long – very long. Luckily, there is a better way to achieve what we're after – **arrays**.

One Dimensional Arrays

Array Concepts

An array is a named data variable capable of storing several values at the same time. It is a collection of **elements** or **cells**. Each of these elements holds a single value - just like the simple variables we have used in our previous programs.

Each cell within an array is numbered in ascending order starting at zero. This

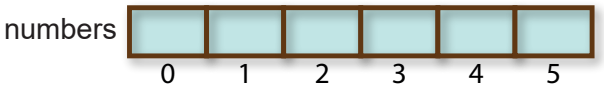
identifying number is known as a **subscript** or **index**.

Exactly how many cells an array contains is determined when the array is first declared.

FIG-8.1 shows how we might visualise a 6 element array called *numbers*.

FIG-8.1

Visualising an Array



The individual cells within the array are identified by a combination of the array name and the cell's index with the index enclosed within square brackets. For example, the second cell within the array shown above is identified using the term

```
numbers[1]
```

Declaring an Array

As with simple variables, we need to supply the compiler with a name as well as stating the type of values to be stored when declaring an array. The only additional piece of information required is the number of elements we want the array to contain. So, to set up the *numbers* array as shown in FIG-8.1 to hold integer values, we would use the declaration

```
int numbers[6];
```

Arrays can be declared to hold values of any of the types we can use for simple variables: `int`, `float`, `char`, etc. Every cell within the array can then hold a single value of the specified type. It is not possible to create an array with cells of differing types.

An array declaration can be added to the declaration of standard variables, so it is quite valid to use a line such as

```
int no1, sum, list[10];
```

which will not only set up the integer variables *no1* and *sum*, but also an array named *list* which contains 10 elements. And, of course, we can set up more than one array in a single declaration:

```
float max[15], min[15], data[50];
```

An array's size can also be set up using a named constant, or arithmetic expression containing constants with lines such as:

```
const SIZE{10};
float results[SIZE];
int values[SIZE*2];
```

However, you cannot create an array using a variable to define its size. For example, the code

```
int no{20};
int list[no];
```

will not compile.

Initialising Arrays

The contents of an array cell are, like the contents of a simple variable, undefined when first created. But it is possible to specify a starting value for each element when

declaring the array. For example, the lines

```
int numbers[6] {12, 0, -6, 8, 17, 14};
int numbers[6] = {12,0,-6,8,17,14};
```

are alternative ways of setting up the *numbers* array with the value required. However, the first method is to be preferred.

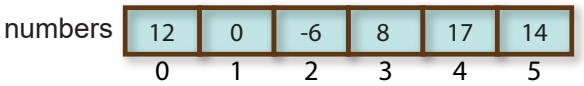
Note, that using parentheses as we did when assigning a value to a simple variable, is not allowed:

```
int numbers[6] (12, 0, -6, 8, 17, 14); // Not allowed
```

A valid declaration of *numbers* will create the array setup shown in FIG-8.2.

FIG-8.2

Initialising an Array



If there are too many values specified within the braces, the compiler will produce an error.

If there are too few values supplied, then the cells which have not specifically been assigned a value are set to zero (or *false* in the case of `bool` arrays). For example, the quick way to set all elements of an array to zero would be to use one of the following declarations:

```
int totals[10] {0};
int totals[10] {};
```

A final option when setting up arrays is to omit the number of cells and allow the compiler to deduce the number from the values given in the initialisation list. For example, the code

```
double values[]{12.8, 6.9, -23.098};
```

will create a three element array called *values*.

Should we require an array of values that cannot be modified later in the code (in other words, an array of constants), all that is required is that we add the keyword `const` at the start of the array declaration:

```
const double rates[] {5.3, 7.9, 12.8, 15,7};
```

Accessing Array Elements

Under most circumstances we cannot perform operations on an array as if it were a single entity. For example, it would be invalid to try to input values for every element in an array called *numbers* with a statement such as

```
cin >> numbers; //Error
```

or try to copy the contents of one array to another:

```
numbers = totals; //Error
```

Instead, we must deal with the individual elements within the array. So, to read in a value for the first element in the array *numbers*, we would specify the array name followed by the index of the cell to be accessed (which is enclosed in brackets):

```
cin >> numbers[0];
```

To assign a value to the next element we could use a statement such as

```
numbers[1] = 4;
```

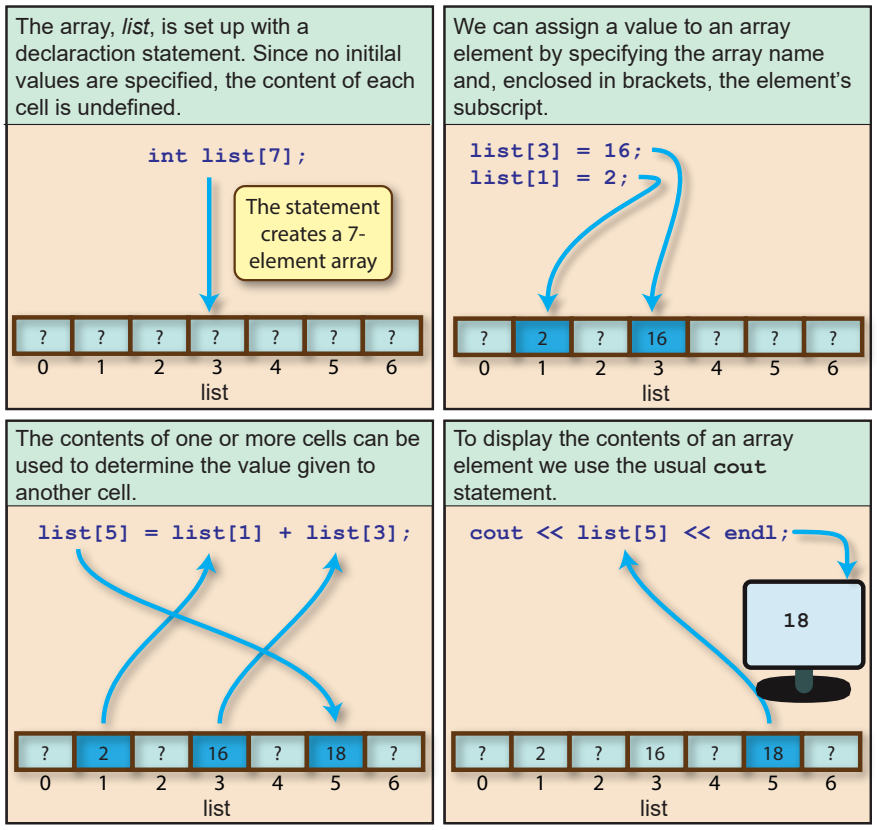
And we could check if the fourth element contained a value of less than zero with the line

```
if (numbers[3] < 0)
```

In fact, we can use an array element in any statement where we might use a simple variable of the same type. The effects of some simple array-accessing statements are shown in FIG-8.3.

FIG-8.3

Accessing an Array



We must ensure that the subscript we supply is a valid one; the compiler will not check that the subscript value is within a range compatible with the size of the array created. Hence, code such as

```
int list[10];

list[12]= 21;  //*** Subscript too high ***
```

will be accepted. If we go on to run the program, it will create unpredictable results.

Index Variables

If what we've seen up to now was all that could be achieved by arrays, they would be of little more use than simple variables. For example, our attempt to read in five values before displaying them would still be as long-winded as before:

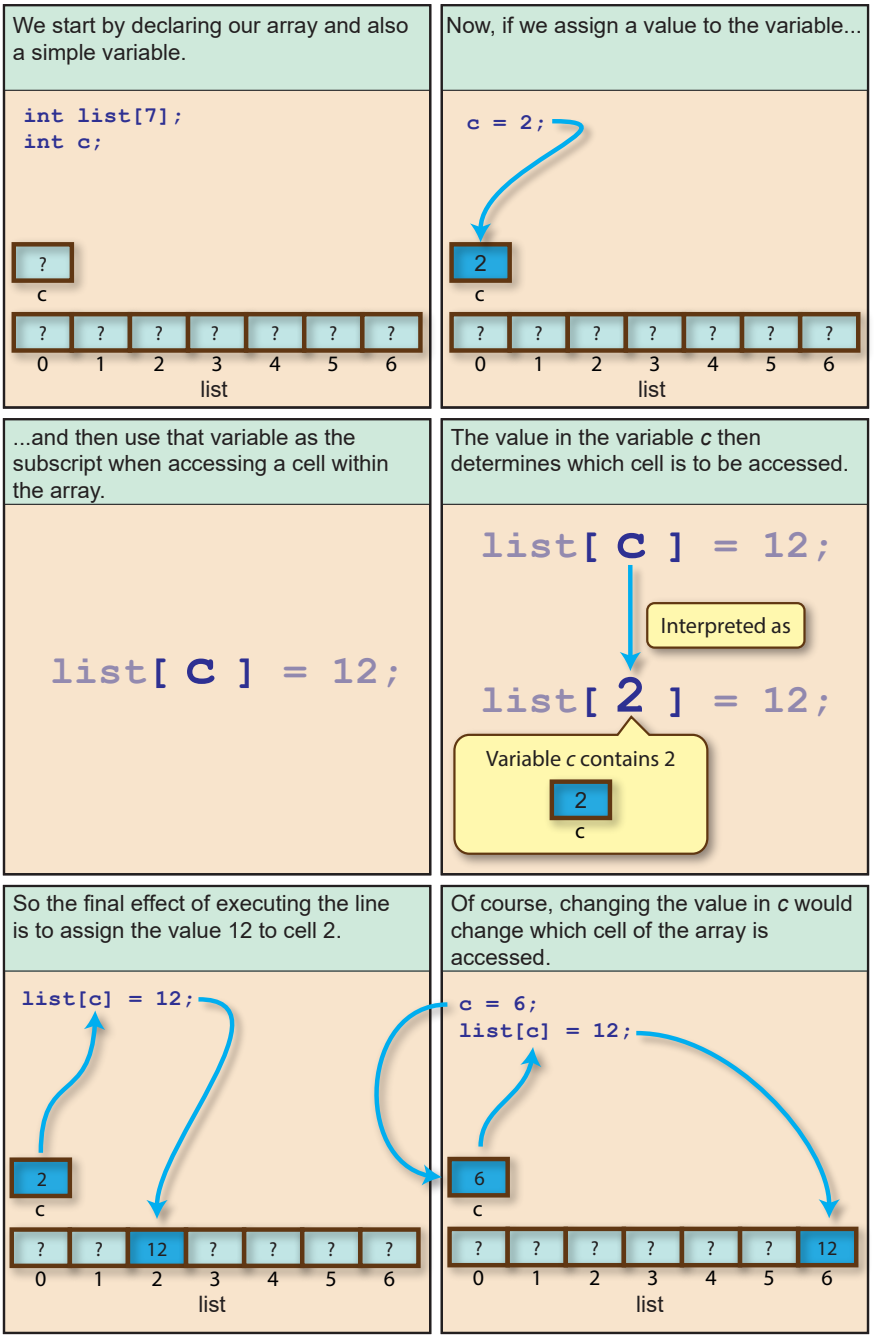
```
int nos[5];
cout << "Enter number : ";
cin >> nos[0];
cout << "Enter number : ";
cin >> nos[1];
```

```
cout << "Enter number : ";
cin >> nos[2];
cout << "Enter number : ";
cin >> nos[3];
cout << "Enter number : ";
cin >> nos[4];
```

What adds power to the array is the fact that the subscript need not be given as a fixed value. Instead, we are free to use a variable or an expression. The value of that variable (or expression) then determines the value of the subscript and hence which element within the array is to be accessed (see FIG-8.4).

FIG-8.4

Variable Subscripts



As we can see, if we set up a variable:

```
int c = 2;
```

and use that variable as the subscript when accessing the element of a previously declared array (*numbers*, in this case):

```
cin >> numbers[c]; // accesses element 2
```

then the actual element of the array that is to be accessed depends on the value of *c*. If *c*'s value changes:

```
c = 3;
```

then the element of the array being accessed also changes:

```
cin >> numbers[c];   //accesses element 3
```

We need take only one further step to realise how we can use arrays to create shorter code for our earlier problem of reading in five values. With the line

```
for(int c{0}; c < 5; c++)
```

the variable *c* will, as the loop repeats itself, take on the values 0 through to 4. So if we use that variable in conjunction with array access, we can read in our five values using the code

```
int numbers[5];

for (int c{0}; c < 5; c++)
{
   cout << "Enter number : ";
   cin >> numbers[c];
}
```

As *c* changes value each time the loop iterates, so the term `list[c]` in the `cin` statement will reference a different element of the array *list*.

Activity 8.1

Start a new project called *Arrays01* and, using the code given above, create a complete program which reads in five numbers storing the values in the five elements of the array *numbers*.

By adding a second `for` loop to your code, get the program to display the contents of the array.

We can improve on the sample code given above, helping to make it more robust against possible errors. Let's assume that at a later date we need our *numbers* array to contain 10 elements rather than only 5. This is easily fixed by changing the declaration line to:

```
int numbers[10];
```

However, as a consequence of this, we must also remember to change the upper limit of the `for` loop to read:

```
for (int c = 0; c < 10; c++)
```

And the second `for` loop, displaying the contents of the array (as added in the previous Activity) would also require a similar change.

We can make life a lot simpler by adding a named constant with a line such as

```
const int SIZE {5};
```

and then using this to set the size of the array as well as the limit of the `for` loops:

```
int numbers[SIZE];
for(int c = 0; c < SIZE; c++)
```

Now, when we need to modify the size of the array, all that is required is a change to the line specifying the value of *SIZE*:

```
const int SIZE {10};
```

Activity 8.2

Modify *Arrays01* so that a constant value, *SIZE*, is used to set the size of *numbers* and the limit of the `for` loops.

On the first run, set SIZE to 5. On the second run set size to 7. Check that the execution adjusts according to the value given.

We can also create more robust `for` loops for arrays where the number of cells has not been specified as in the line

```
int list[] {2,6,9,1,6,2,2,9};
```

We have already encountered the `sizeof` statement which determines the number of bytes assigned to a variable or type. That same command can be used to determine the total number of bytes occupied by an array. For example, we might write

```
cout << sizeof(list);
```

to discover the bytes allocated to our array, *list*. Dividing this figure by the size of a single element will tell us how many elements are in the array:

```
cout << sizeof(list)/sizeof(list[0]);
```

And, if we use the same expression in any `for` loop that traverses the array

```
for (int c{0}; c < sizeof(list) / sizeof(list[0]); c++)
   cout << list[c] << ' ';
cout << endl;
```

then our code will automatically cope with any change to the size of the array.

Activity 8.3

Create a new project, *Arrays02,* with an array, *list,* defined using empty brackets and containing the following set of values: {7,5,2,9,1,1,3,6,5,8}.

Write a `for` loop to display the contents of the array and use the `sizeof` command to determine the limit of the loop.

size()

But there is one other option for dealing with arrays of an unspecified size. This is the `size()` function (which is defined in the `std` namespace). This returns the number of elements in a container. In other words, using `size()` on our array *list* performs the same task as the expression `sizeof(list)/sizeof(list[0]`. This means we can rewrite our for loop as:

```
for (int c{0}; c < size(list); c++)
```

This line assumes `using namespace std;` is included in the program code, otherwise we must use the term `std::size(list)`

Activity 8.4

Modify *Arrays02* to use the `size()` function in its `for` statement.

Filling an Array with Random Values

If we don't want to specify the values held in an array as part of the initialisation code, we are left with the alternative of having to input each value required from the keyboard. And, although this may be acceptable if only a small number of values have to be entered, it becomes impractical if tens, or even hundreds of values are needed.

One way to get round this problem, particularly when setting up the contents of an array to test some other code, is to generate the contents of the array randomly.

How this is done uses features of C++ that we will not cover in depth until later but since it would be useful to have that knowledge now, we'll content ourselves with a statement of how it's done and leave the details of what's going on in the code until later.

To generate a random number, we need to add the following line:

```
#include <ctime>
```

Before we begin generating the numbers, we need to add the line

```
srand(static_cast<int>(time(0));
```



Actually, the compiler can cast the value automatically, so we can shorten the statement to:

```
srand(time(0));
```

And, finally, to generate the integer random number itself, we need to use the expression

```
rand()%max + 1;
```

where *max* is the maximum number we wish to generate (the smallest value being 1),

For example, the line

```
int number = rand()%100 + 1;
```

would set number to a random number in the range 1 to 100.

To set the contents of array *list* to a random values in the range 0 to 200 we could use the code shown in FIG-8.5.

FIG-8.5

Filling an Array with Random Values

```
#include <iostream>
#include <ctime>

using namespace std;

int main()
{
   int list[20];

   srand(static_cast<int>(time(0)));
   for (int c = 0; c < size(list); c++)
      list[c] = rand() % 201;
   cout << "CONTENTS OF list\n";
   for (auto v : list)
      cout << v << " ";
   cout << endl;
}
```

Activity 8.5

Start a project called *ArrayRandom* and implement the code given above.

Index Expressions

An array index can also be given in the form of an arithmetic expression. For example, the code

```
int values[20];
int p = 3;

values[p*2] = 42;
```

will store the value 42 in *values[6]* - which is the seventh cell within the array.

We can even use the contents of one cell as the subscript. So, in the code

```
int values[20];
values[0] = 9;
values[values[0]] = 4;
```

since, *values[0]* contains the value 9, the result will be to store the value 4 in *values[9]*.

Activity 8.6

State the contents of each cell in the array *numbers* after the following code has been executed.

```
int numbers [8] {0};
for (int p{0}; p < 8; p++)
    numbers[p] = p*2;
numbers[numbers[2]-1] = 23;
```

The Range-Based *for* Loop

Data structures (such as the array) which are capable of holding more than a single value are known as **containers**. Containers are said to contain a **range** of values.

The **range-based** `for` loop is designed specifically to iterate through each value in a range. The loop starts by copying the first value in a container to a named variable before executing the loop body. The loop continues by copying the next value in the container to the same named variable and again executes the loop body. This continues until all values in the container have been dealt with, at which point the `for` loop terminates.

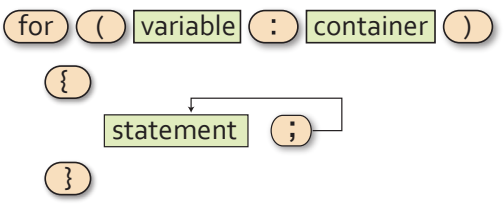
In the following example of a range-based `for` loop every value in the array *numbers* is copied to a variable *v* and displayed.

```
for(int v: numbers)
      cout << v << ' ';
cout << endl;
```

The format of the `for` statement is shown in FIG-8.6.

FIG-8.6

Range-Based for



where:

variable	is the name of the variable which is to hold the current value extracted from *container*. This variable should be of a type appropriate to those values held in *container*. Often *variable* will be declared at this point in the code.
container	is the name of the container (an array or other container type) whose values are to be copied into *variable*.
statement	is any valid C++ statement.


We'll meet other container types in later chapters.

If there is only one statement in the loop body, the braces may be omitted.

The following code will display every value held in the array *list* and also display the total of these values:

```
int total{};
for (int v: list)
{
   total += v;
   cout << v << endl;
}
cout << "Total: " << total << endl;
```

Note that the variable contains a copy of an element from the container so any changes made to the contents of the variable will not effect the contents of the container. For example, executing the code

```
for(int v: list)
      v+=2;
```

will have no effect on the contents of *list*.

Activity 8.7

Modify *Arrays02* to use the range-based `for` statement.

When declaring the variable designed to hold each value in our container, we can make use of the term `auto` and the compiler will derive the type needed from the declaration of the array:

```
for(auto v; list)
```

It is possible in a range-based `for` loop to specify the set of data that is to be worked through without assigning it to an array variable. For example, we could start our code with the line

```
for (auto v; {1,3,9,-5,7})
```

and have *v* take on each of the listed values in turn. Since the values are not assigned to any named array, the list of values is often referred to as an **anonymous array**.

Using One Dimensional Arrays

We've already seen how an array can be used to store values entered at the keyboard and then display those same values on the screen, but arrays can be used in many more ways. Some examples of how arrays can be used to help create efficient solutions to other problems are shown in this section.

Problem: Multiple Counts

A die is said to be biased if each number does not have the same likelihood of being thrown.

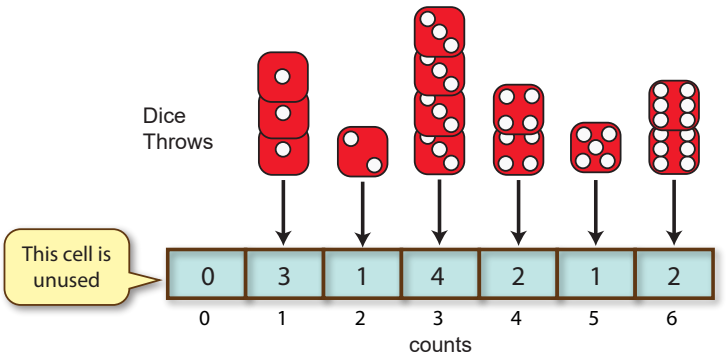
One of the tests used to make sure that a die is not biased is to check that, for a large number of throws, each number should appear approximately the same number of times. This will involve maintaining six separate counts; one for each value on the die.

Solution:

We can use an array to keep count of how often each number occurs. Cell 1 will hold a count of how often the number 1 is thrown, cell 2 the number of times 2 is thrown, etc (see FIG-8.7).

FIG-8.7

Arrays for Multiple Counts



Since the diagram shows only a small number of throws, the distribution of each number can vary more widely.

The structured English for our solution could be written as

```
FOR 1000 times DO
    Generate random number in the range 1 to 6
    Add 1 to appropriate count
ENDFOR
Display all 6 counts
```

The code for the program is given in FIG-8.8.

FIG-8.8

Using an Array for Multiple Counts

```
#include <iostream>
#include <ctime>

using namespace std;

//*** Count occurrence of each value on 10,000 die throws ***
int main()
{
   int counts[7]{ 0 };  //Count of each value
   int dicethrow;       //Current throw

   //*** Seed random number generator ***
   srand(static_cast<unsigned>(time(0)));
   //*** Throw the dice 10000 times ***
   for (int c = 1; c <= 10000; c++)
   {
      //*** Throw die ***
      dicethrow = rand() % 6 + 1;
      //*** Add to appropriate count ***
      counts[dicethrow]++;
   }
   //*** Display each count ***
   for (int c{1}; c < size(counts); c++)
      cout << c << " occurred " << counts[c] << " times" << endl;
}
```

Activity 8.8

Start a new project called *DieTest* then type in and test the program given above.

Some games make use of 10-sided dice. Modify your program so that it will generate numbers in the range 1 to 10 and count how often each value occurs.

Since the number of sides (or faces) on the die may be changed, modify the program to make use of a named constant, FACES, which specifies the number of sides on the die.

Problem: Generating Non-Repeating Random Values

Many countries run lottery systems. The simplest of these requires us to choose 6 numbers in the range 1 to 49.

Of course, we can easily get the computer to generate and display six numbers in this range, but we also need to make sure that none of the six numbers are the same.

Solution:

To ensure that there are no duplicate values, from the second number onwards we need to check that the generated number has not already been used. One way to do this is to start by setting up a cell for each possible number (49 in the case of the lottery), intialising every cell's value to zero.

When a number is generated, its corresponding cell has its value changed to 1.

To discover if a value has already been generated, all we need to do is check that number's cell; if it contains the value zero, the number has not been picked before. This approach is visualised in FIG-8.9.

FIG-8.9 How to Use an Array to Prevent Duplicated Numbers

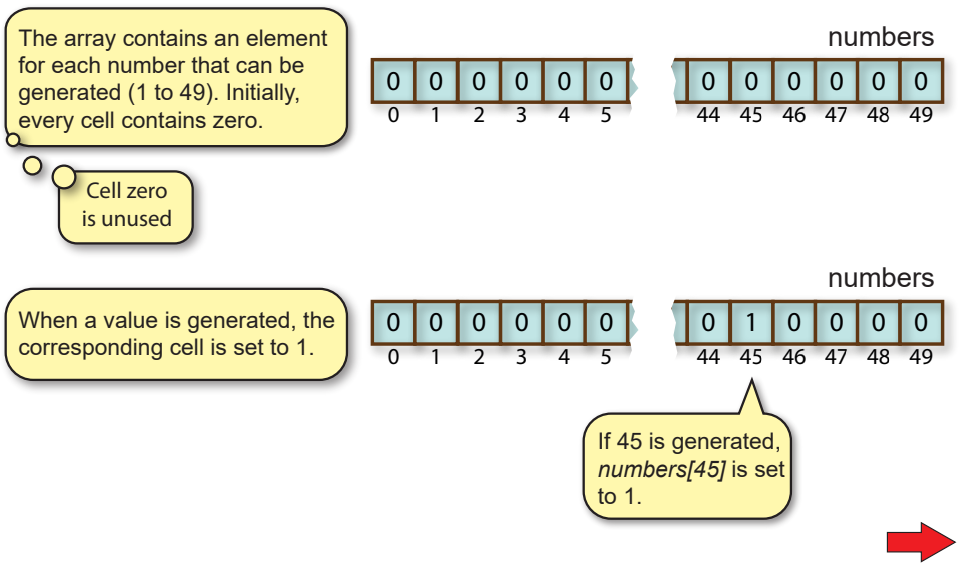
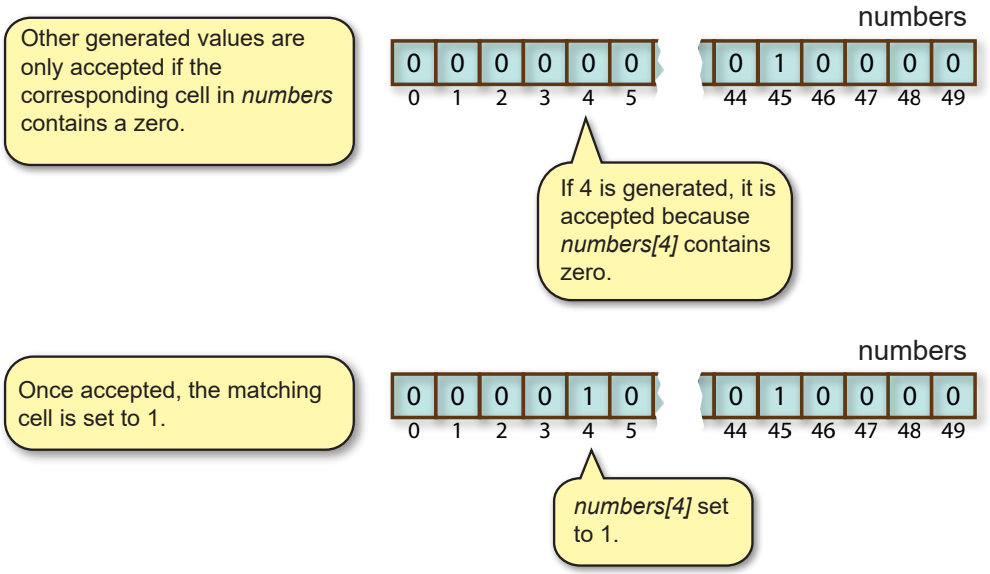


FIG-8.9 (continued) How to Use an Array to Prevent Duplicated Numbers



The structured English for our solution would be:

```
Create an array with cells 1 to 49 set to 0
Generate a random number in the range 1 to 49
Set the corresponding cell to 1
Display the value
FOR 5 times DO
    REPEAT
        Generate a random number (1 to 49)
    UNTIL the corresponding cell is zero
    Set the corresponding cell to 1
    Display the number
ENDFOR
```

The program for this is shown in FIG-8.10.

FIG-8.10

Creating Unique Random Values

```
#include <iostream>
#include <ctime>
using namespace std;

//*** Generates six unique lottery numbers ***
int main()
{
   const int HIGHEST{ 49 };         //Highest lottery number
   int lottery[HIGHEST + 1]{ 0 };   //Values already selected
   int number;                      //Currently generated number

   //*** Seed generator ***
   srand(static_cast<unsigned>(time(0)));

   //***Generate number ***
   number = rand() % HIGHEST + 1;
   //*** Set corresponding cell ***
   lottery[number] = 1;

   //*** Display value ***
   cout << number << ' ';
```

FIG-8.10
(continued)

Creating Unique Random Values

```
   //*** FOR 5 times DO ***
   for (int c = 1; c <= 5; c++)
   {
      //*** Generate an unselected number ***
      do
         //** Generate number **
         number = rand() % HIGHEST + 1;
      while (lottery[number] == 1);
      //*** Set corresponding cell ***
      lottery[number] = 1;
      //*** Display value ***
      cout << number << ' ';
   }
   cout << endl;
}
```

Activity 8.9

Start a new project, *Lottery*. Enter the code shown above and test the program.

There is really no need to treat the first number any differently from the remaining five. Modify the code so that all 6 numbers are generated within the `for` loop.

The code displays the numbers as they are generated rather than in ascending order. Modify the code so that the six numbers are displayed in ascending order. (HINT: You will need to remove the existing `cout` statements from the code.)

Problem: Shuffling

Many applications require items to be re-arranged in a random order. For example, most MP3 players offer a shuffle option which will play music tracks in a random order and shuffling is mandatory for almost every card game.

Solution:

If we start by storing a set of values in an array (these may represent music track numbers or playing card values). We then take the values at two randomly selected positions within the array and swap them over. If we continue to do this many more times, the items will have been effectively shuffled. FIG-8.11 demonstrates the technique involved.

FIG-8.11

Shuffling Values

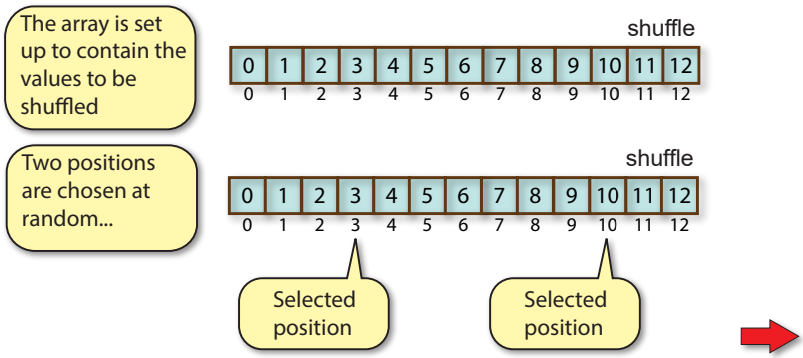
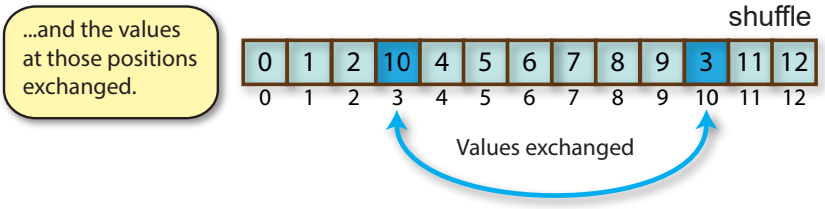


FIG-8.11
(continued)

Shuffling Values



The structured English is:

```
Set all values up within an array
FOR 200 times DO
    Generate a first random subscript.
    Generate a second random subscript
    Swap the values held at the subscript positions
ENDFOR
```

The code for shuffling an array of 20 values is given in FIG-8.12.

FIG-8.12

Implementing Shuffling

```
#include <iostream>
#include <ctime>

using namespace std;


int main()
{
   const int SIZE{20};  //Size of array

   int list[SIZE];      //Holds values to be shuffled
   int sub1, sub2;      //Indexes of two values to be swapped
   int temp;            //Temporary storage during swap process

   //*** Seed generator ***
   srand(static_cast<unsigned>(time(0)));

   //*** Set up values in array ***
   for( int c{0}; c < SIZE; c++)
      list[c] = c;

   //*** Shuffle ***
   for(int c = 1; c <= 200; c++)
   {
      //*** Generate two subscript values ***
      sub1 = rand()%SIZE;
      sub2 = rand()%SIZE;
      //*** Swap values at these positions ***
      temp = list[sub1];
      list[sub1] = list[sub2];
      list[sub2] = temp;
   }

   //*** Display shuffled items ***
   cout << "Order after shuffle :\n";
   for(int c {0}; c < SIZE; c++)
      cout << list[c] << " ";
   cout << endl;
}
```

Activity 8.10

Start a new project called *Shuffle* then type in and test the program given in FIG-8.11.

Modify the program so that the contents of *list* are displayed before and after the shuffle.

To simulate a card pack, we would need a 52 element list. The numbers 0 to 12 could represent ace to king of hearts; 13 to 25 diamonds; 26 to 38 spades; and 39 to 51 clubs. Modify the program to shuffle a deck of cards.

It would be better if we could display the value and suit of a card rather than just a number. For example, displaying 2 of diamonds rather than 14. Modify your program to do this for the first six cards in the shuffled pack. *(For the moment, the Ace, Jack, Queen and King of a suit can be displayed as 1, 11, 12, and 13 of the appropriate suit.)*
(HINT: Use the division and modulo operators (`/` and `%`) to determine the suit and value of a card.)

A final improvement would be to display the names *Ace*, *Jack*, *Queen* and *King* as appropriate. Use a `switch` to implement this change to the program.

Problem: Handling an Array that is not Full

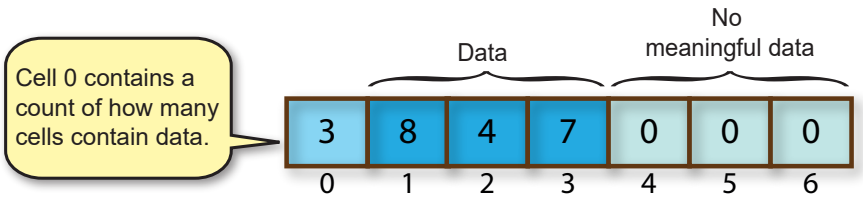
There are times when we will set up an array with space enough to hold a specific number of values, but initially not all the cells will contain meaningful data. For example, a game which remembers the top 5 scores, will contain no top scores when the game is first played. In these situations, we may want to access only the elements of the array in which data has already been placed.

Solution:

One way to handle this problem is to use element zero in the array to keep a count of how many cells in the array contain data (see FIG-8.13).

FIG-8.13

Counting Data Values



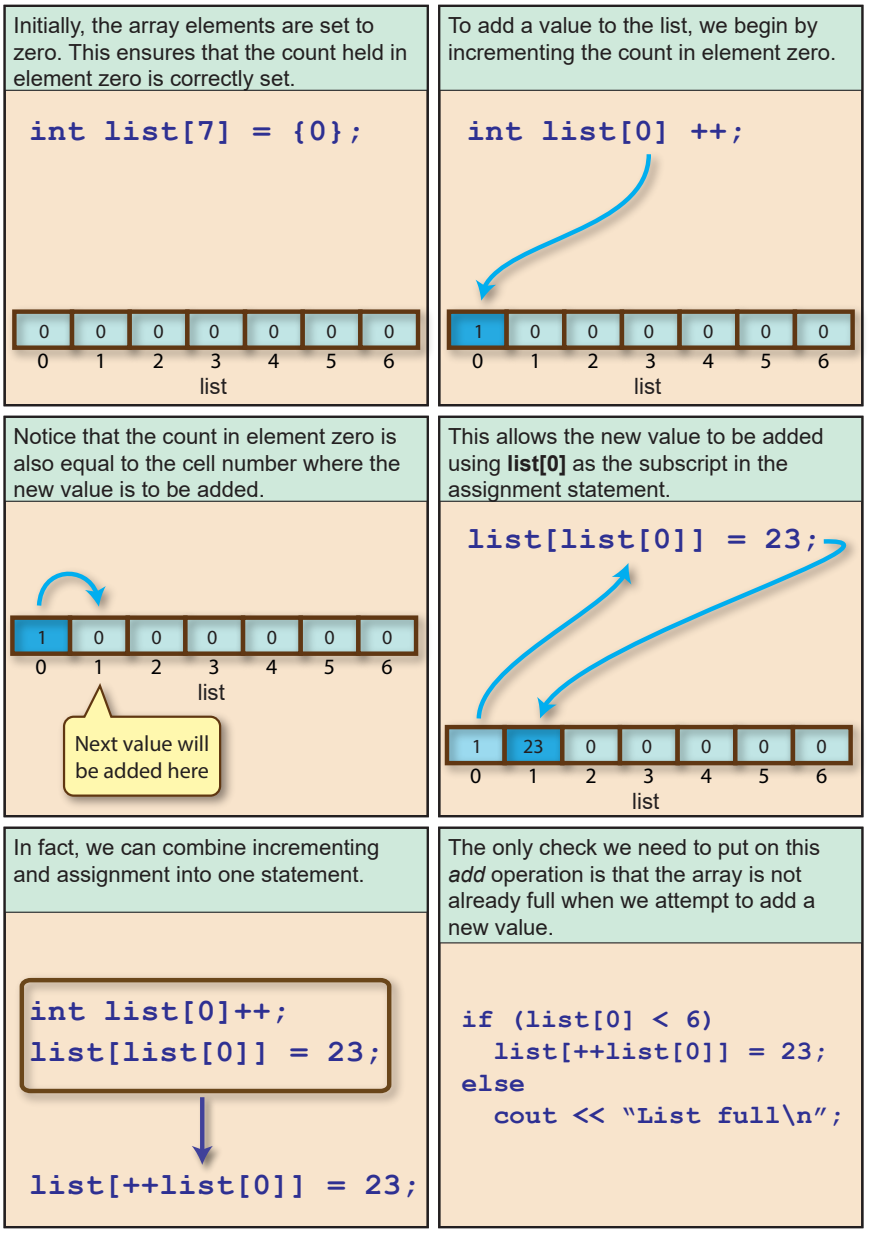
The logic required to insert a value into the list can be written in structured English as:

```
Get value to be added
IF the list is not full THEN
    Add 1 to the count
    Insert the new value at the end of the existing data
ELSE
    Display message "List is full"
ENDIF
```

The main steps involved in this setup are shown in FIG-8.14.

FIG-8.14

Implementing a Count-Based Array



A menu-driven program demonstrating how the values are added to an array using the technique described above is given in FIG-8.15.

Note that the program allows four options: add a value; display how many values are held; display the values held; and quit.

```
#include <iostream>
using namespace std;

int main()
{
const int SIZE{};       //*** Subscript of last cell in array
```

FIG-8.15

Implementing A Count-Based Array

```
   int list[SIZE + 1] {0}; //*** Array holding values entered
   int value;              //*** Value to be added to array
   int option;             //*** Menu option selected

   //*** Repeat until quit selected ***
   do
   {
      //*** Display menu ***
      cout << "1 - Enter value\n";
      cout << "2 - Display number of values held\n";
      cout << "3 - Display all values held\n";
      cout << "4 - QUIT\n";

      //*** Get option ***
      cout << "Enter option required (1 - 4) : ";
      cin >> option;
      while (option < 1 || option > 4)
      {
         cout << "Invalid option. Re-enter : ";
         cin >> option;
      }
      cout << "\n\n";

      //*** Execute option ***
      switch (option)
      {
         case 1:  //*** Add a new value to the list ***
            cout << "Enter value to be added : ";
            cin >> value;
            if (list[0] < SIZE)
               list[++list[0]] = value;
            else
               cout << "List is full\n\n\n";
            break;
         case 2:  //*** Display number of items in the list ***
            cout << "List contains "<<list[0]<<" entries\n\n\n";
            break;
         case 3:  //*** Display the contents of the list ***
            if (list[0] == 0)
               cout << "The list is empty\n\n";
            else
            {
               cout << "Values held are\n";
               for(int c = 1; c <= list[0]; c++)
                  cout << list[c] << "  ";
               cout << "\n\n\n";
            }
            break;
         case 4:  //*** Terminate the program ***
            cout << "Program terminated\n";
      }
   }
   while (option != 4);
}
```

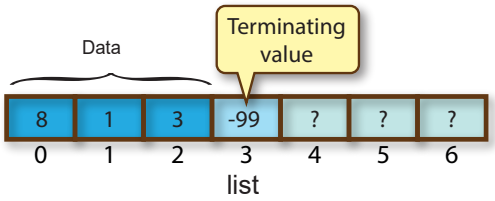
Activity 8.11

Create a project, *ManageArray01,* and implement the code given in FIG-8.14. What happens when you attempt to add another value when the array is full?

Keeping a count of the number of entries in an array is only one way of handling the problem of keeping tabs on just how many elements within an array contain meaningful data. A second approach is to use a "marker" value in the cell following the last value held in the array. For example, we might follow the actual data held by a value of, say, -99 (see FIG-8.16).

FIG-8.16

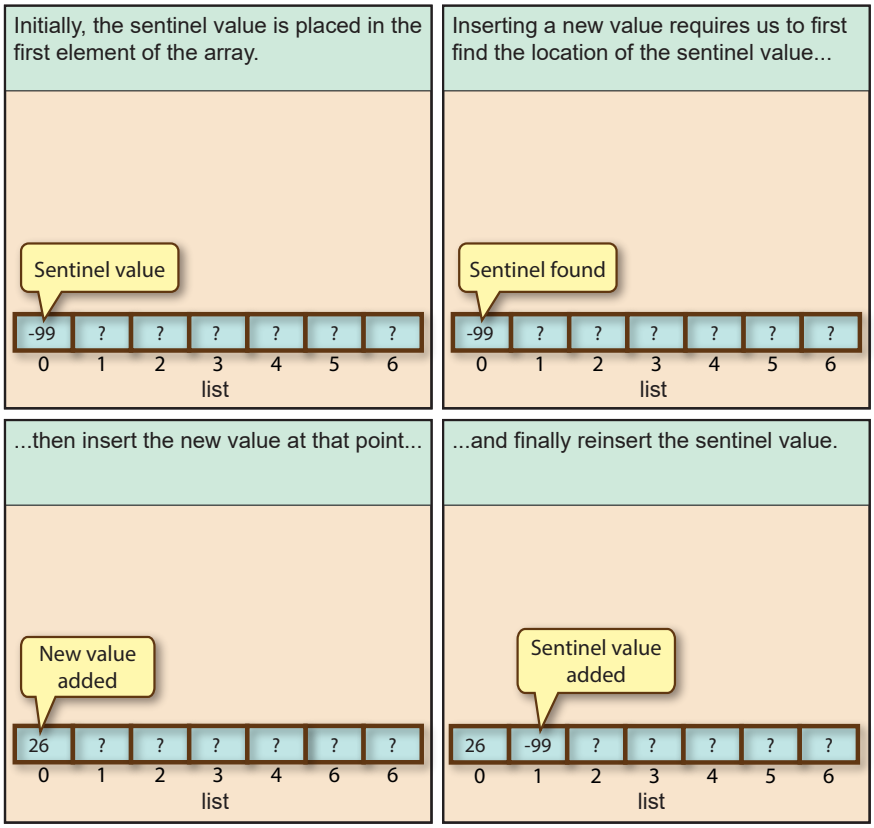
A Sentinel-Based Array



The marker value, often known as the **sentinel**, must be chosen with care. Obviously, we cannot use a value which can occur within the actual data since such an occurrence would mean that the it would be assumed to be the terminating value. For example, we could safely choose the value -1 as a terminating value if we were sure that all the actual data values were positive. The main characteristics of this approach are shown in FIG-8.17.

FIG-8.17

How a Sentinel-Based Array Operates



The complex part of this operation is locating the sentinel value within the array. After several values have been added there is no easy way of knowing the sentinel's position. To find its location we must search through the contents of the array.

Searching a list of values for some specific entry is one of the commonest requirements in a software application. There are many ways of searching a list, but for the moment we will content ourselves by examining only one of these; others will be explored in a later chapter.

If we are looking for the value -99 in a list of values, we can compare -99 with each value in the list and stop when we find a match. This can be achieved by the code

```
int post = 0;
while (list[post] != -99)
   post++;
```

Once the insert position has been found, we need to insert the new value and place -99 to its new position. This would be achieved using the lines

```
list[post] = value;
list[++post] = -99;
```

Other operations are easily handled. For example, we can determine if the list is empty using the expression

```
list[0] == -99
```

and if it's full using

```
list[SIZE] == -99
```

To count the number of entries in the list we are forced to search for the sentinel. Its position in the list will be equal to the number of entries. For example, initially -99 is in cell 0 and there are zero entries in the list; when -99 stored in cell 3 there will be three entries in the list (occupying cells 0, 1 and 2).

Activity 8.12

Create a new project, *ManageArray02*, based on *ManageArray01*. The new program should use a sentinel value rather than a count.

Problem: Inserting a Value into an Array

In the previous problem new values were inserted at the end of the existing data. However, there are many circumstances when the new value will be required to be positioned elsewhere within that data.

Solution:

When we want to add a value between existing values (just as we might add a character into a misspelled word when using a word processor) then we need to create space at the insertion point by moving other values out of the way.

FIG-8.18 shows the steps involved when adding a value into a count-based list.

FIG-8.18

Inserting a Value into an Array

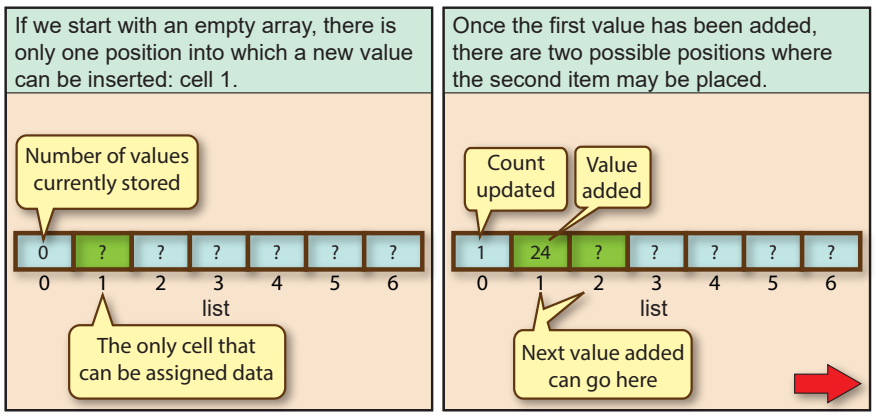
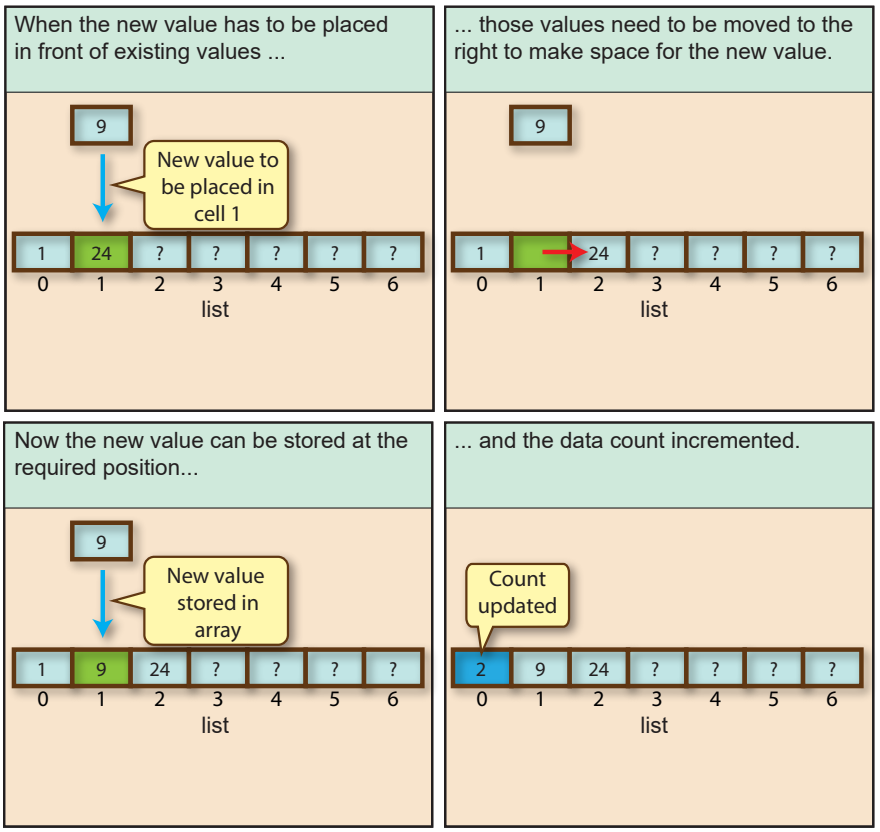


FIG-8.18
(continued)

Inserting a Value into an Array



Activity 8.13

An array is in the state shown in the diagram below.



In which cells may a new value be positioned?

Assuming the value 77 is to be placed in cell 3, show, with the aid of diagrams, the state of the array after

a) Existing values have been moved to make space for the new value.
b) The new value has been inserted.

When a new item is being added, we need to acquire not only its value, but also the cell number into which it is to be placed. This can be done using the following code:

```
cout << "Enter new value : ";
cin >> value;
cout << "Enter its position : ";
cin >> post;
while (post < 1 || post > list[0]+1)
{
   cout << "Invalid position. Must be in the range 1 to " <<
   ↳list[post]+1<< endl;
   cout << "Re-enter position : ";
   cin >> post;
}
```

Notice that the code includes a check to insure that the insert position is valid.

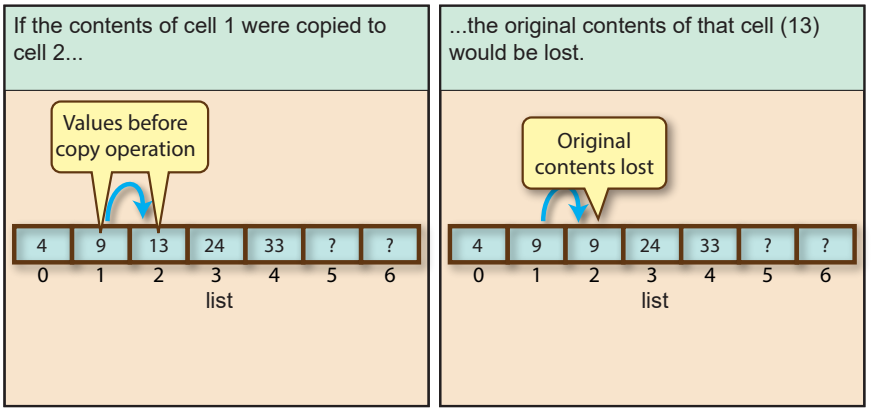
To free space for the new value we need to move all those values between *post* and *list[0]* up one position within the array. This is done using the following code

```
for(int current = list[0]; current >= post; current--)
   list[current+1] = list[current];
```

As we can see, it is necessary to move the value at the end of the list first otherwise we would overwrite the next value (see FIG-8.19).

FIG-8.19

Losing Data when Moving Values within an Array



All that remains now is to add the new value and increment the count held in *list[0]*:

```
list[post] = value;
list[0]++;
```

The program in FIG-8.20 is similar to that in FIG-8.14 but allows the user to select the position at which the new value is to be added.

FIG-8.20

Implementing Data Insertion

```
#include <iostream>
using namespace std;

int main()
{
   const int SIZE{5};
   int list[SIZE + 1] {0};
   int value;
   int option;
   do
   {
      //*** Display menu ***
      cout << "1 - Enter value\n";
      cout << "2 - Display number of values held\n";
      cout << "3 - Display all values held\n";
      cout << "4 - QUIT\n";

      //*** Get option ***
      cout << "Enter option required (1 - 4) : ";
      cin >> option;
      while (option < 1 || option > 4)
      {
         cout << "Invalid option. Re-enter : ";
         cin >> option;
      }
      cout << "\n\n";
```



FIG-8.20
(continued)

Implementing Data Insertion

```
      //*** Execute option ***
      int post = 0;
      switch (option)
      {
         case 1: //*** Add new value at specified position ***
            if (list[0] < SIZE)
            {
               //*** Get value and insert position ***
               cout << "Enter new value : ";
               cin >> value;
               cout << "Enter its position : ";
               cin >> post;
               while (post < 1 || post > list[0]+1)
               {
                  cout << "Invalid position. Must be in range 1
                  ↳to " << list[0]+1 << endl;
                  cout << "Re-enter position : ";
                  cin >> post;
               }
               //*** Free space for new value ***
               for(int current = list[0]; current >= post;
                  ↳current--)
               {
                  int next = current + 1;
                  list[next] = list[current];
               }
               //*** Add new value ***
               list[post] = value;
               //*** Increment count ***
               list[0]++;
            }
            else
               cout << "List is full\n\n\n";
            break;
         case 2: //*** Display number of entries ***
            cout << "The list contains "<<list[0]<<" entries\n\n";
            break;
         case 3: //*** Display entries ***
            if (list[0] == 0)
               cout << "The list is empty\n\n";
            else
            {
               cout << "Values held are\n";
               for(int c = 1; c <= list[0]; c++)
                  cout << list[c] << "  ";
               cout << "\n\n\n";
            }
            break;
         case 4: //*** Terminate program ***
            cout << "Program terminated\n";
      }
   }
   while (option != 4);
}
```

Activity 8.14

Start a new project called *ManageArrays03* then implement and test the program given above.

Problem: Deleting a Value

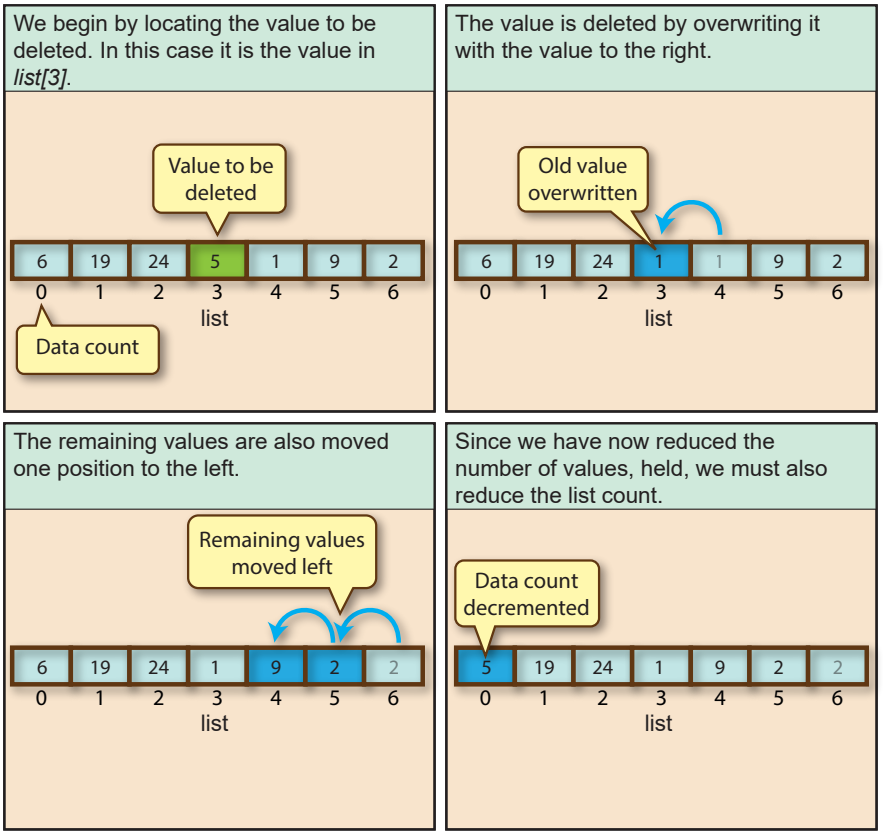
Some situations require an item of data to be deleted from an existing list. Sometimes we need to find and delete a specific value; other times way may want to delete the value at a specific position in the list irrespective of its value.

Solution:

To delete a value from a specific position in a list, we must eliminate the value by moving all those that come after it up one position in the list (see FIG-8.21).

FIG-8.21

How to Delete a Value from a Count-Based Array



Notice that *list[6]* retains a copy of the final value in the list, but this will have no effect on our code since, by reducing the count, the content of *list[6]* is no longer regarded as part of the valid data.

If we want to delete the value held at position *post*, then the logic required to move the other data items is:

```
//*** Delete entry by moving subsequent entries to left ***
for(int current = post+1; current <= list[0]; current++)
   list[current-1] = list[current];
```

When we accept a value for *post*, we must ensure that we are attempting to delete from a position that contains data, so the following code is required:

```
//*** Get position ***
cout << "Enter position of item to be deleted : ";
cin >> post;
while (post < 1 || post > list[0])
```

```
{
   cout <<"The position is invalid./nRe-enter : ";
   cin >> post;
}
```

Finally, we need to reduce the data count by 1:

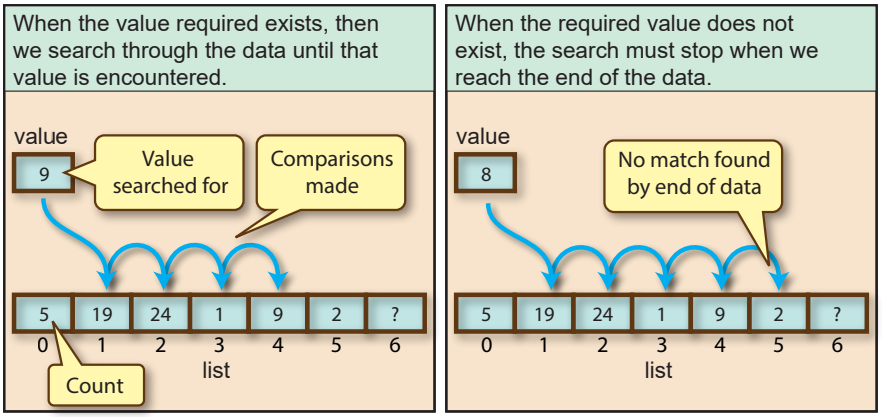
```
list[0]--;
```

When we want to delete a specific value from the list, we must first locate that value. This requires logic very like that required for locating the sentinel in a list.

However, whereas we knew the sentinel would always appear in a list of that type, we cannot make the same guarantees about user-selected values (see FIG-8.22).

FIG-8.22

How to Search a Count-Based Array



Our new logic must allow for both possibilities.

What we require can be described in structured English as

```
Get value to be deleted
Start at beginning of list
WHILE not arrived at end of data AND value to be deleted not found DO
    Move to next entry in the list
ENDWHILE
```

This translates into C++ as:

```
//*** Enter value to be deleted ***
cout << "Enter value to be deleted : ";
cin >> value;
//*** Search for value in list ***
post = 1;
while (post <= list[0] && list[post] != value)
   post++;
```

Once the `while` loop has been completed, we need to check if the match was found. If it was, the cell we stopped at will contain a value matching the required value and this can be checked with the code

```
if (list[post] == value)
```

Having found a match, the contents of the array can then be re-arranged to delete the specified entry and the count decremented.

Activity 8.15

Modify *ManageArray03* adding the ability to delete a value from the list (this should be option 4 with option 5 becoming **quit**). In this version of the program, the value to be deleted, not its position, should be specified. If the value to be deleted cannot be found, an appropriate message should be displayed.

Test Data: list 3, 6, 9, 12
Values to delete : 6, 12, 2

Display the list content after each deletion.

Problem: Checking that a Date is Valid

When presented with three `int` values, *day*, *month* and *year* which are meant to represent a date, then checking that the date is a valid one can be made easier by using an array.

Solution:

What we consider to be a valid date depends on whether we are willing to travel back to a time before the introduction of the Gregorian calendar. Assuming we don't want to go back that far, then the check on *year* being valid is simply

```
if (year > 1752)
```

The check on *month* only needs to ensure its value lies between 1 and 12:

```
if(month >= 1 && month <= 12)
```

It's only when we arrive at checking *day* that things become more complex. Obviously, the lower limit for its value is 1, but the upper limit depends on which month we are in.

To handle this we could set up an array giving the days in each month:

```
int daysinmonth[]{ 0,31,28,31,30,31,30,31,31,30,31,30,31 };
```

Notice that cell with index zero is not being used. This way, the array index number matches the corresponding *month* value.

Now we can check that *day* is valid using the line

```
if(day >= 1 && day <= daysinmonth[month])
```

But wait! What about leap years when February has 29 days?

When we have a leap year, we need to change the contents of *daysinmonth[2]* from 28 to 29.

A year is a leap year if it is exactly divisible by 4 but not 100 OR it is exactly divisible by 400. This condition can be coded as

```
(year%4 == 0 && year % 100 != 0) || (year % 400 == 0)
```

Now, remembering that C++ will automatically convert *true* to 1 and *false* to zero, we can set the days in February with the line

```
daysinmonth[2] = 28 + ((year%4 == 0 && year % 100 != 0) ||
↳(year % 400 == 0));
```

Note the extra set of parentheses around the Boolean expression to ensure these are calculated first.

Activity 8.16

Create a new project, *Validate,* which accepts three integer values displays "*Valid date*" if it is a valid date after 1752 and "*Invalid date*" if the three integers do not form a valid date.

HINT: It may be easier to check if *day*, *month* and *year* are invalid rather than valid when coding.

Character Arrays



Null-terminated strings stored in **char** arrays are referred to as **C-style strings**. This method of handling strings has generally been replaced by the use of **string** objects (see Chapter 21).

The rules for character arrays are somewhat different from those for other types of arrays. Although we can store a set of individual characters within a **char** array using a line such as

```
char letters[] {'a','b','c','d'};
```

the most widespread use of a **char** array is to hold a string value. Strings make use of a terminating null character ('\0') to mark the end of the sequence of characters held. Without the null character, the sequence of characters cannot be treated as a single entity. So, if we wanted the previous example to produce a properly formed string, we would need to include the null character at the end of the sequence:

```
char letters[] {'a','b','c','d','\0'};
```

A simpler option is also available allowing us to create the same effect, but with less typing. This is done using the line:

```
char letters[] {"abcd"};
```

Using this approach, C++ automatically spreads the letters out over the cells of the array and adds the null character.

Rather than display the contents of our **char** array using the standard approach:

```
post = 0
while (letters[post] != '\0')
   cout << letters[post++];
cout << endl;
```

we are allowed to treat a **char** array containing a terminating null character as if it were a simple variable, and use the line

```
cout << letters << endl;
```

to create exactly the same effect.

The same approach can be used when we want to input a value for a **char** array:

```
cin >> letters;
```

Again, C++ will automatically add the terminating null character to any string entered at the keyboard.

Because of the terminating null, we must always remember that the number of characters that can be stored in a **char** array is one less than the number of elements in the array. For example, the declaration

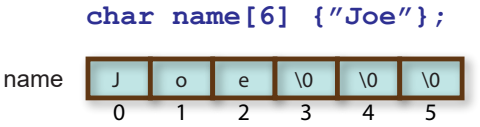
```
char word[11];
```

creates an 11 element array, capable of storing 10 characters with the terminating null character in the eleventh cell.

If we specify the size of a `char` array and store a smaller number of characters in that array, the additional cells will be set to null (see FIG-8.23).

FIG-8.23

Set Size *char* Array



Activity 8.17

Create a new project called *Strings*. In the program set up an array called *sentence* containing 41 cells. Code the program to read in a value for the array and then display the contents of the array.

Test the program using the following test data:

Run 1 : Gort
Run 2 : Klaatu barada nikto

We can see from the result of Run 2 in Activity 8.17, that only the first word of the value entered is displayed. The problem is caused by the `cin` statement. When entering a string, `cin` takes any whitespace character (space, tab, newline) to signify the end of a string. This means that the space between the first and second word is assumed to be the end of the entered data meaning that the last two words are not stored in *sentence*.

Using `cin` to input a string value has an even greater problem. There is no way of ensuring that the user does not enter a string which is too long for the space provided by the named array. Since C++ does not check that the memory space being accessed has actually been assigned to the named array, other variables can be corrupted by the over-long data that the user has entered and the program will terminate unexpectedly.

gets_s()

To solve the two problems encountered when using `cin`, we can use `gets_s()` function. This function's parameter is the name of the array in which the string is to be stored. So, we could read a string into the array *sentence* with the line:

```
gets_s(sentence);
```

This will accept all typed-in characters up to the *Enter* character.



We'll see another method of dealing with this problem when we examine the true nature of `cin` in Chapter 21.

Optionally, we can also include the maximum number of characters we will accept from the input devices:

```
gets_s(sentence,20);
```

Remember that an extra character position is required for the *null* character which `gets_s()` inserts automatically. However, the program will still abort if more than the specified number of characters are entered or, where the number of characters are omitted, when the number of characters entered are too many for the array to hold.

Activity 8.18

Modify the code in *Strings* so that the value is read in using `gets_s()`. Test the program with the two test data values given in the previous Activity.

Perhaps the best way to overcome the problems associated with the use of `cin` is to use `scanf_s()`. When reading a string using `scanf()` we can include in the formatting string, the maximum number of characters to be read. Since we know that the sentence array was set up with 41 elements, that allows a maximum of 40 characters to be read (with the null character in the 41st position), so our input statement would be

```
scanf_s("%40s",sentence);
```

All characters up to the *Enter* key are read and if too many characters are entered, these are ignored and only the specified number of characters stored (with the *null* being added automatically).



There more about array names and addresses in Chapter 10.

Before we move on, some of you may have noticed that the & symbol is missing from the front of the array name in the use of `scanf_s()`. For regular variables, we need the & to signify that we want the address of the variable but with arrays, the name of the array is synonymous with the address of the first cell assigned to the array and therefore no ampersand is required.

So, to summarise, we can input a string to a `char` array using `cin` or `scanf_s()` as well as `gets_s()` and we can output the contents of that `char` array using `cout` or `printf()`. However, we cannot perform normal assignment statements on the `char` array with a line such as

```
sentence = "Perhaps tomorrow";
```

nor can we test the contents of the `char` array with code such as

```
if (sentence == "Today")
```

Both these lines are invalid.

strcpy_s()

To copy a string into a `char` array, we need to use the `strcpy_s()` function which allows us to write a line such as

```
strcpy_s(sentence,41,"Perhaps tomorrow");
```

to assign the string *Perhaps tomorrow* to the `char` array *sentence*. The middle value, 41, represents the maximum number of characters to be copied and is usually set to the size of the receiving array (*sentence,* in this case).

strcmp()

To compare one string value with another we need to make use of another function, `strcmp()`. Using this, we can check if *sentence* contains the string *Today* with the line

```
if (strcmp(sentence,"Today") == 0)
```

The `strcmp()` function returns zero if the two strings are an exact match. It returns a positive number if the first string is greater than the second and a negative number if the second string is greater. We will look at string-related functions in more detail in a later chapter.

Accessing Individual Characters

We can access the individual elements of the `char` array in the traditional way. This gives us access to the individual characters within the string. So, each of the following are valid code snippets:

```
if (sentence[2] == 'a')
sentence[3] = 'x';
```

However, when assigning a value to a specific element of a `char` array, be careful not to overwrite the terminating null character.

Wide Character Arrays

When using an array of wide characters (`wchar_t`), the constant must be given in wide character format (that is, preceding the first double quote with the letter L):

```
wchar_t letters[] = L"abcd";
```

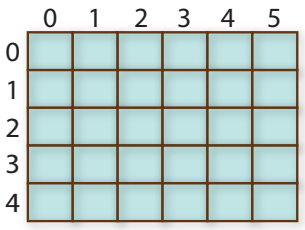
Inputting wide character strings is done using `wcin`, `scanf_s()` or `getws_s()`. Outputting wide character strings can be done using `wcout` or `printf()`.

Multi-dimensional Arrays

If a student takes six exams, we could store that information in a six element array, but if five students each take six exams, then although we might use a 30 element array, a better approach would be to use a two-dimensional array (see FIG-8.24). This allows one row for each student and one column for each of the exams.

FIG-8.24

Visualising a 2D Array



To declare a two-dimensional array we simply extend the format used when setting up one-dimensional arrays:

```
type array_name [no. of rows][no. of columns];
```

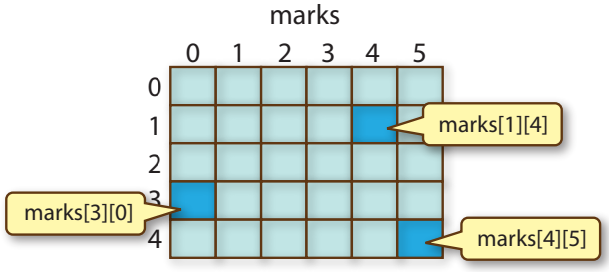
So, to declare the array shown in FIG-8.23 we would use the line

```
int marks[5][6];
```

To access an individual element within a two-dimensional array, we must specify the array name and the row and column numbers. The row and column values are each enclosed in their own set of brackets (see FIG-8.25).

FIG-8.25

Accessing Elements in a 2D Array



To initialise a two-dimensional array with a specific set of start up values we need to add an extra set of braces for each row with a comma separating each row. This gives us the following format

```
type array_name[rows][columns] =    {  {values for row 0},
                                       {values for row 1},
                                           ...
                                       {values for last row}
                                    }
```

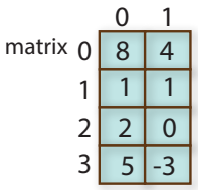
For example,

```
int matrix[4][2] { {8,4},{1,1},{2,0},{5,-3} };
```

This example would result in the array shown in FIG-8.26.

FIG-8.26

Initialising a 2D Array



Activity 8.19

Write the declaration required to set up the array shown below.

board

0	7	9	9	3
1	4	2	8	5
5	5	5	1	7

If we omit specific values in any row, the corresponding cells will be set to zero. For example,

```
int grid[3][5]{ {2,4,6},{12},{22,24,26,28,30} };
```

would produce the setup shown in FIG-8.27.

FIG-8.27

Omitting Values in the Initialisation

grid

2	4	6	0	0
12	0	0	0	0
22	24	26	28	30

If the values for the end row(s) are omitted from the initialisation altogether as in the line

```
int grid[3][5]{ {2,4,6},{12} };
```

that row will be filled with zeros. In the line given above, the third row cells would be set to zero.

With one-dimensional arrays we could have the compiler deduce the number of cells in the array from the number of values given in the initialisation list. For example, in the line

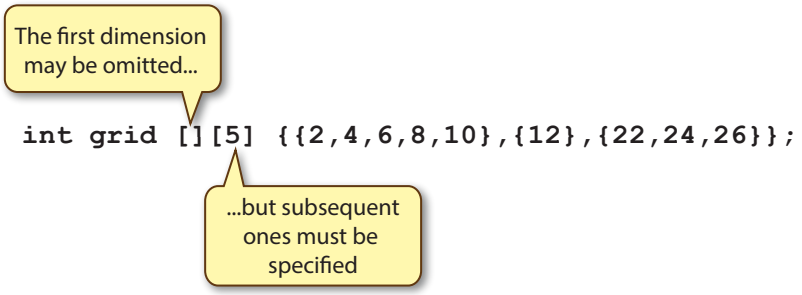
```
int num[] {2,4,6};
```

the compiler would automatically create *num* with three cells. This same approach is only partially available for multi-dimensional arrays. For while it is acceptable to

omit the first dimension, others must be explicitly stated (see FIG-8.28).

FIG-8.28

Omitting Dimension Details



The compiler will automatically calculate the size of the array's first dimension (that is, the number of rows in the array).

We saw earlier how we could use a `for` loop to access each element of an array in turn. That same technique can be used to access a two dimensional array. The only difference this time is that we need to employ two `for` loops.

Returning to our *marks* array, we could read in the marks for the first student (whose data is held in row 0) using the code:

```
for (int exam {0}; exam < 6; exam++)
{
   cout << "Enter code for exam " << exam << " : ";
   cin >> marks[0][exam];
}
```

Note that the row value is fixed at zero:

```
marks [0] [exam];
```

To deal with the next student this will have to be set to 1 while *exam* cycles again through the values 0 to 5.

To deal with all the students, one after another, we just need to add an outer `for` loop and set the first subscript to the new loop's counter:

```
for ( int student{0}; student < 5; student++)
   for (int exam {0}; exam < 6; exam++)
   {
      cout << "Enter code for exam " << exam << " : ";
      cin >> marks[student][exam];
   }
```

Activity 8.20

Create a new project called *Exams01* which sets up an array, *marks*, to store the scores for 5 students each of whom sit 6 exams.

The program should assign random marks in the range 0 to 100 for all students and all exams then display the values held (one output line for each student).



Use the term `rand() % 101` to generate an integer value in the range 0 to 100.

The Range-Based *for* Loop and 2D Arrays

We saw earlier in this chapter how the range-based `for` loop can be used to traverse every value in a one-dimensional array. Although it cannot be used directly in the same way to access every cell in a two-dimensional array, it can also be used to traverse a single row of that array.

Our *marks* array was defined to have 5 rows by 6 columns. To display every value in the first row of that array we could use the lines

```
for (int v; marks[0])
   cout << v << endl;
```

Notice that we have to specify the array name and the row being accessed (`marks[0]`) rather than just giving the array name as we did with one-dimensional arrays.

The only change needed to traverse the second row, would be replacing the term `marks[0]` with `marks[1]`.

To traverse all the rows in the array (each row represents one student), we need to employ a traditional `for` loop to specify the row number:

```
for (int student{0}; student < 6; student++)
```

followed by the range-based `for` loop which uses the value of *student* to determine which row is currently being accessed:

```
for (int student{0}; student < 6; student++)
{
   for (int v; marks[student])
      cout << v << ' ';
   cout << endl;
}
```

Activity 8.21

Modify *Exams01* to make use of a range-based `for` loop when displaying the contents of *marks*.

Although we can use the range-based `for` loop to traverse each row, it is not possible to use it to traverse each column. For that we have to stick with the traditional `for` loop (for the moment).

Problem: Displaying each Student's Highest Score

We have already explored how to display the lowest and highest values in a one-dimensional array. Now we need to discover how to achieve this for each row in a two-dimensional array.

Solution:

If we start by examining the code for finding the largest value in the first row of the marks array, then we have the code:

```
highest = marks[0][0];
for (int v; marks[0])
   if( v > highest)
      highest = v;
cout << "Highest score for student was " << highest << endl;
```

To perform the same logic on each student we need the same change to the code that we had in the earlier example above: change the *row* value from [0] to a dynamic value:

```
for(int column {0}; column < 5; column++)
{
   highest = marks[column][0];
```

```
    for (int v; marks[column])
       if( v > highest)
          highest = v;
    cout << "Highest score for student was " << highest << endl;
}
```

Rather than access a two-dimensional array a row at a time we are, of course, able to access a column at a time but we will have to revert to the standard `for` statement for the inner loop rather than the range-based `for` which cannot be used to access a column of values. The code below displays the *marks* array a column at a time.

```
for (int column {0}; column < 6; column++)
{
   for (int row {0}; row < 5; row++)
      cout << marks[row][column] << ' ';
   cout << endl;
}
```

Activity 8.22

Using an array declared as `int matrix[4][4]`, create a project called *Arrays2D* which uses nested `for` loops to assign the following values to the cells of the array:

a)

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

b)

	0	1	2	3
0	1	5	9	13
1	2	6	10	14
2	3	7	11	15
3	4	8	12	16

c)

	0	1	2	3
0	16	15	14	13
1	12	11	10	9
2	8	7	6	5
3	4	3	2	1

Two-Dimensional *char* Arrays

Since we need a `char` array to store a single string, we are going to need a two-dimensional `char` array if we want to store several strings. For example, if we want to store the days of the week, we could declare our array as

```
char daynames[7][10]= {"Sunday","Monday","Tuesday","Wednesday",
↳"Thursday","Friday","Saturday"};
```

But, as with numeric arrays, we also have the option to omit the first dimension in the array declaration and allow the compiler to determine its value:

```
char daynames[][10]= {"Sunday","Monday","Tuesday","Wednesday",
↳"Thursday","Friday","Saturday"};
```

To display the word *Sunday*, we would use the code

```
cout << daynames[0];
```

Activity 8.23

Modify your *DayOfTheWeek* program (last modified in Activity 6.21) to make use of a two dimensional `char` array for the day names. Test your program with a date for each day of the week.

We can use a range-based `for` loop to access each string in our 2D array. For example, the code

```
for (auto v : daynames)
   cout << v << endl;
```

would display the following output:

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

And, because each value of *v* is really an array of characters, we can even use a subscript to access individual characters within the array. For example,

```
for (auto v : daynames)
   cout << v[0] << endl;
```

would display

```
S
M
T
W
T
F
S
```

3-Dimensional Arrays and Higher

There are situations where we may need an array with even more dimensions. For example, if our students were to sit the exams three times every year, then we would need an array which had three dimensions (5 students, 6 exams, 3 semesters). We would define such an array with the statement:

```
int marks[5][6][3];
```

Should we wish to initialise a three-dimensional array we need braces for each dimension as we can see in the example below:

```
int nos[3][2][4]
{
   {
      {1,3,2,3},{1,1,1,1}
   },
   {
      {1,8,9,7},{6,4,5,3}
   },
   {
      {0,0,0,0},{6,6,1,2}
   }
};
```

Again, the option exists to omit details of the first dimension, leaving it to the compiler to perform the calculation. Using this option, our array declaration would begin

```
int nos[][2][4]{...
```

Accessing each cell in such an array requires three `for` loops. For example, to display the contents of the *nos* array defined above, we would use the code:

```
for (int x{0}; x < 3; x++)
   for (int y{0}; y < 2; y++)
      for(int z{0}; z < 4; z++)
         cout << nos[x][y][z] << endl;
```

The range-based `for` loop can be used to traverse the final dimension only, so for our *nos* array, we could rewrite the above code as

```
for (int x{0}; x < 3; x++)
   for (int y{0}; y < 2; y++)
      for (int v; nos[x][y])
         cout << v << endl;
```

An array can have an almost unlimited number of dimensions although it is unusual to require more than three.

Activity 8.24

Create a project, *Exams02,* which will randomly generate marks (0 to 100) for five students taking exams in math, English, history, geography, physics, and chemistry. An exam in each subject is sat during each of the college year's three semesters (i.e. 5 students, 6 topics, 3 exams). The results should be stored in a three-dimensional array.

The program should display all the marks held with each semester's results being displayed separately.

The program should then display the average mark each student achieved for each exam over the three semesters.

Constant Arrays

Just as we may create a single value constant with a statement such as

```
const double PI = 3.14159265;
```

so we can create a constant array with the value within its elements permanently fixed with statements such as

```
const int DAYSINMONTH[] {31,28,31,30,31,30,31,31,30,31,30,31};
```

or

```
const char WEEKDAYS[7][10] {"Sunday","Monday","Tuesday",
↳"Wednesday","Thursday","Friday","Saturday"};
```

Summary

- An array is a named series of cells or elements.
- Each cell within an array is capable of storing a single value.
- Each cell is sequentially numbered - the first cell is cell zero.
- Array cells are identified within C++ code by the array name followed by the cell's subscript enclosed in brackets.
- The type of value held within an array and the number of cells it contains is stated in the array declaration.
- The content of a cell is initially undefined unless the array is initialised when being declared.
- An array can be initialised by listing the sequence of values to be assigned to the array elements when the array is declared. These initial values are comma-separated and enclosed in braces.

- If too many values are given in the array initialisation list, compilation will fail.
- If too few values are given in an array initialisation list, the cells which have been assigned no explicit value are set to zero (numeric arrays only).
- An array subscript can be defined using an integer constant, variable, or expression.
- If a subscript is out-of-range the program may continue to execute without reporting the error.
- The range-based `for` loop can be used to traverse the values held in an array.
- When using a range-based `for` loop the contents of the array can be explicitly specified within the `for` line without assigning those values to a named array.
- When only part of an array is occupied by meaningful data, the occupied area can be determined using either a count or a sentinel value.
- When a a multi-dimensional array is assigned values during declaration, the first dimension's size (but not its brackets) can be omitted and the compiler will calculate its size.
- The range-based `for` loop can only be used to traverse the final dimension of a multi-dimensional array.
- Use a `char` array to store a string.
- Initialisation of a `char` array can use a string constant.
- Strings held in `char` arrays are automatically terminated by a null character.
- Input and output of a `char` array can be performed as if the array is a single variable.
- Use a 2D `char` array to store several strings.
- We can use a range-based `for` loop to access each string in a 2D char array.
- Constant arrays can be defined, fixing the value in every cell of the array.

Solutions

Activity 8.1

Code for *Arrays01*:

```
#include <iostream>
using namespace std;

int main()
{
  int numbers[5];

  // *** Read in numbers ***
  for (int c = 0; c < 5; c++)
  {
    cout << "Enter number : ";
    cin >> numbers[c];
  }
  // *** Display numbers entered ***
  cout<<"Numbers entered were:\n";
  for(int c = 0; c < 5; c++)
    cout<<numbers[c]<<" ";
  cout << endl;
}
```

Activity 8.2

Modified code for *Arrays01*:

```
#include <iostream>
using namespace std;

int main()
{
  const int SIZE{ 7 };

  int numbers[SIZE];

  // *** Read in numbers ***
  for (int c = 0; c < SIZE; c++)
  {
    cout << "Enter number : ";
    cin >> numbers[c];
  }
  // *** Display numbers entered ***
  cout << "Numbers entered were : \n";
  for (int c = 0; c < SIZE; c++)
    cout << numbers[c] << " ";
  cout << endl;
}
```

Activity 8.3

Code for *Arrays02*:

```
#include <iostream>
using namespace std;

int main()
{
  int list[]{ 7,5,2,9,1,1,3,6,5,8 };
  for (int c = 0; c < sizeof(list) /
  ↳sizeof(list[0]); c++)
    cout << list[c] << ' ';
  cout << endl;
}
```

Activity 8.4

Modified code for *Arrays02*:

```
#include <iostream>
using namespace std;

int main()
{
  int list[]{ 7,5,2,9,1,1,3,6,5,8 };
  for (int c = 0; c < size(list); c++)
    cout << list[c] << ' ';
  cout << endl;
}
```

Activity 8.5

No solution required.

Activity 8.6

Code	*numbers* contents
`int numbers [8] = {0};`	0,0,0,0,0,0,0,0
`for (int p=0;p<=7;p++)` `  numbers[p] = p*2;`	0,2,4,6,8,10,12,14
`numbers[numbers[2]-1]=23;`	0,2,4,23,8,10,12,14

Activity 8.7

Modified code for *Arrays02 (range-based for)*:

```
#include <iostream>
using namespace std;

int main()
{
  int list[]{ 7,5,2,9,1,1,3,6,5,8 };
  for (int v: list)
    cout << v << ' ';
  cout << endl;
}
```

Activity 8.8

Modified code for *DieCount*:

```
#include <iostream>
#include <ctime>
using namespace std;

//*** Count occurrence of each value on 10,000 die
↳throws ***
int main()
{
  const int FACES{ 10 };
  int counts[FACES+1]{ 0 };
  int dicethrow;

  //*** Seed random number generator ***
  srand(static_cast<unsigned>(time(0)));

  //*** Throw the dice 10000 times ***
  for (int c = 1; c <= 10000; c++)
  {
    //*** Throw die ***
    dicethrow = rand() % FACES + 1;
    //*** Add to appropriate count ***
    counts[dicethrow]++;
  }

  //*** Display each count ***
  for (int c = 1; c <= FACES; c++)
    cout << c << " occurred " << counts[c] <<
    ↳" times" << endl;
}
```

Activity 8.9

Modified code for *Lottery* (all numbers generated in `for` loop):

```
#include <iostream>
#include <ctime>
using namespace std;

int main()
{
  const int HIGHEST{ 49 };  //Highest lottery number
  int lottery[HIGHEST + 1]{ 0 }; //Values already
selected
  int number;        //Currently generated number

  //*** Seed generator ***
  srand(static_cast<unsigned>(time(0)));

  //*** FOR 6 times DO ***
  for (int c = 1; c <= 6; c++)
  {
    //*** Generate an unselected number ***
    do
```

```
      //** Generate number **
      number = rand() % HIGHEST + 1;
    while (lottery[number] == 1);
    //*** Set corresponding cell ***
    lottery[number] = 1;
    //*** Display value ***
    cout << number << ' ';
  }
  cout << endl;
}
```

Modified code for *Lottery* (values displayed in ascending order):

```
#include <iostream>
#include <ctime>
using namespace std;

int main()
{
  const int HIGHEST = 49;
  int lottery[HIGHEST+1] = {0};
  int number;

  //*** Seed generator ***
  srand(static_cast<unsigned>(time(0)));

  //*** FOR 6 times DO ***
  for(int c = 1; c <= 6; c++)
  {
    //*** Generate an unselected number ***
    do
      //** Generate number **
      number = rand() % HIGHEST + 1;
    while (lottery[number] == 1);
    //*** Set corresponding cell ***
    lottery[number] = 1;
  }
  // *** Display values in ascending order ***
  // ** Check each element ***
  for(int c=1; c<= 49; c++)
    //** IF set to 1, display subscript value **
    if (lottery[c] == 1)
      cout << c << ' ';
  cout << endl;
}
```

Activity 8.10

Modified code for *Shuffle* (contents displayed before and after shuffle):

```
#include <iostream>
#include <ctime>
using namespace std;


int main()
{
  int list[20];
  int sub1, sub2;
  int temp;

  //*** Seed generator ***
  srand(static_cast<unsigned>(time(0)));

  //*** Set up values in array ***
  for( int c = 0; c <= 19; c++)
    list[c] = c;
  //*** Display initial order ***
  cout << "Initial order : \n";
  for(int c = 0; c <= 19; c++)
    cout << list[c] << ' ';
  cout << "\n\n";

  //*** Shuffle ***
  for(int c = 1; c <= 200; c++)
  {
    //*** Generate two subscript values ***
    sub1 = rand()%20;
    sub2 = rand()%20;
    //*** Swap values at these positions ***
    temp = list[sub1];
    list[sub1] = list[sub2];
    list[sub2] = temp;
  }
  //*** Display shuffled items ***
  cout << "Order after shuffle :\n";
  for(int c = 0; c <= 19; c++)
    cout << list[c] << ' ';
  cout << endl;
}
```

Modified code for *Shuffle* (52 values):

```
#include <iostream>
#include <ctime>
using namespace std;

int main()
{
  int list[52];
  int sub1, sub2;
  int temp;

  //*** Seed generator ***
  srand(static_cast<unsigned>(time(0)));

  //*** Set up values in array ***
  for( int c = 0; c <= 51; c++)
    list[c] = c;
  //*** Display initial order ***
  cout << "Initial order : \n";
  for(int c = 0; c <= 51; c++)
    cout << list[c] << ' ';
  cout << "\n\n";

  //*** Shuffle ***
  for(int c = 1; c <= 2000; c++)
  {
    //*** Generate two subscript values ***
    sub1 = rand()%52;
    sub2 = rand()%52;
    //*** Swap values at these positions ***
    temp = list[sub1];
    list[sub1] = list[sub2];
    list[sub2] = temp;
  }
  //*** Display shuffled items ***
  cout << "Order after shuffle :\n";
  for(int c = 0; c <= 51; c++)
    cout << list[c] << ' ';
  cout << endl;
}
```

Notice how many lines need to be modified in order to change from 20 to 52 elements in the array. Even the number of iterations of the shuffle loop has to be increased because of the increase in values to be rearranged.

It would have been much better to use a named constant for the size of the array. Then, if the number of shuffle iterations was a function of the size of the array, only one line of code would have been modified to change from 20 to 52 elements.

Modified code for *Shuffle* (displaying the suit):

```
#include <iostream>
#include <iomanip>
#include <ctime>
using namespace std;


int main()
{
  int list[52];
  int sub1, sub2;
  int temp;

  //*** Seed generator ***
  srand(static_cast<unsigned>(time(0)));

  //*** Set up values in array ***
  for( int c = 0; c <= 51; c++)
    list[c] = c;

  //*** Shuffle ***
  for(int c = 1; c <= 2000; c++)
  {
    //*** Generate two subscript values ***
    sub1 = rand()%52;
    sub2 = rand()%52;
    //*** Swap values at these positions ***
    temp = list[sub1];
    list[sub1] = list[sub2];
    list[sub2] = temp;
  }
```

```
//*** Display first six cards ***
cout << "First six cards :\n";
int cardvalue; //1 - Ace ... 13 - King
int cardsuit; //0 = hearts, 1 = diamonds, 2=
↳spades 3 = clubs
for(int c = 0; c <= 5; c++)
{
  cardvalue = list[c] % 13 + 1;
  cardsuit = list[c] / 13;
  cout << setw(2) << cardvalue << " of ";
  switch (cardsuit)
  {
    case 0:
      cout << "Hearts\n";
      break;
    case 1:
      cout << "Diamonds\n";
      break;
    case 2:
      cout << "Spades\n";
      break;
    case 3:
      cout << "Clubs\n";
      break;
  }
}
cout << endl;
}
```

Modified code for *Shuffle* (displaying the value and suit):

```
#include <iostream>
#include <iomanip>
#include <ctime>
using namespace std;


int main()
{
  int list[52];
  int sub1, sub2;
  int temp;

  //*** Seed generator ***
  srand(static_cast<unsigned>(time(0)));

  //*** Set up values in array ***
  for( int c = 0; c <= 51; c++)
    list[c] = c;

  //*** Shuffle ***
  for(int c = 1; c <= 2000; c++)
  {
    //*** Generate two subscript values ***
    sub1 = rand()%52;
    sub2 = rand()%52;
    //*** Swap values at these positions ***
    temp = list[sub1];
    list[sub1] = list[sub2];
    list[sub2] = temp;
  }
  //*** Display first six cards ***
  cout << "First six cards :\n";
  int cardvalue; //1 - Ace ... 13 - King
  int cardsuit; //0 = hearts, 1 = diamonds, 2=
  ↳spades 3 = clubs
  for(int c = 0; c <= 5; c++)
  {
    cardvalue = list[c] % 13 + 1;
    cardsuit = list[c] / 13;
    switch (cardvalue)
    {
      case 1:
        cout << "  Ace";
        break;
      case 11:
        cout << " Jack";
        break;
      case 12:
        cout << "Queen";
        break;
      case 13:
        cout << " King";
        break;
      default:
        cout << setw(5) << cardvalue;
        break;
    }
    cout << " of ";
    switch (cardsuit)
    {
      case 0:
        cout << "Hearts\n";
        break;
      case 1:
        cout << "Diamonds\n";
        break;
      case 2:
        cout << "Spades\n";
        break;
      case 3:
        cout << "Clubs\n";
        break;
    }
  }
  cout << endl;
}
```

Activity 8.11

When you attempt to add another value when the array is full, the message *List is full* is displayed.

Activity 8.12

Code for *ManageArray02* (differences from *ManageArray01* are highlighted):

```
#include <iostream>
using namespace std;

int main()
{
  const int SIZE = 5;  //*** Subscript of last cell
                       ↳in array
  int list[SIZE + 1] = {-99};//*** Array holding
                       ↳values entered
  int value;    //*** Value to be added to array
  int option;   //*** Menu option selected
  int post = 0  //***Position accessed in array

  //*** Repeat until quit selected ***
  do
  {
    //*** Display menu ***
    cout << "1 - Enter value\n";
    cout << "2 - Display number of values held\n";
    cout << "3 - Display all values held\n";
    cout << "4 - QUIT\n";

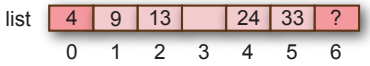
    //*** Get option ***
    cout << "Enter option required (1 - 4) : ";
    cin >> option;
    while (option < 1 || option > 4)
    {
      cout << "Invalid option. Re-enter : ";
      cin >> option;
    }
    cout << "\n\n";
    //*** Execute option ***
    switch (option)
    {
      case 1://*** Add a new value to the list ***
        cout << "Enter value to be added : ";
        cin >> value;
        // ** IF not full, add value and new
        ↳sentinel **
        if (list[SIZE] != -99)
        {
          //** Find sentinel **
          post = 0;
          while (list[post] != -99)
            post++;
          //** Insert value and new sentinel ***
          list[post] = value;
          list[++post] = -99;
        }
        else
          cout << "List is full\n\n\n";
        break;
      case 2://*** Display number of items in the
               ↳list ***
        //** Find sentinel **
        post = 0;
        while (list[post] != -99)
          post++;
        cout << "List contains "<<post<<" entries
        ↳\n\n\n";
```

```
        break;
      case 3://*** Display the contents of the list
             ↳***
        if (list[0] == -99)
          cout << "The list is empty\n\n";
        else
        {
          cout << "Values held are\n";
          post = 0;
          while (list[post] != -99)
            cout << list[post++] << "  ";
          cout << "\n\n\n";
        }
        break;
      case 4://*** Terminate the program ***
        cout << "Program terminated\n";
    }
  }
  while (option != 4);
}
```

Activity 8.13

A new value can be added at positions 1 to 5.

If 77 is to be added at position 3, we need to start by moving the contents of cells 3 and 4 one place to the right.



This makes room for the new value, 77, to be inserted.

And, because we have added a new value to the list, the count held in cell zero must be updated.

Activity 8.14

No solution required.

Activity 8.15

Modified code for *ManageArray03*:

```
#include <iostream>
using namespace std;

int main()
{
  const int SIZE = 5;//Size of data area in list
  int list[SIZE + 1] = {0}; //List of values
  int value;          //Value to be added to list
  int deletevalue;    //Value to be deleted from list
  int option;         //Option selected from menu

  do
  {
    //*** Display menu ***
    cout << "1 - Enter value\n";
    cout << "2 - Display number of values held\n";
    cout << "3 - Display all values held\n";
    cout << "4 - Delete a value\n";
    cout << "5 - QUIT\n";

    //*** Get option ***
    cout << "Enter option required (1 - 5) : ";
    cin >> option;
    while (option < 1 || option > 5)
    {
      cout << "Invalid option. Re-enter : ";
      cin >> option;
    }
    cout << "\n\n";

    //*** Execute option ***
    int post = 0;
    switch (option)
    {
      case 1: //*** Add new value at specified
              ↳position ***
        if (list[0] < SIZE)
        {
          //*** Get value and insert position ***
          cout << "Enter new value : ";
          cin >> value;
          cout << "Enter its position : ";
          cin >> post;
          while (post < 1 || post > list[0]+1)
          {
            cout << "Invalid position. Must be in
            ↳range 1 to " << list[0]+1 << endl;
            cout << "Re-enter position : ";
            cin >> post;
          }
          //*** Free space for new value ***
          for(int current = list[0]; current >=
          ↳post; current--)
          {
            int next = current + 1;
            list[next] = list[current];
          }
          //*** Add new value ***
          list[post] = value;
          //*** Increment count ***
          list[0]++;
        }
        else
          cout << "List is full\n\n\n";
        break;
      case 2: //*** Display number of entries ***
        cout << "The list contains "<<list[0]<<
        ↳" entries\n\n\n";
        break;
      case 3: //*** Display entries ***
        if (list[0] == 0)
          cout << "The list is empty\n\n\n";
        else
        {
          cout << "Values held are\n";
          for(int c = 1; c <= list[0]; c++)
            cout << list[c] << "  ";
          cout << "\n\n\n";
        }
        break;
      case 4:
        cout << "Enter value to be deleted : ";
        cin >> deletevalue;
        //*** Search for value in list ***
        post = 1;
        while (post <= list[0] && list[post] !=
        ↳deletevalue)
          post++;
        //*** IF value to be deleted found... ***
        if (list[post] == deletevalue)
        {
          //*** ...move subsequent values to left
          ↳***
          for(int current = post+1; current <=
          ↳list[0]; current++)
            list[current-1] = list[current];
          //*** Decrement count ***
          list[0]--;
          //*** Display success message ***
          cout << deletevalue << " deleted from
          ↳list\n\n\n";
        }
        else
          //*** ELSE display fail message ***
          cout << "Value to be deleted not found
          ↳\n\n\n";
        break;
      case 5: //*** Terminate program ***
        cout << "Program terminated\n";
    }
  }
  while (option != 5);
}
```

Activity 8.16

Code for *Validate*:

```
#include <iostream>
using namespace std;

int main()
{
  int day, month, year;
  //*** Get date ***
  cout << "Enter date in the form dd/mm/yyyy :";
  scanf("%d/%d/%d", &day, &month, &year);

  bool isvalid{ true };//Date valid indicator
  //*** If year invalid, set isvalid false ***
  if (year <= 1752)
```

```
    isvalid = false;
  //*** If month invalid, set isvalid false ***
  if (month < 1 || month > 12)
    isvalid = false;
  //*** Determine days in each month for year ***
  int daysinmonth[] {0,31,28,31,30,31,30,31,31,30,31
  ↳,30,31 };
  daysinmonth[2] = 28 + ((year % 4 == 0 &&
  ↳year % 100 != 0) || (year % 400 == 0));
  //***  If month invalid, set isvalid false ***
  if (day < 1 || day > daysinmonth[month])
    isvalid = false;
  //*** Display date and validity ***
  cout << day << '/' << month << '/' << year <<
  ↳" : ";
  if (isvalid)
    cout << "Valid date\n";
  else
    cout << "Invalid date\n";
}
```

It would be more efficient to nest the first three `if` statements, only executing the next `if` the previous one did not set *isvalid* to *false*. For example, this would mean that the code would be structured as:

```
if (year <= 1752)
  isvalid = false;
else if (month <1 | | month > 12)
  isvalid = false;
else
  ...
```

But since this adds to the complexity of the code without many benefits, we'll stick with the simpler version.

Activity 8.17

Code for *Strings*:

```
#include <iostream>
using namespace std;

int main()
{
  char sentence[41];

  //*** Read in string ***
  cout << "Enter text:";
  cin >> sentence;

  //*** Display value read ***
  cout << "Value entered was :"<<sentence<<endl;
}
```

The first run correctly reports the text entered as *Gort*.

In the second run, the value displayed is *Klattu*, the second and third words are missing.

Activity 8.18

Modified code for *Strings*:

```
#include <iostream>
using namespace std;

int main()
{
  char sentence[41];

  //*** Read in string ***
  cout << "Enter text:";
  gets(sentence);

  //*** Display value read ***
  cout << "Value entered was :"<<sentence<<endl;
}
```

This time both entered values are displayed correctly.

Activity 8.19

The declaration required is:

```
int board[][5] =
{
  {0,7,7,9,3},
  {1,4,2,8,5},
  {5,5,5,1,7}
};
```

Activity 8.20

Code for *Exams01*:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  int marks[5][6]; //5 students, 6 exams

  //*** Get marks for each student ***

  //** FOR each student DO **
  for(int student = 0; student <= 4; student++)
    //FOR each exam DO ***
    for(int exam = 0; exam <= 5; exam++)
      //*** Generate mark ***
      marks[student][exam] = rand()%101;
  //*** Display all scores ***
  for(int student = 0; student <= 4; student++)
  {
    cout<<"Scores for student "<<student<<" : ";
    for(int exam = 0; exam <= 5; exam++)
      cout <<setw(4)<<marks[student][exam]<<"  ";
    cout << endl;
  }
}
```

Activity 8.21

Modified code for *Exams01* (range-based for):

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  int marks[5][6]; //5 students, 6 exams

  //*** Read in scores for each student ***

  //** FOR each student DO **
  for (int student = 0; student <= 4; student++)
    //FOR each exam DO ***
    for (int exam = 0; exam <= 5; exam++)
    {
      //*** Generate mark ***
      marks[student][exam] = rand()%101;
    }
  //*** Display all scores ***
  for (int student = 0; student <= 4; student++)
  {
    cout << "Scores for student "<<student<<" : ";
    for (int v: marks[student])
      cout << v << ' ';
    cout << endl;
  }
}
```

Activity 8.22

Code for *Arrays2D*:

a)

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  int matrix[4][4];

  //*** Set up required values ***
  for (int row = 0; row <= 3; row++)
    for(int col = 0; col <= 3; col++)
      matrix[row][col] = row*4+col+1;

  //*** Display contents of array ***
  for (int row = 0; row <= 3; row++)
```

```
    {
      for(int col = 0; col <= 3; col++)
        cout << setw(3) << matrix[row][col];
      cout << endl;
    }
}
```

b)

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  int matrix[4][4];

  //*** Set up required values ***
  for (int row = 0; row <= 3; row++)
    for(int col = 0; col <= 3; col++)
      matrix[col][row] = row*4+col+1;

  //*** Display contents of array ***
  for (int row = 0; row <= 3; row++)
  {
    for(int col = 0; col <= 3; col++)
      cout << setw(3) << matrix[row][col];
    cout << endl;
  }
}
```

c)

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  int matrix[4][4];

  //*** Set up required values ***
  for (int row = 0; row <= 3; row++)
    for(int col = 0; col <= 3; col++)
      matrix[row][col] = 16 - (row*4+col);

  //*** Display contents of array ***
  for (int row = 0; row <= 3; row++)
  {
    for(int col = 0; col <= 3; col++)
      cout << setw(3) << matrix[row][col];
    cout << endl;
  }
}
```

Activity 8.23

Modified code for *DayOfWeek*:

```
#include <iostream>
using namespace std;

int main()
{
  int day, month, year;   // date entered
  int M,C,Y, modifiedyear; // calculated values
                          ↳required by formula
  int dayofweek;        // result required


  //*** Get date ***
  cout << "Enter date in the format dd/mm/yyyyy : ";
  scanf("%d/%d/%d",&day,&month,&year);

  //*** Calculate M ***
  M = (month + 9) % 12 + 1;

  //*** Calculate modified year ***
  modifiedyear = year - M / 11;

  //*** Calculate C ***
  C = modifiedyear / 100;

  //*** Calculate Y ***
  Y = modifiedyear % 100;

  //*** Calculate day of week ***
  dayofweek = ((static_cast<int>(2.6 * M - 0.2) +
  ↳day + Y + Y/4 + C/4 - 2*C)%7+7)%7;

  //*** Display result ***
  char daynames[][10] =
  ↳{"Sunday","Monday","Tuesday","Wednesday",
  ↳"Thursday","Friday","Saturday"};
  if (dayofweek>0 && dayofweek <= 6)
    cout << daynames[dayofweek]<<endl;
  else
  {
    cout << "Invalid value in dayofweek.\n";
    cout << " Value is "<<dayofweek<<endl;
  }
}
```

Notice that the array containing the set of day names is not defined within the program until immediately before it is defined. To some extent it is a matter of taste where a values are declared, but waiting until they are required means that they cannot accidentally be used earlier in your code.

Notice also that the possibility of the variable *dayofweek* being out of range is still included.

Activity 8.24

Code for *Exams02*:

```
#include <iostream>
#include <ctime>
#include <iomanip>
using namespace std;

int main()
{
  int marks[5][6][3];     //Contains exam marks

  //*** Generate random seed ***
  srand(static_cast<unsigned>(time(0)));
  //*** Generate exam marks ***
  for (int student = 0; student <= 4 ; student++)
    for(int topic = 0; topic <= 5 ; topic++)
      for(int semester = 0; semester <= 2;
      ↳semester++)
        //*** Generate mark ***
        marks[student][topic][semester] =
        ↳rand()%101;
  //*** Display results ***
  //*** FOR each semester DO ***
  for (int semester = 0; semester <= 2; semester++)
  {
    //*** Display heading ***
    cout << "Semester " << semester+1 << endl;
    cout <<"   M   E   H   G   P   C\n";
    //*** FOR each student DO ***
    for(int student = 0; student <= 4; student ++)
    {
      //*** FOR each topic DO
      for(int topic = 0; topic <= 5; topic++)
        //*** Display topic mark ***
        cout <<setw(4)<< marks[student][topic]
        ↳[semester];
      //*** New line after each student ***
      cout << endl;
    }
  }
}
```



Other Data Structures

In this Chapter:

- ❑ Defining and Using Record Structures
- ❑ Arrays of Records
- ❑ Nested Records
- ❑ Enumerated Types
- ❑ Bit Fields
- ❑ Bitwise Operators
- ❑ Union

Composite Data Structures

Introduction

Although an array can store multiple items of data, that data must all be values of the same type. Where we want to collect together data of varying types we can use a data structure known as a **composite structure**, or, more commonly, a **record** or a **user-defined type** (UDT).

A record is defined as a collection of named **fields** with each field normally able to store a single value.

We use a record for our data when we want to group related items of data. For example, we might use a record structure to store a name, phone number, and email address of an individual.

Setting Up a Record

Defining the Structure

Before we can create a record variable, we must first define the structure of the record we intend to use. This acts as the blueprint for any subsequent variables declared to be of this type.

Let's assume we want to store the ID code, gender, age, and final exam mark a student achieves. In C++ we would declare the record structure with the lines

```
struct StudentType
{
   int   id;
   char  gender;
   int   age;
   int   mark;
};
```

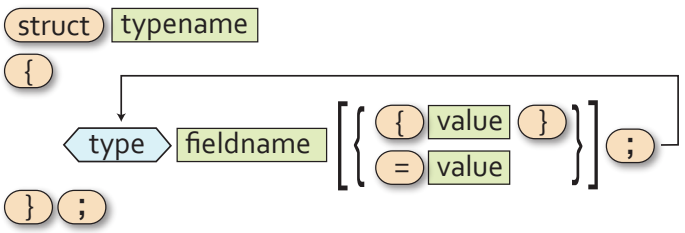
Note that the definition begins with the keyword `struct` (short for *structure*) followed by the name we wish to assign to this particular record design. The name used, *StudentType*, includes the term *Type*. Although not required, adding this helps remind anyone looking at the code that this is the definition of a record layout and not the name of a variable.

The fields within the record structure are enclosed within braces and are identified by their individual types and names - just like standard variables. If required, they can be assigned initial values.

The syntax for creating a `struct` definition is shown in FIG-9.1.

FIG-9.1

struct Syntax



where:

typename	is an identifier giving the type name assigned to the composite structure.
type	is a known data type such as `int`, `float`, `char` etc. This can also be a previously-defined user type.
fieldname	is an identifier giving the name assigned to a field within the structure.
value	is the initial value to be assigned to the field. If no value is given, the field will contain some random value.

Note we can see from the diagram that the structure can contain an unlimited number of fields.

Defining Variables

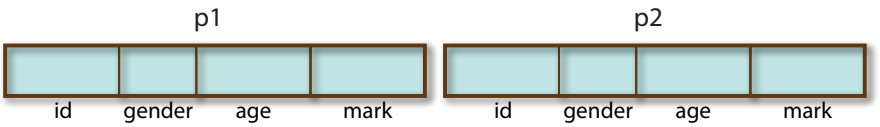
After the record structure has be defined, we are free to create variables of this new structure in much the same way as we might create standard `int`, `float` and `char` variables. For example, we could create variable *p1* and *p2* of type `StudentType` using the line

```
StudentType p1,p2;
```

A visualisation of what is created by this line is shown in FIG-9.2.

FIG-9.2

Two Record Variables



An alternative method of defining the variables' structure is to omit the record structure's type name and declare the variables immediately after the `struct`'s closing brace:

```
struct
{
   int   id;
   char  gender;
   int   age;
   int   mark;
} p1,p2;
```

The first approach is probably the more flexible and is the most widely used.

Arrays within Records

Fields within a record structure can be arrays. For example, we might allow up to five exam marks to be recorded for each student by changing the definition of the last field in our previous structure to be defined as

```
int   marks[5];
```

or we could replace the student's ID with their name by creating the first field in the structure using the line

```
char name[31];
```

which would allow up to 30 characters to be stored as the student's name. These changes would give us the following definition:

```
struct StudentType
{
   char  name[31];
   char  gender;
   int   age;
   int   marks[5];
};
```

Like standard variables, arrays defined within a `struct` can be assigned an initial value. For example,

```
struct StudentType
{
   char  name[31]{"John Doe"};
   char  gender;
   int   age;
   int   marks[5]{0};
};
```

Activity 9.1

Write a record structure layout called *ContactType* which contains the following fields:

name	(up to 40 characters)
email	(up to 60 characters) x 3
mobile	(up to 15 characters)
home	(up to 15 characters)

Create three variables of this type. These should be named *c1*, *c2* and *c3*.

Arrays of Records

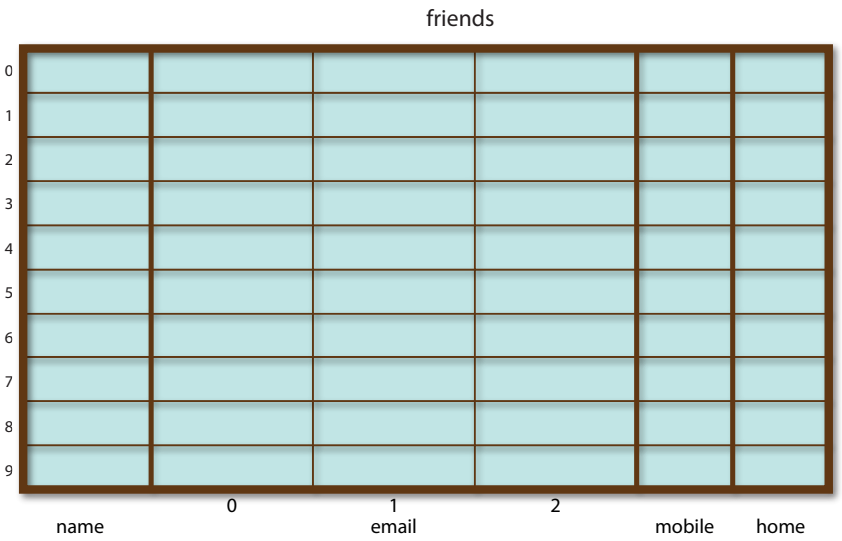
In the Activity 9.1 we created three variables of type `ContactType`, but obviously if we were writing an application to store contact details we might need many more such variables. One way to do this would be to create an array of these structures (sometimes called a **table**). For example, we could create a modest 10 element array in which each element was of type `ContactType` using the line

```
ContactType friends[10];
```

This would produce the array shown in FIG-9.3.

FIG-9.3

An Array of Records



Initialising *struct* Variables

Although we can assign initial values to the fields of a `struct` as part of its definition, that means that every variable we create of that type will have those same values. If we want to assign different values to each variable then, like standard variables, we can specify the starting values for a specific `struct` variable.

The syntax is similar to that of an array, with a value given for each field in the record. For example,

```
ContactType f1 = {"Margaret Adams","margaret@digital.com",
↳"","","0756123111","0163172219"};
```

Notice that the second and third email addresses have been set up as empty strings.

C++ will allow us to enclose the values for the cells of an array field in a set of braces to help indicate which values are assigned to which fields:

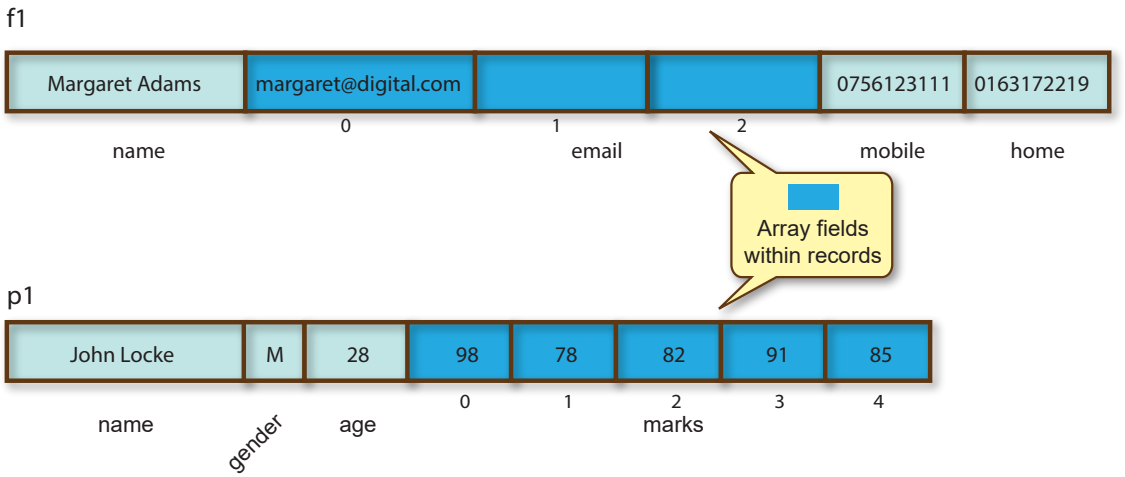
```
ContactType f1= {"Margaret Adams",{"margaret@digital.com",
↳"",""},"0756123111","0163172219"};
```

or

```
StudentType p1 = {"John Locke",'M',28,{98,78,82,91,85}};
```

The effect of these two statements are shown in FIG-9.4.

FIG-9.4 Initialising Records containing Arrays



When we have an array of records, inner braces are used for each record:

```
ContactType friends[2] =
{
   {"Elizabeth Bennett","liz@netherfield.com",
   "MrsDarcy@pemberley.co.uk","", "01176352119","081234563"},
   {"Fitzwilliam Darcy","darcy@pemberley.co.uk","","",
   "0118234556","071112311"}
};
```

As usual when initialising an array, we have the option to omit the number of elements and allow the compiler to calculate the value. This means that the previous declaration could have begun with the line

```
ContactType friends[] =
```

Activity 9.2

Start a new project called *Records* and create a three element array of type *StudentType* and intialise the elements as follows:

Pip,M,17,23,29,31,47,59
Estella,F,19,31,38,42,53,78
Clara,F,20,18,21,27,29,25

Accessing Record Fields

If we assume a program already contains the definition for *StudentType* and a declaration of the variable *p1* as:

```
StudentType p1 = {"John Locke",'M',28,{98,78,82,91,85}};
```

then we can access a field within the record by giving the record's variable name followed by a full stop and the field name. For example, we can access the student's name using the term

```
p1.name
```

In this situation, the humble full stop has the grand title of the **direct member selector**.

We could display the student's name using the line

```
cout << p1.name << endl;
```

or input it from the keyboard using

```
gets_s(p1.name);
```

To access a specific exam mark we need to combine the technique used to access a field with that required to access an element within an array. This means the second mark achieved by our student can be displayed using the line

```
cout << p1.marks[1] << endl;
```

To display all five marks we would use the code:

```
for (int exam = 0; exam <= 4; exam++)
   cout << p1.marks[exam] << endl;
```

When we have an array of records holding the details of six students as created by the line

```
StudentType hdcomp[6];
```

then we need to start by specifying the element to be accessed, followed by the full stop and field required. To display the age of the third student, we would use the line

```
cout << hdcomp[2].age << endl;
```

The first mark of the same student would be accessed using the term

```
hdcomp[2].marks[0]
```

The names of all six students would be displayed by the code:

```
for(int student =0; student < 6; student++)
   cout << hdcomp[student].name << endl;
```

To display the five scores of all six students, we would use the lines:

```
for(int student = 0; student < 6; student++)
{
```

```
      for(int exam = 0; exam < 5; exam++)
         cout << hdcomp[student].marks[exam]<<' ';
      cout << endl;
   }
```

Activity 9.3

Modify *Records* so that the contents of each record are displayed on a separate line.

Range-Based *for* and Records

The solution to Activity 9.3 involves a great deal of typing, much of which is caused by the repeated use of the term

```
fiction[student]
```

One way to avoid this is to copy the record from the array into a stand-alone record by using a range-based `for` loop. This means changing the line

```
for(int student = 0; student < 6; student++)
```

to

```
for (StudentType v: fiction)
```

We can then replace all occurrences of `fiction[student]` with `v`:

```
for (StudentType v: fiction)
{
   //** Display name, gender and age **
   cout << setw(20) << v.name << ' ' << v.gender << ' ' <<
   ↳v.age;
```

And we can use the same tactic when accessing the exam marks:

```
for (int exam : v.marks)
   //** Display mark **
   cout << setw(4) << exam;
```

Notice that we use the term `v.marks` to identify the array being accessed.

Activity 9.4

Rewrite *Records* so that range-based `for`s are used to access each record and each exam mark.

Copying Records

As implied by the use of the range-based `for`, we can copy the complete contents of a record to another variable of the same type with a single statement. For example, if we define *temp* as

```
StudentType temp;
```

we could copy the first record of the array into this new variable with the single line:

```
temp = students[0];
```

Nested Records

If we need to record a date, rather than use three separate variables as we have done in previous chapters, it would be better to define the following record structure:

```
struct DateType
{
   int day;
   int month;
   int year;
};
```

Because all three fields are of the same type we could have defined the fields with a single line:

```
struct DateType
{
   int day, month, year;
};
```

But, as a general rule it is best to define each field on a separate line since this is visually less cluttered and also allows us to add a comment if necessary giving the purpose of each field.

Now, suppose we want to record the name and date of birth of a customer. To do this we need a record structure with two fields; the first for the name and the second for the date of birth. What's new here is that the date of birth is itself a record structure.

The definition is written as

```
struct CustomerType
{
   char name[31];
   DateType dob;
};
```

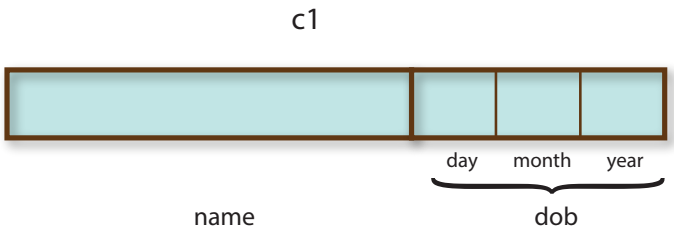
A variable of this type,

```
CustomerType c1;
```

creates the data area shown in FIG-9.5.

FIG-9.5

A Nested Record Structure



Accessing the *name* field of the record isn't a problem (we'd use `c1.name`) but the term `c1.dob` refers to the whole date, and, since that date is itself a record structure, accessing the individual parts of the date require the terms

```
c1.dob.day
c1.dob.month
c1.dob.year
```

Activity 9.5

A game records the name and fastest time achieved by a player. The time is recorded in minutes and seconds.

Create a new project, *GameScores*, and add a `struct` definition called *TimeType* which can be used to record the time.

Write a second `struct` definition called *TopScoreType* which contains two fields: *name* (up to 30 characters) and *time*. The field *time* is of type *TimeType*.

The game records the five highest scores. Declare and array called *topscores* of type *TopScoreType* which contains five elements.

Write the lines of code necessary to set the first recorded score to "Davy Jones", 3:49. Check the value has been assigned by displaying the record.

Positioning the *struct* Definition Code

So far we have placed the definition of any `struct` types within `main()`. And, since the programs compiled and ran, this is obviously acceptable to the compiler. However, this is not the usual place for `struct` type's definition. Instead, it is normally placed before `main()` as shown below:

```
#include <iostream>
#include <iomanip>
using namespace std;

struct StudentType
{
   char name[31]{ "John Doe" };
   char gender;
   int   age;
   int   marks[5]{ 0 };
};

int main()
{
```

This positioning increases the scope of the type, allowing us to declare variables of *StudentType* anywhere within our program.

By placing the code within `main()` we restrict where the variables may be declared to within *main()* itself. Of course, we might reasonably point out that the code within `main()` is the only code any of our previous programs have had! But that will change in later chapters at which point the positioning of our `struct` code will then have consequences.

Summary

- Use `struct` to create a user-defined data type containing two or more elements.
- The term *record structure* is often used to describe a `struct` definition.
- Elements within a `struct` declaration are known as **fields**.
- Fields can be assigned an initial value within the structure's definition.
- A record structure can contain arrays as fields.

- A record structure can contain other record structures as fields.
- Once defined, variables of `struct` data type can be declared. This variable is often referred to as a record.
- An array of records can be defined.
- An array of records is sometimes called a table.
- To access a field within a record, use the form

```
record_name.field_name
```

- The full stop between the variable and field name is known as the direct member selector.
- We can make use of a range-based `for` loop to iterate through each record in a table.
- The contents of a record variable can be copied to another variable of the same type using the assignment operator.
- `struct` definitions should be placed before the code for `main()`.

Enumerated Types

Introduction

Although it is always possible to use a number to represent any value - for example, we previously used the numbers 0 to 6 to represent the days of the week - that approach can make the meaning of a variable somewhat obscure to others reading our code. An alternative approach is to use strings - "Sunday", "Monday", etc. - but this limits us in other ways; for example, we cannot add 3 to the string "Sunday" to change the value held to "Wednesday".

C++ offers another way of handling this type of situation: **enumerated types**.

Using Enumerated Types

Declaring a Standard Enumerated Type

An enumerated type allows us to create a new data type and to specify the set of values that can be assigned to variables of that type. To create the days of the week as an enumerated type we would write:



Like `struct` definitions, `enum` coding should be placed before `main()`.

```
enum DaysType {Sunday,Monday,Tuesday,Wednesday,Thursday,
↳Friday,Saturday};
```

The statement starts with the keyword `enum` followed by an identifier giving the name we wish to assign to this type. Inside the braces, the possible values that may be assigned are listed. Note that the values, known as **enumerators**, are not strings (no double quotes). Instead, they represent the actual values that might be assigned to any variable of this type. These values must conform to the rules of any identifiers so, no spaces or keywords are allowed within the list.

Activity 9.6

Create an enumerated type called *SeasonsType* which allows the values, *Winter*, *Spring, Summer, Fall.*

Alternative declaration syntax allows us to declare variables at the end of the `enum` statement as shown in the line

```
enum DaysType {Sunday,Monday,Tuesday,Wednesday,Thursday,
↳Friday,Saturday} day1, day2;
```

which creates two variables, *day1* and *day2* of type *DaysType*.

Using an anonymous `enum` type is also valid. For example,

```
enum {Sunday,Monday,Tuesday,Wednesday,Thursday,
↳Friday,Saturday} day1, day2;
```

which creates the same variables as before but does no assign a name to the enumerated type.

In reality, enumerate values are held using integer codes, so the first value in the list is associated with the value zero, the next with value 1, etc.

If it suits our purposes we can override these associations and assign different integer values. For example, we could write

```
enum DaysType {Sunday=3,Monday,Tuesday,Wednesday,Thursday,
↳Friday,Saturday};
```

With *Sunday* set to 3, the subsequent values are incremented from this base value, making *Monday* 4, *Tuesday* 5, and *Saturday* 9.

We can assign a value to as many of the enumerated elements as we wish. Unspecified elements are incremented from the previous element as usual. For example,

```
enum DaysType {Sunday=3,Monday=6,Tuesday=11,Wednesday,
↳Thursday,Friday,Saturday};
```

would record *Sunday* as the integer value 3, *Monday* as 6, *Tuesday* as 11, *Wednesday* as 12, and *Saturday* as 15.

We are even allowed to assign the same integer to more than one enumerated value. For example, if we were to set up a months-of-the-year enumerated type, we might want to assign the days in the given month:

```
enum MonthsType {January=31,February=28,March=31,
↳April=30,May=31,June=30,July=31,August=31,September=30,
↳October=31,November=30,December=31};
```

It is possible to use a previously specified enumerator as the value for a subsequent one. For example,

```
enum MonthType{January = 31, Jan = January, ...
```

Perhaps a more practical example for enumerated types with associated values might be to set up a customer type and associated sales discount percentage:

```
enum DiscountType {regular=0, gold=2, bulk=5, special=10};
```

There is a second – and preferred – way of setting up enumerated types and that is to add the term `struct` (or `class`) immediately after the `enum` term as in

```
enum struct DaysType {Sunday,Monday,Tuesday,Wednesday,Thursday,
   ↳Friday,Saturday};
```

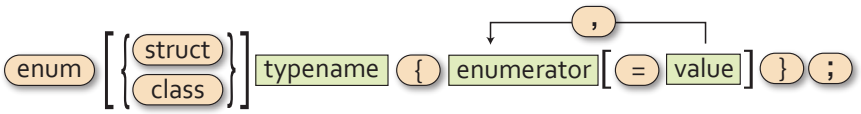
It doesn't matter which term we use (`struct` or `class`) but here we'll save `class` for object-oriented class structures which are covered in a later chapter and therefore stick to using `struct` in our enumerated types.

Adding the new term has no effect on the options available when setting up an enumerated type, but it does affect how we assign values to variables of this type.

The general syntax for defining an enumerated type is shown in FIG-9.6.

FIG-9.6

enum Syntax



where:

typename	is an identifier giving the name assigned to the enumerated type.
enumerator	is an identifier giving one of the values available within the enumerated grouping.
value	is a ordinal value (usually of type `int`) which is

associated with that specific enumerator. If no value is given the value assigned is one greater than that assigned to the previous enumerator in the list. If no value is given to the first enumerator in the list, it is assigned the value zero.

Enumerated types defined without the term `struct` or `class` are known as **unscoped enumerations**; those that include the term `struct` or `class` are known as **scoped enumerations**.

Using Enumerated Variables

Once we have defined a named enumerated type, a variable of that type can be created using a statement such as

```
DiscountType cust;
```

The variable can then be assigned any value defined for the type. How strictly the compiler checks that value depends on whether we have a scoped or unscoped enumeration. Hence, if `struct`/`class` is missing (unscoped enumeration) then we might write

```
cust = gold;
```

When dealing with a scoped enumeration, then we must write the assignment as

```
cust = DiscountType::gold;
```

including the enumerated type name and the scope resolution operator. But we cannot assign an integer value to *cust* even if it's one of those listed in the `enum` definition:

```
cust = 2; //Error
```

Instead, we have to cast the value to the correct type:

```
cust = static_cast<DiscountType>(2);
```

Ideally, we might want to calculate the discount due to the customer with code such as:

```
float discount = totalcost * cust/100.0;
```

How the compiler reacts to this often depends on how *DiscountType* has been defined. When `struct`/`class` has been used in the definition, then the line above will not compile since *cust* cannot be converted to a numeric type, but without `struct`/`class` *cust*'s conversion from *DiscountType* to `int` will be performed automatically. To get what we want when the enumeration has been scoped, we need to cast *cust* to an `int` value:

```
float discount = totalcost * static_cast<int>(cust)/100.0;
```

Using enumerated types allows us to create code which is easier to understand but there are a few limitations. An enumerated value cannot be input nor displayed. Hence, after declaring a variable

```
DaysType day {DaysType::Tuesday};
```

the line

```
cout << day << endl;
```

will produce either 2 (*DaysType* is unscoped) or produce an error (*DaysType* is scoped).

Using either declaration format, the line

```
cin >> day;
```

will fail to compile

At this point, we may be forgiven for thinking that defining *day* as a string (`char` array) would have been more useful, since we could then use the `cin` and `cout` statements without problem. However, using enumerated types ensures that any variable of that type can only be assigned a value specified within the enumerated type definition; strings or integers, on the other hand, can be assigned any value - valid or otherwise.

As a general rule, enumerated variables are used to record the state of various elements within a program but not for I/O operations.

It would also be nice if we could write lines such as

```
day++;
day = day - 1;
```

to change the value in *day* from *Tuesday* to *Wednesday* or *Tuesday* to *Monday*. Unfortunately, the arithmetic operators are not designed to operate on enumerated types. However, with a bit of clever type casting we can get round this problem.

Casting *day* to an integer will allow us to write

```
static_cast<int>(day)+1
```

and casting the result of this back to a *DaysType*, will allow the variable day to be updated

```
day = static_cast<DaysType>(static_cast<int>(day)+1);
```

This only works if the enumerated type's values are associated within consecutive integer values (0, 1, 2, etc.). Any code would also have to handle the problem of increments taking the value outside the range associated with the `enum` type (for example, *day* would have to stay in the range (0 to 6).

Although we can't display the value of an enumerated type directly, we can use a string array to hold the equivalent values and then use the value in the enumerated variable as a subscript to the array. An example of this is shown FIG-9.7.

FIG-9.7

Using Enumerated Type Variables

```
#include <iostream>
using namespace std;

// *** Set up enumerated type ***
enum struct DaysType  {Sunday,Monday,Tuesday,Wednesday,
↳Thursday,Friday,Saturday};

int main()
{
   // *** Set up matching string array ***
   char DaysoftheWeek[7][10]{"Sunday","Monday","Tuesday",
   ↳"Wednesday","Thursday","Friday","Saturday"};
   // *** Declare enumerated variable initialising to Tuesday ***
   DaysType day = DaysType::Tuesday;

   // *** Increment day ***
   day = static_cast<DaysType>(static_cast<int>((day)+1)%7);

   // *** Display corresponding array element ***
   cout << DaysoftheWeek[day] << endl;
}
```

Notice from the program that:

- The value of *day* is incremented using `((day)+1)%7` to ensure the integer value stays in the range 0 to 6.
- The array subscript in the last line does not have to be cast to an integer type.

Activity 9.7

Start a new project called *Enumeration* and enter the code given in FIG-9.7.

Change the `enum` statement to include `struct` and modify any other lines needed to make the program execute again.

We can also test an enumerated variable with a statement such as

```
if (day == DaysType::Monday)
```

Activity 9.8

Modify *Enumeration* so that it also displays "*It's the weekend"* if day is *Saturday* or *Sunday* (don't change the contents of *DaysoftheWeek*).

Test your program by modifying the value assigned to *day*.

Of course, modifying the default integers means we can no longer increment through the various values in the enumerated type nor use all of the relational operators. A test such as

```
if(month == MonthsType::March)
```

will return *true* if *month* has previously been assigned any value associated with the value 31 (*January*, *March*, *May*, *July*, etc.).

The value assigned to the entries in the declaration of an `enum` type can also be of type `char`. For example, the lines

```
enum struct VowelType {A='a', E='e', I='i', O='o', U='u'};
VowelType letter{ VowelType::A };
cout << static_cast<char>(letter) << endl;
```

would output

```
a
```

Summary

- An enumerated type is a user-defined type which defines the set of values which may be assigned to variables of this type.
- Enumerated types can allow for easier understanding of program code.
- Enumerated types ensure that variables must stay within a specified range. Enumerated values cannot be input or output.
- Internally, enumerated values are stored as `int` (or `char`) values.
- If the keyword `enum` is followed by `struct` or `class`, how the values of an enumerated type are referenced changes.
- An enumerated value can be copied to an integer variable.

- An integer value can converted to a enumerated value.
- The integer value used to represent a enumerated value can be specified.
- Multiple enumerated values can be assigned the same integer representation.

Bit Fields

All the data items we have encountered so far occupy a given number of bytes. However, it is possible to define data components that occupy as little as a single bit. This may be desirable to minimise the storage requirements of information. For example, if we need to record the gender of a student, we could define a standard variable for the purpose with the line

```
char gender;
```

and store either the letter 'F' or 'M'. But since there are only two possible values for this variable we could make use of a single bit, storing 0 to represent female and 1 to represent male.

Although this may seem like a trivial space saving exercise, when we have to store information on thousands of people then the savings begin to be significant.

Activity 9.9

How many bits would be needed to store the month of the year when represented by a number in the range 1 to 12?

Although C++ does not allow separate bit-sized variables, we can define record structures which contain bit-sized fields. These fields must be defined as an integral type (`short`, `int`, `long`, `unsigned`, etc.), but, unlike normal fields, the syntax is extended to allow the number of bits in the field to be specified. Hence we could define a record structure holding only gender and month fields with the code

```
struct BitExampleType
{
   unsigned int gender  : 1;
   unsigned int month   : 4;
};
```

The information after the field name gives the number of bits reserved for that field.

We can then create variables of this type

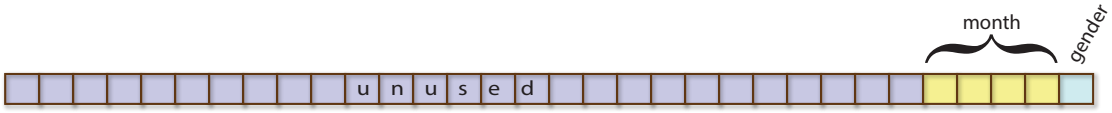
```
BitExampleType v;
```

and access the fields in the standard way

```
v.gender = 0;
v.month = 10;
if (v.gender == 1)
   cout << "Male\n";
```

C++ will assign storage space for the structure based on the underlying type specified in the bit fields. In the case of the *BitExampleType*, this means a 32-bit `unsigned int` space. Field allocation is as shown in FIG-9.8.

FIG-9.8 A Structure Containing Bit Fields

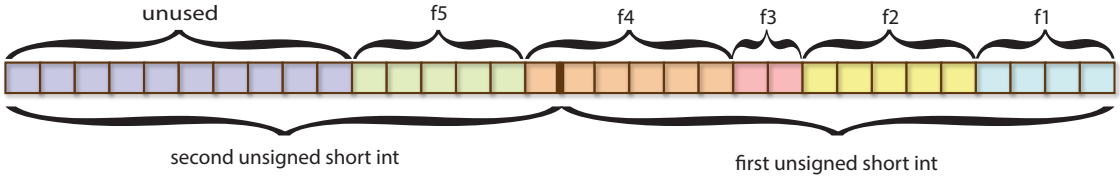


The required bit fields will always be assigned an exact number of storage units of the underlaying type. For example, the code

```
struct MoreBitsType
{
   unsigned short f1    : 4;
   unsigned short f2    : 5;
   unsigned short f3    : 2;
   unsigned short f4    : 6;
   unsigned short f5    : 5;
};
```

create a structure greater than the two bytes of a single `unsigned short int`, so it allocates space for `two unsigned short int`s (see FIG-9.9).

FIG-9.9 Straddling a Data Boundary





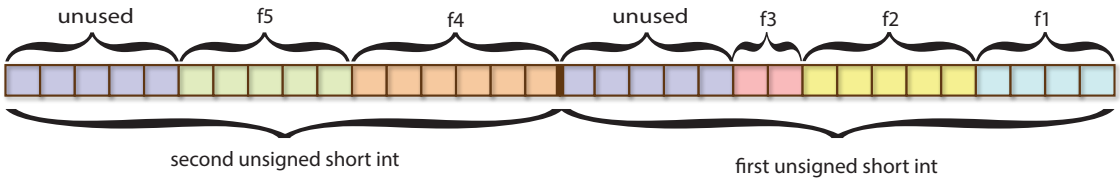
A **word** is a collection of bytes normally handled as a single unit by the processor.

Notice that the field *f4* straddles the two `unsigned short int` spaces. Because of this, accessing *f4* will be a relatively slow process as the machine attempts to combine the bits from the two separate words. Where possible we should avoid fields straddling words in this way. C++ helps with this by allowing us to add unnamed padding fields that allow us to adjust the actual positioning of the named fields within the allocated space. For example, we can move the space allocated to *f4* into the second area using the following updated definition:

```
struct MoreBitsType
{
   unsigned short f1    : 4;
   unsigned short f2    : 5;
   unsigned short f3    : 2;
   unsigned short       : 5; //Unnamed padding
   unsigned short f4    : 6;
   unsigned short f5    : 5;
};
```

This change would result in the allocation shown in FIG-9.10.

FIG-9.10 Adding Unused Fields



We can even get the compiler to do the hard work for us and calculate the number of bits required for the padding by specifying a value of zero bits in the field declaration:

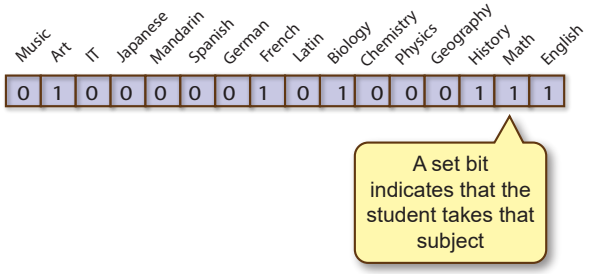
```
struct MoreBitsType
{
   unsigned short f1    : 4;
   unsigned short f2    : 5;
   unsigned short f3    : 2;
   unsigned short       : 0; //Padding calculated automatically
   unsigned short f4    : 6;
   unsigned short f5    : 5;
};
```

Categorising Bits

Whenever we want to record a person or object as belonging to several categories, we can make use of individual bits to indicate membership of specific categories. For example, let's assume that a school offers 16 subjects that its students can choose to study, then we could use the bits of a 16 bit `unsigned short int` to record which subjects an individual student is taking (see FIG-9.11).

FIG-9.11

Bit Flags



We can see from FIG-9.11 that the student in the example takes subjects *Art*, *French*, *Biology*, *History*, *Math* and *English* since the bits matching those subjects are set to 1.

In a program which declares the relevant variable with the line

```
short int subjectstaken;
```

we could set up the bits shown in the diagram above using the line

```
subjectstaken = 0X4147;
```

To interrogate this variable to discover which topics are being taken, we can use some of the operators described below.

Bit-Manipulation Operators

In addition to creating bit fields, it is possible to manipulate the contents of the bits of these fields - or, indeed of any value - using the bit manipulation operators given below.

Bit Complement Operator (~)

The bit complement operator changes the value of the bits within a variable or constant. 1's are changed to 0's and 0's to 1's.

For example,

```
short int v {0X00FF};
```

will create the variable space shown in FIG-9.12.

FIG-9.12

Variable Contents (Binary)



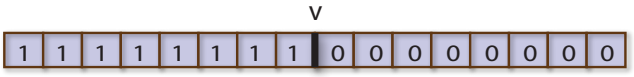
Complementing the contents with the line

```
v = ~v;
```

changes the value held to that shown in FIG-9.13.

FIG-9.13

Result of Performing the Complement Operation



Activity 9.10

How would the value shown in FIG-9.13 be displayed (in decimal) if it were taken to be

a) a `signed short int`
b) an `unsigned short int`

The results of the complement are shown by the program in FIG-9.14.

FIG-9.14

Using Bitwise NOT

```
#include <iostream>
using namespace std;

int main()
{
   // *** Initialise variable ***
   short int v{ 0X00FF };

   // *** Complement variable's bits ***
   v = ~v;

   // *** Display signed and unsigned versions ***
   cout << "Signed value : " << v << endl;
   cout << "Unsigned value : " << static_cast<unsigned short int>
   ↳(v) << endl;
}
```

Activity 9.11

Start a new project called *BitInverse* and implement the code given in FIG-9.14.

Activity 9.12

If the variable *subjectstaken* (see FIG-9.11) is used to record which subject are studied by a pupil, how could we record the subject not taken by the same student in a variable *subjectsnottaken* (the bits for subjects not taken should be set to 1 in this variable).

Shift Left Operator (<<)

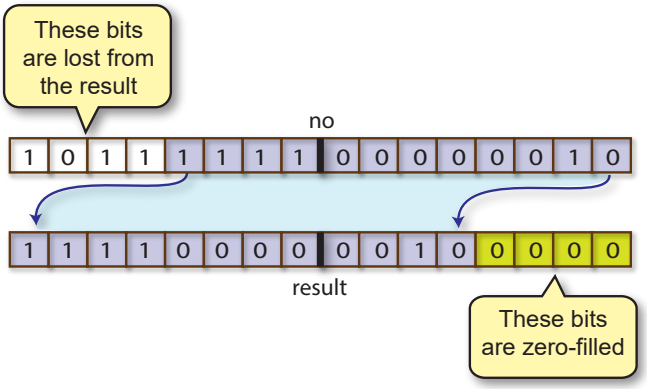
The shift left operator allows us to move a value a specified number of bits to the left. For example, the lines

```
unsigned short int no = 0XBF02;
unsigned short int result = no << 4;
```

would store a value in *result* which is the contents of *no* shifted three bits to the left (see FIG-9.15).

FIG-9.15

Result of the Using Shift Left Operator



Notice that the left-most bits of the original value (in *no*) are lost from *result*. When a value is too large to fit in the allocated space, this is termed **overflow**.

The left-shift operator can be used as an efficient way of multiplying a value by a power of two. Hence, the lines

```
int no = 13;
int result = no << 4;
```

gives the same result as

```
int no = 13;
int result = no * 16;
```

Of course, the result is only accurate if it does not overflow the space allocated.

Activity 9.13

Start a new project called *ShiftLeft* which reads in a `short int` value in the range 1 to 100 and multiplies the value by 16 using the shift left operator. Display the result.

When a value is stored in a `signed` variable, we can also make use of the shift left operator to determine the contents of a specific bit. Since a `short int` value is allocated two bytes (with the bits identified as 0 to 15 starting from the right-most bit) we could determine the setting of bit 10 by shifting it to position 15 with the line

```
short int no = 13;
short int result = no << 5;   //shifts bit 10 to position 15
```

and, since bit 15 is used as a sign bit (see Chapter 2), if that bit is set to 1, C++ will consider the contents of the variable to be negative:

```
if (result < 0)
   cout << "Bit 10 is set\n";
```

Activity 9.14

Modify *ShiftLeft* so that, rather than multiply the entered value by 16, it checks the setting of bit 3 of the entered value. It should then display either *"Bit 3 is set to 1"* or *"Bit 3 is set to 0"* as appropriate.

Shift Right Operator (>>)

The shift right operator moves a value a specified number of places to the right. This

is a good way to divide by a power of two: each shift to the right is equivalent of a single integer division by two with any remainder being lost.

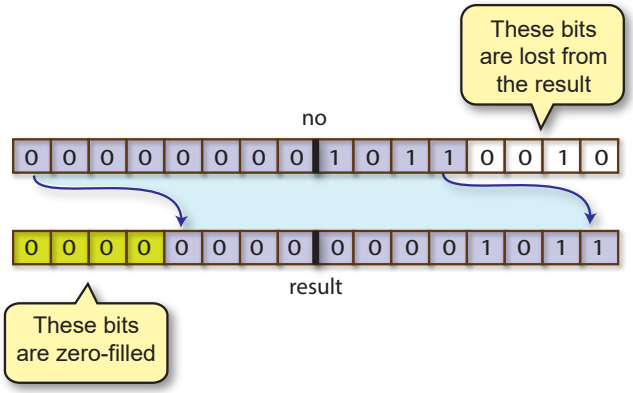
Assuming the `unsigned short int`, *no* has been assigned the value 0X00B2 as in the previous example, then the statement

```
result = no >> 4;
```

would create the result shown in FIG-9.16.

FIG-9.16

Shifting an Unsigned Value Right



However, when right-shifting a signed value then the left hand bits are filled with the same value as that of the sign bit. For example, the variable in the statement

```
short int num = -16;
```

would be stored as shown in FIG-9.17.

FIG-9.17

-16 Stored in a Signed short int



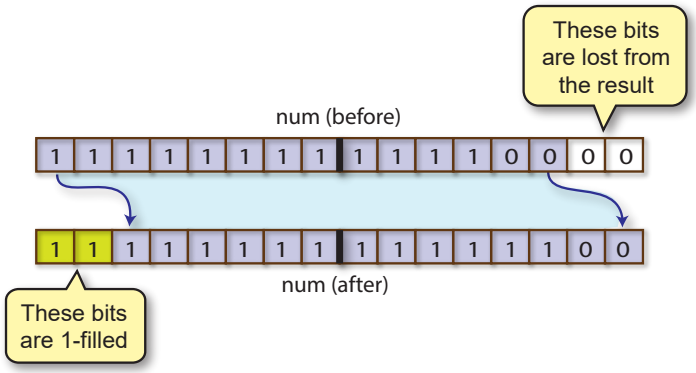
Since the sign bit is set to 1, any new bits introduced to the left of the storage space due to a shift right will also be 1s. For example,

```
num = num >> 2;
```

would create the result shown in FIG-9.18 which is the binary equivalent of -4.

FIG-9.18

Shifting a Signed Value Right



Bitwise AND (&)

The bitwise AND operator ANDs the individual bits of two integral values. Bits from

the corresponding positions in each value are ANDed together; where both bits contain a 1, the result is also a 1, otherwise the result is a 0. A typical statement might be

```
result = no1 & 0X04;
```

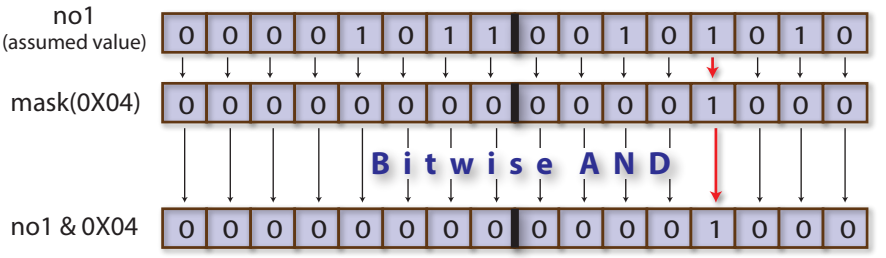
The second value in any bitwise operation is often referred to as a **mask** since it can be used to "mask out" the contents of bits in the first value. For example, suppose we want to know the value stored in bit 3 of the `unsigned short` *no1*; we could do this with the code

```
if (no1 & 0X04 > 0)
   cout << "Bit 3 = 1\n";
else
   cout << "Bit 3 = 0\n";
```

FIG-9.19 shows the process involved.

FIG-9.19

Result of the Bitwise AND Operator



Activity 9.15

A school maintains a record of the subjects being taken by each student in the school. This record structure has the form

```
char name[31]
char group[11]
short int subjects
```

The 16 bits of the subjects *field uses individual bits to record which subjects that student is studying. The bits are designated (left-to-right):*

English, math, history, geography, physics, chemistry, biology, Latin, French, German, Spanish, Mandarin, Japanese, IT, modern studies, economics.

So, for someone taking English, math, history, Latin, Mandarin *and* IT*, the* subjects *field would have the binary value*

1110000100010100

Create a project, *Students*, which holds the details of the 20 students in group *1A2014*. Names should be transferred from a *names* array (this will save you having to type them in) and subjects taken randomly generated making sure that each student takes exactly six subjects.

The program should display the details of each student giving the names of the subjects being taken and finally it should list the names of all students taking IT as one of their subjects.

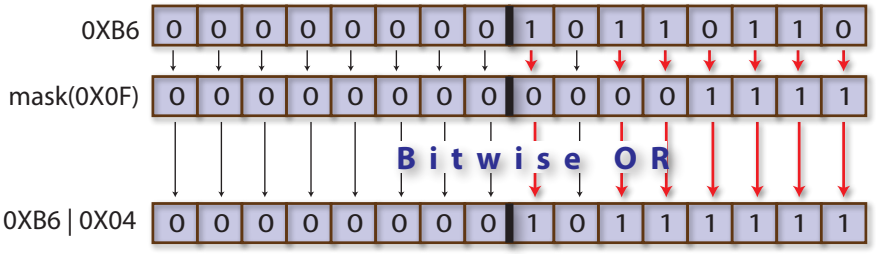
Bitwise OR (|)

The bitwise OR operator performs the OR operation on the bits of the two values

involved. The resulting bit is 0 if both bits of the operands are 0, otherwise the result is a 1. The final result of using the bitwise OR operation on the two `unsigned short` values 0XB6 and 0X0F (0XB6 | 0X0F) is shown in FIG-9.20.

FIG-9.20

Result of the Bitwise OR Operator

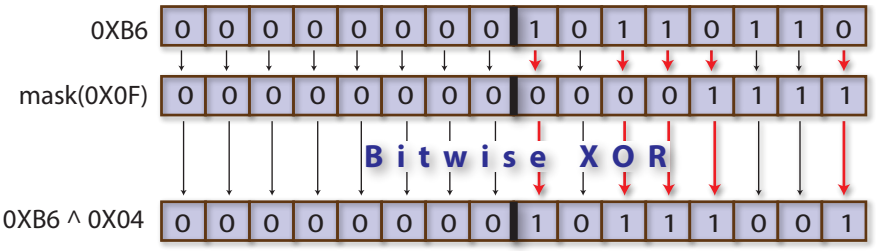


Bitwise Exclusive OR (^)

Using exclusive OR (often written as XOR), the result is a 1 if the two bits involved have different values and a 0 when the two bits have the same value. The result of the calculation `0XB6 ^ 0X0F` is shown in FIG-9.21 (`unsigned short` values are assumed).

FIG-9.21

Result of the Bitwise XORt Operator



The interesting thing about the XOR operation is that, if we apply the same mask value to the result, we end up with the original value. If we take the example shown above we see that

```
0XB6 ^ 0X0F = 0XB9
```

Taking the result (0XB9) and XOR with the original mask (0X0F) we get

```
0XB9 ^ 0X0F = 0XB6
```

which is the original value in the first calculation.

The XOR operation can also be used as a quick and simple (though not very secure) way to encrypt/decrypt data.

Activity 9.16

Start a new project, *Encrypted*, which reads in a string and then encrypts each character by XORing it with the value 0X5A.

The program should display the original string, the encrypted and decrypted versions of the string.

Bitwise Assignment Operators

Just as we can write

```
sum += no;
```

to add the contents of *no* to *sum*, we can also use this same style when making use of the bitwise operators. for example, rather than write

```
ch = ch ^ 0X5A;
```

we can write

```
ch ^= 0X5A;
```

A full list of these **bitwise assignment operators** is given in FIG-9.22.

FIG-9.22

Bitwise Assignment Operators

Operator	Description	Example	Equivalent
<<=	Shift left assignment	no1<<=2	no1=no1<<2
>>=	Shift right assignment	no1>>=3	no1=no1>>3
&=	Bitwise AND assignment	no1&=0X0F	no1=no1&0X0F
|=	Bitwise OR assignment	no1|=no2	no1=no1|no2
^=	Bitwise XOR assignment	no1^=0X07	no1=no1^0X07

Summary

- A bit field is a field containing a specific number of bits of storage.
- Bit fields must be defined within a struct type.
- It is best to avoid allowing bit fields to cross word boundaries.
- To avoid having a bit field straddle a word boundary unnamed bit fields can be defined within a struct type.
- Bit fields and other integer values can be manipulated using bitwise operators
- Use the bit complement operator (~) to invert the value of a set of bits.
- Use the shift left operator (<<) to shift a value a specified number of bits to the left.
- The bit left operator can be used to multiply a value by a specific power of 2.
- Use the shift right operator (>>) to shift a value a specified number of bits to the right.
- Use the bitwise AND operator (&) to perform the AND operation on the bits of two values.
- Use the bitwise OR operator (|) to perform the OR operation on the bits of two values.
- Use the bitwise exclusive OR operator (^) to perform the exclusive OR operation on the bits of two values.
- Use a bitwise assignment operator when using a bitwise operator on a variable that appears on both sides of the assignment operator.

Union

Concept

Union is a way of defining the same area of memory in two or more ways. For example, we might define an area as an `int` and then define that same area of four bytes as two `short int` or four `char` fields.

Defining a Union

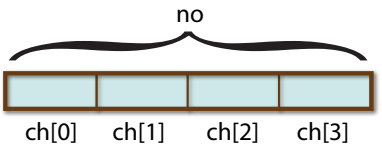
At first glance a union definition looks much like a `struct` definition, the only difference being that the word `struct` is replaced by the keyword `union`. For example,

```
union SameAreaType
{
   int no;
   char ch[4];
};
```

However, the union definition declares the integer *no* and `char` array *ch* occupy the same area of memory (see FIG-9.23).

FIG-9.23

Union



Once we declare a variable of this type

```
SameAreaType v;
```

we can access the alternative definitions of the area using the same syntax as we do to reach a field within a record. Hence,

```
v.no = 0X12345678;
```

assigns a value spread over the four bytes while the line

```
cout<<"byte 0: " <<hex<<static_cast<int>(v.ch[0])<<endl;
```

would display the contents of the least-significant byte which is 78_{16}.

The program in FIG-9.24 displays the contents of all four bytes of an integer.

FIG-9.24

Using Union

```
#include <iostream>
#include <string>
using namespace std;

union SameAreaType
{
   int no;
   char ch[4];
};

int main()
{
   SameAreaType v;
   // *** Assign a value to no ***
   v.no = 0X12345678;
```



FIG-9.24
(continued)

Using Union

```
   // *** Display each byte ***
   for(int c=0; c<= 3; c++)
      cout << "Contents of byte " << c << " is "<< hex <<
      ↳static_cast<int>(v.ch[c]) << endl;
}
```

Activity 9.17

Start a new project called *TestUnion* and implement the code given in FIG-9.24.

We can combine a definition of bit fields and a union to allow us to see the value in the sign, exponent and significand of a floating-point value by making use of a union as shown in FIG-9.25.

FIG-9.25

Union and Bit Fields

```
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;

// *** Structure for a float value ***
struct FloatType
{
   unsigned int significand :23;
   unsigned int exponent   : 8;
   unsigned int sign : 1;
};

// *** Overlay a standard float and FloatType layout ***
union TestUnionType
{
   float no;
   FloatType f;
};

int main()
{
   TestUnionType v;
   // *** Assign a value to the variable ***
   v.no = 3;

   // *** Display the parts of the float value in hexadecimal ***
   cout << "sign "<< v.f.sign<<endl;
   cout << "Exponent "<< hex << v.f.exponent<<endl;
   cout << "Significand " << v.f.significand<<endl;
}
```

Note the order in which the parts of the `float` format are defined within the `struct` statement with the least-significant element given first.

Activity 9.18

Start a new project, *TestUnionBits*, and implement the code given above. Test and save your code.

Try assigning other values to *v.no* and see how the contents of the displayed elements change.

Summary

- Union is a method for defining the an area of memory in two or more ways.
- Union is useful when we want to examine the contents of an area of memory in differing ways.

Type Aliases

Introduction

C++ allows us assign an alternative name to an existing type. That type can be a standard C++ type such as `int`, `float` or `char` but it can also be a user-defined type such as defined in a `struct`, `enum` or `union` statement.

Defining an Alias

typedef

The `typedef` statement is used to define an alternative name for an existing type. For example we could create the alternative name *WholeNumberType* for `int` type using the statement

```
typedef int WholeNumberType;
```

Once the `typedef` statement has been used, we can then use this new name to create `int` variables as in the line

```
WholeNumberType a, b, c;
```

using

An additional method of defining an alias is by employing the keyword `using`. With this approach, `int` would be redefined as *WholeNumberType* using the line

```
using WholeNumberType = int;
```

We'll come across `using` again when dealing with classes.

Activity 9.19

Write the lines of code required to create the alias *Boolean* for type `bool` and declare two variables, *a* and *b* to be of type *Boolean*, setting both variables to *true*.

Pairs and Tuples

Introduction

Although we can create a `struct` type for storing two or more values of varying types, C++17 offers us quicker and more efficient ways of assigning two or more values to a single variable.



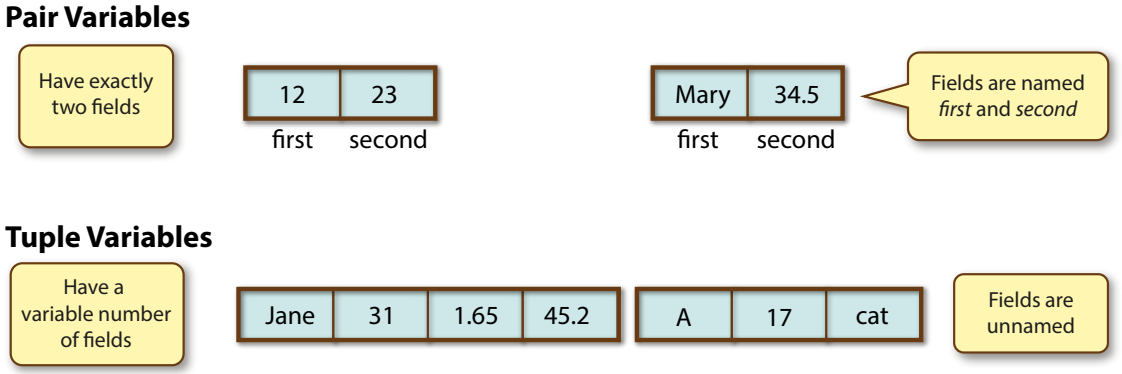
In fact, `pair` and `tuple` are defined as C++ classes. However, at this point in our understanding we may treat them as special types of structures.

The `std::pair` type defines a structure with exactly two fields (named *first* and *second*). The two fields can be of any type and, in fact, both need not be of the same type. For example, we might use such a structure to store a set of two-dimensional coordinates or perhaps a data key and its associated data value.

A tuple (`std::tuple`) is an extension of the `pair` type in that it allows any number of values to be stored in a single variable.

FIG-9.26 visualises possible pair and tuple data.

FIG-9.26 Pairs and Tuples



Coding Pairs

Unlike other variable types, a `pair` variable cannot be created without specify the initial values it is to hold. For example, the line

```
pair pp;
```

would produce a compilation error. Instead, we must a line such as



As usual, the code here assumes that the program in which they are used contains the line

```
using namespace std;
```

```
pair pp{ 44, 3.3 };
```

Parentheses may be used rather than braces:

```
pair pp(44,3.3);
```

Of course, the initial values given can also be variable names or expressions:

```
int no1 = 21;
double dv = 44.4;
pair pp{ no1, dv };
pair pp2{ no1*2, dv/2.0 };
```

An alternative declaration allows us to specify the type of value being stored in the `pair` variable:

```
pair <int, double> pp{ 44, 3.3 };
```

This will force the values given to be stored in the format specified assuming the compiler "knows" how to convert the given value to the type required. For example,

```
pair <int, int> pp2{ 88, 'A' };
```

would store the second value in *pp2* as 65, the ASCII value of a capital A.

Where no conversion is possible, a compiler error message will be displayed. For example, the line

```
pair <int, int> pp2{ 88, "cat" };
```

produces the error message

```
Error    C2440   'initializing': cannot convert from 'initializer list' to
'std::pair<int,int>'
```

We can access the contents of our variable using the field names *first* and *second*:

```
cout << pp.first << endl;
cout << pp.second << endl;
```

We can also use these terms to modify the contents of a `pair` variable as in the line

```
pp.first = 18;
```

Any value assigned in this way must be of the same type (or be convertible to that type) as the original value.

make_pair()

We can construct a `pair` value using the `make_pair()` function. This takes two parameters (of any type) and returns a `pair` value.

The simple program in FIG-9.27 sets up a pair variable then displays its contents.

FIG-9.27

Using a *pair* Variable

```
#include <iostream>

using namespace std;

int main()
{
   pair pv = make_pair( 12,36 );
   cout << pv.first << endl;
   cout << pv.second << endl;
}
```

Activity 9.20

Start a new project called *TestPairAndTuple* and implement the program given above.

Modify the values held in variable *pv* to "Answer" and 42.17 and check that the program runs as before.

A newer approach which replaces the need for `make_pair()` is to use constructor initialisation (see Chapter 16) with a line such as

```
pair pv = pair(12,36);
```

swap()

While we can copy the contents of one `pair` variable to another (assuming both variables contain the same value types) with a statement such as

```
pair pv2 = pv;
```

we can also swap the contents of two `pair` variables using the `swap()` function. For example,

```
swap(pv,pv2);
```

would exchange the contents of *pv* and *pv2* assuming both were `pair` variables where the two variables. Again, this will only work if the combination of types in the two variables match. For example,

```
pair pv = make_pair(12, -9.2);
pair pv2 = make_pair(-7,13.6);
swap(pv,pv2);
```

would be valid since *pv* and *pv2* both have an `int`, `double` values pairing.

On the other hand, the code

```
pair pv = make_pair(12, -9.2);
pair pv2 = make_pair("Wrong",13.6);
swap(pv,pv2);
```

would produce a compilation error since *pv* as `int`, `double` pair while *pv2* as a `char*`, `int` pair.

Activity 9.21

Modify `main()` in *TestPairAndTuple* to create a second pair variable, *pv2*, intialised to ("Wrong",13.6). Use `swap()` to exchange the contents of both variables.

Display the contents of both variables before and after the swap. Does the program compile and run?

Modify the contents of *pv2* to (12, -9.2). Does the program compile and run?

get()

Although we can easily extract a copy of the values in a `pair` variable, we can also extract a reference to those values using the `get()` function. This function has a rather strange format in that the after the function name, the index or type of the value to be extracted must be given in angled brackets before specifying the name of the pair variable in the parentheses. For example to extract a reference to *pv.first*, we would use the expression

```
get<0>(pv)
```

Activity 9.22

In *TestPairAndTuple*, rewrite the last two `cout` statements in `main()` to use `get()` in extracting the values in *pv* and *pv2*.

For the output of *pv*, use the `get<index>` option and for the output of *pv2*, use the `get<type>` option.

Relational Operators

When two `pair` values have identical type pairing (e.g. (`int`, `float`) and (`int`, `float`)) they can be compared using any of the standard relational operators. For example, it is valid to write a line such as

```
if (pv < pv2)
```

The operation works by comparing the first value in each pair. If these are identical then the second values are compared.

Coding Tuples

Unlike `pair` variables which are restricted to exactly two internal values, `tuple` variables can have any number of internal values. However, the fields within a tuple are unnamed.

As with a pair, we must initialise a tuple as part of its declaration:

```
tuple tt{ 1, 3, 8.0, 'F' };
```

We can also specify the data types of the tuple:

```
tuple<int, int, double, char> tt{ 1, 3, 8.0, 'F' };
```

make_tuple()

We can create a tuple by calling the `make_tuple()` function. This takes the values to be placed within the tuple as its set of parameters and returns a tuple value. For example,

```
tuple tv = make_tuple(2, 'c', 9.2, 3.6);
```

When we create a tuple in this way, we still have the option to specify the type of values it holds with a line such as

```
tuple<int,char,double,double>tv = make_tuple(2, 'c', 9.2, 3.6);
```

This allows the compiler to check that the actual values used in constructing the tuple conform to the expected types.

We can also use the simpler constructor initialisation:

```
tuple tv = tuple(2, 'c', 9.2, 3.6);
```

get()

To extract a value from a tuple, we can use the `get()` function which operates in the same fashion as we saw with `pair` types. For example,

```
cout << get<0>(tv) << endl;
```

would display the first value in tuple *tv* (*2*).

We can also use the type option as in the line

```
cout << get<char>(tv) << endl;
```

which would display the `char` value held in the tuple (*c*).

However, the line

```
cout << get<double>(tv) << endl;
```

would fail to compile since there are two `double` values in *tv* (9.2 and 3.6).

Activity 9.23

In *TestPairAndTuple*, remove the previous code from `main()` and create a tuple variable, *tv*.

Display the first two values in *tv* using `get<type>()` and the last two values using `get<index>()`.

swap()

Will swap the contents of two tuples. This will only work when the internal structures of both tuples are identical.

tuple_cat()

The `tuple_cat()` function returns a tuple created by joining together all the tuples given in its parameter list. For example, the lines

```
tuple tv = make_tuple(2, 'c', 9.2, 3.2);
tuple tv2 = make_tuple("Arrow", 23 );
tuple tv3 = tuple_cat(tv, tv2);
```

creates the tuple, *tv3*, as

(2, 'c', 9.2, 3.2, "Arrow", 23)

Activity 9.24

In *TestPairAndTuple*, add tuple *tv2* and *tv3* and display the contents of *tv3* rather than that of *tv*.

tie()

The `tie()` function allows us to copy the values within a tuple to standard variables. To discover how this works, let's assume we have declared two variables:

```
int no;
double dno;
```

and a tuple containing two values, an `int` and a `double`:

```
tuple trf{97,2.8};
```

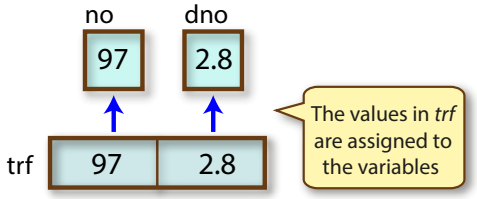
We can copy the tuple's values into *no* and *dno* using the line

```
tie(no, dno) = trf;
```

It looks a bit strange to see a function call on the left-hand side of our assignment statement, but a deeper understanding of what's going on here will have to wait till a later chapter. All we need to know at the moment is that, as a result of this line above, the contents of the tuple are copied to the named variables (see FIG-9.28).

FIG-9.28

Tieing Tuple values to Standard Variables



Activity 9.25

In *TestPairAndTuple*, remove all code from `main()` and create the two variables and tuple as shown above.

Add code to display the contents of the variables and the tuple. Do the values match as shown in the diagram above?

Change the contents of *no* and *dno*, then display the values of *no*, *dno* and the tuple. Are the values in the tuple changed?

The `tie()` function is little used in C++17 where **structured binding** can achieve a similar result (see next section of this chapter).

If there are any values within the tuple that we do not wish to assign to a variable, we can add the term `ignore` at the appropriate point. For example, in the lines

```
tuple trf2{ 97,2.8,'f' };
int no;
char ch;
tie(no,ignore,ch) = trf2;
cout << "no = "<< no << " ch = " << ch << endl;
```

the middle value of *trf2* (2.8) is not copied to any variable, giving us an output of

```
no = 97 ch = f
```

Relational Operators

When two tuples have identical size and type sequence (e.g. (`int`, `float`, `char`) and (`int`, `float`, `char`)) they can be compared using the standard relational operators. For example, it is valid to write a line such as

```
if (tv == tv2)
```

The operation works by comparing each value in the first tuple with the corresponding value in the second. If these are identical then the next values are compared until a difference is found or the end of the sequence is reached.

Summary

pair

- The `pair` type is defined within the `std` namespace.
- A `pair` variable is a data type capable of storing exactly two values.
- The two values can be of any type.
- The two values need not be of the same type.
- The fields within a `pair` variable are named *first* and *second*.
- The simplest way to declare and initialise a `pair` variable is to use the format

  `pair` *identifier*`{`*value1*`,` *value2*`};`
- Use `make_pair()` or `pair()` to create a pair from a sequence of values.
- Use `swap()` to exchange the contents of two pair variables which contain the same sequence of value types.

- Use `get<`*index*`>()` or *variable_name*`.first`, *variable_name*`.second` to access the individual values in a pair.
- All relational operators are available to compare two pair variables of the same structure.

tuple

- The `tuple` type is defined within the `std` namespace.
- A `tuple` variable is a data type capable of storing any number of values which may be of differing types.
- The simplest way to declare and initialise a `tuple` variable is to use the format

  `tuple` *identifier*`{`*value1*`,` *value2, value3, ...*`};`
- Use `make_tuple()` or `tuple()` to create a pair from a sequence of values.
- Use `swap()` to swap the contents of two identically structured tuples.
- Use `get<`*index*`>()` to access the individual values in a tuple.
- Use `tie()` to copy the contents of a tuple to a set of named variables.
- All relational operators are available to compare two tuple variables of the same structure.

Structured Binding

Introduction

Structured binding is a C++17 feature which allows us to bind named variables to the individual elements of a tuple, array, struct or class value. This is similar to the effect of the `tie()` function used in association with tuples.

Demonstrating Structured Binding

We'll start by declaring variables of the type of value for which we might use structured binding.

```
struct CoordsType
{
   double x;
   double y;
   double z;
};

int arr[]{ 2,4,6,8,10 };
CoordsType spot{ 2.9, -1.0, 3.3 };
tuple tv{ 1, 9.8 };
```

Although `tie()` was required in earlier versions of C++ to achieve this link between variables and tuple elements, in C++17 we can achieve the same effect with the following code:

```
auto [no, dno] = tv;
```

with the added bonus that there is no need to explicitly declare *no* or *dno*! Using `auto` allows the compiler to automatically create *no* and *dno* as well as determine their data types.

This ability to create and link individual variables to the elements of a data collection is known as structured binding and its use is not restricted to tuples. It can also be used with arrays, structs, and other data structures which contain multiple values.

For example, all of the following are valid structured binding commands for the variables declared above:

```
auto [v1, v2, v3, v4, v5] = arr;
auto [a, b, c] = spot;
```

Activity 9.26

In *TestPairAndTuple*, modify `main()` by removing the declarations of *no* and *dno*.

Remove the line calling `tie()` and replace it with the line given above. Are the displayed results the same as before?

Of course, as with `tie()`, the newly created variables contain only copies of the original data, so changing the value in one of these variables will have no effect on the original. Hence, the code

```
v1 = 3;
```

```
cout << arr[0] << endl;
```

will display the value 2 since changing the value in *v1* has no effect on *arr[0]*.

But by adding the `&` symbol after `auto`, we can obtain a reference to the original data:

```
auto& [v1, v2, v3, v4, v5] = arr;
```

Since we have now made *v1* and alternative name for *arr[0]*, any changes to the contents of *v1* is also a change to the contents of *arr[0]*. So the code

```
v1 = 3;
cout << arr[0] << endl;
```

now outputs

```
3
```

Another situation where structured binding can prove useful is where the elements of an array are themselves complex structures. For example, if we assume the following declaration in a program

```
int points[4][3] = { {1,2,3},{4,5,6},{7,8,9},{10,11,12} };
```

which gives us a four elements array each of which contains a three-element array (in other words, we have a 2D array). A nice way to access the contents would be to use lines such as

```
for (auto [a, b, c] : points)
   cout << a << "  " << b << "  " << c << endl;
```

This allows us to name the three elements in each "column" of the array.

Solutions

Activity 9.1

The structure is coded as:

```
struct ContactType
{
  char name[41];
  char emails[3][61];
  char mobile[16];
  char home[16];
};
```

with the variables being declared using the line

```
ContactType c1,c2,c3;
```

Activity 9.2

Initial code for *Records*:

```
#include <iostream>
using namespace std;

int main()
{
  struct StudentType
  {
    char name[31];
    char gender;
    int  age;
    int  marks[5];
  };

  StudentType fiction[3]
  {
    {"Pip",'M',17,{23,29,31,47,59}},
    {"Estella",'F',19,{31,38,42,53,78}},
    {"Clara",'F',20,{18,21,27,29,25}}
  };
}
```

Activity 9.3

Modified code for *Records*:

```
#include <iostream>
#include "iomanip"
using namespace std;

int main()
{
  struct StudentType
  {
    char name[31];
    char gender;
    int  age;
    int  marks[5];
  };

  StudentType fiction[3]
  {
    {"Pip",'M',17,{23,29,31,47,59}},
    {"Estella",'F',19,{31,38,42,53,78}},
    {"Clara",'F',20,{18,21,27,29,25}}
  };

  //*** Display students' details ***
  //** FOR each student DO **
  for (int student = 0; student <= 2; student++)
  {
    //** Display name, gender and age **
    cout << setw(20) << fiction[student].name << ' '
    ↳<< fiction[student].gender << ' ' <<
    ↳fiction[student].age;
    //** FOR exam mark DO **
    for(int exam=0; exam <= 4; exam++)
      //** Display mark **
      cout << setw(4)<<fiction[student].marks[exam];
    cout << endl;
  }
}
```

Activity 9.4

Modified code for *Records*:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  struct StudentType
  {
    char name[31];
    char gender;
    int  age;
    int  marks[5];
  };

  StudentType fiction[3]
  {
    {"Pip",'M',17,{23,29,31,47,59}},
    {"Estella",'F',19,{31,38,42,53,78}},
    {"Clara",'F',20,{18,21,27,29,25}}
  };
  //*** Display students' details ***
  //** FOR each student DO **
  for (StudentType v: fiction)
  {
    //** Display name, gender and age **
    cout << setw(20) << v.name << ' ' << v.gender
    ↳<< ' ' << v.age;
    //** FOR exam mark DO **
    for (int exam : v.marks)
      //** Display mark **
      cout << setw(4) << exam;
    cout << endl;
  }
}
```

Activity 9.5

Code for *GameScores*:

```
#include <iostream>
using namespace std;

int main()
{
  struct TimeType
  {
    int minutes;
    int seconds;
  };

  struct TopScoreType
  {
    char name[31];
    TimeType time;
  };

  // *** Stores top scores ***
  TopScoreType topscores[5];

  // *** Assign value to record ***
  strcpy_s(topscores[0].name,31,"Davy Jones");
  topscores[0].time.minutes = 3;
  topscores[0].time.seconds = 49;

  // *** Display contents of first record ***
  cout << topscores[0].name << " " <<
  ↳topscores[0].time.minutes << ":" <<
  ↳topscores[0].time.seconds << endl;
}
```

Activity 9.6

```
enum SeasonsType {Winter, Spring, Summer, Fall};
```

Activity 9.7

Modified code for *Enumeration* (using `enum struct`):

```
#include <iostream>
using namespace std;

// *** Set up enumerated type ***
enum struct DaysType {Sunday, Monday, Tuesday,
↳Wednesday, Thursday,Friday, Saturday};
```

```
int main()
{
   // *** Set up matching string array ***
   char DaysoftheWeek[7][10] {
   ↳"Sunday","Monday","Tuesday","Wednesday",
   ↳"Thursday","Friday","Saturday" };
   // *** Declare enumerated variable initialising to
   ↳Tuesday ***
   DaysType day = DaysType::Tuesday;

   // *** Increment day ***
   day = static_cast<DaysType>((static_cast<int>(
   ↳day)+1) % 7);

   // *** Display corresponding array element ***
   cout << DaysoftheWeek[static_cast<int>(day)] <<
   ↳endl;
}
```

Activity 9.8

Modified code for *Enumeration*:

```
#include <iostream>
using namespace std;

// *** Set up enumerated type ***
enum struct DaysType {Sunday, Monday, Tuesday,
↳Wednesday, Thursday,Friday, Saturday};

int main()
{
   // *** Set up matching string array ***
   char DaysoftheWeek[7][10] {
   ↳"Sunday","Monday","Tuesday","Wednesday",
   ↳"Thursday","Friday","Saturday" };
   // *** Declare enumerated variable initialising to
   ↳Tuesday ***
   DaysType day = DaysType::Friday;//or Saturday

   // *** Increment day ***
   day = static_cast<DaysType>((static_cast<int>(
   ↳day)+1) % 7);

   // *** Display corresponding array element ***
   cout << DaysoftheWeek[static_cast<int>(day)] <<
   ↳endl;
   //*** Display optional message ***
   if (day == DaysType::Saturday || day ==
   ↳DaysType::Sunday)
      cout << "It's the weekend\n";}
```

Activity 9.9

The largest number to be stored is 12. In binary, this is written as 1100, so 4 bits are required.

Activity 9.10

As a signed value, the left most bit, being 1, would indicate a negative value. Negative integer values are stored in 2's complement form. To convert this to decimal

10111110 2's complement
= 01000010 positive form
= 64 + 2
= 66

As an unsigned value, then its value is

128 + 32 + 16 + 8 + 4 + 2

= 190

Activity 9.11

No solution required.

Activity 9.12

In *subjectstaken*, bits that are set to 1 represent subjects that are being taken by the student; bits set to zero represent subjects not being taken. In variable *subjectsnottaken*, we need the exact inverse of the first variable, with a 1 where a subject is not taken and 0 where a subject is taken. This can be achieved by the line

```
subjectsnottaken = ~subjectstaken;
```

Activity 9.13

Code for *ShiftLeft*:

```
#include <iostream>
using namespace std;

//*** Use shift left to multiply by 16 ***
int main()
{
   short int no;      //value entered

   // *** Read in value ***
   cout << "Enter value between 1 and 100 :";
   cin >> no;
   while (no < 1 || no > 100)
   {
      cout << "Number must be in range 1 to 100
      ↳\nPlease re-enter :";
      cin >> no;
   }
   // *** Multiply by 16 ***
   short int result = no << 4;
   // *** Display result ***
   cout << no << " * 16 = " << result << endl;
}
```

Activity 9.14

Modified code for *ShiftLeft*:

```
#include <iostream>
using namespace std;

//*** Use shift left to multiply by 16 ***
int main()
{
   int short no;      //value entered

   // *** Read in value ***
   cout << "Enter value between 1 and 100 :";
   cin >> no;
   while (no < 1 || no > 100)
   {
      cout << "Number must be in range 1 to 100
      ↳\nPlease re - enter :";
      cin >> no;
   }
   // *** Shift bit 3 to position 15 ***
   short int result = no << 12;
   // *** Display result ***
   if (result < 0)
      cout << "Bit 3 is set to 1\n";
   else
      cout << "Bit 3 is set to 0\n";
}
```

Activity 9.15

Code for *Students*:

```
#include <iostream>
#include "iomanip"
#include "ctime"
using namespace std;

// *** Pupil record structure ***
struct PupilType
{
   char name[31];
   char group[11];
   unsigned short int subjects;
};

// *** Generaste and display pupil records ***
int main()
{
   // *** Names used for pupils ***
   char names[][31]=
   {
```

```
    "Newman Noggs","Wackford Squeers",
    ↳"Smike Digby","Mulberry Hawk",
    ↳"Vincent Crummles","Charles Cheerbyle",
    ↳"Tim Linkinwater","Madeliene Bray",
    ↳"Arthur Gride","Peg Silderskew",
    ↳"Matilda Price","John Browdie",
    ↳"Alfred Muntle","Susan Kenwigs",
    ↳"Morleena Kenwigs","Julia Witterly",
    ↳"Ninetta Crummles","Tommy Folair",
    "Henrietta Petowker","Walter Bray"
  };
  // *** Available subjects ***
  char topicnames[][15] = {"English","Math",
  ↳"History","Geography","Physics",
  ↳"Chemistry","Biology","Latin","French",
  ↳"German","Spanish","Mandarin","Japanese","IT",
  ↳"Modern Studies","Economics"};

  //*** Array of students' details ***
  PupilType students[20];

  // *** Set up names and group ***
  for (int stno = 0; stno <= 19 ; stno++)
  {
    strcpy_s(students[stno].name,31,names[stno]);
    strcpy_s(students[stno].group,11,"1A2014");
  }
  //*** Set up possible generated values ***
  int power[16]={0};
  //*** Seed generate number ***
  srand(static_cast<unsigned int>(time(0)));
  //*** Generate 6 unique values ***
  //*** FOR each student DO ***
  for( int stno = 0; stno <= 19; stno++)
  {
    //*** No topics selected so far ***
    int topicsselected = 0;
    //*** FOR 6 times DO
    for(int subj = 0 ; subj <= 5; subj++)
    {
      //*** Select new topic not previously
      ↳selected ***
      unsigned short int topic;
      do
        //*** Possible topic ***
        topic = 1 << rand()%16;
      while ((topicsselected & topic) == topic);
      //*** Add new topic to topics selected ***
      topicsselected += topic;
    }
    students[stno].subjects = topicsselected;
  }

  //*** Display details of each student ***
  for (int stno = 0; stno <= 19; stno++)
  {
       cout << setw(18) << students[stno].name <<
       ↳" " << students[stno].group << " ";
       for (int topicno = 0; topicno <= 15;
       ↳topicno++)
         if((students[stno].subjects &
         ↳(1 << topicno)) == (1 << topicno))
           cout << topicnames[topicno] << " ";
       cout << endl;
  }

  //*** Display names of those taking IT ***
  cout << "\nThe following students study IT :\n";
  for (int stno = 0; stno <= 19; stno++)
    if (students[stno].subjects & 0X2000)
      cout << students[stno].name << endl;
}
```

Activity 9.16

Code for *Encrypted*:

```
#include <iostream>
using namespace std;

//*** Encrypts/decrypts a string ***
int main()
{
  char original[31];   //Original string entered
  char encrypted[31]="";  //Encrypted string
  char decrypted[31]="";  //Decrypted string

  //*** Read message ***
  cout << "Enter text to be encrypted : ";
  gets(original);

  //*** Encrypt message ***
  for (int c = 0; c <= strlen(original); c++)
    encrypted[c] = original[c]^0X5A;

  //*** Decrypt message ***
  for (int c = 0; c <= strlen(original); c++)
    decrypted[c] = encrypted[c]^0X5A;

  //*** Display all three strings ***
  cout << "Original  : " << original << endl;
  cout << "Encrypted : " << encrypted << endl;
  cout << "Decrypted : " << decrypted << endl;
}
```

Activity 9.17

No solution required.

Activity 9.18

No solution required.

Activity 9.19

```
using Boolean = bool;
Boolean a = true, b = true;
```

Activity 9.20

Modified code for *TestPairAndTuple*:

```
#include <iostream>
using namespace std;

int main()
{
  pair pv = make_pair( "Answer",42.17 );
  cout << pv.first << endl;
  cout << pv.second << endl;
}
```

The change of value types held in the pair variable is handled automatically and the values are displayed as expected.

Activity 9.21

Modified code for *TestPairAndTuple*:

```
#include <iostream>
using namespace std;

int main()
{
  pair pv = make_pair( "Answer",42.17 );
  pair pv2 = make_pair("Wrong", 13.6);
  cout << "Before swap\n";
  cout << "pv = (" << pv.first << "  " << pv.second
  ↳<< ")\n";
  cout << "pv2 = (" << pv2.first << "  "
  ↳<< pv2.second << ")\n";
  swap(pv,pv2);
  cout << "\n\nAfter swap\n";
  cout << "pv = (" << pv.first << "  " << pv.second
  ↳<< ")\n";
  cout << "pv2 = (" << pv2.first << "  " <<
  ↳pv2.second << ")\n";
}
```

Activity 9.22

Modified code for *TestPairAndTuple*:

```
#include <iostream>
using namespace std;

int main()
{
  pair pv = make_pair( "Answer",42.17 );
  pair pv2 = make_pair("Wrong", 13.6);
  pair pv3 = make_pair(12, 12);
  cout << "Before swap\n";
```

```
  cout << "pv = (" << pv.first << "  " << pv.second
  ↳<< ")\n";
  cout << "pv2 = (" << pv2.first << "  " <<
  ↳pv2.second << ")\n";
  swap(pv,pv2);
  cout << "\n\nAfter swap\n";
  cout << "pv = (" << get<0>(pv) << "  " <<
  ↳get<1>(pv) << ")\n";
  cout << "pv2 = (" << get<const char*>(pv2)<< "  "
  ↳<< get<double>(pv2) << ")\n";
}
```

Activity 9.23

Modified code for *TestPairAndTuple*:

```
#include <iostream>
using namespace std;

int main()
{
  tuple<int, char, double, double>tv = make_tuple(
  ↳2, 'c', 9.2, 3.6);
  cout << "tv = (" << get<int>(tv) << ',' <<
  ↳get<char>(tv) << ',' << get<2>(tv) << ',' <<
  ↳get< 3 > (tv) << ")\n";
}
```

Activity 9.24

Modified code for *TestPairAndTuple*:

```
#include <iostream>
using namespace std;

int main()
{
  tuple<int, char, double, double>tv = make_tuple(
  ↳2, 'c', 9.2, 3.6);
  tuple tv2 = make_tuple("Arrow", 23);
  tuple tv3 = tuple_cat(tv, tv2);
  cout << "tv3 = (" << get<0>(tv3) << ',' <<
  ↳get<1>(tv3) << ',' << get<2>(tv3) << ',' <<
  ↳get<3>(tv3) << ',' << get<4>(tv3) << ',' <<
  ↳get<5>(tv3) << ")\n";
}
```

Activity 9.25

Modified code for *TestPairAndTuple*:

```
#include <iostream>
using namespace std;

int main()
{
  int no;
  double dno;
  tuple trf{ 97,2.8 };
  tie(no, dno) = trf;
  cout << "no = "<< no << " dno = " << dno << endl;
  cout << "trf = ("<< get<0>(trf) << ',' <<
  ↳get<1>(trf) << ")\n";
  no = 22;
  dno = 5.9;
  cout << "no = " << no << " dno = " << dno << endl;
  cout << "trf = (" << get<0>(trf) << ',' <<
  ↳get<1>(trf) << ")\n";
}
```

The variable have the same values as the elements of the tuple.

When the contents of the variables are changed, the tuple is unaffected.

Activity 9.26

Modified code for *TestPairAndTuple*:

```
#include <iostream>
using namespace std;

int main()
{
  tuple trf{ 97,2.8 };
  auto[no,dno] = trf;
  cout << "no = "<< no << " dno = " << dno << endl;
  cout << "trf = ("<< get<0>(trf) << ',' <<
  ↳get<1>(trf) << ")\n";
  no = 22;
  dno = 5.9;
  cout << "no = " << no << " dno = " << dno << endl;
  cout << "trf = (" << get<0>(trf) << ',' <<
  ↳get<1>(trf) << ")\n";
}
```

The results are identical to the previous version of the program.

10

Pointers and References

In this Chapter:

- ❑ The *Address Of* Operator
- ❑ Raw Pointer Types
- ❑ Pointers to existing Variables
- ❑ Pointers and Constants
- ❑ Pointers and Arrays
- ❑ Arrays of Pointers
- ❑ Pointers, Structs and Unions
- ❑ Pointers, Characters and Strings
- ❑ Pointer Arithmetic
- ❑ Pointers and Dynamic Space Allocation
- ❑ *void* Pointers
- ❑ Smart Pointer Types
- ❑ Reference Variables
- ❑ Pointers, References and Aliases

Raw Pointers

Introduction

When we define a variable, it is allocated one or more bytes in the computer's memory. This means that each variable is associated with a unique address which gives the location of the byte(s) allocated to that variable. For example, when a program executes the line

```
char letter;
```

this might result in the byte at memory location $25FBF7_{16}$ being allocated to the variable *letter*.

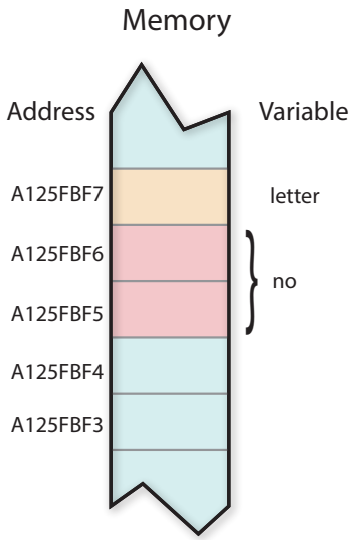
When the variable is a type which is allocated more than a single byte, the associated address is that of the first byte allocated. A theoretical allocation resulting from the statements

```
char letter;
short int no;
```

is shown in FIG-10.1.

FIG-10.1

Variables and Addresses



Since each variable's address is associated with the first (lowest address) assigned to it, the address of *letter* is $A125FBF7_{16}$ and the address of *no* is $A125FBF5_{16}$.

There is no way of knowing in advance the address that will be assigned to a variable, and in any case it is likely to be different every time the program is executed, but we can gain access to that address once the program is running.

Address Of Operator (&)

The address given to a variable can be determined using the *address of* operator, `&`. For example, while

```
cout << no << endl;
```

will output the contents of the variable *no,* the line

```
cout << &no << endl;
```

will output the variable's address.

In practice we are unlikely to ever want to display the address of a variable, but accessing that address can be a powerful tool and is widely used throughout C++.

Activity 10.1

Start a new project called *AddressOf*.

Create an `int` variable called *num* containing the value 21.

Display the contents and address of *num*.

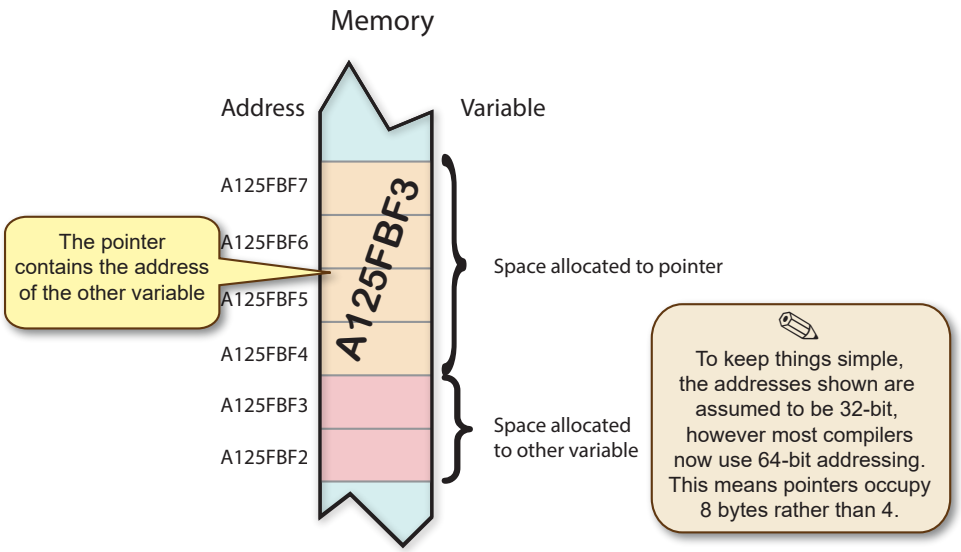
Pointer Variables

What is a Pointer?

When we google a term, the resulting display lists a set of hypertext links to other web pages. These links contain the address of a web page, and when a link is clicked, that address is used to give us access to the specified page. A **pointer** is similar in concept: it contains the address of another data item held within a program and using that address gives us access to the data located at that address (see FIG-10.2).

FIG-10.2

Pointers Contain Addresses



The term **pointer** is used to show that the variable "points to" or **references** a memory location. The pointers defined in this section have to a great extent been replaced in later versions of C++ (from C++11 onwards) by a more sophisticated pointer type (described in the next section). To differentiate between these two types of pointers, the original pointer types discussed here are known as **raw pointers** while the later ones are known as **smart pointers**.

Pointers have their own unique set of operations and the compiler treats pointers differently from other types of variables.

Declaring Pointers

Like any other program variable, a pointer must be declared. To identify a variable as a pointer, the basic data type is followed by an asterisk and then the pointer's name. For example,

```
short int *shptr;
```

The type at the start of the declaration tells the compiler what sort of value to expect at the location whose address is held within the pointer. Hence, we can read the above definition as:

> *shptr* is a pointer designed to contain an address where a `short int` value can be found.

Activity 10.2

Write the definitions necessary to create an `int` pointer called *iptr* and a `float` pointer called *fptr*.

If we want to declare multiple pointers, we need to place an asterisk beside each identifier as in the line:

```
int *ptr1, *ptr2, *ptr3;
```

If we were to omit the second and third asterisks in the line above, *ptr2* and *ptr3* would be created as standard `int` variables, not pointers.

Of course, the best practice is to declare each pointer on separate lines:

```
int *ptr1;
int *ptr2;
int *ptr3;
```

Initialising Pointers

It's not a good idea to create pointers without also initialising them. Although this initial value may be the address of a standard variable as in the code

```
int num;
int * ptr{&num};
```

it is much more likely that we'll want to start with the pointer containing no address at all (the value assigned in this case is called **null** and the pointer variable is known as a **null pointer**). We can do this in several ways. The first, and preferred way, is to use the keyword `nullptr` as in the line

```
int *ptr{ nullptr};
```

We can achieve the same result using empty braces:

```
int *ptr{};
```

or we can have C++ convert the integer value zero to a null value:

```
int *ptr{0};
```

A final option which was used in earlier versions of C++ is to use the term `NULL`:

```
int *ptr{NULL};
```

The graphic representation of a null pointer is shown in FIG-10.3.

FIG-10.3

A Null Pointer



Pointers and *auto*

We can also use `auto` when declaring a pointer variable. For example,

```
int num;          //Creates an int variable
auto ptr {&num};  //Creates an int pointer variable
```

However, although the above lines are perfectly valid, to make it clearer to anyone reading the code at a later date, the preferred approach is to include an asterisk after `auto`:

```
auto* ptr {&num};
```

Both versions of declaring *ptr* have exactly the same effect.

Using a Pointer

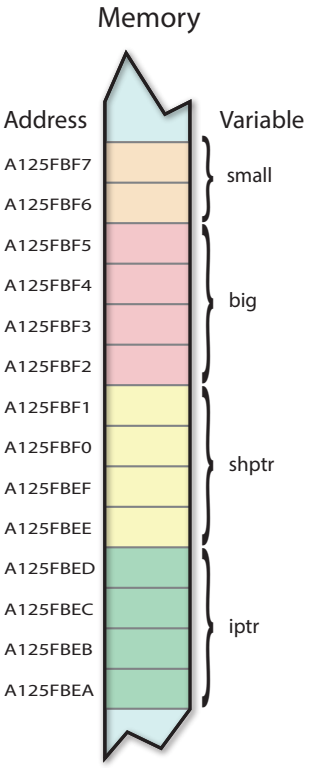
If we assume that a program begins with the following declarations

```
short int   small;
long int    big;
short       *shptr{nullptr};
long int    *lptr{nullptr};
```

then the memory allocated by the compiler might be represented by the diagram given in FIG-10.4.

FIG-10.4

How Variables are Allocated Space



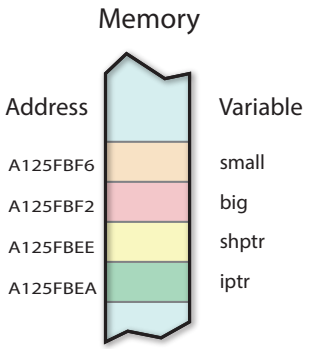
Activity 10.3

For each of the four variables shown in FIG-10.4, list the address of that variable.

For simplicity, we can reduce the previous diagram to that shown in FIG-10.5.

FIG-10.5

Simplified Allocation Diagram



While standard program assignment statements such as

```
small = 12;
big = 65;
```

result in the specified values being stored in the space allocated to variables *small* and *big*, the pointer variable must be assigned addresses with statements such as

```
shptr = &small;
iptr = &big;
```

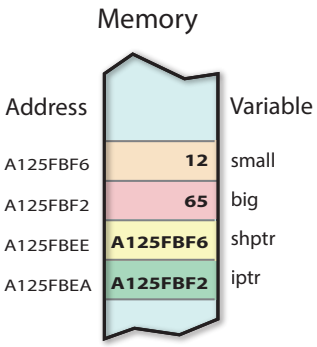
Note that a pointer can only contain the address of a variable of the type for which it is designed. Hence, we would be presented with a compilation error if we were to try an assignment such as

```
shptr = &big;
```

The result of the four valid statements given above on memory contents is shown in FIG-10.6.

FIG-10.6

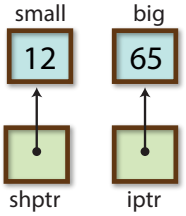
Values Assigned to Variables



Since the actual addresses used to store variables are not normally known and will vary between each execution of a program, the variable memory space diagram is usually simplified to exclude actual addresses. Arrowed lines are then used to show where a pointer variable is referencing. Using this simplified version, the diagram in FIG-10.5 changes to that shown in FIG-10.7.

FIG-10.7

A Pointer Reference Diagram



Dereferencing a Variable

Once a pointer contains a valid address, it can be used to access the data at that address (just as a hypertext link is used to access a web page). This operation is known as **dereferencing** the pointer.

To dereference a pointer, we use the pointer name preceded by an asterisk. Hence, the statement

```
cout << *iptr;
```

is an instruction to display the contents held at the address referenced by the pointer *iptr*. In our example, this will result in the value 65 being shown on the screen. In this context, the asterisk is known as the **indirection operator**.

A dereferenced pointer may be used in any situation where it would be valid to use a normal variable of the type referenced by the pointer. This means a statement such as

```
small = 21;
```

can be replaced by

```
*shptr = 21;
```

and yet produce exactly the same effect.

FIG-10.8 gives several other examples of normal variable statements and their dereferenced equivalent.

FIG-10.8

Equivalent Pointer Statements

Standard Commands	Pointer Equivalent
`big++;`	`*iptr++;`
`cin >> small;`	`cin >> *shptr;`
`if(small == 14)`	`if(*shptr == 14)`
`cout << 10 - big;`	`cout << 10 - *iptr;`
`for(big=0;big<5;big++)`	`for(*iptr=0;*iptr<5;*iptr++)`

There are situations where attempting dereferencing can confuse the compiler. For example, if the expression

```
12/big
```

is replaced by

```
12/*iptr
```

the compiler will assume the symbols `/*` mark the start of a comment and the program will suddenly produce a glut of new errors. This problem is overcome by enclosing the pointer dereferencing in parentheses or placing a space between the two characters:

```
12/(*iptr)
```

or

```
12/ *iptr
```

Other combinations may confuse anyone examining our code rather than the compiler. For example, while the meaning of

```
cout << 5*big << endl;
```

is obvious, the line

```
cout << 5**iptr << endl;
```

while acceptable to the compiler, may make a human somewhat confused. Again the solution is to add parentheses or spaces:

```
cout << 5*(*iptr) << endl;
```

or

```
cout << 5 *   *iptr << endl;
```

Activity 10.4

By adding three pointer variables, convert the executable part (starting at the first `cout` statement) of the following code to use dereferencing when accessing variables.

```
#include <iostream>
using namespace std;
int main()
{
    int no1, no2, ans;
    cout << "Enter two numbers : ";
    cin >> no1 >> no2;
    if (no1 < no2)
        ans = no1;
    else
        ans = no2;
    cout << "Smaller value is " << ans << endl;
}
```

Copying the Contents of a Pointer

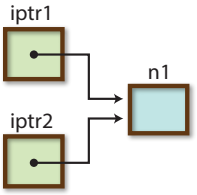
As with standard variables, we may copy the contents of one pointer to another as long as both pointers reference the same data type. For example, the code

```
int n1, *iptr1, *iptr2;
iptr1 = &n1;
iptr2 = iptr1;
```

results in the contents of *iptr1* being copied to *iptr2*, meaning both pointers contain the address of *n1*. This result is visualised in FIG-10.9.

FIG-10.9

Pointers Referencing the Same Address



Remember we need to use the pointer name without the indirection operator if we wish to access the contents of the pointer variable itself and to include the indirection operator when we need to access the data held at the location referenced by the pointer:

`iptr` - accesses the pointer's contents
`*iptr` - accesses the contents of the location referenced by the pointer

Activity 10.5

Assuming a program uses two pointer variables *iptr1* and *iptr2*, write the Boolean expression needed to

a) check to see if the two pointers contain the same address
b) check if the two locations referenced by the pointers contain the same value.

Pointers to Constants

We already know that we can create named data constants within a program. Typical constant declarations would be

```
const double PI {3.14159265};
const double E {2.7183};
```

It might be reasonable to assume that, should we so desire, we might create a pointer to reference one of these values with a statement such as

```
double *dptr {&PI};
```

However, such a statement would not compile. The reason for this is that while the pointer is designed to point to a `double` value, PI is a `const double`, which the compiler takes to be a different type from a standard `double` since the value at that location cannot be changed. So, if we want a pointer to reference *PI*, that pointer must be declared as type `const double`:

```
const double *dptr {&PI};
```

Since *dptr* references a constant, any attempt to change the value at the referenced location with a line such as

```
*dptr = 3.14;
```

would be flagged as an error.

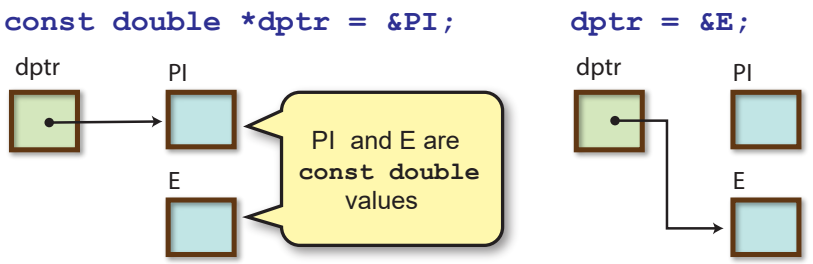
However, the content of the pointer itself is not a constant, which means that elsewhere in the program code it can be made to reference another constant of the same type:

```
dptr = &E;
```

(see FIG-10.10).

FIG-10.10

Changing the contents of a `const double` Pointer



Constant Pointers

If necessary, the address held within a pointer can be fixed without the possibility of it being changed later in the code. This is achieved by defining the pointer as a

constant. For example, to create a constant pointer to a standard Boolean variable we could use the lines

```
bool over21 {true};
bool* const bptr {&over21};
```

Notice that in this case the term `const` comes after the basic type (`bool`) when declaring the pointer. This time, although the address held in the pointer cannot be changed, the contents of the area referenced can. Hence, it is acceptable to write

```
*bptr = false;
```

to change the contents of *over21*.

Constant Pointers to Constants

The final possibility is to define a constant pointer to a constant value. For example, the code

```
const double PI {3.1415265};
const double *const dptr {&PI};
```

not only creates a pointer referencing a value which cannot be changed but also a pointer whose contents cannot be changed. Notice how the term `const` is included both before and after the underlying type (`double`).

Activity 10.6

Write definitions for the following:

a) A pointer, *fptr,* to the `float` variable, *x*.
b) A pointer, *iptr,* to the `int` named constant, *MAX*.
c) A constant pointer, *dptr*, to the `double` named constant, *TAX*.

A Pointer to a Pointer

If we can create a pointer to any type of variable, it follows that we should be able to create a pointer to a pointer. Let's start by creating an ordinary integer variable

```
int no {7};
```

and a pointer to that variable:

```
int *iptr {&no};
```

Now we want a pointer to *iptr*. This second pointer is not a pointer to an integer, but a pointer to an integer pointer and must be declared as such:

```
int*  *pptr;
```

The line starts with the type, `int*` - integer pointer type - then the fact that we are creating a pointer (`*`) and, finally, the name of the pointer, *pptr*.

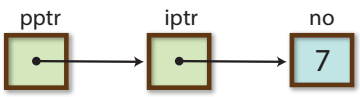
All we need to do now is expand the line so that our new pointer contains the address of the first pointer:

```
int* *pptr {&iptr};
```

This gives us the setup shown in FIG-10.11.

FIG-10.11

A Pointer to a Pointer



Dereferencing *iptr* (`*iptr`) gives us access to the value 7 in *no*; dereferencing *pptr* (`*pptr`) gives us access to the address in *iptr*. If we wish to access the value 7 in *no* using *pptr,* then it must be **double dereferenced** which we can do using the term

```
**pptr
```

so the line

```
cout << **pptr << endl;
```

would output the value 7 to the screen.

Activity 10.7

Create a new project called *UsingPointers*. The program should contain a standard `float` variable, *fno*, a pointer to a `float`, *fptr*, and a pointer to a `float` pointer, *pfptr*. Assign the address of *fno* to *fptr* and assign the address of *fptr* to *pfptr*.

Using only *pfptr*, read in a value to be stored in *fno* and then display the contents of *fno*.

And, before you ask, yes, we can have a pointer to a pointer to a pointer! Luckily, there's not much need to create such a variable.

Pointers and Arrays

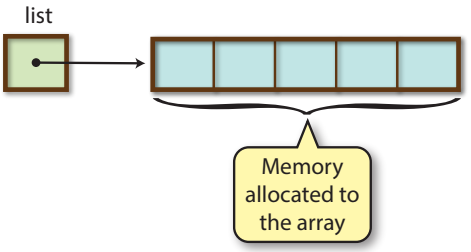
When we create an array, the name given to the array is automatically treated in a similar fashion to a constant pointer. Although there are a few circumstances when this similarity breaks down, these are rarely met in practice. Therefore, we can visualise the result of executing the line

```
int list[5];
```

as shown in FIG-10.12.

FIG-10.12

An Array's Pointer



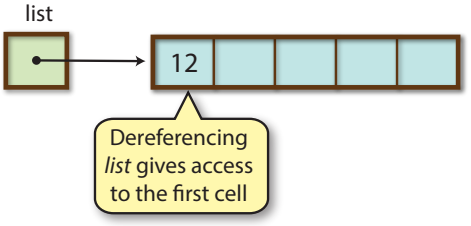
We already know that dereferencing a pointer gives us access to the location it references. But how does this work in the case of an array? Dereferencing a pointer to an array will access the first element of that array. So if we were to write

```
*list = 12;
```

the result would be that shown in FIG-10.13.

FIG-10.13

Using the Dereference Operator on an Array



But how do we access the other elements of the array using the pointer? Again, this is something we already know. When we covered arrays in an earlier chapter we saw that accessing elements of the array required the array name followed by the element number enclosed in brackets. To access the second element in the array we used the term

```
list[1]
```

But now that we know *list* has been set up as a constant pointer, then the expression above must somehow dereference the pointer and then adjust the actual address being accessed.

The address adjustment is calculated using the value within the brackets (given after the array name). So the term

```
list[0]
```

means *get the address given in* list*, make no adjustment, and access that location.*

```
list[1]
```

means *get the address in* list*, adjust the address by 1 element, and access that location.*

Of course, since `int` values occupy four bytes, adjusting the address by one element means calculating the new address as the address in *list + 4* .

This gives us access to the value held in the second element of the array. Being a constant pointer, the address held in *list* is not changed by these operations, the calculated address being held in a temporary storage area.

Activity 10.8

Assuming the declaration `short int nums[8];` results in *nums[0]* being stored at memory location 25FD00, what addresses would be accessed using the following terms?

a) `*nums`
b) `nums[0]`
c) `nums[3]`
d) `nums[7]`

If we create a standard `int` pointer with the line

```
int *iptr {};
```

that pointer is allowed to contain the same address as an `int` array pointer, so we can write

```
int list[5];
iptr = list;
```

The terms *iptr* and *list* are now interchangeable. We can access the array elements using terms such as

```
list[2]
```
or
```
iptr[2]
```

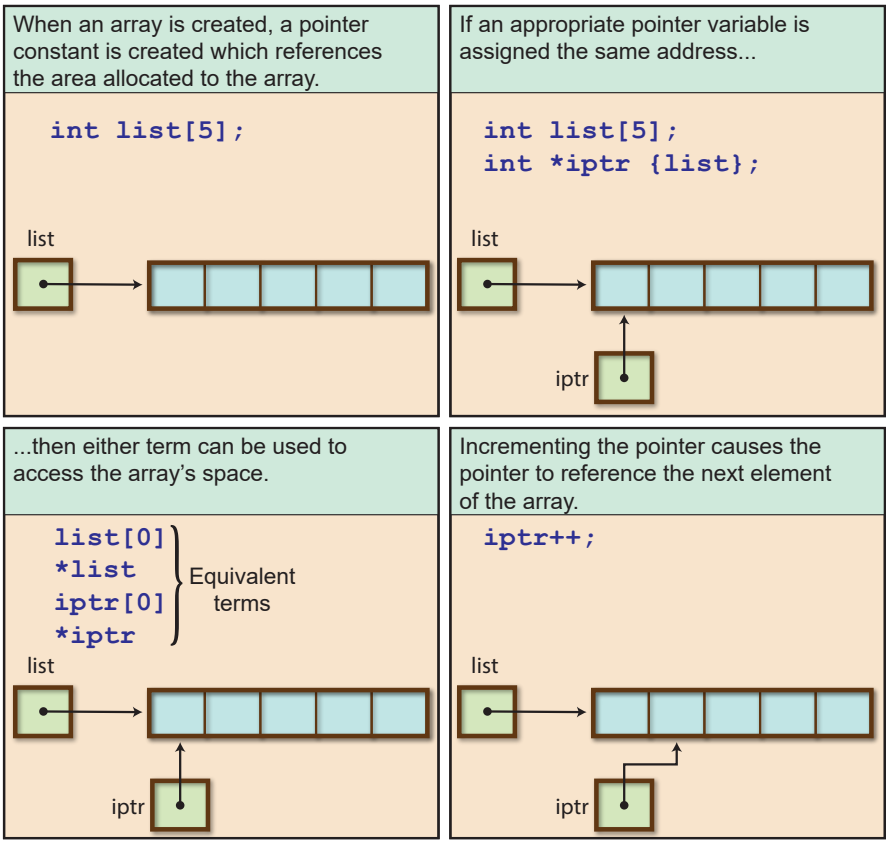
But, since *iptr* is not a pointer constant, we can change the address it holds. To have the pointer contain the address of the second element in the array we can use the term

```
iptr++;
```

This increments the pointer by a number determined by the size of the elements within the array. The result of each operation is shown in FIG-10.14.

FIG-10.14

Using a Standard Pointer to Access an Array



To demonstrate the use of this technique, the program in FIG-10.15 makes use of a pointer to display the values held within an array.

FIG-10.15

Pointers and Arrays

```
#include <iostream>
using namespace std;

// *** Access an array through a pointer variable ***
int main()
{
   int list[]{2,7,1,9,12};

   //*** Make pointer reference array ***
   int *iptr {list};

   //*** Display array contents by incrementing pointer ***
   for(int c = 0; c < 5; c++)
      cout << *iptr++ << endl;
}
```

Notice that the term

```
*iptr++
```

used in the program above increments the contents of the pointer and not the contents of the location referenced by the pointer. This is because the increment operator (++)

has a higher priority than the indirection operator (`*`). To increment the contents of the referenced area, we need to use parentheses to force the dereferencing to occur first:

```
(*iptr)++
```

Activity 10.9

Start a new project called *ArrayPointers* and implement the code shown in FIG-10.15.

Activity 10.10

Using the pointer *iptr*, modify *ArrayPointers* so that the contents of each array cell are incremented by 1 before the contents are displayed.

Another option is to access subsequent elements without changing the contents of *iptr* with a expression such as

```
*(iptr + 1)
```

So, if *iptr* references the first element of the array *list*, then

```
*(iptr + 1) = 23;
```

will store the value 23 in the second element of the array without changing the contents of the pointer *iptr*. Of course, this is just a different way of writing

```
iptr[1] = 23;
```

Activity 10.11

Modify *ArrayPointers* so that, after incrementing each value in *list*, *iptr* is reset to reference the start address of *list* and then displays the contents of the array using the expression `*(iptr + c)` in a `for` loop (which uses *c* as its counter).

Unfortunately, we cannot use a pointer to an array in exactly the same circumstances in which we can use the array itself. For example, if our array is *list* and *iptr* references that array, while the expression

```
size(list)
```

is valid,

```
size(iptr)     //Invalid expression
```

is not.

And although we can use the range-based `for` loop on the array:

```
for(int v: list)
```

we cannot do the same with the pointer:

```
for(int v: iptr)  //Invalid
```

The reason for this is that the array name, although we can treat it as a pointer, actually gives access to more detail about the array such as the number of elements it contains. It is this information that the `size()` function and the range-based `for` structure require in order to operate.

Array Element Addresses

Although the address of the first cell within an array can be obtained from the name of the array, to access the address of other cells in that array, we need to make use of the *address of* operator (`&`).

Having created an array with the line

```
float nums[10];
```

we can get the address of the second and third cells using the terms

```
&nums[1]    and    &nums[2]
```

The first cell's address can be obtained in the same way:

```
&nums[0]
```

This gives us the option to use either

```
nums
```

or `&nums[0]`

when accessing the start address of the array.

If we wanted to have a pointer reference the third element of the array *list* (as used earlier), then we would write:

```
iptr = &list[2];
```

Arrays of Pointers

Just as we can create an array of integers, reals, characters or Boolean values, so we can create an array of pointers. For example, the line

```
int *addr[5];
```

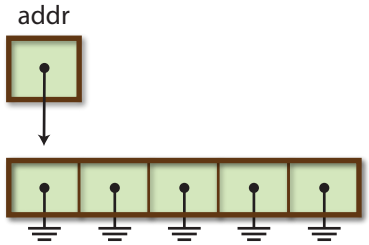
would create an array, *addr*, of five pointers. We can even initialise the array, setting all the pointers to null by extending the line to read:

```
int *addr[5]={nullptr};
```

(see FIG-10.16).

FIG-10.16

An Array of Pointers



We'll have more to say about arrays of pointers later.

Pointers and Multi-Dimensional Arrays

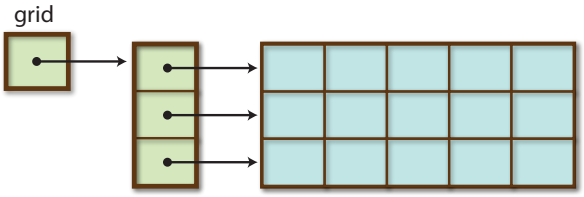
When an array has more than one dimension, then multiple pointers are automatically created. So, if a program contains the line

```
int grid[3][5];
```

we not only get an array which is 3 rows by 5 columns, but we also get an accompanying set of pointers (see FIG-10.17).

FIG-10.17

How Pointers are Used in a 2D Array



From the diagram we can see that *grid* is a pointer to an array of three pointers, one for each row of the `int` array.

We already know that we can get at the contents of an array through dereferencing its pointer, which can be done in two ways:

using the indirection operator:	`*grid`
or by using brackets:	`grid[index value goes here]`

So, using the second option, the terms

```
grid[0]
grid[1]
grid[2]
```

will give us access to the three pointers in the pointer array. Dereferencing these pointers will give us access to `int` data area. We have the option to use

```
*grid[0]
```

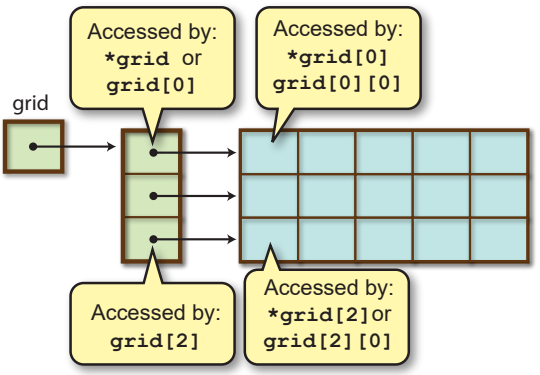
to access the first element of the first row, but a better method is to use the brackets:

```
grid[0][0]
```

(see FIG-10.18).

FIG-10.18

Pointer Referencing of a 2D Array



And suddenly we have arrived at exactly the same syntax as we used in the earlier chapter on arrays. But, this time, we know exactly what mechanism is being employed to allow us to access the cells of the `int` array.

Pointers, Structs and Unions

Pointers to Structs

We can create a pointer to a `struct` record variable in a similar manner to that for any other type. First declare the structure itself

```
struct DateType
{
```

```
    int day;
    int month;
    int year;
};
```

then create a variable of that type

```
DateType now;
```

and a pointer to that type

```
DateType *dtptr { &now };
```

Accessing the structure's fields via the pointer needs a little work, Using an expression such as

```
*dtptr.day
```

won't work because the direct member selector operator (.) has a higher priority than the indirection operator (`*`). We can get round this using parentheses

```
(*dtptr).day
```

but since this seems a little clumsy, an alternative method of accessing a field via a pointer is to use the **indirect member selector operator** (`->`) created from the *minus* and *greater than* characters:

```
dtptr->day
```

If the address of a field within a record structure is required then the terms

```
&now.day
```

or

```
&dtptr->day
```

can be used.

Activity 10.12

Start a new project, *StructPointer*, and, using the *DateType* structure defined above, create *now* as a variable of that type and *dtptr* as a pointer to that variable.

Use the pointer and `scanf()` and `printf()` to read in and display a date. There is no need to check if the date is a valid one.

We can also create a pointer to a field within a record. Just create a pointer as the same type as the field and then reference the field. So, assuming we want to point to the *month* field in *now*, we would write

```
int *iptr { &now.month };
```

However, we cannot create a pointer to a bit field within a structure.

Unions

Pointers to unions can be set up and used in the same way as those for a record structure. For example,

```
union NumbersType
{
    int num;
```

```
    short nos[2];
};

NumbersType ex;
NumbersType *ntptr {&ex};

ntptr->num = 0X0FFF0EFF;
```

Pointers, Chars and Strings

Pointer to a *char* Variable

We can create a pointer to a `char` variable in much the same way as we did for `int` and `float` type values. For example,

```
char ch {'A'};
char *cptr {&ch};
cout << *cptr << endl;
```

will store the address of the `char` variable *ch* in the `char` pointer *cptr* and then dereference that pointer to display the contents of *ch*, creating the output

```
A
```

Pointer to a *char* Array

When we create a C-style string within a `char` array, then the pointer must hold the start address of that array:

```
char message[]{ "Hello world" };
char *cptr { message };
```

The important point to remember here is that an array name, used without a subscript, acts like a pointer to the start address of the data held within that array so there is no need to use the `&` operator to assign the address of the array to our pointer.

Also, when we output the string held within a `char` array, we know that `cout` takes only the array name as in the line:

```
cout << message << endl;
```

So it should come as no surprise that when using the pointer to output the message, there is no need to dereference it:

```
cout << cptr << endl;
```

If we were to dereference the pointer as in the line

```
cout << *cptr << endl;
```

then only the first character in the string would be displayed:

```
H
```

Pointer to a String Constant

Although we can't directly create a pointer to a `char` constant such as 'A', we can create a pointer to a string constant such as "*Hello world*". However, if we were to attempt this with the line

```
char *cptr {"Hello"};
```

the compiler would throw up an error. The reason for this is that we are attempting to have a standard `char` pointer reference a string constant (remember the contents of

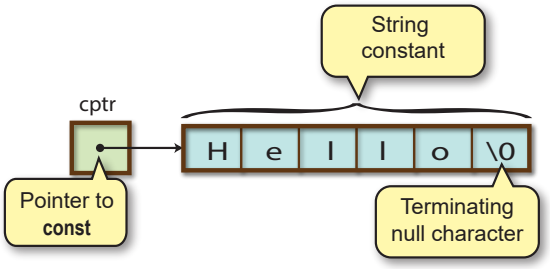
a string constant cannot be changed). To solve our problem, we need to declare *cptr* as a pointer to a string constant:

```
const char *cptr {"Hello"};
```

The effect of this statement is shown in FIG-10.19.

FIG-10.19

A **const char** Pointer Referencing a String Constant



As with the pointer to the array in a previous example, dereferencing the pointer will result in the first character in the string, *H*, being displayed.

If we wish to store a sequence of strings, rather than create a 2D **char** array with code such as

```
char names[][20]{ "Maggie Tulliver", "Tom Tulliver",
↳"Elizabeth Tulliver","Jeremy Tulliver","Lucy Dean" };
```

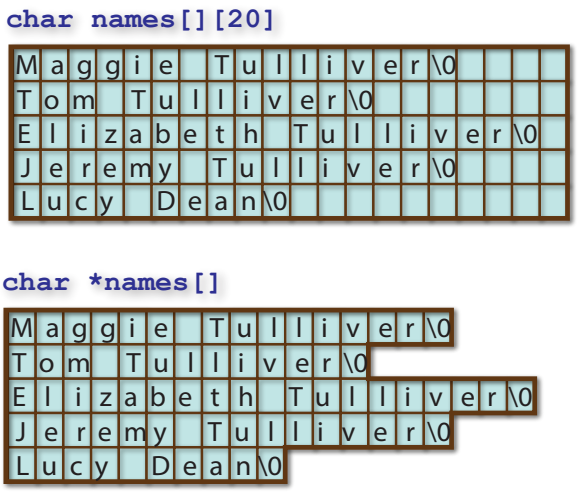
we could make use of an array of pointers:

```
const char *names[]{ "Maggie Tulliver", "Tom Tulliver",
↳"Elizabeth Tulliver","Jeremy Tulliver","Lucy Dean" };
```

The advantage of the second approach is that each name occupies only the number of bytes required rather than a fixed number of bytes as specified in the 2D array (see FIG-10.20).

FIG-10.20

Optimising Memory Using an Array of *char* Pointers



In both cases accessing a name uses exactly the same syntax. For example, *Tom Tulliver* is accessed using the term

```
names[1]
```

On the other hand, using a 2D array allows us to modify individual characters within the existing strings. This can be seen in the code

```
char names[][20]{ "Maggie Tulliver", "Tom Tulliver",
↳"Elizabeth Tulliver","Jeremy Tulliver","Lucy Dean" };

names[0][7] = 'G';
```

which changes the first name to

Maggie Gulliver

Such a modification of the string's contents is not possible when using pointers since the pointer has to be defined as a pointer to a constant.

Pointer Arithmetic

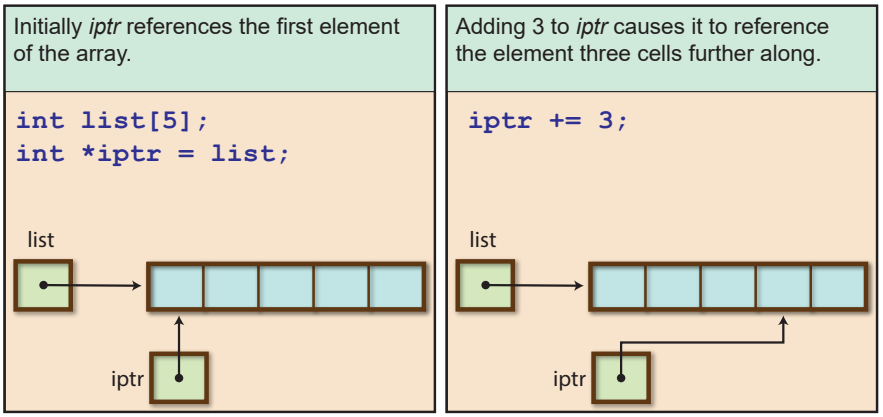
We've seen that we can increment an array pointer using the `++` operator, but we can also adjust the pointer by larger amounts with a statement such as

```
iptr += 3;
```

which would change the pointer's contents to the address of the element three positions along from the current location (see FIG-10.21).

FIG-10.21

Using a Standard Pointer to Access an Array



Similarly, a pointer can be decremented with statements such as

```
iptr--;
```

and

```
iptr -= 3;
```

We can calculate a new address without actually changing the contents of a pointer using an expression such as

```
iptr+2
```

For example, the code

```
int list[5];
int *iptr {list};
*(iptr+2) = 20;
```

would store the value 20 in *list[2]*. Note that expression `iptr+2` must be enclosed in parentheses because the dereference operator (`*`) has a higher priority than the addition operator (`+`).

Since the array name can itself be treated as a pointer, we can use it in place of the pointer name. Hence,

```
*(list+2) = 20;
```

has exactly the same effect as the previous example and is an alternative way of writing

```
list[2] = 20;
```

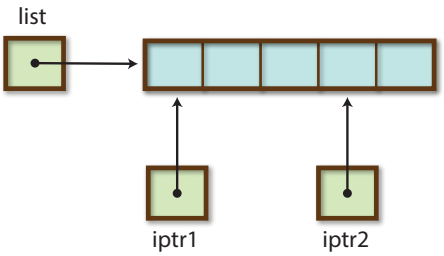
It is also possible to subtract the contents of one pointer from another. The effect of this is best explained with an example. The lines

```
int list [10];
int *ptr1=list, *ptr2=list;
ptr2 += 3;
```

create the setup shown in FIG-10.22.

FIG-10.22

Standard Pointers Referencing Different Parts of an Array



When we subtract the contents of one pointer from another, the result is not an address, but an integer. That integer represents the number of data elements between the two addresses. Hence, assuming the previous code has already been executed, the line

```
int result = iptr2 - iptr1;
```

would store the value 3 in *result* since the addresses in *iptr2* and *iptr1* are three elements apart.

Pointers and Dynamic Allocation



The method of reserving dynamic space described in the next few pages has largely been superseded by the use of smart pointers but can often be seen in older code.

For the most part, all the examples we have seen using pointers could be performed more easily with standard variables. However, one of the main uses of pointers is to be able to reserve data space as and when required during the execution of a program. This is known as **dynamic memory allocation**.

Imagine we wish to store the names of everyone who logs into a computer program. We could store that information in an array of strings. The problem is that we have no way of knowing how large to make the array. Should there be room for 100 names or 100,000 names? If we make the array too small, then the program will either crash or fail to store all the names; if we make the array too large, we waste space.

Using pointers and dynamic memory allocation, we can reserve exactly the correct amount required no matter how many names are entered (assuming we have sufficient main memory).

The *new* Statement

Once we have created the required type of pointer with a statement such as

```
int *iptr{nullptr};
```

we can use the `new` command to allocate space for a data value of the type specified in the pointer declaration. In this case we want to create space in which we can store an `int` value.

Not only does the **new** statement allocate space, but it also returns the address of the space allocated. This address can be assigned to a pointer. So to have our *iptr* pointer reference the newly allocated space we would write

```
iptr = new int;
```

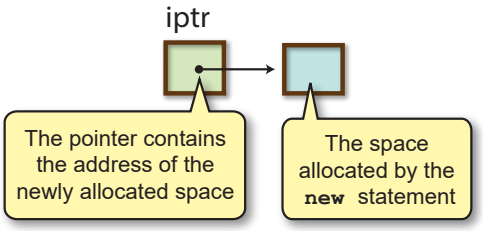
We have the option to perform the pointer declaration and the assignment in a single statement:

```
int *iptr { new int };
```

The result of such a statement is visualised in FIG-10.23.

FIG-10.23

Pointer to a Dynamically Allocated Memory



Notice that the allocated space does not have a name the way normal variables do. The only way to access the space is by dereferencing the pointer.

The allocated space will initially contain an undefined value, but we have the option to specify an initial value when using new. For example, the line

```
int *iptr {new int(2)};
```

will store the value 2 in the newly allocated space.

The program in FIG-10.24 demonstrates the use of the dynamic allocation.

FIG-10.24

Using Dynamic Memory Allocation

```
#include <iostream>
using namespace std;

int main()
{
   // *** Define pointer, and allocate space ***
   int *iptr{ new int };

   // *** Store value in allocated space ***
   *iptr = 12;

   // *** display value stored ***
   cout << *iptr << endl;
}
```

Activity 10.13

Start a new project called *DynamicSpace* and implement the code in FIG-10.24. Modify the program so that the allocated space is initialised to 12 when created.

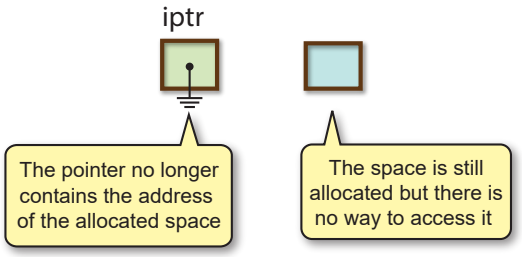
Since the pointer is needed to access the data space created using **new**, it is important that we do not modify the contents of that pointer. If we were to change the value held in the pointer with a statement such as

```
iptr = nullptr;
```

then the link between the pointer and data space would be lost with no way of recovering that link (see FIG-10.25).

FIG-10.25

Unreachable Data Space



We can use `new` to create data space for any type of variable. For example, if a program declares a `struct` using the code

```
struct DateType
{
   int day;
   int month;
   int year;
};
```

then it is possible to create a data block for a single record with the code

```
DateType *dtptr { new DateType };
```

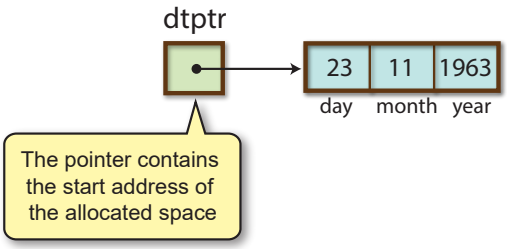
Fields can be initialised in the same way as we do when declaring a standard `struct` variable:

```
DateType *dtptr { new DateType{23,11,1963} };
```

The setup created is illustrated in FIG-10.26.

FIG-10.26

Creating *struct* Space



Once created, the fields within the dynamically allocated data space are normally accessed using the indirect member selector operator. For example, we could assign the value 31 to the *day* field using the line

```
dtptr -> day = 31;
```



It's also possible to use the more awkward dereference operator (e.g. `(*dtptr).month`)

Activity 10.14

Create a new program called *DynamicRecord*, which sets up a *DateType* structure as defined above. Create a *DateType* pointer and allocate space for a record.

Set the date within the record to 23, 11, 1963 and display the contents of the record.

The *delete* Statement

When an executing program reaches the end of the block in which a variable has been

declared, the space allocated to that variable is automatically freed making it available to be assigned to variables created in later parts of the program code. However, any space that has been reserved using the `new` command will not be freed automatically (but the pointer to it will be if it has reached the end of its declaration block!). Instead, the programmer must free up that space using the `delete` statement.

To free up the space referenced by *iptr*, we would use the line

```
delete iptr;
```

Although this statement frees the space referenced by *iptr*, it does not modify the contents of *iptr*. This means that the pointer still contains the address of the previously-allocated space, but this space is no longer available and any attempt at accessing it will create an error condition. A pointer containing the address of a location that has been freed is known as a **dangling pointer**.

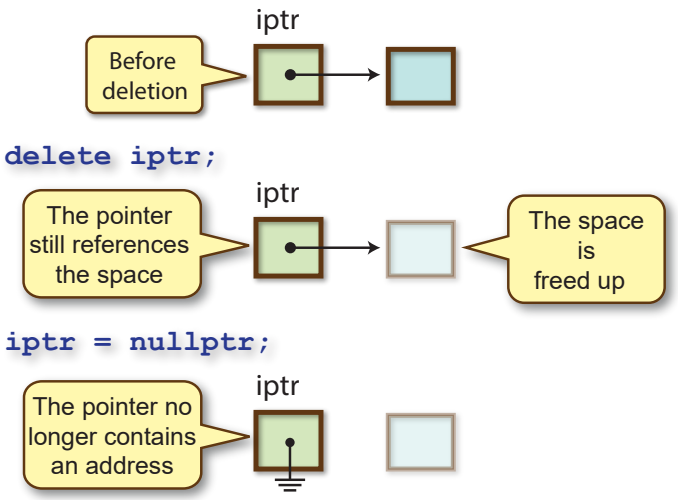
To avoid dangling pointers, any `delete` statement used where the pointer itself is not about to be deleted should be followed by a statement setting the pointer involved to null:

```
delete iptr;
iptr = nullptr;
```

The stages of freeing up the space are shown in FIG-10.27.

FIG-10.27

Freeing dynamically Allocated Space



The pointer itself (*iptr*) will have its own space freed automatically by the program when the end of the block in which it was declared is reached.

Activity 10.15

Modify *DynamicSpace,* so that the space allocated is deleted at the end of the program.

Note that using `delete` on a null pointer has no effect. Using `delete` with a dangling pointer (in other words, trying to free space that has already been freed) will cause a runtime error.

Free Store

While standard variables are stored in the stack, dynamically allocated data space is

allocated in an area called **free store**. Free store is, in effect, all the main memory within the computer which is not allocated to other software such as the operating system and other applications which may be currently loaded into memory.

Free store space which is allocated using `new` and not released using `delete` will, eventually be released when the program which reserved the space terminates. However, if a program continually reserves new space without releasing previously reserved areas which are no longer required, we get a situation known as **memory leakage** and this can, in extreme cases cause the program to crash when it runs out of free space.

To prevent memory leaks all that is required is that we release dynamically allocated space as soon as it is no longer required.

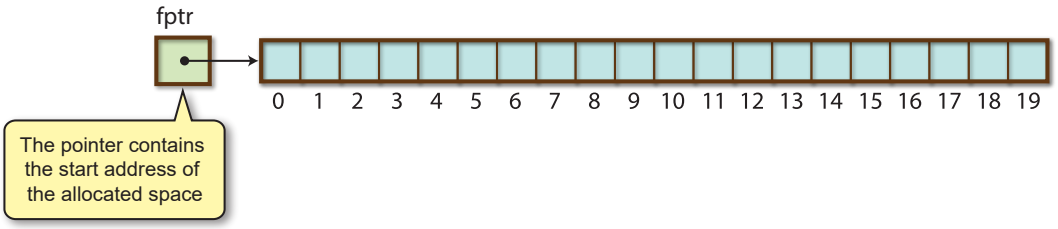
Creating Blocks of Space

Creating space for a single value is all very well, but of limited use. We can also create blocks of memory which allows us to save many values. In effect, this is similar to declaring an array. To create a data block, we must include the number of data spaces required in the `new` statement. This value must be enclosed in brackets. Hence, to create space for 20 `float` values we would write

```
float *fptr {new float[20]};
```

FIG-10.28 illustrates the effect of the above command.

FIG-10.28 Reserving Data Blocks



Once created, we can treat the block exactly like an array, using the pointer name in the same way as we might use an array name. This means that the expression

```
fptr[0]
```

gives us access to the first element in the data block and

```
fptr[19]
```

gives access to the last.

But we can also use pointer arithmetic to access the various elements. For example,

`*fptr` and `*(fptr+19)`

gives us access to the contents of the first and last elements.

We don't have to set the size of the data block in our code. Instead, we could read in the value for a variable and use that value to set the size of the block. This would require the following basic code:

```
int size;
cout << "Enter number of values to be stored : ";
cin >> size;
```

```
int *iptr { new int[size] };
```

Like standard arrays, we can initialise the contents of the data block created using the **new** statement with statements such as

```
int *iptr { new int[10]{ 0 } };     //Set all cells to zero
int *iptr2 { new int[5]{1,2,3} };   //Set cells to 1,2,3,0,0
```

However, one option we cannot use in conjunction with **new** is to have the compiler determine the actual size of the block. Hence the following line is invalid:

```
int *iptr3 { new int[] { 2,4,6,8 } };
```

Blocks of record structures can also be created. For example, five *DateType* elements would be created with the line

```
DateType *dtptr { new DateType[5] };
```

When the time comes to delete data blocks, the **delete** statement must include brackets. So to delete the space allocated for the five *DateType* records, we would use the line

```
delete [] dtptr;
```

Activity 10.16

Create a new project called *DynamicBlock* and use a pointer named *iptr* to reference space for four integer values. Have the program read in and display the contents of the block, then delete the allocated space.

Mismatched Deletions

If we create a pointer to a single data space and another to a memory block with statements such as

```
int *ptr1 {new int};
int *ptr2 {new int[5]};
```

C++ does not distinguish between these two situations so it has no way of knowing if a pointer references space for a single value or space where multiple values can be stored.

As a consequence, it's up to us as the programmer to make sure we use the appropriate **delete** command when freeing the allocated space. For example, should we accidentally use the statement

```
delete ptr2;
```

when attempting to free the data block referenced by *ptr2*, we would create a potential error situation.

void Pointers

The reason that pointers need to be associated with a data type (such as **int** or *DateType* as we saw in previous examples) is two-fold. Firstly, the pointer needs to know the type involved so that it will allocate the appropriate space when used in a **new** statement. Secondly, when a pointer variable is dereferenced, it needs to know what sort of data is stored at the specified address in order to correctly interpret the binary sequence found there.

Having given these reasons for a pointer to be specified with a data type, it is time to

admit that it is also possible to create a pointer without giving the type of data it is designed to reference. To do this, we declare the pointer as a `void` pointer. For example,

```
void *ptr;
```

Void pointers can reference any type of value. So if a program started with the lines

```
int no=12;
float  real=3.14;
char word[]="Hello";
```

then the void pointer *ptr* could be assigned an address using any of the following

```
ptr = &no;
ptr = &real;
ptr = word;
```

But, since void pointers have no way of knowing what type of data they are referencing, it is not possible to use the indirection operator with a void pointer. Hence, a statement such as

```
cout << *ptr << endl;
```

would be invalid.

In order to dereference a void pointer, it must first be cast to the type of value it references. So if we had used the line

```
ptr = &no;
```

to assign an address `int` variable *no* to the pointer, we could display the value held at that address using the line

```
cout << *static_cast<int*>(ptr) << endl;
```

which casts *ptr* as an `int` pointer (`static_cast<int*>(ptr)`) and then dereferences it (the asterisk at the start of the expression).

The program in FIG-10.29 uses a void pointer to a dynamically-allocated *DateType* space and then casts the pointer to access the fields of the record.

FIG-10.29

Using void Pointers

```
#include <iostream>
using namespace std;

//*** Using a void pointer to access a record ***
struct DateType
{
   int day;
   int month;
   int year;
};

int main()
{
   void *ptr;        //void pointer

   // *** Create DateType space ***
   ptr = new DateType;
   // *** Cast pointer to access space and assign values ***
   static_cast<DateType*>(ptr)->day = 23;
   static_cast<DateType*>(ptr)->month = 11;
```



FIG-10.29
(continued)

Using void Pointers

```
    static_cast<DateType*>(ptr)->year = 1963;
    // *** Display values held ***
    cout << static_cast<DateType*>(ptr)->day<<'/'<<
    ↳static_cast<DateType*>(ptr)->month<<'/'<<
    ↳static_cast<DateType*>(ptr)->year << endl;
}
```

Activity 10.17

Create a new project, *VoidPointer*, and implement the code given in FIG-10.29. Test and save your program.

When we need to delete the space referenced by a void pointer, we need to cast the pointer to the type of value involved otherwise the program has no way of knowing how much space to free. Deleting the space allocated in the program above would require the lines

```
delete static_cast<DateType*>(ptr);
ptr = nullptr;
```

In reality, there are very few reasons to use void pointers and they should only be employed as a last resort.

Summary

- Every constant and variable is assigned one or more bytes of memory.
- The address of the first byte assigned to a variable or named constant can be accessed using the address of operator (`&`).
- A pointer variable is designed to contain the address of a data value.
- C++'s original pointer types are known as raw pointers.
- When a pointer is defined the type of value at the address it contains must be specified followed by an asterisk then the pointer variable's name.
- The dereference operator (`*`) is used in conjunction with a pointer to access the data at the address held in the pointer.
- To show that a pointer does not contain an address, assign it the value `nullptr`.
- The address held in one pointer can be copied to a second pointer using the standard assignment statement as long as both pointers are declared as referencing the same data type.
- A pointer designed to contain the address of a named constant must begin its declaration with the keyword `const`. (e.g. `const int *ptr;`).
- The declaration of a pointer whose contents cannot be changed must include the term `const` after the data type (e.g. `int* const ptr;`).
- A pointer which references a named constant and whose contents cannot be changed must include the term `const` before and after the basic type (e.g. `const int *const ptr;`).
- A pointer to a pointer must be declared using two pointer operators (e.g. `int ** ptr;`).

- When an array is declared, a pointer constant is created and assigned the name of the array. This pointer contains the start address of the array.
- Using the dereference operator on an array name gives access to the data held in element zero of the array.
- Using bracket notation with an array dereferences the array pointer and calculates an address based on an offset from the start address of the array. This offset address is based on the subscript given within the brackets.
- If a standard pointer references an element within an array, the pointer can be made to contain the start address of the next array element by incrementing the pointer.
- It is possible to create an array of pointers.
- When a 2D array is created, the array name gives access to a pointer containing the address of an array of pointers.
- Pointers can reference `struct` or `union` variables.
- When a pointer references a `struct` or `union` data type, use the indirect member selector operator (`->`) to reference the data held within the fields of the data item.
- A pointer to a `char` value can access a single character but when used with other commands such as `cout` a complete string is accessed as long as the pointer is not dereferenced.
- When an integer value is added or subtracted from a pointer, the resulting address it contains will change by a multiple of the number of bytes occupied by the data type it is designed to reference.
- The subtraction of one pointer from another results in an integer giving the number of data values that can be stored between the two addresses.
- Use the `new` statement to dynamically allocate memory for a specified data type and store the start address in a pointer.
- Use brackets after the data type to reserve a block of memory for several data values.
- Use `delete` to free up memory previously allocated by the `new` command.
- Use `delete` and brackets to delete a memory block allocated previously by a `new` statement.
- A `void` pointer can contain the address of any data type.
- A `void` pointer must be cast to the appropriate type before the data it references can be accessed.

Smart Pointers

Introduction

As we have highlighted already, there are perhaps two main problems with using standard pointers to reference dynamically allocated space.

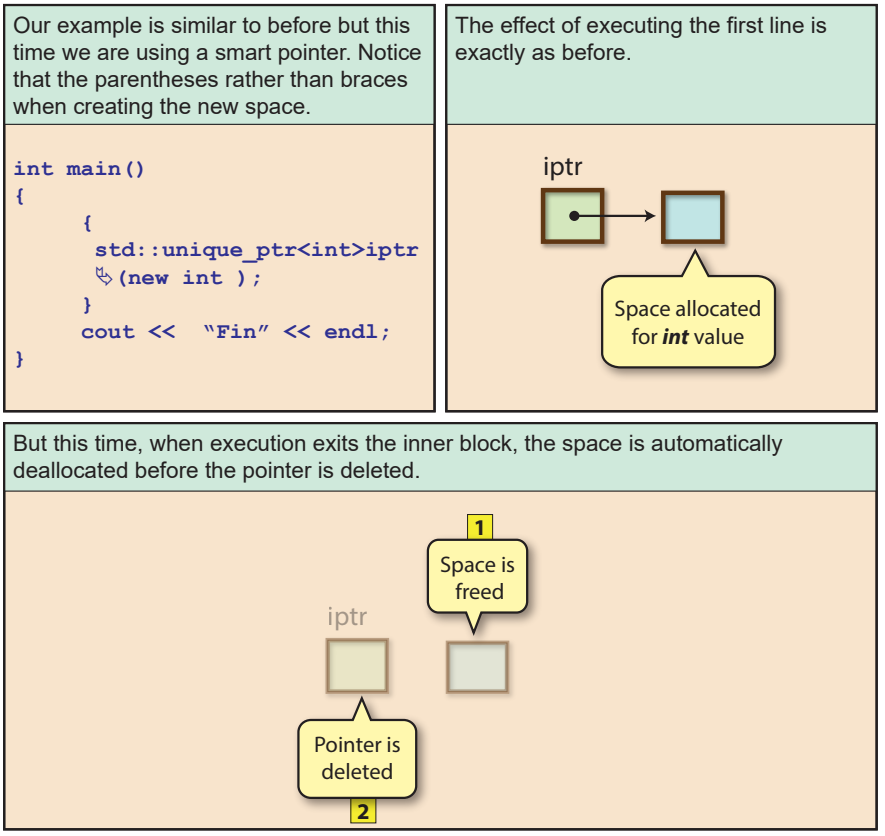
The first of these is that often a programmer will forget to deallocate that space before the pointer that references it is destroyed or before that pointer's contents changes to reference a different area. With no pointer referencing the allocated space, it becomes impossible to free up that space. This situation is known as a **memory leak**.

The second problem is that we may have a pointer which still references a space that has already been deallocated (a dangling pointer). Should we try to dereference the pointer at this stage, the best we can hope for is that the program will unexpectedly terminate. We can solve both these problems by abandoning the use of traditional raw pointers and making use of the newer **smart pointers**.

So what advantage do we obtain from using a smart pointer of this type? The biggest advantage of smart pointers is that when the pointer no longer exists the space it references is automatically deleted ensuring that we never suffer from memory leakage (see FIG-10.30).

FIG-10.30

Using a Smart Pointer



The inner block's code creates a smart pointer to a dynamically allocated `int` space.

unique_ptr Pointers

The first type of smart pointer allows only one pointer to reference a given space and

is defined with a line such as

```
std::unique_ptr<int> iptr1;
```

Notice that the type `unique_ptr` is defined within the `std` namespace and that the type of pointer we want is defined within the angled brackets. *iptr* is our pointer variable. Notice also that there is no requirement to use an asterisk to indicate that we are declaring a pointer variable.

The above syntax is used when creating a `unique_ptr` type which references space for a single value. When the pointer references a block of memory, then the declaration must include standard brackets after the basic type within the angled brackets:

```
std::unique_ptr<int[]> iptr2;
```

make_unique<>()

Although it is quite valid to use `new` when creating the space referenced by a unique pointer, the preferred method is to use the function `make_unique<>()`. So, rather than write

```
std::unique_ptr<int> iptr1(new int(23));// 23 in one space
```

we should write

```
std::unique_ptr<int> iptr1(std::make_unique<int>(23));
```

By using `make_unique<>()`, we can then use `auto` (but without the asterisk) to declare our pointer:

```
auto iptr1{std::make_unique<int>(23)};
```

Empty parentheses will store zero in the allocated space:

```
auto iptr1{std::make_unique<int>()};  // Stores zero
```

Once declared, our smart pointer can be dereferenced in exactly the same way as our earlier raw pointers. For example, we might write the lines:

```
*iptr = 12;
cout << *iptr << endl;
```

unique_ptr Operations

All smart pointers are actually objects. And although we are yet to cover that concept, all we need to know at the moment is that objects contain within them the ability to perform a set of defined operations. These operations are coded as a set of functions

To access a smart pointer's built-in operations we need to specify the pointer's name, a full stop, and the call to the function.

reset()

An important restriction when using smart pointers is that we cannot directly assign it a new address. For example, if we were to start with the line

```
std::unique_ptr<int> iptr{};
```

iptr would contain the null pointer value and a later line which attempted to assign it an address such as

```
iptr = new int;
```

will fail to compile.

The `reset()` function allows us to change the address held within a `unique_ptr` variable. Hence, to have *iptr* reference a dynamically allocated space we need to use the line

```
iptr.reset(new int);
```

Using `reset()` without a parameter,

```
iptr.reset();
```

will set the pointer to `nullptr`.



The dot notation is similar to field access used in struct variables, but in this case relates to class notation which we will cover in a later chapter.

When we use `reset()` to change the address held in our smart pointer, the space it referred to previously is automatically freed.

When we define a `unique_ptr`, no other pointer can contain the same address. For example, the following lines would fail to compile:

```
std::unique_ptr<int> iptr1(new int);
std::unique_ptr<int> iptr2;
iptr2.reset(iptr1);        //ERROR! Two pointers cannot contain
                           //the same address
```

get()

To find the address held in a smart pointer we cannot simply access its contents in the same way we could using a raw pointer. Instead, we must use the pointer's `get()` function. For example,

```
cout << iptr.get() << endl;
```

If no address is stored in the smart pointer, the value `nullptr` is returned.

swap()

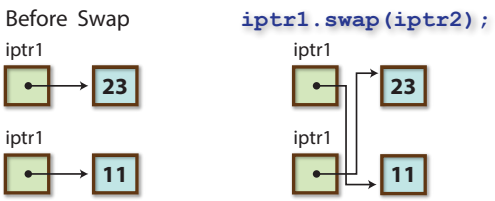
The contents of two `unique_ptr` variables can be swapped using `swap()`. For example, the code

```
auto iptr1(make_unique<int>(23));
auto iptr2(make_unique<int>(11));
iptr1.swap(iptr2);
```

has the effect shown in FIG-10.31.

FIG-10.31

The swap() Operation



release()

The `release()` function allows us to do something we wouldn't normally want to do! It clears the address in the smart pointer (setting it to null) without freeing associated memory.

However, it does return the address of the allocated space allowing us to store it elsewhere. For example, the code below clears a smart pointer but copies the address of the allocated space to a raw pointer.

```
auto sptr(make_unique<int>(23)); //Smart pointer
```

```
int *rptr;                          //Raw pointer
//*** Clear the smart pointer ***
//*** and copy address to raw pointer ***
rptr = sptr.release();
//*** Display value referenced by raw pointer ***
cout << *rptr << endl;
```

The Boolean Operator

We can already discover if a smart pointer contains an address using an expression such as

```
if(iptr.get() != nullptr)
```

but we can also perform this test by simply writing

```
if (iptr)
```

Activity 10.18

Start a new project called *UniquePointer* which creates a `unique_ptr<double>` variable, *dptr*, which references a dynamically allocated `double` space containing the value 3.14159265 and then display the value held.

shared_ptr Pointers

It looks like we have solved both the problems that we encountered when using raw pointers. But of course, in more complex code there are bound to be situations where we want more than one pointer to contain the same address. To do this we need to declare our smart pointers as `shared_ptr` types.

A `shared_ptr` smart pointer is created in a similar way to `unique_ptr` types:

```
std::shared_ptr<int> iptr1 (new int);
```

or the preferred

```
auto iptr1 {std::make_shared<int>(23)};
```

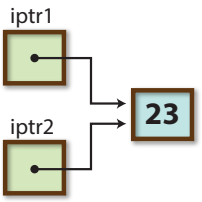
Using a second `shared_ptr` variable, we can have both pointers contain the same address:

```
std::shared_ptr<int> iptr2;
iptr2 = iptr1;
```

The resulting setup is illustrated in FIG-10.32.

FIG-10.32

shared_ptr Pointers



When using `shared_ptr` pointer variables, the program maintains a count of how many pointers are referencing the same dynamically allocated space. Only when that count reaches zero is that space deallocated.

Activity 10.19

Start a new project called *SharedPointers* containing shared pointers *iptr1* and *iptr2*. Dereference *iptr1* and assign the referenced space the value 12.

Add a `cout` statement which dereferences *iptr2* to display the value referenced.

shared_ptr Operations

Most of the operations available for `shared_ptr` smart pointers have the same name and a similar purpose as those we have already seen for `unique_ptr` types. These common functions are summarised below:

reset()	changes the address held in the pointer. If this is the only `shared_ptr` pointer to the current address, the space is freed.
get()	returns the address held in the pointer.
swap()	swaps the contents of two pointers.
Boolean operator	returns false if the pointer has the value `nullptr`.

use_count()

We can discover how many smart pointers reference the same area as the current pointer using `use_count()`. For example, we might write

```
if (fptr.use_count() > 1)
```

to check if any pointer other than *fptr* references the same area.

This function will return zero if the pointer is set to `nullptr`.

Smart Pointers to Memory Blocks

unique_ptr

Both types of smart pointers can be used to address larger data blocks. For example, we can create a `unique_ptr` pointer to an array of 20 elements using the line

```
std::unique_ptr <float[]> fptr {new float[20]};
```

Note that the type specified within the angled brackets needs to include the square brackets to signify that the pointer points to a block of memory rather than space for a single value.

However, although the line given above is perfectly valid, the preferred method of declaring a block is to use the `make_unique<>()` function:

```
auto fptr {std::make_unique<float[]>(20)};
```

As with raw pointers, we can then use the array notation to access the elements within the block created. For example,

```
fptr[1] = 4.8;
```

shared_ptr

Surprisingly, it's not possible to create a data block referenced by a *shared_ptr* in the same way as we did for the *unique_ptr* type. In other words, a line such as

```
auto fptr{ std::make_unique<float[]>(20) }; //ERROR
```

would fail to compile.

Instead, we have to resort to the `new` command as in the line

```
shared_ptr <float[]> fptr{ new float[20] };
```

if we wish to reference a memory block.

Once created, we can use the braces to dereference any cell in the block, such as in the lines

```
fptr[2] = 9.9F;
cout << fptr[2] << endl;
```

Activity 10.20

Start a new project called *BlockPointers.*

Use a `unique_ptr` to reference a 10-element `int` array. Store the value 8 in the fourth element.

Add a `shared_ptr` to reference an 8-element double array. Store the value 3.14 in the seventh element.

Display the contents of the fourth cell of the first array and the seventh cell of the second array.

Summary

- Smart pointers automatically free up dynamically allocated space when the pointer itself is destroyed.
- Smart pointers largely replace raw pointers when creating dynamically allocated space.
- There are two main types of smart pointers: `unique_ptr` and `shared_ptr`.
- Using `unique_ptr` pointers, no two pointers can reference the same address.
- Using `shared_ptr` pointers, two or more pointers can reference the same address.
- When using `shared_ptr` pointers C++ keeps a count of how many pointers reference the same address and only free up the referenced space when that count reaches zero.
- Use `new` or `make_unique<>()` to create space referenced by a `unique_ptr` pointer.
- Use `new` or `make_shared<>()` to create space referenced by a `shared_ptr` pointer.
- Use `reset()` with a smart pointer to change the address held by a smart

pointer.

- When `reset()` is used with a `unique_ptr` pointer, the space previously referenced is freed.
- When `reset()` is used with a `shared_ptr` pointer, the space previously referenced is freed only if the count of the number of pointers referencing the old address has reached zero.
- Use `get()` with a smart pointer to get the address held within the pointer.
- Use `swap()` with two smart pointers to swap the address contained in each.
- Use `release()` with a `unique_ptr` pointer to set the value held in the pointer to `nullptr` without freeing the space previously referenced.
- Use `use_count()` when dealing with `shared_ptr` pointers to discover the total number of pointers referencing the space referenced by the current pointer.
- When reserving a block of memory referenced by a `unique_ptr` pointer, the preferred method is to use `make_unique<`*type*`[]>(`*no. of cells*`)`.
- When reserving a block of memory referenced by a `shared_ptr` pointer, the only option is to use `shared_ptr <`*type*`[]>` *pointer_name*`{ new` *type*`[`*no. of cells*`] };`

Reference Variables

Introduction

While a reference variable is designed to contain the address of a standard variable just as a pointer does, that is where the similarity ends. Reference variables are defined using the `&` symbol in place of a pointer's `*` character. In the example below, the reference variable *iref* is set to contain the address of the `int` variable *no*:

```
int no = 12;
int &iref {no};
```

Notice that the ampersand before the variable whose address is being copied, which we would include when using a pointer, is missing (for a pointer we would have written `int *iptr { &no };`).

A reference variable must be assigned the address it is to hold as part of its declaration and cannot be changed later.

If we want to access *no* through the reference variable we would use statements such as

```
iref = 20;
cout << no;
```

As we can see, there is no dereferencing using the indirection operator. In effect, *iref* has become an alternative name for *no*. Both terms, *no* and *iref*, can be used interchangeably. The idea is visualised in FIG-10.33.

FIG-10.33

Creating a Reference Variable

Code	Description	Visual
`int no {12};`	Reserve space for an *int* value Label the space *no* Store the value 12 in the space	*no* **12**
`int &iref {no};`	Assign *no* the alternative name *iref*	*no* ***iref*** **12**
`iref = 20;`	Store the value 20 in the variable named *iref*	*no* ***iref*** **20**
`cout << no;`	Display the value in the variable named *no*	***no*** *iref* **20** → **20**

Using a Reference Variable

The Address of a Reference Variable

We know that we can find the address of a variable using a statement such as

```
cout << &no << endl; // Display the address of no
```

When we interrogate a reference variable in the same way, it does not give the address of the reference variable itself, but rather the address of the variable that it references. The short program below (FIG-10.34) demonstrates this effect.

FIG-10.34

Using a Reference Variable

```
#include <iostream>
using namespace std;

int main()
{

   int no{ 12 };
   int &iref{ no };
   //*** Output address of each variable ***
   cout << &no << endl;
   cout << &iref << endl;
   //*** Output contents ***
   cout << no << endl;
   cout << iref << endl;
}
```

The first two output lines of the program display the same address while the second two output the same value (12).

Range-Based *for* Loops

If we have a one dimensional array whose contents are to be modified then, up to this point, we've had to use a traditional `for` loop to access the elements of that array. For example,

```
//*** Set up numbers ***
int numbers[]{ 2,4,6,8 };
//*** Add one to each number ***
for (int c = 0; c < 4; c++)
   numbers[c]++;              //Changes contents of numbers
```

Attempting the same operation using a range-based `for` loop won't work since we are working with a copy of the original array cell value:

```
//*** Set up numbers ***
int numbers[]{ 2,4,6,8 };
//*** Add one to each number ***
for (auto v: numbers)
   v++;                    //Has no effect on numbers
```

However, if we declare *v* as a reference variable, it becomes an alias for the actual cell within *numbers*, and so can change the values held there:

```
//*** Set up numbers ***
int numbers[]{ 2,4,6,8 };
//*** Add one to each number ***
for (auto &v: numbers)
   v++;                    //Changes contents of numbers
```

Activity 10.21

Start a new project called *Reference* containing an array called *numbers* which has the initial values 2,4,6,8,10,12,14.

Using a range-based `for` loop as shown above, increment the contents of every even numbered cell (cells 0, 2, 4, and 6) by 1.

Display the contents of *numbers* before and after the contents are updated.

We also need to make use of at least one reference variable when trying to use nested range-based `for` loops to access the cells in a two-dimensional array.

Let's assume we simply want to use range-based `for` loops to display the contents of a two dimensional array such as

```
int grid[3][5]{{2,4,6,8,10},{12,14,16,18,20},{22,24,26,28,30}};
```

To access a single row of our array, we need to use a line such as

```
for (auto line: grid)
```

and then access the values within that row using the line

```
for(auto v: line)
```

Unfortunately, this won't work because using a copy of the row (stored in *line*) does not supply enough information on the size of row. Instead, we need the first `for` to use a reference variable:

```
for (auto &line: grid)
```

Now, we are free to complete our code:

```
for (auto &line : grid)
{
   for (auto v : line)
      cout << v << " ";
   cout << endl;
}
```

Of course, if we wanted to change the contents of *grid*, then *v* would also need to be a reference variable.

Activity 10.22

Start a new project called *Reference02* which contains a 2D array called *grid* which contains the values {2,4,6,8,10},{12,14,16,18,20},{22,24,26,28,30}

Using nested range-based `for` loops as shown above, display the contents of *grid*.

When dealing with higher dimensional arrays using nested range-based `for` loops, all but the inner-most loop must make use of reference variables.

Reference to Constant

Let's assume our program defines the constant PI:

```
const double PI {3.14159};
```

and that we want to create an alternative name for our constant using a reference variable. If we attempt a line such as

```
 double &PIE {PI};
```

our code will not compile. Instead, we must define PIE as a `const` too:

```
const double &PIE {PI};
```

Reference to an Array

It is not possible to use a reference variable to create an alternative name for an array.

Hence, code such as

```
double nums[10];
double &list{ nums };   //INVALID
```

will not compile. And this shouldn't come as too much of a surprise when we remember that an array name on its own acts like a constant pointer.

However, it is possible to create an alternative name for a single element of the array:

```
double nums[10];
double &first{ nums[0] };   //VALID
```

Reference to Pointers

We can create a reference to a raw pointer with code such as:

```
int *ptr{ new int(12) };
int* &alt{ ptr };
```

and then dereference the reference variable to access the data:

```
cout << *alt << endl;
```

A reference to a constant pointer would be coded as:

```
const int *fixed{ new int(14) };
const int* &alt{ fixed };
```

Unfortunately, we cannot use the above approach to create a reference to an array. So a second attempt at doing this with lines such as

```
double nums[10];
const double &list{ nums };   //INVALID
```

will still fail.

Aliases for Pointers and Reference Types

In the last chapter we saw how `using` can be employed to assign aliases to type names. We can also make use of these keywords to create aliases for pointer types. For example, while the obvious approach to creating *p1*, *p2*, and *p3* as `unique_ptr<int>` pointers is to write

```
std::unique_ptr<int> p1, p2, p3;
```

we could, as an alternative, write

```
using UPointer = std::unique_ptr<int>;
UPointer p1, p2, p3;
```

The same approach can be used with reference variable declaration, allowing us to write lines such as

```
using IntReference = int&;
int no = 12;
IntReference r {no};
```

Summary

- A reference variable contains the address of another variable but does not make use of the *address of* operator (`&`) or the dereference operator (`*`).
- In practice, we can think of a reference variable as assigning an alternative name to an existing variable.

- A reference variable is declared by following the basic type with an ampersand (e.g. `int &refvar`)
- A reference variable must be assigned the variable to which it is linked as part of the declaration.
- When accessing the address of a reference variable the address of the variable to which it is linked is returned.
- A reference variable can be used in a range-based `for` loop to allow the values in the original collection which is being traversed to be modified.
- When using nested range-based `for` loops to access higher dimensioned arrays, all but the inner-most loop must use reference variables even when the values in the array are not being changed.
- When creating a reference for a named constant, the reference variable declaration must begin with the term `const`.
- It is not possible to define a reference variable for an array.
- When declaring a reference variable for a pointer the asterisk must be included in the data type (e.g. `int* &ref {ptr};`).
- An alias can be defined for pointer and reference types.

Solutions

Activity 10.1

Code for *AddressOf*:

```
#include <iostream>
using namespace std;

//*** Displays the contents and address of num ***
int main()
{
  int num{ 21 };

  cout << num << endl;
  cout << &num << endl;
}
```

Activity 10.2

```
int * iptr;
float * fptr;
```

Activity 10.3

small : 25FBF6
big : 25FBF2
shptr : 25FBEE
iptr : 25FBEA

Activity 10.4

Modified code:

```
#include <iostream>
using namespace std;

int main()
{
  int no1, no2, ans;
  int *iptr1 = &no1, *iptr2 = &no2, *iptr3 = &ans;

  cout << "enter two numbers ";
  cin >> *iptr1 >> *iptr2;
  if (*iptr1 < *iptr2)
    *iptr3 = *iptr1;
  else
    *iptr3 = *iptr2;
  cout << "Smaller value is " << *iptr3 << endl;
}
```

Activity 10.5

a) `iptr1 == iptr2`
b) `*iptr == *iptr2`

Activity 10.6

```
float *fptr = &x;
const int *iptr = &MAX;
const double *const dptr = &TAX;
```

Activity 10.7

Code for *UsingPointers*:

```
#include <iostream>
using namespace std;

int main()
{
  float fno;
  float *fptr = &fno;
  float **pfptr = &fptr;

  cout << "Enter a number : ";
  cin >> **pfptr;
  cout << "Value is " << **pfptr << endl;
}
```

Activity 10.8

a) 25FD00
b) 25FD00 (same as a))
c) 25FD0B
d) 25FD1B

Activity 10.9

No solution required.

Activity 10.10

Modified code for *ArrayPointers*:

```
#include <iostream>
using namespace std;

// *** Access array through a pointer variable ***
int main()
{
  int list[]={2,7,1,9,12};
  //*** Make pointer reference array ***
  int *iptr=list;
  //*** Increment each element of the array ***
  for(int c =0; c< 5; c++)
    ++*iptr++;
  //*** Reset pointer ***
  iptr= list;
  //*** Display array contents by incrementing
  ↳pointer ***
  for(int c = 0; c < 5; c++)
    cout << *iptr++ << endl;
}
```

The rather unreadable line

```
++ *iptr ++;
```

increments both the contents at the address referenced and the contents of the pointer itself.
Perhaps a more readable version would be

```
*iptr++  += 1;
```

Activity 10.11

Modified code for *ArrayPointers*:

```
#include <iostream>
using namespace std;

// *** Access an array through a pointer variable
***
int main()
{
  int list[]{ 2,7,1,9,12 };

  //*** Make pointer reference array ***
  int *iptr{ list };
  //*** Increment contents of each element ***
  for (int c = 0; c < 5; c++)
    ++(*(iptr+c));
  //*** Display array contents by incrementing
  ↳pointer ***
  for (int c = 0; c < 5; c++)
    cout << *iptr++ << endl;
}
```

Note that, since the pointer's own value is not changed during the incrementation stage, there is no need to reset it before displaying the values.

Activity 10.12

Code for *StructPointer*:

```
#include <iostream>
using namespace std;

  struct DateType
  {
    int day;
    int month;
    int year;
  };

// *** Pointer to a record structure ***
```

```
int main()
{
  //*** Date variable ***
  DateType date;

  //*** Have pointer reference variable ***
  DateType *dtptr = &date;

  //*** Read the date ***
  scanf("%d/%d/%d",&dtptr->day,&dtptr->month,
  ↳&dtptr->year);
  //*** Display the date ***
  printf("%d/%d/%d\n",dtptr->day,dtptr->month,
  ↳dtptr->year);
}
```

Activity 10.13

Modified code for *DynamicSpace*:

```
#include <iostream>
using namespace std;

int main()
{
  //*** Define pointer, allocate space, and set to 12
  ↳***
  int *iptr {new int(12)};

  // *** display value stored ***
  cout << *iptr << endl;
}
```

Activity 10.14

Code for *DynamicRecord*:

```
#include <iostream>
using namespace std;

struct DateType
{
  int day;
  int month;
  int year;
};


// *** Dynamic memory allocation for a record ***
int main()
{
  //*** Create dynamic space for date ***
  DateType *dtptr = new DateType;

  //*** Set the date ***
  dtptr->day = 23;
  dtptr->month = 11;
  dtptr->year = 1963;

  //*** Display the date ***
  cout << dtptr->day << '/' << dtptr->month << '/'
  ↳<< dtptr->year << endl;
}
```

Activity 10.15

Modified code for *DynamicRecord*:

```
#include <iostream>
using namespace std;

struct DateType
{
  int day;
  int month;
  int year;
};

// *** Dynamic memory allocation for a record ***
int main()
{

  //*** Create dynamic space for date ***
  DateType *dtptr = new DateType;

  //*** Set the date ***
  dtptr->day = 23;
  dtptr->month = 11;
  dtptr->year = 1963;

  //*** Display the date ***
  cout << dtptr->day << '/' << dtptr->month << '/'
  ↳<< dtptr->year << endl;

  //*** Free space previously allocated ***
  delete dtptr;
}
```

Activity 10.16

Code for *DynamicBlock*:

```
#include <iostream>
using namespace std;

// *** Dynamic memory allocation for int array ***
int main()
{
  //*** Create dynamic space for array ***
  int *iptr = new int[4];

  //*** Read in values ***
  for(int c = 0; c <= 3; c++)
  {
    cout << "Enter number " << c+1 << " : ";
    cin >> iptr[c];
  }

  //*** Display values ***
  for(int c = 0; c <= 3; c++)
    cout << iptr[c] << endl;

  //*** Free space previously allocated ***
  delete [] iptr;
}
```

Activity 10.17

No solution required.

Activity 10.18

Code for *UniquePointer*:

```
#include <iostream>
using namespace std;

int main()
{
  unique_ptr<double> dptr{ new double(3.14159265) };
  cout << *dptr << endl;
}
```

Activity 10.19

Code for *SharedPointers*:

```
#include <iostream>
using namespace std;

int main()
{
  shared_ptr<int> iptr1(new int);
  shared_ptr<int> iptr2;
  //** Assign contents of iptr1 to iptr2 ***
  iptr2 = iptr1;
  //*** Store 12 in the dynamic space ***
  *iptr1 = 12;
  //*** Display value referenced by iptr2 ***
  cout << "iptr2 references an address containing "
  ↳<< *iptr2 << endl;
}
```

Activity 10.20

Code for *BlockPointer*:

```
#include <iostream>
using namespace std;

int main()
{
  auto uqfptr{make_unique<float[]>(10) };
  shared_ptr <float[]> shfptr{ new float[8] };
  //*** Assign values ***
  uqfptr[3] = 8;
  shfptr[6] = 3.14;
```

```
  //*** Display values ***
  cout << uqfptr[3] << endl;
  cout << shfptr[6] << endl;
}
```

Activity 10.21

Code for *Reference*:

```
#include <iostream>
using namespace std;

int main()
{
  //*** Set up numbers ***
  int numbers[]{ 2,4,6,8,10,12,14 };
  //*** Display contents ***
  for (auto v : numbers)
    cout << v << " ";
  cout << endl;
  //*** Add one to each number ***
  for (auto &v: numbers)
    v++;
  //*** Display contents ***
  for (auto v : numbers)
    cout << v << " ";
  cout << endl;
}
```

Activity 10.22

Code for *Referenc02*:

```
#include <iostream>
using namespace std;

int main()
{
  //*** Set up numbers ***
  int grid[3][5]{ {2,4,6,8,10},{12,14,16,18,20},
  ↳{22,24,26,28,30} };
  //*** Display contents ***
  for (auto &line : grid)
  {
    for (auto v : line)
      cout << v << " ";
    cout << endl;
  }
}
```

11

User-Defined Functions – 1

In this Chapter:

- ❑ Creating Mini-Specs
- ❑ Coding a Function
- ❑ Function Prototypes
- ❑ Function Parameters
- ❑ Value, Reference and Pointer Parameters
- ❑ Array Parameters
- ❑ Returning a Value
- ❑ *const* Parameters

User-Defined Functions - 1

Introduction

Most desktop computers are made up of several separate components: keyboard, screen, mouse, and inside the main casing are other discreet pieces such as the hard disk, video card and sound card.

Why are computers made this way, as a collection of separate pieces rather than have everything encased in a single frame? Well, there are several reasons. Firstly, by using separate components each can be designed to perform just one specific task such as: get information from the user (keyboard); display information (the screen); store information (the disk) etc. This allows all of these items to be made and tested separately. Also, if a component breaks down or needs to be replaced, we simply have to unplug that component and replace it with a new one.

Why is all of this relevant to creating programs? Years of experience have shown that the advantages of this modular approach to construction doesn't just apply to physical items such as computers, it also applies to software.

Rather than create programs as one continuous set of instructions, we can split the program into several **routines** (also known as **subroutines** or **functions**), each routine being designed to perform just one specific function. This is particularly important in long programs and when several programmers are involved in creating the software.

Once we have split our program into several functions, each designed to do a specific job, we then execute each function in the order required to solve our problem.

Writing our own Functions

Designing a Function

The first stage in creating a function is to decide what task the function has to perform. For example, we might want a function to do something as simple as display a line of asterisks or move a sprite about the screen.

A good function will perform only a single task and be relatively short - perhaps no more than 20 to 30 lines of code (often much less).

When a team of people are involved in creating the software, it is important that the exact purpose of each function is written out in detail so there can be no misconceptions between the people designing the routine and those programming it.

Functions must also be given a name. This name should reflect the purpose of the function and often starts with a verb, since functions perform tasks.

So let's have a first attempt at writing out a design for a function that is to draw a line of asterisks on the screen (see below).

FUNCTION NAME	:	DrawLine
DESCRIPTION	:	Draws a horizontal line of 10 asterisks.

This function document is known as a **mini-spec** and, although it does not yet show all the features that will appear in a more complex mini-spec, it contains all the details we need to create our simple function. The only tricky part is to write a description that is unambiguous - something that is not always that easy! Notice the word *horizontal* has been included so that there is no possibility of the programmer deciding to create a function that produces a vertical line of asterisks.

Activity 11.1

List any other details that might be added to the description to make the requirements more exact.

Coding a Function

The first word in the function's code is:

```
void
```

This term is used to state that the function does not return a value as a result of executing the code it will contain. Next comes the name of the function:

```
void DrawLine
```

The name must be a valid and unique identifier.

The final part of this first line is a parentheses-enclosed list of any values that need to be supplied to the routine (these are known as the function's **parameters**). In this case there are none, but nevertheless, the parentheses must still be supplied:

```
void DrawLine()
```

An alternative way of showing that the function requires no parameters is to include the word `void` within the parentheses:



The first line of a function is known as the **function heading**.

```
void DrawLine(void)
```

Because a function always includes parentheses, when referring to a function by name in a piece of text – such as this book – it is common practice to include them in the function name. So rather than refer to the function as *DrawLine* (as we did earlier) the term *DrawLine()* is used. This convention also helps to differentiate between variable names and function names.

After the function heading has been created, a set of braces is used to enclose the statements of the function:

```
void DrawLine()
{

}
```

Between the braces are the lines of code required to perform the task specified for the function. In this case only a single line is needed:



The complete function code is also referred to as the **function definition**.

```
void DrawLine()
{
      cout << "**********\n";
}
```

Notice that the layout of the function is similar to `main()` as defined in all our previous programs. And this is because `main()` is itself a function. What makes `main()` a little different is that when any C++ program is run, function `main()` is automatically sought out and its code executed.

It is also good practice to precede the function with a brief comment of the function's purpose:

```
//*** Draws a line of 10 asterisks ***
void DrawLine()
{
      cout << "**********\n";
}
```

Calling a Function

The code within a function will only be executed if that function is explicitly **called** (function `main()` is the only exception to this rule). To call a function our program need only specify the function's name and supply parameter values within parentheses. Since our routine has no parameters, the parentheses remain empty. So, to execute the code within the function *DrawLine()* we would include the line

```
DrawLine();
```

at an appropriate point within `main()`.

The Final Code

Our complete program will now contain two functions: *DrawLine()* and `main()`. The order in which the routines appear has some effect on program execution. In the first listing (see FIG-11.1) the *DrawLine()* code appears first.

FIG-11.1

Using a Function

```
#include <iostream>
using namespace std;

//*** Draws a line of 10 asterisks ***
void DrawLine()
{
   cout << "**********\n";
}



int main()
{
   DrawLine();
}
```

Remember only the code within `main()` is executed automatically when we run a program, so the compiler doesn't mind if `main()` isn't the first routine in the program.

Activity 11.2

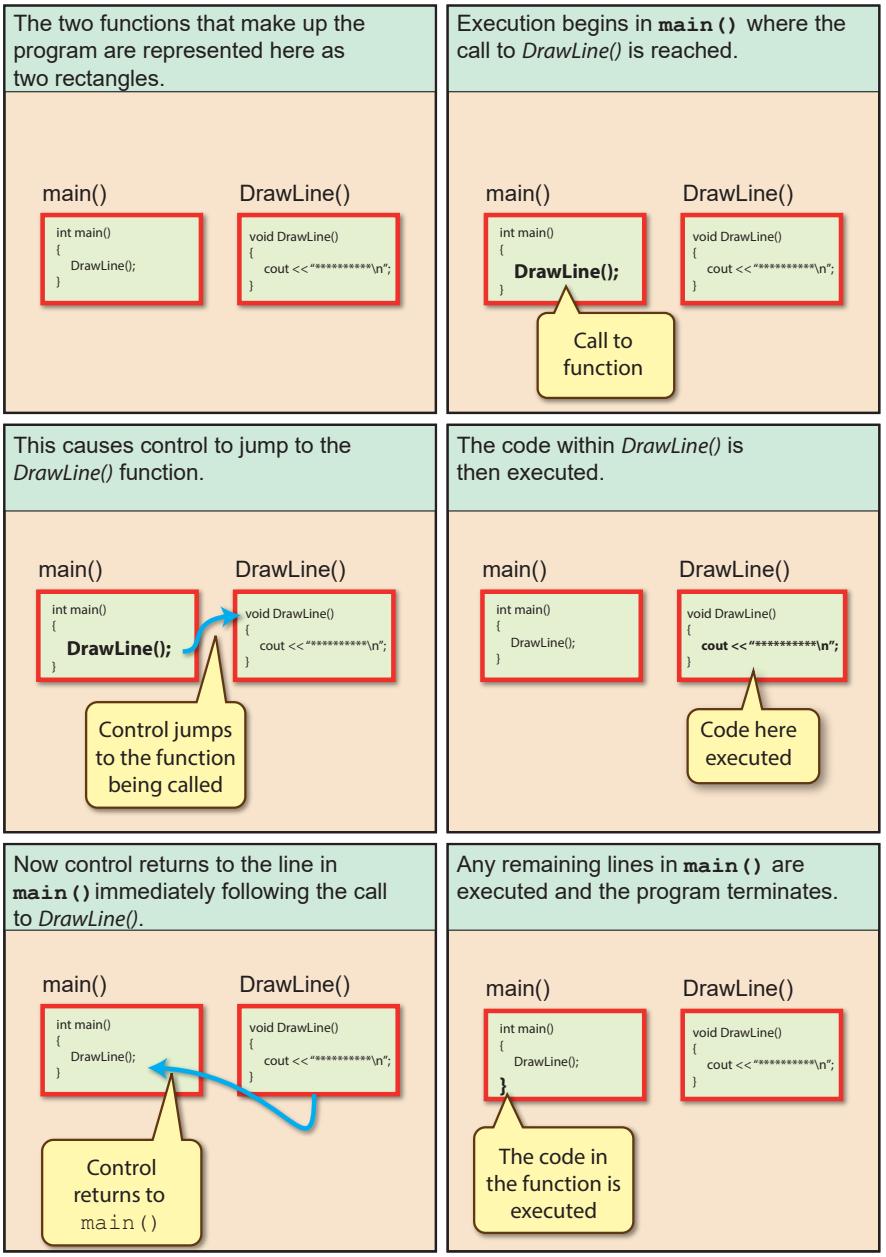
Start a new project called *UsingFunctions* and enter the code given in FIG-11.1. and check that it produces a line of asterisks.

How the Code is Executed

When a call is made to a function, control transfers to that function, its code is executed, and then control returns to the line immediately following the original call to the function. FIG-11.2 shows the stages involved during the execution of the program shown above.

FIG-11.2

The Function Calling Mechanism



A function can be called as often as required from any point in the program. So, if we wanted to draw two lines of asterisks, we would change the code in `main()` to

```
int main()
{
   DrawLine();
   DrawLine();
}
```

which adds a second call to the function.

Activity 11.3

Modify *UsingFunctions* so that two calls are made to *DrawLine()* as in the code shown above. Test your program to check that two lines of asterisks are drawn.

Repositioning the Function Code

Generally, programmers place the code for `main()` and the start of the program with the code for any other routines being placed after this.

Unfortunately, this creates a problem. The compiler can only accept a call to a function if it recognises that function's name and parameter list. When the function's code is placed before `main()` and before the call to the function, that's not a problem since the compiler has read the lines detailing the function before it arrives at the function call. However, if we move the function's code to the end of the listing, the compiler encounters the call to the function before the code for the function. This causes a compilation error.

To get round this problem we need to add a **function prototype** to the program, positioning it before the code for `main()`. A function prototype simply gives details of a function's return value type, name, and parameters without the actual code of the function itself. So a function prototype for *DrawLine()* would be

```
void DrawLine();
```

Notice that it's actually nothing more than the first line of the function itself with a semicolon at the end.

Activity 11.4

Modify *UsingFunctions* by moving the code for function *DrawLine()* to after that for `main()`.

Attempt to compile your program (*you should get an error message*).

Add a function prototype for *DrawLine()* immediately before the line

```
int main()
```

and recompile the program and check that it operates correctly.

Local Variables

We are at liberty to declare variables within a function (after all, we have already been doing that in `main()`). Variables declared within a function are **local** to that function. That is, they are allocated space within the computer's memory only while that function is being executed. Once execution finishes the allocated memory space is freed up to be reused later. This also means that these variables can only be accessed within the function in which they are declared and are referred to as **local variables** with their scope limited to the function in which they are declared.

So, knowing we can declare local variables, we can write a slightly longer version of the *DrawLine()* function which creates the same result as before:

```
//*** Draws a line of 10 asterisks ***
void DrawLine()
{
   //*** Draw the line ***
   for (int count = 1 ; count <= 10; count++)
      cout << '*';
   cout << '\n';
}
```

Activity 11.5

Modify *UsingFunctions* replacing the original *DrawLine()* with the code given above. Does the function prototype need to be changed?

Test your program to make sure it gives the same results as before.

Value Parameters

Single Parameter

So far we have had no need of parameters. But what if we wanted to allow the length of the line created by *DrawLine()* to be specified when the function is called. That way we would be able to create lines of different lengths. To do that we need to pass a value into the function when it is called.

If we are to change a function in order to make use of a parameter, we need to start by rewriting the description of that function adding details of the parameter such as the its name and type.

In the case of the *DrawLine()* function, the new mini-spec would be:

FUNCTION NAME	:	DrawLine
PARAMETERS		
In	:	length : integer
DESCRIPTION	:	Draws a horizontal line of *length* asterisks

Notice that the parameter is described as an **in parameter**. This description is used because the value is being "given to" or "passed into" the function.

To implement the passing of a parameter to the function in our code, we need to rewrite the first line of the function, specifying the parameter as a variable within the parentheses:

```
void DrawLine(int length)
```

The parameter given in the function definition is known as the **formal parameter**.

Notice that the formal parameter needs to be given a type and a name - just as if we were declaring a variable. And, in reality, that is exactly what we're doing. It is the value in this variable that we will use to determine the length of our line.

Next we have to rewrite the code within the function to make use of the parameter we have added:

```
//*** Draws a line containing length asterisks   ***
void DrawLine(int length)
{
   // *** Draw the line ***
   for (int c = 1 ; c <= length; c++)
      cout << '*';
   cout << '\n';
}
```

Notice how the expression in the `for` statement has changed so that the loop continues while the condition `count <= length` is *true*. Now we need to change the function prototype for the function given at the start of the program to reflect the fact that the function now requires a parameter. The new prototype is

```
void DrawLine(int length);
```

We can see that the parameter's name is given. However, we can omit the parameter name if we want, since it is not required in the prototype. Finally, the call made to the function from `main()` must supply a value for the parameter:

```
DrawLine(8);
```

The value given here will be copied into the function's formal parameter, *length*, just before the code of the function is executed. The new version of the program is shown in FIG-11.3.

The parameter specified when the function is called is known as the **actual parameter.**

FIG-11.3

Using a Function Parameter

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
void DrawLine(int length);


int main()
{
   DrawLine(8);
}

//*** Draws a line containing length asterisks   ***
void DrawLine(int length)
{
   // *** Draw the line ***
   for (int c = 1 ; c <= length; c++)
      cout << '*';
   cout<<'\n';
}
```

The return type and parameter type list is known as the **signature** of the function. For example the signature of *DrawLine()* at this point is `void (int)`

Activity 11.6

Modify *UsingFunctions* to match the code shown above. Test the program using various numbers in the function call line.

The mechanism used to handle parameter passing to a function is shown in FIG-11.4.

FIG-11.4

How Parameters are Handled

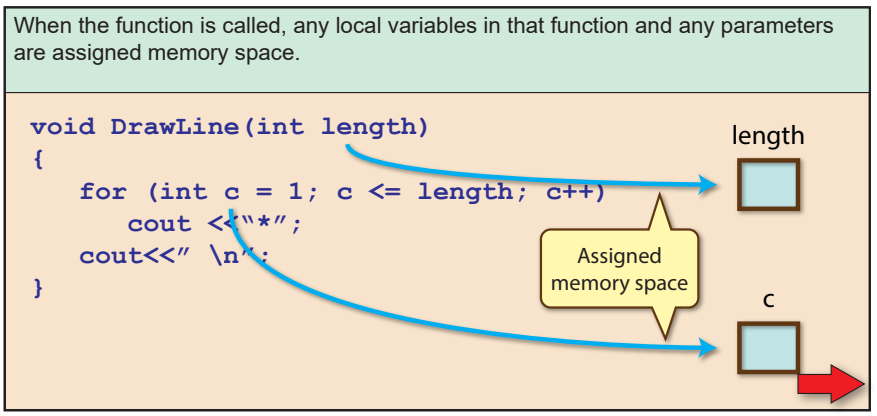
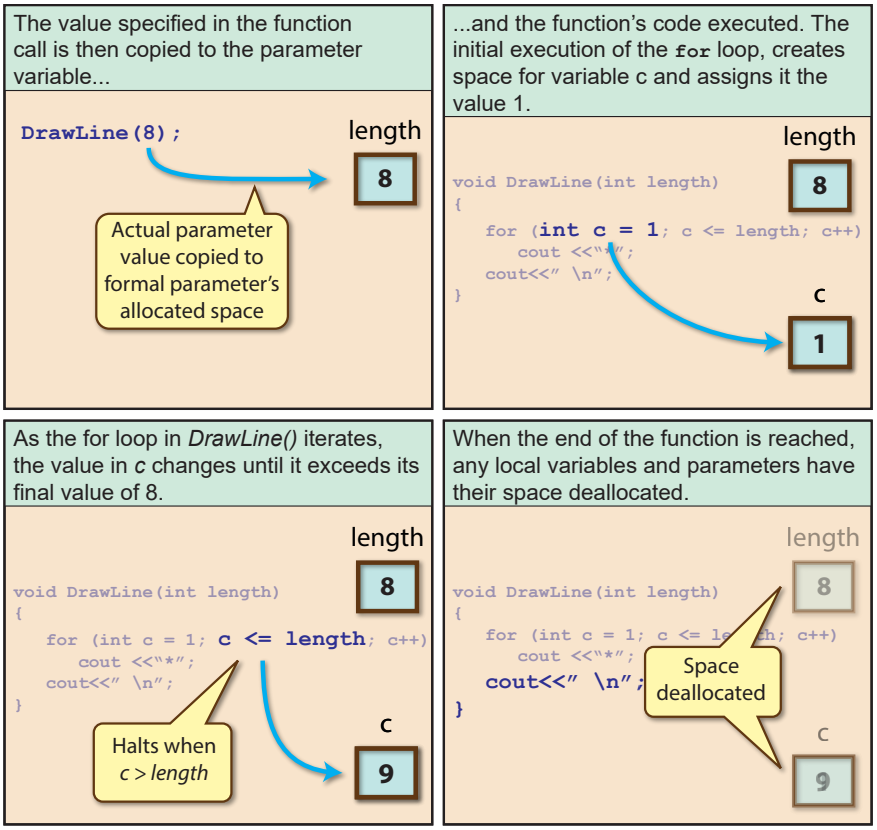


FIG-11.4
(continued)

How Parameters are Handled



Because the value of the actual parameter (8 in the example above) is copied to the formal parameter (*length*), this type of parameter passing is known as **pass-by-value**.

The value passed to a function when it is called may also be given in the form of a variable or expression.

This means that we could allow the user to set the length of the line using the code

```
cin >> size;
DrawLine(size);
```

This code assumes we have previously declared an int variable named *size*.

The value entered and stored in *size* would be copied into *length* when the function was called. If an expression is used as a parameter as in the line

```
DrawLine(size * 2);
```

the value of the expression `size * 2` is calculated and that value then copied into the function's parameter *length*.

Multiple Parameters

There is no practical limit on the number of parameters that may be supplied to a function, with each parameter being a different type if necessary.

In the next version of *DrawLine()*, not only is the length of the line passed as a parameter, but also the character used to construct the line.

First, we'll make the appropriate changes to the mini-spec:

```
FUNCTION NAME   :   DrawLine
PARAMETERS
   In           :   length    : integer
                    ch        : character

DESCRIPTION     :   Draws a horizontal line containing length
                    characters. The character used is ch.
```

The corresponding changes to the function prototype, function call and function code are then implemented (see FIG-11.5).

FIG-11.5

Using Multiple Parameters

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
void DrawLine(int, char);


int main()
{
   DrawLine(8, '#');
}

//*** Draws a line containing length ch characters ***
void DrawLine(int length, char ch)
{
   // *** Draw the line ***
   for (int c = 1 ; c <= length; c++)
      cout << ch;
   cout<<'\n';
}
```

Note how the code has been changed to make use of the parameter *ch* in drawing the lines of the rectangle. We can see in the call to the function that parameter *length* will be assigned the value 8 and *ch* the value '#'.

Activity 11.7

Modify your *UsingFunctions* project to match the code given in FIG-11.5 and test the program using various characters.

Parameter Type Differences

Back in Chapter 4 we saw that C++ can automatically handle many conversions between different data types. For example, C++ will quite happily execute the line

```
double v = 12;
```

even though *v* is of type `double` whereas 12 is of type `int`.

It will even allow us to attempt to copy a `double` into an `int`:

```
int v = 12.6;
```

though in this case, the 0.6 is lost and only 12 is stored.

Most surprising of all perhaps is that we can even copy a `char` value to an `int` variable or vice versa:

```
int v = 'A';   //Stores 65
char ch = 65;  //Stores A
```

These same rules apply when it comes to the actual parameters used in a function call. For example, if we were to call *DrawLine()* with the line

```
DrawLine(3.9, 65);
```

we will end up with the following output:

```
AAA
```

since only 3 will be copied to the *length* parameter and 65 will be stored as 'A' in the *ch* parameter of *DrawLine()*.

However, there is a danger in this automatic conversion. For example, let's say we've forgotten the order of the parameters when calling *DrawLine()* and think that the character should come first and the length second. This might cause us to write a line such as:

```
DrawLine('A',66);
```

which would end up producing the following output:

```
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
↪BBBBBBBBBBBB
```

a line of 65 letter B's!

What's happened here is that 'A' has been converted to 65 before being stored in the *length* parameter and 66 (which is the ASCII code for 'B') has been converted to 'B' before being stored in the *ch* parameter.

Activity 11.8

Modify your *UsingFunctions* project making various calls to *DrawLine()* with differing parameter types and observe the effects created.

struct Parameters

If we have defined a record structure in our program, we can pass parameters of that type to a function. For example, if we have defined a *DateType* structure, we could pass that to a function which then displays the day of the week on which that date fell.

The heading for this function would be:

```
void GetDayOfWeek(DateType dt)
```

The program in FIG-11.6 shows the complete code for the *GetDayOfWeek()* function as well as demonstrating its use.

FIG-11.6

Defining and Using *GetDayOfWeek()*

```
#include <iostream>
using namespace std;
```



FIG-11.6
(continued)

Defining and Using
GetDayOfWeek()

```
//*** User defined types ***
struct DateType
{
   int day;
   int month;
   int year;
};

//*** Function prototypes ***
void GetDayOfWeek(DateType dt);

void main()
{
   //*** Get date ***
   DateType date;
   cout << "Enter date in the format dd / mm / yyyyy : ";
   scanf("%d / %d / %d", &date.day, &date.month, &date.year);
   //*** Call function ***
   GetDayOfWeek(date);
}


//*** Displays the day of the week on which dt fell ***
void GetDayOfWeek(DateType dt)
{
   //*** Calculate modified date ***
   int M = (dt.month + 9) % 12 + 1;
   int modifiedyear = dt.year - M / 11;
   int C = modifiedyear / 100;
   int Y = modifiedyear % 100;
   //*** Calculate day of week ***
   int dayofweek = ((static_cast<int>(2.6 * M - 0.2) + dt.day + Y
   ↳+ Y / 4 + C / 4 - 2 * C) % 7 + 7) % 7;
   //*** Display result ***
   cout << dt.day << '/' << dt.month << '/' << dt.year <<
   ↳" fell on a " << dayofweek << endl;
}
```

Activity 11.9

Start a new project called *DOWFunction* which implements the code given above.

Test the program with various dates.

Activity 11.10

Start a new project called *TimeFunction*. In the program declare the following type:

```
struct TimeType
{
      int hrs;
      int mins;
      int secs;
};
```

Add a function called *AddTimes()* which takes two *TimeType* parameters (*t1* and *t2*) and displays the sum of the two times.



Activity 11.10 (continued)

In the main program, accept two time values from the keyboard and use these as the parameters to a call to *AddTimes()*. Check that the value displayed is the sum of the two times.

Pre-conditions and Post-conditions

Pre-conditions

When a function uses a parameter, we will often need to restrict the range of values which may sensibly be assigned to that parameter.

For example, when we specify what length of line we want *DrawLine()* to produce, it doesn't make sense to pass a negative value to the function. Equally, it makes little sense to request a line hundreds of characters long, since there is a limit to how many characters can fit on a single line of the program's window. We might therefore expect the value specified for *length* to be in a range such as 0 to 100.

When we place limitations on the conditions under which a function can operate successfully, these limitations are known as the **pre-condition** of the function.

We can therefore state that the pre-condition for *DrawLine()* is that the parameter *length* lies between 0 and 100.

We would start by adding this restriction to the mini-spec:

```
FUNCTION NAME   :   DrawLine
PARAMETERS
    In          :   length     : integer
                    ch         : character
PRE-CONDITION   :   0 <= length <= 100

DESCRIPTION     :   Draws a horizontal line of length characters. The
                    character used is ch.
```

We must be careful when choosing the pre-condition for a function. A pre-condition should only prevent situations which the function itself cannot handle. We should not create a pre-condition simply because we feel that using a value outside a range seems unreasonable to us. If the function's code can handle the situation, then allow that situation to occur.

For example, we have stated in the pre-condition of *DrawLine()* that *length* must be in the range 0 to 100. But could the code handle values outside that range? The answer to this question is, yes. Values of less than zero will simply mean that the `for` loop will iterate zero times and so the only output will be that caused by the `cout << "\n";` statement which will move subsequent output onto the next line on the screen. Values greater than 100, the characters produced may spread over several lines, but the function will still work as described. So we were probably mistaken to impose a pre-condition on the *length* parameter of *DrawLine()*.

A more realistic use of a pre-condition would be on a function such as



Since C++ contains a built-in square root function there would be little point in writing this function ourselves, but it is used here to illustrate the use of a pre-condition.

`void CalculateSquareRoot(double v)` which calculates the square root of *v*. Now, since negative numbers do not have a normal square root value, we could quite reasonably state the pre-condition of this function as:

PRE-CONDITION: v >= 0

In our *GetDayOfWeek()* function, a pre-condition would be that the contents of *dt*, the parameter, forms a valid date after the year 1752:

PRE-CONDITION: dt.day, dt.month, dt.year form a valid date AND dt.year>1752

A pre-condition is, in reality, a Boolean expression – it is something that must be *true* for the function to perform to specification. Therefore, when a function has no restrictions we would describe the pre-condition within the mini-spec as

PRE-CONDITION : true

since *true* is always *true*! However, sometimes the term *none* is used in the pre-condition to signify that there are no restrictions on the parameter values.

Activity 11.11

Write out the pre-condition for *AddTimes()* that checks that *t1* and *t2* are valid times (hours >= 0, minutes 0 to 59, seconds 0 to 59).

Post-conditions

If we are writing a formal mini-spec we may even include a **post-condition** statement. This is a statement of what must be *true* when the routine being defined has finished executing.

For some routines which are designed specifically to change the value held in a variable, writing a post-condition can be quite simple, but for other routines, the post-condition will be written in English and may be similar to the description in the routine. For example, the post-condition for *DrawLine()* could be written as:

POST-CONDITION : A horizontal line of *length* characters is displayed on the screen AND the line is constructed from character *ch*.

Note the use of the Boolean operator AND to link the two parts of the pre-condition. This is appropriate since, like the pre-condition, the post-condition should be constructed as a Boolean expression.



In future mini-specs in this book, we'll content ourselves with a description element and omit post-conditions.

The post-condition and description parts of a mini-spec are obviously similar (often identical) but the description may give more detail about intermediate steps involved whereas the post-condition only gives the final result and may be written in more mathematical terms where appropriate.

Coding Pre-Conditions

Now we need some way of implementing a pre-condition test within our code. We do this by adding an `if` statement at the beginning of the function which causes that function to be aborted if the pre-condition is not met.

The *return* Statement

The `return` statement is designed to be placed within a function. When executed, this command causes the remaining statements in the function to be ignored, ending execution of the routine and returning to the code which called the function in the first place.

We can make use of the `return` statement to terminate execution of a function when its parameter(s) fall outside an acceptable range.

In the case of *CalculateSquareRoot()*, we would add the following lines right at the start of the routine to check the value of *v*:

```
// *** if v negative, terminate function ***
if (v < 0)
   return;
```

This `if` statement at the start of the function is known as a **parameter guard** and should always be included when there are any parameter values which the function's code cannot handle correctly.

Activity 11.12

In *TimeFunction*, modify *AddTimes()* so that the function exits if either *t1* or *t2* does not contain a valid time.

Complex Pre-conditions

Some pre-conditions can require significant amounts of code in order to check that they have been met. For example, checking that the parameter for *GetDayOfWeek()* forms a valid date after 1752 requires code such as the following:



We'll make use of this code in a moment.

```
//*** Check date is valid ***
//** If invalid year, exit **
if (dt.year <= 1752)
   return;
//** If invalid month, exit **
if (dt.month < 1 || dt.month > 12)
   return;
//** If invalid day, exit **
int daysinmonth[]{ 0,31,28,31,30,31,30,31,31,30,31,30,31 };
//* If leap year add 1 to days in Feb ***
if ((dt.year%4 == 0 && dt.year%100 != 0)||(dt.year%400 == 0))
   daysinmonth[2]++;
if (dt.day < 1 || dt.day > daysinmonth[dt.month])
   return;
```

Returning a Value from a Function

Mini-Spec Out Parameters

Not only can we supply values to a function (in the form of parameters), but functions can also return a result. For example, let's assume we wish to create a function called *SumIntegers()* which takes an integer parameter, *value*, and returns the sum of all the integer values between 1 and *value*. When describing such a routine in a mini-spec, we add the returning value as an **out parameter**.



The term **out parameter** may be a little misleading since the value returned by a function rarely goes in the parameter list when the code is written.

FUNCTION NAME	:	SumIntegers	
PARAMETERS			
In	:	value	: integer
Out	:	result	: integer
PRE-CONDITION	:	true	
DESCRIPTION	:	Sets *result* to the sum of the integers between 1 and *value*.	

Implementing an Out Parameter in a Function

To return a value from a C++ function we first replace the term `void` in the function signature with the type of value to be returned. In the case of *SumIntegers()*, the function prototype would be

```
int SumIntegers(int);
```

with the function heading being:

```
int SumIntegers(int value)
```

Notice that the value being returned is not included in the parameter list. Instead, when coding the body of the function, we use a `return` statement which includes the value to be returned. So *SumIntegers()* would be coded as:



There is a much better way to tackle this problem! See Chapter 25.

```
//*** Returns the sum of all integers between 1 and value ***
int SumIntegers(int value)
{
   int result = 0;
   for (int c = 1; c <= value; c++)
      result += c;
   return result;
}
```



Should control reach the end of a function without executing a `return` statement, that is equivalent to a `return` statement that returns no value.

When a function returns a value, the call to that function is a little different from before. Rather than call the function with a code such as

```
cin >> num;
SumIntegers(num);
```

we need to make use of the value being returned by the function. Often we will store the returned value in a variable as in the line

```
int answer = SumIntegers(num);
```

But the returned value could be displayed

```
cout << SumIntegers(num);
```

Or used in an arithmetic expression

```
int answer = 60 - SumIntegers(num);
```

Or in a Boolean condition

```
if (SumIntegers(num) > 100)
```

We can even use the value returned by one function as the parameter for another. For example the command

```
DrawLine(SumIntegers(5),'_');
```

would use the value returned by *SumIntegers()* as the first parameter of our call to *DrawLine()*.

As a general rule, since calling the function returns a value, we can call the function anywhere in our code where a value of the type returned by the function is required.

Activity 11.13

Start a new project called *SumFunction* containing the code for *SumIntegers()*. The program should read in an integer value (*v*) which lies between 1 and 20 and display the value returned by the call *SumIntegers(v)*.

We saw earlier that checking to see if the parameter to *GetDayOfWeek()* was a valid date required several lines of code to be added to the function. In fact, the code is so

complex that we might consider moving it to a separate function, *IsValidDate()*, which returns a Boolean value:

```
//*** Check date is valid ***
bool IsValidDate(DateType dt)
{
   bool result{ true };
   //** If invalid year, result is false **
   if (dt.year <= 1752)
      result = false;
   else
      //** If invalid month, result is false **
      if (dt.month < 1 || dt.month > 12)
         result = false;
      else
      {
         //** If invalid day, result is false **
         int daysinmonth[]{0,31,28,31,30,31,30,31,31,30,31,30,
         ↳31};
         //* If leap year and 1 to days in Feb ***
         if ((dt.year%4 == 0&&dt.year%100 != 0)||(dt.year%400
         ↳== 0))
            daysinmonth[2]++;
         if (dt.day < 1 || dt.day > daysinmonth[dt.month])
            result = false;
      }
   //*** Return date's validity ***
   return result;
}
```

The advantages of this are that it reduces the complexity of *GetDayOfWeek()* and, as a separate function, it can be tested separately and may be easily transferred for use in other programs. Using this new routine, the parameter guard code for *GetDayOfWeek()* reduces to

```
if (!IsValidDate(dt))
   return;
```

Activity 11.14

Modify *DOWFunction* to include the function *IsValidDate()* (and its prototype). Add the parameter guard code at the start of *GetDayOfWeek()* and check that the program still operates correctly.

In our function, *GetDayOfWeek()*, we had the day-of-week value for a given date displayed on the screen. In fact, it is most unusual for a function to display a value in this way. Doing so restricts the usefulness of the function since it is only capable of displaying the value. By returning the value we have many more options as to how the calculated value is used (as we saw with *SumIntegers()*).

Our first attempt at changing the code for *GetDayOfWeek()* so that it returns a value would be:

```
//*** Returns the day of the week on which dt fell ***
int GetDayOfWeek(DateType dt)
{
   if (!IsValidDate(dt))
      return;
   //*** Calculate modified date ***
   int M = (dt.month + 9) % 12 + 1;
   int modifiedyear = dt.year - M / 11;
   int C = modifiedyear / 100;
```

```
    int Y = modifiedyear % 100;
    //*** Return day of week ***
    return ((static_cast<int>(2.6 * M - 0.2) + dt.day + Y + Y /
    ↳4 + C / 4 - 2 * C) % 7 + 7) % 7;
}
```

Unfortunately, this new version won't compile!

Pre-Conditions and Return Values

If a function that returns a value contains a coded pre-condition, how can we handle the situation where the pre-condition is not met? Where a function has no return value then the situation is handled by simply exiting from the routine, but things are a bit more complicated when a value must be returned.

Our latest version of *GetDayOfWeek()* requires an `int` value to be returned. But the parameter guard attempts to exit the function without returning any value:

```
if (!IsValidDate(dt))
    return;                 //NO VALUE RETURNED
```

This is the cause of our compilation error. To solve this problem, we need to specify some `int` value as part of the `return` statement executed when the parameter guard forces an exit from the function.

When an error occurs, the one way of handling this is to return a value outside the normal range of possible return values. Since we know *GetDayOfWeek()* is designed to return a value in the range 0 to 6 any other value would do to represent an unmet pre-condition. We'll choose -1. Our parameter guard is now coded as

```
if (!IsValidDate(dt))
    return -1;
```

Activity 11.15

Modify *DOWFunction* so that *GetDayOfWeek()* returns the day-of-week calculated by the function (remove the `cout` statement within the function) and -1 if the date is invalid.

Display the returned value to check that the result is correct. Display "*Invalid date*" if -1 is returned by the function.

We will look at another approach to handling error conditions within a function in a later chapter.

Returning a *struct* Value

In project *TimeFunction* we created a function called *AddTimes()*. The current version of that function displays the sum of two given times. From what we've said earlier, it is obvious that we would be better served if this function returned the result rather than displayed it.

To do this we must show that the function returns a value of type *TimeType* (as defined within the program). So the function prototype would become

```
TimeType AddTimes(TimeType, TimeType);
```

and the function heading

```
TimeType AddTimes(TimeType t1, TimeType t2)
```

The code for the routine can then end with the lines

```
TimeType result;
result.hrs = total_secs / 3600;
result.mins = (total_secs - result.hrs * 3600) / 60;
result.secs = total_secs % 60;
return result;
```

However, we must also return a *TimeType* value if the pre-condition is not met. Again, we'll have to decide on a value that flags up the fact that the function has been unable to operate as expected. That means selecting values for the *hrs*, *mins*, *secs* fields in this returned value. So let's assume we've settled on the time value of -1:-1:-1 as an indicator of a fail. This would allow the final, updated version of the *AddTimes()* function to be coded as:

```
TimeType AddTimes(const TimeType &t1, const TimeType &t2)
{
   TimeType result {-1,-1,-1};
   if (t1.hrs < 0 || t1.mins < 0 || t1.mins > 59 || t1.secs < 0
   ↳|| t1.secs > 59||t2.hrs < 0 || t2.mins < 0 || t2.mins > 59
   ↳|| t2.secs < 0 || t2.secs > 59)
   return result;
   int total_secs = (t1.hrs + t2.hrs) * 60 * 60 + (t1.mins +
   ↳t2.mins) * 60 + (t1.secs + t2.secs);
   result.hrs = total_secs / 3600;
   result.mins = (total_secs - result.hrs * 3600) / 60;
   result.secs = total_secs % 60;
   return result;
}
```

Activity 11.16

Modify the code for *AddTimes()* in the *TimeFunction* project to match the code given above, returning a *TimeType* value.

`main()` should display the total time or the message "*One or more of the times given was invalid*".

Returning a *pair* or *tuple* Value

In *AddTimes()* we returned a value of type *TimeType*. In effect, this means we end up returning three `int` values (in fields *hour*, *minute* and *second*).

But there will be times when we want a function to return more than one value where no appropriate struct type exists to hold those values. In this situation, returning a pair or tuple can be useful.

For example, let's assume we want to create a function called *Extremes()* that returns the largest and smallest values found in an array of integers. We could code such a function to return a `pair` value. The function's prototype is

```
pair<int,int> Extremes(int*, int);
```

Notice that we must specify the type of values to be stored in the pair when giving the return type.

The function's code is

```
//*** Return lowest and highest number in
pair<int, int> Extremes(int* list, int sz)
{
```

```
    //*** Set lowest and highest ***
    int lowest = list[0];
    int highest = list[0];
    //*** Search for lower and higher values ***
    for (int c = 1; c < sz; c++)
       if (list[c] < lowest)
          lowest = list[c];
       else if (list[c] > highest)
          highest = list[c];
    //*** Return result ***
    return make_pair(lowest, highest);
}
```

Activity 11.17

Start a new project called *TestPairTupleReturn* and implement *Extremes()* using the following code in `main()`:

```
int main()
{
   int nums[] = { 8,2,9,4,1,4,6,9,0,-1,7 };
   pair result = Extremes(nums, size(nums));
   cout << "Lowest number is " << result.first <<
   ↳" Highest number is " << result.second << endl;
}
```

Add a second function called *LimitsAndAverage()* which returns the lowest, highest and average values of a given int array. This function should return a tuple of the form `<int, double, int>`.

Modify `main()` to call the new function and display the values it returns. Use structured binding (see Chapter 9) to store the values returned in variables *low*, *high* and *av*.

Reference Parameters

Some functions are designed to modify the contents of a parameter. For example, let's say we wanted to write a function which added 1 to an `int` value passed as a parameter. We might try writing the program shown in FIG-11.7.

FIG-11.7

Attempting Parameter Updating

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
void AddOne(int);


int main()
{
   int num{ 22 };
   AddOne(num);
   cout << num << endl;
}

void AddOne(int v)
{
   v++;
}
```

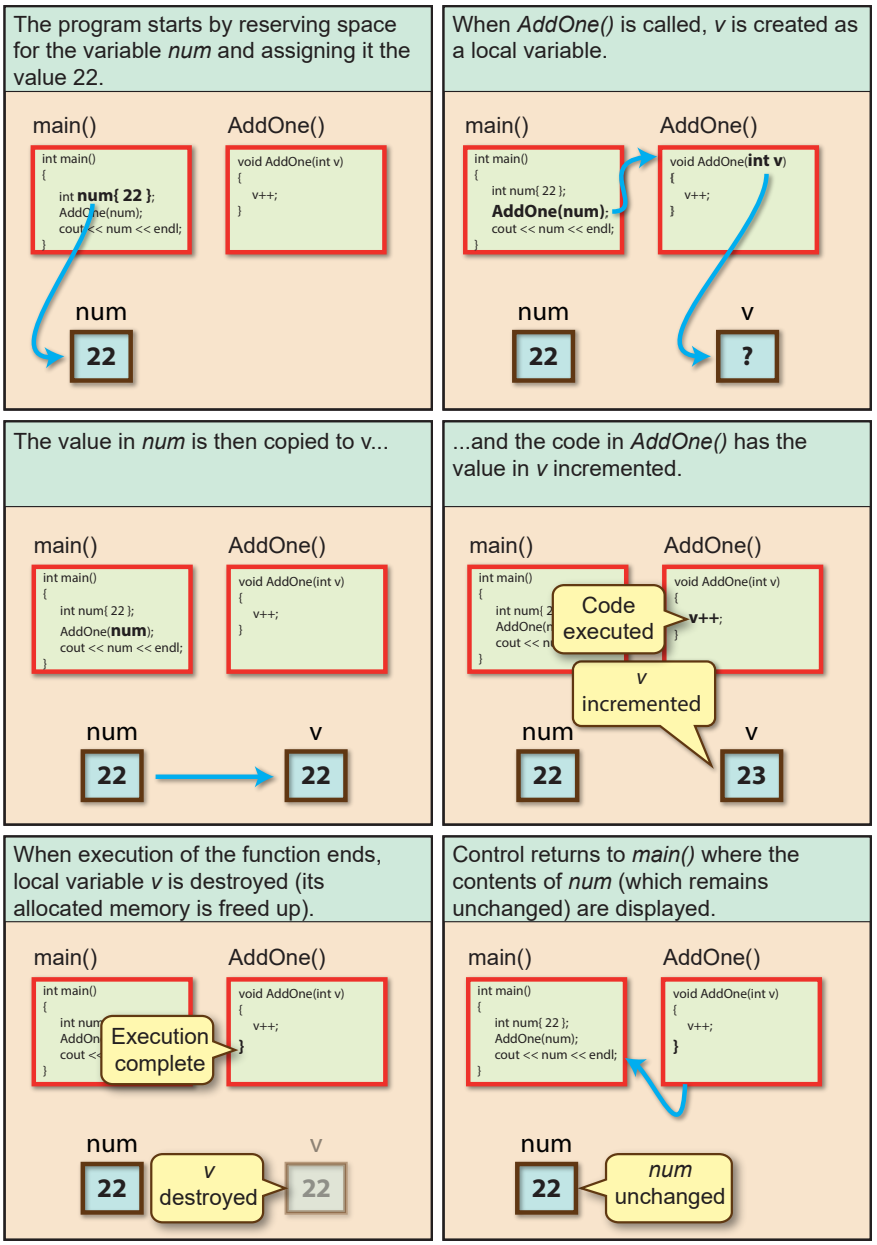
Activity 11.18

What value would be displayed by the code given above?

But, of course, this won't have the desired effect, because the function *AddOne()* works on a copy of the value in *num*, not on the contents of *num* itself (see FIG-11.8).

FIG-11.8

Pass By Value Mechanism



If we want the contents of the actual parameter to a function to be modified within the function, then we need to specify that parameter as a **reference parameter**.

We signify that a parameter is a reference parameter by adding an ampersand (`&`) after the parameter type.

In the cast of *AddOne()*, the function prototype and function heading are changed to:

```
void AddOne(int&);
void AddOne(int& v)
```

No other changes are required.

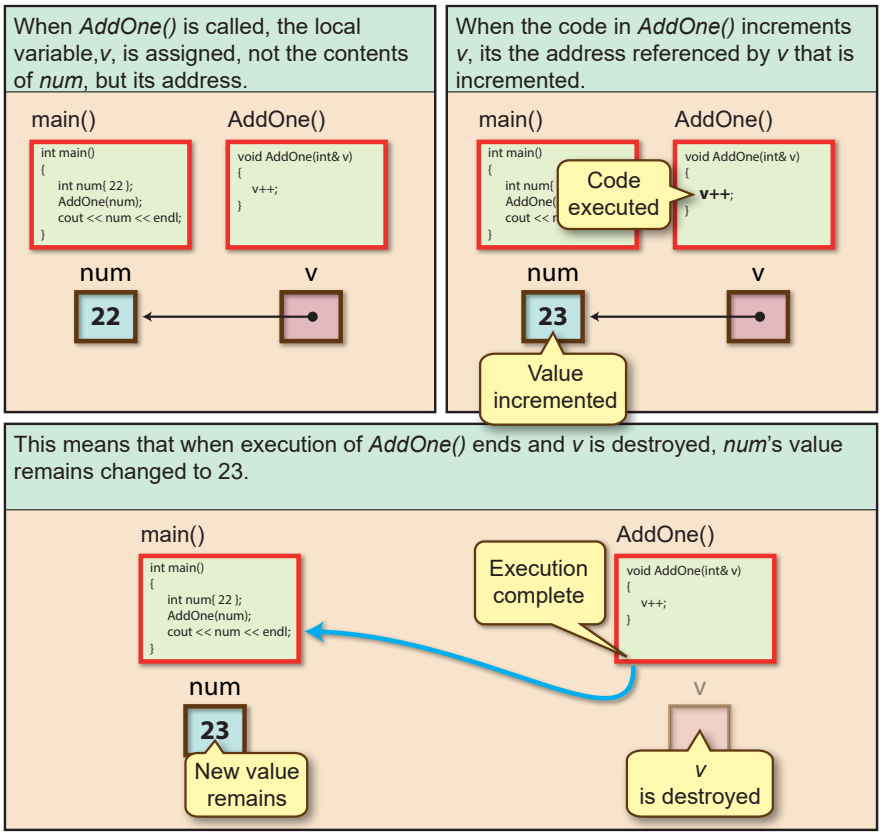
Activity 11.19

Start a new project called *RefParameter* and implement the code given in FIG-11.7, but making the parameter to *AddOne()* a reference parameter. What value is displayed?

How the reference parameter is handled by C++ is shown in FIG-11.9.

FIG-11.9

Pass By Reference Mechanism



Notice that, as we've seen before, reference parameters do not need to be dereferenced (using *) in order to access the location they point to and for the most part we can think of them as alternative names for the actual parameters.

Because we are passing a reference rather than a value, this type of parameter passing is known as **pass-by-reference**.

If we wish to keep things simple, we can think of a call-by-reference formal parameter as being an alternative name for the actual parameter used when the function is called.

Reference Parameters and Mini-Specs

When we want a parameter of a function to be updated by that function then we refer to that parameter as an **in-out parameter**. And this is exactly how we describe them in our mini-specs. For example, our mini-spec for *AddOne()* would be:

```
FUNCTION NAME   :   AddOne
PARAMETERS
   In-Out       :   v       : integer

PRE-CONDITION   :   true

DESCRIPTION     :   Adds 1 to v.
```

Activity 11.20

Start a new project called *TestSwap* which contains a function *Swap()* which conforms to the mini-spec below.

```
FUNCTION NAME   :   Swap
PARAMETERS
   In-Out       :   v1      : TimeType
                    v2      : TimeType

PRE-CONDITION   :   true

DESCRIPTION     :   Swaps the contents of v1 and v2.
```

TimeType is defined as:

```
struct TimeType
{
      int hrs;
      int mins;
      int secs;
};
```

The `main()` function in the program should read in two *TimeType* values and check that the contents of the variables holding the input values are correctly swapped by *Swap()*.

Why Use Reference Parameters?

In fact, wanting to update the value of a parameter is only one reason for using reference parameters. The second, and probably more important one is to remove the need to make an exact copy of a parameter within the function.

This second reason becomes significant when a function's parameter is a user-defined type or an array. Such variables can occupy large amounts of memory with record structures having numerous fields, and arrays possibly having hundreds or thousands of elements.

Not only does copying these take up large amounts of additional memory, but can also take significant time to copy all of the values held in the actual parameter variables to the formal parameter variables.

If we use reference variables for these type of parameters, then only a copy of the address of the original is created saving both time and space.

If we follow up this advice we should rewrite our *AddTimes()* function so that it makes use of reference parameters. This would change the signature of the function to read

```
TimeType AddTimes(TimeType &t1, TimeType &t2)
```

Activity 11.21

Modify the code for *AddTimes()* in the *TimeFunction* project so that it uses reference parameters.

Check that the function still operates as before.

Reference Parameters and Restrictions on Actual Parameters

Since a reference parameter expects to be supplied with the address of a variable, this means that the actual parameter used when the function is called cannot be a constant or expression.

For example, with our *DrawLine()* function which took details of the length and character used as pass-by-value parameters, we were free to call the function using actual parameters which were any of the following:

constants:
```
DrawLine(12, '>');
```

or expresssions:
```
DrawLine(3*6, static_cast<char>(31+34));
```

or variables:
```
DrawLine(size, letter);
```

However, with a function such as *AddOne()* which uses a call-by-reference parameter, we can only use the last of these options

variables:
```
AddOne(x);
```

const Parameters

The downside of using a reference parameter to save space and time is that those parameters can then have their contents changed within that function even when that's not the intention of the function.

But surely we wouldn't do that! Perhaps not deliberately – but it's easy to make mistakes when we're writing code.

Activity 11.22

Modify the code for *AddTimes()* in the *TimeFunction* by adding the line

```
t1.secs = 99;
```

immediately before the `return` statement. Observe the value displayed for *t1*.

Let's see how we can ensure this situation never happens.

When we create a function with *in* parameters – in other words, with parameters whose values should not be changed within the function – we can add the keyword `const` before the parameter type. This tells the compiler that the contents of that parameter cannot be changed within the function and will produce a compilation error if we try.

In the case of *AddTimes()* that would mean changing the heading to

```
TimeType AddTimes(const TimeType &t1, const TimeType &t2)
```

Activity 11.23

In the *TimeFunction* project change the prototype and heading of *AddTimes()* to declare the two parameters as `const` reference parameters. What happens when you try to compile the code?

Remove the line

```
t1.secs = 99;
```

and recompile the program.

In fact it's a good habit to use `const` on all *in* parameters, even those that are standard pass-by-value types, since it clearly documents the fact that those parameters should not have their values changed within the function.

Another advantage of using `const` when defining a reference or pointer parameter (pointer parameters are covered in the next section) is that it allows the actual parameter to be a constant or an expression. For example, if the program declares a function *f()* to have the prototype

```
void f(const double&);
```

then all of the following function calls would be valid:

```
f(3.8);
f(no1 * no2);
f(PI);
```

These examples assume *no1* and *no2* have previous been declared and that *PI* is a named constant.

We can see this in practice in the program shown in FIG-11.10.

FIG-11.10

Constants and Expressions an Actual Parameter with a const Reference

```
#include <iostream>
using namespace std;
//*** Function prototypes ***
void f(const double&);

int main()
{
   const double PI = 3.14159;
   double no1 = 42.0;
   double no2 = 1.8;
   f(3.8);
   f(no1*no2);
   f(PI);
}

void f(const double& v)
{
   cout << v << endl;
}
```

Activity 11.24

Start a new project called *ConstParam* and implement the code given above. Check that both calls to *F()* operate correctly.

Change the prototype and heading of *F()* removing the term `const`. What happens when you try to compile this version?

Pointer Parameters

Another option when passing parameters is to use pointers (**pass-by-pointer**). Like reference parameters, using pointer parameters allows us to change the contents of the values referenced by those pointer parameters. It also saves the space and time overhead created when using pass-by-value parameters.

In FIG-11.11 we can see two versions of the *Swap()* function. Our original uses reference parameters while the new one uses pointers.

FIG-11.11 Pass-By-Reference Vs Pass-By-Pointer

Reference Parameters

```
void Swap(TimeType &v1, TimeType &v2)
{
      TimeType temp{ v1 };
      v1 = v2;
      v2 = temp;
}
```

Pointer Parameters

```
void Swap(TimeType *v1, TimeType *v2)
{
      TimeType temp{ *v1 };
      *v1 = *v2;
      *v2 = temp;
}
```

We might be tempted to think that the actual pointer parameter must be a pointer when the formal parameter is a pointer. For example, when using the pointer parameter of *Swap()*, we might begin to try writing code in `main()` such as

```
TimeType *ptr1{&t1};
TimeType *ptr2{&t2);
Swap(ptr1,ptr2);
```

But remember, pointer variables are designed to hold addresses, so if we wanted to use the addresses of *t1* and *t2* as the parameters to the pointer version of *Swap()*, we need only supply their addresses as the actual parameters:

```
Swap(&t1, &t2);
```

Activity 11.25

Modify your *TestSwap* project so that it includes a second function called *Swap2()*. This function should perform the same operation as *Swap()* but with pointer parameters.

In `main()` replace the call to *Swap()* with a call to *Swap2()* and check that the result is the same as before.

As a general guideline reference parameters are to be preferred over pointer parameters since the function coding does not need to include dereferencing operations and there is less opportunity to get things wrong. Also note that pointers

may contain a null value and most functions using a pointer parameter would have to contain code to deal with that possibility.

The only circumstance under which pointer parameters might be required is when dynamically allocated space is being accessed.

If we do end up using pointer parameters, we have the option to include `const` (as we did with reference parameters) to ensure that the data pointed to cannot be changed. For example, *AddTimes()* could be written with the following prototype when using pointer parameters:

```
TimeType AddTimes(const TimeType*, const TimeType*);
```

Of course, pointers can have their own contents changed and if we did not want that to occur within a function, we would place the term `const` after the parameter type:

```
TimeType AddTimes(TimeType const*, TimeType const*);
```

Array Parameters

We can also use an array as a function parameter. The simplest way to do this is to give the array type, parameter name and a set of brackets. For example, let's assume we want to create a function, *Smallest()*, which returns the smallest value held in an `int` array. Our first attempt at a prototype and heading for such a function might be:

```
int Smallest(int[]);

int Smallest(int list[])
```

Notice that there is no attempt to specify the size of the array being passed to the function. This allows us to pass a one-dimensional array of any size when calling the function. But it also presents us with a problem: how do we detect the actual size of the array being used when the function is called?

You may recall that in Chapter 8 we discussed the `size()` function which will return the number of elements in an array and it would seem that we might use this within the code for *Smallest()*, perhaps in a line such as

```
for(int c = 0; c < size(list); c++)
```

to access every element of the array used as the actual parameter when the function is called.

Unfortunately, when an array is passed to a function, not all the information about the array becomes available within that function. This means that `size()` doesn't work when we attempt to use it on a function's array parameter within the function's own code.

Instead, we have to resort to adding a second parameter to our function which is used to specify the number of elements in the array:

```
int Smallest(int list[], int elements)
```

Now, when creating the code within the function we can write a line such as

```
for(int c = 0; c < elements; c++)
```

We're now ready to create the complete code for *Smallest()*:

```
//*** Returns the smallest value in an array ***
int Smallest(int list[], int elements)
{
   //*** Start with value in first cell as smallest ***
```

```
   int small = list[0];
   //*** FOR each remaining value DO ***
   for (int c = 1 ; c < elements; c++)
      // *** IF cell value smaller, save it ***
      if (list[c] < small)
         small = list[c];
   //*** Return smallest value found ***
   return small;
}
```

`main()` would call the function by passing the name of the array to be used and the number of elements it contains as the actual parameters:

```
int main()
{
   int nums[10] {5,7,2,9,5,4,3,6,4,9};
   cout << "Smallest value is " << Smallest(nums,size(nums))
   ↳<< endl;
}
```

Notice, that although we can't use `size()` within the code for *Smallest()*, we can use it to determine the value of the second parameter when *Smallest()* is called.



From Chapter 8:
Creating random numbers in the range 1 to 100 needs the lines

```
#include<time.h>
srand(static_cast
↳<int>(time(nullptr)));
```

and the expression

```
rand()%100+1;
```

Activity 11.26

Start a new project called *ArrayParams* which contains two functions called *Smallest()* and *Largest()* which respectively return the smallest and largest values in an `int` array.
Test the function by creating a 15-element array which contains randomly generated values between 1 and 100. The main function should display the complete contents of the array and then the highest value.

Modify the program so that the array contains 20 values.

Arrays and Pointer Parameters

A look at the call to *Smallest()* should give us a clue as to what is really going on when we use an array as a parameter to a function. We know from Chapter 10 that the name of an array is actually treated as a pointer to the block of memory set up by the array declaration. So, when we call up a function and use an actual parameter which is an array name as in the line

```
Smallest(nums,size(nums));
```

we must be passing a pointer to the function. And, indeed, that is exactly what we are doing. Specifying an array in a function's parameter list is really just a declaration that a pointer is to be passed to the function. We can even change the function's code to make this use of a pointer explicit:

```
//*** Returns smallest value in an array of size cells ***
void Smallest(int *list, int elements)
{
   //*** Start with value in first cell as smallest ***
   int small = list[0];
   //*** FOR each remaining value DO ***
   for (int c = 1 ; c < elements; c++)
      // *** IF cell value smaller, save it ***
      if (list[c] < small)
         small = list[c];
   //*** Return smallest value found ***
   return small;
}
```

Notice that the code within the function remains unchanged as does the function call.

We can even get away with leaving the prototype (with its brackets) unchanged, but it would cause less confusion to others who might read our code to change the prototype to

```
void Smaller(int*,int);
```

Activity 11.27

In *ArrayParams*, modify *Smallest()* so that its parameter *list* makes explicit use of a pointer and check that the program still operates as before.

When using an array as a parameter, we can again guard against unintentional changes to the contents of the array by adding `const` to the relevant parameter(s). For example, we could show that the code within function *Largest()* must not change the value of the array passed to it by defining the prototype as

```
int Largest(const int[], int);
```

and the function heading to

```
int Largest(const int list[], int elements)
```

Activity 11.28

In *ArrayParams*, modify *Largest()* so that the value of both its parameters cannot be changed within the function body.

Multi-Dimensional Array Parameters

When we want to pass a multi-dimensional array to a function we must specify all but the first dimension of the array. For example, let's assume that the function *GetHighestMarkInClass()* is a function designed to return the highest mark achieved by any student in the class and that each student in the class sits exactly five exams.

This would mean that the prototype of the function would be written as

```
int GetHighestMarkInClass(const int[][5], const int);
```

A complete program which makes use of this function is shown in FIG-11.12.

FIG-11.12

Multi-Dimensional Array Parameters

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
int GetHighestMarkInClass(const int[][5], const int);

int main()
{
   int exammarks[][5]{ {23,65,85,33,23},{78,72,81,90,55},
   ↳{88,93,95,90,79} };
   cout << "Highest mark in class was " <<
   ↳GetHighestMarkInClass(exammarks, 3) << endl;
}
```

FIG-11.12
(continued)

Multi-Dimensional Array Parameters

```
//*** Returns the highest mark achieved by any student ***
int GetHighestMarkInClass(const int marks[][5], const int stds)
{
   //*** If invalid number of students, exit ***
   if (stds < 1)
      return -1;
   //*** Find the highest ***
   int highest = marks[0][0];
   for (int student = 0; student < stds; student++)
      for (int exam = 0; exam < 5; exam++)
         if (marks[student][exam] > highest)
            highest = marks[student][exam];
   //*** Return highest mark found ***
   return highest;
}
```

Activity 11.29

Start a new project called *HighMark* that implements the code given above. What value is displayed by the program?

Returning an Array

Rather than just return the highest mark in the class, we might want to return the highest mark achieved by each student. This would require the function to return an array. If our new function is called *GetHighestMarkPerStudent()*, then it might try creating the following prototype:

```
int[] GetHighestMarkPerStudent(const int[][5], const int);
```

However, we're in for a disappointment. C++ doesn't allow us to return arrays in the way shown above! Instead, we have two alternative ways of solving our problem.

The first is to add the array that is to contain the result as a parameter to the function:

```
void GetHighestMarkPerStudent(const int[][5],const int,int[]);
```

The complete code for the new function is:

```
//*** Returns highest mark per student in highs ***
void GetHighestMarkPerStudent(const int marks[][5],
↳const int stds, int highs[])
{
   //*** If invalid number of students, exit ***
   if (stds < 1)
      return ;
   //*** Find the highest ***
   for (int student = 0; student < stds; student++)
   {
      //*** Find this student's highest mark ***
      int highest = marks[student][0];
      for (int exam = 0; exam < 5; exam++)
         if (marks[student][exam] > highest)
            highest = marks[student][exam];
      //*** Save this student's highest mark ***
      highs[student] = highest;
   }
}
```

Activity 11.30

Modify *HighMark*, so that, after displaying the overall highest mark, it also displays the highest mark for each student.

The second method of returning an array is to have the function return a pointer, making the prototype for *GetHighestMarkPerStudent()*

```
int* GetHighestMarkPerStudent(const int[][5], const int);
```

We'll discuss returning a pointer in the next section of this chapter.

Returning Reference and Pointer Values

We've seen that parameters can be passed to a function using three methods:

pass-by-value	which passes a copy of the actual parameter.
pass-by-reference	which passes the address of the actual parameter. The actual parameter does not need to be dereferenced to access the associated data.
pass-by-pointer	which also passes the address of the actual parameter, but the parameter needs to be dereferenced to access the associated data.

We can also use all three of these types when specifying a return type. So far all our functions have only returned value types. We'll compare these return types using different versions of the function *Smallest()*.

Returning a Reference

If we have *Smallest()* return a reference value, then we are, in effect, returning an address. However, we must be careful not to return the address of a variable which is local to function. To understand why this would cause a problem, let's make a first attempt at returning a reference to the minimum value found by *Smallest()* by changing its heading to:

```
int& Smallest(int *list, int elements)
```

We might do this if we wanted to return the address of *small* since it contains a copy of the smallest value found. In this case we would rewrite *Smallest()* as:

```
int& Smallest(int *list, int elements)
{
   //*** Start with value in first cell as smallest ***
   int small = list[0];
   //*** FOR each remaining value DO ***
   for (int c = 1; c < elements; c++)
      // *** IF cell value smaller, save it ***
      if (list[c] < small)
         small = list[c];
   //*** Return reference to small ***
   return small;
}
```

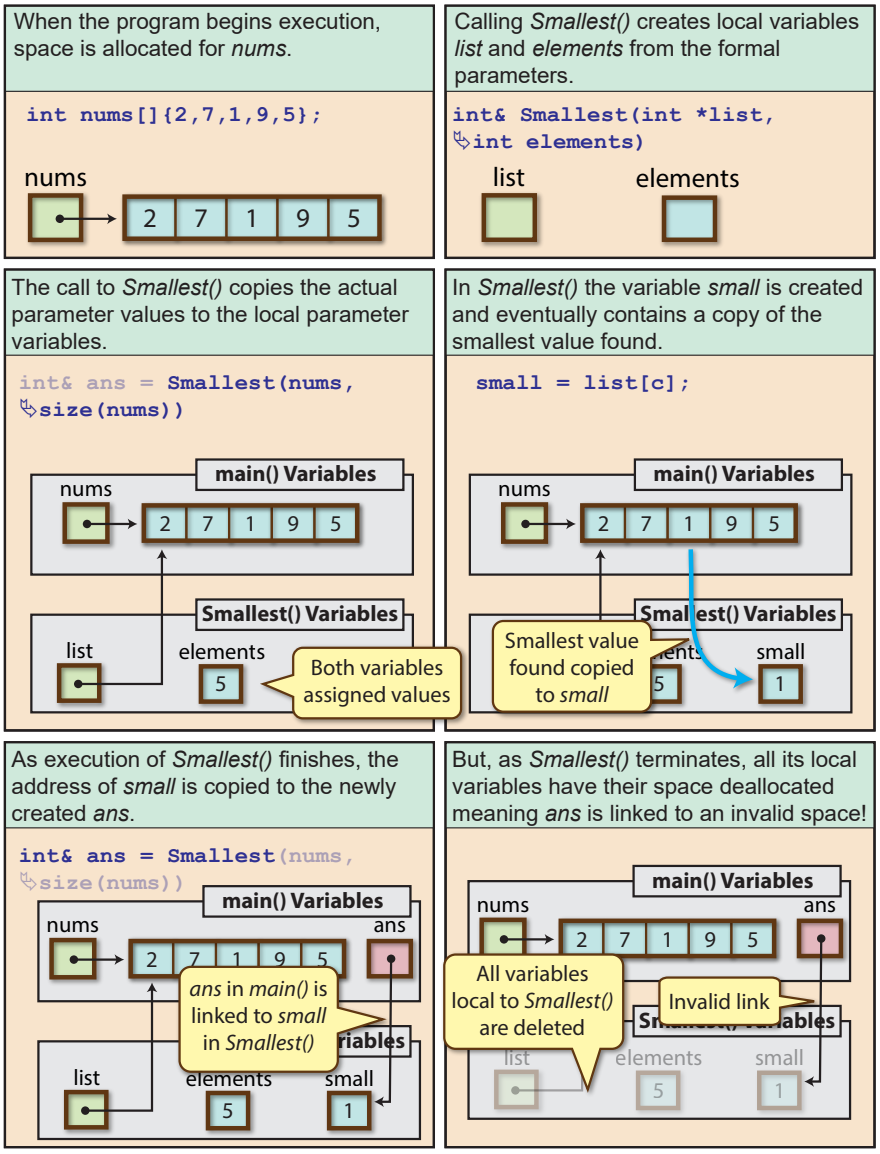
Notice that the only change to the code is in the function heading where we now indicate that an `int` reference value is being returned. The `return` statement at the end of the function remains unchanged. This is because the compiler automatically

realises that it is a reference to *small* that is to be returned, not its contents. FIG-11.13 shows the consequences of calling the above version of the *Smallest()* function using the lines

```
int nums[]{2,7,1,9,5};
int &ans = Smallest(nums,size(nums));
cout << ans << endl;
```

FIG-11.13

Referencing a Local Variable

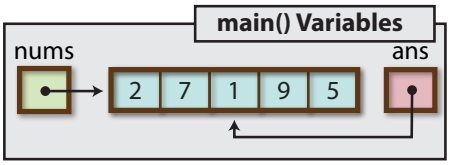


As we can see from the example above, a returned reference value should never contain the address of a local variable which no longer exists when the called function is complete. In fact, this is exactly equivalent to the same **dangling pointer** problem we encountered in Chapter 10.

The main reason for having a function return a reference value is to have it contain the address of one of the pass-by-reference or pass-by-pointer parameters to that function. To demonstrate this, let's create a second version of *Smallest()* – called *Smallest2()* – which returns the address of the cell in *nums* containing the smallest value. The result we're after is shown in FIG-11.14.

FIG-11.14

Correct use of a Returned Reference



The strategy of the new function is to save the position in the array at which the smallest value is found and then return the address of that cell. The functions code is:

```
int& Smallest2(int list[], int elements)
{
   //*** Assume cell zero contains the smallest ***
   int post = 0;
   //*** FOR each remaining position DO ***
   for (int c = 1; c < elements; c++)
      // *** If value at position smaller, save position ***
      if (list[c] < list[post])
         post = c;
   //*** Return address of smallest value found ***
   return list[post];
}
```

If we have `main()` display the value found at the returned address, it's code becomes

```
int main()
{
   int nums[15];
   //*** Generate random values in array ***
   srand(static_cast<int>(time(0)));
   for (int& v : nums)
      v = rand()%100+1;
   //*** Display all values in array ***
   for (auto v : nums)
      cout << v << "  ";
   cout << endl;
   //*** Find location of smallest value ***
   int& ans = Smallest2(nums, size(nums));
   //*** Display value found ***
   cout << ans << endl;
}
```

Activity 11.31

Modify *ArrayParams*, adding *Smallest2()* and changing `main()` to the code given above. Check that the smallest value is correctly displayed.

Modify `main()` so that the value at the returned address is changed to 99, then displays the new contents of *nums*.

Because *Smallest2()* returns an address, we can actually replace the code

```
int& ans = Smallest2(nums, size(nums));
ans = 99;
```

with the single line

```
Smallest2(nums, size(nums)) = 99;
```

However, since this looks rather strange when starting out in C++, it is probably best avoided at this stage.

Activity 11.32

We know that when an array is passed to a function it is best to add the term `const` *so that there is no possibility of the array's contents being changed within the function.*

Modify the prototype and heading of *Smallest2(),* adding the `const` term. Does the program still operate as before?

The change made in Activity 11.32 creates a compilation error.

The reason for this is that we've now said that contents of the array parameter cannot be changed. C++ takes this to mean that we cannot then create a non-const reference to an element of that array since we could then use that reference to change a value within the array. In order to keep the compiler happy, we must now specify *Smallest2()*'s return type as

```
const int&
```

This in turn means that *ans* must be defined as a `const int&` type and also that we can no longer use the line

```
ans = 99;
```

since changing a `const` value is not allowed!

Activity 11.33

In *ArrayParams*, modify *Smallest2()* so that the array parameter is defined as type `const int&`. Make any other changes necessary to have the program execute.

Returning a Pointer

Since we know a reference variable holds an address of some other variable, we can see that for the most part, there is little difference between pointers and reference variables.

The main difference is that we must explicitly dereference pointers in our code whereas the compiler handles the dereferencing of reference variables by itself without any intervention from the programmer.

It follows that there is very little difference when it comes to returning a pointer rather than a reference.

We must still look out for the same types of pitfalls (returning the address of a variable that no longer exists when the function exists).

Perhaps the only time returning a pointer might be considered useful is when the function has created dynamic space (using `new`) and we need to return the start address of that space. For example, let's say we want a function called *CreateRandomList()* which is implemented in accordance with the following mini-spec:

FUNCTION NAME	:	CreateRandomList
PARAMETERS		
In	:	count : integer
	:	min : integer
	:	max : integer
Out	:	list[count] :integer
PRE-CONDITION	:	count > 0 AND min <= max
DESCRIPTION	:	Returns a list of *count* random integer values in the range *min* to *max*.

Our code for this would be:

```
int* CreateRandomList(int count, int min, int max)
{
   if (count < 1 || min > max)
      return nullptr;
   //*** Seed numbers ***
   srand(static_cast<int>(time(0)));
   //*** Create space ***
   int* temp = new int[count];
   //*** Fill with random values ***
   for (int c = 0; c < count; c++)
      temp[c] = rand() % (max - min) + min;
   //*** Return address of list ***
   return temp;
}
```

However, it then becomes the responsibility of `main()` to delete the space allocated in *CreateRandomList()*.

Activity 11.34

Create a new project called *PointerReturn* which contains the code for *CreateRandomList()*.

In `main()` call *CreateRandomList()* to create a list containing 10 random numbers in the range 1 to 100. Also have `main()` display the contents of the list before deleting the space allocated to the list.

One problem is that `main()` *has no way of finding out how many values are in the list. We could overcome this by making the list one element longer and using the first element to store the number of elements in the list.* `main()` *can then interrogate the first element to discover the length of the list.*

Modify the code in *CreateRandomList()* so that the number of values in the list is stored in the first cell.

If a function returns a pointer, then there is always the danger that the pointer's value may be null. Our code should always test for a null value before going on to make use of the returned pointer.

For example, although *CreateRandomList()* will return a null pointer if the actual

parameters are invalid, `main()` has not tested for this possibility and therefore there will be unforseen problems if a null value is returned.

Activity 11.35

Try running *PointerReturn* with the first parameter in the call to *CreateRandomList()* set to -1. What happens when the program is run?

Modify the code in `main()` so that the message "*No list created*" is displayed if *CreateRandomList()* returns `nullptr`.

We also find that the C-style string-handling functions that come as part of the C++ standard library often return values of type `char*` to give the address of the string created (or modified) within that function.

Functions and Smart Pointers

We can easily solve the problem highlighted in the *PointerReturn* project where `main()` has the responsibility of deleting the space created within *CreateRandomList()*. If we rewrite the routine to return a `unique_ptr` type, by using a smart pointer the program will automatically deallocate the space created as soon as it is no longer referenced – in this case, that will be when the program terminates.

An updated version of *PointerReturn* which makes use of a `unique_ptr` is shown in FIG-11.15.

FIG-11.15

Returning a Smart Pointer

```
#include <iostream>
#include <time.h>>
using namespace std;

//*** Function prototypes ***
unique_ptr<int[]> CreateRandomList2(int, int, int);


int main()
{
   //*** Set up random list ***
   unique_ptr<int[]> list = CreateRandomList2(10, 10, 100);
   if (list == nullptr)
      cout << "No list created\n";
   else
   {
      //*** Display its contents ***
      for (int c = 1; c <= list[0]; c++)
         cout << list[c] << "  ";
      cout << endl;
      //*** Delete the space allocated ***
      //delete[] list;                        //NO LONGER REQUIRED
   }
}

unique_ptr<int[]> CreateRandomList(int count, int min, int max)
{
   //*** If invalid parameter, exit ***
   if (count < 1 || min > max)
      return nullptr;
```



FIG-11.15
(continued)

Returning a Smart Pointer

```
   //*** Seed numbers ***
   srand(static_cast<int>(time(0)));
   //*** Create space ***
   auto temp { make_unique<int[]>(count + 1) };
   //*** Record number of values in list ***
   temp[0] = count;
   //*** Fill with random values ***
   for (int c = 1; c <= count; c++)
      temp[c] = rand() % (max - min) + min;
   //*** Return address of list ***
   return temp;
}
```

Activity 11.36

Modify *PointerReturn* as shown above and check that it still operates correctly.

Activity 11.37

Reload project *HighMark* and create a second version of *GetHighestMarkPerStudent()* which returns the highest mark for each student using a smart pointer.

auto Return

It is possible to use `auto` as the return type for a function. For example, we could write the prototype of *Smallest()* as

```
auto Smallest(const int[], const int);
```

and the function heading would be

```
auto Smallest(const int list[], const int elements)
```

When we use this approach, C++ determines the actual return type by examining the `return` statements within the body of the code. For example, the code for *Smallest()* ends with the line

```
return small;
```

and *small* has previously been defined as type `int`, so C++ sets the return type for *Smallest()* to `int`.

If we want to use `auto` in conjunction with a function which returns a reference value, then we should include the ampersand. For example, we previously created another version of *Smallest(), Smallest2()*, which returned a reference to an `int`. The prototype for this function could be written as

```
auto& Smallest2(const int[], const int);
```

When returning a constant, we can use

```
const auto
```

as the return type

and for a constant reference:

```
const auto&
```

The problem with using `auto` as the return type is that reading the prototype declarations no longer offer an immediate understanding of what type is being returned.

Also, we must ensure that where a function body has several `return` statements that they all return values of the same type. For example, in the trivial function below

```
auto f()
{
   if (rand() % 100 < 50)
      return 1.0;
   else
      return -1;
}
```

C++ would fail to compile the code since either a `double` or `int` value can be returned. On the other hand, if the function heading had been

```
int f()
```

C++ would have accepted the code, converting 1.0 to an `int` value if that particular `return` statement was executed.

An alternative way of specifying the prototype and heading of a function is to use `auto` as the first term in the line and specify the actual return type after the parameter list. For example,

```
auto Smallest(const int[], const int)->int;

auto Smallest(const int list[], const int elements)->int
{
   ...
}
```

Note the return value at the end of the line must be preceded by `->`.

We'll see a use for this option in Chapter 25.

Function Attributes



Some of these attributes can be used on other elements of a program such as variables and classes but are most likely to be used with functions.

There are a few compiler instructions that can be added to functions. These instructions are termed **attributes** (not to be confused with class attributes which we will cover in a later chapter).

[[deprecated]]

When an old function has been replaced with a newer function with a different name, the old version is often retained for backward compatibility with existing programs. However, we won't want new programs to make use of the old function now that a replacement is available.

To have the compiler issue a "function deprecated"-style message we can add the term `[[deprecated]]` to the prototype of the old function. For example:

```
[[deprecated]] int f_old(int, int);
int f_new(int,int);
```

If a program makes a call to the old function with a line such as

```
int result = f_old(12,3);
```

a compiler warning will be issued. With some compilers, calling the deprecated

function may result in failure to compile. Visual Studio2019 issues the following compilation:

'f_old': was declared deprecated

The attribute can be extended to include a message as in the line

```
[[deprecated("Deprecated. f_old() may be unsafe. Try using
↳f_new()"]] int f_old(int, int);
```

A call to *f_old()* would then result in a message such as

'f_old': Deprecated. f_old() may be unsafe. Try using f_new()

[[nodiscard]]

Sometimes when we call a function we are not interested in the value it returns. For example, if we want a program to pause, we might call the function `getchar()` which pauses a program until a key is pressed and then returns the `int` value of the key. However, in that situation we're not interested in which key has been pressed and so rather than write

```
char ch = static_cast<int>(getchar());
```

we are much more likely to write

```
getchar();
```

which calls the function but makes no attempt to capture or use the value it returns.

But there are some functions where ignoring the returned value might mean we are most likely creating faulty code whose logic is suspect.

When we create a function that returns a value and that value is always going to be important, we can add the term `nodiscard` to the end of the function prototype:

```
[[nodiscard]]double f(int v);
```

We we do this, any program that calls the function without making use of the returned value causes the compiler to issue a warning. Note that the program will still run.

[[maybe_unused]]

If a program contains code for a function which is never called, the compiler may issue a warning (visual Studio doesn't). To stop any warning from appearing, we can add the attribute `[[maybe_unused]]`:

```
[[maybe_unused]]int f();
```

A function may be uncalled because it is called only during the testing phase of the program and not in the final version of the code after the app has been released.

Function Overheads

When we call a function there's a lot going on behind the scenes. For a start the program must record the address it should return to in the main program when the code within the function has finished executing. Various data elements such as parameter values and variables local to the function are also stored temporarily. All of this data is stored within an area of memory known as the **stack**.

When a called function is finished executing the space it used within the stack is freed up ready for use by subsequent functions.

Because of this background housekeeping, calling a function adds to the time taken to execute a program as well as the memory space required. However, normally this overhead is a price worth paying for the added clarity and reduced chance of errors.

Summary

- A function is a separate section of code designed to perform one specific activity.
- The details of a function should first be described in a mini-spec.
- A mini-spec must contain all necessary information about a routine since coding will be created using the mini-spec as a rigid guide.
- Three different types of parameters are allowed in a mini-spec specification:

IN parameters	the value is passed into the function but not returned.
OUT parameters	the value is calculated within the function and returned by the function.
IN-OUT parameters	the value is passed into the function, updated and then returned by the function.

- When a function's parameter must be within a restricted range of values, then that range should be specified as a pre-condition within the mini-spec.
- Functions have a return type, name and a parentheses-enclosed list of parameters.
- The first line of a function is known as the function header.
- The return type and the parameter list of a function without the function name is known as the function signature.
- The complete function code is known as the function definition.
- If functions are called earlier in the code than the position of their definition, then it is necessary to add a function prototype at the start of the program code.
- A function prototype contains the same details as a function heading but the parameter names are omitted and the line ends with a semicolon.
- The parameter names used when writing the function header are known as the formal parameters.
- A function is called by giving the name of the function followed by a parentheses-enclosed list of parameters.
- The parameters given in a function call are known as the actual parameters.
- Actual parameter values are copied to the formal parameters when function execution begins.
- Function parameters can be of any type including array, struct and pointer.
- Parameters which do not meet a function’s pre-conditions should normally cause the function to terminate execution.
- Normally a function returns zero or one values.
- When a function returns a single value, the function definition must contain the keyword `return` followed by the value to be returned by the function.
- Functions designed to return a value which terminate early should return an

error value.

- Parameters are passed to a function in one of three ways:

  pass-by-value
  pass-by-reference
  pass-by-pointer

- Pass-by-value parameters copy the value in the actual parameter to the formal parameter before executing the routine.
- Pass-by-reference parameters copy the address of the actual parameter to the formal parameter.
- A standard pass-by-reference parameter can only accept a named variable as the actual parameter.
- A const pass-by-reference parameter can accept a named variable, constant or expression as the actual parameter.
- Pass-by-pointer parameters, like pass-by-reference, pass an address to the formal parameter, but the formal parameter must be dereferenced to access the data.
- When code within a function uses a pass-by-reference parameter it is the contents of the address specified in the formal parameter that is accessed/ modified, not the contents of the formal parameter.
- Using pass-by-reference allows multiple values to be returned by a function.
- Mini-spec IN parameters are implemented using call-by-value parameters.
- Mini-spec OUT and IN-OUT parameters are implemented using call-by-reference parameters.
- When the storage requirements for an IN parameter is large, the parameter may be declared as call-by-reference in order to save time and space.
- To ensure that the contents of an IN parameter which has been declared as call-by-reference are not modified, the parameter can be declared using the term **`const`**.
- Use **`[[deprecated]]`** to mark a function as deprecated.
- Use **`[[nodiscard]]`** to warn against ignoring the value returned by a function.
- Use **`[[maybe_unused]]`** to warn that a function may not be called.

Testing Functions

Introduction

In previous chapters we've discussed how to test a program based on the code it contains, with different testing strategies for sequence, selection and iteration structures.

The main problem with this approach is that it creates test data to test what we've written and not what we have forgotten to write! For example, if our thinking has been a bit faulty and we've omitted a necessary `if` statement from our code then, since the statement doesn't exist, we won't create a test for the omitted condition.

Black Box Testing

We can begin to create the test data we are going to use on a function's code as soon as the mini-spec has been created.

Although it may seem too early to start thinking about testing when we have no idea of the code that is going to be produced for a function, in fact, doing so offers us opportunities to create test data that may not occur to us when examining the actual code.

Let's assume we have decided to buy a used camera. We don't need to know how the innards of camera operate in order to test it. We know what operations the camera is designed to perform, so we can test it by taking pictures and movies, checking that these appear correctly in the viewing screen and are correctly focused and exposed. We might also check that the images can be transferred to another device and check that the battery charges correctly.

This same strategy allows us to create test data for a function. For example, let's say we have the following mini-spec for a function called *IsLeapYear*():

The UK and its dominions (including the American colonies) began using the current Gregorian calendar in 1752 but skipped 11 days in September of that year, so it was 1753 before we had a full 365 day Gregorian year.

```
FUNCTION NAME   :   IsLeapYear
PARAMETERS
    In          :   year   : integer
    Out         :   result : integer
PRE-CONDITION   :   None
DESCRIPTION     :   result is set to 1 if year is a leap year, and zero if
                    year is not a leap year.
                    If year is less than 1753, -1 is returned.
                    Leap years are those years which are divisible by
                    4 but not 100 or years that are divisible by 400.
```

Activity 11.38

Using the approach described above, create a set of test values for the *IsLeapYear()* function, including the expected results.

Because we create these test values without being able to see a function's "inner workings" (that is, its code), we refer to the tests constructed in this way as **black box testing.**

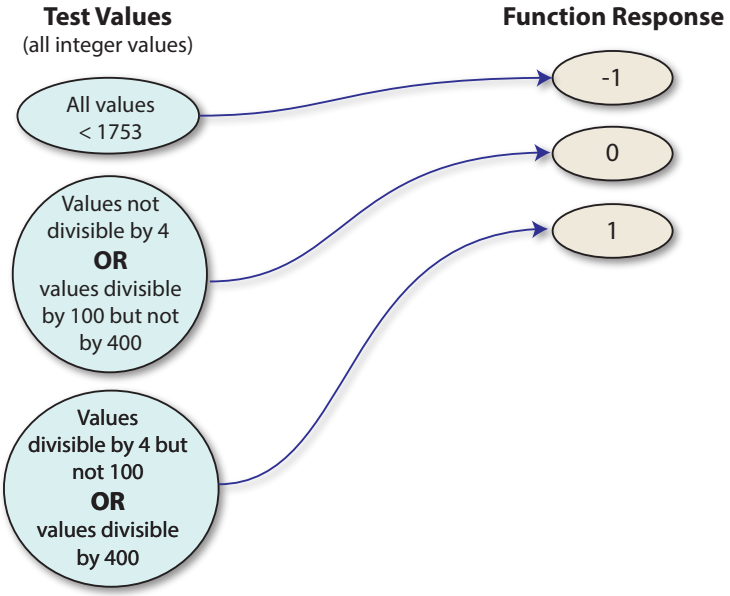
Equivalence Class Partitioning

If we choose our data wisely, we can minimise the number of runs we need to conduct when performing black box testing. To achieve this we can split all the possible test values we might use into groups in such a way that all the members of a group create the same result. This is known as **equivalence class partitioning**.

To start with, we need to identify the possible outputs from the function under test. In the case of *IsLeapYear()*, we have three possible responses: -1, 0, 1. This gives us the partitioning shown in FIG-11.16.

FIG-11.16

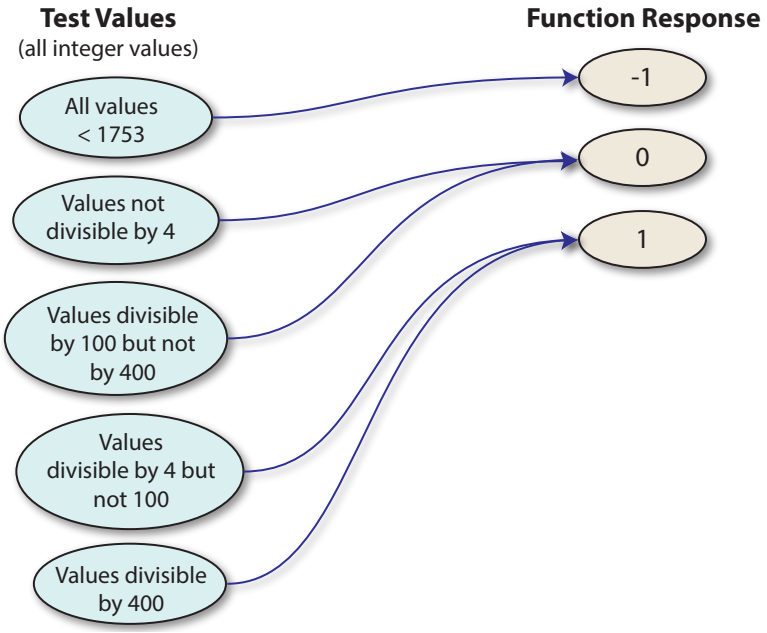
Equivalence Classes



Even though they elicit the same response, equivalence classes containing AND or OR descriptions are best subdivided (see FIG-11.17).

FIG-11.17

Equivalence Classes Subdivided



Activity 11.39

If necessary, adjust your set of test values for the *IsLeapYear()* function, so that one value from each equivalence class is selected.

Boundary Testing

Most of the equivalence class groupings in the black box testing of *IsLeapYear()* consist of non-contiguous values. That is to say, a value such as 2008 belongs to one group while the values 2007 and 2009 belong to a different class. But the equivalence classes of some functions are contiguous.

Let's assume we want to write a function called *Classify()* which returns a *char* value classifying a player's score in a game. The classifications are shown in FIG-11.18.

FIG11.18

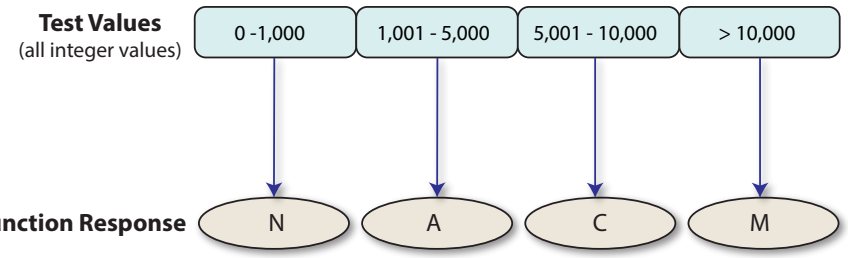
Classify() Rules

Score	Classification	Char Returned
0 - 1,000	Novice	N
1001 - 5,000	Apprentice	A
5001 - 10,000	Craftsman	C
>10,001	Master	M

In this situation, our equivalence classes can be drawn as shown in FIG-11.19.

FIG-11.19

Classify() Equivalence Classes



Now we have created classes where we have values on the boundary between one class and another. For example, 1,000 is on the boundary of our first class, being the highest number before we move into the next class. On the other hand, 1,001 is also a boundary value, being the lowest number in the second class before we move back into the first class. Other boundary values are 5,000, 5,001, 10,000 and 10,001.

Boundary values turn out to be excellent choices for test data since they show up the following type of errors. Imagine our *Classify()* function's heading is

```
char Classify(const int score)
```

and it starts with the lines

```
char result {'N'};
if (score > 1001 && score < 5000)
   result = 'A';
```

It is only by using the boundary values of 1001 and 5000 – which will both return the wrong classification – that we would discover the error in our code's `if` statement, which should have been coded as

```
if (score >= 1001 && score <= 5000)
```

White Box Testing

When we create test data based on a function's code – as we did in our earlier programs – this is known as **white box testing** or **glass box testing**. It gets the name from the idea that we are examining the "inner workings" of the program or function in order to create our test code.

The code for *IsLeapYear()* is

```
//*** Returns 1 if year is a leap year ***
//*** Returns zero if not a leap year  ***
//*** Returns -1 if year before 1753   ***
int IsLeapYear(int yr)
{
   int result;
   if (yr < 1753)
      result = -1;
   if ((yr % 4 == 0 && yr % 100 != 0) || (yr % 400 == 0))
      result = 1;
   else
      result = 0;
   return result;
}
```

We can see that we have a multi-way selection structure (created by the nested `if` structure) that requires testing.

The first condition is `year < 1753`, for which we need two test values: one to make the condition *true*, the other to make it *false*. The second `if` has a more complex condition

```
(yr % 4 == 0 && yr % 100 != 0) || (yr % 400 == 0)
```

and for this we need to create as many combinations of *true* and *false* as is possible. The theoretical combination is shown in FIG-11.20 (impossible combinations in grey).

FIG-11.20

IsLeapYear() Test Combinations

yr%4 == 0	yr%100 != 0	yr%400 == 0
false	false	false
false	false	true
false	true	false
false	true	true
true	false	false
true	false	true
true	true	false
true	true	true

Combining Test Requirements

We now have three strategies for creating test data: black box testing, boundary testing, and white box testing.

Where possible, we should employ all three approaches to help us select the actual test values we intend to use.

In the case of the function *Classify()*, using the black box approach would mean we

would create a test value for each classification (see FIG-11.21).

FIG-11.21

Classify() Equivalence Class Test Data

Test Data For *Classify()*	
Test Value	**Expected Result**
score = 500	N
score = 1500	A
score = 7000	C
score = 12000	M

The boundary value approach would have us add several more values (see FIG-11.22).

FIG-11.22

Classify() Equivalence Class Test Data with Boundary Values

Test Data For *Classify()*	
Test Value	**Expected Result**
score = 500	N
score = 1000	N
score = 1001	A
score = 1500	A
score = 5000	A
score = 5001	C
score = 7000	M
score = 10000	C
score = 10001	M
score = 12000	M

To create the final set of test values, we need to examine the function's code:

```
char Classify(const int score)
{
   char result;
   if (score <= 1000)
      result = 'N';
   else if (score <= 5000)
      result = 'A';
   else if (score <= 10000)
      result = 'C';
   else
      result = 'M';
   return result;
}
```

Looking at the code we can see that there are three mutually exclusive conditions to be tested. We need test values which will make each condition *true* during one test run and *false* during another. But, if we look at the test values we created with our black box and boundary approaches, we can see that we already have test values which satisfy these criteria, so there is no need to add any new values.

Test Drivers

Imagine we have to replace a light bulb in a hard-to-reach location. The smart thing to do would be to test the new light bulb before going to all the trouble of putting it in place. Similar thinking often makes it a good idea to test a function separately before we add it to the program for which it is designed.

If we were to test *Classify()* by inserting it into the game, we would have to play through the game before the game logic made a call to the routine we were trying to validate. Not only that, but it might be difficult to achieve exactly the score matching the chosen test value.

A more controlled approach would be to write a short program, containing the function to be tested and any code necessary to call the routine, supply parameter values, and display its returned value. Programs of this type are known as **test drivers**. A test driver for *Classify()* is shown in FIG-11.23.

FIG-11.23

Using a Test Driver

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
char Classify(const int);

int main()
{
   int testvalue;    //test score
   int count{ 1 };   //Current test number

   //*** Get test value ***
   cout << "Enter test value 1 : ";
   cin >> testvalue;
   while (testvalue != -1)
   {
      //*** Call function under test and display result ***
      cout << "Test value " << count << " : " << testvalue
      << " Returned by Classify() : " << Classify(testvalue)
      << endl;
      //*** Get next test value ***
      count++;
      cout << "Enter test value " << count << " : ";
      cin >> testvalue;
   }
   cout << "Test terminated\n";
}


//*** Classifies a game score ***
char Classify(const int score)
{
   char result;
   if (score <= 1000)
      result = 'N';
   else if (score <= 5000)
      result = 'A';
   else if (score <= 10000)
      result = 'C';
   else
      result = 'M';
   return result;
}
```

Activity 11.40

Start a new project *ClassifyTestDriver* and implement the code given in FIG-11.23. Using the test data given in FIG-11.22, test *Classify()* to check that it produces the expected results.

Function Complexity

The more complex a function's code is, the more likely it is to contain an error and the more difficult and time-consuming it is to test.

Our first thoughts might be that the more lines there are within a function, the more complex it is. However, this is not true. For example, a function might contain 20 `cout` lines used to display a rectangle of asterisks on the screen. This is not a complex function. If we run it and it produces the expected rectangle, then our testing is done.

On the other hand, a 20 line function containing nested `if`s and `while` loops, each of which use compound conditions, is a complex function.

As a rule-of-thumb, a function's complexity is given as

the number of simple conditions used + 1

For example, the code for *IsValidDate()* has a complexity of 9 (see FIG-11.24).

FIG-11.24

Classify() - Function Complexity

```
bool IsValidDate(DateType dt)
{
        bool result{ true };
        //** If invalid year, exit **
        if (dt.year <= 1752)                                          +1
                result = false;;
        //** If invalid month, exit **
        if (dt.month < 1 || dt.month > 12)                            +1
                result = false;
        //** If invalid day, exit **
        int daysinmonth[]{ 0,31,28,31,30,31,30,31,31,30,31,30,31 };
        //* If leap year and 1 to days in Feb ***
        if ((dt.year % 4 == 0 && dt.year % 100 != 0) || (dt.year % 400 == 0))   +2
                daysinmonth[2]++;
        if (dt.day < 1 || dt.day > daysinmonth[dt.month])             +1
                result = false;
        //*** Return date's validity ***
        return result;
}
```

Complexity = 9

As we can see in the example above, in compound conditions such as

```
if(dt.day < 1 || dt.day > daysinmonth[dt.month])
```

each simple condition is counted separately, giving the above line a count of 2.

Each value in each `case` of a `select` statement also adds 1 to the complexity count. For example, the structure

```
select(num)
   case 1, 3, 5, 7:
         ...
   endcase
   case 2, 4, 6, 8:
         ...
   endcase
endselect
```

has a complexity of 8.

The generally accepted rule is that functions with a complexity of 10 or less are acceptable and those above that figure should be split into two or more separate functions if practical.

Of course, this is only a rough guide, we can easily create functions with a complexity greater than 10 which would be simple to test and should not be modified in any way. For example, if function *DisplayMonth()* takes a number between 1 and 12 and returns the name of the corresponding month, it would have the following code:

```
void DisplayMonth(int m)
{
   if (m < 1 || m > 12)
      return;
   switch (m)
   {
   case 1:
      cout << "January\n";
      break;
   case 2:
      cout << "February\n";
      break;
   case 3:
      cout << "March\n";
      break;
   ...
   ...
   case 12:
      cout << "December\n";
      break;
   }
}
```

This function will have a complexity value of 15 (`if` = 2, `select` = 12, +1) and yet is not a truly complex piece of code.

Activity 11.41

Give the complexity value of the following function:

```
int Largest(const int list[], const int elements)
{
      int large = list[0];
      for (int c = 1; c < elements; c++)
            if (list[c] > large)
                  large = list[c];
      return large;
}
```

Summary

- Black box testing is the creation of test data based on the description of what operations a function is expected to perform and without reference to the function’s code.
- Boundary testing is the selection of test data based on the values that lie on the boundary between one response and another returned by a function.
- White box testing is the selection of test data based on the structure of a function’s code.
- A combination of all three test methods creates a good set of test data values.
- A test driver is a main program written specifically to test a function.
- A function's complexity has some relation to the number of conditions in the function's code.

Solutions

Activity 11.1

Other details we might mention include the colour, size and font to be used when displaying the asterisks.

Activity 11.2

No solution required.

Activity 11.3

Modified code for *UsingFunctions*:

```
#include <iostream>
using namespace std;

//*** Draws a line of 10 asterisks ***
void DrawLine()
{
  cout << "********\n";
}


int main()
{
  DrawLine();
  DrawLine();
}
```

Activity 11.4

Modified code for *UsingFunctions*:

```
#include <iostream>
using namespace std;

int main()
{
  DrawLine();
  DrawLine();
}

//*** Draws a line of 10 asterisks ***
void DrawLine()
{
  cout << "********\n";
}
```

This version will not compile since the compiler finds the calls to *DrawLine()* before it encounters that function's definition.

UsingFunctions with the function prototype included:

```
#include <iostream>
using namespace std;

//*** Function prototype ***
void DrawLine();

int main()
{
  DrawLine();
  DrawLine();
}

//*** Draws a line of 10 asterisks ***
void DrawLine()
{
  cout << "********\n";
}
```

Activity 11.5

Modified code for *UsingFunctions*:

```
#include <iostream>
using namespace std;

//*** Function prototype ***
void DrawLine();

int main()
{
  DrawLine();
  DrawLine();
}

//*** Draws a line of 10 asterisks ***
void DrawLine()
{
  // *** Draw the line ***
  for (int count = 1 ; count <= 10; count++)
    cout << '*';
  cout<<'\n';
}
```

Activity 11.6

No solution required.

Activity 11.7

No solution required.

Activity 11.8

No solution required.

Activity 11.9

No solution required.

Activity 11.10

Code for *TimeFunction*:

```
#include <iostream>
using namespace std;

//*** User defined types ***
struct TimeType
{
  int hrs;
  int mins;
  int secs;
};

//*** Function prototypes ***
void AddTimes(TimeType t1, TimeType t2);

void main()
{
  TimeType tm1, tm2;

  //*** Input times ***
  cout << "Enter first time (hh:mm:ss) : ";
  scanf("%d:%d:%d", &tm1.hrs, &tm1.mins, &tm1.secs);
  cout << "Enter second time (hh:mm:ss) : ";
  scanf("%d:%d:%d", &tm2.hrs, &tm2.mins, &tm2.secs);
  AddTimes(tm1, tm2);
}

void AddTimes(TimeType t1, TimeType t2)
{
  int total_secs = (t1.hrs + t2.hrs) * 60 * 60 +
  ↳(t1.mins + t2.mins) * 60 + (t1.secs + t2.secs);
  TimeType result;
  result.hrs = total_secs / 3600;
  result.mins = (total_secs - result.hrs * 3600)/60;
  result.secs = total_secs % 60;
  cout << t1.hrs << ':' << t1.mins << ':' << t1.secs
  ↳<< " + " << t2.hrs << ':' << t2.mins << ':'
  ↳<< t2.secs << " = " << result.hrs << ':'
  ↳<< result.mins << ':' << result.secs << endl;
}
```

Activity 11.11

We could be brief and simply write

Pre-condition: t1 is a valid time AND t2 is a valid time.

But a better description would be

Pre-condition: t1.hrs >= 0 AND t1.mins >= 0 AND t1.mins <= 59 AND t1.secs >= 0 AND t1.secs <= 59 AND t2.hrs >= 0 AND t2.mins >= 0 AND t2.mins <= 59 AND t2.secs >= 0 AND t2.secs <= 59

Activity 11.12

Modified code for *AddTimes()* in *TimeFunction*:

```
void AddTimes(TimeType t1, TimeType t2)
{
  //*** If either time invalid, exit ***
  if (t1.hrs < 0 || t1.mins < 0 || t1.mins > 59 ||
  ↳t1.secs < 0 || t1.secs > 59 || t2.hrs < 0 ||
  ↳t2.mins < 0 || t2.mins > 59 || t2.secs < 0 ||
  ↳t2.secs > 59)
    return;
  //*** Calculate total time ***
  int total_secs = (t1.hrs + t2.hrs) * 60 * 60 +
  ↳(t1.mins + t2.mins) * 60 + (t1.secs + t2.secs);
  TimeType result;
  result.hrs = total_secs / 3600;
  result.mins = (total_secs - result.hrs * 3600)/60;
  result.secs = total_secs % 60;
  cout << t1.hrs << ':' << t1.mins << ':' << t1.secs
  ↳<< " + " << t2.hrs << ':' << t2.mins << ':'
  ↳<< t2.secs << " = " << result.hrs << ':'
  ↳<< result.mins << ':' << result.secs << endl;
}
```

If either time is invalid, the function will exit without displaying any value.

An alternative way of creating the parameter guard is to NOT the original pre-condition:

```
if (!( t1.hrs >= 0 && t1.mins >= 0 && t1.mins <= 59
↳&& t1.secs >= 0 && t1.secs <= 59 &&  t2.hrs >= 0
↳&& t2.mins >= 0 && t2.mins <= 59 && t2.secs >= 0
↳&& t2.secs <= 59))
```

Activity 11.13

Code for *SumFunction*:

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
int SumIntegers(int);

int main()
{
  int num;

  cout << "Enter a number : ";
  cin >> num;
  while (num < 1 || num > 20)
  {
    cout << "Value must be between 1 and 20\n";
    cout << "PLEASE RE-ENTER : ";
    cin >> num;
  }
  cout << "The sum of all integer values between 1
  ↳and " << num << " is " << SumIntegers(num) <<
  ↳endl;
}


//*** Returns the sum of all integers between 1 and
value ***
int SumIntegers(int value)
{
  int result = 0;
  for (int c = 1; c <= value; c++)
    result += c;
  return result;
}
```

Activity 11.14

Modified code for *DOWFunction*:

```
#include <iostream>
using namespace std;

//*** User defined types ***
struct DateType
{
  int day;
  int month;
  int year;
};

//*** Function prototypes ***
bool IsValidDate(DateType dt);
int GetDayOfWeek(DateType dt);


void main()
{
  //*** Get date ***
  DateType date;
  cout << "Enter date in the format dd/mm/yyyyy : ";
  scanf("%d / %d / %d", &date.day, &date.month,
  ↳&date.year);
  //*** Display day of week  ***
  cout << date.day << '/' << date.month << '/' <<
  ↳date.year << " fell on a " << GetDayOfWeek(date);
}

//*** Displays day of the week on which dt fell ***
int GetDayOfWeek(DateType dt)
{
  if (!IsValidDate(dt))
    return;
  //*** Calculate modified date ***
  int M = (dt.month + 9) % 12 + 1;
  int modifiedyear = dt.year - M / 11;
  int C = modifiedyear / 100;
  int Y = modifiedyear % 100;
  //*** Return day of week ***
  return ((static_cast<int>(2.6 * M - 0.2) + dt.day
  ↳+ Y + Y / 4 + C / 4 - 2 * C) % 7 + 7) % 7;
}

//*** Check date is valid ***
bool IsValidDate(DateType dt)
{
  bool result{ true };
  //** If invalid year, exit **
  if (dt.year <= 1752)
    result = false;;
  //** If invalid month, exit **
  if (dt.month < 1 || dt.month > 12)
    result = false;
  //** If invalid day, exit **
  int daysinmonth[]{ 0,31,28,31,30,31,30,31,31,30,
  ↳31,30,31 };
  //* If leap year and 1 to days in Feb ***
  if ((dt.year % 4 == 0 && dt.year % 100 != 0) ||
  ↳(dt.year % 400 == 0))
    daysinmonth[2]++;
  if (dt.day < 1 || dt.day > daysinmonth[dt.month])
    result = false;
  //*** Return date's validity ***
  return result;
}
```

Activity 11.15

Modified code for *DOWFunction*:

```
#include <iostream>
using namespace std;

//*** User defined types ***
struct DateType
{
  int day;
  int month;
  int year;
};

//*** Function prototype ***
bool IsValidDate(DateType dt);
int GetDayOfWeek(DateType dt);


void main()
{
  //*** Get date ***
  DateType date;
  cout << "Enter date in the format dd/mm/yyyyy : ";
  scanf("%d / %d / %d", &date.day, &date.month,
  ↳&date.year);
  //*** Display day of week  ***
```

```
  int dow = GetDayOfWeek(date);
  if (dow != -1)
    cout << date.day << '/' << date.month << '/' <<
    ↳date.year << " fell on a " << dow << endl;
  else
    cout << "Invalid date\n";
}

//*** Displays day of the week on which dt fell ***
int GetDayOfWeek(DateType dt)
{
  if (!IsValidDate(dt))
    return -1;
  //*** Calculate modified date ***
  int M = (dt.month + 9) % 12 + 1;
  int modifiedyear = dt.year - M / 11;
  int C = modifiedyear / 100;
  int Y = modifiedyear % 100;
  //*** Return day of week ***
  return ((static_cast<int>(2.6 * M - 0.2) +
  ↳dt.day + Y + Y / 4 + C / 4 - 2 * C) % 7 + 7)%7;
}

//*** Check date is valid ***
bool IsValidDate(DateType dt)
{
  bool result{ true };
  //** If invalid year, exit **
  if (dt.year <= 1752)
    result = false;;
  //** If invalid month, exit **
  if (dt.month < 1 || dt.month > 12)
    result = false;
  //** If invalid day, exit **
  int daysinmonth[]{ 0,31,28,31,30,31,30,31,31,30,
  ↳31,30,31 };
  //* If leap year and 1 to days in Feb ***
  if ((dt.year % 4 == 0 && dt.year % 100 != 0) ||
  ↳(dt.year % 400 == 0))
    daysinmonth[2]++;
  if (dt.day < 1 || dt.day > daysinmonth[dt.month])
    result = false;
  //*** Return date's validity ***
  return result;
}
```

Activity 11.16

Modified code for *TimeFunction*:

```
#include <iostream>
using namespace std;

//*** User defined types ***
struct TimeType
{
  int hrs;
  int mins;
  int secs;
};

//*** Function prototypes ***
TimeType AddTimes(TimeType t1, TimeType t2);


void main()
{
  TimeType tm1, tm2;

  //*** Input times ***
  cout << "Enter first time (hh:mm:ss) : ";
  scanf("%d:%d:%d", &tm1.hrs, &tm1.mins, &tm1.secs);
  cout << "Enter second time (hh:mm:ss) : ";
  scanf("%d:%d:%d", &tm2.hrs, &tm2.mins, &tm2.secs);
  TimeType answer = AddTimes(tm1, tm2);
  if (answer.hrs != -1)
    cout << tm1.hrs << ':' << tm1.mins << ':'
    ↳<< tm1.secs << " + " << tm2.hrs << ':'
    ↳<< tm2.mins << ':' << tm2.secs << " = "
    ↳<< answer.hrs << ':' << answer.mins << ':'
    ↳<< answer.secs << endl;
  else
    cout << "One or more of the times given was
    ↳invalid\n";
}

TimeType AddTimes(TimeType t1, TimeType t2)
{
  TimeType result {-1,-1,-1};
  //*** If either time invalid, exit ***
  if (t1.hrs < 0 || t1.mins < 0 || t1.mins > 59 ||
  t1.secs < 0 || t1.secs > 59||t2.hrs < 0 ||
  t2.mins < 0 || t2.mins > 59 || t2.secs < 0 ||
  t2.secs > 59)
    return result;
  //*** Calculate total time ***
  int total_secs = (t1.hrs + t2.hrs) * 60 * 60 +
  (t1.mins + t2.mins) * 60 + (t1.secs + t2.secs);
  result.hrs = total_secs / 3600;
  result.mins = (total_secs - result.hrs * 3600)/60;
  result.secs = total_secs % 60;
  return result;
}
```

Activity 11.17

Code for *TestPairTupleReturn* (version 1):

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
pair<int,int> Extremes(int*, int);

int main()
{
  int nums[] = { 8,2,9,4,1,4,6,9,0,-1,7 };
  pair result = Extremes(nums, size(nums));
  cout << "Lowest number is " << result.first <<
  ↳" Highest number is " << result.second << endl;
}


//*** Return lowest and highest number in list ***
pair<int, int> Extremes(int* list, int sz)
{
  //*** Set lowest and highest ***
  int lowest = list[0];
  int highest = list[0];
  //*** Search for lower and higher values ***
  for (int c = 1; c < sz; c++)
    if (list[c] < lowest)
      lowest = list[c];
    else if (list[c] > highest)
      highest = list[c];
  //*** Return result ***
  return make_pair(lowest, highest);
}
```

Code for *TestPairTupleReturn* (version 2):

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
pair<int,int> Extremes(int*, int);
tuple<int, int, double>LimitsAndAverage(int*,int);

int main()
{
  int nums[] = { 8,2,9,4,1,4,6,9,0,-1,7 };
  pair result = Extremes(nums, size(nums));
  cout << "Lowest number is " << result.first <<
  ↳" Highest number is " << result.second << endl;
  auto [low,high,av] = LimitsAndAverage(nums,
  ↳size(nums));
  cout << "Lowest number is " << low <<
  ↳" Highest number is " << high << " Average is "
  ↳<< av << endl;
}


//*** Return lowest and highest number in list ***
pair<int, int> Extremes(int* list, int sz)
{
  //*** Set lowest and highest ***
  int lowest = list[0];
  int highest = list[0];
  //*** Search for lower and higher values ***
  for (int c = 1; c < sz; c++)
    if (list[c] < lowest)
      lowest = list[c];
    else if (list[c] > highest)
      highest = list[c];
  //*** Return result ***
  return make_pair(lowest, highest);
}
```

```
//*** Return lowest, highest and average number in
↳list ***
tuple<int, int, double> LimitsAndAverage(int* list,
↳int sz)
{
  //*** Set lowest and highest and total ***
  int lowest = list[0];
  int highest = list[0];
  double total = list[0];
  //*** Search for lower and higher values ***
  //*** and calculate total                ***
  for (int c = 1; c < sz; c++)
  {
    total += list[c];
    if (list[c] < lowest)
      lowest = list[c];
    else if (list[c] > highest)
      highest = list[c];
  }
  //*** Return result ***
  return make_tuple(lowest, highest, total/sz);
}
```

Activity 11.18

The program would display the value 22 since the contents of the variable *num* are not affected by the code executed in *AddOne()*.

Activity 11.19

Code for *RefParameter*:

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
void AddOne(int &v);

int main()
{
  int num{ 22 };
  AddOne(num);
  cout << num << endl;
}

void AddOne(int &v)
{
  v++;
}
```

The value displayed is 23 since the contents of *num* are incremented by the function.

Activity 11.20

Code for *TestSwap*:

```
#include <iostream>
using namespace std;

//*** User-defined Structures ***
struct TimeType
{
  int hrs;
  int mins;
  int secs;
};

//*** Function Prototypes ***
void Swap(TimeType&, TimeType&);

int main()
{
  TimeType t1, t2;

  //*** Read in times ***
  cout << "Enter first duration (hh:mm:ss) : ";
  scanf("%d:%d:%d", &t1.hrs, &t1.mins, &t1.secs);
  cout << "Enter second duration (hh:mm:ss) : ";
  scanf("%d:%d:%d", &t2.hrs, &t2.mins, &t2.secs);
  //*** Display entered values ***
  cout << "Before swap : " << endl;
  cout << "t1 : " << t1.hrs << ':' << t1.mins
  ↳<< ':' << t1.secs << endl;
  cout << "t2 : " << t2.hrs << ':' << t2.mins
  ↳<< ':' << t2.secs << endl;
  //*** Swap contents ***
  Swap(t1, t2);
  //*** Display new values ***
  cout << "After swap : " << endl;
  cout << "t1 : " << t1.hrs << ':' << t1.mins << ':'
  ↳<< t1.secs << endl;
  cout << "t2 : " << t2.hrs << ':' << t2.mins << ':'
  ↳<< t2.secs << endl;
}

//*** Swaps the contents of v1 and v2 ***
void Swap(TimeType &v1, TimeType &v2)
{
  TimeType temp{ v1 };
  v1 = v2;
  v2 = temp;
}
```

Activity 11.21

Modified code for *TimeFunction*:

```
#include <iostream>
using namespace std;

//*** User defined types ***
struct TimeType
{
  int hrs;
  int mins;
  int secs;
};

//*** Function prototypes ***
TimeType AddTimes(TimeType &t1, TimeType &t2);

void main()
{
  TimeType tm1, tm2;

  //*** Input times ***
  cout << "Enter first time (hh:mm:ss) : ";
  scanf("%d:%d:%d", &tm1.hrs, &tm1.mins, &tm1.secs);
  cout << "Enter second time (hh:mm:ss) : ";
  scanf("%d:%d:%d", &tm2.hrs, &tm2.mins, &tm2.secs);
  TimeType answer = AddTimes(tm1, tm2);
  if (answer.hrs != -1)
    cout << tm1.hrs << ':' << tm1.mins << ':'
    ↳<< tm1.secs << " + " << tm2.hrs << ':'
    ↳<< tm2.mins << ':' << tm2.secs << " = "
    ↳<< answer.hrs << ':' << answer.mins << ':'
    ↳<< answer.secs << endl;
  else
    cout << "One or more of the times given was
    ↳invalid\n";
}


TimeType AddTimes(TimeType &t1, TimeType &t2)
{
  TimeType result {-1,-1,-1};
  if (t1.hrs < 0 || t1.mins < 0 || t1.mins > 59 ||
  ↳t1.secs < 0 || t1.secs > 59||t2.hrs < 0 ||
  ↳t2.mins < 0 || t2.mins > 59 || t2.secs < 0 ||
  ↳t2.secs > 59)
    return result;
  int total_secs = (t1.hrs + t2.hrs) * 60 * 60 +
  ↳(t1.mins + t2.mins) * 60 + (t1.secs + t2.secs);
  result.hrs = total_secs / 3600;
  result.mins = (total_secs - result.hrs * 3600)/60;
  result.secs = total_secs % 60;
  return result;
}
```

Activity 11.22

Modified code for *AddTimes()* in *TimeFunction*:

```
TimeType AddTimes(TimeType &t1, TimeType &t2)
{
  TimeType result {-1,-1,-1};
  if (t1.hrs < 0 || t1.mins < 0 || t1.mins > 59 ||
  ↳t1.secs < 0 || t1.secs > 59||t2.hrs < 0 ||
  ↳t2.mins < 0 || t2.mins > 59 || t2.secs < 0 ||
  ↳t2.secs > 59)
    return result;
  t1.secs = 99;
  int total_secs = (t1.hrs + t2.hrs) * 60 * 60 +
  ↳(t1.mins + t2.mins) * 60 + (t1.secs + t2.secs);
```

```
  result.hrs = total_secs / 3600;
  result.mins = (total_secs - result.hrs * 3600)/60;
  result.secs = total_secs % 60;
  return result;
}
```

When we run the program with valid times the result shows that the *secs* field of *t1* has been changed to 99.

Activity 11.23

With the prototype of *AddTimes()* changed to

```
TimeType AddTimes(const TimeType, const TimeType);
```

and the function's heading to

```
TimeType AddTimes(const TimeType &t1,
↳const TimeType &t2)
```

the program no longer compiles with an error highlighted for the line

```
t1.secs = 99;
```

When the above line is removed, the program compiles and runs as before.

Activity 11.24

The first version operates correctly.

When the term `const` is removed from the prototype and heading, the call `F(PI)` causes an error.

Activity 11.25

Modified code for *TestSwap*:

```
#include <iostream>
using namespace std;

//*** User-defined Structures ***
struct TimeType
{
  int hrs;
  int mins;
  int secs;
};

//*** Function Prototypes ***
void Swap(TimeType&, TimeType&);
void Swap2(TimeType*, TimeType*);

int main()
{
  TimeType t1, t2;

  //*** Read in times ***
  cout << "Enter first duration (hh:mm:ss) : ";
  scanf("%d:%d:%d", &t1.hrs, &t1.mins, &t1.secs);
  cout << "Enter second duration (hh:mm:ss) : ";
  scanf("%d:%d:%d", &t2.hrs, &t2.mins, &t2.secs);
  //*** Display entered values ***
  cout << "Before swap : " << endl;
  cout << "t1 : " << t1.hrs << ':' << t1.mins << ':'
  ↳<< t1.secs << endl;
  cout << "t2 : " << t2.hrs << ':' << t2.mins << ':'
  ↳<< t2.secs << endl;
  //*** Swap contents ***
  Swap2(&t1, &t2);
  //*** Display new values ***
  cout << "After swap : " << endl;
  cout << "t1 : " << t1.hrs << ':' << t1.mins << ':'
  ↳<< t1.secs << endl;
  cout << "t2 : " << t2.hrs << ':' << t2.mins << ':'
  ↳<< t2.secs << endl;
}

//*** Swaps the contents of v1 and v2 ***
void Swap(TimeType &v1, TimeType &v2)
{
  TimeType temp{ v1 };
  v1 = v2;
  v2 = temp;
}

//*** Swaps the contents of v1 and v2 ***
void Swap2(TimeType *v1, TimeType *v2)
{
  TimeType temp{ *v1 };
  *v1 = *v2;
  *v2 = temp;
}
```

Activity 11.26

Code for *ArrayParams*:

```
#include<iostream>
#include<time.h>
using namespace std;

//*** Function Prototypes ***
int Smallest(const int[], int);
int Largest(const int[], int);

int main()
{
  int nums[15];
  //*** Generate random values in array ***
  srand(static_cast<int>(time(0)));
  for (int& v : nums)
    v = rand()%100+1;
  //*** Display all values in array ***
  for (auto v : nums)
    cout << v << "  ";
  cout << endl;
  //*** Display smallest and largest found ***
  int smallvalue = Smallest(nums, size(nums));
  cout << "Smallest value found was : " <<
  ↳smallvalue << endl;
  int largevalue = Largest(nums, size(nums));
  cout << "Largest value found was : " <<
  ↳largevalue << endl;
}


int Smallest(const int list[], int elements)
{
  //*** Start with value in first cell as smallest
  ↳***
  int small = list[0];
  //*** FOR each remaining value DO ***
  for (int c = 1; c < elements; c++)
    // *** IF cell value smaller, save it ***
    if (list[c] < small)
      small = list[c];
  //*** Return smallest value found ***
  return small;
}


int Largest(const int list[], int elements)
{
  //*** Start with value in first cell as largest ***
  int large = list[0];
  //*** FOR each remaining value DO ***
  for (int c = 1; c < elements; c++)
    // *** If cell value larger, save it ***
    if (list[c] > large)
      large = list[c];
  //*** Return largest value found ***
  return large;
}
```

Activity 11.27

Only two lines need to be changed in *ArrayParams* in order to use a pointer as the first parameter to *Smallest()*. These are to the function prototype and heading:

```
int Smallest(int*, int);
int Smallest(int* list, int elements)
```

No changes are required to the body of the function nor to the function call.

Activity 11.28

Only two lines need to be changed in *ArrayParams* in order to ensure neither parameter to *Largest()* can be changed within the function body. These are to the function prototype and heading:

```
int Largest(const int[], const int);

int Largest(const int list[], const int elements)
```

No changes are required to the body of the function nor to the function call.

Activity 11.29

Using the data given, the program displays the number 95.

Activity 11.30

Modified code for *HighMark*:

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
int GetHighestMarkInClass(const int [][5],
↳const int);
void GetHighestMarkPerStudent(const int [][5],
↳const int, int[]);

int main()
{
   int exammarks[][5]{ {23,65,85,33,23},{78,72,81,
   90,55},{88,93,95,90,79} };
   int highest[3];
   cout << "Highest mark in class was "
   << GetHighestMarkInClass(exammarks, 3) << endl;
   GetHighestMarkPerStudent(exammarks, 3, highest);
   cout << "Highest mark for each student:\n";
   for (int c = 0; c < 3; c++)
      cout << "student " << c + 1 << " : "
      ↳<< highest[c] << endl;
}


//*** Returns the highest mark achieved by any
student ***
int GetHighestMarkInClass(const int marks[][5],
const int stds)
{
   //*** If invalid number of students, exit ***
   if (stds < 1)
      return -1;
   //*** Find the highest ***
   int highest = marks[0][0];
   for (int student = 0; student < stds; student++)
      for (int exam = 0; exam < 5; exam++)
         if (marks[student][exam] > highest)
            highest = marks[student][exam];
   //*** Return highest mark found ***
   return highest;
}

//*** Returns highest mark per student in highs ***
void GetHighestMarkPerStudent(const int marks[][5],
↳const int stds, int highs[])
{
   //*** If invalid number of students, exit ***
   if (stds < 1)
      return ;
   //*** Find the highest ***
   for (int student = 0; student < stds; student++)
   {
      //*** Find this student's highest mark ***
      int highest = marks[student][0];
      for (int exam = 0; exam < 5; exam++)
         if (marks[student][exam] > highest)
            highest = marks[student][exam];
      //*** Save this student's highest mark ***
      highs[student] = highest;
   }
}
```

Activity 11.31

Modified code for *ArrayParams*:

```
#include<iostream>
#include<time.h>
using namespace std;

//*** Function Prototypes ***
int Smallest(const int*, int);
int& Smallest2((int[], int);
int Largest(const int[], const int);

int main()
{
   int nums[15];
   //*** Generate random values in array ***
   srand(static_cast<int>(time(0)));
   for (int& v : nums)
      v = rand() % 100 + 1;
   //*** Display all values in array ***
   for (auto v : nums)
      cout << v << "  ";
   cout << endl;
   //*** Find location of smallest value ***
   int& ans = Smallest2(nums, size(nums));
   //*** Display value found ***
   cout << ans << endl;
   //*** Change value at location to 99 ***
   ans = 99;
   //*** Display all values in array ***
   for (auto v : nums)
      cout << v << "  ";
   cout << endl;
}


int Smallest(const int list*, int elements)
{
   //*** Start with value in first cell as smallest
   ↳***
   int small = list[0];
   //*** FOR each remaining value DO ***
   for (int c = 1; c < elements; c++)
      // *** IF cell value smaller, save it ***
      if (list[c] < small)
         small = list[c];
   //*** Return smallest value found ***
   return small;
}

int& Smallest2(int list[], int elements)
{
   //*** Assume cell zero contains the smallest ***
   int post = 0;
   //*** FOR each remaining position DO ***
   for (int c = 1; c < elements; c++)
      // *** IF value at position smaller, save
      ↳position ***
      if (list[c] < list[post])
         post = c;
   //*** Return address of smallest value found ***
   return list[post];
}


int Largest(const int list[], const int elements)
{
   //*** Start with value in first cell as largest ***
   int large = list[0];
   //*** FOR each remaining value DO ***
   for (int c = 1; c < elements; c++)
      // *** If cell value larger, save it ***
      if (list[c] > large)
         large = list[c];
   //*** Return largest value found ***
   return large;
}
```

Activity 11.32

The new versions of the prototype and heading for *Smallest2()* are:

```
int& Smallest2(const int, int elements);
int& Smallest2(const int list[], int elements)
```

Once these changes have been made, the program no longer compiles.

Activity 11.33

In *ArrayParams* the prototype and heading for *Smallest2()* become:

```
const int& Smallest2(const int[], int);
const int& Smallest2(const int list[], int elements)
```

In **main()** the value returned must be stored in a **const int&** type:

```
//*** Find location of smallest value ***
const int& ans = Smallest2(nums, size(nums));
```

Activity 11.34

Code for *PointerReturn (version 1)*:

```
#include <iostream>
#include <time.h>>
using namespace std;

//*** Function prototypes ***
int* CreateRandomList(int, int, int);

int main()
{
  //*** Set up random list ***
  int *list = CreateRandomList(10, 10, 100);
  //*** Display its contents ***
  for (int c = 0; c < 10; c++)
    cout << list[c] << "  ";
  cout << endl;
  //*** Delete the space allocated ***
  delete[] list;
}

int* CreateRandomList(int count, int min, int max)
{
  //*** If invalid parameter, exit ***
  if (count < 1 || min > max)
    return nullptr;
  //*** Seed numbers ***
  srand(static_cast<int>(time(0)));
  //*** Create space ***
  int* temp = new int[count];
  //*** Fill with random values ***
  for (int c = 0; c < count; c++)
    temp[c] = rand() % (max - min) + min;
  //*** Return address of list ***
  return temp;
}
```

Code for *PointerReturn (version 2)*:

```
#include <iostream>
#include <time.h>>
using namespace std;

//*** Function prototypes ***
int* CreateRandomList(int, int, int);

int main()
{
  //*** Set up random list ***
  int *list = CreateRandomList(10, 10, 100);
  //*** Display its contents ***
  for (int c = 1; c <= list[0]; c++)
    cout << list[c] << "  ";
  cout << endl;
  //*** Delete the space allocated ***
  delete[] list;
}


int* CreateRandomList(int count, int min, int max)
{
  //*** If invalid parameter, exit ***
  if (count < 1 || min > max)
    return nullptr;
  //*** Seed numbers ***
  srand(static_cast<int>(time(0)));
  //*** Create space ***
  int* temp = new int[count+1];
  //*** Record number of values in list ***
  temp[0] = count;
  //*** Fill with random values ***
  for (int c = 1; c <= count; c++)
    temp[c] = rand() % (max - min) + min;
  //*** Return address of list ***
  return temp;
}
```

Activity 11.35

With an invalid parameter to *CreateRandomList()*, the program produces a runtime error.

Modified code for *PointerReturn*

```
#include <iostream>
#include <time.h>>
using namespace std;

//*** Function prototypes ***
int* CreateRandomList(int, int, int);

int main()
{
  //*** Set up random list ***
  int *list = CreateRandomList(-1, 10, 100);
  if (list == nullptr)
    cout << "No list created\n";
  else
  {
    //*** Display its contents ***
    for (int c = 1; c <= list[0]; c++)
      cout << list[c] << "  ";
    cout << endl;
    //*** Delete the space allocated ***
    delete[] list;
  }
}


int* CreateRandomList(int count, int min, int max)
{
  //*** If invalid parameter, exit ***
  if (count < 1 || min > max)
    return nullptr;
  //*** Seed numbers ***
  srand(static_cast<int>(time(0)));
  //*** Create space ***
  int* temp = new int[count+1];
  //*** Record number of values in list ***
  temp[0] = count;
  //*** Fill with random values ***
  for (int c = 1; c <= count; c++)
    temp[c] = rand() % (max - min) + min;
  //*** Return address of list ***
  return temp;
}
```

Activity 11.36

No solution required.

Activity 11.37

Modified code for *HighMark*:

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
int GetHighestMarkInClass(const int [][5],
↳const int);
unique_ptr<int[]> GetHighestMarkPerStudent(
↳const int[][5], const int);

int main()
{
  int exammarks[][5]{ {23,65,85,33,23},
  ↳{78,72,81,90,55},{88,93,95,90,79} };
  //int highest[3];
  cout << "Highest mark in class was " <<
  ↳GetHighestMarkInClass(exammarks, 3) << endl;
  auto highest = GetHighestMarkPerStudent(exammarks,
  ↳3);
  if (highest)
  {
    cout << "Highest mark for each student:\n";
    for (int c = 0; c < 3; c++)
      cout << "student " << c + 1 << " : "
      ↳<< highest[c] << endl;
```

```
   }
}

//*** Returns the highest mark achieved by any
student ***
int GetHighestMarkInClass(const int marks[][5],
const int stds)
{
   //*** If invalid number of students, exit ***
   if (stds < 1)
      return -1;
   //*** Find the highest ***
   int highest = marks[0][0];
   for (int student = 0; student < stds; student++)
      for (int exam = 0; exam < 5; exam++)
         if (marks[student][exam] > highest)
            highest = marks[student][exam];
   //*** Return highest mark found ***
   return highest;
}

//*** Returns highest mark per student in highs ***
unique_ptr<int[]> GetHighestMarkPerStudent(
↳const int marks[][5], const int stds)
{
   //*** If invalid number of students, exit ***
   if (stds < 1)
      return nullptr;
   //*** Create space for highest marks ***
   unique_ptr<int[]> highs =
   ↳make_unique<int[]>(stds);
   //*** Find the highest ***
   for (int student = 0; student < stds; student++)
   {
      //*** Find this student's highest mark ***
      int highest = marks[student][0];
      for (int exam = 0; exam < 5; exam++)
         if (marks[student][exam] > highest)
            highest = marks[student][exam];
      //*** Save this student's highest mark ***
      highs[student] = highest;
   }
   return highs;
}
```

Note that `main()` has to check that the returned value is not a null pointer before displaying the highest mark per student.

Activity 11.38

A possible set of test values would be

Test Data For *IsLeapYear()*	
Test Value	**Expected Result**
year = 1500	-1
year = 2015	0
year = 2016	1

Activity 11.39

An updated set of test values would be

Test Data For *IsLeapYear()*	
Test Value	**Expected Result**
year = 1500	-1
year = 2015	0
year = 2016	1
year = 2000	1
year = 1900	0

Activity 11.40

No solution required.

Activity 11.41

```
int Largest(const int list[], const int elements)
{
      int large = list[0];
      for (int c = 1; c < elements; c++)
            if (list[c] > large)
                  large = list[c];
      return large;
}
```

+1

Complexity = 3

12

User-Defined Functions - 2

In this Chapter:

- ❑ Default Parameter Values
- ❑ Inline Functions
- ❑ Function Overloading
- ❑ Operator Overloading
- ❑ Recursive Functions
- ❑ Global Structures and Variables
- ❑ Static Variables
- ❑ Pointers to Functions
- ❑ Creating a Library

User-Defined Functions - 2

Introduction

In the previous chapter we covered the basics of creating functions in C++. However, there are several more advanced topics to be discussed for a full appreciation of what options we have available when creating functions.

Default Parameter Values

Sometimes one or more of the actual parameters passed to a function will be the same value almost every time that function is called. For example, in the last chapter we created the function *DrawLine()* the final version of which took the length and character used as parameters and used these to display a sequence of the specified character.

Now, it may be that, although we have the option to specify which character is used to when drawing the line in *DrawLine()*, most of the time the asterisk character will be used. In this situation, we can make use of C++'s option to assign a default value to a parameter.

This default value is given only in the function's prototype. So, if we wanted to assign the asterisk as the default value for *DrawLine()*'s *ch* parameter, the prototype would be

```
void DrawLine(int length, char ch = '*');
```
or
```
void DrawLine(int, char = '*');
```

The function definition remains unchanged.

But what advantage do we gain from having a default value for a parameter? Well, if we want to use the default value when the function is called, we can omit a value for that parameter altogether when the function is called. This means that the line

```
DrawLine(9);
```

would draw a line of nine asterisks.

Where we want a different character to be used, we simply supply two parameters in the function call as we did before:

```
DrawLine(12, '+'); //Draws a line 12 + characters
```

A default value can be given to as many parameters as we wish. If we wanted `DrawLine()`'s *length* parameter to default to 10, we would rewrite the prototype as

```
void DrawLine(int = 10, char '*');
```

Now if we were to call the function with the line

```
DrawLine()
```

we would get a line of 10 asterisks.

With this latest version of the function we now have three choices when calling `DrawLine()`:

- Supply no values

  `DrawLine();` gives a 10-asterisk line

- Supply only the length

```
DrawLine(5);
```
gives a 5-asterisk line

- Supply length and character

```
DrawLine(3,'$');
```
gives a 3 dollar sign line

Notice that it is not possible to supply only the *ch* parameter's value and use the default *length* value. The compiler would have no way of knowing which parameter we wanted to override – *length* or *ch* – and so it imposes the following rule when overriding default parameter values in a function call:

> **When a default value of a parameter is overridden, all the parameters to the left of that parameter must also be given explicit values.**

Another limitation is imposed when defining a function with default parameters:

> **When a parameter is given a default value, then all parameters to the right of that one must also be given a default value**.

So if we only wanted to set the default value for *length* of 10 and used the prototype

```
void DrawRectangle(int = 10, char);
```

the compiler would object and insist on a default value for the `char` parameter as well.

Activity 12.1

Modify your *DrawLine()* function in project *UsingFunctions* to default to using the asterisk character. Test your change by creating a line of 20 asterisks without specifying the character in the call to *DrawLine()*.

Modify your *DrawLine()* function again, so that it defaults to a length of 10 and the asterisk character. Test your function by displaying a line of 10 asterisks without specifying any parameter values in the call to *DrawLine()*.

Inline Functions

Every function used within a program is stored in its own memory area. When the function is called, control jumps to that area, executes the code there and then returns to instruction immediately following the original function call.

However, when a function is called, a program has to do a considerable amount of behind-the-scenes work saving the address of where it must return to after the function's code has been executed and also saving the contents of the processor's internal registers so that their values can be restored when control returns from the function.

This hidden burden means that function calling is less efficient than writing the program as a single continuous block of code (as our first programs were). For a function containing several lines of code, that overhead represents only a small percentage of the total time required to execute the lines of code within the function, but for functions containing only a line or two of code, the overhead can be a significant percentage of the total time spent handling the called function.

In such cases it is possible to declare a function as an **inline function**. When a

function is made inline, then the pre-compiler removes the call to that function and replaces it with the actual function code (making appropriate adjustments for the parameters used). We can think of the process as being akin to the *find and replace* operation offered by text editing packages: find the function call, replace it with the actual function code.

For example, in the previous chapter we created an *IsLeapYear()* function which returned a Boolean value stating if the year is a leap year. This requires only a single line of code and is therefore the ideal candidate to be an inline function. The code for this would be:

```
inline bool IsLeapYear(const int year)
{
   return (((year%4==0 && year % 100 != 0)||(year%400==0))&&
   ↳year > 1752);
}
```

Starting a function definition with the keyword `inline` requests the compiler to make the function inline by replacing any function call in the source code with the function's actual code immediately before compiling the program. But it is up to the compiler to decide if it is going to implement this request! There is no guarantee that the routine will be treated as an inline function.



For example, the compiler will not inline a function containing loops.

An inline function must be placed before `main()`.

The compiler may even make a function inline without us requesting it to do so if it decides that would be the most efficient option.

Activity 12.2

In your *DOWFunction* project, replace the original version of *IsLeapYear()* with the inline one given above.

Make use of this new function at the appropriate point in *IsValidDate()* when determining the number of days in February.

Test the function by checking that 29/2/2000 is valid and that 29/2/2100 is not.



Also modify *IsValidDate()* to use a const reference parameter.

constexpr

The term `constexpr` (short for **constant expression**) is a term often added as the first element in a function heading. For example, we might write

```
constexpr int Add(int no1, int no2)
```

Adding this term is taken as a request to have the compiler optimise the code for the function call at compile time and hence reduce the execution time of a program.

This is somewhat similar to a request for an inline function. But unlike `inline`, `constexpr` can also be applied to a variable declaration which is assigned an initial value. allowing the expression used in assigning their value to be determined at compile time where possible.

A trivial example of both a `constexpr` function is shown in FIG-12.1. Note that a `constexpr` function, should have its full code positioned before any call to that function so, in the example, no function prototype is included.

FIG-12.1

Using constexpr

```
#include <iostream>
using namespace std;

constexpr int Add(int a, int b)
{
   return a + b;
}



int main()
{
   int list[Add(6, 4)]{2,4,6,8};
   cout << size(list) << endl;
}
```

Activity 12.3

Start a new project called *TestConstExpr* and implement the code given above.

Does the code compile and run?

Remove the term `constexpr` from the heading of *Add()* and try running the program again. Does it compile and run?

In `main()`, add a second `int` array called *seq* with the line

```
int seq[list[2]]{};
```

Does the program compile and run?

Add the term `constexpr` as the first term in the line declaring list. Does the program compile and run?

Although `constexpr` is used extensively to optimise program code with the aim of reducing execution time, it is rarely required when learning the basics of C++17, so we will make little use of it in future examples.

Static Variables

As we know, local variables within a function are destroyed when the function finishes executing. There are situations when this becomes a problem. For example, let's assume we'd like to have a function remember how often it has been called. Ideally, we would want to write something like:

```
void Function()
{
   int timescalled {0};
   timescalled ++;
   cout << "Function() called " << timescalled << times\n";
}
```

Of course, this will only ever display the value 1 since *timescalled* gets recreated each time the function is called.

However, if we declare the variable *timescalled* as a `static` variable,

```
static int timescalled {0};
```

then it continues to exist even after the function code has completed execution. Also, the initialisation process (where *timescalled* is set to zero) is only executed once so the variable retains its value from function call to function call.

The program in FIG-12.2 demonstrates the use of a static variable within a function.

FIG-12.2

Using Static Variables

```
#include <iostream>
using namespace std;


//*** Function prototype ***
void Counting();


int main()
{
   for(int c = 1; c <= 10; c++)
      Counting();
}


void Counting()
{
   static int timescalled = 0;
   timescalled ++;
   cout << "Counting() called " << timescalled << " times\n";
}
```

Activity 12.4

Start a new project called *StaticTest* which implements the code given above.

Run the program and observe the display produced.

Remove the word `static` from the declaration of *timescalled* and see how this affects the display produced.

Activity 12.5

The Fibonacci series is a sequence of integer values. The next value in the sequence is always formed from the sum of the previous two values. The first two values in the sequence are 0 and 1. The sequence begins 0, 1, 1, 2, 3, 5, 8.

Create a new project called *Fibonacci* which contains a function *GetNextFibonacci()* which returns the next number in the sequence each time it is called. It returns 1 the first time it is called, 2, the second time, etc.

To allow the function to be called repeatedly, use the following code in main():

```
do
{
      cout << GetNextFibonacci() << " ";
}
while (getchar() != 'q');
```

Lifetime and Scope

The **lifetime** of a variable is the part of the program during which it has space in memory. This is not always identical to its **scope** which is the part of the program where the variable may be accessed.

For example, a `static` variable is assigned space when its definition is first reached during execution and that space remains allocated for the remainder of the program's execution. However, that same static variable is only in scope during the execution of the block in which it is defined and cannot be accessed outside that block.

We already know that any variable only has scope within the block in which it is declared (and any inner blocks). The same is true for any types we declare using `struct`, `union`, `typedef` and `using` keywords.

If we want these definitions to be understood throughout the various functions that make up our program, we need to position these definitions outside these functions.

Hence, rather than define `DateType` within `main()`, we must defined it separately, before `main()` or any function prototypes as in the code:

```
struct DateType
{
   int day;
   int month;
   int year;
};

// *** Function prototypes here ***

// *** Code for main() and other functions here ***
```

By doing this we make our definitions available to all the code within our source file. The terms **global scope** or **file scope** is often used for these items.

Global Variables

Variables can also be declared outside any function and hence given global scope. Unsurprisingly, these are known as **global variables**.

The short program in FIG-12.3 demonstrates the use of a global variable, displaying the contents of global variable *num* in both `main()` and *functA()*.

FIG-12.3

Using a Global Variable

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
void functA();

//*** GLOBAL Variables ***
int num = 12;

int main()
{
   cout << "In main() num = " << num << endl;
   functA();
}
```



FIG-12.3
(continued)

Using a Global Variable

```
void functA()
{
   cout << "In functA() num = " << num << endl;
}
```

Activity 12.6

Start a new project called *Scope* and enter the program given in FIG-12.3.

Run the program and observe what values are displayed.

If a global variable is not explicitly assigned an initial value it is automatically set to zero if it is numeric, the *null* character if it is of type `char`, and *false* if it is of type `bool`:

```
//*** Global variables ***
int a;      //Set to zero
float b;    //Set to zero
char c;     //Set to \0
bool d;     //Set to false

int main()
{
   ...
```

Scope Resolution

Having a global variable does not stop a function from containing another variable of the same name. When this occurs, the most-recently declared variable has precedence meaning that the variable which is local to the function is used rather than the global variable.

Activity 12.7

Modify function *functA()* in *Scope* so that it is coded as:

```
void functA()
{
    int num{ 20 }; // Local variable
    cout << "In functA() num = " << num << endl;
}
```

Run the program and observe what values are displayed.

When a variable is declared within a function, it has scope only within that function. One consequence of this is that it is allowable to duplicate variable names in your program as long as each of the identically-named variables have their own unique scope. For example, we have often used the variable *c* as a `for` loop counter in different functions without any complains from the compiler.

Even though a local variable of the same name exists, it is still possible to access the global variable by using the scope resolution operator (::). Hence, within *functA()*,

`num`	refers to the local variable
`::num`	refers to the global variable

Activity 12.8

Modify *functA()* in *Scope* so that it displays the value of the global variable rather than that of the local variable.

Generally the use of global variables is considered bad programming since they can then be accessed from anywhere within a program and this can make fault-finding more difficult.

Function Overloading

Occasionally, we require several functions, all of which perform an identical task, but using different parameters.

For example, in Chapter 11 Activity 11.14 we created a function called *IsValidDate()* which returned the day of the week (coded as an integer between 0 and 6) on which a given date fell. That date was passed to the function in a *DateType* record structure. The function had the prototype

```
bool IsValidDate(const DateType&);
```

Now let's assume we want a function that performs exactly the same task, but this time, rather than supply a *DateType* parameter, we want to supply the function with three separate integer values representing day, month and year. This new function would have the prototype

```
bool IsValidDate(const int, const int, const int);
```

and be coded as

```
//*** Check date is valid ***
bool IsValidDate(const int day, const int month,
↳const int year)
{
   bool result{ true };
   //** If invalid year, exit **
   if (year <= 1752)
      result = false;;
   //** If invalid month, exit **
   if (month < 1 || month > 12)
      result = false;
   //** If invalid day, exit **
   int daysinmonth[]{ 0,31,28,31,30,31,30,31,31,30,31,30,31 };
   //* If leap year and 1 to days in Feb ***
   if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
      daysinmonth[2]++;
   if (day < 1 || day > daysinmonth[dt.month])
      result = false;
   //*** Return date's validity ***
   return result;
}
```

In older programming languages both routines could not exist within the same program: functions with identical names were not allowed. The second version of the routine would have to be given a different name from the first - even although they both perform exactly the same operation. But in C++ we are free to give both routines the same name.

In fact, if necessary, we could create several functions all with the same name as long

as the parameter list for each differed in some way.

It is possible for the return type to vary between the identically named function but there must also be some variation in the function parameter list. Creating two or more functions with the same name is known as **function overloading**.

Activity 12.9

Open your project *DOWFunctions* and add the second version of *IsValidDate()*.

Next, add an overloaded version of *GetDayOfWeek()* which also takes three `int` values rather than a *DateType* parameter. Use a call to the three `int` version of *IsValidDate()* within *GetDayOfWeek()*'s code.

Test that both versions of the *GetDayOfWeek()* function operate correctly.

When one of the identically named functions is called, the compiler identifies which one to execute by examining the parameters being supplied in the function call. In the case of *GetDayOfWeek()*, if three integer values are provided in the call, the new version of the routine is executed; if a single *DateType* value is given, then the original version is executed.

Note that we can also overload a function by using the same parameter types but in a different order. For example, it would be acceptable to write

```
int MyFunction(int,double);
int MyFunction(double,int);
```

when creating two different functions.

Restrictions on Overloading

Attempting to overload a function name by changing only the return type is NOT sufficient. Hence, trying to overload a function name with the prototypes

```
int MyFunction2(int,int);
bool Myfunction2(int,int);
```

would not be allowed.

When overloading a function name, C++ does not differentiate between pointers and arrays. For example, functions with the prototypes

```
int Smallest(int[], int);
int Smallest(int*, int);
```

would not be considered different and the code would fail to compile. The reason for this is easily understood if we look at how a call to *Smallest()* would be attempted:

```
int ans = Smallest(list,size(list);
```

The actual parameter list (assuming it is the name of a previously defined array) can be treated as an array name or a pointer and so there is no way to tell which version of the function should be called.

Exactly the same problem means that we cannot overload functions where the difference in parameters is only value or reference types. With prototypes

```
double Times3(double);
double Times3(double&);
```

there would be no difference in how the functions where called:

```
double result = Times3(num);
```

So we cannot overload where the parameter difference is only call-by-value and call-by-reference.

A third restriction is that there can be no differentiation between const and non-const call-by-value parameters. Therefore the following overload is not allowed:

```
void f(const int);
void f(int);
```

However, where pointer or reference parameters are used, then the compiler can differentiate between const and non-const parameters, meaning it would allow the following:

```
void f(const int*);
void f(int*);
```

The first of these would be executed if called with the actual parameter defined as a `const int*` and the second would be called where the actual parameter was of type `int*`.

Functions where one or more parameters have been assigned default values also cause problems when overloading. For example, if we have the following function prototypes

```
void f(int, int = 0);
void f(int);
```

a call such as

```
f(12);
```

would be ambiguous. Do we mean to call the first version of *f()* with the second parameter defaulting to zero? Or do we want to call the second version of *f()*? So, we have another situation that won't compile.

Problems with Overloading

We already know that C++ will do automatic type conversions where necessary when copying an actual parameter to a formal parameter when a function begins execution. This can cause problems when our code uses function overloading. In the code below we have two functions named *f()*:

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
void f(char, char);
void f(int, int);

int main()
{
   f(65, 'A');
}


void f(int a, int b)
{
   cout << "Two int version called\n";
}
```

```
void f(char a, char b)
{
   cout << "Two char version called\n";
}
```

But the call to *f()* in `main()` is not an exact match for either function. And although C++ could easily convert the actual parameters to match the formal parameters of either function, because there is no obvious way of deciding which to call, the compiler will flag an error.

Even if `main()` called *f()* with the line

```
f(65.2, 66.3);
```

the compiler would not be able to decide on which version to call.

The most secure way of solving this problem (other than to change the actual parameters!), would be to cast the parameter to the type matching which function should be called:

```
f(static_cast<int>('A'), 66); //Calls two int version of f()
```

Operator Overloading

Another project we created in the last chapter made use of a *TimeType* data structure defined as

```
struct TimeType
   {
      int hrs;
      int mins;
      int secs;
   };
```

We used this in a function called *AddTimes()* which we coded as

```
TimeType AddTimes(const TimeType& t1, const TimeType& t2)
{
   //*** Calculate total time ***
   int total_secs = (t1.hrs + t2.hrs) * 60 * 60 +
   (t1.mins + t2.mins) * 60 + (t1.secs + t2.secs);
   result.hrs = total_secs / 3600;
   result.mins = (total_secs - result.hrs * 3600)/60;
   result.secs = total_secs % 60;
   return result;
}
```

When using *TimeType* variables, we could add two *TimeType* values together and store the total in a third variable with code such as

```
TimeType tm1{1,3,57}, tm2{3,40,32}, total;
total = AddTimes(tm1,tm2);
```

But it would be better if we could use the same type of syntax as we do when adding to a numeric value as in the line

```
total = tm1 + tm2;
```

To achieve this we need to overload the existing + operator. This is done by creating a new function but this time its name must be `operator+`. How it's done is shown below:

```
TimeType operator+(const TimeType& t1, const TimeType& t2)
{
```

```
    //*** Calculate total time ***
    int total_secs = (t1.hrs + t2.hrs) * 60 * 60 +
    (t1.mins + t2.mins) * 60 + (t1.secs + t2.secs);
    TimeType result;
    result.hrs = total_secs / 3600;
    result.mins = (total_secs - result.hrs * 3600)/60;
    result.secs = total_secs % 60;
    return result;
}
```

Notice that the code is identical as that for *AddTimes()*. Only the function name has changed.

Although the overloaded + operator can be called in exactly the same manner as any other function,

```
total = operator+(tm1,tm2);
```

the whole point of going to the trouble of overloading an operator is to use it between the two parameters to the function as in:

```
total = tm1 + tm2;
```

Now the fact that we are calling a function is totally obscured by the syntax of the line.

Activity 12.10

Add `operator+` to *TimeFunction* and check that it operates correctly.

Add a second `operator+` whose parameters are *TimeType t* and `int` *s* that returns a *TimeType* value calculated as *t* time + *s* seconds.



Make no attempt to validate the times passed to these functions.

But overloaded operators can cause problems for the unwary. For example, we might think that our newly-defined + operator would work in the line

```
total = 7 + tm1;
```

However, remembering that we are actually calling a function, this latest version of the line has the parameters in the wrong order (`int`, *TimeType* instead of *TimeType*, `int`) and so will fail to compile.

If it was important to allow this latest version, we would need to create yet another version of the `operator+` function with the parameter order reversed to `int`, *TimeType*.

Activity 12.11

Add a third `operator+` to *TimeFunction* whose parameters are `int` *s* and *TimeType t* and that returns a *TimeType* value calculated as *t* time + *s* seconds.

We can overload any existing C++ operator, but you can't introduce new ones. So deciding that we'd like to make use of, say, the $ symbol as an operator is not allowed.

If we'd like to be able to compare two *TimeType* variables, we could overload the relational operators. For example, by adding the function

```
bool operator<(const TimeType& t1, const TimeType& t2)
{
    int total_secs1 = t1.hrs * 3600 + t1.mins * 60 + t1.secs;
    int total_secs2 = t2.hrs * 3600 + t2.mins * 60 + t2.secs;
```

```
   return (total_secs1 < total_secs2);
}
```

we overload the "less than" operator to work with *TimeType* data values and can then write lines such as

```
if (tm1 < tm2)
```

Activity 12.12

Add `operator<` to *TimeFunction*.

Add `operator==` which returns *true* if two times are equal.

Check that both relational operators work as expected.

Overloading the ++ and -- Operators

Because the ++ and -- operators can appear either before or after a variable name, with different results when used in an expression, the pre and post versions of the operators must be overloaded separately.

Of course, that introduces a problem: how do we overload an operator when both versions need exactly the same parameters?

To handle this, C++ allows us to add a dummy `int` parameter to the post version of the routines. So our first attempt at overloading the ++ operator for *TimeType* could be:

```
//*** Adds one second to time (pre) ***
void operator++(TimeType& t)
{
   int total_secs = t.hrs * 3600 + t.mins * 60 + t.secs+1;
   t.hrs = total_secs/3600;
   t.mins = (total_secs-t.hrs*3600)/60;
   t.secs = total_secs % 60;
}


//*** Adds one second to time (post)***
void operator++(TimeType& t,int)
{
   int total_secs = t.hrs * 3600 + t.mins * 60 + t.secs+1;
   t.hrs = total_secs/3600;
   t.mins = (total_secs-t.hrs*3600)/60;
   t.secs = total_secs % 60;
}
```

Notice that the dummy parameter doesn't even have to be assigned a name when coding the function.

Activity 12.13

Add both versions of the overloaded `operator++` to *TimeFunction* and check that both operate correctly by setting up a value for *TimeType* variables *tm1* and having the program increment the day using the lines

```
tm1++;
++tm1;
```

Actually, these simple functions used to overload the ++ operator aren't quite correct. Although they work fine when a *DateType* variable is incremented in isolation, they don't work correctly when used within a longer expression. For example, the line

```
TimeType tm2 = ++tm1;
```

will fail. The reason behind this may be a bit clearer if we use the other format of the same statement:

```
TimeType tm2 = operator++(tm1);
```

The statement is taking the value returned by the function `operator++()` and storing it in *tm2*. Unfortunately, the function we coded does not return a value!

Activity 12.14

Modify `main()` in *TimeFunction* so that it attempts to perform the statement

```
TimeType tm2 = ++tm1;
```

What happens when you attempt to compile the program?

To solve this problem, we need to update the code for the ++ functions, this time making them return *TimeType* values:

```
//*** Adds one second to time (pre) ***
TimeType operator++(TimeType& t)
{
   int total_secs = t.hrs * 3600 + t.mins * 60 + t.secs+1;
   t.hrs = total_secs/3600;
   t.mins = (total_secs-t.hrs*3600)/60;
   t.secs = total_secs % 60;
   return t;
}


//*** Adds one second to time (post)***
TimeType operator++(TimeType& t,int)
{
   TimeType temp{t};
   int total_secs = t.hrs * 3600 + t.mins * 60 + t.secs+1;
   t.hrs = total_secs / 3600;
   t.mins = (total_secs-t.hrs*3600) / 60;
   t.secs = total_secs % 60;
   return temp;
}
```

Notice that the first function returns the time after one second has been added while the second function returns the time before adding one second. This reflects the effects of the pre and post operations.

Activity 12.15

In *TimeFunction,* modify your prototypes and code for the ++ operators to match that given above. What values are assigned to *tm1* and *tm2* when you run the program now?

Change the line assigning a value to *tm2* so that the post ++ operation is used. What values are displayed this time?

Activity 12.16

In Activities 9.7 and 9.8 you created a project called *Enumeration* which made use of an enumerated type called *DaysType*.

Modify *Enumeration* so that both the ++ and `--` operators are overloaded for *DaysType* allowing one day to be added to (`++`) or subtracted from (`--`) the current day.

In both cases, overload the pre and post operators.

The values should cycle, hence incrementing *Saturday* should result in *Sunday* and decrementing *Sunday* should result in *Saturday*.

Recursive Functions

A recursive function is one which calls itself. This may seem a very strange concept if we haven't come across it before, but it can produce very elegant and short solutions to problems that might otherwise require significantly more code when other approaches are used.

The classic example of a recursive function is a solution to the factorial calculation.

When an integer number is multiplied by all lesser integers down to 1, then the result is said to be the **factorial** of the original value. In mathematics the factorial operation is identified using the exclamation mark (!). So the term

5!

is called *5 factorial* and is equal to

5 x 4 x 3 x 2 x 1

or

120

1! is simply 1 and 0! is also defined as 1.

Coding a *Factorial()* function would be quite simple using only a `for` loop, but here we will investigate how such a function can be defined recursively. The secret to the recursive solution is to realise that

5! = 5 x 4!

and that

4! = 4 x 3!

and so on down to

2! = 2 x 1!

and

1! = 1 x 0!

We can generalise the above and state that for values greater than 0,

v = v * (v-1)!

With this insight, we're ready to start coding a recursive version of *Factorial()*:

The heading of our function is easy enough

```
long Factorial(long v)
```

If *v* has the value 0, then we know the value to be returned is 1. This can be coded as:

```
long Factorial(long v)
{
   if ( v == 0)
      return 1;
```

But if *v* has a value greater than 0, then

v! = v x (v-1)!

So the complete function code is

```
long Factorial(long v)
{
      if ( v == 0)
         return 1;
      else
         return v * Factorial(v-1);
}
```

Notice that the function now calls itself in the last line. FIG-12.4 shows how this situation is handled when the program is run.

FIG-12.4

How Recursion Works

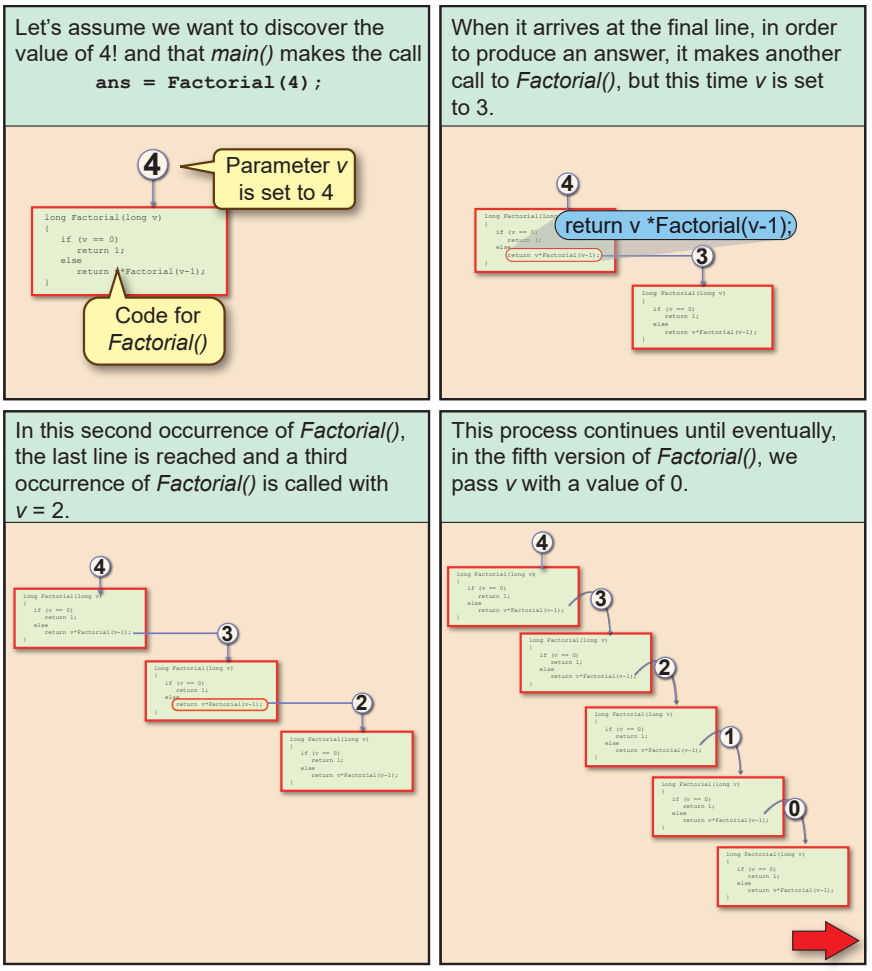
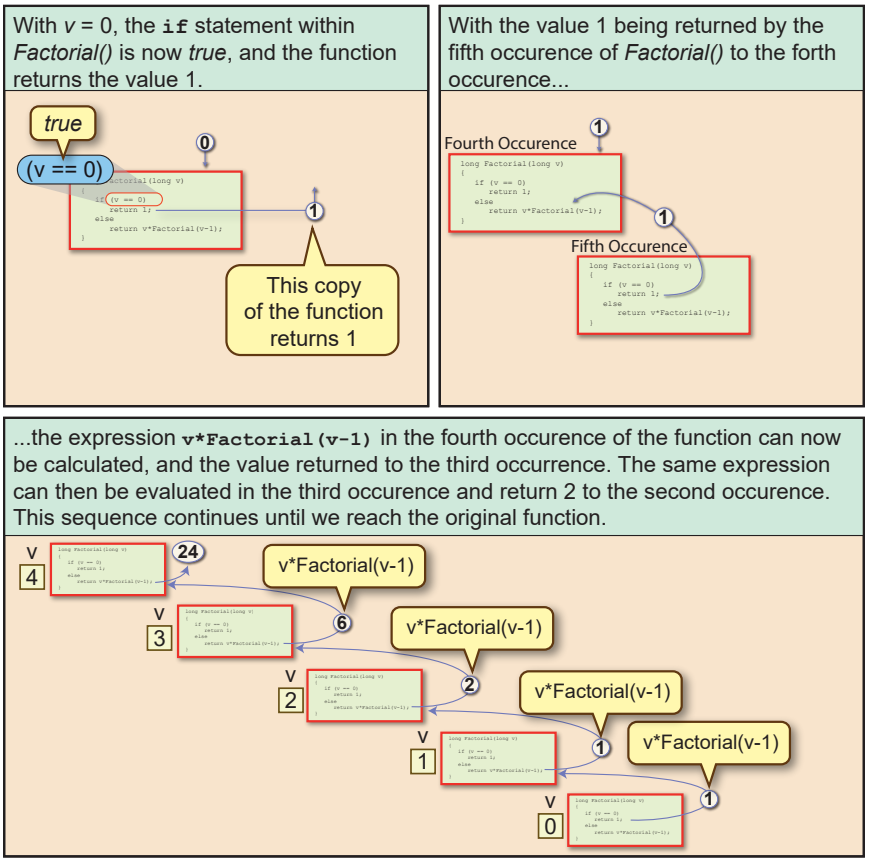


FIG-12.4
(continued)

How Recursion Works



In the end, the line

```
ans = Factorial(4);
```

assigns the value 24 to *ans*.

The only inaccuracy in the description of how recursion works is that FIG-12.4 gives the impression that a duplicate of the recursive function is created each time it calls itself. In fact, there is only ever one copy of the function's code within the machine at execution time and it is that copy that is called on each occasion with the software automatically saving (and later restoring) the values being used by earlier calls in the recursive sequence.

Activity 12.17

Start a new project called *RecursiveFact* and add the code for the *Factorial()* function given earlier.

Pointers to Functions

Until this point we have treated pointers as a method of referencing data, but pointers can also be used to reference functions. Like variables, a function's code is allocated memory and therefore has a start address. By having a pointer contain that start address, we have created a pointer to the function.

For example, if a program contained our original version of *DrawLine()* whose

prototype was:

```
void DrawLine();
```

then we can create a pointer, *funptr*, capable of referencing that function using the following line:

```
void(*funptr)();
```

Just as a data pointer declaration must specify the type of data being referenced, so a function pointer declaration must specify the signature of the function it references. The signature of a function is its return type and parameter types. This information is in the function prototype. For example, *DrawLine()*, returns no value and takes no parameters, hence its details are

```
void ()
```

Between the two parts of the signature we declare our function pointer variable enclosed in parentheses of its own and hence we arrive at the line

```
void (*funptr)();
```

If we are declaring function pointers in various parts of our program, it may be worth while defining a type for them as in the line

```
using  vfptrTypev  = void(* )();
```

which would then allow the pointers themselves to be declared as:

```
vfptrTypev funptr;
```

Notice that the type name reflects the return type and parameter list using the *v* (for `void`) at the start and end of the name. Of course, this is only a convention, but one that helps tell at a glance the signature of the type of function that the pointer can reference. The pointer *funptr* can now contain the address of any function whose signature is `void ()`.

A function's name, without the parentheses, gives the start address of a function, so we can have the pointer *funptr* reference *Line()* using the code:

```
funptr = DrawLine;
```

Alternatively, you can also use the *address of* operator:

```
funptr = &DrawLine;
```

Now, with the function's address safely stored in the pointer, we are free to call the function using the pointer name:

```
funptr();
```

As you can see, the syntax is an exact match for the traditional function call. Another option is to call the function by dereferencing the pointer:

```
(*funptr)();
```

Activity 12.18

Create a `using` statement to set up a type for pointing to functions with the prototype

```
int functionname(int,int);
```

The program in FIG-12.5 creates three functions, *FA()*, *FB()* and *FC()* all with the same return void type and void parameter list. A pointer to this type of function is then used to execute each function in turn.

FIG-12.5

Using Function Pointers

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
void FA();
void FB();
void FC();

int main()
{
   using  vfptrTypev = void(*)();
   vfptrTypev  fptr;
   fptr = FA;
   fptr();
   fptr = FB;
   fptr();
   fptr = FC;
   fptr();
}

void FA()
{
   cout << "FA() executed\n";
}

void FB()
{
   cout << "FB() executed\n";
}

void FC()
{
   cout << "FC() executed\n";
}
```

Activity 12.19

Start a new project called *FunctionPointers* which implements the code given above. Check that the code causes each function to be executed.

Add a function called *Smaller()* which takes two `int` values and returns the smaller of the two values.

Create a pointer to this function and using the pointer to execute the function using actual parameter values 43 and 15.

Function Pointers as Parameters

The real power of function pointers comes when they are used as parameters to other functions. At this point we will have to satisfy ourselves with a trivial example to highlight the mechanism involved, but in a later chapter we will create a more useful example.

Let's assume that a program contains the following two functions:

```
// *** Returns the sum of two values ***
int Add(int no1, int no2)
{
   return (no1 + no2);
}
```

```
// *** Returns the difference of two values ***
int Sub(int no1, int no2)
{
   return (no1 - no2);
}
```

Both functions have the same signature, returning an `int` and taking two `int` parameters.

We can set up a type definition for a pointer to this function type with the line

```
using ifptrTypeii = int(*)(int,int);
```

Now, let's create a function that takes a pointer to this type:

```
// *** Returns the result produced by the helper function ***
int Operate(ifptrTypeii fptr, int no1, int no2)
{
   int result = fptr(no1,no2);
   return result;
}
```

This last function returns the result created when the function referenced by *fptr* is executed.

Activity 12.20

What value would you expect *Operate()* to return if its parameter *fptr* contained the address of:

a) *Add()* b) *Sub()*

A complete program which makes use of the code given above is shown in FIG-12.6.

FIG-12.6

Pointers to Functions

```
#include <iostream>
using namespace std;

// *** Type for pointer to int function(int,int) ***
using  ifptrTypeii  = int(*)(int,int);


// *** Function prototypes ***
int Add(int,int);
int Sub(int,int);
int Operate(ifptrTypeii,int,int);


int main()
{
   //*** Display result from Operate() ***
   //*** using the address of Add() as parameter ***
   cout << Operate(Add,12,3) << endl;
}

// *** Returns the sum of two numbers ***
int Add(int no1, int no2)
{
   return (no1 + no2);
}
```



FIG-12.6
(continued)

Pointers to Functions

```
// *** Returns the difference of two numbers ***
int Sub(int no1, int no2)
{
   return (no1 - no2);
}


// *** Returns the value produced by the helper function ***
int Operate(ifptrTypeii fptr, int no1, int no2)
{
   int result = fptr(no1,no2);
   return result;
}
```

Activity 12.21

Start a new project, *FunctionPointerPara* and implement the code given above. Run the program and observe the result produced.

Modify the code so that the address of *Sub()* rather than that of *Add()* is used as a parameter to *Operate()* then run the program again.

What result was obtained this time?

main()

Returning a Value

All the programs we have written so far define `main()` as returning an `int` value:

```
int main()
```

Writing `main()` in this way allows our program to return a value to the operating system. If our program is being run by a batch file, this means that the batch file can respond to the value returned by the program.

The tradition is to have `main()` return zero if the program has run as expected and to return some other, non-zero value if a problem has occurred.

So far none of the examples have specified a return value for `main()`. This doesn't upset most compilers, which will automatically assume zero is to be returned. However, there is no problem with us explicitly adding the line

```
return 0;
```

immediately before the closing brace of `main()`.

The program in FIG-12.7 returns either 1 or zero depending on the value entered by the user.

FIG-12.7

Returning a Value from *main()*

```
#include <iostream>
using namespace std;

int main()
{
   char key;
```



FIG-12.7
(continued)

Returning a Value from *main()*

```
   // *** Accept key ***
   cout << "Press entry key:";
   cin >> key;

   //*** Return 0 if 'q' otherwise return 1 ***
   if (key == 'q')
      return 0;
   else
      return 1;
}
```

Activity 12.22

Start a new project called *MainFunction* then enter and compile the code given above.

When we want to handle the value returned by a C++ program, then we need to run the program from within a batch file. In a Windows environment the value returned by the program is stored in a variable known as *%errorlevel%* (in Unix it's stored in *$*).

A very simple Windows batch file is shown in FIG-12.8.

FIG-12.8

A Batch File that Runs *MainFunction*

```
@echo off
MainFunction.exe
echo Value returned:  %errorlevel%
@echo on
pause
```

The batch file executes *MainFunction* (line 2) and displays the value returned by the program (line 3).

Activity 12.23

Use a simple text editor (such as Notepad) to enter the text given in FIG-12.8.

Save the file as *RunMain.bat* in the *Debug* folder of project *MainFunction* (which you created in the previous Activity).

Load *File Explorer* and double click on the file *RunMain.bat* to execute it.

When prompted for a key, press 'q' and *Enter* observing the result displayed.

Run the batch file for a second time, pressing a different key. What result is displayed this time?

Making use of the value returned by `main()` is at its most useful when writing housekeeping programs which require a batch file to respond differently to each possible value being returned.

Parameters

Not only can `main()` return a value, but it can also be defined with a set of parameters when required.

These parameters are always an integer and an array of C-style strings. Traditionally, these parameters are named as in the line below:

```
int main(int argc, char *argv[])
```

The first parameter is a count of how many strings appear in the second parameter.

If we define `main()` in this way, then the program's executable file needs to be run from a command line environment (or from a batch file) and the actual parameter values are supplied to the program by the operating system. The program in FIG-12.9 displays the values of *argc* and the contents of *argv[]*.

FIG-12.9

Handling *main()* Parameters

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
   cout << argc << " strings were passed to the program\n";
   cout << "The strings were : \n";
   for (int c = 0; c < argc; c++)
      cout << argv[c] << endl;
}
```

Activity 12.24

Start a new project called *MainParameters* then enter and compile the code given in above.

The value of *argc* will always be at least 1 as *argv[0]* will contain the name of the program being run.

From the command line environment (assuming the current directory contains the executable version of the program) we execute the program by supplying its name. This is followed by a space-separated list of parameters. For example, the line

```
MainParameters a black cat
```

would execute the file named *MainParameters.exe* and supply it with three additional parameter strings ("*a*", "*black*" and "*cat*"). These strings are held in *argv[]* and supplied to the program. The program above would then create the following output:

```
MainParameters
a
black
cat
```

Activity 12.25

Open the command line window (do this by clicking on *Start* and entering *cmd.exe* in the search box and pressing **Enter**).

Make the current directory the *Debug* folder of project *MainParameters* (use the CD command). Run the program using the line

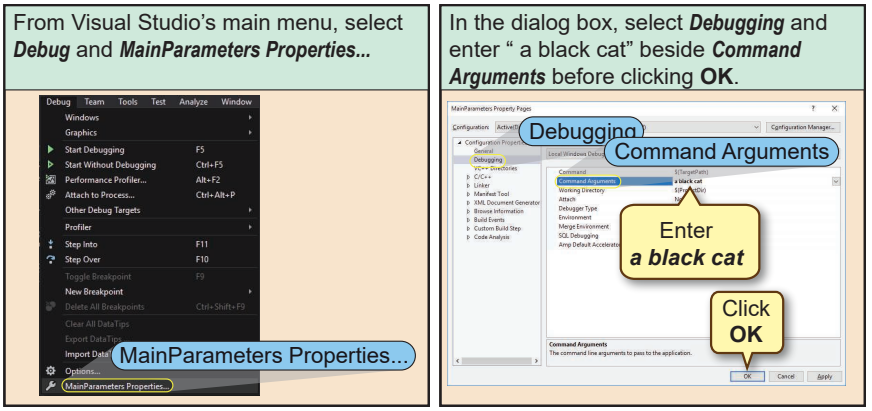
```
MainParameters a b c d
```

What output is produced?

We can also specify the parameters to `main()` from within Visual Studio using the steps shown in FIG-12.10.

FIG-12.10

Setting *main()*'s Parameters from the IDE



Summary

- Parameters can be assigned default values in the function prototype.
- Default parameter values can be overridden when the function is called.
- Inline functions replace calls to that function with the actual function code.
- Inline functions must have their definition placed before any calls to those functions.
- Use `constexpr` to request final evaluation at compile time rather than runtime.
- Static variables retain their value between calls to the function in which they are defined.
- A variable's lifetime is the block(s) of code during which the variable is allocated memory space.
- A variable's scope is the block(s) of code where the variable may be accessed directly.
- The scope resolution operator (`::`) can be used to access variables not normally in scope.
- Several functions may have the same name as long as their parameter lists differ in some permissible way. This is known as function overloading.
- Operators may also be overloaded.
- When overloading the increment (++) and decrement (--) operators two versions of each should be coded: one for pre-execution and one for post-execution.
- The post-execution versions of overloaded calls to ++ and -- must contain an unused `int` parameter.
- A recursive function is one which calls itself.
- Recursive functions often offer elegant solutions but are rarely efficient.
- A pointer may contain the start address of a function.
- When declaring a function pointer, the declaration must specify the signature

of the type of function to be referenced.

- A function pointer can be used as a parameter to another function.
- By default `main()` returns zero. This value can be accessed by the operating system.
- `main()` can be coded with two parameters. These will be assigned the number of arguments passed to `main()` and an array of C-style strings containing the arguments passed.
- Values can be passed to `main()` by executing the program from the command line where the program name is followed by the strings to be passed to `main()`.
- From within Visual Studio parameters to `main()` can be specified by:
    1. Selecting **Debug|project name Properties...**
    2. Selecting *Debugging*
    3. Entering arguments beside *Command Arguments*

Data Type Related Functions

Introduction

In this section we are going to create a series of functions used to manipulate *DateType* variables.

Although what we are about to do reflects how programming was often tackled before the introduction of object-oriented programming and therefore is totally obsolete as far as C++ programming is concerned, in a later chapter we will tackle this same subject again using an object-oriented approach and the Activities we complete here will contrast between the two approaches.

Defining the Data Structure

When we are about to create a set of functions for manipulating a data structure, the first thing we need to do is define the data components of our structure. Since this exercise is to do with the manipulation of dates, we already have that structure:

```
DateType
{
   int day;
   int month;
   int year;
};
```

Defining the Operations

At this point we want to create a formal description of the operations we will want to perform on variables of our new type. Typically, these will be written in the form of mini-specs, but in this case we'll settle for a more informal description of each operation (some of which we created previously).

Previously Created Operations

```
int GetDayOfWeek(const DateType& dt)
int GetDayOfWeek(const int d, const int m, const int y)
```

This overloaded function returns an integer value representing the day of the week on which date *dt* or *d/m/y* fell.

```
bool IsValidDate(const DateType& dt)
bool IsValidDate(const int d, const int m, const int y)
```

Returns *true* if *dt* (or *d/m/y*) forms a valid date after 1752, otherwise *false* is returned.

Before we start adding new operations that manipulate our *DateType* structure, we'll bring together what we have so far in a new project. This is shown in FIG-12.11.

FIG-12.11

DateType and
Associated Functions

```
#include <iostream>
using namespace std;

//*** User defined types ***
struct DateType
{
   int day;
   int month;
   int year;
};

//*** Function prototype ***
bool IsValidDate(const DateType& dt);
bool IsValidDate(const int, const int, const int);
int GetDayOfWeek(const DateType& dt);
int GetDayOfWeek(const int, const int, const int);

void main()
{

}

//*** Displays the day of the week on which dt fell ***
int GetDayOfWeek(const DateType& dt)
{
   if (!IsValidDate(dt))
      return -1;
   //*** Calculate modified date ***
   int M = (dt.month + 9) % 12 + 1;
   int modifiedyear = dt.year - M / 11;
   int C = modifiedyear / 100;
   int Y = modifiedyear % 100;
   //*** Return day of week ***
   return ((static_cast<int>(2.6 * M - 0.2) + dt.day + Y + Y / 4 +
   ↳C / 4 - 2 * C) % 7 + 7) % 7;
}

//*** Displays day of the week on which day/month/year fell ***
int GetDateOfWeek(const int day, const int month, const int year)
{
   if (!IsValidDate(day,month,year))
      return -1;
   //*** Calculate modified date ***
   int M = (month + 9) % 12 + 1;
   int modifiedyear = year - M / 11;
   int C = modifiedyear / 100;
   int Y = modifiedyear % 100;
   //*** Return day of week ***
   return ((static_cast<int>(2.6 * M - 0.2) + day + Y + Y / 4 + C
   ↳/ 4 - 2 * C) % 7 + 7) % 7;
}


//*** Check date is valid ***
bool IsValidDate(const DateType& dt)
{
   bool result{ true };
   //** If invalid year, exit **
   if (dt.year <= 1752)
      result = false;
```



FIG-12.11
(continued)

DateType and
Associated Functions

```
   //** If invalid month, exit **
   if (dt.month < 1 || dt.month > 12)
      result = false;
   //** If invalid day, exit **
   int daysinmonth[]{ 0,31,28,31,30,31,30,31,31,30,31,30,31 };
   //* If leap year and 1 to days in Feb *
   if ((dt.year % 4 == 0&&dt.year % 100 != 0)||(dt.year % 400==0))
      daysinmonth[2]++;
   if (dt.day < 1 || dt.day > daysinmonth[dt.month])
      result = false;
   //*** Return date's validity ***
   return result;
}


//*** Check date is valid ***
bool IsValidDate(const int day, const int month, const int year)
{
   bool result{ true };
   //** If invalid year, exit **
   if (year <= 1752)
      result = false;;
   //** If invalid month, exit **
   if (month < 1 || month > 12)
      result = false;
   //** If invalid day, exit **
   int daysinmonth[]{ 0,31,28,31,30,31,30,31,31,30,31,30,31 };
   //* If leap year and 1 to days in Feb *
   if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
      daysinmonth[2]++;
   if (day < 1 || day > daysinmonth[month])
      result = false;
   //*** Return date's validity ***
   return result;
}
```

Activity 12.26

Start a new project called *DateLibrary* and enter the code given above.

Check that the code compiles.

We're now ready to start adding new operations.

IsLeapYear(const int y)

Actually, we have already produced a function called *IsLeapYear()* but this one will be simpler, returning *true* if *y* is a leap year and *false* if it is not. The code is:

```
//*** Returns true if y is a leap year   ***
bool IsLeapYear(const int year)
{
   bool result{ true };
   if (year < 1753)
      result = false;
   else if (!(year % 4 == 0&&year % 100 != 0)||(year % 400==0))
      result = false;
   return result;
}
```

Activity 12.27

Modify *DateLibrary* by adding the code for *IsLeapYear()*.

Notice that both versions of IsValidDate() *check for a leap year in order to adjust the number of days in February*.

Adjust the code in both versions of this function so that they make use of the *IsLeapYear()* function.

SetDate(const int d, const int m, const int y)

The problem with creating a *DateType* variable,

```
DateType d1;
```

and assigning values to the fields in that variable,

```
d1.day = 23;
d1.month = 11;
d1.year = 1963;
```

is that the code can easily assign an invalid value:

```
d1.day = 42;
```

This won't cause an obvious problem immediately, but when further statements try to make use of the stored date (such as converting it to a day of the week), then errors will be produced.

To avoid the possibility of assigning an invalid value to a *DateType* variable, we can create a function specifically designed to assign a value to a variable of that type.

If appropriately coded, this will ensure that only a valid date can be stored in a *DateType* variable. The code for such a function appears below:

```
//*** Sets the date in date to d/m/y ***
DateType SetDate(int d, int m, int y)
{
   DateType result{1,1,2001};
   // *** Days in each month of year ***
   int daysinmonth[] = {0, 31,28,31, 30,31,30, 31,31,30,
   ↳31,30,31};

   //*** If month invalid, exit ***
   if (m < 1 || m > 12)
      return result;
   //*** Add 1 to days in February if leap year ***
   if (IsLeapYear(y))
      daysinmonth[2]++;
   //*** If days in month invalid, exit ***
   if (d <1 || d > daysinmonth[m])
      return result;
   //*** If year less than 1, exit ***
   if (y < 1)
      return result;
   //*** Assign date ***
   result.day = d;
   result.month = m;
   result.year = y;
   return result;
}
```

Examining the code, we can see that the day, month and year are checked before being assigned to the result and if any are found to be invalid the returned date is set to 1st January 2001.

Activity 12.28

Modify *DateLibrary* by adding the code for *SetDate()* and check that it works as expected when the parameters form a valid date and when they form an invalid one.

We would now expect any programmer who creates a *DateType* variable to use *SetDate()* to assign it a value. That would guarantee that the variable MUST contain a valid date.

It also means that we could remove the valid date check from the *GetDayOfWeek(DateType)* function since we know the date being passed in must be valid.

Unfortunately, there is no way of forcing the programmer to use *SetDate()* – they can still access the fields of the variable directly and so bypass *SetDate()*. In a later chapter when we discuss object-oriented programming, we'll see how to stop direct access to the fields within a date.

GetDaysBetween(const DateType& dt1, const DateType& dt2)

Our next function, *GetDaysBetween()* will take two *DateType* values (*dt1* and *dt2*) as parameters and return an `int` value giving the number of days between the two dates (*dt1* - *dt2*). The returned value may be negative if the second date comes after the first.

This isn't as easy a problem as it may first appear since it has to take into account such things as the number of days in a leap year. In fact, the easiest way to tackle this function's code is to convert each date into the number of days from a fixed date far in the past. That way we will just be subtracting two integer values.

Luckily for us, such a formula is well-known (in certain circles) which calculates how many days lie between a given date and 1st January 4713BC. This number is known as the **Julian Day Number** and has the following formula:

```
a = [(14 - month)/12]
m = month + 12a - 3
y = year + 4800 - a

JDN = day + [(153m + 2) / 5] + 365y + [y/4] - [y/100] + [y/400]  - 32045
                                                  [ ] means truncated integer value
```

where:

day, month, year	specify the date whose Julian Day Number is to be found.

Activity 12.29

Before we tackle the code for GetDaysBetween(), *we'll create a helper function called* DateToJDN() *which takes a* const DateType& *value and returns a* `long long int` *giving the Julian Day Number for that date.*



Activity 12.29 (continued)

In *DateLibrary,* add the necessary code for *DateToJDN()* and test it by checking that two consecutive dates return numbers which differ by 1. Test it again with two dates exactly one year apart checking that the values returned are 365 (or 366) days apart.

The *GetDaysBetween()* function can now call *DateToJDN()* to convert the two dates to integer values. All that is required after that is simple subtraction.

Activity 12.30

Modify *DateLibrary* by adding the code for the *GetDaysBetween()* function which returns a `long int` value, and check that it works as expected.

AddDays(const DateType& dt, const int days)

This function returns a *DateType* value which is calculated by adding *days* to *dt*.

The first stage in performing this operation is easy enough. We can use *DateToJDN()* to convert *dt* to an integer value and then simply add *days*. However, we then need to convert this new Julian Day Number back into a regular date. Again, luckily, the formula for doing this is well known:

```
f = jdn + 1401 + [(([(4 * jdn + 274277)/146097])  *3)/4 ]-38
e = 4f + 3
g = [(e mod 1461)/4]
h = 5g + 2
day = [(h mod 153)/5] + 1;
month = ([h/153] + 2) mod 12 + 1
year = [e/1461] - 4716 + [(12 + 2 - month)/12]
```

We'll make use of this formula in another helper function which can be called *JDNToDate()*. The new function is coded as:

```
// *** Converts a JDN to a date ***
DateType JDNToDate(const long jdn)
{
   DateType date;

   int f = jdn+1401+(((4 * jdn + 274277) / 146097) * 3)/4 - 38;
   int e = 4 * f + 3;
   int g = (e % 1461) / 4;
   int h = 5 * g + 2;
   date.day = (h % 153) / 5 + 1;
   date.month = (h / 153 + 2) % 12 + 1;
   date.year = e / 1461 - 4716 + (12 + 2 - date.month) / 12;
   return date;
}
```

This allows us to code *AddDays()* as:

```
// *** Returns a date which is a specified number of ***
// *** days after a given date                        ***
DateType AddDays(const DateType& d, const int days)
{
   int totaldays = DateToJDN(d) + days;
   DateType result = JDNToDate(totaldays);
   return result;
}
```

Activity 12.31

Modify *DateLibrary* by adding the code for *JDNToDate()* and *AddDays()*.

Adjust the code in `main()` to test *AddDays()*.

Operator +

Rather than use *AddDays()* to add a number of days to a date, we could define the same code to the + operator which would allow us to write statements such as

```
dt2 = dt1 + 3;
```

In fact we'll overload the + operator so that we can also write statements such as

```
dt2 = 3 + dt1;
```

The code for both routines is really just a duplication of what we already have within *AddDays()*:

```
// *** Returns a date equal to (dt+days) ***
DateType operator+(const DateType& dt, const int days)
{
   int totaldays = DateToJDN(dt) + days;
   DateType result = JDNToDate(totaldays);
   return result;
}


// *** Returns a date equal to (dt+days) ***
DateType operator+( const int days, const DateType& dt )
{
   int totaldays = DateToJDN(dt) + days;
   DateType result = JDNToDate(totaldays);
   return result;
}
```

Activity 12.32

Modify *DateLibrary* by adding the code for both versions of the + operator.

Adjust the code in `main()` to test the new operations.

Operator -

The first version of the code for operator - will be identical to *GetDaysBetween()* but we can also have a second version which subtracts a number of days from a given date. This means we can write statememts such as

```
int difference = dt1 - dt2;
```

or

```
DateType dt2 = dt1 - 15;
```

Notice that it doesn't make sense to try and subtract a date from an integer, so unlike the + operator, we won't need to write a function with the parameters in the order `(int, DateType)`.

The code for our new functions is:

```
// *** Returns the number of days between two days ***
long operator-(const DateType& dt1, const DateType& dt2)
{
   return (DateToJDN(dt1) - DateToJDN(dt2));
}


// *** Returns a date equal to (dt1 - days) ***
DateType operator-(const DateType& dt, const int days)
{
   int totaldays = DateToJDN(dt) - days;
   DateType result = JDNToDate(totaldays);
   return result;
}
```

Activity 12.33

Modify *DateLibrary* by adding the code for both versions of the - operator.

Adjust the code in `main()` to test the new operations.

Operator == and Operator <

The last two operators we are going to add are the relational operators == and <. These will take two *DateType* values and each return a `bool` value.

To test for equality all that is needed is that *DateToJDN()* returns the same number for both dates. For *less than*, the function will return *true* if the *DateToJDN()* value for the first date is less than the *DateToJDN()* value of the second date.

Activity 12.34

Modify *DateLibrary* by creating the code for the == and < operators..

Adjust the code in `main()` to test the new operations.

Summary

- A set of functions are often created in order to manipulate the contents of a user-defined structure.
- Manipulating data via a set of functions can help eliminate invalid values being stored.

Function Templates

Introduction

In the last chapter we created a *Swap()* function to swap the contents of two *TimeType* variables. The code for this function was

```
//*** Swaps the contents of v1 and v2 ***
void Swap(TimeType &v1, TimeType &v2)
{
   TimeType temp{ v1 };
   v1 = v2;
   v2 = temp;
}
```

If we wanted to create a similar routine to operate on two *DateType* variables, the code would be

```
//*** Swaps the contents of v1 and v2 ***
void Swap(DateType &v1, DateType &v2)
{
   DateType temp{ v1 };
   v1 = v2;
   v2 = temp;
}
```

Notice that we have exactly the same code in both routines and that the only difference is the parameter and local variable types (*TimeType* and *DateType*).

And if, for some reason, we wanted to write a third *Swap()* function to swap the contents of two `int` variables, again we would create the same code with only the types being different.

It is to avoid such near duplicate functions that C++ introduced the concept of templates.

Creating a Function Template

When we use templating to define a function, we deliberately avoid specifying the variable type involved in the function parameters, return type, and/or local variables. This is known as **deferred typing**. The actual type information is not supplied until the function is actually called. To show how this works, we'll write a templated version of the *Swap()* function.

Step 1 is to start with the `template` keyword:

```
template
```

This is followed by the keyword `typename` followed by the identifier we are going to use in place of the standard type specifier (such as *TimeType*, *DateType* or `int`). We can choose any identifier we like, but traditionally we use `T`. These two terms are enclosed in angled brackets. So our first line expands to become:

```
template <typename T>
```

On the next line we start on the function's heading by specifying the function's return type. Since, in the case of *Swap()* every version specifies `void`, we can use that here too. This is followed by the function name:

```
template <typename T>
void Swap
```

Next comes the parameter list but now, instead of the standard type names, we use `T`:

```
template <typename T>
void Swap(T& v1, T& v2)
```

Notice that we still have to include the ampersand to show that we want to use a reference parameter. We'd also need to use the asterisk if using pointer parameters, `[]` for array parameters and `const` for constant parameters.

Now we can code the body of the function. The only change needed here is where the local variable type changes in association with a parameter type change. In this case, that means the variable *temp*:

```
template <typename T>
void Swap(T& v1, T& v2)
{
   T temp{ v1 };
   v1 = v2;
   v2 = temp;
}
```

And that's the coding complete.

Like any other function, we'll need to create a function prototype and this needs to include the template line as well as the first line of the function, although these are often shown on a single line:

```
template <typename T> void Swap(T&, T&);
```

Activity 12.35

Create a function template for a function called *Bigger()* which returns a copy of the larger of two values passed to the function (const references to *n1* and *n2*).

Activity 12.36

Start a new project called *Templates* and implement the function template for *Swap()* and *Bigger()*. Place no code in `main()` at this stage.

Using a Function Template

Normally, function templates are called in exactly the same way as a standard function. The compiler examines the actual parameter types used in the call and automatically creates a version of the function for that type (this is known as **function instantation**).

For example, if *tm1* and *tm2* are of type *TimeType*, *dt1* and *dt2* are of *DateType* and *num1* and *num2* are `int` types, then the calls

```
Swap(tm1,tm2);
Swap(dt1,dt2);
Swap(num1,num2);
```

will create versions of *Swap()* in which *T* is replaced by *TimeType*, *DateType* and `int` respectively.

The program in FIG-12.12 demonstrates the use of *Swap()* and *Bigger()*.

FIG-12.12

Using Function Templates

```
#include <iostream>
using namespace std;

//*** User-defined types ***
struct DateType
{
   int day;
   int month;
   int year;
};

struct TimeType
{
   int hrs;
   int mins;
   int secs;
};

//*** Function templates ***
template <typename T> void Swap(T&, T&);
template <typename T> T Bigger(const T&, const T&);

int main()
{
   DateType dt1{ 1,1,2001 };
   DateType dt2{ 14,2,2019 };
   TimeType tm1{ 2,45,56 };
   TimeType tm2{ 1,17,33 };
   int num1{ 17 };
   int num2{ 23 };
   Swap(dt1, dt2);
   Swap(tm1, tm2);
   Swap(num1, num2);
   cout << "dt1 = " << dt1.day << '/' << dt1.month << '/'
   ↳<< dt1.year << endl;
   cout << "dt2 = " << dt2.day << '/' << dt2.month << '/'
   ↳<< dt2.year << endl;
   cout << "tm1 = " << tm1.hrs << '/' << tm1.mins << '/'
   ↳<< tm1.secs << endl;
   cout << "tm2 = " << tm2.hrs << '/' << tm2.mins << '/'
   ↳<< tm2.secs << endl;
   cout << "num1 = " << num1 << endl;
   cout << "num2 = " << num2 << endl;
   cout << "Bigger value is " << Bigger(num1, num2) << endl;
}

//*** Swaps the contents of v1 and v2 ***
template<typename T>
void Swap(T& v1, T& v2)
{
   T temp{ v1 };
   v1 = v2;
   v2 = temp;
}

//*** Returns a copy of the larger of n1 and n2 ***
template<typename T>
T Bigger(const T& n1, const T& n2)
{
   return (n1 > n2) ? n1 : n2;
}
```



If we had overloaded the > operator for *DateType* and *TimeType*, *Bigger()* would have worked for those data types too.

Notice that we could only use our `int` variable when calling *Bigger()*. However, if the program contained code to overload the < operator (which is used in *Bigger()*) for the *DateType* and *TimeType* structures, we could used *Bigger()* on those too.

Activity 12.37

Expand project *Templates* to match the code given above and check that the program operates as expected.

Other Considerations

Explicit Instantiation

If we were to call *Bigger()* with two different parameter types, for example using the line

```
cout << Bigger(12.7, 83);
```

we'd get a compilation error since the compiler is expecting both parameters to be of the same type and here we have a `double` and an `int`.

The simplest way round this problem is to explicitly state what types are to be used. We do this with a line such as

```
cout << Bigger<double>(12.7,83);
```

Note the instruction to the compiler to use `double` in place of type *T*.

Multiple Templated Types

When creating a function template we're not restricted to having a single undefined type – we're free to have as many different types as needed. For example, let's say that function *F()* takes three parameters which may be of differing types and the value returned may also change. In this case we can use different identifiers for each unspecified type. Traditionally the identifiers `T1`, `T2`, `T3`, etc. are used. So our heading for *F()* would be

```
template <typename T1, typename T2, typename T3, typename T4>
T1 F(T2 x, T3 y, T4 z)
{
   //Function body
}
```

Summary

- Function templates are a way of creating functions where the type specification for parameters, return values and/or local variables can be deferred until the function is actually called.
- Function templates stop the need for coding multiple versions of the same routine to cater for differing data types.
- Use the keyword `template` when creating a function template.
- Traditionally, the terms T, T1, T2, etc. are used in place of data types when defining a template.
- When using a function template, the compiler can often determine the actual data types required from the actual parameters used.

- An alternative method of specifying the data types used in a function temple is to enclose the function types in angle brackets after the function name.

Variadic Functions

Introduction

On occasion it would be useful to be able to create a function with an unlimited number of parameters just like `printf()`. These are known as **variadic functions**. And, if the type of values involved is unknown, then we can create a variadic function template, usually shortened to **variadic template**.

Implementation

Parameters of a Single Type

Often, a variadic function's parameter list starts with one or more named parameters followed by a comma and an ellipsis. For example, let's say we want to create a function *Sum()* which returns the sum of a set of integers. Our first parameter would be a count of how many numbers were to follow in the parameter list, giving us a function prototype of

```
int Sum(int,...);
```

When we come to code this function, the heading would be:

```
int Sum(int count,...)
```

The ellipsis (...) represents the unknown set of parameters which are termed a **function parameter pack**. The obvious problem, when we come to code the body of the function, is that we have no names for the parameters following *count*. But, of course, C++ is equipped to handle this problem. The *cstdarg* header file contains definitions of the various tools designed to help, so when creating a variadic function we must add the line

```
#include <cstdarg>
```

to the start of our program.



va = **V**ariable **a**rguments

In our function we must create a `va_list` variable (`va_list` type is defined in *cstdarg*):

```
va_list args;
```

This variable will hold all the information necessary to access the unnamed parameters in our function.

Next we need to call the function `va_start()` which will enable access to our parameters. This function requires two parameters of its own: our `va_list` variable and the last named variable in our own function's parameter list:

```
va_start(args,count);
```

Finally, we can now begin to access the unnamed parameters, This requires a call to the `va_arg()` function which will return a value from the unnamed parameter list. Each call to this function returns the next value in the list.

`va_arg()` requires two parameters: our `va_list` variable and the type the fetched value is to be converted to. Since all of our values in a call to *Sum()* will be integers (of type `int`), a typical to `va_arg()` will be:

```
value = va_arg(args,int);
```

When our *Sum()* function has processed all of the unnamed parameters, a call to

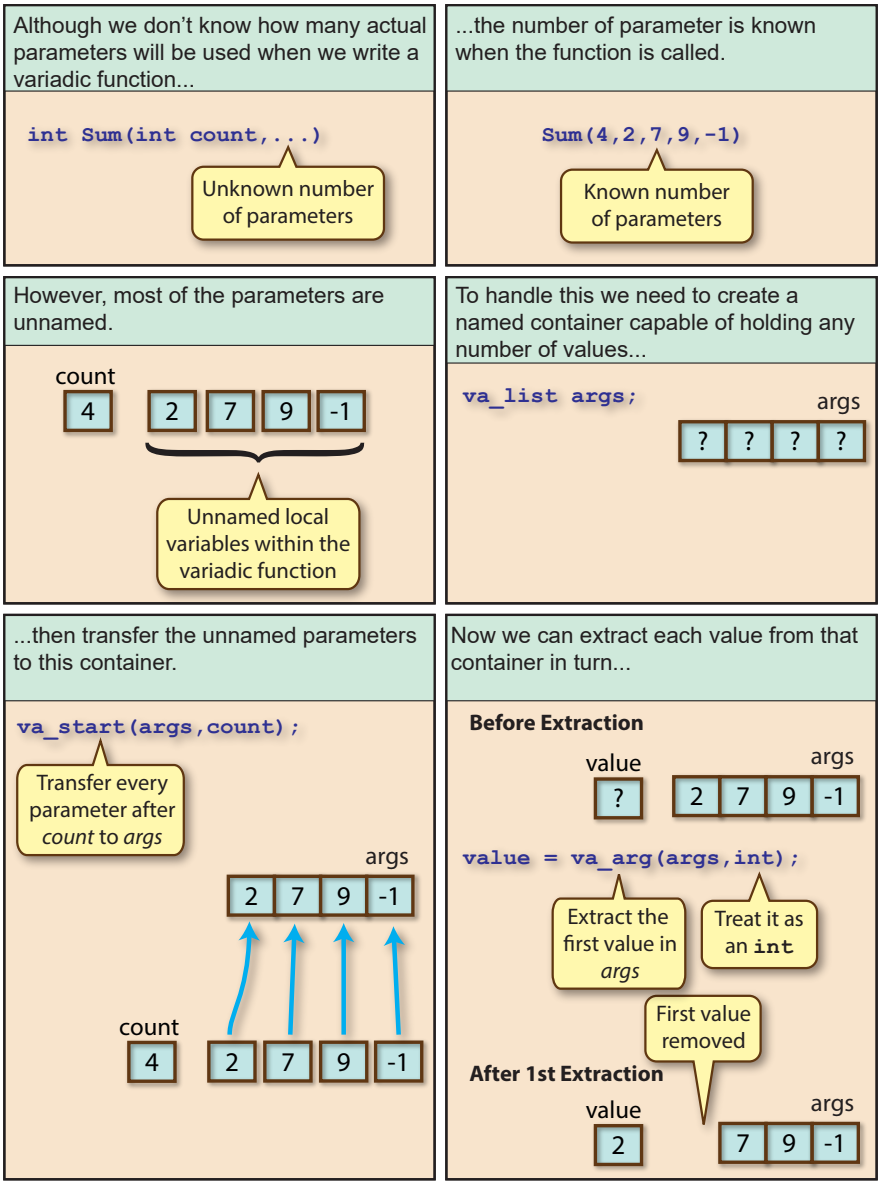
function `va_end()` performs a cleanup our `va_list` variable. This will be coded as

```
va_end(args);
```

The main part of the process is explained in FIG-12.13.

FIG-12.13

How a Variadic Function Operates



Now we are ready to code our *Sum()* function as shown in FIG-12.14.

FIG-12.14

A Variadic Function

```
int Sum(int count,...)
{
   va_list args;
   va_start(args, count);
   int result = 0;
   for (int c = 1; c <= count; c++)
      result += va_arg(args, int);
   va_end(args);
   return result;
}
```

Activity 12.38

Create a new project, *TestVariadic,* containing the function given above and write a `main()` section to test calls to *Sum()* with differing number of parameters.

What happens if the first parameter, giving the count of the values that follow, is wrong?

Multiple Parameter Types

Things are a bit more difficult if we want our unnamed parameters to be of differing types. The problem being that in order to retrieve a value from the parameter list we need to call `va_arg()` and that requires the parameter type to be given.

If the parameters to our function are in a predictable order then this isn't really a problem. For example, if we were to create a function that had parameters consisting of names and ages as in the call:

```
ListNameAndAge(3,"Maggie Tulliver",18,"Lucy Deane", 38,
↳"Philip Wakem",23);
```

The coding for this function is given in FIG-12.15.

FIG-12.15

A Variadic Function with Varying Parameter Types

```
void ListNameAndAge(int count, ...)
{
   va_list args;
   va_start(args, count);
   char* name;
   int age;
   for (int c = 1; c <= count; c++)
   {
      name = va_arg(args, char*);
      age = va_arg(args, int);
      cout << name << "   " << age << endl;
   }
   va_end(args);
}
```

Activity 12.39

Add the code for *ListNameAndAge()* to *TestVariadic* and change the code in `main()` to call the function with the line shown above.

Where the parameter types are mixed in an unpredictable order we could use a string as the first parameter, with each letter of the string giving an indication of the next parameter type. For example, if our function is call *DisplayAll()* we might call it with a line such as

```
DisplayAll("cidsi", 'X', 23, 12.78, "Hello", 42);
```

where "cidsi" tells the function that the parameters that follow are: a `char` (c), an `int` (i), `double` (d), a string(s), and another `int` (i).

The code for this function is shown in FIG-12.16.

FIG-12.16

Handling a Variadic Function with Varying Parameter Types

```
void DisplayAll (const char * format, ...)
{
   va_list args;
   va_start(args, format);
   int idx = 0;
   while (format[idx] != '\0')
   {
      switch (format[idx])
      {
         case 'c':
            cout << va_arg(args, char) << endl;
            break;
         case 's':
            cout << va_arg(args, char*) << endl;
            break;
         case 'i':
            cout << va_arg(args, int) << endl;
            break;
         case 'd':
            cout << va_arg(args, double) << endl;
            break;
      };
      idx++;
   }
   va_end(args);
}
```

Activity 12.40

Add the code for *DisplayAll()* to *TestVariadic* and change the code in `main()` to call the function with the line shown earlier.

Variadic Templates

The trouble with the above version of *DisplayAll()* is that it depends on the initial parameter containing the correct sequence of characters to match the type of the actual parameters that follow it.

A better approach is to create a variadic function template. The first line of such a template is typically written as

```
template <typename T, typename... Ts>
```

which tells us that the first parameter is of type *T* and that there is an unlimited number of parameters of varying types, *Ts*. Here the ellipsis represent a **template parameter pack** allowing the function to be called with an unlimited number of actual parameters of varying types.

There is nothing special about the names *T* and *Ts*. They are simply the names that are most frequently used. Now comes the code for the function itself:

```
void DisplayAll2(T v1, Ts... v2)
{
   cout << v1 << endl;
   DisplayAll2(v2...);
}
```

Three points of note:

- *v1* is used to represent the name of the first parameter and *v2* is used to represent the name of all the remaining parameters.
- There is no need for `va_list` and `va_start`() or `va_arg()`.
- The function calls itself in the final line of the code.

Looking at the recursive call within the code, we can see that one less parameter is used with only *v2...* being passed to the function. *v2* represents the first and ... the rest of the remaining parameters.

On each iterative call to the *DisplayAll2()* function, the first of the actual parameters is stored in *v1* and the remainder are *v2*. This means on each call there is one less actual parameter. Eventually, of course, there will be zero parameters. Because we will eventually end up with zero parameters, we need to overload *DisplayAll2()* by coding a zero-argument version:

```
void DisplayAll2(){}
```

And that is all that is required. FIG-12.17 shows a complete program which is equivalent to that in FIG-12.16.

FIG-12.17

Using Variadic Function Templates

```
#include <iostream>
using namespace std;

void DisplayAll2(){}

template <typename T, typename... Types>
void DisplayAll2(T v1, Types... v2)
{
   cout << v1 << endl;
   DisplayAll2(v2...);
}


int main()
{
   DisplayAll2('X', 23, 12.78, "Hello", 42);
}
```

Activity 12.41

Add the code for *DisplayAll2()* to *TestVariadic* and change the code in `main()` to call the function with the parameters shown above.

Fold Expressions

Not all variadic templates define a recursive function. Another commonly used feature in these situations is **fold expressions**. A fold expression is a succinct way of applying a binary operation (such as `+`, `-`, `*`, or `/`) to a parameter list of unknown length.

The simplest way to understand fold expressions is to see an example of it in use.

Let's say we want to create a function which calculates the sum of a sequence of values of unknown length and type. We start by defining the template types:

```
template <typename... Ts>
```

followed by the function heading:

```
auto Sum(Ts... vs)
```

vs represents the – as yet unknown – sequence of parameters (the parameter pack).

Notice that the return type is given as `auto` since we cannot know exactly what type will be returned until the function is called.

Finally, we have the function body:

```
auto Sum(Types...vs)
{
   return (... + vs);
}
```

The magic here is the fold expression

```
... + vs
```

which the compiler takes as an instruction to apply the + operation to every argument in the actual parameter list in order to produce the return value. The first argument when expanding the expression (unfolding the expression) is assumed to be zero, so

```
... + vs
```

is actually interpreted by the compiler as

```
0 + ... + vs
```

However, we are free to explicitly state an initial term using any value we chose:

```
return (10 + ... + vs);
```

Activity 12.42

Start a new project called *TestFoldExpression* and implement *Sum()* as shown above. Test the function by using the call `Sum(1,3,7);` What result is produced?

Modify the fold expression so that the initial value in the expression is 10 rather than the default zero. How does this effect the result?

The order of the elements can also be changed. For example, it is equally valid to write

```
   vs + ...
```

or

```
   vs + ... + 10
```

Arithmetic operators are not the only ones which can be used in a fold expression. A complete list of available operators are shown in FIG-12.18.

FIG-12.18

Operators Available for Use in Fold Expressions



The function *Display()* (see below) uses the `<<` operator in its fold expression in order to display all of the values listed as parameters

```
template <typename... Ts>
void Display(Ts ...vs)
{
   (cout << ... << vs) << endl;
}
```

Notice the need to enclose the part that is expanded in parentheses so that a new line is taken only after all the values have been displayed.

Activity 12.43

In *TestFoldExpression,* add *Display()* as shown above and test it with the call `Display(1,2,3,4,5,6);`

sizeof...()

Previously, we used `sizeof()` to discover the number of bytes occupied by a variable or type. For example:

```
cout << sizeof(int) << endl;
```

The `sizeof...()` function, on the other hand, can be used to discover how many items are in the parameter pack.

Activity 12.44

In *TestFoldExpression's Sum()* function, start with the line

```
cout << "There are " << sizeof...(vs) << " parameters\n";
```

Test the updated function by calling `Sum()` with the following parameters

a) 1, 3, 7
b) 2, -9, 4, 2, 8

See the end of Chapter 19 for another situation where a fold expression can prove useful.

Summary

- A variadic function is one which can handle a varying number of parameters.
- When declaring the function use an ellipsis (`...`) to represent an unknown number of parameters.
- At least one parameter before the ellipsis should be named.
- Within the function we must create a variable of type `va_list` which will contain the unnamed actual parameters passed to the function.
- Use `va_start()` to transfer the unnamed actual parameters to `va_list` variable.
- Use `va_arg()` to access the next unnamed value in `va_list` variable.
- Use `va_end()` to delete the space allocated to `va_list`.
- In order to use `va_list` and `va_start()`, `va_arg(), and va_end()` a program must contain the line

```
#include <cstdarg>
```

- By defining a recursive function template of the function we can eliminate the need for `va_list`, `va_start()` and `va_arg()`.
- A fold expression can be used to apply a binary operation to a parameter list of unknown length (at the time of coding).
- Use `sizeof...()` to discover the number values in a parameter pack.

Solutions

Activity 12.1

Modified code for *UsingFunctions (default *)*:

```
#include <iostream>
using namespace std;

//*** Function prototype ***
void DrawLine(int, char = '*');

int main()
{
  DrawLine(20);
}

//*** Draws a line containing length ch chars ***
void DrawLine(int length, char ch)
{
  // *** Draw the line ***
  for (int c = 1; c <= length; c++)
    cout << ch;
  cout << '\n';
}
```

Modified code for *UsingFunctions (default 10,*)*:

```
#include <iostream>
using namespace std;

//*** Function prototype ***
void DrawLine(int=10, char = '*');

int main()
{
  DrawLine();
  DrawLine(6);
  DrawLine(8, '_');
}

//*** Draws a line containing length ch chars ***
void DrawLine(int length, char ch)
{
  // *** Draw the line ***
  for (int c = 1; c <= length; c++)
    cout << ch;
  cout << '\n';
}
```

Activity 12.2

Modified code for *DOWFunction*:

Prototypes changes:

```
bool IsValidDate(const DateType&);
bool IsLeapYear(const int);
```

Function code:

```
//*** Returns true if year is a leap year after 1752
↳***
  inline bool IsLeapYear(const int year)
  {
    return (((year % 4 == 0 && year % 100 != 0) ||
    ↳(year % 400 == 0)) && year > 1752);
  }
```

Change to *IsValidDate()*:

```
//*** Check date is valid ***
bool IsValidDate(const DateType& dt)
{
  bool result{ true };
  //** If invalid year, exit **
  if (dt.year <= 1752)
    result = false;;
  //** If invalid month, exit **
  if (dt.month < 1 || dt.month > 12)
    result = false;
  //** If invalid day, exit **
  int daysinmonth[]{ 0,31,28,31,30,31,30,31,31,30,
  ↳31,30,31 };
  //* If leap year and 1 to days in Feb ***
  if (IsLeapYear(dt.year))
    daysinmonth[2]++;
  if (dt.day < 1 || dt.day > daysinmonth[dt.month])
    result = false;
  //*** Return date's validity ***
  return result;
}
```

Code for main() (invalid date):

```
void main()
{
  //*** Get date ***
  DateType date{ 29,2,2100 };
  if (IsValidDate(date))
    cout << "Valid date\n";
  else
    cout << "Invalid date\n";
}
```

Activity 12.3

The program runs and displays the array size as 10.

Removing `constexpr` causes the compilation to fail because the size of an array must be given as a constant – that is, a value that can be determined at compile time.

Modified code for *TestConstExpr*:

```
#include <iostream>
using namespace std;

constexpr int Add(int a, int b)
{
  return a + b;
}

int main()
{
   int list[Add(6, 4)]{2,4,6,8};
   int seq[list[2]]{};
   cout << size(list) << endl;
}
```

This version of the program fails for the same reason as before – the size specified for *seq* is not a constant.

Changing the first line in `main()` to

```
constexpr int list[Add(6, 4)]{2,4,6,8};
```

solves this problem, allowing the program to compile and run. The new array, *seq*, has six elements – the value stored in *list[2]*.

Activity 12.4

The original version of the program displays the messages

```
Counting() called 1 times
Counting() called 2 times
...
Counting() called 10 times
```

With the keyword `static` removed the message

```
Counting() called 1 times
```

is displayed 10 times.

Activity 12.5

Code for *Fibonacci*:

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
unsigned long long GetNextFibonacci();

int main()
{
  do
  {
```

```
    cout << GetNextFibonacci() << " ";
  }
  while (getchar() != 'q');
}


//*** Returns next number in Fibonacci series ***
//*** Starts with the third number            ***
unsigned long long GetNextFibonacci()
{
  static unsigned long long before2{ 0 };
  static unsigned long long before1{ 1 };
  unsigned long long result = before2 + before1;
  before2 = before1;
  before1 = result;
  return result;
}
```

Activity 12.6

Since the same variable, *num*, can be accessed in **`main()`** and *funcA()* the messages displayed are:

```
In main() num = 12
In functA() num = 12
```

Activity 12.7

With a local variable having the same name as the global variable, the program selects the most recently defined when executing *functA()*. That means it is the value in the local variable that is displayed by *functA()*:

```
In main() num = 12
In functA() num = 20
```

Activity 12.8

Modified code for *functA()*:

```
void functA()
{
  int num{ 20 };
  cout << "In functA() num = " << ::num << endl;
}
```

With the global variable being accessed within *functA()* the display reverts to the original

```
In main() num = 12
In functA() num = 12
```

Activity 12.9

Modifications to *DOWFunction*:

Prototypes:

```
bool IsValidDate(const int, const int, const int);
```

Overloaded *IsValidDate()* code:

```
//*** Check date is valid ***
bool IsValidDate(const int day, const int month,
const int year)
{
  bool result{ true };
  //** If invalid year, exit **
  if (year <= 1752)
    result = false;;
  //** If invalid month, exit **
  if (month < 1 || month > 12)
    result = false;
  //** If invalid day, exit **
  int daysinmonth[]{ 0,31,28,31,30,31,30,31,31,30,
  ↳31,30,31 };
  //* If leap year and 1 to days in Feb ***
  if (IsLeapYear(year))
    daysinmonth[2]++;
  if (day < 1 || day > daysinmonth[month])
    result = false;
  //*** Return date's validity ***
  return result;
}
```

```
main():

  void main()
  {
    //*** Get date ***
    DateType date{ 29,2,2100 };
    if(IsValidDate(29,2,2100))
      cout << "Valid date\n";
    else
      cout << "Invalid date\n";
    if (IsValidDate(date))
      cout << "Valid date\n";
    else
      cout << "Invalid date\n";
  }
```

Activity 12.10

Modified code for *TimeFunction*:

```
#include <iostream>
using namespace std;

//*** User defined types ***
struct TimeType
{
  int hrs;
  int mins;
  int secs;
};

//*** Function prototypes ***
TimeType AddTimes(const TimeType &t1,
↳const TimeType &t2);
TimeType operator+(const TimeType&,
↳const TimeType&);
TimeType operator+(const TimeType&, const int );


void main()
{
  TimeType tm1, tm2;

  //*** Input times ***
  cout << "Enter first time (hh:mm:ss) : ";
  scanf("%d:%d:%d", &tm1.hrs, &tm1.mins, &tm1.secs);
  cout << "Enter second time (hh:mm:ss) : ";
  scanf("%d:%d:%d", &tm2.hrs, &tm2.mins, &tm2.secs);
  //*** Add times ***
  TimeType answer = tm1 + tm2;
  //*** Display times ***
  cout << tm1.hrs << ':' << tm1.mins << ':'
  ↳<< tm1.secs << " + " << tm2.hrs << ':'
  ↳<< tm2.mins << ':' << tm2.secs << " = "
  ↳<< answer.hrs << ':' << answer.mins << ':'
  ↳<< answer.secs << endl;
  //*** Add another 57 seconds ***
  answer = answer + 57;
  cout << "+57 seconds = " << answer.hrs << ':'
  ↳<< answer.mins << ':' << answer.secs << endl;
}

//*** Add two times ***
TimeType AddTimes(const TimeType &t1,
↳const TimeType &t2)
{
  TimeType result {-1,-1,-1};
  if (t1.hrs < 0 || t1.mins < 0 || t1.mins > 59 ||
  ↳t1.secs < 0 || t1.secs > 59||t2.hrs < 0 ||
  ↳t2.mins < 0 || t2.mins > 59 || t2.secs < 0 ||
  ↳t2.secs > 59)
  return result;
  int total_secs = (t1.hrs + t2.hrs) * 60 * 60 +
  ↳(t1.mins + t2.mins) * 60 + (t1.secs + t2.secs);
  result.hrs = total_secs / 3600;
  result.mins = (total_secs - result.hrs * 3600)/60;
  result.secs = total_secs % 60;
  return result;
}

//*** Add two times ***
TimeType operator+(const TimeType& t1,
↳const TimeType& t2)
{
  //*** Calculate total time ***
  int total_secs = (t1.hrs + t2.hrs) * 60 * 60 +
  ↳(t1.mins + t2.mins) * 60 + (t1.secs + t2.secs);
```

```
  TimeType result;
  result.hrs = total_secs / 3600;
  result.mins = (total_secs - result.hrs * 3600)/60;
  result.secs = total_secs % 60;
  return result;
}

//*** Add seconds to a time ***
TimeType operator+(const TimeType& t, const int s)
{
  //*** Calculate total time ***
  int total_secs = t.hrs * 60 * 60 + t.mins * 60 +
  ↳t.secs + s;
  TimeType result;
  result.hrs = total_secs / 3600;
  result.mins = (total_secs - result.hrs * 3600)/60;
  result.secs = total_secs % 60;
  return result;
}
```

Activity 12.11

Modifications to *TimeFunction*:

Prototypes:

```
TimeType operator+(const int, const TimeType&);
```

Overloaded *operator+()*:

```
TimeType operator+( const int s, const TimeType& t)
{
  //*** Calculate total time ***
  int total_secs = t.hrs * 60 * 60 + t.mins * 60 +
  ↳t.secs + s;
  TimeType result;
  result.hrs = total_secs / 3600;
  result.mins = (total_secs - result.hrs * 3600)/60;
  result.secs = total_secs % 60;
  return result;
}
```

In `main()`, we need only change the line

```
answer = answer + 57;
```

to

```
answer = 57 + answer;
```

Activity 12.12

Modifications to *TimeFunction*:

Prototypes:

```
bool operator<(const TimeType&, const TimeType&);
bool operator==(const TimeType&, const TimeType&);
```

Overloaded *operator<()*:

```
bool operator<(const TimeType& t1,
↳const TimeType& t2)
{
  int total_secs1 = t1.hrs * 3600 + t1.mins * 60 +
  ↳t1.secs;
  int total_secs2 = t2.hrs * 3600 + t2.mins * 60 +
  ↳t2.secs;
  return (total_secs1 < total_secs2);
}
```

Overloaded *operator==()*:

```
bool operator==(const TimeType& t1,
↳const TimeType& t2)
{
  int total_secs1 = t1.hrs * 3600 + t1.mins * 60 +
  ↳t1.secs;
  int total_secs2 = t2.hrs * 3600 + t2.mins * 60 +
  ↳t2.secs;
  return (total_secs1 == total_secs2);
}
```

`main()`:

```
void main()
{
  TimeType tm1, tm2;

  //*** Input times ***
  cout << "Enter first time (hh:mm:ss) : ";
  scanf("%d:%d:%d", &tm1.hrs, &tm1.mins, &tm1.secs);
  cout << "Enter second time (hh:mm:ss) : ";
  scanf("%d:%d:%d", &tm2.hrs, &tm2.mins, &tm2.secs);
  if (tm1 == tm2)
    cout << "times are the same\n";
  else if (tm1 < tm2)
    cout << tm1.hrs << ':' << tm1.mins << ':'
    ↳<< tm1.secs << " is less than " << tm2.hrs
    ↳<< ':' << tm2.mins << ':' << tm2.secs << endl;
  else
    cout << tm1.hrs << ':' << tm1.mins << ':'
    ↳<< tm1.secs << " is greater than " << tm2.hrs
    ↳<< ':' << tm2.mins << ':' << tm2.secs << endl;
}
```

Activity 12.13

Modifications to *TimeFunction*:

Prototypes:

```
void operator++(TimeType&);
void operator++(TimeType&, int);
```

Overloaded *operator++() pre*:

```
//*** Adds one second to time (pre) ***
void operator++(TimeType& t)
{
  int total_secs = t.hrs * 3600 + t.mins * 60 +
  ↳t.secs + 1;
  t.hrs = total_secs / 3600;
  t.mins = (total_secs - t.hrs * 3600) / 60;
  t.secs = total_secs % 60;
}
```

Overloaded *operator++() post*:

```
//*** Adds one second to time (post)***
void operator++(TimeType& t, int)
{
  int total_secs = t.hrs * 3600 + t.mins * 60 +
  ↳t.secs + 1;
  t.hrs = total_secs / 3600;
  t.mins = (total_secs - t.hrs * 3600) / 60;
  t.secs = total_secs % 60;
}
```

`main()`:

```
void main()
{
  TimeType tm1, tm2;

  //*** Input times ***
  cout << "Enter first time (hh:mm:ss) : ";
  scanf("%d:%d:%d", &tm1.hrs, &tm1.mins, &tm1.secs);
  tm1++;
  ++tm1;
  cout << tm1.hrs << ':' << tm1.mins << ':'
  ↳<< tm1.secs << endl;
}
```

Activity 12.14

The line

```
tm2 = tm1++;
```

creates a compilation error because our ++ operator does not return a value that can be assigned to *tm2*.

Activity 12.15

Modifications to *TimeFunction*:

Prototypes:

```
TimeType operator++(TimeType&);
TimeType operator++(TimeType&, int);
```

Overloaded *operator++() pre*:

```
//*** Adds one second to time (pre) ***
TimeType operator++(TimeType& t)
```

```
{
  int total_secs = t.hrs * 3600 + t.mins * 60 +
  ↳t.secs + 1;
  t.hrs = total_secs / 3600;
  t.mins = (total_secs - t.hrs * 3600) / 60;
  t.secs = total_secs % 60;
  return t;
}
```

Overloaded *operator++() post*:

```
//*** Adds one second to time (post)***
TimeType operator++(TimeType& t, int)
{
  TimeType temp{ t };
  int total_secs = t.hrs * 3600 + t.mins * 60 +
  ↳t.secs + 1;
  t.hrs = total_secs / 3600;
  t.mins = (total_secs - t.hrs * 3600) / 60;
  t.secs = total_secs % 60;
  return temp;
}
```

`main()`:

```
void main()
{
  TimeType tm1, tm2, tm3;

  //*** Input times ***
  cout << "Enter first time (hh:mm:ss) : ";
  scanf("%d:%d:%d", &tm1.hrs, &tm1.mins, &tm1.secs);
  tm2 = tm1++;
  tm3 = ++tm1;
  cout << tm1.hrs << ':' << tm1.mins << ':'
  ↳<< tm1.secs << endl;
  cout << tm2.hrs << ':' << tm2.mins << ':'
  ↳<< tm2.secs << endl;
  cout << tm3.hrs << ':' << tm3.mins << ':'
  ↳<< tm3.secs << endl;
}
```

Activity 12.16

Modified code for *Enumeration*:

```
#include <iostream>
using namespace std;

// *** Set up enumerated type ***
enum struct DaysType
{
  Sunday, Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday
};

//*** Function prototypes ***
DaysType operator++(DaysType&);
DaysType operator++(DaysType&, int);
DaysType operator--(DaysType&);
DaysType operator--(DaysType&, int);

int main()
{
  // *** Set up matching string array ***
  char DaysoftheWeek[7][10]{ "Sunday","Monday",
  ↳"Tuesday", "Wednesday","Thursday","Friday",
  ↳"Saturday" };
  DaysType day = DaysType::Saturday;
  DaysType day2, day3, day4, day5;

  // *** Increment day ***
  day2 = ++day;
  day3 = day++;
  //*** Display corresponding array element for each
  ↳variable ***
  cout << "day = " << DaysoftheWeek[
  ↳static_cast<int>(day)] << endl;
  cout << "day2 = " << DaysoftheWeek[
  ↳static_cast<int>(day2)] << endl;
  cout << "day3 = " << DaysoftheWeek[
  ↳static_cast<int>(day3)] << endl;
  day4 = --day;
  day5 = day--;
  //*** Display corresponding array element for each
  ↳variable ***
  cout << "day = " << DaysoftheWeek[
  ↳static_cast<int>(day)] << endl;
  cout << "day4 = " << DaysoftheWeek[
  ↳static_cast<int>(day4)] << endl;
  cout << "day5 = " << DaysoftheWeek[
  ↳static_cast<int>(day5)] << endl;
}

DaysType operator++(DaysType& dy)
{
  dy = static_cast<DaysType>((static_
cast<int>(dy)+1) % 7);
  return dy;
}

DaysType operator++(DaysType& dy, int)
{
  DaysType result = dy;
  dy = static_cast<DaysType>((static_
cast<int>(dy)+1) % 7);
  return result;
}
DaysType operator--(DaysType& dy)
{
  dy = static_cast<DaysType>((static_
cast<int>(dy)+6) % 7);
  return dy;
}

DaysType operator--(DaysType& dy, int)
{
  DaysType result = dy;
    dy = static_cast<DaysType>((static_
cast<int>(dy)+6) % 7);
  return result;
}
```

Activity 12.17

Code for *RecursiveFact:*

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
long Factorial(long);

int main()
{
  int num;
  cout << "Enter number : ";
  cin >> num;
  cout << num << "! = " << Factorial(num) << endl;
}


//*** Returns v! ***
long Factorial(long v)
{
  if (v == 0)
    return 1;
  else
    return v * Factorial(v - 1);
}
```

Activity 12.18

The type name is set up using the following statement:

```
using  ifptrTypeii  = int(*)(int,int);
```

Activity 12.19

Modified code for *FunctionPointers*:

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
void FA();
void FB();
void FC();
int Smaller(int, int);

int main()
{
  using  vfptrTypev = void(*)();
  using ifptrTypeii = int(*)(int, int);
  vfptrTypev  fptr;
  ifptrTypeii fptr2;
  fptr = FA;
  fptr();
```

```
  fptr = FB;
  fptr();
  fptr = FC;
  fptr();
  fptr2 = Smaller;
  cout << "Smaller value = " << fptr2(43, 15)
  ↳<< endl;
}

void FA()
{
  cout << "FA() executed\n";
}

void FB()
{
  cout << "FB() executed\n";
}

void FC()
{
  cout << "FC() executed\n";
}

int Smaller(int n1, int n2)
{
  return (n1 < n2) ? n1 : n2;
}
```

Activity 12.20

a) The sum of the two values supplied.
b) The difference between the two values.

Activity 12.21

When run the value 15 is displayed (12+3).

To have the *Operate()* function execute *Sub()* rather than *Add()*, we need to change the line

```
cout << Operate(Add,12,3) << endl;
```

to

```
cout << Operate(Sub,12,3) << endl;
```

This will give the result 9 (12 - 3).

Activity 12.22

No solution required.

Activity 12.23

Run 1 (q entered): The output is:

Value returned : 0

Run 2 (other key entered): The output is:

Value returned : 1

Activity 12.24

No solution required.

Activity 12.25

The program outputs the following:

```
5 strings were passed to the program
The strings were :
MainParameters
a
b
c
d
```

Activity 12.26

No solution required.

Activity 12.27

Changes to *DateLibrary*:

Prototypes:

```
//*** Function prototype ***
bool IsValidDate(const DateType&);
bool IsValidDate(const int, const int, const int);
int GetDayOfWeek(const DateType&);
int GetDayOfWeek(const int, const int, const int);
bool IsLeapYear(const int);
```

Added function code:

```
//*** Returns true if y is a leap year   ***
bool IsLeapYear(const int year)
{
  bool result{ true };
  if (year < 1753)
    result = false;
  else if (!(year % 4 == 0 && year % 100 != 0) ||
(year % 400 == 0))
    result = false;
  return result;
}
```

Modified function code:

```
//*** Check date is valid ***
bool IsValidDate(const DateType& dt)
{
  bool result{ true };
  //** If invalid year, exit **
  if (dt.year <= 1752)
    result = false;;
  //** If invalid month, exit **
  if (dt.month < 1 || dt.month > 12)
    result = false;
  //** If invalid day, exit **
  int daysinmonth[]{ 0,31,28,31,30,31,30,31,31,30,
  ↳31,30,31 };
  //*** If leap year and 1 to days in Feb ***
  if (IsLeapYear(dt.year))
    daysinmonth[2]++;
  if (dt.day < 1 || dt.day > daysinmonth[dt.month])
    result = false;
  //*** Return date's validity ***
  return result;
}

//*** Check date is valid ***
bool IsValidDate(const int day, const int month,
↳const int year)
{
  bool result{ true };
  //** If invalid year, exit **
  if (year <= 1752)
    result = false;;
  //** If invalid month, exit **
  if (month < 1 || month > 12)
    result = false;
  //** If invalid day, exit **
  int daysinmonth[]{ 0,31,28,31,30,31,30,31,31,30,
  ↳31,30,31 };
  //*** If leap year and 1 to days in Feb ***
  if (IsLeapYear(year))
    daysinmonth[2]++;
  if (day < 1 || day > daysinmonth[month])
    result = false;
  //*** Return date's validity ***
  return result;
}
```

Activity 12.28

Changes to *DateLibrary*:

Prototypes:

```
//*** Function prototype ***
bool IsValidDate(const DateType&);
bool IsValidDate(const int, const int, const int);
int GetDayOfWeek(const DateType&);
int GetDayOfWeek(const int, const int, const int);
```

```
bool IsLeapYear(const int);
DateType SetDate(const int, const int, const int);
```

Added function code:

```
//*** Sets the date in date to d/m/y ***
DateType SetDate(const int d, const int m,
↳const int y)
{
  DateType result{ 1,1,2001 };
  // *** Days in each month of year ***
  int daysinmonth[] = { 0, 31,28,31, 30,31,30,
  ↳31,31,30,31,30,31 };

  //*** If month invalid, exit ***
  if (m < 1 || m > 12)
    return result;
  //*** Add 1 to days in February if leap year ***
  if (IsLeapYear(y))
    daysinmonth[2]++;
  //*** If days in month invalid, exit ***
  if (d <1 || d > daysinmonth[m])
    return result;
  //*** If year less than 1, exit ***
  if (y < 1)
    return result;
  //*** Assign date ***
  result.day = d;
  result.month = m;
  result.year = y;
  return result;
}
```

Test code in `main()` (run 1 - valid):

```
void main()
{
  DateType dt1 = SetDate(29, 2, 2020);
  cout << dt1.day << '/' << dt1.month << '/'
  ↳<< dt1.year << endl;
}
```

Test code in `main()` (run 2 - invalid date):

```
void main()
{
  DateType dt1 = SetDate(29, 2, 2021);
  cout << dt1.day << '/' << dt1.month << '/' << dt1.
year << endl;
}
```

Activity 12.29

Changes to *DateLibrary*:

Prototypes:

```
//*** Function prototype ***
bool IsValidDate(const DateType&);
bool IsValidDate(const int, const int, const int);
int GetDayOfWeek(const DateType&);
int GetDayOfWeek(const int, const int, const int);
bool IsLeapYear(const int);
DateType SetDate(const int, const int, const int);
long long int DateToJDN(const DateType&);
```

Added function code:

```
// *** Converts a date to a JDN ***
long long int DateToJDN(const DateType& d)
{

  int a = (14 - d.month) / 12;
  int m = d.month + 12 * a - 3;
  int y = d.year + 4800 - a;
  long long int result = d.day + (153 * m + 2) / 5 +
  ↳365 * y + y / 4 - y / 100 + y / 400 - 32045LL;
  return result;
}
```

Test code in `main()` (run 1 - 1 day):

```
void main()
{
  DateType dt1{ 1,1,2019 };
  DateType dt2{ 2,1,2019 };
  cout << DateToJDN(dt1) << "  " << DateToJDN(dt2)
  ↳<< endl;
}
```

Test code in `main()` (run 2 - 365 days):

```
void main()
{
  DateType dt1{ 1,1,2019 };
  DateType dt2{ 1,1,2020 };
  cout << DateToJDN(dt1) << "  " << DateToJDN(dt2)
  ↳<< endl;
}
```

Test code in `main()` (run 3 - 366 days):

```
void main()
{
  DateType dt1{ 1,1,2020 };
  DateType dt2{ 1,1,2021 };
  cout << DateToJDN(dt1) << "  " << DateToJDN(dt2)
  ↳<< endl;
}
```

Activity 12.30

Changes to *DateLibrary*:

Prototypes:

```
//*** Function prototype ***
bool IsValidDate(const DateType&);
bool IsValidDate(const int, const int, const int);
int GetDayOfWeek(const DateType&);
int GetDayOfWeek(const int, const int, const int);
bool IsLeapYear(const int);
DateType SetDate(const int, const int, const int);
long GetDaysBetween(const DateType&,
↳const DateType&);
long long int DateToJDN(const DateType&);
```

Added function code:

```
// *** Returns number of days between two dates ***
long GetDaysBetween(const DateType& dt1, const
DateType& dt2)
{
  long days1 = DateToJDN(dt1);
  long days2 = DateToJDN(dt2);
  long result = days1 - days2;
  return result;
}
```

Test code in `main()` (run 1 - 1 day):

```
void main()
{
  DateType dt1 = SetDate(1, 3, 2020);
  DateType dt2 = SetDate(27, 2, 2020);
  cout << GetDaysBetween(dt1, dt2) << endl;
}
```

Activity 12.31

Changes to *DateLibrary*:

Prototypes:

```
//*** Function prototype ***
bool IsValidDate(const DateType&);
bool IsValidDate(const int,const int, const int);
int GetDayOfWeek(const DateType&);
int GetDayOfWeek(const int, const int, const int);
bool IsLeapYear(const int);
DateType SetDate(const int, const int, const int);
long GetDaysBetween(const DateType&,
↳const DateType&);
DateType AddDays(const DateType& , const int);
long long int DateToJDN(const DateType&);
DateType JDNToDate(const long);
```

Added function code:

```
// *** Returns a date which is a specified number ***
// *** of days after a given date                 ***
DateType AddDays(const DateType& d, const int days)
```

```
{
  int totaldays = DateToJDN(d) + days;
  DateType result = JDNToDate(totaldays);
  return result;
}


// *** Converts a JDN to a date ***
DateType JDNToDate(const long jdn)
{
  DateType date;

  int f = jdn + 1401+(((4 * jdn + 274277) / 146097)
  ↳* 3) / 4 - 38;
  int e = 4 * f + 3;
  int g = (e % 1461) / 4;
  int h = 5 * g + 2;
  date.day = (h % 153) / 5 + 1;
  date.month = (h / 153 + 2) % 12 + 1;
  date.year = e / 1461 - 4716 + (12 + 2 -
  ↳date.month) / 12;
  return date;
}
```

Test code in `main()` (3 days added):

```
void main()
{
  DateType dt1 = SetDate(27, 2, 2020);
  DateType dt2 = AddDays(dt1, 3);
  cout << dt2.day << '/' << dt2.month << '/' << dt2.
year << endl;
}
```

Activity 12.32

Changes to *DateLibrary*:

Prototypes:

```
//*** Function prototype ***
bool IsValidDate(const DateType&);
bool IsValidDate(const int,const int, const int);
int GetDayOfWeek(const DateType&);
int GetDayOfWeek(const int, const int, const int);
bool IsLeapYear(const int);
DateType SetDate(const int, const int, const int);
long GetDaysBetween(const DateType&,
↳const DateType&);
DateType AddDays(const DateType& , const int);
DateType operator+(const DateType&, const int);
DateType operator+(const int, const DateType&);
long long int DateToJDN(const DateType&);
DateType JDNToDate(const long);
```

Added function code:

```
// *** Returns a date equal to (dt+days) ***
DateType operator+(const DateType& dt,
↳const int days)
{
  int totaldays = DateToJDN(dt) + days;
  DateType result = JDNToDate(totaldays);
  return result;
}

// *** Returns a date equal to (dt+days) ***
DateType operator+( const int days,
↳const DateType& dt )
{
  int totaldays = DateToJDN(dt) + days;
  DateType result = JDNToDate(totaldays);
  return result;
}
```

Test code in `main()` (run 1 - 3 days added dt+3):

```
void main()
{
  DateType dt1 = SetDate(27, 2, 2020);
  DateType dt2 = dt1 + 3;
  cout << dt2.day << '/' << dt2.month << '/'
  ↳<< dt2.year << endl;
}
```

Test code in `main()` (run 2 - 3 days added 3 + dt):

```
void main()
{
  DateType dt1 = SetDate(27, 2, 2020);
  DateType dt2 = 3 + dt1;
  cout << dt2.day << '/' << dt2.month << '/'
  ↳<< dt2.year << endl;
}
```

Activity 12.33

Changes to *DateLibrary*:

Prototypes:

```
//*** Function prototype ***
bool IsValidDate(const DateType&);
bool IsValidDate(const int,const int, const int);
int GetDayOfWeek(const DateType&);
int GetDayOfWeek(const int, const int, const int);
bool IsLeapYear(const int);
DateType SetDate(const int, const int, const int);
long GetDaysBetween(const DateType&, const
DateType&);
DateType AddDays(const DateType& , const int);
DateType operator+(const DateType&, const int);
DateType operator+(const int, const DateType&);
long operator-(const DateType&, const DateType&);
DateType operator-(const DateType&, const int);
long long int DateToJDN(const DateType&);
DateType JDNToDate(const long);
```

Added function code:

```
// *** Returns number of days between two days ***
long operator-(const DateType& dt1,
↳const DateType& dt2)
{
  return (DateToJDN(dt1) - DateToJDN(dt2));
}



// *** Returns a date equal to (dt1 - days) ***
DateType operator-(const DateType& dt,
↳const int days)
{
  int totaldays = DateToJDN(dt) - days;
  DateType result = JDNToDate(totaldays);
  return result;
}
```

Test code in `main()`:

```
void main()
{
  DateType dt1 = SetDate(27, 2, 2020);
  DateType dt2 = SetDate(10, 2, 2019);
  DateType dt3 = dt2 - 42;
  int diff = dt1 - dt2;
  cout << dt3.day << '/' << dt3.month << '/'
  ↳<< dt3.year << endl;
  cout << "Difference : " << diff << endl;
}
```

Activity 12.34

Changes to *DateLibrary*:

Prototypes:

```
//*** Function prototype ***
bool IsValidDate(const DateType&);
bool IsValidDate(const int,const int, const int);
int GetDayOfWeek(const DateType&);
int GetDayOfWeek(const int, const int, const int);
bool IsLeapYear(const int);
DateType SetDate(const int, const int, const int);
long GetDaysBetween(const DateType&, const
DateType&);
DateType AddDays(const DateType& , const int);
DateType operator+(const DateType&, const int);
DateType operator+(const int, const DateType&);
long operator-(const DateType&, const DateType&);
DateType operator-(const DateType&, const int);
bool operator<(const DateType&, const DateType&);
```

```
bool operator==(const DateType&, const DateType&);
long long int DateToJDN(const DateType&);
DateType JDNToDate(const long);
```

Added function code:

```
//*** Returns true if dt1 < dt2 ***
bool operator<(const DateType& dt1,
↳const DateType& dt2)
{
  return (DateToJDN(dt1) < DateToJDN(dt2));
}

//*** Returns true if dt1 == dt2 ***
bool operator==(const DateType& dt1,
↳const DateType& dt2)
{
  return (DateToJDN(dt1) == DateToJDN(dt2));
}
```

Test code in `main()`

```
void main()
{
  DateType dt1 = SetDate(27, 2, 2020);
  DateType dt2 = SetDate(1, 3, 2020);
  if (dt2 == dt1 + 3)
    cout << "Dates equal\n";
  if (dt1 < dt2)
    cout << "Date less than\n";
}
```

Activity 12.35

Code for *Bigger()* function template:

Prototype:

```
template <typename T> T Bigger(const T&, const T&);
```

Definition:

```
template <typename T>
T Bigger(const T& v1, const T& v2)
{
  if (v1 > v2)
    return v1;
  else
    return v2;
}
```

Activity 12.36

Initial code for *Templates*:

```
#include <iostream>
using namespace std;

//***Function templates ***
template <typename T> void Swap(T&, T&);
template <typename T> T Bigger(const T&, const T&);


int main()
{
}


//*** Swaps the contents of v1 and v2 ***
template <typename T>
void Swap(T& v1, T& v2)
{
  T temp{ v1 };
  v1 = v2;
  v2 = temp;
}


//*** Returns a copy of the larger of n1 and n2 ***
template <typename T>
T Bigger(const T& n1, const T& n2)
{
  if (v1 > v2)
    return v1;
  else
    return v2;
}
```

Activity 12.37

The program executes as expected, automatically creating the appropriate versions of each templated function.

Activity 12.38

Code for *TestVariadc*:

```
#include <iostream>
#include <cstdarg>

using namespace std;

//*** Function prototype ***
int Sum(int,...);

int main()
{
  cout << Sum(4, 2, 5, 12, 3) << endl;
  cout << Sum(7, 1,1,2,3,5,8,13) << endl;
}

int Sum(int count,...)
{
  va_list args;
  va_start(args, count);
  int result = 0;
  for (int c = 1; c <= count; c++)
    result += va_arg(args, int);
  va_end(args);
  return result;
}
```

If the count is too small, the extra parameters are ignored.

If the count is too large, spurious values are returned (other compilers may give different results).

Activity 12.39

Modified code for *TestVariadc*:

```
#include <iostream>
#include <cstdarg>

using namespace std;

//*** Function prototypes ***
int Sum(int,...);
void ListNameAndAge(int, ...);

int main()
{
  ListNameAndAge(3, "Maggie Tulliver", 18,
  ↳"Lucy Deane", 38,"Philip Wakem", 23);
}

int Sum(int count,...)
{
  va_list args;
  va_start(args, count);
  int result = 0;
  for (int c = 1; c <= count; c++)
    result += va_arg(args, int);
  va_end(args);
  return result;
}

void ListNameAndAge(int count, ...)
{
  va_list args;
  va_start(args, count);
  char* name;
  int age;
  for (int c = 1; c <= count; c++)
  {
    name = va_arg(args, char*);
    age = va_arg(args, int);
    cout << name << "   " << age << endl;
  }
  va_end(args);
}
```

Activity 12.40

Modified code for *TestVariadc*:

```
#include <iostream>
#include <cstdarg>

using namespace std;

//*** Function prototypes ***
int Sum(int,...);
void ListNameAndAge(int, ...);
void DisplayAll(const char*, ...);


int main()
{
  DisplayAll("cidsi",'X',23,12.78,"Hello",42);
}

int Sum(int count,...)
{
  va_list args;
  va_start(args, count);
  int result = 0;
  for (int c = 1; c <= count; c++)
    result += va_arg(args, int);
  va_end(args);
  return result;
}

void ListNameAndAge(int count, ...)
{
  va_list args;
  va_start(args, count);
  char* name;
  int age;
  for (int c = 1; c <= count; c++)
  {
    name = va_arg(args, char*);
    age = va_arg(args, int);
    cout << name << "   " << age << endl;
  }
  va_end(args);
}

void DisplayAll (const char * format, ...)
{
  va_list args;
  va_start(args, format);
  int idx = 0;
  while (format[idx] != '\0')
  {
    switch (format[idx])
    {
      case 'c':
        cout << va_arg(args, char) << endl;
        break;
      case 's':
        cout << va_arg(args, char*) << endl;
        break;
      case 'i':
        cout << va_arg(args, int) << endl;
        break;
      case 'd':
        cout << va_arg(args, double) << endl;
        break;
    };
    idx++;
  }
  va_end(args);
}
```

Activity 12.41

No solution required.

```
#include <iostream>
#include <cstdarg>

using namespace std;

//*** Function prototypes ***
int Sum(int,...);
void ListNameAndAge(int, ...);
void DisplayAll(const char*, ...);
void DisplayAll2();
template <typename T, typename... Types>
void DisplayAll2(T, Types...);
int main()
{
  DisplayAll2('X', 23, 12.78, "Hello", 42);
}

int Sum(int count,...)
{
  va_list args;
  va_start(args, count);
  int result = 0;
  for (int c = 1; c <= count; c++)
    result += va_arg(args, int);
  va_end(args);
  return result;
}

void ListNameAndAge(int count, ...)
{
  va_list args;
  va_start(args, count);
  char* name;
  int age;
  for (int c = 1; c <= count; c++)
  {
    name = va_arg(args, char*);
    age = va_arg(args, int);
    cout << name << "   " << age << endl;
  }
  va_end(args);
}

void DisplayAll (const char * format, ...)
{
  va_list args;
  va_start(args, format);
  int idx = 0;
  while (format[idx] != '\0')
  {
    switch (format[idx])
    {
      case 'c':
        cout << va_arg(args, char) << endl;
        break;
      case 's':
        cout << va_arg(args, char*) << endl;
        break;
      case 'i':
        cout << va_arg(args, int) << endl;
        break;
      case 'd':
        cout << va_arg(args, double) << endl;
        break;
    };
    idx++;
  }
  va_end(args);
}

void DisplayAll2() {}

template <typename T, typename... Types>
void DisplayAll2(T v1, Types... v2)
{
  cout << v1 << endl;
  DisplayAll2(v2...);
}
```

Activity 12.42

Code for *TestFoldExpression*:

```
#include <iostream>
using namespace std;

template <typename... Types>
auto Sum(Types ...vs)
{
  return(... + vs);
}

int main()
{
  cout << Sum(1, 3, 7) << endl;
}
```

The output is

```
11
```

Modified code for *TestFoldExpression*:

```
#include <iostream>
using namespace std;

template <typename... Types>
auto Sum(Types ...vs)
{
  return(10 + ... + vs);
}

int main()
{
  cout << Sum(1, 3, 7) << endl;
}
```

The output is

```
21
```

Activity 12.43

Modified code for *TestFoldExpression*:

```
#include <iostream>
#include <vector>
using namespace std;

template <typename... Types>
auto Sum(Types ...vs)
{
  return(10 + ... + vs);
}

template <typename... Types>
void Display(Types ...vs)
{
  (cout << ... << vs) << endl;
}

int main()
{
  cout << Sum(1, 3, 7) << endl;
  Display(1, 2, 3, 4, 5, 6);
}
```

The output from *Display()* is

```
123456
```

Activity 12.44

Modified code for *TestFoldExpression (version b)*:

```
#include <iostream>
#include <vector>
using namespace std;

template <typename... Types>
auto Sum(Types ...vs)
{
cout << "There are " << sizeof...(vs) <<
↳" parameters\n";
  return(10 + ... + vs);
}

template <typename... Types>
void Display(Types ...vs)
{
  (cout << ... << vs) << endl;
}

int main()
{
  cout << Sum(2, -9, 4, 2, 8) << endl;
  Display(1, 2, 3, 4, 5, 6);
}
```

The output from the call to *Sum()* is

```
There are 5 parameters
17
```

13

Standard Function Library

In this Chapter:

- ❑ Maths Functions
- ❑ Date and Time Functions
- ❑ Random Number Functions
- ❑ System Functions

Standard Functions

Introduction

Though we are free to create our own functions, there's no point in reinventing the wheel, so many commonly used functions come ready-made and available within C++'s extensive library. Many of these have been inherited from C, others have been introduced as part of C++. While by no means all available functions are listed, many of the most-used ones inherited from C are briefly described in this chapter. For other functions, check out your compiler's manual.

To use a function, we may have to add a `#include` statement to access the function's prototype. The filename required by the `#include` statement is given in the heading section for each group of functions.

Maths Functions <cmath>

There are several maths functions available within C++. Most are listed here without explanation since they represent standard mathematical functions which should be familiar to us from our school days. Others, which we may not have encountered before, are given a fuller explanation and examples. All of the functions listed in this section are overloaded to accept and return `float`, `double`, and `long double` values.

cos()

Function Prototype	:	double cos(double ang)
Pre-condition	:	None
Description	:	Returns the cosine of angle *ang (*given in radians).
Example	:	`double ans = cos(0.5);`

sin()

Function Prototype	:	double sin(double ang)
Pre-condition	:	None
Description	:	Returns the sine of angle *ang* (given in radians).
Example	:	`double ans = sin(-0.2);`

tan()

Function Prototype	:	double tan(double ang)
Pre-condition	:	None
Description	:	Returns the tangent of angle *ang* (given in radians).
Example	:	`double ans = tan(0.02);`

acos()

Function Prototype	:	double acos(double cs)
Pre-condition	:	None
Description	:	Returns the angle (in radians) for which *cs* is the cosine.
Example	:	`double ans = acos(0.5);`

asin()

Function Prototype	:	double asin(double sn)
Pre-condition	:	None
Description	:	Returns the angle (in radians) for which *sn* is the sine.
Example	:	`double ans = asin(0.3);`

atan()

Function Prototype	:	double atan(double tn)
Pre-condition	:	None
Description	:	Returns the angle (in radians) for which *tn* is the tangent. Note that the results will lie in the range $-\pi/2$ to $+\pi/2$.
Example	:	`double ans = atan(17.4);`

atan2()

Function Prototype	:	double atan2(double y, double x)
Pre-condition	:	$y \neq 0$ OR $x \neq 0$
Description	:	Returns the angle (in radians) for which *tn* is the tangent. (*x, y*) represents the end point of a line drawn from the origin. Note that the results will lie in the range $-\pi$ to $+\pi$.
Example	:	`double ans = atan2(0.0,-1.0);`

cosh()

Function Prototype	:	double cosh(double ang)
Pre-condition	:	None
Description	:	Returns the hyperbolic cosine of angle *ang* (given in radians).
Example	:	`double ans = cosh(1.2);`

sinh()

Function Prototype	:	double sinh(double ang)
Pre-condition	:	None
Description	:	Returns the hyperbolic sine of angle *ang* (given in radians).
Example	:	`double ans = sinh(0.1);`

tanh()

Function Prototype	:	double tanh(double ang)
Pre-condition	:	None
Description	:	Returns the hyperbolic tangent of angle *ang* (given in radians).
Example	:	`double ans = tanh(0.35);`

acosh()

Function Prototype	:	double acosh(double hcs)
Pre-condition	:	None
Description	:	Returns the angle (in radians) for which *hcs* is the hyperbolic cosine.
Example	:	`double ans = acosh(0.6);`

asinh()

Function Prototype	:	double asinh(double hsn)
Pre-condition	:	None
Description	:	Returns the angle (in radians) for which *hsn* is the hyperbolic sine.
Example	:	`double ans = asinh(0.9);`

atanh()

Function Prototype	:	double atanh(double htn)
Pre-condition	:	None
Description	:	Returns the angle (in radians) for which *htn* is the hyperbolic tangent.
Example	:	`double ans = atanh(0.8);`

exp()

Function Prototype	:	double exp(double x)
Pre-condition	:	None
Description	:	Returns the value e^x.
Example	:	`double ans = exp(2.0);`

exp2()

Function Prototype	:	double exp2(double x)
Pre-condition	:	None
Description	:	Returns the value 2^x.
Example	:	`double ans = exp2(8.0);`

log()

Function Prototype	:	double log(double x)
Pre-condition	:	x >= 0
Description	:	Returns the value $\log_e x$.
Example	:	`double ans = log(1.0);`

log10()

Function Prototype	:	double log10(double x)
Pre-condition	:	x >= 0
Description	:	Returns the value $\log_{10} x$.
Example	:	`double ans = log10(0.1);`

log2()

Function Prototype	:	double log2(double x)
Pre-condition	:	x > 0
Description	:	Returns the value $\log_2 x$.
Example	:	`double ans = log2(0.5);`

frexp()

Function Prototype	:	double frexp(double x, int * ex)
Pre-condition	:	None
Description	:	Returns the value of the significand of *x* (as held in the binary format) and sets **ex* to the exponent of *x*.

To test the `frexp()` function, let's start by reminding ourselves how a number such as 7.5 would be shown in binary:

$$7.5_{10} = 111.1_2$$

The normalised form of the binary number would be

$$.1111 \times 2^3$$

Now, since the value .1111 represents 0.5 + 0.25 + 0.125 + 0.0625, the binary significand of our number is 0.9375 and the exponent is 3.

A program to test the function is shown in FIG-13.1.

FIG-13.1

Using *frexp()*

```
#include <iostream>
#include <cmath>
using namespace std;


int main()
{
   double x{7.5};    //Number to be split
   int ex;           //Holds exponent
   double signif;    //Holds significand

   //*** Extract significand and exponent ***
   signif = frexp(7.5,&ex);

   //*** Display significand and exponent ***
   cout << "Significand: " << signif << "  exponent: " << ex
   ↳<< endl;
}
```

Activity 13.1

Start a new project *Scientific* which implements the code given in FIG-13.1 and test the program to check that the expected results are obtained.

pow()

Function Protoype	:	double pow(double x, double y)
Pre-condition	:	NOT (x=0 AND y=0 OR x=0 AND y>0 OR x < 0 AND y not an integer)
Description	:	Returns x^y.
Example	:	`double ans = pow(2.0,8.0) // 2^8 returns 256`

sqrt()

Function Protoype	:	double sqrt(double x)
Pre-condition	:	x >= 0
Description	:	Returns $\sqrt{x}$
Example	:	`double result = sqrt(64.0) // returns 8`

cbrt()

Function Protoype	:	double cbrt(double x)
Pre-condition	:	None
Description	:	Returns $\sqrt[3]{x}$
Example	:	`double result = cbrt(16.0) //returns 2`

hypot()

Function Protoype	:	double exp(double x, double y)
Pre-condition	:	None
Description	:	Returns hypotenuse of triangle whose other two sides are of length *x* and *y*.
Example	:	`double hyp = hypot(3.0, 4.0) //returns 5`

ceil()

Function Protoype	:	double ceil(double x)
Pre-condition	:	None
Description	:	Returns the smallest integer not less than *x*.
Example	:	`double ans = ceil(2.1) //returns 3` `double ans2 = ceil(-2.1) // returns -2`

floor()

Function Protoype	:	double floor(double x)
Pre-condition	:	None
Description	:	Returns the largest integer not greater than *x*.
Example	:	`double ans = floor(2.1) //returns 2` `double ans2 = floor(-2.1) // returns -3`

fmod()

Function Protoype	:	double fmod(double x, double den)
Pre-condition	:	den ≠ 0
Description	:	Returns the remainder after *x* is divided by *den*.
Example	:	`double ans = fmod(7.6, 3.5) //returns 0.1`

trunc()

Function Protoype	:	double trunc(double x)
Pre-condition	:	None
Description	:	Returns the integral part of *x*.
Example	:	`double ans = trunc(-8.9) //returns -8.0`

round()

Function Protoype	:	double round(double x)
Pre-condition	:	None
Description	:	Returns *x* rounded to the nearest integer.
Example	:	`double ans = round(14.6) //returns 15` `double ans2 = round(14.4) //returns 14`

copysign()

Function Protoype	:	double copysign(double x, double y)
Pre-condition	:	None
Description	:	Returns a value with the magnitude of *x* and the sign of *y*.
Example	:	`double ans = copysign(7.3,-1.6) //returns -7.3` `double ans2 = copysign(-0.6,7.1) //returns 0.6`

fmax()

Function Protoype	:	double fmax(double x, double y)
Pre-condition	:	None
Description	:	Returns the larger of the two values *x* and *y*.
Example	:	`double ans = fmax(5.3, 2.6) //returns 5.3`

fmin()

Function Protoype	:	double fmin(double x, double y)
Pre-condition	:	None
Description	:	Returns the smaller of the two values *x* and *y*.
Example	:	`double ans = fmin(5.3, 2.6) //returns 2.6`

abs()



`abs()` is also overloaded for `int`, `long`, and `long long` parameters and return types.

Function Protoype	:	double abs(double x)
Pre-condition	:	None
Description	:	Returns the absolute value of *x*.
Example	:	`double ans = abs(4.2) //returns 4.2` `double ans2 = abs(-4.2) //returns 4.2`

Time Functions <ctime>

The functions in this section relate to time and date. Some of these make use of a few type aliases and what may be thought of as named constants.

clock_t is an alias for a numeric type. Exactly which type depends on the compiler implementation. The value in a variable of this type represents a number of units of time usually referred to as **clock ticks**.

CLOCKS_PER_SEC
is the number of clock ticks in one second.

time_t is an alias for a numeric type. The value held in a variable of this type represents the current time as the number of seconds since 0:00 1/1/1970.

tm is a record structure containing the various elements of a date/time and is defined as

```
struct tm
{
   int tm_sec;       //second
   int tm_min;       //minute
   int tm_hour;      //hour
   int tm_mday;      //day of the month
   int tm_mon;       //month (0 to 11)
   int tm_year;      //years since 1900
   int tm_wday;      //week day (0-6)
   int tm_yday;      //day of year (0-365)
   int tm_isdst;     //daylight saving time flag
};
```

clock()

Function Prototype	:	clock_t clock()
Pre-condition	:	None
Description	:	Returns number of clock ticks since the program started.

This function is useful when you want to know how long a section of code has taken to execute during code optimisation. The program in FIG-13.2 displays the number of seconds taken to display the cube roots of the numbers 1 to 1000.

FIG-13.2

Using *clock()*

```
#include <iostream>
#include <ctime>
using namespace std;

int main()
{
   double cuberoot;  //Cube root of number
   // *** Calculate and display cube roots ***
   for (int c = 1; c <= 1000; c++)
   {
      cuberoot = cbrt(c);
      cout << "Cube root of " << c << " is " << cuberoot << endl;
   }
   // *** Display the time taken ***
   cout << "Time taken : " << clock() / static_cast<double>
   ↳( CLOCKS_PER_SEC) << " secs" << endl;
}
```

Notice that the number of seconds is obtained by dividing the value returned by `clock()` by the `CLOCKS_PER_SEC`.

Activity 13.2

Start a new project, *ElapsedTime*, which implements the code given in FIG-13.2 and test the program to check that the expected results are obtained.

Modify the code to display the number of clock ticks rather than seconds.

time()

Function Prototype	:	time_t time(time_t *t)
Pre-condition	:	None
Description	:	Returns the current time and also sets *t* to this time.

ctime()

Function Prototype	:	char* ctime(const time_t *t)
Pre-condition	:	None
Description	:	Returns the current date/time as a string in the format *www mmm dd hh:mm:ss yyyy*. *t* is set to the number of seconds that have passed since midnight 1/1/1970 until the current time. where:

www	is the week day
mmm	is the month
hh:mm:ss	is the time
yyyy	is the year

The program in FIG-13.3 sets up the current time (using `time()`) then displays that time as a string (using `ctime()`).

FIG-13.3

Using *time()* and *ctime()*

```
#include <iostream>
#include <ctime>
using namespace std;


int main()
{
   time_t t1, t2;    // time/date in seconds
   char *td;

// time date as string

   //*** Get time/date in seconds ***
   t2 = time(&t1);   //Same value placed in t1 and t2

   //*** Show value returned using both options ***
   cout << "Time returned in t1 = " << t1 << endl;
   cout << "Time returned in t2 = " << t2 << endl;
```



FIG-13.3
(continued)

Using *time()* and *ctime()*

```
   //*** Convert to string ***
   td = ctime(&t1);

   //*** Display string ***
   cout << td << endl;
}
```

Activity 13.3

Start a new project, *Clock*, which implements the code given in FIG-13.3 and test the program to check that the expected results are obtained.

The parameter to `time()` *can be a null pointer (*`nullptr`*) in which case the function returns the time but makes no attempt to update the parameter.*

Modify *Clock*'s call to `time()` so that a null pointer is used as the parameter.

mktime()

Function Prototype	:	time_t mktime(tm *t)
Pre-condition	:	None
Description	:	Returns a *time_t* value representing the time in seconds from 1/1/1970 to the time/date in *t*.

The program in FIG-13.4 uses `mktime()` to display the number of seconds difference between a given time/date and 1/1/1970.

FIG-13.4

Using *mktime()*

```
#include <iostream>
#include <ctime>
using namespace std;

int main()
{
   time_t t;   // time/date in seconds
   tm dt;      // time/date record

   //*** Set time/date to 1 minute after midnight 1/1/1970 ***
   dt.tm_sec = 0;             //second
   dt.tm_min = 1;             //minute
   dt.tm_hour = 0;            //hour
   dt.tm_mday = 1;            //day of the month
   dt.tm_mon = 1 - 1;         //month (0 to 11)
   dt.tm_year = 1970 - 1900;  //years since 1900

   //*** Get seconds passed since midnight 1/1/1970 ***
   t = mktime(&dt);

   //*** Display seconds since midnight 1/1/1970
   cout << "Time in seconds : " << t << endl;
   cout << "Time/Date record : " << dt.tm_sec << ':' << dt.tm_min
   ↳<< ':' << dt.tm_hour << "   " << dt.tm_mday << '/'
   ↳<< dt.tm_mon+1 << '/' << dt.tm_year+1900 << endl;
}
```

Activity 13.4

Start a new project, *SecondsFrom*, which implements the code given in FIG-13.4 and check that the result is 60 seconds.

Modify the time/date setting in *dt* to be one year later. What time in seconds value is displayed?

difftime()

Function Prototype	:	double difftime(tm t1, tm t2)
Pre-condition	:	None
Description	:	Returns the difference in seconds between *t1* and *t2* (calculated as *t1* - *t2*).

asctime()

Function Prototype	:	char* asctime(const tm *t)
Pre-condition	:	None
Description	:	Returns a string constructed from the value stored in *t*. The string returned is in the format www mmm dd hh:mm:ss yyyy

gmtime()

Function Prototype	:	tm* gmtime(const time_t *t)
Pre-condition	:	None
Description	:	Returns a *tm* value equivalent to *t* adjusted to UTC (Coordinated Universal Time). UTC was previously known as GMT (Greenwich Mean Time).

For example, let's assume we are working on a computer in New York and the time is 1300. This is five hours ahead of UTC so the time returned by `gmtime()` will be 0800. (During daylight savings time there is a six hour difference.)

localtime()

Function Prototype	:	tm* localtime(const time_t *t)
Pre-condition	:	None
Description	:	Returns a *tm* value equivalent to *t*. *time()* should first be called to set the value of *t*.

Activity 13.5

Start a new project, *LocalClock*, and create a program which uses `localtime()` to store the current date/time in a *tm* variable and then displays the contents of each field in that variable.

Modify the program so that the value in *tm* is converted to a string using `asctime()` and display the string created.

C++ is capable of generating pseudo random numbers (as used in an earlier chapter). The numbers generated are produced using a mathematical formula and the results give the appearance of true random values.

The formula makes use of an initial starting value (known as the **seed value**) and this seed value determines the subsequent values produced.

srand()

Function Prototype	:	void srand(unsigned int x)
Pre-condition	:	None
Description	:	Sets the seed value for the random number generator to *x*. If *x* = 1 the seed value is reset to its default value.
Example	:	`srand(17);  //Sets the seed value to 17`

A specific seed value will produce a specific sequence of random values. Use the same seed value again and the same sequence will be produced.

Other than during testing, most programs want to create a different sequence of random values on each run. To achieve this the argument for `srand()` should be varied with each program execution. A common method of achieving this is to use the value returned by `time()` as in the line

```
srand(time(nullptr));
```

The compiler will cast the value returned by `time()` if we don't explicitly cast it ourselves.

rand()

Function Prototype	:	int rand()
Pre-condition	:	None
Description	:	Returns a value between 0 and RAND_MAX (this named constant is defined within `cstdlib`).
Example	:	`int ans = rand();`

Usually, we want a random value to lie within a shorter range. For example, 1 to 6 to simulate a die throw or 0 to 100 to represent a random percentage. To do this we only need to use the remainder operator (`%`). For example,

```
int percentage = rand() % 101;
```

will give a result in the range 0 to 100 while

```
int die = rand() % 6 + 1;
```

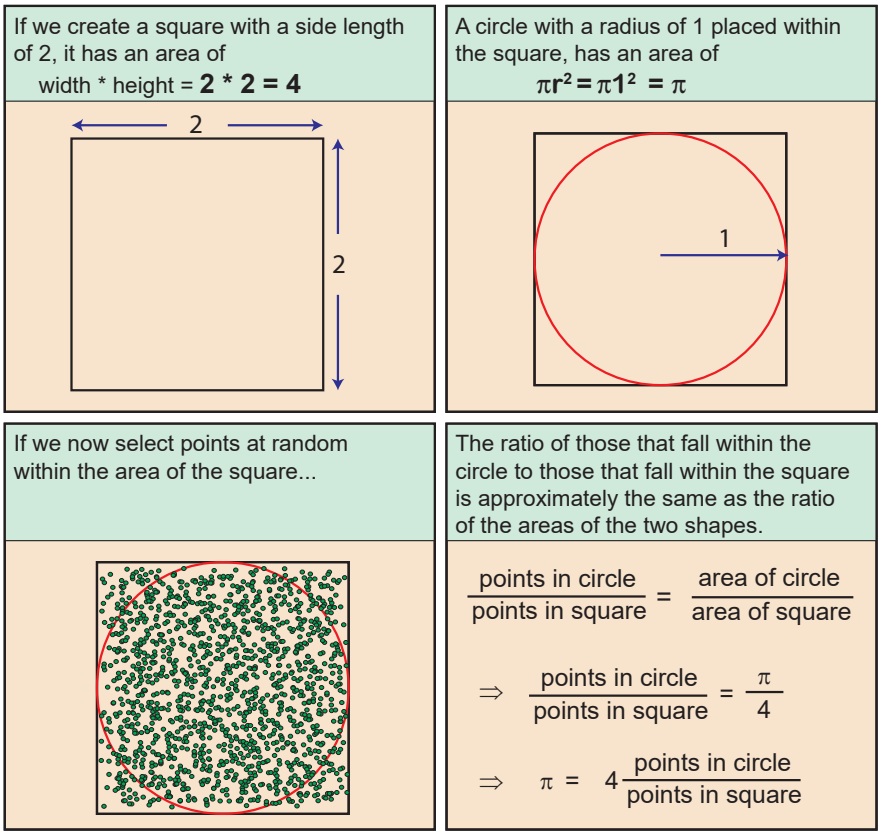
gives a value in the range 1 to 6.

We must remember that the numbers generated by `rand()` are not truly random and although the results may be useful for games or for triggering apparently random events, this function should never be used where it is critical that the values produced must be truly random.

Perhaps surprisingly, we can use random numbers to calculate an approximation of π. The technique used is known as the **Monte Carlo method** and how it is used to calculate *pi* is explained in detail in FIG-13.5.

FIG-13.5

The Monte Carlo Method for Calculating Pi



The more random points that are generated, the more accurate the approximation of *pi* should be.

If we assume that the square and circle are centred at the origin, then points within the square should have coordinates that lie between (-1.0,-1.0) and (1.0,1.0).

The program in FIG-13.6 generates the coordinates of several thousand random points within the area of the square. To discover if the generated point lies within the circle, it checks that the coordinates, *x* and *y*, are no more than 1 unit (which is the radius of the circle) from the origin. It does this with the test $(x^2 + y^2) <= 1$.

It then uses the formula given above to calculate a value for *pi*, showing how this approximation changes as more and more points are generated.

FIG-13.6

Calculating Pi

```
#include <iostream>
#include <ctime>
using namespace std;

int main()
{
   int box{ 0 };     //Points inside box
   int circle{ 0 };
   double x;
   double y;
   //*** Seed random number generator
   srand(time(0));
   //*** Calculate half the range of poosible random values ***
   double half_max = RAND_MAX / 2.0;
```

FIG-13.6
(continued)

Calculating Pi

```
   //*** Generate 100,000 points ***
   for (int c = 0; c <= 100000; c++)
   {
      //*** generate point (-1,-1) to (1,1) ***
      x = (static_cast<double>(rand())/ half_max) - 1;
      y = (static_cast<double>(rand()) / half_max) - 1;
      box++;
      if (x*x + y*y <= 1.0)
         circle++;
      //*** Display approx of pi ***
      cout <<  (4.0 * circle)/box<<"\b\b\b\b\b\b\b";
   }
}
```

Activity 13.6

Start a new project, *MonteCarloPi*, which implements the code given above and check how close it gets to *pi.*

System Functions <cstdlib>

This group of functions interact with the underlying operating system in some way - usually when the program in which they are used is being run by a batch file.

exit()

Function Prototype	:	void exit(int x)
Pre-condition	:	None
Description	:	Exits the program but first deallocates variable space (though not dynamically allocated space) and closes any file streams in use. The value of *x* is returned to the operating system. Generally, zero indicates successful termination and any other value an abnormal termination.

getenv()

Function Prototype	:	char* getenv(const char* x)
Pre-condition	:	None
Description	:	Returns the value of the operating system variable *x*.

All operating systems maintain their own set of variables. For example, most have a variable called PATH which contains a string giving the folders that should automatically be searched when looking for a file. Another variable is USERNAME which holds the name of the current user.

FIG-13.7 contains a program which uses the function `getenv()` to display the current user name.

FIG-13.7

Using *getenv()*

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Current user is " << getenv("USERNAME") << endl;
}
```

Activity 13.7

Start a new project, *User*, which implements the code given in FIG-13.7.

Test and save your project.

Modify the code so that the PATH information is displayed.

system()

Function Prototype	:	int system(const char* x)
Pre-condition	:	None
Description	:	Executes command line statement *x*. The returned value depends on the implementation. If *x* = null, the function returns 1 if the command line processor is available, otherwise zero is returned.

The program in FIG-13.8 checks for a command line processor and then executes the DIR command.

FIG-13.8

Using *system()*

```
#include <iostream>
using namespace std;

int main()
{
   if (!system(NULL))
      cout << "No command line processor found";
   else
      system("DIR");
}
```

Activity 13.8

Start a new project,*CommandLine*, which implements the code given in FIG-13.8.

Character Input Functions

getchar()

Function Prototype	:	int getchar()
Pre-condition	:	None
Description	:	Returns the `int` value of the next character from the keyboard.

Summary

- C++ comes with a number of built-in functions.
- Many of the functions available within C++ are inherited from C.
- To make use of these built-in functions, an appropriate `#include` statement may be needed within the program using the functions.
- If required, use `#include<cmath>` when calling maths functions.
- Use `cos()` to find the cosine of an angle given in radians.
- Use `sin()` to find the sine of an angle given in radians.
- Use `tan()` to find the tangent of an angle given in radians.
- Use `acos()` to convert a cosine value back to an angle (in radians).
- Use `asin()` to convert a sine value back to an angle (in radians).
- Use `atan()` to convert a tangent value back to an angle (in radians).
- Use `atan2()` to convert tangent details back to an angle between $-\pi$ and $+\pi$.
- Use `cosh()` to find the hyperbolic cosine of an angle given in radians.
- Use `sinh()` to find the hyperbolic sine of an angle given in radians.
- Use `tanh()` to find the hyperbolic tangent of an angle given in radians.
- Use `acosh()` to convert a hyperbolic cosine value back to an angle in radians.
- Use `asinh()` to convert a hyperbolic sine value back to an angle in radians.
- Use `atanh()` to convert a hyperbolic tangent value back to an angle in radians.
- Use `exp()` to find e^x.
- Use `exp2()` to find 2^x.
- Use `log()` to find $log_e x$.
- Use `log10()` to find $log_{10} x$.
- Use `log2()` to find $log_2 x$.
- Use `frexp()` to find the significand of a binary normalised value.
- Use `pow()` to find x^y.
- Use `sqrt()` to find the square root of a number.
- Use `cbrt()` to find the cube root of a number.
- Use `hypot()` to find the hypotenuse of a right-angled triangle given the other two sides.
- Use `ceil()` to find the smallest integer not less than x.
- Use `floor()` to find the largest integer not greater than x.
- Use `fmod()` to find the real remainder after dividing two real numbers.
- Use `trunc()` to truncate a real number
- Use `round()` to round a real value to the nearest integer.

- Use `copysign()` on *x* and *y* to return a value *x* with the same sign as *y*.
- Use `fmax()` to find the larger of two numbers.
- Use `fmin()` to find the smaller of two numbers.
- Use `abs()` to find the absolute value of a number.
- If required, use `#include<ctime>` when calling time functions.
- If required use `#include <ctime>` when calling any of the following time-related functions.
- Use `clock()` to find the number of clock ticks that have occurred since the current program began execution.
- Use `time()` to find the current time and to set the time.
- Use `ctime()` to find the current date and time.
- Use `mktime()` to find the number of seconds between a given date-time and 1/1/1970.
- Use `difftime()` to find the difference in seconds between two times.
- Use asctime() to convert a time to sting giving day-of-week, date and time.
- Use `gmtime()` to convert a time to UTC time.
- Use `localtime()` to convert a time to local time.
- If required, use `#include<cstdlib>` when calling the following functions.
- Use `srand()` to seed the random number generator.
- Use `rand()` to generate a random number.
- Use `exit()` to deallocate variable space and exit a program.
- Use `getenv()` to discover the value of an OS variable.
- Use `system()` to execute a command line instruction.
- Use `getchar()` to read a single character from the keyboard.

Solutions

Activity 13.1

No solution required.

Activity 13.2

Modified code for *ElapsedTime*:

```
#include <iostream>
#include <ctime>
using namespace std;

int main()
{
  double cuberoot; //Cube root of number
  // *** Calculate and display cube roots ***
  for (int c = 1; c <= 1000; c++)
  {
    cuberoot = cbrt(c);
    cout << "Cube root of " << c << " is " <<
    ↳cuberoot << endl;
  }
  // *** Display the time taken ***
  cout << "Time taken : " << clock() << " ticks"
  ↳<< endl;
}
```

Activity 13.3

Modified code for *Clock*:

```
#include <iostream>
#include <ctime>
using namespace std;

int main()
{
  time_t t1, t2;   // time/date in seconds
  char *td;   // time date as string

  //*** Get time/date in seconds ***
  t2 = time(nullptr);
  //*** Show value returned ***
  cout << "Time returned in t2 = " << t2 << endl;
  //*** Convert to string ***
  td = ctime(&t2);
  //*** Display string ***
  cout << td << endl;
}
```

Activity 13.4

The first display is

```
Time in seconds : 60
Time/Date record : 0:1:0   1/1/1970
```

Modified code for *SecondsFrom*:

```
#include <iostream>
#include <ctime>
using namespace std;

int main()
{
  time_t t; // time/date in seconds
  tm dt;    // time/date record

  //*** Set time/date to 1 minute after midnight
  ↳1/1/1971 ***
  dt.tm_sec = 0;        //second
  dt.tm_min = 1;        //minute
  dt.tm_hour = 0;         //hour
  dt.tm_mday = 1;         //day of the month
  dt.tm_mon = 1 - 1;      //month (0 to 11)
  dt.tm_year = 1971 - 1900; //years since 1900

  //*** Get seconds passed since midnight 1/1/1970
  ↳***
  t = mktime(&dt);

  cout << "Time in seconds : " << t << endl;
  cout << "Time/Date record : " << dt.tm_sec << ':'
  ↳<< dt.tm_min << ':' << dt.tm_hour << "   "
  ↳<< dt.tm_mday << '/' << dt.tm_mon+1 << '/'
  ↳<< dt.tm_year+1900 << endl;
}
```

The display this time is

```
Time in seconds : 31536060
Time/Date record : 0:1:0   1/1/1971
```

Activity 13.5

Code for *LocalClock*:

```
#include <iostream>
#include <ctime>
using namespace std;

int main()
{
  //*** Day names ***
  char daynames[7][10] { "Sunday", "Monday",
  ↳"Tuesday", "Wednesday", "Thursday", "Friday",
  ↳"Saturday" };

  time_t now;

  //*** Set date time from system clock ***
  time(&now);

  //*** Transfer info to tm format ***
  tm *datetime = localtime(&now);

  //*** Display the values in datetime variable ***
  cout << "Date : " << datetime->tm_mday << '/'
  ↳<< datetime->tm_mon+1 << '/' <<
  ↳datetime->tm_year+1900 << endl;
  cout << "Time : " << datetime->tm_hour << ':'
  ↳<< datetime->tm_min << ':' <<
  ↳datetime->tm_sec << endl;
  cout << "Today is a " <<
  ↳daynames[datetime->tm_wday] << endl;
  cout << "This is day " << datetime->tm_yday+1
  ↳<< " of the year" << endl;
  if (datetime->tm_isdst == 1)
    cout << "British Summer Time is in operation\n";
  else
    cout << "British Summer Time is not in
    ↳operation\n";
}
```

Modified code for *LocalClock*:

```
#include <iostream>
#include <ctime>
using namespace std;

int main()
{
  //*** Day names ***
  char daynames[7][10] = { "Sunday", "Monday",
  ↳"Tuesday", "Wednesday", "Thursday", "Friday",
  ↳"Saturday" };

  time_t now;

  //*** Set date time from system clock ***
  time(&now);

  //*** Transfer info to tm format ***
  tm *datetime = localtime(&now);

  //*** Display the values in datetime variable ***
  cout << asctime(datetime) << endl;
}
```

Activity 13.6

With a bit of luck, you may get pi correct to two decimal places with 100,000 iterations.

Activity 13.7

Modified code for *User*:

```
#include <iostream>
```

```
using namespace std;

int main()
{
  cout << "Current user is " << getenv("PATH") <<
  ↳endl;
}
```

Activity 13.8

No solution required.

14

Strings and char Functions

In this Chapter:

- ❑ General String Handling Functions
- ❑ String Conversion Functions
- ❑ Character Classification Functions
- ❑ Character Modification Functions

C-Style String Functions

Introduction

C++ has two strategies for handling strings.

One of these it has inherited from its parent language C. This stores strings in `char` arrays and handles these arrays with the help of various functions. We have already covered some of these functions such as `gets_s()`, `strcpy_s()` and `strcmp()`. The remainder of this chapter covers the other string-handling functions available for manipulating `char` array strings.

The second strategy makes use of classes and hence belongs in a later chapter.

Defining Strings

Recap

So far we have encountered string constants (which are enclosed in double quotes), and string variables in the guise of `char` arrays.

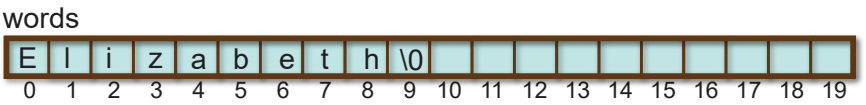
When a string is stored (either as a constant or in a `char` array) a *null* character is automatically added to mark the end of the string's sequence of characters. For example, if we set up a string variable with the line

```
char words[20] = "Elizabeth";
```

the structure created is that shown in FIG-14.1.

FIG-14.1

Storing a String



It is also quite valid to initialise a `char` array in the same way as we might initialise a numeric array with code such as

```
char word[20] = {'E','l','i','z','a','b','e','t','h', '\0'};
```

but, as we can see from the diagram, in this case, the *null* character must be added explicitly rather than leaving it to the program..

C++ treats `char` arrays differently from other types of arrays. Many statements allow the `char` array to be treated as if it were a single variable rather than a collection of cells. For example, we can read a value for words from the keyboard using the line:

```
cin >> words;
```

To display the contents of the same array we use the line

```
cout << words << endl;
```

The `cout` statement will display all the characters held in the *words* array up to (but excluding) the *null* character.

The biggest problem with string variables is that we need to ensure that they contain

sufficient space for any value that we attempt to store in them. Attempting to enter too much text into a `char` array will cause the program to crash or to have unpredictable effects. To avoid the danger of having insufficient space, programmers generally make their `char` arrays over-large just in case a particularly long string is assigned to it.

Although we get a runtime error when we try to enter too many characters for the space available, it might come as more of a surprise that if we entered the data *one two* only *one* was displayed.

Because of the limitations on entering a string which contains spaces, we can make use of the `gets_s()` function which takes the return character as the end of string indicator:

```
gets_s(words);
```

Now that we know a bit more about arrays and pointers, we can see why, having set up two strings with the statement

```
char txt[21], moretxt[21];
```

an attempted assignment using the line

```
txt = "Hello";
```

or a test for equality with

```
if (txt == moretxt)
```

are both invalid. (Remember an array name used without braces gives the start address of the array). This is why we need functions such as `strcpy_s()` and `strcmp()` perform the required assignment and comparison operations. Below are a list of other string functions available to us.

String Handling Functions



There is no requirement for additional `#include` statements when using any of the string functions shown in this section.

Other string handling functions available include the following:

strlen()

Function Protoype	:	size_t strlen(const char * txt)
Pre-condition	:	None
Description	:	Returns the number of characters in *txt*. The *null* character terminating the string is not included in the count.
Example	:	`int length = strlen("ABCD"); // length = 4`

strcat_s()

Function Protoype	:	errno_t strcat_s(char *dest, rsize_t n, const char * src)
Pre-condition	:	*dest* must be large enough to contain result.
Description	:	Adds a maximum of *n* characters to the end of *dest*. Adds a terminating null character to *dest*. Returns zero if successful, otherwise a non-zero value.
Example	:	`char txt[21] ="ABCD";` `strcat_s(txt,4,"EFG"); //txt contains ABCDEFG`

The program in FIG-14.2 makes use of the two functions above joining one string to a second only after checking that there is sufficient space for the operation.

FIG-14.2

Joining Two Strings

```
#include <iostream>
using namespace std;

int main()
{
   char st1[21], st2[21];
   // *** Read two strings ***
   cout << "Enter first string : ";
   gets_s(st1);
   cout << "Enter second string : ";
   gets_s(st2);

   //*** If there's enough space, join them ***
   if (strlen(st1) + strlen(st2) <= 20)
      strcat_s(st1,21,st2);
   else
      // ** else display error message **
      cout << "Result too long to store\n";

   // *** Display the contents of the two strings ***
   cout << "st1 contains " << st1 << "\nst2 contains " << st2 <<
   ↳endl;
}
```

Activity 14.1

Start a new project called *UsingStrings* and implement the code given above.

Test the program by using strings which are accepted and strings that are too large.

The copy, compare and append string functions we have seen so far deal with complete strings. Other functions allow the same type of operations using only part of a string (known as a **substring**).

strncpy_s()

Function Protoype	:	errno_t strncpy_s(char *st1, const char *st2, ↳size_t num)
Pre-condition	:	None
Description	:	Copies the first *num* characters in *st2* to *st1*. A terminating *null* character is added to *st1*. Previous contents of *st1* are deleted. Returns zero if successful, otherwise a non-zero value is returned.
Example	:	`char txt[21];` `strncpy_s(txt,"ABCDEFGHI",3); //txt contains ↳ABC`

errno_t is a redefinition of type `int`.

strncmp()

Function Protoype	:	int strncmp(const char *st1, const char *st2, ↳size_t num)
Pre-condition	:	None
Description	:	Compares the first *num* characters in *st2* to *st1*. If *st2* contains less than *num* characters *st2* is compared to *st1*. Returns 0 if the compared strings match; returns a positive value if *st1* > *st2*; returns a negative value if *st1* < *st2*.

For example, the lines

```
char txt[] {"ABCDEF"};
if (strncmp(txt,"ABXYZ",2) == 0)
   cout << "Compared characters are the same\n";
```

would cause *strncmp()* to return zero (since the first two characters in both strings are equal).

strncat_s()

Function Protoype	:	errno_t strncat_s(char *st1, rsize_t n, ↳const char *st2, size_t num)
Pre-condition	:	None
Description	:	Adds the first *num* characters in *st2* to the end of *st1*. n is the size of the *st1*'s space. Returns zero if successful, otherwise a non-zero value.

For example, the code

```
char txt[21] {"ABC"};
char more[] {"123456"};
strncat_s(txt,21,more,2);
```

will result in the first two characters in *more* being added to the end of *txt* with a null character being inserted to terminate the new string.

strchr()

Function Protoype	:	char* strchr(const char *st1, const int ch)
Pre-condition	:	None
Description	:	Returns a pointer to the first character in *st1* to match *ch*. If no match is found, *null* is returned.

For example, the lines

```
char txt[] {"ABCDEFCDE"};
char *ptr = strchr(txt,'C');
```

would return a pointer containing the address given by the term `&txt[2]`, which is

stored in *ptr*.

You may have noticed that the second parameter to `strchr()` is an `int` rather than a `char` as we might expect. This doesn't cause any problems since any `char` value used in the function call will automatically be converted to an `int`. However, we should be aware that we could perform exactly the same operation as before (searching for a capital C) using the line

```
char *ptr = strchr(txt,67);
```

since 67 is the ASCII code for a capital C.

This function can be used to achieve various results. For example, it can be used to count how often a character occurs within a string or to change each occurrence of a specified character to a new character.

Activity 14.2

Start a new project, *CharacterCount*, which reads in a string and a character then displays how often the character occurs within the string.

strrchr()

Function Protoype	:	char* strrchr(char *st1, const int ch)
Pre-condition	:	None
Description	:	Returns a pointer to the last character in *st1* to match *ch*. If no match is found, *null* is returned.

For example, the lines

```
char txt[] {"ABCDEFCDE"};
char *ptr = strrchr(txt,'C');
```

would return a pointer containing the address given by the term `&txt[6]`, which is stored in *ptr*.

strspn()

Function Protoype	:	size_t strspn(const char *st1, const char *st2)
Pre-condition	:	None
Description	:	Returns the number of characters examined in *st1* before finding a character that is not in *st2*.

For example, the lines

```
char txt[] {"ABCDEFCDE"};
cout << strspn(txt,"ADBC");
```

would display the value 4 since four characters in *txt* need to be examined before finding a character not in "ADBC".

strcspn()

Function Protoype	:	size_t strcspn(const char *st1, const char *st2)
Pre-condition	:	None
Description	:	Returns the number of characters examined in *st1* before finding a match for any character in *st2*. If no match is found, the size of *st1* is returned.

For example, the lines

```
char txt[] {"ABCDEFCDE"};
cout << strcspn(txt,"FXMP");
```

would display the value 5 since five characters in *txt* need to be examined before finding a character which matches any of those given in the second parameter.

strpbrk()

Function Protoype	:	char* strpbrk(const char *st1, const char *st2)
Pre-condition	:	None
Description	:	Returns a pointer to the first character in *st1* that matches a character in *st2*. If no match is found, *null* is returned.

For example, the lines

```
char txt[] {"ABCDEFCDE"};
char *chptr = strpbrk(txt,"FXMP");
```

would return the address *&txt[5]* since this is the first cell to contain a match for a character in the function's second parameter, *FXMP*.

strstr()

Function Protoype	:	char* strstr(char *st1, const char *st2, ↳size_t num)
Pre-condition	:	None
Description	:	Returns a pointer to the first occurrence of *st2* in *st1*. If no match is found, *null* is returned.

For example, the lines

```
char txt[] {"ABABCXFABCDMNP"};
char *chptr = strstr(txt,"ABCD");
```

would return the address *&txt[7]* since this is the first cell in the sequence which matches the second parameter, *ABCD*.

strtok()

Function Protoype	:	char* strtok(char *st1, const char *sep)
Pre-condition	:	None
Description	:	This function is used to separate the text in *st1* into tokens. Any character found in *st1* which matches a character in *st2* is assumed to mark the end of a token and is replaced by a null character. The first call to the function should reference the the string to be tokenised. Subsequent calls should use a value of NULL for *st1*. The current position within the *st1* is retained in a static variable. The function returns the start address of the token found. If the end of *st1* is reached, the function returns the value *null*.

Since this is a rather unusual function, it is perhaps worth looking in more detail at how it is used.

The program in FIG-14.3 uses the function to separate the words in a piece of text into separate strings. Pointers to each of these strings are held in an array of pointers and used to display the words extracted.

FIG-14.3

Using strtok()

```
#include <iostream>
using namespace std;

int main()
{
   //*** Text being examined ***
   char txt[] = "She took down a jar from one of the shelves as
   ↳she passed. It was labelled \"ORANGE MARMALADE\" but, to her
   ↳great disappointment, it was empty; she did not like to drop
   ↳the jar, so managed to put it into one of the cupboards as she
   ↳fell past it.";

   char separators[] {" "};   //Characters treated as separators
   char *tokens[100]{nullptr};   // Pointers to extracted tokens
   int count{0};              //Number of tokens found

   //*** Get pointer to first token ***
   char *ptr = strtok(txt, separators);
   //*** WHILE token found DO ***
   while (ptr != nullptr)
   {
      //*** Save address of token and increment count***
      tokens[count++] = ptr;
      //*** Get pointer to next token ***
      ptr = strtok(NULL, separators);
   }
   //*** Display tokens found ***
   for (int c = 0; c < count; c++)
      cout << tokens[c] << endl;
}
```

Activity 14.3

Start a new project, *Words*, which implements the code in FIG-14.3 and test your program.
Because only the space character is listed as a separator, quote marks, commas, semi-colons and full stops become part of a token.
Modify your code so that these characters are also treated as separators.

As we can see, this project has a similar effect to the one created in the previous Activity. However, here we need to make sure we have every possible separator character listed to get the correct results; in the previous program only the characters which were to be included in a token were required.

sprintf()

Back in Chapter 5 we looked at the `printf()` function which was the original method employed by C to output characters. `sprintf()` performs a similar operation but stores the output produced in a string (`char` array) rather than sending it to a stream. The function's mini-spec is:

Function Protoype	:	int sprintf(char* result, const char form, v1,v2...)
Pre-condition	:	None
Description	:	Values *v1*, v2, etc are converted to character format using formating info in *form* and, along with any other characters in *form* are stored in *result*.

Formating details given in parameter *form* start are the same as those used in the `printf()` function meaning they start with a % symbol, followed by flag details, field size and format code. (see FIG-14.4 and FIG-14.5).

FIG-14.4

Format Flags

Flag	Effect
+	Positive values preceded by +. Negative values by -
Δ	Positive values preceded by a space. Negative values by -
-	left-aligned output
#	with g and G, trailing zeros not removed with F,f,g,G,e and E, a decimal point is always included with o,x and X, 0 or 0X is included in the output
0	pad fields with zeros instead of spaces

FIG-14.5

Format Codes

Symbol	Meaning
d or **i**	signed integer
u	unsigned integer
f or **F**	fixed point float
e or **E**	scientific notation float (uses e or E)
g or **G**	fixed or scientific notation float as appropriate
x or **X**	hexadecimal (lower or upper case letters)
o	octal
c	character
s	string

This function can be particularly useful when trying to create a specific format within a string. For example, if we stored the current time in variables *hour*, *minute*, *second* and wanted the result to be of the form

hh:mm:ss

then we could store the required string in string *time* using the code:

```
char time[9];
sprintf(time, "%02d:%02d:%02d", hour, minute, second);
```

String Conversion Functions

atof()

Function Prototype	:	double atof(const char *str)
Pre-condition	:	None
Description	:	Returns a value equivalent to the numeric digits at the start of the string referenced by *str*. Whitespaces at the start of the string are ignored. If the first non-whitespace character is not a numeric digit (or + or -), then 0.0 is returned.
Example	:	`double ans = atof("-13.03abc"); //ans=-13.03`

strtod()

Function Prototype	:	double strtod(const char *str, char **eptr)
Pre-condition	:	None
Description	:	Returns a value equivalent to the numeric digits at the start of the string referenced by *str*. Whitespaces at the start of the string are ignored. If the first non-whitespace character is not a numeric digit (or + or -), then 0.0 is returned. If *eptr* is not *null*, then *eptr* contains the address of the first non-numeric character in the string following the numeric section. If *eptr* is null, then no pointer is created to the remainder of the string.

To get a better feel for how `strtod()` operates look at the program in FIG-14.6.

FIG-14.6

Using *strtod()*

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
   char *FirstNonNumeric ;
   double ans = strtod("13.3abc", &FirstNonNumeric);
   cout << "Number at start of string is " << ans << endl;
   cout << "Remainder of string is " << FirstNonNumeric << endl;
}
```

Activity 14.4

Start a new project, *StrToNum*, which implements the code given in FIG-14.6.

What values are displayed by the program?

strtof()

Function Prototype	:	float strtof(const char *str, char **eptr)
Pre-condition	:	None
Description	:	Returns a value equivalent to the numeric digits at the start of the string referenced by *str*. Whitespaces at the start of the string are ignored. If the first non-whitespace character is not a numeric digit (or + or -), then 0.0 is returned. If *eptr* is not null, then *eptr* is set to the address of the first non-numeric character in the string following the numeric section. If *eptr* is *null*, then no pointer is created to the remainder of the string.

strtold()

Function Prototype	:	long double strtold(const char *str, char **eptr)
Pre-condition	:	None
Description	:	Returns a value equivalent to the numeric digits at the start of the string referenced by *str*. Whitespaces at the start of the string are ignored. If the first non-whitespace character is not a numeric digit (or + or -), then 0.0 is returned. If *eptr* is not *null*, then *eptr* is set to the address of the first non-numeric character in the string following the numeric section. If *eptr* is *null*, then no pointer is created to the remainder of the string.

atoi()

Function Prototype	:	int atoi(const char *str)
Pre-condition	:	None
Description	:	Returns a value equivalent to the numeric digits at the start of the string referenced by *str*. Whitespaces at the start of the string are ignored. If the first non-whitespace character is not a numeric digit (or + or -), then 0 is returned.
Example	:	`int ans = atoi("-12"); //ans=-12`

atol()

Function Prototype	:	long int atol(const char *str)
Pre-condition	:	None
Description	:	Returns a value equivalent to the numeric digits at the start of the string referenced by *str*. Whitespaces at the start of the string are ignored. If the first non-whitespace character is not a numeric digit (or + or -), then 0 is returned.
Example	:	`long ans = atol("1234"); //ans=1234`

strtol()

Function Prototype	:	long strtol(const char *str, char **eptr, int bs)
Pre-condition	:	None
Description	:	Returns a value equivalent to the numeric digits at the start of the string referenced by *str*. Whitespaces at the start of the string are ignored. If the first non-whitespace character is not a numeric digit (or + or -), then 0.0 is returned. If *eptr* is not *null*, then *eptr* is set to the address of the first non-numeric character in the string following the numeric section. If *eptr* is *null*, then no pointer is created to the remainder of the string. The number at the start of the string is assumed to be a base *bs* number. The number within the string can start with an optional + or -. Hexadecimal numbers can start with a 0X or 0x. The characters which are accepted as part of the number depends on the base selected. For example, if *bs* =2 then only the digits 0 and 1 are accepted as part of the number; if *bs* = 36 then the characters 0 to 9 and A to Z are accepted. If *bs* = 0, the base is determined as 8, 10 or 16 with base 8 values starting with 0 and base 16 starting with 0X (or 0x).
Example	:	`long ans = ("0XFF",NULL,16)  // ans = 255`

strtoul()

This function takes the same parameters as `strtol()` but returns an `unsigned long` value.

strtoull()

This function takes the same parameters as `strtol()` but returns an `unsigned long long` value.

atoll()

Function Prototype	:	long long int atoll(const char *str)
Pre-condition	:	None
Description	:	Returns a value equivalent to the numeric digits at the start of the string referenced by *str*. Whitespaces at the start of the string are ignored. If the first non-whitespace character is not a numeric digits (or + or -), then 0 is returned.
Example	:	`long long ans = atoll("9876"); //ans=9876`

strtoll()

Function Prototype	:	long long strtoll(const char *str, char **eptr, int bs)
Pre-condition	:	None
Description	:	Returns a value equivalent to the numeric digits at the start of the string referenced by *str*. Whitespaces at the start of the string are ignored. If the first non-whitespace character is not a numeric digit (or + or -), then 0.0 is returned. If *eptr* is not *null*, then *eptr* is set to the address of the first non-numeric character in the string following the numeric section. If *eptr* is *null*, then no pointer is created to the remainder of the string. The number at the start of the string is assumed to be a base *bs* number. The number within the string can start with an optional + or -. Hexadecimal numbers can start with a 0X or 0x. The characters which are accepted as part of the number depends on the base selected. For example, if *bs* =2 then only the digits 0 and 1 are accepted as part of the number; if *bs* = 36 then the characters 0 to 9 and A to Z are accepted. If *bs* = 0, the base is determined as 8, 10 or 16 with base 8 values starting with 0 and base 16 starting with 0X (or 0x).
Example	:	`long long ans = ("041",NULL,0)  // ans = 65`

Summary

- Traditionally, strings are stored as an array of characters.
- The end of a string is marked using a null character.
- Use `strlen()` to find out how many characters are in a string.
- Use `strcat_s()` to join two strings together.
- Use `strncpy_s()` to copy a specified number of characters from one string to the a second string.

- Use `strncmp()` to compare part of one string with a second string.
- Use `strncat()` to add a specified number of characters from one string to the end of a second string.
- Use `strchr()` to find the address of the first occurrence of a character within a string.
- Use `strrchr()` to find the address of the last occurrence of a character within a string.
- Use `strspn()` to find the number of characters in a string which precede the occurrence of any character not from a second string.
- Use `strcspn()` to find the number of characters in a string which precede the occurrence of any character from a second string.
- Use `strpbrk()` to return the address of the first occurrence of one of a specified set of characters in a string.
- Use `strstr()` to find the first occurrence of a substring within a string.
- Use `strtok()` to split a string into tokens.
- Use `sprintf()` to output formated values to a string.
- Use `atof()` to convert a string containing numeric characters to a `double` value.
- Use `atoi()` to convert a string to an `int`.
- Use `atol()` to convert a string to an `long int`.
- Use `atoll()` to convert a string to an `long long int`.
- Use `strtod()` to convert a string to a `double` value and return a pointer to the first non-numeric character in the string.
- Use `strtof()` to convert a string to a `float` value and return a pointer to the first non-numeric character in the string.
- Use `strtold()` to convert a string to a `long double` value and return a pointer to the first non-numeric character in the string.
- Use `strtol()` to convert a string to a `long int` value and return a pointer to the first non-numeric character in the string. The base of numeric characters in the string can be specified.
- Use `strtoul()` to convert a string to an `unsigned long int` value and return a pointer to the first non-numeric character in the string. The base of numeric characters in the string can be specified.
- Use `strtoull()` to convert a string to an `unsigned long long int` value and return a pointer to the first non-numeric character in the string. The base of numeric characters in the string can be specified.
- Use `strtoll()` to convert a string to an `long long int` value and return a pointer to the first non-numeric character in the string. The base of numeric characters in the string can be specified.

Character Functions

Introduction

As well as containing a set of functions that operate on strings (`char` arrays), C++ also contains a set of functions used to classify or modify individual characters. In order to use these `char`-based functions, we may need to add the line

```
#include <cctype>
```

at the start of our programs although not all compilers require this.

char Classification Functions

isblank()



The example (and others below) assumes *ch* has been declared as a `char` variable.

C++ will automatically convert this to an `int` type when calling the function.

Function Protoype	:	int isblank(int ch)
Pre-condition	:	None
Description	:	Returns a non-zero value if *ch* is the code for a space character or tab character. Otherwise zero is returned.

We might reasonably have expected the parameter to this function to be of type `char` and the return value to be of type `bool`. However, this – and the remaining functions given here – were created originally for C at a time when `bool` and `char` types were not defined as part of the language.

Despite the type differences, C++ will perform the necessary type conversions for us allowing us to write code such as

```
char letter{'\t'};
if (isblank(letter))
   cout << "This is a space or tab character\n";
else
   cout << "This is not a space or tab character\n";
```

isspace()

Function Protoype	:	int isspace(int ch)
Pre-condition	:	None
Description	:	Returns a non-zero value if *ch* is the code for a whitespace character (space, tab, newline, return, vertical feed, or form feed). Otherwise zero is returned.

isscntrl()

Function Protoype	:	int iscntrl(int ch)
Pre-condition	:	None
Description	:	Returns a non-zero value if *ch* is the code for a non-printable character (codes 0x00 to 0x1F and code 0x7F). Otherwise zero is returned.

isprint()

Function Protoype	:	int isprint(int ch)
Pre-condition	:	None
Description	:	Returns a non-zero value if *ch* is the code for any printable character (includes the space character, but not other whitespace characters). Otherwise zero is returned.

isgraph()

Function Protoype	:	int isgraph(int ch)
Pre-condition	:	None
Description	:	Returns a non-zero value if *ch* is the code for a printable character which is visible. Otherwise zero is returned.

The result produced by `isgraph()` is identical to the results produced by `isprint()` except in the case of the *space* character which returns a non-zero value from `isprint()` and a zero value from `isgraph()`.

isalnum()

Function Protoype	:	int isalnum(int ch)
Pre-condition	:	None
Description	:	Returns a non-zero value if *ch* is the code for an alpha-numeric character (0-9, A-Z, a-z). Otherwise zero is returned.

isalpha()

Function Protoype	:	int isalpha(int ch)
Pre-condition	:	None
Description	:	Returns a non-zero value if *ch* is the code for an alphabetic character (A-Z, a-z). Otherwise zero is returned.

isdigit()

Function Protoype	:	int isdigit(int ch)
Pre-condition	:	None
Description	:	Returns a non-zero value if *ch* is the code for a numeric character (0-9). Otherwise zero is returned.

isxdigit()

Function Protoype	:	int isxdigit(int ch)
Pre-condition	:	None
Description	:	Returns a non-zero value if *ch* is the code for a hexadecimal character (0-9,a-f,A-F). Otherwise zero is returned.

ispunct()

Function Protoype	:	int ispunct(int ch)
Pre-condition	:	None
Description	:	Returns a non-zero value if *ch* is the code for a printable character which is not alpha-numeric (includes ,.+-?$%&*( ), etc.). Otherwise zero is returned.

Activity 14.5

Start a new project, *Characters*, and create a program containing a function called *PunctuationList()*, which returns the list of punctuation marks found in a supplied string. The function has the heading

```
void PunctuationList(char * p, const char *s)
```

where:

p	is the list of punctuation marks returned by the function.
s	is the string to be examined

Test the function using the string

"O! Don't cut my throat, sir," I pleaded.

isupper()

Function Protoype	:	int isupper(int ch)
Pre-condition	:	None
Description	:	Returns a non-zero value if *ch* is the code for an uppercase letter (A-Z). Otherwise zero is returned.

islower()

Function Protoype	:	int islower(int ch)
Pre-condition	:	None
Description	:	Returns a non-zero value if *ch* is the code for a lowercase letter (a-z). Otherwise zero is returned.

Activity 14.6

Start a new project, *CaseConverter*, and create a program containing a function called *ToUpperCase()*, which converts a complete string to uppercase. The function has the heading

```
ToUpperCase( char *s)
```

Test your program using the text
Coffin? - Spouter? - Rather ominous in that particular connexion, thought I.

Add a second function, *ToLowerCase()* which converts the string to lowercase and test it using the same text as above.

The program below (see FIG-14.5) analyses a piece of text and classifies the characters it contains. The output displays the original text, along with the number of uppercase and lowercase letters, punctuation marks and whitespaces.

FIG-14.5

Classifying Text

```
#include <iostream>
using namespace std;

int main()
{
   char text[]
   {
      "        JABBERWOCKY\n"
      "Twas brillig, and the slithy toves\n"
      "Did gyre and gimble in the wabe\n"
      "All mimsy were the borogoves,\n"
      "And the mome raths outgrabe.\n"
      "'Beware the Jabberwock, my son!\n"
      "The jaws that bite, the claws that catch!\n"
      "Beware the Jubjub bird, and shun\n"
      "The frumious Bandersnatch!'\n"
   };
   int counts[4]{ 0 };
   //*** Categorise each character ***
   for (auto v : text)
   {
      if (isalpha(v))
         if (isupper(v))
            counts[0]++;
         else
            counts[1]++;
      else if (ispunct(v))
         counts[2]++;
      else if (isspace(v))
         counts[3]++;
   }

   //*** Display results ***
   cout << "TEXT:\n" << endl;
   cout << text << endl;

   cout << "\n\nANALYSIS:\n"<< endl;
   cout << "Uppercase letters : " << counts[0] << endl;
   cout << "Lowercase letters : " << counts[1] << endl;
   cout << "Punctuation marks : " << counts[2] << endl;
   cout << "Spaces            : " << counts[3] << endl;
}
```

Activity 14.7

Start a new project called *TextAnalysis* and implement the code given above.

How many capital letters are counted?

char Modifier Functions

Although all of the functions we've looked at so far in this section are designed to categorise characters, there are two remaining functions that return modified versions of the character input.

toupper()

Function Protoype	:	int toupper(int ch)
Pre-condition	:	None
Description	:	If *ch* is the code for a lowercase letter (a-z), the code for the uppercase version of that letter is returned. If *ch* is not a lowercase letter, a copy of *ch* is returned.

tolower()

Function Protoype	:	int tolower(int ch)
Pre-condition	:	None
Description	:	If *ch* is the code for an uppercase letter (A-Z), the code for the lowercase version of that letter is returned. If *ch* is not an uppercase letter, a copy of *ch* is returned.

Activity 14.8

Modify *TextAnalysis* so that *text* is initialised to "Lewis Carroll 1832 - 1898" and display the text in uppercase. Remove the analysis code.

Summary

- Use `isblank()` to determine if a character is a space or tab.
- Use `isspace()` to determine if a character is a space, tab, return, newline, vertical feed, or form feed.
- Use `iscntrl()` to determine if a character is a control character (all characters with ASCII codes between 0 and 31 or code 127).
- Use `isprint()` to determine if a character is a printable character (includes the space character).
- Use `isgraph()` to determine if a character is a printable, visible character (excludes space).

- Use `isalnum()` to determine if a character is alphanumeric.
- Use `isalpha()` to determine if a character is alphabetic.
- Use `isdigit()` to determine if a character is numeric.
- Use `isxdigit()` to determine if a character is a hexadecimal digit (0-9, A-F, a-f).
- Use `ispunct()` to determine if a character is a punctuation mark.
- Use `isupper()` to determine if a character is an uppercase letter.
- Use `islower()` to determine if a character is a lowercase letter.
- Use `toupper()` to convert a letter to uppercase.
- Use `tolower()` to convert a letter to lowercase.

Solutions

Activity 14.1

No solution required.

Activity 14.2

Code for *CharacterCount*:

```
#include <iostream>
using namespace std;

int main()
{
  //*** Character searched for ***
  const char REQUIRED {'a'};

  //*** Text to be examined ***
  char txt[] {"Mary had a little lamb"};

  int count{0}; // Number of occurrences of required
                ↳character
  char *ptr{};  // Pointer to where character found

  //*** WHILE match found DO ***
  while ((ptr = strchr(txt, static_cast<int>
  ↳(REQUIRED))) != NULL)
  {
    //*** Increment count ***
    count++;
    //*** Erase this occurrence of the character
    ↳from string ***
    *ptr = '\xFF';
  }

  //*** Display number of times character found ***
  cout << REQUIRED << " occurs " << count <<
  ↳" times\n";
}
```

Activity 14.3

In *Words*, to add the extra separators, change the line

```
char separators[] = " ";  //Characters treated as
↳separators
```

to

```
char separators[] = " ,;.:"; //Characters treated as
↳separators
```

Activity 14.4

The program displays *13.3* and *abc*.

Activity 14.5

Code for *Characters*:

```
#include <iostream>
#include <cctype>
using namespace std;

void PunctuationList(char*, const char *);

int main()
{
  char txt[61];         //Text entered
  char punct[61]{""};     //Punctuation marks found

  //*** Get text ***
  cout << "Enter text : ";
  gets(txt);

  //*** Determine the punctaution marks in text ***
  PunctuationList(punct, txt);

  //*** Display the results ***
  cout << "The text contains " << strlen(punct) <<
  ↳" punctuation characters. These are " << punct
  ↳<< endl;
}

//*** Creates in p a list of punctuation marks found
↳in s ***
void PunctuationList(char *p, const char *s)
{
  int c = 0;  //Poisition in text being examined

  //*** Empty punction list ***
  p[0] = NULL;

  //***WHILE characters remaining in text DO ***
  while (s[c] != NULL)
  {
    //*** IF current character is a punctuation
    ↳mark THEN ***
    if (ispunct(s[c]))
    {
      //*** Add the character to list ***
      p[strlen(p) + 1] = NULL;
      p[strlen(p)] = s[c];
    }
    //*** Move to next character in text ***
    c++;
  }
}
```

Activity 14.6

Code for *CaseConverter*:

```
#include <iostream>
#include <cctype>
using namespace std;

//*** Function prototypes ***
void ToUpperCase(char *);
void ToLowerCase(char *);

int main()
{
  char txt[31]; // Text to be converted

  //*** Get text to be converted ***
  cout << "Enter text to be converted : ";
  gets(txt);

  //*** Convert text ***
  ToUpperCase(txt);

  //*** Display converted text ***
  cout <<"Uppercase version is : " << txt << endl;
}

//*** Converts s to uppercase ***
void ToUpperCase(char *s)
{
  for (int c = 0; c < strlen(s); c++)
    s[c] = toupper(s[c]);
}

//*** Converts s to lowercase ***
void ToLowerCase(char *s)
{
  for (int c = 0; c < strlen(s); c++)
    s[c] = tolower(s[c]);
}
```

Activity 14.7

The output produced is

```
TEXT:
      JABBERWOCKY
Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

'Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!'


ANALYSIS:
Uppercase letters : 22
```

Lowercase letters : 194
Punctuation marks : 12
Spaces : 54

Activity 14.8

Modified code for *TextAnalysis*:

```
#include <iostream>
using namespace std;

int main()
{
  char text[]{ "Lewis Carroll 1832 - 1898" };
  int counts[4]{ 0 };

  for (auto &v : text)
    v = toupper(v);

  cout << "TEXT:\n" << endl;
  cout << text << endl;
}
```

Notice that *v* has to become a reference variable in order to modify the contents of *text*.

15

Traditional File Handling in C

In this Chapter:

- ❑ Streams
- ❑ Text Files and Binary Files
- ❑ Reading and Writing a File
- ❑ Updating Records
- ❑ Searching a File
- ❑ File Housekeeping

Traditional File Handling in C

Introduction

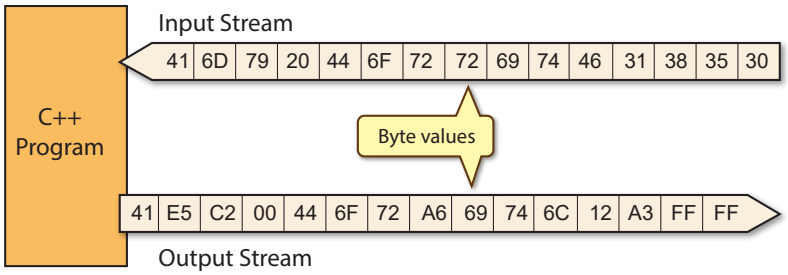
Data held exclusively within the variables of a program will be lost the moment that program terminates. If we want to hold on to data for longer, then it needs to be stored within a file on some backing storage media such as a hard disk or flash memory.

It is common practice when talking about C and C++ to use the all-encompassing term **stream** when referring to any data being read or written to any external device or even memory buffers.

We can consider a stream to be nothing more than a sequential series of bytes. If we are reading from a stream, then we are extracting one or more bytes from the stream; if writing to a stream we are adding bytes to that stream (see FIG-15.1).

FIG-15.1

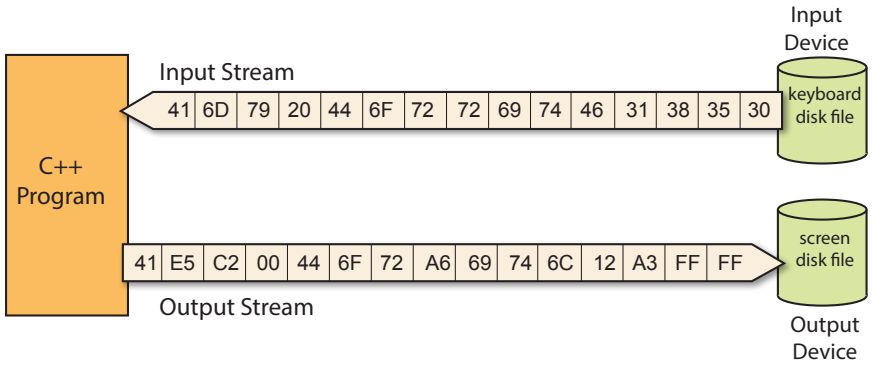
Streams



To use a stream we must first link it to a physical device or file. When we open a data file, we create a stream through which the data associated with the file is accessed (see FIG-15.2).

FIG-15.2

Streams and Peripherals



Any file or peripheral device can be linked to a stream. Hence, data entered at the keyboard is considered a stream as is the data sent to the screen or printer. In this chapter we will be exploring how to use a stream linked to a disk file.

The overall strategy for any program accessing a file can be summarised as:

```
Define a stream pointer variable
Associate the pointer with a disk file
```

Open the stream (for reading, writing, or updating)
Read/Write data to the stream
Close the stream

C++ has its own methods for handling files and streams but it has also inherited older techniques from C. It is these older methods (which are still widely used) which are examined in this chapter. The newer C++ only methods are described in a later chapter.

File Formats

Before we begin to look at the instructions for file handling, it is worth pointing out that the language makes use of two types of files. These are:

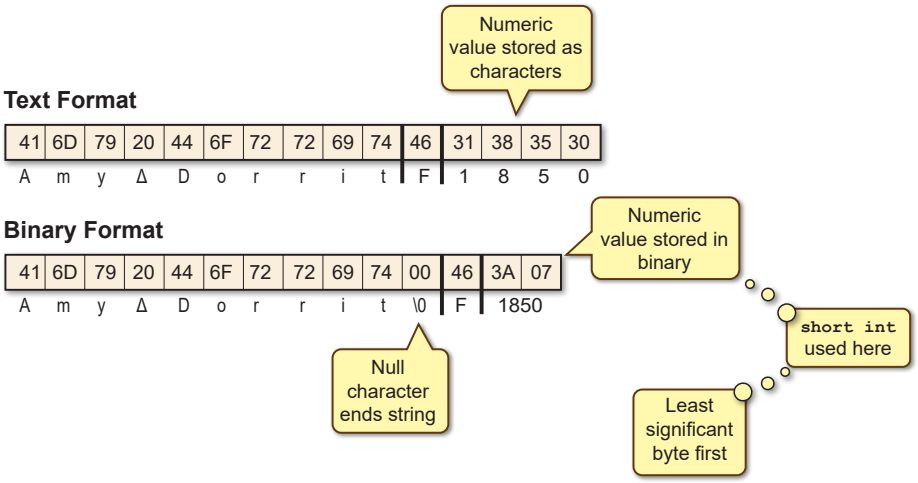
- Text files
- Binary files

The main difference between these two types of files is a simple one: a text file holds all its data in character format, while a binary file holds its data in the same format as that used within a variable.

FIG-15.3 shows how the data *"Amy Dorrit", 'F', 1850* is written to the two types of file.

FIG-15.3

Text and Binary Storage Formats



Binary files have several advantages over text files. These include:

- Numeric data often occupies less space in binary mode.
- Execution time may be less since numeric values do not have to be converted to characters before being stored.
- It is easier to determine where one value ends and another begins.
- If a binary file contains many records, those records can be accessed with greater efficiency because every record is exactly the same size.

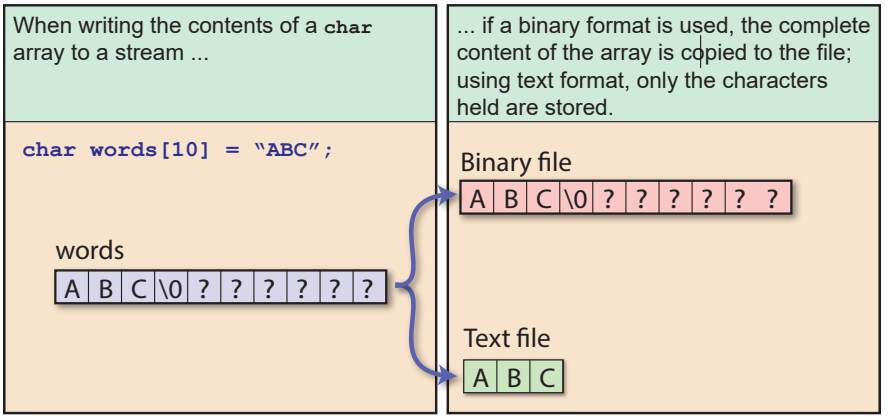
However, there are some disadvantages:

- The contents of the file cannot be understood when displayed using a simple text editor.
- A dedicated program is required to make sense of the file contents.
- The value of every byte in a variable is copied exactly to the file even when those bytes contain no data. In particular every byte of a `char` array is stored - even when that element contains no valid character.

This last point is explained further in FIG-15.4.

FIG-15.4

Storing Strings in Text and Binary Formats



As a general rule, text files are used when the data being stored is mostly non-numeric such as might be created by a text editor or an error log generator with no requirement to do extensive searching of the data. Binary files are suited to data-oriented applications where entries may be inserted, deleted or modified and where the data needs to be searched such as in a database.

Using Text Files

The easiest way to understand the concept behind the use of streams is to build a simple example.

Creating a Stream Pointer

A stream pointer is used to reference a stream. The data area referenced by this pointer contains the name and status of the file being used as well as a buffer area to store information transferred to/from the stream. Stream pointers are declared as type `FILE` (which is a term defined in *iostream* and is part of the *std* namespace). A typical declaration would be:

```
FILE *myfile;
```

Linking the Stream Pointer to a Disk File

The stream pointer needs to be linked to a disk file (or a peripheral device). This is done using the `fopen()` function.

This function takes two parameters: the name of the file to be opened and the I/O mode to be used in conjunction with the file format. If the file is to be read then it must already exist. If the file is to be written, then it will be created automatically if it does not already exist. When writing to an existing file any data already held in the file can be deleted or added to.

The exact options for `fopen()` are explained in the function descriptor below.

Function Prototype : `FILE* fopen(const char* fn, const char mode)`
Pre-condition : *fn* is a valid filename AND *mode* is a valid mode
Description : Opens the file specified by *fn* and returns a reference to the stream that has been associated with the file. *mode* specifies the I/O mode and file format.
Basic options for *mode* are:

mode	Meaning
r	Opens for reading. File must exist.
w	Opens for writing. If file does not exist, it is created. If it does exist, current contents are deleted.
a	Opens for writing. If file does not exist, it is created. If it does exist, new data is added to end of existing data.
r+	Opens for updating. The file must exist. Current contents can be read and modified.
w+	Opens for updating. If file does not exist, it is created. If it does exist, current contents are deleted. New values can be read, written or updated.
a+	Opens for updating. If file does not exist, it is created. If it does exist, new data is added to end of existing data. Data can be updated.

Add *t* or *b* to the mode value to signify text (*t*) or binary (*b*) file.
If the file cannot be opened, null is returned.

A typical start to a file handling program would be

```
#include <iostream>
using namespace std;

int main()
{
   FILE *myfile;   // Stream reference variable

   // *** Open text file for writing ***
   myfile = fopen("C:\\MyC++Programs\\TestFile.dat","wt");
```

The double backslash in the filename path is required since the \ character is one which can only be defined using an escape sequence. The character pair will be treated as a single backslash.

It is possible that the file cannot be opened for writing. This might occur because the backing store device is full or the program does not have write permission. So to make our code more rigorous, we should go on to include the lines

```
if (myfile == NULL)
{
   cout << "Could not open file\n";
   exit(1);
}
```

which will display an error message and exit the program if there's a problem.

Closing a File

When we have finished reading or writing to a file, we must close it. This closes the stream and, when writing to the file, transfers any remaining data from the stream buffer to the actual file on backing store. To close a file use the `fclose()` function. It's details are given below.

Function Prototype	: `int fclose(FILE *strmptr)`
Pre-condition	: *strmptr* must reference an open stream.
Description	: Closes the stream referenced by *strmptr*. The buffer associated with *strmptr* is empty with any remaining data being written to the file if the stream is an output one. The space referenced by *strmptr* is freed and the file closed. Returns zero if the stream is closed successfully, otherwise -1 is returned.

Writing to a Text File

Now that we know how to open and close a file, we need to find out how to write to a file. The simplest way to write to a text file is to use the `fprintf()` function. This uses the same formatting string approach that we saw back in Chapter 4 when looking at the `printf()` function. Like `printf()`, `fprintf()` converts all values (including numeric ones) to characters. The only difference between the two functions is that `fprintf()` parameters include a `FILE` pointer. The function details are given below.

Function Prototype	: `int fprintf(FILE* strmptr, const char* form,...)`
Pre-condition	: *strmptr* must reference an open write stream.
Description	: Outputs the values specified using the form string to format the output. Returns the number of bytes written to the stream.

The format options (which must be preceded by the `%` character) are shown in FIG-15.5.

FIG-15.5

fprintf() Format Options

Symbol	Meaning
d or **i**	signed integer
u	unsigned integer
f or **F**	fixed point float
e or **E**	scientific notation float (uses e or E)
g or **G**	fixed or scientific notation float as appropriate
x or **X**	hexadecimal (lower or upper case letters)
o	octal
c	character
s	string

So to write the data shown in FIG-15.3 to the file, we could use the statement

```
fprintf(myfile,"%s%c%d","Amy Dorrit",'F',1850);
```

Notice that, since we have no use for the value returned by `fprintf()`, it is not assigned to a variable.

The complete program used to write to the file is given in FIG-15.6.

FIG-15.6

Writing to a Text File

```
#include <iostream>
using namespace std;

int main()
{
   FILE *myfile;   // Stream pointer

   // *** Open file ***
   myfile = fopen("C:\\MyC++Programs\\test.dat","wt");
   // *** If there's a problem, exit program ***
   if (myfile == NULL)
   {
      cout << "Could not open file\n";
      exit(0);
   }

   //*** Write data to file ***
   fprintf(myfile,"%s%c%d","Amy Dorrit",'F',1850);

   //*** Close the file ***
   fclose(myfile);
   //*** Tell user program is complete ***
   cout << "Data written to file\n";
}
```

Since no output is produced on the screen when writing to a file, the program includes a final message to inform the user that the data has been written to the file.

Activity 15.1

Start a new project called *WriteToFile* and implement the code given in FIG-15.5 changing the path information in the `fopen()` statement to an appropriate value for your computer.

To check that the data was written correctly, open the newly created file in a text editor such as Notepad to examine its contents.

Although `fprintf()` is the obvious choice when writing to a text file, other functions are available. These include `fputs()` for writing a single string and `fputc()` for writing a single character.

Function Prototype	: `int fputs(const char *txt, FILE *strmptr)`
Pre-condition	: *strmptr* must reference an open write stream.
Description	: Outputs the string *txt* to the file referenced by *strmptr*. The terminating null character is not written to the file. Returns -1 if an error occurs, otherwise a positive value is returned.

Function Prototype	: `int fputc(int ch, FILE *strmptr)`
Pre-condition	: *strmptr* must reference an open write stream.
Description	: Outputs the character *ch* (which is given in numeric form) to the file. Returns the code of the value written to the file if successful, otherwise -1 is returned.

When writing a string to the file we can use a statement such as

```
fputs("Amy Dorrit", myfile);
```

while a character could be written with the line

```
fputc('F',myfile);
```

Without the use of `fprintf()`, writing a numeric value takes a bit more effort since it must first be converted to a string:

```
char temp[6];
itoa(1850,temp,10);
fputs(temp,myfile);
```

Separating Data Items

Although we've managed to write the data to the file, a problem is going to arise when we try to read the information back since, with the three values written to the file being joined into a single sequence of characters, there is no way to tell where one item of data ends and the next begins.

To solve this problem, we could write a newline character (`\n`) after each data item by changing the format string in the `fprintf()` call to

```
"%s\n%c\n%d\n"
```

Activity 15.2

Modify *WriteToFile*, so that a newline character is inserted after each data item.

Run the program then examine the contents of the file checking that each data item occupies a separate line. Save your program.

When writing the contents of arrays or structures to a file, each cell or field must be written separately.

Activity 15.3

Start a new project called *WriteRecord*. In your program, define the following data structure type:

```
struct BestScores
{
   char name[31];
   int scores[5];
};
```



Activity 15.3 (continued)

Create a variable of this type, assigning it the values (*"Arthur Clennam",219, 208,185,170,162*).

Write the data from each field in the record to a file named *Scorefile.dat*, separating each value by a newline character.

Run the program and examine the contents of the file.

Reading a Text File

The easiest option when reading from a file is to use `fscanf()`. This is a file-handling version of `scanf()` which we covered back in Chapter 4.

Function Prototype	: `int fscanf(FILE* strmptr, const char* form,...)`
Pre-condition	: *strmptr* must reference an open read stream.
Description	: The types of values to be read are given in the format string *form*. This is followed by the address of the variables in which the values being read are to be stored. Returns the number variables that have been assigned a value.

The options available within the formatting string are shown in FIG-15.7. Each option should be preceded by the `%` character.

FIG-15.7

Writing to a Text File

Type Specifier	Meaning
c,C	char/wide char
d	integer
e, E,f, g, G	fixed point float
o	octal
s	string
u	unsigned integer
x,X	hexadecimal

We might be tempted to think that we can read back the contents of the file we created in the *WriteToFile* project by making use the following lines:

```
myfile = fopen("C:\\MyC++Programs\\test.dat","rt");
// *** If there's a problem, exit program ***
if (myfile == NULL)
{
   cout << "Could not open file\n";
   exit(0);
}
//*** Read data from file ***
char name[21];
char gender;
int year;
fscanf(myfile,"%s %c %d",name,&gender,&year);
```

but, unfortunately, we would be disappointed by the results. Like its relative `scanf_s()`, `fscanf()` assumes the data for a string is complete when any whitespace character (space, tab or newline) is detected. This means it will stop reading a value for *name* when the space between *Amy* and *Dorrit* is encountered.

An alternative way to read strings is to use the `fgets()` function which uses the newline character as an end-of-string marker.

Function Prototype	: `char* fgets(char * txt, int max, FILE* strmptr)`
Pre-condition	: *strmptr* must reference an open read stream.
Description	: Reads a maximum of *max* characters into *txt* from the file referenced by *strmptr*. Returns *txt* unless there is a problem during the read operation in which case *null* is returned. If the data read ends with a newline character, this is included in the string stored in *txt*.

Since any string returned by `fgets()` may include a newline character, we can look for this and delete it using the code

```
int size = strlen(txt);
if (size > 0 && txt[size-1] == '\n')
   txt[size-1] = '\0';
```

The complete program for reading back the contents of *test.dat* is given in FIG-15.8.

FIG-15.8

Reading From a Text File

```
#include <iostream>
using namespace std;

int main()
{
   FILE *myfile;   //Stream pointer

   //*** Open the file for reading ***
   myfile = fopen("C:\\MyC++Programs\\Scorefile.dat", "rt");
   // *** If there's a problem, exit program ***
   if (myfile == NULL)
   {
      cout << "Could not open file\n";
      exit(0);
   }

   //*** Read data from file ***
   char name[21]; // Name read from file
   char gender;   // Gender read from file
   int year;      // Year read from file
   fgets(name, 20, myfile);
   fscanf(myfile, "%c %d", &gender, &year);
   //*** Close file ***
   fclose(myfile);

   //*** Delete any newline character in name ***
   int size = strlen(name);
   if (size > 0 && name[size - 1] == '\n')
      name[size - 1] = '\0';
   //*** Display info read ***
   cout << "Data read : ";
```



FIG-15.8
(continued)

Reading From a Text File

```
    cout << name << " " << gender << "   " << year << endl;

    //*** Tell user display is complete ***
    cout << "End of file data read\n";
}
```

Activity 15.4

Start a new project called *ReadFromFile* which implements the code given in FIG-15.8.

Activity 15.5

Start a new project called *ReadFromFile2* which reads the data held in *Scorefile.dat* (created in Activity 15.3). Display the values read, removing any newline characters attached to strings read from the file.

The most basic of the file-reading functions is `fgetc()` which will read a single character from a file.

Function Prototype	: `int fgetc(FILE* strmptr)`
Pre-condition	: *strmptr* must reference an open read stream.
Description	: Reads a single character from the file referenced by *strmptr*. Returns the read character's value as an integer.

Activity 15.6

Start a new project called *ReadCharsFromFile* which uses *fgetc()* to read the first 15 characters in the file *test.dat* displaying the value read in both integer and character format.

When we read a file one character at a time, we run into the problem of how to detect the fact that the last character has been read and that it is time to close the file.

We can detect the end-of-file situation by comparing the character just read with a special named constant value, EOF. So, the logic for reading a text file one character at a time is:

```
Open text file for reading
Read a character from the file
WHILE character not EOF DO
    Process character
    Read a character from the file
ENDWHILE
Close file
```

Activity 15.7

Modify *ReadCharsFromFile* so that all characters in the file are read and displayed with reading stopping when EOF is detected by *fgetc()*. A count of the number of characters read should also be displayed.

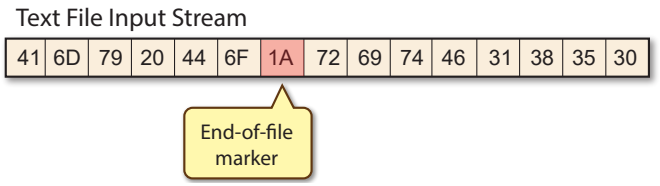
End-Of-File Byte

The end of a file is calculated by the operating system from the details it holds on the file's length. There is no special end-of-file character stored within the file.

However, this was not true in previous operating systems. In CP/M, a forerunner to MSDOS and Windows, the character $1A_{16}$ was used to mark the end of a file (see FIG-15.9).

FIG-15.9

Text File's End-of-File Marker Byte



And this value can still be used in a Windows environment to trigger EOF in a text file. A simple example of this is given by the program in FIG-15.10 where a byte containing the value $1A_{16}$ is written to a file between the words *Amy* and *Dorrit*. If another program attempts to read the contents of the file one character at a time, the byte containing $1A_{16}$ will trigger the EOF condition and halt the program before the true end of the file is reached.

FIG-15.10

Using the End-of-File Byte

```
#include <iostream>
using namespace std;

int main()
{
   FILE *myfile;

   //*** Open file for writing ***
   myfile = fopen("C:\\MyC++Programs\\test.dat","wt");
   // *** If there's a problem, exit program ***
   if (myfile == NULL)
   {
   //*** Write data to file ***
      cout << "Could not open file\n";
      exit(0);
   }
   fprintf(myfile,"%s%c%s\n%c\n%d\n","Amy",'\x1A',"Dorrit",'F',
   ↳1850); // EOF(1A) value written
   //*** Close the file ***
   fclose(myfile);
}
```

Activity 15.8

Start a new project, *FileMarker*, and implement the code given above. Run the program to modify the contents of *test.dat*.

Reload and run *ReadCharsFromFile* and observe what values are displayed.

Reload and run *WriteToFile*, to restore the original contents of *test.dat*.

Adding Data to a Text File

Often we will want to add new data to the end of a file, placing the latest values

immediately after the last of the existing values within the file. To do this we need to open the file in append mode.

```
myfile = fopen("C:\\C++MyPrograms\\test.dat","at");
```

Any subsequent instruction which writes data to the file will add that new data to the end of the file, immediately after any existing data.

Activity 15.9

Create a new project, *AddDataToFile*, which adds the data

Arthur Clennam, M, 1830

to the file *test.dat*.

Use Notepad (or similar) to check that the file now contains information on both *Amy Dorrit* and *Arthur Clennam*.

Binary Files

As already stated, one of the main differences between binary and text files is that binary files store exact copies of how data is stored in memory while text files store every element of a data value in character format.

Other differences on Microsoft Windows platforms are:

- the newline character (`\n`) is written to a binary file using a single byte (value $0A_{16}$); in a text file newline is written over two bytes ($0D_{16}$ $0A_{16}$).
- the value $1A_{16}$ has no special significance in a binary file; in a text file the value is taken as an end-of-file marker.

Now that we are aware of these differences between the two file types, we can get on with looking at an example of binary file usage.

To indicate to a program that a binary file is being used, the final parameter for the `fopen()` function must end with a *b*.

It is the responsibility of the programmer to ensure that the correct type of file is being opened. For example, there is no safeguard built in to ensure a text file is not opened for reading as a binary file.

Writing to a Binary File

A common use of binary files is to hold data records. For example, let's assume a program defines the following structure:

```
struct DetailsType
{
   char name[31];
   char sex;
   int age;
};
```

and that an array, *fiction*, is created to hold five of these records:

```
   DetailsType fiction[5];
```

then we can write the contents of this array (assuming some other part of the program has populated it with data) using the `fwrite()` function.

Function Prototype	: `size_t fwrite(const void *ptr, size_t blksz, size_t nblks, FILE* strmptr)`
Pre-condition	: *strmptr* must reference an open output binary stream.
Description	: Writes *nblks* blocks to the file. Each block is *blksz* bytes in length. The data written is taken from the area referenced by *ptr*. Returns the number of blocks written. If successful, this should match *nblks*.

At first glance, this is quite a strange approach to writing when we compare it to the more obvious functions used for text files. However, let's see how we would use the function to write the contents of the *fiction* array to a binary file.

The first parameter required of `fwrite()` is a pointer to the start of the data that is to be written to the file. Since the data is held in the array *fiction*, our command would start with

```
fwrite(fiction,
```

Next we need to specify the size of each record (or data block) in bytes. Now, with a bit of effort we can work this out as:

```
name  : 31 bytes
sex   : 1 byte
age   : 4 bytes

Total : 36 bytes
```

But the easiest (and safest) way to get the size of the record is to use the `sizeof` command, making our statement

```
fwrite(fiction, sizeof(DetailsType),
```

Since the array, *fiction*, contains five records, and assuming we want to write all of these to the file referenced by *myfile*, the complete `fwrite()` statement is:

```
fwrite(fiction,sizeof(DetailsType),5,myfile);
```

Notice that not only a whole record, but the whole array is written to disk in a single statement.

The program in FIG-15.11 shows the complete code for writing the contents of the *fiction* array to a binary file. The following simple logic is employed:

```
Open binary file for writing
Write records to file
Close file
```

FIG-15.11

Writing to a Binary File

```
#include <iostream>
using namespace std;

//*** Define structure ***
struct DetailsType
```



FIG-15.11
(continued)

Writing to a Binary File

```
{
   char name[31];
   char sex;
   int age;
};

int main()
{
   //*** Set up data to be written ***
   DetailsType fiction[]= {{"Amy Dorrit",'F',20},
   {"Arthur Clennam",'M',40},{"Jeremiah Flintwitch",'M',67},
   {"Pet Meagles",'F',18},{"Daniel Doyce",'M',36}};

   //*** File pointer ***
   FILE *myfile;

   //*** Open binary file for writing ***
   myfile = fopen("C:\\MyC++Programs\\fictional.dat","wb");
   // *** If there's a problem, exit program ***
   if (myfile == NULL)
   {
      cout << "Could not open file\n";
      exit(0);
   }

   //*** Write all records to the file ***
   fwrite(fiction,sizeof(DetailsType),5,myfile);

   //*** Close the file ***
   fclose(myfile);
   cout << "Data written to file" << endl;
}
```

Activity 15.10

Create a new project, *WriteToBinaryFile,* which implements the code in FIG-15.11.

Try looking at the contents of the *Fictional.dat* using at text editor such as Notepad.

Reading from a Binary File

A binary file is read using the `fread()` function which uses a similar set of parameters to `fwrite()`.

Function Prototype	: `size_t fread(const void *ptr, size_t blksz, size_t nblks, FILE* strmptr)`
Pre-condition	: *strmptr* must reference an open input binary stream .
Description	: Reads *nblks* blocks from the file. Each block is *blksz* bytes in length. The data is read into the area referenced by *ptr*. Returns the number of blocks read. If successful, this should match *nblks*.

The program in FIG-15.12 reads back the information we wrote to a file in the previous program.

FIG-15.12

Reading from a Binary File

```
#include <iostream>
using namespace std;

//*** Define structure ***
struct DetailsType
{
   char name[31];
   char sex;
   int age;
};


int main()
{
   //*** Set up array to receive data ***
   DetailsType fiction[5];
   //*** File pointer ***
   FILE *myfile;

   //*** Open binary file for reading ***
   myfile = fopen("C:\\MyC++Programs\\fictional.dat","rb");
   // *** If there's a problem, exit program ***
   if (myfile == NULL)
   {
      cout << "Could not open file\n";
      exit(0);
   }
   //*** Read all records from the file ***
   fread(fiction,sizeof(DetailsType),5,myfile);

   //*** Close the file ***
   fclose(myfile);

   //*** Display data read ***
   for(int c = 0; c <= 4; c++)
      cout << fiction[c].name << ' ' << fiction[c].sex << ' '
      ↳<< fiction[c].age << endl;
}
```

Activity 15.11

Create a project called *ReadFromBinaryFile* using the code in FIG-15.12 which reads the contents of the previously created binary file, *Fictional.dat*, and displays the values read.

Detecting the End of File

When a file containing an unknown number of records is to be read from start to end, we need some way of knowing when the last record has been read. The EOF byte check used when reading a text file is not appropriate in a binary file because the value $1A_{16}$ could easily appear in a byte allocated to a numeric value. Instead we can check that, when using `fread()`, that the value returned by the function is equal to the number of data blocks specified in the function's parameter list.

When reading one block (record) at a time, the standard approach is to use the following logic:

```
Open file for reading
Attempt to read a record
WHILE record successfully read DO
    Process record read
    Attempt to read a record
ENDWHILE
Close file
```

If `fread()` returns zero, then no record was read and we can assume that this means that the end of the file has been reached. The program in FIG-15.13 implements the logic given above.

FIG-15.13

Detecting the End-of-File in a Binary File

```
#include <iostream>
using namespace std;

//*** Define structure ***
struct DetailsType
{
   char name[31];
   char sex;
   int age;
};


int main()
{
   DetailsType fiction; //Record to hold data read from file

   FILE *myfile;         //File pointer

   //*** Open binary file for reading ***
   myfile = fopen("C:\\C++MyPrograms\\fictional.dat","rb");
   // *** If there's a problem, exit program ***
   if (myfile == NULL)
   {
      cout << "Could not open file\n";
      exit(0);
   }

   //*** Read one record at a time from the file ***
   while (fread(&fiction,sizeof(DetailsType),1,myfile) != 0)
      cout<<fiction.name<<' '<<fiction.sex<<' '<<fiction.age<<endl;

   //*** Close the file ***
   fclose(myfile);

   //*** Tell user the process is complete ***
   cout << "Reading complete\n";
}
```

However, the code above does not guarantee that all records in the file have been read. If the file has been corrupted in some way, `fread()` will fail to read a record at some point (and hence return zero) even though the end of the file has not been reached.

Although there may be little that can be done to retrieve the corrupt data, the program should at least report that it has terminated reading before the end of the file has been reached.

This can be achieved using the `feof()` function which returns 1 when the true end of the file is detected.

Function Prototype	: `int feof(FILE* strmptr)`
Pre-condition	: *strmptr* must reference an open input binary stream .
Description	: Returns an non-zero value if the end of file has been detected, otherwise zero is returned.

The `feof()` function should be called after `fread()` has returned zero. Typical code would be

```
if (!feof())
   cout << "A file read error has occurred.\n";
```

Activity 15.12

Create a new project called *DisplayRecords* which makes use of the code in FIG-15.13 to read and display all the records in *fictional.dat*.

Add the code to display an error message if not all the records are read.

Adding Data to a Binary File

Adding new data to the end of binary file is little different from adding data to a text file. The file must be opened in append mode (`"ab"`) and new data written to the file will automatically be added after the existing file data.

Activity 15.13

Create a new project, *AddToBinaryFile*, which adds the following record to the file *fictional.dat*.

Edmund Sparkler, M, 24

By running *DisplayRecords* which you created in the previous Activity, check that the new record has been successfully added to the file.

Searching a Binary File

A common requirement of any file handling system is to search for a specific record within the file. In the case of our *fictional.dat* file, we might do this by specifying the name we are searching for and then examining each record in turn until a match is found or we reach the end of the file. The logic for such a program could be described as:

```
Open file for reading
Get name to be searched for
Attempt to read a record
WHILE record read AND name in record not one being searched for DO
    Attempt to read a record
ENDWHILE
Close file
IF name in last record read matches one searched for THEN
    Display record details
ELSE
    Display "No match found"
ENDIF
```

Activity 15.14

Create a new project called *FindRecord* which implements the logic given above, allowing the details of a required name to be extracted from *fictional.dat* and displayed on the screen. The name search should not be case sensitive.

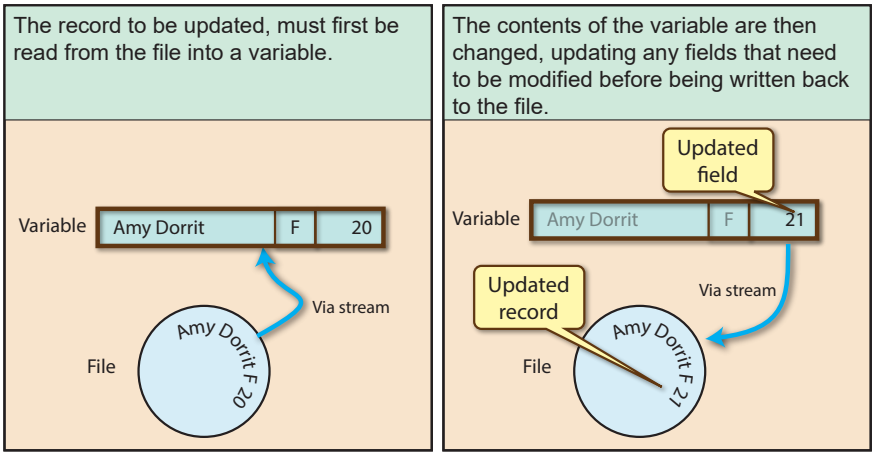
Updating Records

Information in a file often needs to be modified. For example, if we have contact details on our phone we will, no doubt, already have had to change the address or cell phone number for people who have moved or bought a new phone.

To update records in a binary file we need to start by retrieving the required record from the file, copying the data into a variable of the appropriate type, changing the contents of that variable as required, and then writing the data back to the file (see FIG-15.14).

FIG-15.14

Updating a File Record - Concept



There is one more step to be considered when updating records and that is the file position indicator. When we use any sort of text editor or word processor, the cursor shows our current position in the text. Type more characters and they will be inserted at the position of the cursor. Data files operate on the same sort of principle, maintaining a file position indicator showing where in the file data the next read or write statement will access. When a read or write statement is executed the file position indicator moves on automatically (see FIG-15.15).

FIG-15.15

File Position Indicator

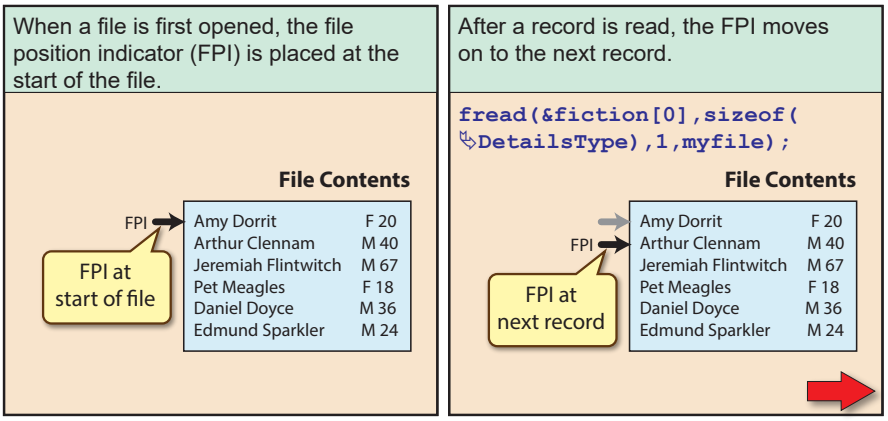
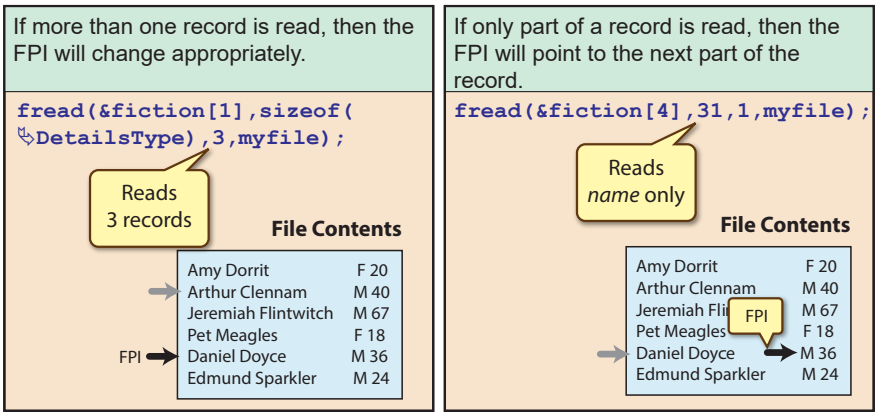


FIG-15.15
(continued)

File Position Indicator



After we read a record from a file, the file position indicator will have moved on to point to the next record. If we try to write our updated record back to the file, it will use the FPI to determine where to write that record.

If a program reads the *Amy Dorrit* record, changes that information, and then tries to write the modified data back to the file, this will result in the *Arthur Clennam* record being overwritten since the file position indicator will, by this time, have moved passed the original *Amy Dorrit* record to point at the second record in the file.

In order to update the *Amy Dorrit* record correctly, we need to reposition the FPI to the start of the original record before attempting to write the update record back to the file. This can be done using the `fseek()` function.

Function Prototype : `int fseek(FILE* strmptr, long offset, int post)`
Pre-condition : *strmptr* must reference an open stream .

Description : Moves the FPI of *strmptr* by *offset* bytes from position *post*.
post must be 0, 1, or 2 or the named constants:

Symbolic Name	Value	Starting position
SEEK_SET	0	Start of file
SEEK_CUR	1	Current position
SEEK_END	2	End of file

Returns zero if successful, otherwise a non-zero value is returned.

The logic we require to update the first record in our *fictional.dat* file, changing the value of the age field from 20 to 21 is:

Open file in r+b mode
Read first record from file to variable
Change value in age field of variable to 21
Move the file position indicator back to the start of the file
Write the variable to the file

The program required to perform this task is given in FIG-15.16.

FIG-15.16

Updating a File Record - Application

```
#include <iostream>
using namespace std;

//*** Define structure ***
struct DetailsType
{
   char name[31];
   char sex;
   int age;
};

int main()
{
   DetailsType fictionrecord;  //*** Holds one record ***
   FILE *myfile;               //*** File pointer ***

   //*** Open binary file for updating ***
   myfile = fopen("C:\\MyC++Programs\\fictional.dat","r+b");
   // *** If there's a problem, exit program ***
   if (myfile == NULL)
   {
      cout << "Could not open file\n";
      exit(0);
   }
   //*** Read one record ***
   fread(&fictionrecord,sizeof(DetailsType),1,myfile);
   //*** Update age field ***
   fictionrecord.age++;
   //*** Write record back to file ***
   fseek(myfile,0,SEEK_SET);
   fwrite(&fictionrecord,sizeof(DetailsType),1,myfile);
   //*** Close file ***
   fclose(myfile);
   cout << "First record's age updated\n";
}
```

Activity 15.15

Start a new project called *Ageing* and implement the code given in FIG-15.16.

Modify the program so that the age in each record is incremented by 1.

Check that the program has operated correctly by rerunning *DisplayRecords* to check the new contents of the file.

Moving to the Start of a File

When a file is first opened, the file position indicator points to the start of the data. After accessing the file data, we can return the indicator to the start of the file using the `fseek()` function, but another method of returning the indicator to the start of the data is to use the `rewind()` statement.



The term **rewind** comes from the days when data files were stored on magnetic tape rather than on disk or flash drives.

Function Prototype	: `void rewind(FILE* strmptr)`
Pre-condition	: *strmptr* must reference an open stream.
Description	: Returns the file position indicator to the start of the file data.

Finding the Value of the File Position Indicator

If you want to find the current position of the FPI, this can be done using the `ftell()` function.

Function Prototype	: `long int ftell(FILE* strmptr)`
Pre-condition	: *strmptr* must reference an open stream.
Description	: Returns the current value of the file position indicator. If an error occurs, -1 is returned.

When applied to a binary file, `ftell()` returns the FPI's position as the number of bytes from the start of the file.

For text files, the value returned is the number of characters from the start of the file. This is not always the number of bytes since some characters (such as the *newline* character in a Windows environment) can occupy more than one byte.

Activity 15.16

Make the following modifications to the *FindRecord* project:
the name search should not be case-sensitive
the matching record's position within the file should be displayed beside the record's contents. Assume the first record's position is 1, the next 2, etc.

Modify *FindRecord* again so that when the required record is found, the user is presented with an option to add 1 to the *age* field in that record. If the user chooses to update the field, the record's new contents should be read from the file and displayed.

File Management

As well as accessing file contents, we may need to perform various housekeeping activities such as renaming, or deleting files. Standard C++ defines a very limited set of commands for these tasks, but further options may be available for the compiler we are using.

Renaming a File

To rename an existing file, use the `rename()` function.

Function Prototype	: `int rename(const chr* oldfn, const chr* newfn)`
Pre-condition	: None.
Description	: Renames the file *oldfn* as *newfn*. If the *newfn* specifies a different folder from *oldfn* the file may be moved to the new folder (this depends on the implementation). If *newfn* already exists the function may fail or overwrite the existing file (depends on the implementation). Returns zero if successful, otherwise a non-zero value is returned.



Even when a file is not being moved to a new folder, any path details supplied in *oldfn* must be repeated in *newfn*.

Activity 15.17

Create a new project, *FileRename*, which renames the file *fictional.dat* as *dickens.dat*.

The program should display either "File renamed" or "File not found" as appropriate.

Deleting a File

To delete and existing file, use the `remove()` function.

Function Prototype	: `int remove(const chr* fn)`
Pre-condition	: None.
Description	: Deletes the file *fn*. Returns zero if successful, otherwise a non-zero value is returned.

Activity 15.18

Create a new project, *FileDelete*, which deletes the file *dickens.dat*.

The program should display either "File deleted" or "File not found" as appropriate.

Printer Output

There are no specific commands in C++ (or C) for outputting to a printer. How this is achieved depends on the operating system and compiler being used, so you will have to read the relevant documentation for your set up.

Microsoft's Visual C++ does not offer any way to perform printer output without making use of specific extensions to the C++ language.

Summary

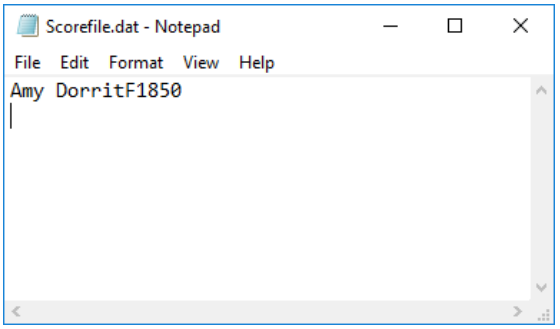
- Data arriving from or sent to a peripheral device or file is treated as a sequence of bytes known as a stream.
- C++ allows streams to be accessed using methods inherited from C (as well as newer C++ methods which are covered in a later chapter).
- Files can be stored in either text or binary format.
- A text file converts all data to text characters.
- A binary file stores data in the same format as used in program variables.
- Stream pointers are of type FILE*.
- Use `fopen()` to associate a stream pointer with an actual file.

- Use `fclose()` to close a stream when processing is complete.
- Use `fprintf()` to write data to a text file.
- Other options when writing to a text file are `fputs()` and `fputc()`.
- In order to identify the separate data values in a text file, a *newline* character is normally inserted after each value.
- Use `fscanf()` when reading values from a text file.
- When a string within a text file contains whitespaces, use `fgets_s()` to read the string.
- Use `fgetc()` when reading from a text file one byte at a time.
- In Microsoft Windows, a byte containing the value $1A_{16}$ is taken as an end-of-file marker.
- Use `fwrite()` to write data to a binary file.
- When writing records to a binary file, every record will contain exactly the same number of bytes.
- Using `fwrite()` several records (or data blocks) can be written to the file in a single statement.
- Use `fread()` to read data from a binary file.
- The value returned by `fread()` can be used to determine if the end of file has been reached.
- Use `feof()` as a double check that the end-of-file has been reached (rather than corrupt data causing `fread()` to fail).
- Use `fseek()` to move to a specific point in a binary file.
- `fseek()` allows positioning to be measured from the start of the file, current position in the file, or the end of the file.
- When updating a file record, remember to reposition the **File Position Indicator** (FPI) at the start of the record's position before writing the new version of the record back to the file.
- Use `ftell()` to discover the FPI's current value.
- Use `rename()` to rename a file.
- Use `remove()` to delete a file.
- There are no specific commands to allow printer output although each implementation will normally have their own extensions to allow this.

Solutions

Activity 15.1

Notepad should show the following:

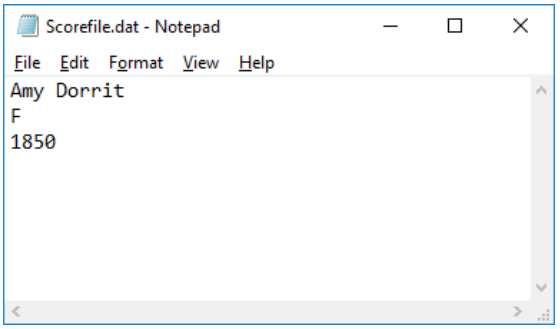


Activity 15.2

The line used to write the details to the file should be changed to

```
fprintf(myfile, "%s\n%c\n%d\n","Amy Dorrit",'F'
↳,1850);
```

When the contents of the file are displayed, although the newline characters are not visible, their effect can be seen since each data element now occupies a separate line (see below).



Activity 15.3

Code for *WriteRecord*:

```
#include <iostream>
using namespace std;

struct BestScores
{
  char name[31];
  int scores[5];
};

int main()
{
  //*** Record to be written to file ***
  BestScores player { "Arthur Clennam", { 219, 208,
  ↳185, 170, 162 } };

  FILE *myfile;  //Stream pointer

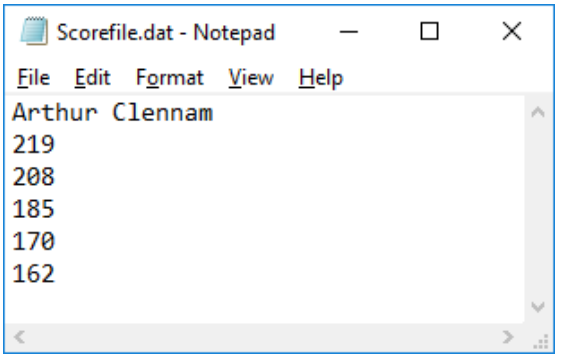
  //*** Open the file for writing ***
  myfile = fopen("C:\\C++MyPrograms\\Scorefile.dat",
  ↳"wt");
  // *** If there's a problem, exit program ***
  if (myfile == NULL)
  {
    cout << "Could not open file\n";
    exit(0);
  }

  //*** Write data to file ***
  fprintf(myfile, "%s\n", player.name);
  for (int c = 0; c <= 4; c++)
    fprintf(myfile, "%d\n", player.scores[c]);

  //*** Close file ***
  fclose(myfile);

  //*** Tell user program is complete ***
  cout << "Data written to file\n";
}
```

File contents:



Activity 15.4

No solution required.

Activity 15.5

Code for *ReadFromFile2*:

```
#include <iostream>
using namespace std;

struct BestScores
{
  char name[31];
  int scores[5];
};

int main()
{
  FILE *myfile;  //Stream pointer

  //*** Open the file for reading ***
  myfile = fopen("C:\\C++MyPrograms\\Scorefile.dat",
  ↳"rt");
  // *** If there's a problem, exit program ***
  if (myfile == NULL)
  {
    cout << "Could not open file\n";
    exit(0);
  }

  //*** Read data from file ***
  BestScores player; //Record to hold data from file
  fgets(player.name,30,myfile);
  for (int c = 0; c <= 4; c++)
    fscanf(myfile, "%d", &player.scores[c]);

  //*** Close file ***
  fclose(myfile);

  //*** Delete newline character from name ***
  int size = strlen(player.name);
  if (size > 0 && player.name[size - 1] == '\n')
    player.name[size - 1] = '\0';

  //*** Display data read ***
  cout << "Data read: ";
  cout << player.name << "  ";
  for (int c = 0; c <= 4; c++)
    cout << player.scores[c] << "  ";
  cout << endl;
```

```
  //*** Tell user display is complete ***
  cout << "End of data read\n";
}
```

Activity 15.6

Code for *ReadCharsFromFile*:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  FILE *myfile;  //Stream pointer

  //*** Open the file for reading ***
  myfile = fopen("C:\\C++MyPrograms\\test.dat",
  ↳"rt");
  // *** If there's a problem, exit program ***
  if (myfile == NULL)
  {
    cout << "Could not open file\n";
    exit(0);
  }

  //*** Read and display 15 characters from file ***
  char ch;  //Character read from file
  for (int c = 1; c <= 15; c++)
  {
    ch = fgetc(myfile);
    cout << setw(3) << static_cast<int>(ch) << "  "
    ↳<< ch << endl;
  }

  //*** Close file ***
  fclose(myfile);

  //*** Tell user display is complete ***
  cout << "15 characters displayed\n";
}
```

Activity 15.7

Modified code for *ReadCharsFromFile*:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  FILE *myfile;  //Stream pointer

  //*** Open the file for reading ***
  myfile = fopen("C:\\C++MyPrograms\\test.dat",
  ↳"rt");
  // *** If there's a problem, exit program ***
  if (myfile == NULL)
  {
    cout << "Could not open file\n";
    exit(0);
  }

  //*** Read all characters from file ***
  char ch;         //Character read from file
  int count = 0;   // Count of characters read
  //*** Read a character ***
  ch = fgetc(myfile);
  //*** WHILE not EOF DO ***
  while ( ch != EOF)
  {
    //*** Display numeric and character value ***
    cout << setw(3) << static_cast<int>(ch) << "  "
    ↳<< ch << endl;
    //*** Icrement count ***
    count++;
    //*** Get next character ***
    ch = fgetc(myfile);
  }

  //*** Close file ***
  fclose(myfile);

  //*** Tell user number of characters read ***
  cout << count << " characters displayed\n";
}
```

Activity 15.8

When *ReadCharsFromFile* is rerun, it halts after displaying the first three characters: *Amy*.

Activity 15.9

Code for *AddDataToFile*:

```
#include <iostream>
using namespace std;

int main()
{
  FILE *myfile;  // Stream pointer

  // *** Open file ***
  myfile = fopen("C:\\C++MyPrograms\\test.dat",
  ↳"at");

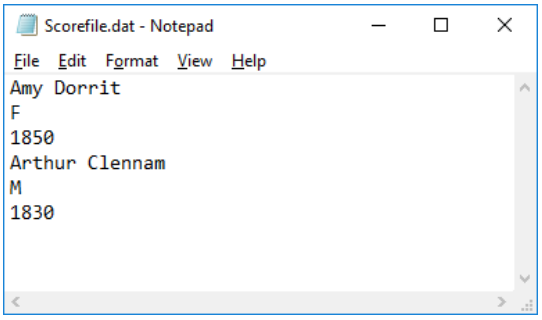
  // *** If there's a problem, exit program ***
  if (myfile == NULL)
  {
    cout << "Could not open file\n";
    exit(0);
  }

  //*** Add data to file ***
  fprintf(myfile, "%s\n%c\n%d\n", "Arthur Clennam",
  ↳'M', 1830);

  //*** Close the file ***
  fclose(myfile);

  //*** Tell user program is complete ***
  cout << "Data added to file\n";
}
```

The new contents of the file should display as shown below:

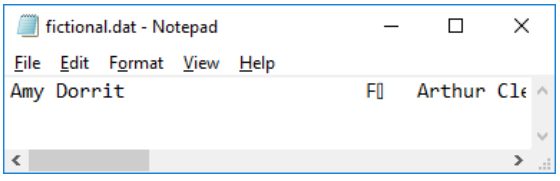


Activity 15.10

Although the names and sex fields of each record are readable, the age field displays apparently random characters. These characters are, in fact, the byte values of the stored ages being displayed as characters.

Also note that, since we are no longer writing newline characters at the end of each data item. all data appears on a single line.

Part of the file as it appears in Notepad is shown below:



Activity 15.11

The output from the program should be:

Amy Dorrit F 20

Arthur Clennam M 40

Jeremiah Flintwitch M 67

Pet Meagles F 18

Daniel Doyce M 36

Activity 15.12

Code for *DisplayRecords*:

```
#include <iostream>
#include <iomanip>
using namespace std;

//*** Define structure ***
struct DetailsType
{
  char name[31];
  char sex;
  int age;
};

int main()
{
  FILE *myfile;     //Stream pointer
  DetailsType record; //Holds record read from file

  //*** Open the file for reading ***
  myfile = fopen("C:\\MyC++Programs\\fictional.dat",
  ↳"rb");
  // *** If there's a problem, exit program ***
  if (myfile == NULL)
  {
    cout << "Could not open file\n";
    exit(0);
  }

  //*** WHILE record read from file DO ***
  while(fread(&record,sizeof(DetailsType),1,
  ↳myfile)==1)
    //*** Display record read ***
    cout<<record.name<<setw(21-strlen(record.name))
    ↳<<" "<< record.sex<<" "<<record.age<<endl;

  //*** IF not EOF, display error message ***
  if (!feof(myfile))
    cout << "Unexpected end of data\n";

  //*** Close file ***
  fclose(myfile);

  //*** Tell user number of characters read ***
  cout << "\nAll records displayed\n";
}
```

Activity 15.13

Code for *AddToBinaryFile*:

```
#include <iostream>
using namespace std;

//*** Define structure ***
struct DetailsType
{
  char name[31];
  char sex;
  int age;
};

int main()
{
  FILE *myfile;     //Stream pointer
  DetailsType record ={"Edmund Sparkler",'M',24 };
  ↳//Holds record to be added to file

  //*** Open the file for reading ***
  myfile = fopen("C:\\MyC++Programs\\fictional.dat",
  ↳"ab");
  // *** If there's a problem, exit program ***
  if (myfile == NULL)
  {
    cout << "Could not open file\n";
    exit(0);
  }

  //*** Write new record to file ***
  fwrite(&record, sizeof(DetailsType), 1, myfile);

  //*** Close file ***
  fclose(myfile);

  //*** Tell user record added ***
  cout << "New record added to file.\n";
}
```

Rerunning *DisplayRecords* should create the following output:

Amy Dorrit	F 20
Arthur Clennam	M 40
Jeremiah Flintwitch	M 67
Pet Meagles	F 18
Daniel Doyce	M 36
Edmund Sparkler	M 24

All records displayed

Activity 15.14

Code for *FindRecord*:

```
#include <iostream>
#include <iomanip>
using namespace std;

//*** Define structure ***
struct DetailsType
{
  char name[31];
  char sex;
  int age;
};

int main()
{
  FILE *myfile;     //Stream pointer
  DetailsType record; //Holds record read from file
  char name[21];   //Holds name being searched for

  //*** Open the file for reading ***
  myfile = fopen("C:\\MyC++Programs\\fictional.dat",
  ↳"rb");
  // *** If there's a problem, exit program ***
  if (myfile == NULL)
  {
    cout << "Could not open file\n";
    exit(0);
  }

  //*** Get name to be serached for ***
  cout << "Enter name required :";
  gets_s(name);

  //*** WHILE record read from file and names do not
  ↳match DO read next record ***
  while (fread(&record, sizeof(DetailsType), 1,
  ↳myfile) == 1 && strcmp(name, record.name) != 0);

  //*** Close file ***
  fclose(myfile);

  //*** IF match found THEN ***
  if (strcmp(name, record.name) == 0)
    //*** Display record read ***
    cout<<record.name<<setw(21-strlen(record.name))
    ↳<<" "<<record.sex<< " " << record.age << endl;
  else
    cout << "No match found\n";
}
```

Activity 15.15

Running the original version of *Ageing* should show that Amy Dorrit's age has changed to 21.

Modified code for *Ageing*:

```
#include <iostream>
#include <conio.h>
using namespace std;

//*** Define structure ***
struct DetailsType
{
  char name[31];
  char sex;
  int age;
};

int main()
{
  DetailsType fictionrecord; // Holds one record
  FILE *myfile;          // File pointer

  //*** Open binary file for updating ***
  myfile = fopen("C:\\MyC++Programs\\fictional.dat",
  ↳"r+b");
  // *** If there's a problem, exit program ***
  if (myfile == NULL)
  {
    cout << "Could not open file\n";
    exit(0);
  }
  //*** WHILE more records Do, Read one record ***
  while (fread(&fictionrecord, sizeof(DetailsType),
  ↳1, myfile) == 1)
  {
    //*** Update age field ***
    fictionrecord.age++;
    //*** Write record back to file ***
    fseek(myfile,-sizeof(DetailsType), SEEK_CUR);
    fwrite(&fictionrecord, sizeof(DetailsType), 1,
    ↳myfile);
    //*** Seek current position ***
    fseek(myfile, 0, SEEK_CUR);
  }
  //*** Close file ***
  fclose(myfile);

  //*** Tell user update complete ***
  cout << "All ages updated\n";
}
```

After running this version of *Ageing* once, the output from *DisplayRecords* is:

Amy Dorrit	F 22
Arthur Clennam	M 41
Jeremiah Flintwitch	M 68
Pet Meagles	F 19
Daniel Doyce	M 37
Edmund Sparkler	M 25

All records displayed

Activity 15.16

Modified code for *FindRecord*:

```
#include <iostream>
#include <iomanip>
using namespace std;

//*** Define structure ***
struct DetailsType
{
  char name[31];
  char sex;
  int age;
};

//*** Function prototypes ***
void ToLowerCase(char*);

int main()
{
  FILE *myfile;     //Stream pointer
  DetailsType record; //Holds record read from file
  char name[21];   //Holds name being searched for
  char lowername[21];     //Record name in lowercase

  //*** Open the file for reading ***
  myfile = fopen("C:\\MyC++Programs\\fictional.dat",
  ↳"rb");
  // *** If there's a problem, exit program ***
  if (myfile == NULL)
  {
    cout << "Could not open file\n";
    exit(0);
  }

  //*** Get name to be searched for ***
  cout << "Enter name required :";
  gets_s(name);
  //*** Convert name to lowercase ***
  ToLowerCase(name);

  //*** Read a record from the file ***
  int read=fread(&record,sizeof(DetailsType),1,
  ↳myfile);
  //*** WHILE record read from file DO ***
  while (read == 1)
  {
    //*** Make a copy of name in record ***
    strcpy_s(lowername,21,record.name);
    //*** Convert copy to lowercase ***
    ToLowerCase(lowername);
    //*** IF matching names, exit loop ***
    if (strcmp(lowername, name) == 0)
      break;
    //*** Read next record ***
    read = fread(&record, sizeof(DetailsType), 1,
    ↳myfile);
  }

  //*** IF match found THEN ***
  if (strcmp(lowername,name) == 0)
    //*** Display record read ***
    cout << "Record no. " << ftell(myfile)/
    ↳sizeof(DetailsType) << ":  "<<record.name <<
    ↳setw(21 - strlen(record.name)) << " " <<
    ↳record.sex << " " << record.age << endl;
  else
    cout << "No match found\n";

  //*** Close file ***
  fclose(myfile);
}

void ToLowerCase(char *s)
{
  for (int c = 0; c < strlen(s); c++)
    s[c] = tolower(s[c]);
}
```

The program makes use of the `ToLowerCase()` function developed in an earlier chapter.

Notice that significant changes were required to create a lowercase copy of the name held within the file record.

Also note that the file is now closed after the record is displayed. This allows us to use `ftell()` to discover the record's position when displaying the record's details.

Activity 15.17

Code for *FileRename*:

```
#include <iostream>
using namespace std;

int main()
{
  //*** Rename fictional.dat as dickens.dat ***
  int result = rename("C:\\MyC++Programs\\fictional.
  ↳dat", "C:\\MyC++Programs\\dickens.dat");
```

```
  //*** Tell user result of operation ***
  if (!result)
    cout << "File renamed\n";
  else
    cout << "File not found\n";
}
```

Notice that there is no need to repeat path information in the new filename string.

Activity 15.18

Code for *FileDelete*:

```
#include <iostream>
using namespace std;

int main()
{
  //*** Delete dickens.dat ***
  int result = remove("C:\\MyC++Programs\\dickens.
  ↳dat");
  //*** Tell user result of operation ***
  if (!result)
    cout << "File deleted\n";
  else
    cout << "File not found\n";
}
```

16

Starting Classes

In this Chapter:

- ❑ Identifying Classes and Objects
- ❑ Class Properties: Attributes and Operations
- ❑ Class Diagrams
- ❑ Creating Classes in C++
- ❑ Constructors and Destructors
- ❑ Encapsulation
- ❑ this
- ❑ Inline Methods
- ❑ Class Constants
- ❑ Static Attributes and Methods
- ❑ Overloading << and >>
- ❑ Friends
- ❑ Classes and Pointers
- ❑ Const Methods
- ❑ Classes and Arrays
- ❑ Designing Classes

Classes and Objects

Introduction

C++ is similar in many respects to its parent language C but the obvious difference is that C++ is an object-oriented language.

The idea behind object-orientation has been around for many decades now, but each language implements the concept in its own unique style. In this chapter we will discover the basic ideas behind object-oriented programming and how those ideas are implemented in C++.

What is an Object?

Real World Objects

Our lives are populated with objects. Typical examples include the digital camera, TV, cell phones, books, clouds, people, customers, etc.

Some objects, such as cameras, TVs and books have a physical existence; other objects represent roles or jobs (e.g. customer, shopkeeper) and a final group of objects are incidents or events such as a date, a traffic jam or joining a website.

One way to describe an object is to write down its characteristics and the operations that the object can perform. For example, we could describe a camera's characteristics by giving its weight, dimensions, lens type, and sensor size while its operations could be listed as take a picture, zoom, focus, set aperture, view stored images, adjust image size, delete recorded images, download images to another device, etc.

Non physical objects such as a bank account could be described in a similar way: with information such as account holder's name and address, account number, current balance and operations: list transactions, deposit money in account, withdraw money, set up a standing order.

Activity 16.1

List the physical characteristics of a beach ball and typical tasks performed on or with a beach ball.

Classes and Objects

It is important to differentiate between a general description of a group of identical objects and an individual object. For example, an architect may create a blueprint for a house and then a builder would use that design to build a physical house. In programming, the blueprint is known as a **class** while the specific item is known as an **object** (or **instance**) of that class. It's very similar to defining a `struct` and then creating a variable of that type.

Activity 16.2

Identify each of the following as either a class or an object:

a) Dogs b) Lassie c) Integers d) 27

An object-oriented approach to software design views a system as containing two main elements: a collection of objects and the relationships between these objects.

If we take the example of a User Interface (UI), the design will suggest a set of objects such as buttons, edit boxes, menus, etc. In other applications objects might include things like a customer's account, a purchase order, a data file, or a character in a video game.

Once the objects required by the system have been identified, the relevant characteristics and operations of the classes to which these objects belong are defined.

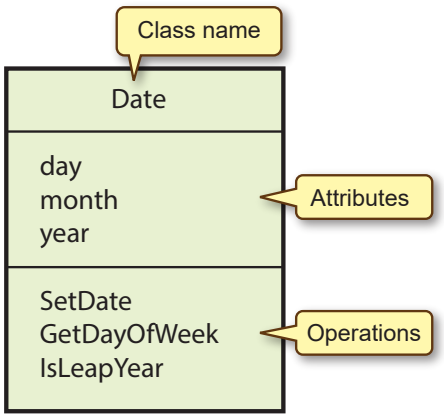
In the world of object-oriented design, a class's characteristics are known as its **attributes** while the procedures that can be performed are known as the **operations**. Collectively, the attributes and operations are known as the **features**, **properties**, or **members** of the class.

By far the easiest way to get the hang of all this is to look at a simple example. Back in Chapter 12 we created a set of functions for use with the *DateType* structure, so let's see how we could make use of this to create a class called *Date*.

Having identified our class and its properties, we can create a visual representation of this as shown in FIG-16.1.

FIG-16.1

A Basic Class Diagram



Note that for the sake of clarity, only a few of the functions we created in the earlier chapter are included above.

Activity 16.3

In the Imperial measuring system (still in use in the USA) short distances are measured in yards, feet and inches. There are 12 inches in a foot and 3 feet in a yard. So if we identified ImperialDistance *as a class within a system we were creating, we would list these three values (*yards*,* feet *and* inches*) as the attributes of that class. For the moment we can identify only two operations: one to set the value of the attributes (*SetDistance*) and one to convert the distance to metres (*ConvertToMetric*).*

Draw a class diagram similar to that in FIG-16.1 for a class called *ImperialDistance.*

Taking the design for *Date* further, each attribute must be defined as a C++ data type (or another class). In the case of the attributes *day*, *month* and *year*, all three of these can be defined as `int` types.

Next we need to add parameters and return types to the operations list. Since we are working exclusively in C++, we'll make use of the types available in that language. A more generic diagram might use other terms when describing attribute and operation types.

The updated class diagram is shown in FIG-16.2.

FIG-16.2

A Detailed Class Diagram



There are several things to notice in this version of the diagram:

- the type is given after the attribute or parameter name
- the type of value returned by an operation is given at the end of the operation (operations which do not return a value have no return type)
- the *DateType* parameter which was specified for each of these functions when we last used them in Chapter 12 has been removed.

It is this last point that needs to be explained in some detail.

Those of us of a certain age may remember cassette recorders. These devices performed record and play operations using magnetic tape cassettes. Just pop a tape into the machine and record or play. Replace the cassette with a new one and record or play other sounds. But without a cassette tape in place the device was useless. This is exactly how our *DateType* functions we created back in Chapter 12 worked: in order to use the functions, we had to supply a *DateType* variable (just as our recorder needed a cassette) that the function could operate on.

Jump forward to the 21st century and we now have digital recorders. Like the old cassette players, these can also record and play audio. But this time the storage is built into the device. All we do is select record or play and the internal storage is accessed automatically. This is the approach taken by classes. Classes are designed to contain both the storage requirements (attributes) and the operations that will make use of that storage. Combing these two elements into a single unit is known **encapsulation**.

This means that when an operation accesses *day*, *month* or *year*, it automatically assumes that this refers to the *day month* and *year* attributes held within that particular object. The attributes of a class are said to have **class scope**, meaning they can be freely accessed by any operation in the same class.

Activity 16.4

Add type details to your *ImperialDistance* class diagram.

A class diagram might also specifically state the range of values which an attribute may have. For example, *day* must always have a value in the range 1 to 31 while *month* must be between 1 and 12. These would be shown in the class diagram as:

```
day    : {1..31}
month  : {1..12}
year   : int
```

Activity 16.5

In your *ImperialDistance* class diagram, specify ranges for *inches* and *feet*.

The operations of a class can be specified using a set of mini-specs in a similar fashion to those we created in earlier chapters. This time we would also list any attributes of the class which are read from or written to by the operation. An example of an operation's mini-spec is given in FIG-16.3.

FIG-16.3

A Mini-Spec for the *SetDate* Operation

Class	: Date
Operation	: SetDate
Parameters	
In	: d : int
	m : int
	y : int
Out	: None
Attributes	
Read	: None
Written	: day, month, year
Pre-condition	: *d,m,y* forms a valid date
Post-condition	: *day = d*
	month = m
	year = y
Description	: Sets the date to *d/m/y*.

Activity 16.6

Create a mini-spec similar to that shown above for the *SetDistance* operation in the *ImperialDistance* class.

For some classes it is more appropriate to have a separate *Set* operation for each attribute, but when those attributes are closely linked such as those in a distance, a date, or a time object, then it is quite valid to have a single *Set* operation such as *SetDate* or *SetDistance* set the value of more than one attribute.

Classes in C++



Actually, classes can also be constructed using the term `struct`, but we'll ignore that option here.

Our design is now ready to be coded. In C++ a class's definition begins with the keyword `class`. We can think of this definition as a blueprint for later variables or objects. In this respect, a `class` definition is similar to that of a `struct`.

The keyword `class` is followed by the class name (e.g. *Date*, *ImperialDistance*). Although there are no fixed rules, generally class names start with a capital letter and if the name is constructed from several words, each new word starts with a capital. Underscores are not normally used in a class name.

After the class name, enclosed in braces, are an **access specifier**, the attributes and operations of the class. The operations are given in the form of function prototypes.

A first attempt at a definition of the *Date* class is given in below:

```
class Date
{
   public:
      int day;
      int month;
      int year;

      void SetDate(int,int,int);
      int GetDayOfWeek();
      bool IsLeapYear();
};
```

Let's look at the purpose of each section of the code:

`class Date`	This tells the compiler we are declaring a class and gives the name of the class. By convention, class names are capitalised.
`public:`	This keyword is the access specifier and tells the compiler that all the following properties of the class may be freely accessed by any function - even functions which are not part of the *Date* class.

`int day;`
`int month;`
`int year;`
	These are the attributes of the class and are similar to fields within a record.

`void SetDate(int,int,int);`
`int GetDayOfWeek();`
`bool IsLeapYear();`
	These are the prototypes for the various functions that will be used to implement the operations of the class.

Activity 16.7

Start a new project called *DateClass* (with `#include` and `using` statements) and implement the code for class *Date* as shown above. Save your code.

After the class declaration, we add the code for each of the class operations. Since the code for the class operations is not placed within the braces of the class declaration, it is necessary to tell the compiler to which class a function belongs. This is done in the first line of the function. For example, the first line of *SetDate()* is coded as

```
void Date::SetDate(int d, int m, int y)
```

Notice that this differs from a normal function heading only in that the class name and scope resolution operator (`Date` and `::`) precedes the function name. The complete code for the *Date* class is shown in FIG-16.4.

FIG-16.4

Implementing the Date Class

```
#include <iostream>
using namespace std;

class Date
{
```



FIG-16.4
(continued)

Implementing the Date Class

```
   public:
      int day;
      int month;
      int year;

      void SetDate(int, int, int);
      int GetDayOfWeek();
      bool IsLeapYear();
};


//*************************************
//***        Date Class Methods      ***
//*************************************

// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
   // *** Days in each month of year ***
   int daysinmonth[] = {0, 31,28,31, 30,31,30, 31,31,30, 31,30,31};
   //*** If month invalid, exit ***
   if (m < 1 || m > 12)
      return;
   //*** Add 1 to days in February if leap year ***
   daysinmonth[2] += (y%400 == 0)||(y%4 == 0 && y%100 != 0);
   //*** If days in month invalid, exit ***
   if (d <1 || d > daysinmonth[m])
      return;
   //*** If year less than 1, exit ***
   if (y < 1)
      return;
   //*** Assign date ***
   day = d;
   month = m;
   year = y;
}


// *** Returns the day of the week of a date ***
int Date::GetDayOfWeek()
{
   int M, modifiedyear, C, Y;
   //*** Calculate M ***
   M = (month + 9) % 12 + 1;
   //*** Calculate modified year ***
   modifiedyear = year - M / 11;
   //*** Calculate C ***
   C = modifiedyear / 100;
   //*** Calculate Y ***
   Y = modifiedyear % 100;
   //*** Calculate day of week ***
   int dayofweek = ((static_cast<int>(2.6 * M - 0.2) + day + Y +
   ↳Y/4 + C/4 - 2*C)%7+7)%7;
   return dayofweek;
}


// *** Returns true if the date's year is a leap year ***
bool Date::IsLeapYear()
{
   return ((year%400 == 0)||(year%4 == 0 && year%100 != 0));
}
```



We can't call *IsLeapYear()* to check for a leap year since that method checks the value stored in the *Date* attribute *year* whereas we wish to check the parameter *y*.

The coded version of a class operation is referred to as a **method**. Think of the code as the method by which we implemented the operation.

Activity 16.8

Modify your *DateClass* project to match the code given above. To get the code to compile, add the following line to the end of your program:

```
int main(){}
```

Check that your program compiles.



There are 2.54 cm in an inch.

Activity 16.9

Start a new project called *ImperialDistanceClass* and create code for the *ImperialDistance* class operations using the *Date* class as a guideline.

Compile and save your program.

In a large software project our job might be over at this point after we have created (and tested) a class. It would be other programmers who make use of objects of this class when writing their own section of the project. We can identify these two groups of programmers as the **class programmers** and the **application programmers**.

The application programmer uses objects from one or more classes, created by the class programmer, to implement the application design.

One goal of the class designer/programmer is to create a sufficient range of error-free methods within a class to allow the application programmer to easily achieve the results required by their project. For example, the present version of the *Date* class would be of little use to an application programmer since it does not allow for basic operations such as adding days to a date, finding out the number of days between two dates, or comparing two dates for equality.

A second goal of the class programmer is to have written a class in such a way that the application programmer cannot mishandle objects of that class. We'll see exactly what this means in a moment.

Normally, the application programmer will not have access to the source code for a class but will, instead, have a description of the public operations of that class. This is exactly the situation we met when using C++'s standard functions such as `sin()`, `cos()`, etc: we had no access to the original code, but nevertheless we have sufficient knowledge of the purpose, parameters and return value of each function to call the function appropriately.

If our mythical application programmer requires a *Date* class object, this is created like any other variable:

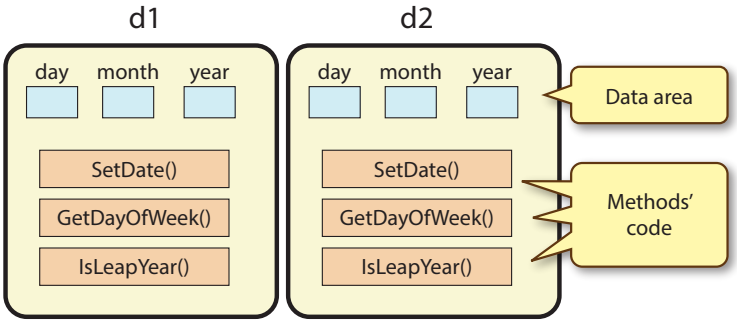
```
Date d1, d2;
```

d1 and *d2* are now objects of class *Date*.

Unlike a standard `struct` construct, the objects *d1* and *d2* contain not only data fields, but also their own copies of the routines defined for the class (see FIG-16.5).

FIG-16.5

A Visual Representation of Two *Date* Objects



Within the methods' code we will see that the properties *day*, *month* and *year* are accessed in exactly the way a standard function might access global variables (they have class scope). This is because the attributes of any class are always freely available to the methods of that class.

The methods in each object are designed to operate only with the attributes of that specific object. So, when we execute *SetDate()* in object *d1*, it is the contents of the *day*, *month* and *year* attributes in *d1* that will be modified.

The application programmer can access properties embedded in each object using the same notation as we did with record structures. So expressions such as

`d1.day` and `d2.month`

give access to a data element in each object while expressions such as

`d1.SetDate()` and `d2.GetDayOfWeek()`

give access to methods in each object.

The term

```
d1.GetDayOfWeek()
```

will return the day of the week of the date stored within object *d1*; the term

```
d2.GetDayOfWeek()
```

will return the day of the week of the date stored within object *d2*.

The program in FIG-16.6 demonstrates how the properties of an object are used to set a date and display which day of the week the date falls on.

FIG-16.6

Using a Date Object

```
#include <iostream>
using namespace std;

class Date
{
   public:
      int day;
      int month;
      int year;

      void SetDate(int, int, int);
      int GetDayOfWeek();
      bool IsLeapYear();
};
```

FIG-16.6
(continued)

Using a Date Object

```
//*************************************
//***        Date Class Methods       ***
//*************************************

// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
   // *** Days in each month of year ***
   int daysinmonth[] = {0, 31,28,31, 30,31,30, 31,31,30, 31,30,31};
   //*** If month invalid, exit ***
   if (m < 1 || m > 12)
      return;
   //*** Add 1 to days in February if leap year ***
   daysinmonth[2] += (y%400 == 0)||(y%4 == 0 && y%100 != 0);
   //*** If days in month invalid, exit ***
   if (d <1 || d > daysinmonth[m])
      return;
   //*** If year less than 1, exit ***
   if (y < 1)
      return;
   //*** Assign date ***
   day = d;
   month = m;
   year = y;
}

// *** Returns the day of the week of a date ***
int Date::GetDayOfWeek()
{
   int M, modifiedyear, C, Y;
   //*** Calculate M ***
   M = (month + 9) % 12 + 1;
   //*** Calculate modified year ***
   modifiedyear = year - M / 11;
   //*** Calculate C ***
   C = modifiedyear / 100;
   //*** Calculate Y ***
   Y = modifiedyear % 100;
   //*** Calculate day of week ***
   int dayofweek = ((static_cast<int>(2.6 * M - 0.2) + day + Y +
   ↳Y/4 + C/4 - 2*C)%7+7)%7;
   return dayofweek;
}

// *** Returns true if the date's year is a leap year ***
bool Date::IsLeapYear()
{
   return ((year%400 == 0)||(year%4 == 0 && year%100 != 0));
}


int main()
{
   // *** Day names ***
   char daynames[7][10] = {"Sunday","Monday","Tuesday",
   ↳"Wednesday","Thursday","Friday","Saturday"};

   Date d1; // Date object

   // *** Set date ***
   d1.SetDate(23,11,1963);
```



FIG-16.6
(continued)

Using a Date Object

```
    // *** Display date details ***
    cout << d1.day << '/' << d1.month << '/' << d1.year <<
    ↳" was a " << daynames[d1.GetDayOfWeek()] <<endl;
}
```

Activity 16.10

Modify *DateClass* to match the code in FIG-16.6 and observe the output produced.

Although we have a class that will operate successfully as long as the application programmer makes valid calls to the class methods, there are a few problems with this code.

The first problem is that should the application programmer attempt a call to the `SetDate()` method using invalid parameters, then the contents of the attributes *day*, *month* and *year* will remain undefined.

Activity 16.11

In *DateClass,* change the call to *SetDate()* so that the parameters are 29, 2, and 2021. How does this affect the display when the program is run?

Activity 16.12

Modify *ImperialDistanceClass* creating an object of the class, assigning it a valid distance and then displaying the equivalent distance in metres.

What happens if an invalid distance (e.g. 2,3,10) is assigned to the object?

Overloading Methods

We saw, when creating traditional functions back in Chapter 12 that we could overload functions. That is, we could create multiple functions with the same name as long as their parameter lists differed in some way. We can also do this with the methods of a class.

To demonstrate overloaded methods we'll create a second version of *SetDate()*. This new version takes two integer values. The first of these is the number of days into the year (1 to 365/366). The second parameter is the year. For example, 32,2021 would represent 1/2/2021 (32 days into the year 2021).

The method's logic creates a date of the form *d/m/y* by subtracting the days in each month until we end up with a zero or negative value and then adding back the number of days in the last month to have been deducted. The methods code is:

```
// *** Sets date using days-into-year (diy) and year (y) ***
void Date::SetDate(int diy, int y)
{
   // *** Days in each month of year ***
   int daysinmonth[] = { 0, 31,28,31, 30,31,30, 31,31,30, 31,
   ↳30,31 };
   //*** If leap year add one day to February ***
   daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0 &&
```

```
      ↳y % 100 != 0);
      //*** Calculate month ***
      int remaining = diy;
      int m = 0;
      while (remaining > 0)
      {
         m++;
         remaining -= daysinmonth[m];
      }
      remaining += daysinmonth[m];
      //*** Set date ***
      SetDate(remaining, m, y);
   }
```

Activity 16.13

In *DateClass,* add the second version of *SetDate()* and test this new method with the values 60,2020, and 60,2021.

Constructors and Destructors

Constructors

When an object is first created a special method known as a **constructor** is executed. A default constructor is automatically created by C++ for every class, but this default constructor is of little use since it contains no code! Luckily, we can replace the default constructor with a version of our own.

Typically, a coded constructor is used to initialise each of the attributes of an object to a specific value. This will solve the problem we encountered in the last two Activities.

A constructor method must be given the same name as the class and have no return type (not even `void`) but other than that it is prototyped and defined in the same way as any other method in a class.

A constructor for the *Date* class which sets *day*, *month* and *year* to 1/1/2001 would have the prototype

```
Date();
```



This is known as a **zero-argument constructor** since it requires no parameters.

and be coded as

```
Date::Date()
{
   day = 1;
   month = 1;
   year = 2001;
};
```

The coded constructor will automatically replace the default one created by the compiler.

Activity 16.14

Modify *DateClass,* so that the *Date* class contains the constructor code given above. What happens this time when you call *SetDate()* with parameters 29, 2, and 2021?

Activity 16.15

Modify *ImperialDistanceClass,* so that *ImperialDistance* class contains a constructor which sets the distance to zero.

What happens this time when you call *SetDistance()* with parameters are 2, 3, and 10?

Overloading Constructors

Like other methods of a class, the constructor can be overloaded. For example, we could create a second *Date* class constructor which takes three `int` parameters and assigns to the *Date* object being created.

This would require the prototype (in the class code)

```
Date(int,int,int);
```

and the code

```
Date::Date(int d, int m, int y)
{
   // *** Days in each month of year ***
   int daysinmonth[] = {0, 31,28,31, 30,31,30, 31,31,30,
   ↳31,30,31};

   bool valid = true;  // Parameters valid

   //*** If month invalid, invalid parameters ***
   if (m < 1 || m > 12)
      valid = false;
   //*** Add 1 to days in February if leap year ***
   daysinmonth[2] += (y%400 == 0)||(y%4 == 0 && y%100 != 0);
   //*** If days in month invalid, invalid parameters ***
   if (d <1 || d > daysinmonth[m])
      valid = false;
   //*** If year less than 1, invalid parameters ***
   if (y < 1)
      valid = false;
   //*** Assign date ***
   if (valid)
   {
      day = d;
      month = m;
      year = y;
   }
   else
   {
      day = 1;
      month = 1;
      year = 2001;
   }
}
```

As we can see, the code is similar to that for the first version of *SetDate()*. However, rather than exit from the routine when invalid parameters are offered, the method sets the date attributes to 1/1/2001 rather than leaving them undefined.

In `main()` we could make use of the new constructor when declaring *d1* as a *Date*

object with the line:

```
Date d1(23,11,1963);
```

When a *Date* object is declared in `main()`, with the line

```
Date d1;
```

this will still cause the zero-argument constructor to be executed and the date set to 1/1/2001.

Activity 16.16

Modify *DateClass,* adding the second constructor given above to the *Date* class.

Test the new constructor by initialising *d1* to 23/11/1963 and checking that the appropriate value is assigned.

We can even make use of default values for the parameters to a constructor (and other methods in a class). For example, if we rewrote the prototype for the second *Date* constructor to read

```
Date(int=1,int=1,int=2001);
```

we end up removing the need for a coded zero-argument constructor since this will be performed automatically:

```
Date d1;          // Uses default values to set d1 to 1/1/2001
Date d2(23,11,1963); // Overrides default values to set d2 to
↳23/11/1963
```

Activity 16.17

Modify *DateClass,* so that the zero-argument constructor can be removed.

Check that the default value works to create a *Date* object set to 1/1/2001.

Activity 16.18

Modify *ImperialDistanceClass,* adding a three-argument constructor (`int, int, int`) with default values 0,0,0.

Test the constructor, checking that the appropriate value is assigned to any *ImperialDistance* object created.

The Copy Constructor

C++ creates not one but two default constructors. We've already seen that the first default constructor does not perform any task, but the second constructor – known as the **copy constructor** – allows a new object to be initialised with a copy of an existing object of the same type.

For example, if we have previously created a *Date* object with the line

```
Date d1(1,1,2001);
```

then we can create a new *Date* object containing a copy of the first date using one of the following lines

```
Date d2{d1};
Date d2(d1);
Date d2 = d1;
```

Each of these options makes use of the copy constructor to copy the values held in the attributes of *d1* into the corresponding attributes of *d2*.

The copy constructor's code for the *Date* class is equivalent to:

```
Date::Date(const Date& d)
{
   day = d.day;
   month = d.month;
   year = d.year;
};
```

Note that the parameter to the copy constructor is a `const` reference. This is a requirement of all copy constructors. For example, if we were to write a copy constructor for the *ImperialDistance* class, its first line would be

```
ImperialDistance::ImperialDistance(const ImperialDistance& d)
```

As with the zero-argument constructor, we are free to override the copy constructor but for simple classes, where none of the attributes are pointers, this is usually unnecessary.

Activity 16.19

In *DateClass*, check out the copy constructor of the *Date* class by first creating a *Date* object, *d1*, set to 23/1/2001 and then creating a new *Date* object, *d2*, which is initialised with a copy of the values in *d1*. The value of *d2* should be displayed.

The Class Assignment Operator

Although not a constructor, this is an appropriate point to mention the other piece of code that C++ generates for free when we create a class - the assignment operator.

If we have two class objects, then we can copy the contents from one object to the other using the standard assignment operator:



Here we are assuming *d1* and *d2* are *Date* objects.

```
d1 = d2;
```

This default assignment operator, like the default copy constructor, copies the contents of every attribute (in this case, *day*, *month* and *year*) from the right-hand object (*d2*) to the corresponding fields in the left-hand object (*d1*).

For regular type attributes this logic will normally be fine, but when any of the attributes involved are pointers, then we may want to override this version of the assignment operator with our own code. We'll see an example of this later.

Note that in the following example, the second line will use the copy constructor while the third makes use of the assignment operator:

```
Date d1(23,11,1963), d2;
Date d3(d1);      //Copy constructor
d2 = d1;          //Assignment operator
```

Even if we think we are using the assignment operator as in the second line below

```
Date d1(26,10,1961);
Date d2 = d1;
```

we're not! Any assignment to an object in the line in which that object is declared always makes use of a constructor.

Typecasting and Constructors

As we saw in an earlier chapter, C++ can automatically convert values from one type to another. So the line

```
double x = 21;
```

will convert the `int` value 21 to a `double` before storing that value in *x*.

The same automatic conversion is possible with class objects. For example, if we were to write

```
Date d1;
d1 = 21;
```

then the `int` value 21 would automatically be converted to a date.

Activity 16.20

In *DateClass*, modify *main()* so that *d2* is assigned the value 21 after previously having been created holding a copy of *d1*. What value is displayed?

The reason for the result obtained by Activity 16.20 is that when presented with a value which is not of the same class as the object being assigned the value, C++ examines the constructors available for that class to see if any of these can accept the value being assigned as a valid parameter. If it can, then that constructor is executed and the *Date* object created is used in the assignment. In the case of the statement above, the *Date* class has a constructor whose prototype is

```
Date(int=1, int=1, int=2001);
```

Since the parameters have default values, when presented with a single value, 21, C++ uses this as the first parameter to the constructor and uses the default values (1 and 2001) for the remaining two parameters.

We can also use a class constructor explicitly along with the class assignment operator to assign a value to an object. Hence,

```
Date d1;
d1 = Date(23,11,1963);
```

creates an unnamed *Date* object and assigns its value to *d1*.

explicit

The fact that C++ will make use of a constructor to automatically convert a value to an object of a class is not always desirable. For example, it's a bit of a stretch to think that a program converting the value 21 to the date 21/1/2001 would be acceptable.

To control this sort of situation, we can tell our constructor not to indulge in these automatic conversions by adding the keyword `explicit` at the start of a constructor's prototype. In the case of the *Date* class constructor, this would mean that the prototype

would now read

```
explicit Date(int = 1, int = 1, int = 2001);
```

There is no need to make any changes to the constructor's definition.

Now, if we were to attempt the line

```
d2 = 21; //d2 is a Date object
```

the compiler would throw up an error. The only way to have this line accepted is to explicitly cast 21 to a *Date* object:

```
d2 = static_cast<Date>(21);
```

With this explicit request to convert 21 to a *Date* object, the compiler will again make use of the constructor to perform the conversion from `int` to *Date*.

Activity 16.21

Modify *DateClass,* to create an explicit constructor. What error do you get when you attempt to convert 21 automatically to a date?

Change the assignment line to explicitly convert 21 to a *Date* object. What happens when you compile and run the program this time?

Destructors

A **destructor** is a function whose code is automatically executed when an object has its space deallocated. Again, C++ automatically creates a class's destructor, but we can override this by creating our own.

A class's destructor is named after the class to which it belongs. However, to distinguish it from the constructor, the name must begin with a tilde (~) character. A destructor cannot return a value nor take any parameters.

We are only likely to need a destructor when a class contains a raw pointer attribute which references dynamically allocated space (smart pointers will handle the release automatically). Other reasons a class may require a destructor is when it needs to release some resource such as a file, or internet connection.

In these situations the destructor's code would free up the space referenced by the raw pointer, close the file, or disconnect from the internet. We'll see an example of a class which makes use of dynamic space in the next chapter.

For the moment, let's create a trivial destructor for the *Date* class which simply displays a "*Date object deleted*" message.

This requires the class to include the following prototype

```
~Date();
```

and the code

```
Date::~Date()
{
   cout << "Date object deleted\n";
};
```

Activity 16.22

Modify *DateClass,* so that the *Date* class includes the destructor coded above.

Run the program. Is the destructor message displayed?

After testing, remove the destructor's code from the *Date* class.

Data Hiding

A problem with the *Date* class is that the application programmer is free to bypass the `SetDate()` method entirely and just set the values held in attributes *day*, *month* and *year* directly with code such as

```
Date d1;
d1.day = 2
d1.month = 1;
d1.year = 1950;
```

If they use this approach, then there are no safeguards to stop an invalid value being stored. For example the line

```
d1.month = 13;
```

will execute without a complaint and any other methods which assume the date is valid will return unpredictable results.

To overcome this problem we can restrict the application programmer's access to some of the properties in a class. For example, by stopping access to *day*, *month* and *year*, we remove any chance of invalid values being stored within those attributes.

Hiding properties in this way is known as **data hiding**.

private

The application programmer has direct access to the attributes *day, month* and *year* in any *Date* objects he creates because those attributes are marked as `public` in the class definition. However, properties can also be labelled as `private`. Private properties in objects created by the application programmer's code cannot be accessed directly but may still be accessed via the methods of the class.

So to implement this change in *Date* we re-code the class declaration as

```
class Date
{
   private:
      int day;
      int month;
      int year;
   public:
      Date(int=1, int=1, int=2001);
      void SetDate(int,int,int);
      void SetDate(int,int);
      int GetDayOfWeek();
      bool IsLeapYear();
};
```

With this new version of the class, the application programmer can no longer gain

direct access to the three attributes of the class. Attempts to use statements such as

```
d1.month = 2;
```

will now fail to compile. If an object's attributes are to have their contents changed, then the `SetDate()` method must be employed:

```
d1.SetDate(23,11,1963);
```

Activity 16.23

Modify *DateClass,* so that the attributes of the class are private. What happens when you try to compile the program?

And now the side-effect of marking attributes as `private` becomes apparent. Not only have we stopped the application programmer from storing values directly in *day*, *month* and *year*, but we have also stopped him from discovering what values are already stored there! `private` means absolutely no access - even if we only want to look at the value held.

To get round this we need to add new functions which do nothing more than return copies of the values in the class's attributes. For example, if we want to find out what value is stored in the *days* attribute we would write a new method as shown below:

```
int Date::GetDay()
{
   return day;
}
```

With this code (and the method prototype) embedded in the class, the application programmer, who has discovered that

```
cout << d1.day;
```

causes a compilation error, can now write

```
cout << d1.GetDay();
```

Activity 16.24

Modify *DateClass,* adding new methods *GetDay()*, *GetMonth()* and *GetYear()* to the class.

Uses these new methods in `main()` to allow the contents of the *Date* object *d2* to be displayed.

It is not only attributes that can be labelled as private – methods can be too. It is only appropriate to make helper functions private. A helper function is designed only to help in the coding of some other method and not as a function we wish the application programmer to gain access to. To stop that access, the helper function should be marked as private. Going back to the functions we created for *DateType* in Chapter 11, *DateToJDN()* and *JDNToDate()* are good examples of helper functions since they were only needed so that the functions *AddDays()* and *DaysDifference()* could be coded.

We will start by adding these two helper functions to our *Date* class - but this time giving them slightly different names. The class declaration now reads:

```
class Date
{
```

```
   private:
      int day;
      int month;
      int year;

      long ToJDN();
      Date FromJDN(long);

   public:
      Date(int=1,int=1,int=2001);
      void SetDate(int, int, int);
      void SetDate(int,int);
      int DayOfWeek();
      bool IsLeapYear();
      int GetDay();
      int GetMonth();
      int GetYear();
};
```

And the new methods are coded as:

```
// *** Returns the JDN of the date ***
long Date::ToJDN()
{
   int a = (14 - month)/12;
   int m = month + 12*a - 3;
   int y = year + 4800 - a;
   long result = day + (153*m + 2) / 5 + 365*y + y/4 - y/100 +
   ↳y/400  - 32045;
   return result;
}

// *** Sets date to eqivalent of JDN ***
Date Date::FromJDN(long jdn)
{
   int f = jdn + 1401+(((4 * jdn + 274277)/146097)*3)/4 -38;
   int e = 4 * f + 3;
   int g = (e % 1461)/4;
   int h = 5 * g + 2;
   Date d;
   d.day = (h % 153)/5 + 1;
   d.month = (h/153 + 2) % 12 + 1;
   d.year = e/1461 - 4716 + (12 + 2 - d.month)/12;
   return d;
}
```

Notice that this second method returns a *Date* object. This is no different in implementation than our previous version of the function, *JDNToDate(),* which returned a *DateType* structure.

Activity 16.25

Add the two private functions *ToJDN()* and *FromJDN()* to *Date* class. Check that your code compiles.

Overloading Operators

All the things we could achieve with standard functions can also be done within a class. This includes overloading operators. Implementing `operator+` in the *Date* class would require the following code:

```
//*** Add d days to date ***
Date Date::operator+(int d)
{
   Date dt = FromJDN(ToJDN()+d);
   return dt;
}
```

Notice that it makes use of the two private functions in calculating the result.

As before, we have two ways of using an overloaded operator. We can treat it as just a standard method allowing the format

```
Date dt2 = dt1.operator+(7); //Add one week to date
```

or we can treat it in the same way as standard operators when dealing with normal numeric values:

```
Date dt2 = dt1 + 7; //Add one week to date
```

Activity 16.26

Add `operator+` as a public method to *Date* class. To check that your code is correct, create a new version of `main()` which sets a *Date* object *d1* to 30/5/2023 then adds 7 days to the date and displays the result.

Activity 16.27

Add the following operators to the *Date* class:

- `-` (subtraction: returns days between two dates)
- `-` (subtraction: returns a date after subtracting a number of days)
- `==` (test for equality: returns *true* or *false*).
- `!=` (test for inequality: returns true or false).

Modify `main()` to test each operation.

The Call Operator and Functors

The opening and closing parentheses (known as the **call operator** since it used whenever we call a function) can also be overloaded. The code associated with the call operator is executed when parentheses are given immediately after an existing object. We'll demonstrate this by creating a simple class A with a single feature which overloads the call operator. (see FIG-16.7).

FIG-16.7

The Call Operator

```
#include <iostream>
using namespace std;

class A
{
   public:
      void operator()();
};
void A::operator()()
{
   cout << "Call operator executed\n";
}
int main()
{
   A test;     //Create object test
   test();     //Execute the call operator
}
```

Notice that the object, *test*, begins to look a lot like a function. If we saw the program's

```
test();
```

line in isolation from the remainder of the code, we would, quite reasonably, think *test* was the name of a function rather than an object. Because of this, any class that implements a call operator method is classified as a **functor** – a class whose objects act like functions.

Like any method, the call operator can accept parameters. For example, we could change the code for class A's call operator to

```
void A::operator()(int v)
{
   cout <<"Call operator executed. Parameter value "<<v<< endl;
}
```

Activity 16.28

Start a new project called *TestCallOperator* and implement the code given in FIG-16.7.

Modify the call operator code to accept an `int` parameter and include the value of the parameter in the `cout` statement as shown above.

Modify the call parameter again to accept two parameters and to return the product of the two values. Also modify the code in `main()` to display the returned value.

Although this represents a trivial example of a functor, we'll discover in a later chapter how widely used this option is.

Inline Methods



Defining a function as inline only acts as a request to the compiler. The final decision on whether to do so is left up to the compiler.

In Chapter 12 we learned that inline functions can be used to create a more efficient program by replacing the call to a function with the function's code thereby saving the overhead of parameter passing and saving a return address. A class method can be defined as **inline** in one of two ways. The first option is to use the `inline` keyword in the method definition, the second is simply to embed the method's code within the class declaration. For example, if we were to make *IsLeapYear()* and inline function we could write

```
inline bool IsLeapYear()
{
   return ((year%400 == 0)||(year%4 == 0 && year%100 != 0));
}
```

or we could move the code for the function into the class:

```
class Date
{
   private:
      int day;
      int month;
      int year;
      long ToJDN();
      Date FromJDN(long);
   public:
      Date(int=1,int=1,int=2001);
      void SetDate(int, int, int);
```

```
        void SetDate(int, int);
        int DayOfWeek();
        bool IsLeapYear()
        {
           return ((year%400 == 0)||(year%4 == 0 &&
           ↳year%100 != 0));
        }
        int GetDay();
        int GetMonth();
        int GetYear();
        Date operator+(int);
        int operator-(Date);
        Date operator-(int);
        bool operator==(Date);
        bool operator !=(Date);
        void operator+=(int);
};
```

The *this* Pointer

Returning to the *Date* class, another operator we could overload is `+=` allowing us to add a number of days to the current date. For example, with this operator overloaded the application programmer could write lines such as

```
Date d1(23,11,1963);
d1 += 10;
```

However, when we come to code such a routine for our class, we run into an unexpected problem:

```
void Date::operator+=(int d)
{
    ??? = FromJDN(ToJDN()+d);
}
```

We want the value calculated by the method to be assigned to the *Date* object which has called this method, but how do we refer to that object since the actual object executing the routine can be different each time the operation is used? Luckily, C++ solves this problem by maintaining a pointer to whatever object is currently executing a method of a class.

The pointer is called `this` and is available for use within every standard method of a class. So, when we are writing a method within a class and wish to refer to the object which is currently executing that method, we can dereference the `this` pointer. This allows us to overcome our problem in writing the code for the `+=` operator since it can now be coded as:

```
void Date::operator+=(int d)
{
    *this = FromJDN(ToJDN()+d);
}
```

Activity 16.29

Add `operator+=` as a public method to *Date* class.

To check that the method operates correctly set a *Date* object to 23/12/1980 and add 10 days to the date, displaying the new date.

In fact, C++ makes extensive use of the `this` pointer, secretly adding it to any code we write when referencing attributes of a class within one of its methods. For example, when we write

```
bool Date::IsLeapYear()
{
   return ((year%400 == 0)||(year%4 == 0 && year%100 != 0));
}
```

The C++ compiler secretly changes this to:

```
bool Date::IsLeapYear(Date *this)
{
   return ((this->year%400 == 0)||(this->year%4 == 0 &&
   ↳this->year%100 != 0));
}
```

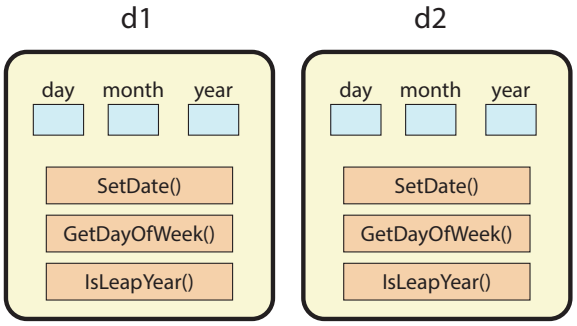
This is necessary because, although we may imagine each object having its own copy of every member defined for that class, the reality is that, in order to optimise storage requirements, each object has its own copy of every attribute but shares use of a communal copy of the code for the class's methods with every other object of that class.

So, when executing a method, some mechanism is needed to make sure the attributes of the appropriate object are accessed. This requirement is fulfilled by the `this` pointer. (see FIG-16.8).

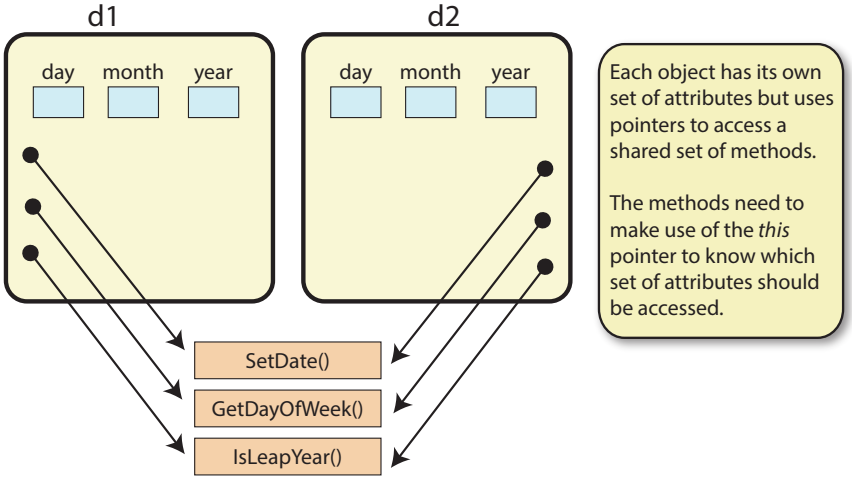
FIG-16.8

Why the *this* Pointer is Required when Executing Methods

Conceptual Model



Reality Model



And although C++ automatically adds the `this->` prefix to any attribute accessed within a class's methods, we may also add this prefix explicitly without it objecting.

Activity 16.30

Modify the *IsLeapYear()* method in the *Date* class so that it explicitly uses `this ->` when accessing attributes of the class.

Check that this has no effect on the program's compilation or execution.

Return the code for *IsLeapYear()* to its original version.

We need to make use of the `this` pointer again if we want to overload the `++` and `--` operators for our *Date* class. For example, the pre-fix version of `++` would be coded as

```
Date Date::operator++()
{
   *this = FromJDN(ToJDN()+1);
   return *this;
}
```

while the post-fix version would be written as

```
Date Date::operator++(int)
{
   DateType result = *this;
   *this = FromJDN(ToJDN()+1);
   return result;
}
```

Activity 16.31

Add both versions of `++` and `--` as public methods in *Date* class. Check that all four new methods operate correctly.

Class Constants

A class declaration can contain named constants. For example, we might begin our *Date* class declaration by adding a constant for the number of days in a standard year:

```
class Date
{
   private:
      const int DAYSINYEAR = 365;
```

Activity 16.32

Add the DAYSINYEAR constant given above to the *Date* class and compile your program. What error messages are produced?

Although the compiler has not objected to the new line of code we've added, it suddenly objects to all the assignment statements that attempt to copy one *Date* value to another.

When we copy the contents of one *Date* object to another as in the line

```
Date dt1(23,11,1963),dt2;
dt2 = dt1;
```

every attribute of *dt1* is copied to the corresponding attribute of *dt2*. However, when a class contains a constant value, this approach is no longer appropriate since we cannot attempt to assign a constant a value (even if it is the same value as it already contains).

Because of this, we are forced to define a replacement version of the `=` operator within the *Date* class so that no attempt to copy the `const` value is included. Such an operation has the prototype

```
Date& operator=(const Date&);
```

and is coded as

```
//*** Copies d to current object ***
Date& Date::operator=(const Date& d)
{
   day = d.day;
   month = d.month;
   year = d.year;
   return *this;
};
```

Notice that the method is designed to return a reference to the object being assigned the new value. Although not necessary for simple statements such as

```
dt1 = dt2;
```

the return value is required for more complex statements such as

```
dt1 = dt2 = dt3;
```

Activity 16.32

Add the necessary code for the = operator to the *Date* class. Does the program compile?

Add another method to the *Date* class called *GetDaysInYear()* which returns the number of days in the year (using the value held in *year* to determine if it's a leap year). The method should make use of the DAYSINYEAR constant.

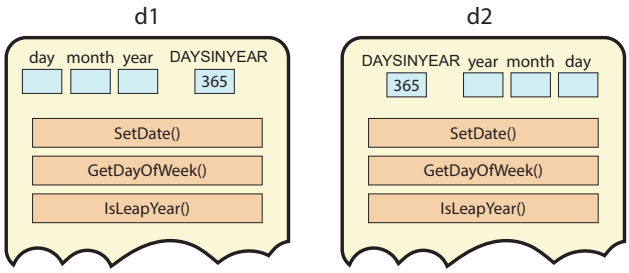
Modify `main()` to display the number of days in the year of the date within object *d1*.

Static Properties

When a standard `const` value is defined within a class, every object of that class acquires its own copy of that `const` (see FIG-16.9).

FIG-16.9

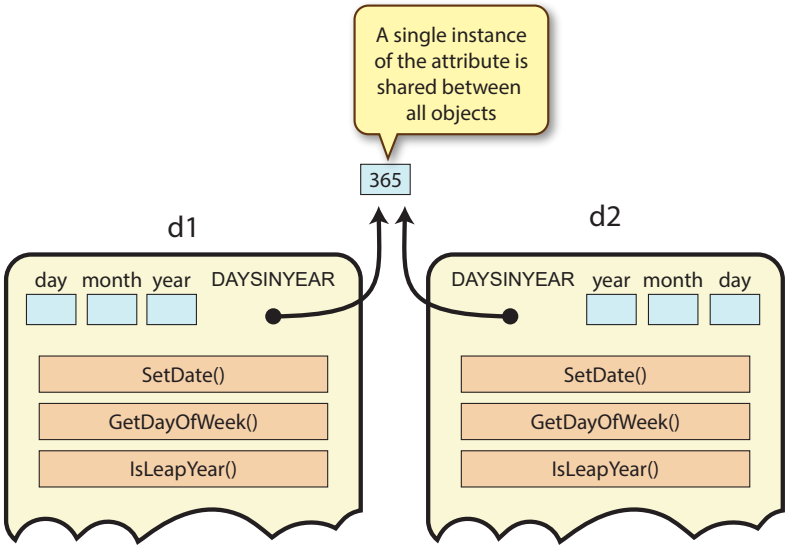
How a Class `const` Value is Stored



This situation is rather inefficient and also unnecessary. We've already seen how objects share the same method code for reasons of efficiency. We can do the same thing with constant values declared within a class, creating only a single instance of that constant and sharing it between all objects of that class (see FIG-16.10).

FIG-16.10

How a Class **static const** Value is Stored



To achieve this effect we must declare the constant as a **static const** as in the line

```
class Date
{
   private:
      static const int DAYSINYEAR = 365;
```

Another advantage of using a static constant is that it removes the need to redefine the = operator for the class since the default assignment operator code has no effect on static attributes.

If we change the declaration of the *DAYSINYEAR* attribute making it public, and then create two *Date* class objects, we could access the attribute in the usual fashion:

```
Date d1, d2;
cout << d1.DAYSINYEAR << endl;
cout << d2.DAYSINYEAR << endl;
```

Static properties of a class have another strange property: they come into existence the moment a program begins execution. This means they can exist before any objects of that class are declared:

```
int main()
{
   // DAYSINYEAR exists now
   cout << "Hello world\n";
   int no = 12;
   Date d1; //Date object exists now
      ...
```

To allow the application programmer to access the static access before creating a *Date* object, the following syntax can be used:

```
class_name::static_value_name
```

As we can see, the class name is used rather than an object name and the scope resolution operator replaces the normal member selector. So, if we wanted to access DAYSINYEAR before *Date* objects *d1* or *d2* are created, we would use the expression

```
Date::DAYSINYEAR
```

Activity 16.33

Change *Date* class's DAYSINYEAR to a public static constant then remove the prototype and definition of the `=` operator from the *Date* class.

Write a `main()` function which first displays the value held in DAYSINYEAR using the term *Date::DAYSINYEAR* and then declares a *Date* object called *d1* before displaying the value in *DAYSINYEAR* for a second time using the terms *d1.DAYSINYEAR*.

If we were to return the DAYSINYEAR class constant to private status, the application programmer could only gain access to its constants if a public *get* method is included to retrieve the constant's value.

Activity 16.34

Change *Date* class's *DAYSINYEAR* to a private static constant.

Add a public method called *GetDAYSINYEARconst()* which returns a copy of the value in *DAYSINYEAR*.

In `main()`, declare a *Date* object called *d1* and then attempt to display the value held in *DAYSINYEAR* using the terms *d1.GetDAYSINYEARconst()* and *Date::GetDAYSINYEARconst()*.

The trouble now is that although it is fine to call the new method when accessed via a *Date* object as in the expression `d1.GetDAYSINYEARconst()`, it is not valid to attempt to call it using the class name. This means that although the static constant DAYSINYEAR exists before any *Date* object is created, there is no way to access it until such an object is brought into existence.

Static Methods

Methods, as well as attributes can be defined as static. When this is done, the method, like the attribute, exists even before any objects of that class are declared. One reason for creating a static method is to allow access to private static attributes before any class objects are created. For example, if we change the prototype for *Date*'s *GetDAYSINYEARconst()* to



The term `static` is not required in the function's heading.

```
static int GetDAYSINYEARconst();
```

then, now that we have a static method, we are free to use the expression

```
Date::GetDAYSINYEARconst()
```

in our program.

Activity 16.35

Add the word `static` to the start of *Date* class's Get*DAYSINYEARconst()* prototype.

Gaining access to private static attributes is not the only reason to use static methods. Sometimes we want to collect together a set of functions which have a common

theme (just as we might sit all our programming books on the same shelf).

These functions are not designed to manipulate a common set of attributes but are placed together to give a neat and organised approach. For example, the Java programming language holds all of its maths functions within a class called *Math*.

In this *Math* class we will find functions to perform such mathematical operations as square root, sine, cosine, tangent, raise to a power, etc. Now, if we want to find, say, the square root of a number, it wouldn't make sense to have to create a *Math* object in order to access the square root method. Instead, since all the methods are static, we would just write the Java equivalent to:

```
cout << Math::Sqrt(25.78) << endl;
```

Static methods are also referred to as **class methods**.

Since a static method can be called without an object of its class being created, such methods cannot access standard (non-static) attributes of the class.

Static Variables

It's not just constant values that can be declared as static. Regular attributes can also be static, meaning that the only one instance of that property is created irrespective of the number of objects that exist.

One common use for such an attribute is to keep a count of the number of objects of a class that currently exist. For example, let's assume we wish to keep a count of the number of *Date* objects that exist in a program. We would start by adding a new static attribute and operation to the class:

```
class Date
{
   private:
      static const int DAYSINYEAR = 365;
      static int count;
      int day;
      int month;
      int year;

      long ToJDN();
      Date FromJDN(long);

   public:
      Date(int=1,int=1,int=2001);
      void SetDate(int, int, int);
      void SetDate(int, int);
      int GetDayOfWeek();
      inline bool IsLeapYear();
      int GetDay();
      int GetMonth();
      int GetYear();
      Date operator+(int);
      int operator-(Date);
      Date operator-(int);
      bool operator==(Date);
      void operator+=(int);
      Date operator++();
      Date operator++(int);
      Date operator--();
      Date operator--(int);
```

```
        int GetDaysInYear();
        static int GetDAYSINYEARConst();
        static int GetCount();
        ~Date();
};
```

In the *Date* class constructor we will increment *count:*

```
//*** Initialises new date object to d/m/y ***
Date::Date(int d, int m, int y)
{
   // *** Days in each month of year ***
   int daysinmonth[] = {0, 31,28,31, 30,31,30, 31,31,30,
   ↳31,30,31};

   bool valid = true;  // Parameters valid

   //*** If month invalid, invalid parameters ***
   if (m < 1 || m > 12)
      valid = false;

   //*** Add 1 to days in February if leap year ***
   daysinmonth[2] += (y%400 == 0)||(y%4 == 0 && y%100 != 0);

   //*** If days in month invalid, incalid parameters ***
   if (d <1 || d > daysinmonth[m])
      valid = false;

   //*** If year less than 1, invalid parameters ***
   if (y < 1)
      valid = false;

   //*** Assign date ***
   if (valid)
   {
      day = d;
      month = m;
      year = y;
   }
   else
   {
      day = 1;
      month = 1;
      year = 2001;
   }
   //*** Add one to count of Date objects **
   count++;
}
```

and in the destructor, we will decrement *count:*

```
//*** Class destructor ***
Date::~Date()
{
   //*** Decrement count of Date objects ***
   count--;
}
```

The other new method, *GetCount()* returns the value held in *count*:

```
//*** Returns the number of Date objects existing ***
int Date::GetCount()
{
   return count;
}
```

There's one last thing we need to do and that is to set the initial value for our new static attribute. This is done using the line

```
int Date::count = 0;
```

which is traditionally placed immediately after the class declaration.

Activity 16.36

Change the code for *Date* class to include the attribute *count* as defined above along with the *GetCount()* method.

Modify the constructor to increment *count* and create a destructor which decrements *count*.

Test the new class using the following code:

```
int main()
{
   cout << "There are " << Date::GetCount() << " Date objects\n";
   Date dt1(23,11,1963);
   cout << "There are " << Date::GetCount() << " Date objects\n";
   {
      Date dt2;
      cout << "There are "<<Date::GetCount() << " Date objects\n";
   }
   cout << "There are " << Date::GetCount() << " Date objects\n";
}
```

Overloading << and >>

By now you must be tired of always writing code such as

```
cout << d1.GetDay() << '/' << d1.GetMonth() << '/' <<
↳d1.↳GetYear() << endl;
```

How much nicer it would be to simply write

```
cout << d1 << endl;
```

and get the same result.

To do this, we need to overload the `<<` operator.

Our first guess at how to do this might be to add the overloaded operator to the *Date* class in just the same way as we added previous operators. But unfortunately, things are a little more complicated than that.

We have made use of the command `cout` since the very first program back in Chapter 3, so perhaps it will come as something of a surprise to realise that `cout` isn't a command at all, but is in fact an object of the class `ostream`. When we write a line of code such as

```
cout << 12;
```

we are actually making use of the `<<` operator as defined within the `ostream` class. We could just as validly rewrite the line as

```
cout.operator<<(12);
```

This should make it clearer that if we want to write a line such as



We'll be covering the ostream class in detail in a later chapter.

```
cout << d1; //d1 is a Date object
```

which could be rewritten as

```
cout.operator<<(d1);
```

that it is actually the code for the `ostream` class that needs to be changed to accommodate outputting a *Date* class object and not, as we first thought, the *Date* class.

Now, since we do not have access to the source code for the `ostream` class, modifying its code is not an option. This means we have to resort to creating a standard function, which is not part of any class, to overload the `<<` operator.

The function will take an `ostream` object reference and a *Date* object reference as its parameters:

```
void operator<<(ostream& ct, Date& d)
```

Our new function has no access privileges to private properties of either class so its code needs to make use of methods such as *GetDay()* to access the required attributes.

The code for the complete routine is

```
void operator<<(ostream& ct, Date& d)
{
   ct << d.GetDay() << '/'<< d.GetMonth() << '/'<< d.GetYear();
}
```

Now, rather than write the correct but unwieldy

```
operator<<(cout,d1)  //Assuming d1 is a Date object
```

we can write

```
cout << d1;
```

The function isn't quite complete yet, because it needs to return an `ostream&` value to allow it to be used in longer output statements. For example, if we wanted to write

```
cout << d1 << d2;
```

there would be a problem. This is more clearly seen if we rewrite the statement as

```
operator<<(operator<<(cout,d2),d1);
```

From this we can see that the value returned from the inner function call is required as the first parameter to the outer function call. To achieve the required results, our function needs to return a value of type `ostream&`. The updated version of the function is

```
ostream& operator<<(ostream& ct, Date& d)
{
   ct << d.GetDay() << '/'<< d.GetMonth() << '/'<< d.GetYear();
   return ct;
}
```

Friends

Writing functions which are not part of the *Date* class and yet are obviously logically part of that class seems a bit disjointed. We can overcome this problem by defining our new function as a friend of the *Date* class.

```
class Date
{
   private:
```

```
        .
        .
    public:
        .
        .
        friend ostream& operator<<(ostream&,Date&);
};
```

The advantage of declaring the overloaded operator as a friend of the *Date* class is that it now has access to the private attributes, meaning we can use a term such as

```
d.day
```

rather than

```
d.GetDay()
```

allowing us to rewrite the code for `operator<<()` as:

```
ostream& operator<<(ostream& ct, Date& dt)
{
   ct << dt.day <<'/' << dt.month << '/' << dt.year;
   return ct;
}
```

Activity 16.37

Add `operator<<` as a friend in the *Date* class and test that the new operator works correctly by using it to display the contents of a *Date* object.

At this point, it won't now come as too much of a surprise to learn that `cin` is also an object. It is an instance of the class `istream`.

This time we will cheat a little and use `scanf()` within the new function rather than `cin` since we can then enter the / symbols between each part of the date. The new function has the following code.

```
istream& operator>>(istream& cn, Date& dt)
{
   scanf("%d/%d/%d",&dt.day, &dt.month, &dt.year);
   return cn;
}
```

Notice that we still need an `istream` parameter which is returned by the function even though the parameter is not otherwise used within the code.

Activity 16.38

Add `operator>>` as a friend in the *Date* class.

Test the new operator by reading and displaying the value of a *Date* object.

This latest function highlights the problem with friends - they get to override the privacy of the attributes and unless they take the same precautions as regular class methods, there is the danger that they may allow the contents of an object to be corrupted.

Activity 16.39

Modify the code for `operator>>` so that only a valid date will be stored in the *Date* object. Where the date entered is invalid, the *Date* object's values should remain unchanged.

More Friends

Another reason to create functions which are not part of a class is highlighted by the following example.

If we create a *Date* object and then add a few days to the value, we could do this with the following code:

```
Date d1(22,11,1963), d2;
d2 = d1 + 1;   // d2 is 23/11/1963
```

This works because we defined the `+` operator in the *Date* class. But remember what is really happening here is more accurately reflected when we rewrite the last line as

```
d2 = d1.operator+(1);
```

so it should come as no surprise that rearranging the assignment statement to read

```
d2 = 1 + d1;
```

won't work. This time the `+` is not the operator defined in *Date* – the elements are in the wrong order. And because of this, the program will not compile.

However, we can solve our problem by writing a separate, standard function:

```
Date operator+(int days, Date& d)
{
   Date dt = d.FromJDN(d.ToJDN() + days);
   return dt;
}
```

Since we made the methods `FromJDN()` and `ToJDN()` private, this function will only compile if we make it a friend of the *Date* class, thereby giving access to the two routines.

Activity 16.40

Add the new `operator+` function described above as a friend of the *Date* class and test it using a second *Date* object which is set to the first *Date* object's date + 10 days.

More on Class Diagrams

We first saw class diagrams at the start of this chapter, but since then we have covered many additional class features which it would be useful to identify within a diagram.

The following features can be added:

+	at the start of a property indicates that it is public.
-	at the start of a property indicates that it is private.
#	at the start of a property indicates that it is protected.



We'll talk about protected attributes in the next chapter.

= value	after a property shows its initial value. OR after a parameter, giving it's default value.
underlined	below a static property.
«constructor»name	for constructors.
«destructor»name	for destructors.
«friend»name	for functions which are friends of the class.
in, **out**, **inout**	one of these terms can be used in front of each operation parameter to indicate which direction a parameter's value is being passed.

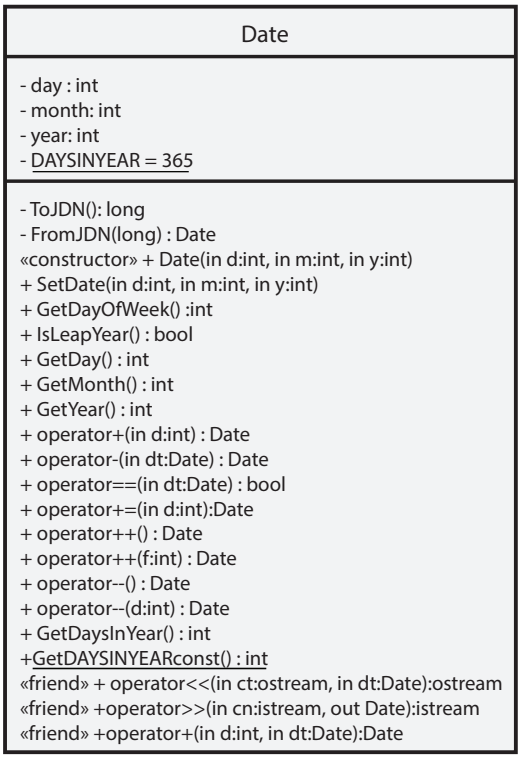
When naming the constructors and the destructor in a class diagram, we can use the C++ convention of naming them after the class or you can use other terms such as *Create* and *Destroy* or *New* and *Delete*.

There are no class diagram conventions for showing finer details such as the fact that a method uses `const` parameters or does not modify any of the attributes of the class. In any case, the decision to use `const` parameters or call by reference variables is often a decision made at a later stage in the development process.

Of course, since the purpose of a class diagram is an aid to our task, we are really free to add whatever detail we might find helpful later when we come to develop our code. FIG-16.11 shows a class diagram for the latest version of the *Date* class using the new features described above.

FIG-16.11

Date Class Diagram



In addition, we would need to create mini-specs for each operation mentioned in the class diagram.

Visualising An Object

The class diagram is an essential tool when designing a new class, but a more informal way of looking at a *Date* object, which, at this early stage in the learning process, may give us a greater insight into how objects operate is shown in FIG-16.12.

FIG-16.12

How an Object Operates

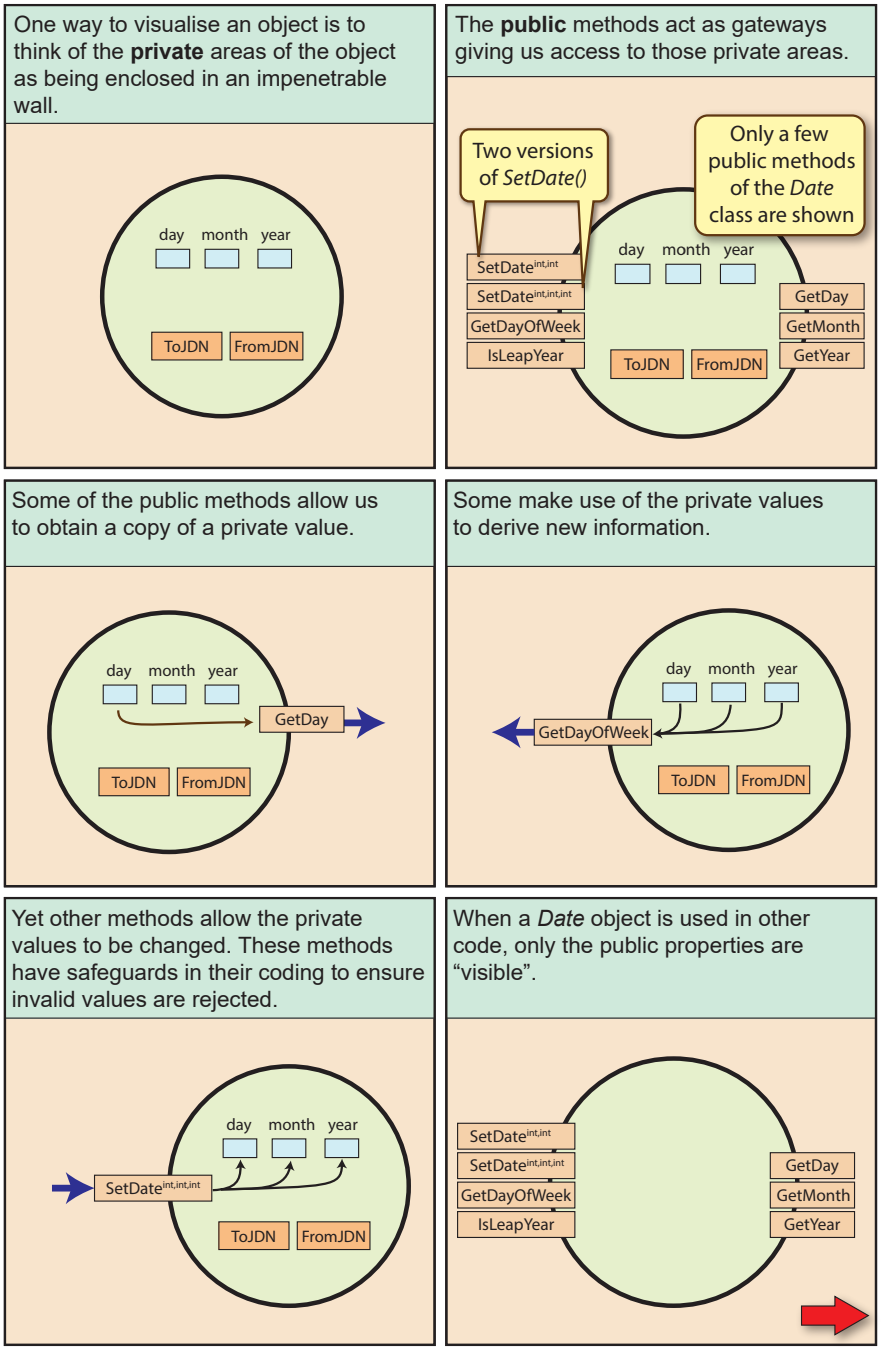
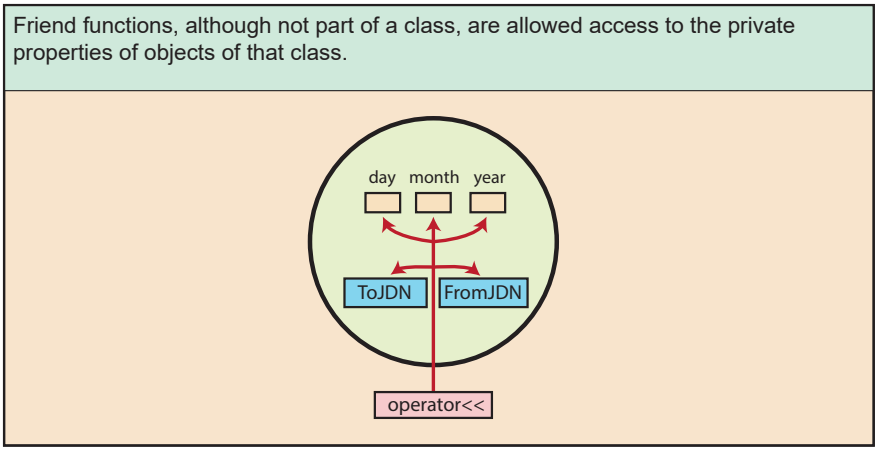


FIG-16.12
(continued)

How an Object Operates



Student Class

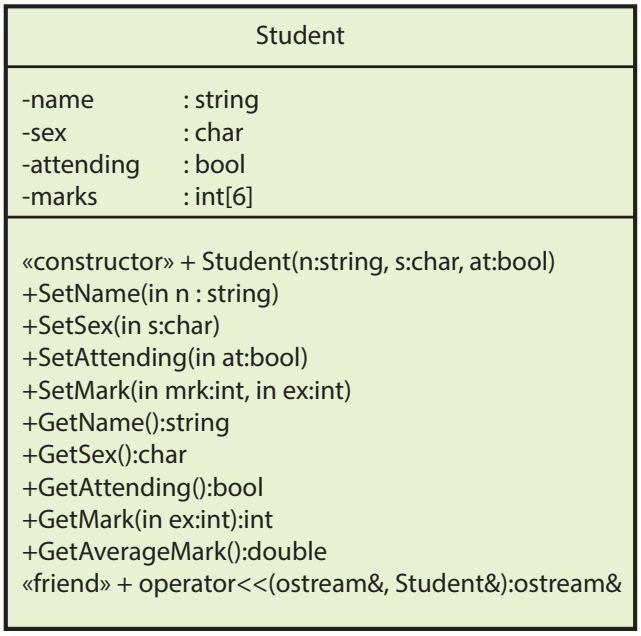
We've spent most of this chapter constructing the *Date* class. Obviously, as an introduction to classes, we've made use of a very simple class. And, although we've looked at various types of operations we can add to a class, we've done little with the type of attributes that can be used within a class.

In fact, so far we've only used `int` attributes (one of which was static). But of course, we can have `float`, `char`, `bool` attributes as well as arrays, pointers, structs, and even objects.

To demonstrate a larger range of attribute types, we'll create another class, *Student*, which contains details of a college student along with their marks for six exams. The class diagram for *Student* is shown in FIG-16.13.

FIG-16.13

Student Class Diagram



The code for most of the new class is shown in FIG-16.14.

FIG-16.14

Student Class Code

```
#include <iostream>
using namespace std;

class Student
{
   private:
      char name[21]{ "" };
      char sex{ '?' };
      bool attending{ true };
      int marks[6]{ 0 };
   public:
      Student(const char*, char, bool = true);
      void SetName(char*);
      void SetSex(char);
      void SetAttending(bool);
      void SetMark(int, int);
      char* GetName();
      char GetSex();
      bool GetAttending();
      int GetMark(int);
      double GetAverageMark();

      friend ostream& operator<<(ostream&, Student&);
};

//************************************
//***    Student class Operations   ***
//************************************

Student::Student(const char* n, char s, bool at)
{
   strcpy_s(name,strlen(n)+1,n);
   s = toupper(s);
   if (s == 'M' || s == 'F')
      sex = s;
   attending = at;
}


//*** Set the student's name to n ***
void Student::SetName(char * n)
{
   strcpy_s(name,strlen(n)+1 n);
}


//*** Set student's sex to s ***
void Student::SetSex(char s)
{
   s = toupper(s);
   if (s == 'M' || s == 'F')
      sex = s;
}


//*** Set student's attendence status ***
void Student::SetAttending(bool at)
{
   attending = at;
}
```



FIG-16.14
(continued)

Student Class Code

```
//*** Set student's mark for specified exam ***
void Student::SetMark(int mrk, int ex)
{
   //*** If not valid exam, return ***
   if (ex < 0 || ex > 5)
      return;
   //*** If not valid mark, return ***
   if (mrk < 0 || mrk > 100)
      return;
   //*** Store mark ***
   marks[ex] = mrk;
}

//*** Return student's name ***
char* Student::GetName()
{
   return name;
}

//*** Return student's attendence status ***
bool Student::GetAttending()
{
   return attending;
}

//*** Return student's exam mark ***
int Student::GetMark(int ex)
{
   if (ex < 0 || ex > 5)
      return -1;
   return marks[ex];
}

//*** Return student's average mark ***
double Student::GetAverageMark()
{// TO BE CODED}

//*** Display contents of student object ***
ostream& operator<<(ostream& ct, Student& st)
{//TO BE CODED}

int main()
{
   Student st1("Madeline Bray", 'F');
   Student st2("Nicholas Nickleby",'M');
   Student st3("Ada Clare", 'F', false);
   Student st4("Richard Carstone", 'M');
   //*** Assign random marks ***
   for (int ex = 0; ex < 6; ex++)
      st1.SetMark(rand() % 101, ex);
   for (int ex = 0; ex < 6; ex++)
      st2.SetMark(rand() % 101, ex);
   for (int ex = 0; ex < 6; ex++)
      st3.SetMark(rand() % 101, ex);
   for (int ex = 0; ex < 6; ex++)
      st4.SetMark(rand() % 101, ex);
   //*** Display object contents ***
   cout << st1 << endl;
   cout << st2 << endl;
   cout << st3 << endl;
   cout << st4 << endl;
}
```

Activity 16.41

Start a new project, *StudentClass*, and implement the code given above.

Add the code for the two missing methods, *GetAverageMark()* and *operator<<()*. *GetAverageMark()* should return the average of the six marks.

operator<<() should display the contents of the complete object and finish with the average mark. The attribute *attending* should be displayed as "*attending*" or "*not attending*".

The first array in our *Student* class is the *name* attribute which makes use of an array of type `char`.

In any modern C++ code we would use a `string` class object (covered in Chapter 20) for storing data such as the student's name but if we ever encounter existing code from a few years back it may well use an array of `char` or a `char` pointer to store strings. If you're never going to encounter older C++ code, feel free to skip over this topic.

Using *char* Pointers

Although very useful, standard arrays can be a bit problematic. Make them too small and we cannot store all our data; make them too large and we waste space. In the case of string constants, C++ solves this problem by reserving exactly the space required.

Older code might use this same approach when storing a string value by employing a pointer in conjunction with the `new` command to reserve the exact amount of space required. For example, in the *Student* class we could use a `char` pointer rather than a `char` array for the name, it would be:

```
class Student
{
   private:
      char* name;
      char sex;
         ...
```

A function which sets the *name* attribute in objects of this class would be coded as:

```
void Student::SetName(const char* n)
{
   //*** If no new name, exit ***
   if (n == nullptr)
      return;
   //*** Delete any space already being used ***
   if (name != nullptr)
      delete[]name;
   //*** Create enough space for the new name ***
   name = new char[strlen(n)+1]; //Extra byte for final null
   //*** Copy name into this new space ***
   strcpy_s(name,strlen(n)+1,n);
}
```

When we use a pointer as an attribute or parameter, it's important that any method which accesses that attribute checks for the value `nullptr` (meaning the pointer does not reference a data area) and handles that situation appropriately. In the above method we can see a parameter guard to check against *n* containing null and that space is only deallocated if *name* is not set to null.

When a class makes use of dynamically allocated space, it becomes important that we add a destructor to the class which deletes any allocated space.

```
Student::~Student()
{
   if(name != nullptr)
      delete [] name;
}
```

We might be asking at this point, why not use a smart pointer? That way we could avoid having to delete the allocated space in the destructor. However, we must remember, we're only likely to find this way of doing things in older code before the `string` class and smart pointers were introduced.

Activity 16.42

Modify *StudentClass* so that the *name* attribute is implemented as a `char` pointer (which should have an initial value of `nullptr`).

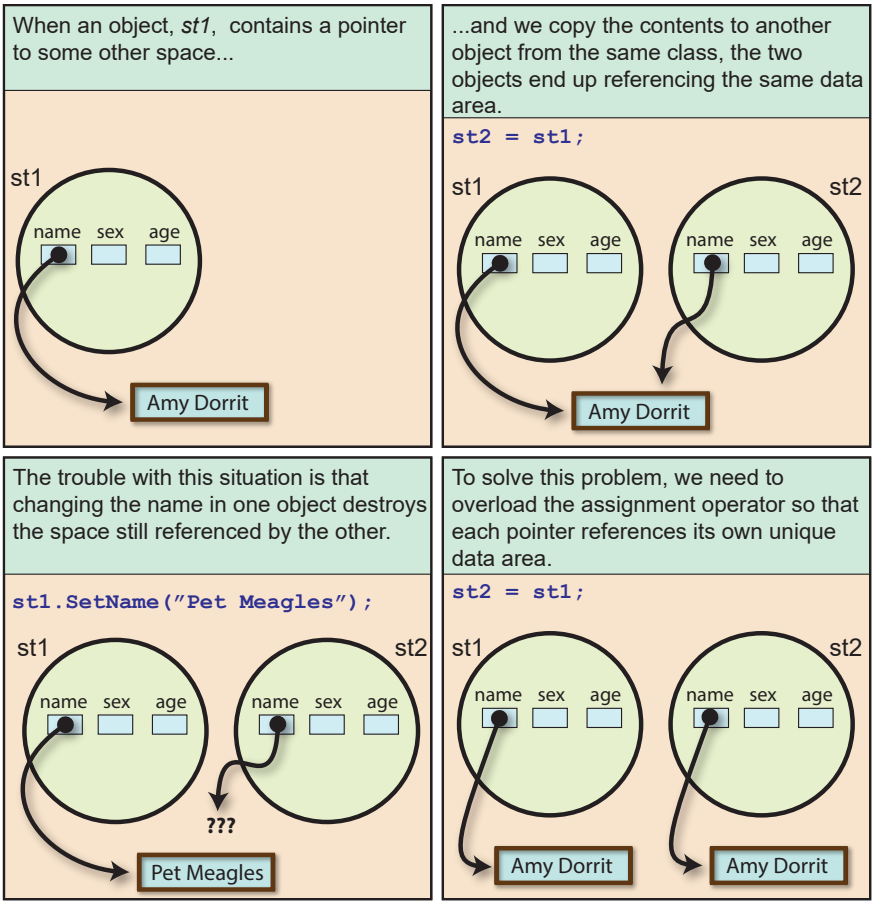
Make any other necessary changes. Check that the program operates exactly as before.

Overloading the Assignment Operator

Classes containing pointers to dynamically allocated space present another problem when we try to copy the contents of one object to another (see FIG-16.15).

FIG-16.15

The Trouble with Pointers



Activity 16.43

Modify *StudentClass* so that the program logic is:

Create Student objects st1 to st4 (as before)
Copy st1 to object st2
Display the contents of st1 and st2
Change the name in st1 to "Pet Meagles"
Display the contents of st1 and st2

What happens when you run the program?



Expect the program to crash!

Now that we've experienced the problems of the default assignment operator, we can correct the problem with the following code:

```
//*** Copies a Details object to current Details object ***
Student& Student::operator=(Student& st)
{
   //*** Delete any space already being used ***
   if (name != nullptr)
      delete[]name;
   //*** Set up space for copy of name ***
   int size = strlen(st.name)+1;
   name = new char[size];
   //*** Copy name ***
   strcpy_s(name,size,st.name);
   //*** Copy sex ***
   sex = st.sex;
   //*** Copy attending ***
   attending = st.attending;
   //*** Copy marks ***
   for(int c = 0; c < 6; c++)
      marks[c] = st.marks[c];
   //*** Return reference to updated object ***
   return *this;
}
```

The function returns a reference to the updated object. This allows the operator to be used in situations such as

```
st3 = st2 = st1;  //All Detail objects
```

Activity 16.44

Modify *StudentClass,* adding the = operator to the class.

How does this affect the results produced?

The assignment operator is closely related to the copy constructor: if we need to create code for one of these methods, we will almost certainly have to create near identical code for the other.

For example, in the *Details* class, with the *name* attribute now implemented as a pointer, the code

```
Student st2(st1);
```

will require exactly the same logic to copy the contents of *st1* to *st2* as we employed in the *operator=()* method.

Activity 16.45

Modify *StudentClass,* adding a copy constructor to the class. Test the new code by copying the contents of *st1* to a new *Student* object using the line

```
Student st5(st1);
```

There are even situations where we may be forgiven for thinking we are using the assignment operator when, in fact, we are calling the copy constructor. For example, in the line

```
Details st2 = st1;
```

it is the copy constructor that is called to copy the contents of *dt1* to *dt2*. This is because the assignment is being made as the object is being created. On the other hand, the lines

```
Details st2;
st2 = st1;
```

first use the zero-argument constructor to create *st2* and then the assignment operator to copy *st1* to *st2*.

const Methods

Back in Chapter 11, we saw that reference parameters are used in a function when we want to update that parameter within the function. But a second reason for using a reference parameter is that no copy is made of the actual parameter's value. This can save significant time and space when that parameter is a complex record structure or object.

When we want to pass a parameter as a *pass-by-reference* value for the sake of efficiency but without the intention to change any values held within the parameter, we had the option to add the term `const` in the parameter list:

```
GetDayOfWeek(const Date& d)
```

This same option is open to us when creating methods within a class. For example, if we wanted to make the parameter to the *Detail* class's *operator=()* method a `const`, we would start the method's code with the line

```
Student& Student::operator=(const Student& dt)
```

Activity 16.46

Modify *StudentClass,* so that the `=` operator now uses a `const` parameter (two lines of code need to be changed). Check that the program operates exactly as before.

Although making a parameter to a class method a `const` isn't a problem, the same can't be said when the application programmer tries to write a function in the same way. For example, let's say we want to write a function that returns the number of characters in a *Student* object's name field. We could try writing this as

```
int GetNameSize(const Student& st)
{
   return strlen(st.GetName());
}
```

Remember this is not a method of the *Details* class, but a separate function written by the application programmer.

When we call this function in `main()` with a line such as

```
cout << GetNameSize(st1) << endl;
```

and try to compile the code, we would be presented with a compilation error for the line within *GetNameSize()*.

Activity 16.47

Modify *SudentClass,* adding *GetNameSize()* as a standard function and using the function to display the size of the name in *st1*.

Try compiling your code *(there will be an error)* .

The reason for our problem is relatively simple: we have stated that the parameter to *GetNameSize()* is a `const` parameter. That means that *st* (the parameter) cannot be modified in any way within the function. But the function calls a method of the *Student* class, *GetName()*, and the compiler has no way of knowing if that method modifies the contents of *st*. Because it does not know this, the compiler throws up an error.

To solve this general problem, we need to tell the compiler which methods defined within the *Student* class modify attributes of that class and which don't.

For example, we know that the methods

```
SetName()
SetSex()
SetAttending()
```

all modify *Student* class attributes. On the other hand,

```
GetName()
GetSex()
GetAttending()
```

only retrieve the current values of attributes but do not change them.

To let the compiler know that a method does not modify any attribute within its own class, we add the term `const` to the end of the function prototype. Hence, the *Details* class code would now begin with:

```
class Student
{
   private:
      char* name{ nullptr };
      char sex{ '?' };
      bool attending{ true };
      int marks[6]{ 0 };
   public:
      Student(const char*, char, bool = true);
      Student(const Student&);
      void SetName(const char*);
      void SetSex(char);
      void SetAttending(bool);
      void SetMark(int, int);
      char* GetName() const;
      char GetSex() const;
      bool GetAttending() const;
      int GetMark(int) const ;
      double GetAverageMark() const;
```

```
        Student& operator=(const Student&);
        ~Student();

        friend ostream& operator<<(ostream&, Student&);
};
```

The need for `const` does not apply to *friend* functions.

The functions themselves must also include the `const` term in their heading as in the code

```
char* Student::GetName()const
{
   return name;
}
```

Activity 16.48

Modify *StudentClass,* so that the appropriate methods are defined with a `const` ending and check that the program now executes.

const Methods and *char* Arrays Attributes

There is one situation where declaring a `const` method can cause a slight problem. Let's assume we have declared a *Trivial* class and made use of it as shown in FIG-16.16.

FIG-16.16

Trivial Class

```
#include <iostream>

using namespace std;

class Trivial
{
   private:
      char name[31];
   public:
      void SetName(const char*);
      char* GetName() const;
};

void Trivial::SetName(const char* n)
{
   if(n != nullptr)
      strncpy(name, n, 30);
}


char* Trivial::GetName() const
{
   return name;
}


int main()
{
   Trivial tv;
   tv.SetName("Libby");
   cout << tv.GetName() << endl;
}
```

Activity 16.49

Create a new project called *TrivialClass*, and create a source file containing the code given above.

What happens when you attempt to compile the code?

The problem is that, since we have declared *GetName()* as a `const` method, C++ will not return a pointer to the data in case the pointer is dereferenced and used to modify the data - hence the compilation error.

However, if we declare the return value to be a pointer to a `const` value, then C++ will be mollified and allow the compilation. So, to have the program compile we need to change the prototype and heading for *GetName()* to read

```
const char* GetName() const
```

Activity 16.50

Change the appropriate two lines in *TrivialClass* and rerun your project.

Object Pointers

Just as we can create pointers which reference basic data types and record structures, so we can create pointers to class objects. For example, we could create a pointer to a *Student* object using the line:

```
Student* stptr;
```

The most likely use of such a pointer is to have it reference a dynamically allocated object as in:

```
stptr = new Student;
```

The statement above will create a new object, executing its zero-argument constructor to initialise the object's attributes, and assign the start address of the object to *stptr*. If we have written a constructor that accepts arguments, that can be used in the creation of the object:

```
stptr {new Student("Henry Gowan",'M')};
```

Once the object has been created and referenced by the pointer, we use the same syntax to access the public properties of the object as we employed earlier with record (struct) pointers. Hence, we could retrieve the name held in the object with the expression

```
stptr->GetName()
```

which dereferences the pointer to gain access to the object's elements.

Activity 16.51

Modify *StudentClass,* so that *st1* and *st2* are pointers to *Student* objects (with *st3* and *st4* remaining as standard objects). Change the output statements where necessary to display the contents of all four objects.

An Array of Objects

If we require several objects of the same type, we are free to create an array of objects. For example, we might write

```
Student group[4];
```

When we create an array of objects in this way, it is the zero-argument constructor that is used to initialise each of the objects. If the class does not have a zero-argument constructor, then attempting to create the array will produce a compilation error.

Accessing public properties of the object requires a combination of array element access and record field access. Hence, to execute the *GetName()* method of the third object in the array we would use the term

```
group[2].GetName()
```

Activity 16.52

Modify *StudentClass* so that all four *Student* objects are stored in a *Student* class array called *group*. Assign the same values as before to the objects and display their contents.

Arrays of Class Pointers

Another option when handling several objects from the same class is to have an array of class pointers. For example, we could create space for five *Date* pointers with the line:

```
Date* event[5];
```

Of course, if we do this, then we must also create space for each object before using it:

```
for(int c = 0; c <=4; c++)
   event[c] = new Details;
```

Activity 16.53

Modify *StudentClass* so that it uses an array of pointers rather than an array of objects.

We have already seen that the main advantage of such a system is to ensure, by writing robust methods, that the data held is not corrupted by allowing invalid values to be assigned.

Time Class

Like most new concepts it takes a bit of experience before we become fully understanding of all the subtleties behind those ideas.

To help gain a better grasp of coding classes as well as setting up code that we will revisit in later chapters, the Activity that follows will get you to develop a new class, *Time*, from scratch.

The class diagram for the new class is shown in Activity 16.53.

Of course, to accompany a class diagram, we would expect to see mini-specs for each of the operations stated in the diagram. However, in this case, since much of what is required shares a close similarity to methods already coded for the *Date* class, we'll show just two mini-specs here.

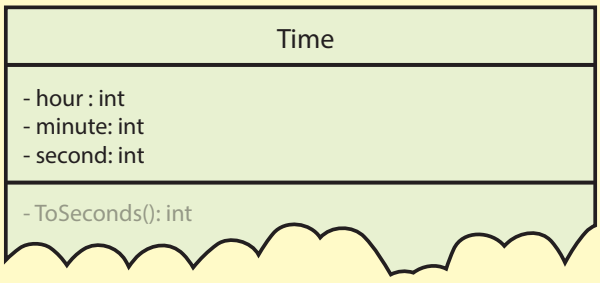
Those are for the private methods, *ToSeconds()* and *FromSeconds()*.

Class	: Time
Operation	: ToSeconds
Parameters	
In	: None
Out	: result : int
Attributes	
Read	: hour, minute, second
Written	: None
Pre-condition	: None
Post-condition	: *result* = *hour* *3600 + *minute*60+*second*
Description	: Sets *result* to the number of seconds since midnight to the current time.

Class	: Time
Operation	: FromSeconds
Parameters	
In	: s : int
Out	: result : Time
Attributes	
Read	: None
Written	: None
Pre-condition	: None
Post-condition	: *result.hour* = *s*/3600 *result.minute* = (*s* - *result.hour**3600)/60 *result.second* = *s*%60
Description	: Converts *s* seconds into hours, minutes and seconds storing the value in *result*.

Activity 16.54

Start a new project called *TimeClass* and develop all of the code necessary to define a *Time* class as shown below. Do not code the methods yet.

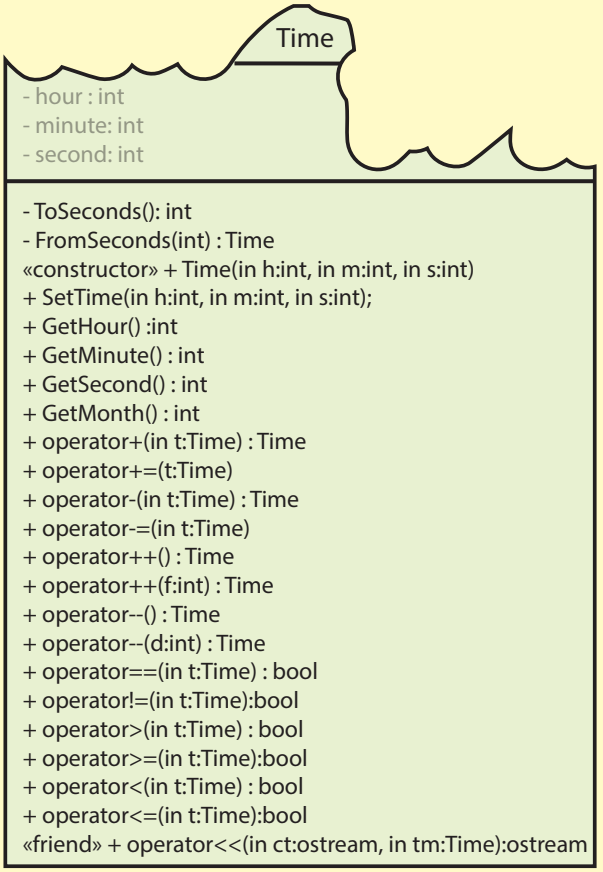






Remember a UML class diagram doesn't tell us everything about a method's coding. For example, it does not tell us if we should use const values, or reference values or a pointer. It does not tell us if a method should be declared as const. These are things that are normally decided at a later stage or in further documentation.

Activity 16.54 (continued)



Make your own decision on the use of *const*, and references based on the classes covered previously in this chapter.

Activity 16.55

In project *TimeClass* implement all of the operations for the *Time* class and create a `main()` function to test one arithmetic operator, one relational operator and the << operator.

The constructor and *SetTime()* method should allow the *hour* attribute to be set to values above 23 but not below zero. No upper limit will be useful when an object of this type is used to represent a duration time rather than a clock time.

Summary

- A class is a blueprint for a structure containing both data and related operations.
- This combining of the data and operations is known as encapsulation.
- The elements of a class are collectively known as properties, features or members.

- The data elements of a class are known as attributes.
- The tasks of a class are known as operations.
- An object is a program component which realises a class specification.
- An object is also called an instance of a class.
- A class diagram is a rectangle split into three spaces. These contain: the class name; the attributes' names and types, and the operations' names, parameters and return type.
- When designing a class, use:

+	to mark a public feature
-	to mark a private one
= value	to assign a value to a named constant
_	underline to mark a static feature
«constructor»	to mark a constructor
«destructor»	to mark a destructor.

- In C++, a class is defined using the term `class`.
- When coded, the operations of a class are known as methods.
- There are two parts to coding a class: the class declaration and the methods' code.
- The call operator is the name given to the set of open and close parentheses.
- A functor is any class that defines a method for the call operator.
- Inline methods can be coded within the class declaration or marked with the keyword `inline`.
- Properties of a class can be marked as public or private.
- When coding the methods of a class, the class name and the scope resolution operator must precede the method name.
- C++ automatically adds a zero-argument constructor, a copy constructor, an assignment operator, and a destructor to every class.
- The default copy constructor and assignment operator copy the attribute values in one object to the corresponding attributes in another object.
- The default constructors, the assignment operator and destructor can be overridden by defining new operations within the class.
- A constructor cannot return a value.
- A destructor cannot return a value or take parameters.
- All methods within a class can be overloaded.
- A program using an object can access only its public properties.
- A class constructor is run automatically when an object is created.
- If an object is created with the value of an existing object, the copy constructor is executed.
- The class destructor is run automatically when an object is deleted.
- A program will, where possible, automatically make use of a class's

constructors to convert another type to an object of that class.

- To stop a constructor being used automatically to convert other values, start the definition of each constructor with the term `explicit`.
- When constructors are declared as explicit, then conversions to an object must be made explicit.
- Data hiding limits access to the features of a class when using an object of that class.
- C++ maintains a pointer called *this* which references the current object whose features are being accessed.
- Constants can be defined within a class.
- Attributes of a class can be marked as static.
- There is only a single copy of each static attribute (not one in each object).
- Methods can also be declared as static.
- Static features in a class exist even before any object of that class has been created.
- Public static features can be accessed using the format

```
class name :: static feature name
```

- The code within a static methods cannot access non-static attributes of its class.
- When overloading the insertion (`>>`) and extraction (`<<`) operators, to display an object, the code must create standard functions rather than operations of the class.
- Standard functions can have access to the private features of a class by declaring the functions as friends of the class.
- When an attribute of a class is a pointer, make sure that the copy constructor and assignment operator handle the pointer appropriately.
- A method which does not modify the value of any class attribute should add the term `const` to the end of its declaration and heading.
- Only methods marked as `const` have access to the attributes of an object or parameter marked as `const`.

Solutions

Activity 16.1

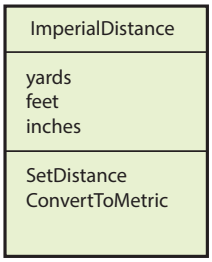
Beach ball characteristics include:
- Size
- Colour
- Weight
- Material used

Operations include:
- Inflate
- Deflate
- Throw
- Catch
- Bounce

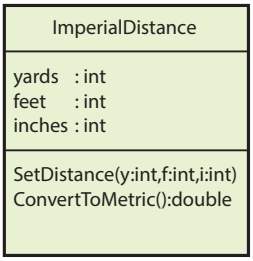
Activity 16.2

a) Class
b) Object
c) Class
d) Object

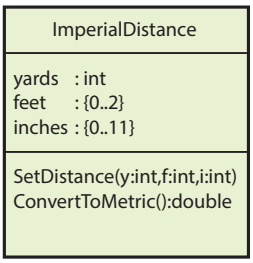
Activity 16.3



Activity 16.4



Activity 16.5



Activity 16.6

Class	: ImperialDistance
Operation	: setDistance
Parameters	
In	: y : int
	f : int
	i : int
Out	: None
Attributes	
Read	: None
Written	: yards, feet, inches
Pre-condition	: y,f,i forms a valid Imperial distance
Post-condition	: yards = y
	feet = f
	inches = i
Description	: Sets the distance to *y* yards, *f* feet, *i* inches.

Activity 16.7

Code for *DateClass*:

```
#include <iostream>
using namespace std;

//*** Class declaration ***
class Date
{
  public:
     int day;
     int month;
     int year;

     void SetDate(int, int, int);
     int GetDayOfWeek();
     bool IsLeapYear();
};
```

Activity 16.8

The code should compile.

Activity 16.9

Code for *ImperialDistanceClass*:

```
#include <iostream>
using namespace std;

class ImperialDistance
{
  public:
     int yards;
     int feet;
     int inches;

     void SetDistance(int,int,int);
     double ConvertToMetric();
};



//*************************************
//*** ImperialDistance Class Methods ***
//*************************************

// *** Sets distance to y yds, f ft, and i in ***
void ImperialDistance::SetDistance(int y, int f,
int i)
{
  //*** If any parameter invalid, exit ***
  if (i < 0 || i > 11 || f < 0 || f > 2 || y < 0)
     return;
  //*** Assign distance ***
  yards = y;
  feet = f;
  inches = i;
}

// *** Returns equivalent distance in metres ***
double ImperialDistance::ConvertToMetric()
```

```
{
  return ((yards*36 + feet*12 + inches)*0.0254);
}

int main(){}
```

Activity 16.10

The supplied date fell on a Saturday.

Activity 16.11

Since the parameters to *SetDate()* are invalid, the attributes of *d1* are never assigned values and so the random values within the memory assigned to those attributes are displayed.

Activity 16.12

Coding for `main()` in *ImperialDistanceClass*:

```
int main()
{

  ImperialDistance dist;  // ImperialDistance object

  // *** Set distance ***
  dist.SetDistance(0,0,0);

  // *** Display distance ***
  cout << dist.yards << " yards  " << dist.feet <<
  ↳" feet  " << dist.inches << " inches " << endl;
}
```

When we change the call to *SetDistance()* to use the invalid parameters 2,3,10, we have a similar situation to that of the last Activity. The parameters to *SetDistance()* are invalid and hence the attributes of the distance objects retain their original random values which are displayed on the screen.

Activity 16.13

Modified code for *DateClass*:

```
#include <iostream>
using namespace std;

class Date
{
  public:
    int day;
    int month;
    int year;

    void SetDate(int, int, int);
    void SetDate(int,int);
    int GetDayOfWeek();
    bool IsLeapYear();
};


//*************************************
//***       Date Class Methods       ***
//*************************************
// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = {0, 31,28,31, 30,31,30,
  ↳31,31,30, 31,30,31};
  //*** If month invalid, exit ***
  if (m < 1 || m > 12)
    return;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += (y%400 == 0)||(y%4 == 0 &&
  ↳y%100 != 0);
  //*** If days in month invalid, exit ***
  if (d <1 || d > daysinmonth[m])
    return;
  //*** If year less than 1, exit ***
  if (y < 1)
    return;
  //*** Assign date ***
  day = d;
  month = m;
  year = y;
}

// *** Sets date using days-into-year (diy) and year
↳(y) ***
void Date::SetDate(int diy, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = { 0, 31,28,31, 30,31,30,
  ↳31,31,30, 31,30,31 };
  //*** If leap year add one day to February ***
  daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0
  ↳&& y % 100 != 0);
  //*** Calculate month ***
  int remaining = diy;
  int m = 0;
  while (remaining > 0)
  {
    m++;
    remaining -= daysinmonth[m];
  }
  remaining += daysinmonth[m];
  //*** Set date ***
  SetDate(remaining, m, y);
}

// *** Returns the day of the week of a date ***
int Date::DayOfWeek()
{
  int M, modifiedyear, C, Y;
  //*** Calculate M ***
  M = (month + 9) % 12 + 1;
  //*** Calculate modified year ***
  modifiedyear = year - M / 11;
  //*** Calculate C ***
  C = modifiedyear / 100;
  //*** Calculate Y ***
  Y = modifiedyear % 100;
  //*** Calculate day of week ***
  int dayofweek = ((static_cast<int>(2.6 * M - 0.2)
  ↳+ day + Y + Y/4 + C/4 - 2*C)%7+7)%7;
  return dayofweek;
}


// *** Returns true if year is a leap year ***
bool Date::IsLeapYear()
{
  return ((year%400 == 0)||(year%4 == 0 &&
  ↳year%100 != 0));
}


int main()
{
  // *** Day names ***
  char daynames[7][10] =
  ↳{"Sunday","Monday","Tuesday","Wednesday",
  ↳"Thursday","Friday","Saturday"};

  Date d1;  // Date object

  // *** Set date ***
  d1.SetDate(60,2021);

  // *** Display date details ***
  cout << d1.day << '/' << d1.month << '/' <<
  ↳d1.year << " was a " << daynames[d1.DayOfWeek()]
  ↳<< endl;
}
```

Note that the final line of our new method uses a call to the original *SetDate()* to store the date. This makes sure only valid dates can be stored.

Activity 16.14

Modified code for *DateClass*:

```
#include <iostream>
using namespace std;

class Date
{
  public:
    int day;
    int month;
    int year;

    Date();
```

```
    void SetDate(int, int, int);
    void SetDate(int,int);
    int GetDayOfWeek();
    bool IsLeapYear();
};

//*************************************
//***       Date Class Methods        ***
//*************************************

//*** Initialise new Date objects to 1/1/2001 ***
Date::Date()
{
  day = 1;
  month = 1;
  year = 2001;
}

// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = {0, 31,28,31, 30,31,30,
  ↳31,31,30, 31,30,31};
  //*** If month invalid, exit ***
  if (m < 1 || m > 12)
    return;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += (y%400 == 0)||(y%4 == 0 &&
  ↳y%100 != 0);
  //*** If days in month invalid, exit ***
  if (d <1 || d > daysinmonth[m])
    return;
  //*** If year less than 1, exit ***
  if (y < 1)
    return;
  //*** Assign date ***
  day = d;
  month = m;
  year = y;
}

// *** Sets date using days-into-year (diy) and year
↳(y) ***
void Date::SetDate(int diy, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = { 0, 31,28,31, 30,31,30,
  ↳31,31,30, 31,30,31 };
  //*** If leap year add one day to February ***
  daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0
  ↳&& y % 100 != 0);
  //*** Calculate month ***
  int remaining = diy;
  int m = 0;
  while (remaining > 0)
  {
    m++;
    remaining -= daysinmonth[m];
  }
  remaining += daysinmonth[m];
  //*** Set date ***
  SetDate(remaining, m, y);
}



// *** Returns the day of the week of a date ***
int Date::DayOfWeek()
{
  int M, modifiedyear, C, Y;
  //*** Calculate M ***
  M = (month + 9) % 12 + 1;
  //*** Calculate modified year ***
  modifiedyear = year - M / 11;
  //*** Calculate C ***
  C = modifiedyear / 100;
  //*** Calculate Y ***
  Y = modifiedyear % 100;
  //*** Calculate day of week ***
  int dayofweek = ((static_cast<int>(2.6 * M - 0.2)
  ↳+ day + Y + Y/4 + C/4 - 2*C)%7+7)%7;
  return dayofweek;
}


// *** Returns true if year is a leap year ***
bool Date::IsLeapYear()
{
  return ((year%400 == 0)||(year%4 == 0 &&
  ↳year%100 != 0));
}

int main()
{
  // *** Day names ***
  char daynames[7][10] =
  ↳{"Sunday","Monday","Tuesday","Wednesday",
  ↳"Thursday","Friday","Saturday"};

  Date d1; // Date object

  // *** Set date ***
  d1.SetDate(29,2,2001);

  // *** Display date details ***
  cout << d1.day << '/' << d1.month << '/' <<
  ↳d1.year << " was a " << daynames[d1.DayOfWeek()]
  ↳<< endl;
}
```

Since all *Date* objects default to 1/1/2001 the moment they are created, when the attempt to assign an invalid date to `d1` fails, the object retains the date 1/1/2001 which is displayed and reported as being a Monday.

Activity 16.15

Modified code for *ImperialDistanceClass*:

```
#include <iostream>
using namespace std;

class ImperialDistance
{
  public:
    int yards;
    int feet;
    int inches;

    ImperialDistance();
    void SetDistance(int,int,int);
    double ConvertToMetric();
};

//*************************************
//*** ImperialDistance Class Methods ***
//*************************************

//*** Zeroises newly created distance objects ***
ImperialDistance::ImperialDistance()
{
  yards = 0;
  feet = 0;
  inches = 0;
}

// *** Sets the distance to y yards, f feet, and i
inches ***
void ImperialDistance::SetDistance(int y, int f,
↳int i)
{
  //*** If any parameter invalid, exit ***
  if (i < 0 || i > 11 || f < 0 || f > 2 || y < 0)
    return;
  //*** Assign distance ***
  yards = y;
  feet = f;
  inches = i;
}


// *** Returns equivalent distance in metres ***
double ImperialDistance::ConvertToMetric()
{
  return ((yards*36 + feet*12 + inches)*0.0254);
}



int main()
{
  ImperialDistance dist; // ImperialDistance object

  // *** Set distance ***
  dist.SetDistance(2,3,10);
```

```
    // *** Display distance ***
    cout << dist.yards << " yards  " << dist.feet
    ↳<< " feet  " << dist.inches << " inches "<<endl;
}
```

When the invalid parameters are used with *SetDistance()*, the object's value remains set to zero.

Activity 16.16

Modified code for *DateClass*:

```
#include <iostream>
using namespace std;

class Date
{
  public:
    int day;
    int month;
    int year;

    Date();
    Date(int,int,int);
    void SetDate(int, int, int);
    void SetDate(int,int);
    int GetDayOfWeek();
    bool IsLeapYear();
};

//*************************************
//***        Date Class Methods      ***
//*************************************

//*** Initialise new Date objects to 1/1/2001 ***
Date::Date()
{
  day = 1;
  month = 1;
  year = 2001;
}

//*** Initialise new Date objects to d/m/y ***
Date::Date(int d, int m, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = {0, 31,28,31, 30,31,30,
  ↳31,31,30,31,30,31};

  bool valid = true;  // Parameters valid

  //*** If month invalid, invalid parameters ***
  if (m < 1 || m > 12)
    valid = false;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += (y%400 == 0)||(y%4 == 0 &&
  ↳y%100 != 0);
  //*** If days invalid, invalid parameters ***
  if (d <1 || d > daysinmonth[m])
    valid = false;
  //*** If year less than 1, invalid parameters ***
  if (y < 1)
    valid = false;
  //*** Assign date ***
  if (valid)
  {
    day = d;
    month = m;
    year = y;
  }
  else
  {
    day = 1;
    month = 1;
    year = 2001;
  }
}


// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = {0, 31,28,31, 30,31,30,
  ↳31,31,30, 31,30,31};
  //*** If month invalid, exit ***
  if (m < 1 || m > 12)
    return;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += (y%400 == 0)||(y%4 == 0 &&
  ↳y%100 != 0);
  //*** If days in month invalid, exit ***
  if (d <1 || d > daysinmonth[m])
    return;
  //*** If year less than 1, exit ***
  if (y < 1)
    return;
  //*** Assign date ***
  day = d;
  month = m;
  year = y;
}


// *** Sets date using days-into-year (diy) and year
↳(y) ***
void Date::SetDate(int diy, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = { 0, 31,28,31, 30,31,30,
  ↳31,31,30, 31,30,31 };
  //*** If leap year add one day to February ***
  daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0
  ↳&& y % 100 != 0);
  //*** Calculate month ***
  int remaining = diy;
  int m = 0;
  while (remaining > 0)
  {
    m++;
    remaining -= daysinmonth[m];
  }
  remaining += daysinmonth[m];
  //*** Set date ***
  SetDate(remaining, m, y);
}


// *** Returns the day of the week of a date ***
int Date::DayOfWeek()
{
  int M, modifiedyear, C, Y;
  //*** Calculate M ***
  M = (month + 9) % 12 + 1;
  //*** Calculate modified year ***
  modifiedyear = year - M / 11;
  //*** Calculate C ***
  C = modifiedyear / 100;
  //*** Calculate Y ***
  Y = modifiedyear % 100;
  //*** Calculate day of week ***
  int dayofweek = ((static_cast<int>(2.6 * M - 0.2)
  ↳+ day + Y + Y/4 + C/4 - 2*C)%7+7)%7;
  return dayofweek;
}

// *** Returns true if year is a leap year ***
bool Date::IsLeapYear()
{
  return ((year%400 == 0)||(year%4 == 0 &&
  ↳year%100 != 0));
}



int main()
{
  // *** Day names ***
  char daynames[7][10] =
  ↳{"Sunday","Monday","Tuesday",
  ↳"Wednesday","Thursday","Friday","Saturday"};

  Date d1(23,11,1963); // Date object

  // *** Display date details ***
  cout << d1.day << '/' << d1.month << '/' <<
  ↳d1.year << " was a " << daynames[d1.DayOfWeek()]
  ↳<< endl;
}
```

Activity 16.17

In *DateClass*, the class declaration needs to be changed to:

```
class Date
{
  public:
```

```
    int day;
    int month;
    int year;

    Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int,int);
    int GetDayOfWeek();
    bool IsLeapYear();
};
```

Note that the zero-argument constructor has been removed.

In `main()`, the parameters need to be removed when declaring *d1*:

```
int main()
{
  // *** Day names ***
  char daynames[7][10] =
  ↳{"Sunday","Monday","Tuesday",
  ↳"Wednesday","Thursday","Friday","Saturday"};

  Date d1;  // Date object

  // *** Display date details ***
  cout << d1.day << '/' << d1.month << '/' <<
  ↳d1.year << " was a " << daynames[d1.DayOfWeek()]
  ↳<< endl;
}
```

Without a line to change the value in *d1*, the date displayed is 1/1/2001.

Activity 16.18

Modifications to *ImperialDistanceClass*:

Class declaration:

```
class ImperialDistance
{
public:
  int yards;
  int feet;
  int inches;

  ImperialDistance(int=0, int=0, int=0);
  void SetDistance(int, int, int);
  double ConvertToMetric();
};
```

Code for new constructor:

```
//*** Initialises newly created distance objects ***
ImperialDistance::ImperialDistance(int y, int f, int
i)
{
  bool valid{ true };
  //*** If yards invalid, invalid parameters ***
  if (y < 0 )
    valid = false;

  //*** If feet invalid, invalid parameters ***
  if (f <0 || f > 2)
    valid = false;
  //*** If inches invalid,  invalid parameters ***
  if (i < 0 || i > 11)
    valid = false;
  //*** Assign distance ***
  if (valid)
  {
    yards = y;
    feet  = f;
    inches = i;
  }
  else
  {
    yards = 0;
    feet = 0;
    inches = 0;
  }
}
```

Modified code for *main()*:

```
int main()
{

  ImperialDistance dist(2,3,10); // ImperialDistance
                                 ↳object

  // *** Display distance ***
  cout << dist.yards << " yards  " << dist.feet
  << " feet  " << dist.inches << " inches " << endl;
}
```

When an invalid distance is attempted, the distance remains set at zero.

Activity 16.19

Code for *DateClass*'s *main()*:

```
int main()
{
  Date d1(23,1,2001);

  //*** Create new Date object containing same date
  ↳***
  Date d2(d1);

  //*** Display value of new Date object ***
  cout << d2.day << '/' << d2.month << '/'
  ↳<< d2.year << endl;
}
```

Activity 16.20

Code for *DateClass*'s *main()*:

```
int main()
{
  Date d1(23,1,2001);

  //*** Create new Date object containing same date
  ↳***
  Date d2(d1);

  d2 = 21;

  //*** Display value of new Date object ***
  cout << d2.day << '/' << d2.month << '/'
  ↳<< d2.year << endl;
}
```

The value displayed is 21/1/2001. The *month* and *year* values are taken from the constructor's defaults.

Activity 16.21

Modified code for *DateClass* declaration:

```
class Date
{
  public:
    int day;
    int month;
    int year;

    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
};
```

This version creates a compilation error since 21 cannot be automatically converted to a *Date* object.

Modified code for *main()* :

```
int main()

  {
    Date d1(23,1,2001);

    //*** Create new Date object containing same date
    ↳***
    Date d2(d1);

    d2 = static_cast<Date>(21);
```

```
    //*** Display value of new Date object ***
    cout << d2.day << '/' << d2.month << '/'
    ↳<< d2.year << endl;
}
```

This version compiles and runs successfully.

Activity 16.22

The class declaration in *DateClass* should be changed to

```
class Date
{
  public:
    int day;
    int month;
    int year;

    explicit Date(int=1,int=1,int=2001);
    ~Date();
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
};
```

and the code for the new destructor added:

```
Date::~Date()
{
  cout << "Date object deleted\n";
};
```

The message *Date object deleted* should be displayed when the program is run.

Activity 16.23

The new code for the *Date* declaration is:

```
class Date
{
  private:
    int day;
    int month;
    int year;
  public:
    explicit Date(int=1,int=1,int=2001);
    ~Date();
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
};
```

No other changes are required.

The compilation fails. Error messages from the compiler should be of the form:

'Date::day' : cannot access private member declared in class 'Date'

Activity 16.24

The class declaration for *Date* becomes:

```
class Date
{
  private:
    int day;
    int month;
    int year;
  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
};
```

The code for the new methods is:

```
//*** Returns the value in attribute day ***
int Date::GetDay()
{
  return day;
}

//*** Returns the value in attribute month ***
int Date::GetMonth()
{
  return month;
}

//*** Returns the value in attribute year ***
int Date::GetYear()
{
  return year;
}
```

Modified code for *main()*:

```
int main()
{
  Date d1(23,1,2001);

  //*** Create new Date object containing same date
  ↳***
  Date d2(d1);

  d2 = static_cast<Date>(21);

  //*** Display value of Date object ***
  cout << d2.GetDay() << '/' << d2.GetMonth() << '/'
  ↳<< d2.GetYear() << endl;
}
```

Activity 16.25

The class declaration for *Date* becomes:

```
class Date
{
  private:
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN();

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int DayOfWeek();
    bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
};
```

The code for the new methods is:

```
// *** Returns the JDN of the date ***
long Date::ToJDN()
{
  int a = (14 - month)/12;
  int m = month + 12*a - 3;
  int y = year + 4800 - a;
  long result = day + (153*m + 2) / 5 + 365*y + y/4
  ↳- y/100 + y/400  - 32045;
  return result;
}


// *** Sets date to eqivalent of JDN ***
Date Date::FromJDN(long jdn)
{
   int f = jdn + 1401+(((4 * jdn + 274277)/
  ↳146097)*3)/4 -38;
   int e = 4 * f + 3;
   int g = (e % 1461)/4;
   int h = 5 * g + 2;
   Date d;
   d.day = (h % 153)/5 + 1;
   d.month = (h/153 + 2) % 12 + 1;
   d.year = e/1461 - 4716 + (12 + 2 - d.month)/12;
   return d;
}
```

Activity 16.26

The new code for the *Date* declaration is:

```
class Date
{
  private:
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
};
```

The code for the new method is:

```
//*** Add d days to date ***
Date Date::operator+(int d)
{
  Date dt = FromJDN(ToJDN()+d);
  return dt;
}
```

The code for `main()` is:

```
int main()
{

  Date d1(30,5,2023);

  //*** Add 7 days ***
  d1=d1+7;

  //*** Display date ***
  cout << d1.GetDay() << '/' << d1.GetMonth() <<
  ↳'/' << d1.GetYear() << endl;
}
```

Activity 16.27

The new code for the *Date* declaration is:

```
class Date
{
  private:
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
};
```

The code for the three new methods is:

```
//*** Returns a Date calculated by subtracting d
days to date ***
Date Date::operator-(int d)
{
  Date dt = FromJDN(ToJDN() - d);
  return dt;
}


//*** Returns the difference in days between two
↳dates ***
int Date::operator-(Date dt)
{
  int result = ToJDN() - dt.ToJDN();
  return result;
}


//*** Returns true if two dates are equal ***
bool Date::operator==(Date dt)
{
  return (ToJDN() == dt.ToJDN());
}

//*** Returns true if two dates are not equal ***
bool Date::operator!=(const Date dt) const
{
  return (ToJDN() != dt.ToJDN());
}
```

One possible version of `main()` used to test these methods is:

```
int main()
{
  Date d1(30,5,2013);

  // *** New date 7 days before first date ***
  Date d2= d1 - 7;

  //*** Test if dates are equal ***
  if (d1 == d2)
    cout << "The dates are the same\n";
  if (d1 != d2)
  {
    cout << "The dates are different\n";
    cout << "There are " << d1 - d2 <<
    ↳" days between the two dates\n";
  } //*** Display dates ***
  cout << "d1: "<< d1.GetDay() << '/' <<
  ↳d1.GetMonth() << '/' << d1.GetYear() << endl;
  cout << "d2: "<< d2.GetDay() << '/' <<
  ↳d2.GetMonth() << '/' << d2.GetYear() << endl;
}
```

Of course, we could have used `else` rather than a second *if* statement in the code given above – but that would not test the `!=` operator.

Activity 16.28

The final version of *TestCallOperator*:

```
#include <iostream>
using namespace std;

class A
{
  public:
    A() { cout << "Constructor called\n"; };
    int operator()(int, int);
};

int A::operator()(int v, int w)
{
  return (v * w);
}

int main()
{
  A test;
  cout << test(3,5) << endl;
}
```

Activity 16.29

The new *Date* class declaration is:

```
class Date
{
  private:
```

```
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
};
```

The code for the new method is:

```
//*** Adds a given number of days to the date ***
void Date::operator+=(int d)
{
  *this = FromJDN(ToJDN()+d);
}
```

To test the new method, `main()` is rewritten as:

```
int main()
{
  Date d1(23,12,1980);

  //*** Add 10 days ***
  d1 += 10;

  //*** Display date ***
  cout << d1.GetDay() << '/' << d1.GetMonth() << '/'
  ↳<< d1.GetYear() << endl;
}
```

Activity 16.30

Modified coded for *IsLeapYear()*:

```
// *** Returns true if year is a leap year ***
bool Date::IsLeapYear()
{
  return ((this->year%400 == 0)||(this->year%4 == 0
  ↳&& this->year%100 != 0));
}
```

Activity 16.31

The new *Date* class declaration is:

```
class Date
{
  private:
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
};
```

The code for the new methods is:

```
//*** Adds one day to date, returns new date ***
Date Date::operator++()
{
  *this = FromJDN(ToJDN()+1);
  return *this;
}

//*** Adds one day to date, returns original date
↳***
Date Date::operator++(int)
{
  Date result = *this;
  *this = FromJDN(ToJDN()+1);
  return result;
}

//*** Subtracts one day from date, returns new date
↳***
Date Date::operator--()
{
  *this = FromJDN(ToJDN()-1);
  return *this;
}

//*** Subtracts one day from date, returns original
↳date ***
Date Date::operator--(int)
{
  Date result = *this;
  *this = FromJDN(ToJDN()-1);
  return result;
}
```

The code for `main()` is:

```
int main()
{
  Date d1(31, 12, 2000);

  //*** Add a day (prefix) ***
  Date d2 = ++d1;
  //*** Display dates ***
  cout << "PREFIX++\n";
  cout << "d1="<< d1.GetDay() << '/'
  ↳<< d1.GetMonth() << '/' << d1.GetYear() << endl;
  cout << "d2="<< d2.GetDay() << '/'
  ↳<< d2.GetMonth() << '/' << d2.GetYear() << endl;

  //*** Add a day (postfix) ***
  d1 = Date(31, 12, 2000);
  d2 = d1++;
  //*** Display dates ***
  cout << "POSTFIX++\n";
  cout <<"d1=" << d1.GetDay() << '/'
  ↳<< d1.GetMonth() << '/' << d1.GetYear() << endl;
  cout <<"d2=" << d2.GetDay() << '/'
  ↳<< d2.GetMonth() << '/' << d2.GetYear() << endl;

  //*** Subtract a day (prefix) ***
  d1 = Date(31, 12, 2000);
  d2 = --d1;
  //*** Display dates ***
  cout << "PREFIX--\n";
  cout << "d1=" << d1.GetDay() << '/'
  ↳<< d1.GetMonth() << '/'<< d1.GetYear() << endl;
  cout << "d2=" << d2.GetDay() << '/'
  ↳<< d2.GetMonth() << '/'<< d2.GetYear() << endl;

  //*** Subtract a day (postfix) ***
  d1 = Date(31, 12, 2000);
  d2 = d1--;
  //*** Display dates ***
  cout << "POSTFIX--\n";
  cout << "d1=" << d1.GetDay() << '/'
  ↳<< d1.GetMonth() << '/' << d1.GetYear() << endl;
  cout << "d2=" << d2.GetDay() << '/'
  ↳<< d2.GetMonth() << '/' << d2.GetYear() << endl;
}
```

Activity 16.32

When you attempt to compile the code, an error message such as

"Date::operator=(const Date &)" (declared implicitly) cannot be referenced -- it is a deleted function

Activity 16.33

Modified code for *Date* class:

```
class Date
{
  private:
    const int DAYSINYEAR = 365;
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:

    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    inline bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date)
    bool operator!=(Date);;
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    Date& operator=(const Date&);
    int GetDaysInYear();
};


//*** Initialises new date object to d/m/y ***
Date::Date(int d, int m, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = {0, 31,28,31, 30,31,30,
  ↳31,31,30,31,30,31};

  bool valid = true;  // Parameters valid

  //*** If month invalid, invalid parameters ***
  if (m < 1 || m > 12)
    valid = false;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += (y%400 == 0)||(y%4 == 0 &&
  ↳y%100 != 0);
  //*** If days in month invalid, invalid parameters
  ↳***
  if (d <1 || d > daysinmonth[m])
    valid = false;
  //*** If year less than 1, invalid parameters ***
  if (y < 1)
    valid = false;
  //*** Assign date ***
  if (valid)
  {
    day = d;
    month = m;
    year = y;
  }
  else
  {
    day = 1;
    month = 1;
    year = 2001;
  }
}

//*** Copies date d to date ***
Date& Date::operator=(const Date& d)
{
  day = d.day;
  month = d.month;
  year = d.year;
  return *this;
}

//*** Returns number of days in year ***
int Date::GetDaysInYear()
{
  return DAYSINYEAR + IsLeapYear();
}
```

Modified code for *main()*:

```
int main()
{
  Date d1(31, 12, 2000);
  cout << d1.GetDay() << '/' << d1.GetMonth() << '/'
  ↳<< d1.GetYear() << endl;
  cout << "That year has " << d1.GetDaysInYear()
  ↳<< " days\n";

}
```

Activity 16.34

Modified code for *Date* class:

```
class Date
{
  private:
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    static const int DAYSINYEAR = 365;
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    inline bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    int GetDaysInYear();
};
```

`main()` is coded as:

```
int main()
{
  cout << "DAYSINYEAR value is " << Date::DAYSINYEAR
  ↳<< endl;
  Date d1;
  cout << "DAYSINYEAR value is " << d1.DAYSINYEAR
  ↳<< endl;
}
```

Activity 16.35

Modified code for *Date*:

```
class Date
{
  private:
    static const int DAYSINYEAR = 365;
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
```

```
    int GetDayOfWeek();
    inline bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    int GetDaysInYear();
    int GetDAYSINYEARconst();
};
```

New method's code:

```
//*** Returns the value of DAYSINYEAR ***
int Date::GetDAYSINYEARconst()
{
  return DAYSINYEAR;
}
```

Although the first version of `main()`,

```
int main()
{
  Date d1;
  cout << "DAYSINYEAR value is " <<
  ↳d1.GetDAYSINYEARconst() << endl;
}
```

operates correctly, using the expression

```
Date::GetDAYSINYEARconst()
```

will not compile. You can only use the class name in accessing a property of a class if that property (in this case the *GetDAYSINYEARconst()*) is defined as static.

Activity 16.36

Modified code for *Date*:

```
class Date
{
  private:
    static const int DAYSINYEAR = 365;
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int DayOfWeek();
    inline bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    int GetDaysInYear();
    static int GetDAYSINYEARconst();
};
```

The program will now compile when using the term `Date::GetDAYSINYEARconst()`.

Activity 16.37

Modified code for *Date*:

```
//*** Class declaration ***
class Date
{
  private:
    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    static const int DAYSINYEAR = 365;
    static int count;
    explicit Date(int = 1, int = 1, int = 2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    int GetDaysInYear();
    static int GetDAYSINYEARconst();
    static int GetCount();
    ~Date();
};
int Date::count = 0;
```

Modified code for *Date* constructor:

```
//*** Initialise new Date objects to d/m/y ***
Date::Date(int d, int m, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = { 0, 31,28,31, 30,31,30,31,
  ↳31,30,31,30,31 };

  bool valid = true;  // Parameters valid

  //*** If month invalid, invalid parameters ***
  if (m < 1 || m > 12)
    valid = false;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0
  ↳&& y % 100 != 0);
  //*** If days invalid, invalid parameters ***
  if (d <1 || d > daysinmonth[m])
    valid = false;
  //*** If year less than 1, invalid parameters ***
  if (y < 1)
    valid = false;
  //*** Assign date ***
  if (valid)
  {
    day = d;
    month = m;
    year = y;
  }
  else
  {
    day = 1;
    month = 1;
    year = 2001;
  }
  //*** Add one to count of Date objects **
  count++;
}
```

New methods' code:

```
//*** Returns the number of Date object existing ***
int Date::GetCount()
{
  return count;
}
```

```
//*** Destructor decrements count ***
Date::~Date()
{
  //*** Decrement count of Date objects ***
  count--;
}
```

Code for *main()*:

```
int main()
{
  cout << "There are " << Date::GetCount()
  ↳<< " Date objects\n";
  Date dt1(23, 11, 1963);
  cout << "There are " << Date::GetCount()
  ↳<< " Date objects\n";
  {
    Date dt2;
    cout << "There are " << Date::GetCount()
    ↳<< " Date objects\n";
  }
  cout << "There are " << Date::GetCount()
  ↳<< " Date objects\n";
}
```

The program should display the following lines:

There are 0 Date objects
There are 1 Date objects
There are 2 Date objects
There are 1 Date objects

Activity 16.38

Modified code for *Date*:

```
class Date
{
  private:
    static const int DAYSINYEAR = 365;
    static int count;

    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    inline bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    int GetDaysInYear();
    static int GetDAYSINYEARconst();
    static int GetCount();
    Date::~Date();

    friend ostream& operator<<(ostream&, Date&);
};
int Date::count = 0;
```

Code for *operator<<*:

```
//*** Displays a Date ***
ostream& operator<<(ostream& ct, Date& dt)
{
  ct << dt.day <<'/' << dt.month << '/' << dt.year;
  return ct;
}
```

Code for *main()*:

```
int main()
{
  Date d1(23,11,1963);
  cout << d1 << endl;
}
```

Activity 16.39

Modified code for *Date*:

```
class Date
{
  private:
    static const int DAYSINYEAR = 365;
    static int count;

    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    inline bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    int GetDaysInYear();
    static int GetDAYSINYEARConst();
    static int GetCount();
    Date::~Date();

    friend ostream& operator<<(ostream&, Date&);
    friend istream& operator>>(istream&, Date&);

};
int Date::count = 0;
```

Code for *operator>>*:

```
//*** Accepts a Date value from the keyboard ***
istream& operator>>(istream& cn, Date& dt)
{
  scanf("%d/%d/%d",&dt.day, &dt.month, &dt.year);
  return cn;
}
```

Code for *main()*:

```
int main()
{
  Date d1;
  cout << "Enter date : ";
  cin >> d1;
  cout << d1 << endl;
}
```

Activity 16.40

Modified code for *operator>>*:

```
//*** Accepts a Date value from the keyboard ***
istream& operator>>(istream& cn, Date& dt)
{
  int d,m,y;
  //*** Read date ***
  scanf("%d/%d/%d",&d, &m, &y);
  //*** Transfer values to date object ***
  dt.SetDate(d,m,y);
  return cn;
}
```

Activity 16.41

Modified code for *DateClass*:

```
class Date
{
  private:
    static const int DAYSINYEAR = 365;
    static int count;

    int day;
    int month;
    int year;

    long ToJDN();
    Date FromJDN(long);

  public:
    explicit Date(int=1,int=1,int=2001);
    void SetDate(int, int, int);
    void SetDate(int, int);
    int GetDayOfWeek();
    inline bool IsLeapYear();
    int GetDay();
    int GetMonth();
    int GetYear();
    Date operator+(int);
    int operator-(Date);
    Date operator-(int);
    bool operator==(Date);
    bool operator!=(Date);
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    int GetDaysInYear();
    static int GetDAYSINYEARConst();
    static int GetCount();
    Date::~Date();

    friend ostream& operator<<(ostream&, Date&);
    friend istream& operator>>(istream&, Date&);
    friend Date operator+(int, Date&);
};
int Date::count = 0;
```

Code for *operator+*:

```
Date operator+(int days, Date& d)
{
  Date dt = d.FromJDN(d.ToJDN() + days);
  return dt;
}
```

Test code in *main()*:

```
int main()
{
  Date d1(23,11,1963);
  Date d2;
  d2 = d1 + 10;
  cout << d2 << endl;
}
```

Activity 16.42

Code for *StudentClass*:

```
#include <iostream>
using namespace std;

class Student
{
  private:
    char name[21]{ "" };
    char sex{ '?' };
    bool attending{ true };
    int marks[6]{ 0 };
  public:
    Student(const char*, char, bool = true);
    void SetName(char*);
    void SetSex(char);
    void SetAttending(bool);
    void SetMark(int, int);
    char* GetName();
    char GetSex();
    bool GetAttending();
    int GetMark(int);
    double GetAverageMark();
    friend ostream& operator<<(ostream&, Student&);
};

Student::Student(const char* n, char s, bool at)
{
  strcpy_s(name,strlen(n)+1, n);
  s = toupper(s);
  if (s == 'M' || s == 'F')
    sex = s;
  attending = at;
}

//*** Set the student's name to n ***
void Student::SetName(char * n)
{
  strcpy_s(name,strlen(n)+1,n);
}

//*** Set student's sex to s ***
void Student::SetSex(char s)
{
  s = toupper(s);
  if (s == 'M' || s == 'F')
    sex = s;
}

//*** Set student's attendence status ***
void Student::SetAttending(bool at)
{
  attending = at;
}

//*** Set student's mark for specified exam ***
void Student::SetMark(int mrk, int ex)
{
  //*** If not valid exam, return ***
  if (ex < 0 || ex > 5)
    return;
  //*** If not valid mark, return ***
  if (mrk < 0 || mrk > 100)
    return;
  //*** Store mark ***
  marks[ex] = mrk;
}

//*** Return student's name ***
char* Student::GetName()
{
  return name;
}

//*** Return student's sex ***
char Student::GetSex()
{
  return sex;
}

//*** Return student's attendence status ***
bool Student::GetAttending()
{
  return attending;
}

//*** Return student's exam mark ***
int Student::GetMark(int ex)
{
  if (ex < 0 || ex > 5)
    return -1;
  return marks[ex];
}

//*** Return student's average mark ***
double Student::GetAverageMark()
{
  double total{ 0.0 };
  for (auto v : marks)
    total += v;
  return total / 6.0;
}


//*** Display contents of student object ***
ostream& operator<<(ostream& ct, Student& st)
{
  ct << st.name << "  " << st.sex << "  ";
  if (st.attending)
    ct << "attending  ";
  else
    ct << "not attending  ";
```

```
   for (int ex = 0; ex < 6; ex++)
      ct << st.marks[ex] << "  ";
   ct << st.GetAverageMark() << endl;
   return ct;
}


int main()
{
   Student st1("Madeline Bray", 'F');
   Student st2("Nicholas Nickleby",'M');
   Student st3("Ada Clare", 'F', false);
   Student st4("Richard Carstone", 'M');
   //*** Assign random marks ***
   for (int ex = 0; ex < 6; ex++)
      st1.SetMark(rand() % 101, ex);
   for (int ex = 0; ex < 6; ex++)
      st2.SetMark(rand() % 101, ex);
   for (int ex = 0; ex < 6; ex++)
      st3.SetMark(rand() % 101, ex);
   for (int ex = 0; ex < 6; ex++)
      st4.SetMark(rand() % 101, ex);
   //*** Display object contents ***
   cout << st1 << endl;
   cout << st2 << endl;
   cout << st3 << endl;
   cout << st4 << endl;
}
```

Activity 16.43

Modified code in *StudentClass*:

Changes to *Student* class declaration:

```
class Student
{
  private:
     char* name{ nullptr };
     char sex{ '?' };
     bool attending{ true };
     int marks[6]{ 0 };
  public:
     Student(const char*, char, bool = true);
     void SetName(char*);
     void SetSex(char);
     void SetAttending(bool);
     void SetMark(int, int);
     char* GetName();
     char GetSex();
     bool GetAttending();
     int GetMark(int);
     double GetAverageMark();
     ~Student();

     friend ostream& operator<<(ostream&, Student&);
};
```

Changes to *Student* constructor:

```
Student::Student(const char* n, char s, bool at)
{
   name = new char[strlen(n) + 1];
   strcpy_s(name,strlen(n)+1,n);
   s = toupper(s);
   if (s == 'M' || s == 'F')
      sex = s;
   attending = at;
}
```

Changes to *Student SetName()*:

```
//*** Set the student's name to n ***
void Student::SetName(char * n)
{
   //*** If no new name, exit ***
   if (n == nullptr)
      return;
   //*** Delete any space already being used ***
   if (name != nullptr)
      delete[]name;
   //*** Create enough space for the new name ***
   name = new char[strlen(n) + 1];
   //*** Copy name into this new space ***
   strcpy_s(n,strlen(n)+1,name);
}
```

New code for *Student* destructor:

```
Student::~Student()
{
   if(name != nullptr)
      delete [] name;
}
```

Activity 16.44

The program is likely to crash or show unpredictable results.

Activity 16.45

Modified code in *StudentClass*:

Changes to *Student* class declaration:

```
class Student
{
  private:
     char* name{ nullptr };
     char sex{ '?' };
     bool attending{ true };
     int marks[6]{ 0 };
  public:
     Student(const char*, char, bool = true);
     void SetName(const char*);
     void SetSex(char);
     void SetAttending(bool);
     void SetMark(int, int);
     char* GetName();
     char GetSex();
     bool GetAttending();
     int GetMark(int);
     double GetAverageMark();
     Student& operator=(Student&);
     ~Student();

     friend ostream& operator<<(ostream&, Student&);
};
```

New code for *Student operator==()*:

```
//*** Copies a Details object to current Details
object ***
Student& Student::operator=(Student& st)
{
   //*** Delete any space already being used ***
   if (name != nullptr)
      delete[]name;
   //*** Set up space for copy of name ***
   int size = strlen(st.name) + 1;
   name = new char[size];
   //*** Copy name ***
   strcpy_s(name,size,st.name);
   //*** Copy sex ***
   sex = st.sex;
   //*** Copy attending ***
   attending = st.attending;
   //*** Copy marks ***
   for (int c = 0; c < 6; c++)
      marks[c] = st.marks[c];
   //*** Return reference to updated object ***
   return *this;
}
```

The program should operate correctly this time.

Activity 16.46

Modified code in *StudentClass*:

Changes to *Student* class declaration:

```
class Student
{
  private:
     char* name{ nullptr };
     char sex{ '?' };
     bool attending{ true };
     int marks[6]{ 0 };
  public:
     Student(const char*, char, bool = true);
     Student(const Student&);
     void SetName(const char*);
     void SetSex(char);
     void SetAttending(bool);
```

```
    void SetMark(int, int);
    char* GetName();
    char GetSex();
    bool GetAttending();
    int GetMark(int);
    double GetAverageMark();
    Student& operator=(Student&);
    ~Student();

    friend ostream& operator<<(ostream&, Student&);
};
```

Code for copy constructor:

```
Student::Student(const Student& st)
{
  //*** Delete any space already being used ***
  if (name != nullptr)
    delete[]name;
  //*** Set up space for copy of name ***
  int size = strlen(st.name) + 1;
  name = new char[size];
  //*** Copy name ***
  strcpy_s(name,size,st.name);
  //*** Copy sex ***
  sex = st.sex;
  //*** Copy attending ***
  attending = st.attending;
  //*** Copy marks ***
  for (int c = 0; c < 6; c++)
    marks[c] = st.marks[c];
}
```

Modified code for *main()*:

```
int main()
{
  Student st1("Madeline Bray", 'F');
  Student st2("Nicholas Nickleby",'M');
  Student st3("Ada Clare", 'F', false);
  Student st4("Richard Carstone", 'M');
  //*** Assign random marks ***
  for (int ex = 0; ex < 6; ex++)
    st1.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st2.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st3.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st4.SetMark(rand() % 101, ex);
  //*** Create new object equal to st1 ***
  Student st5(st1);
  //*** Display contents of both objects ***
  cout << st1 << endl;
  cout << st5 << endl;
}
```

Activity 16.47

Modified code in *StudentClass*:

Changes to prototype of *operator=()*:

```
Student& operator=(const Student&);
```

Changes to the code for *operator=()*:

```
Student& operator=(const Student& st)
```

Activity 16.48

Modified code in *StudentClass*:

Prototype for *GetNameSize()*:

```
//*** Standard function prototypes ***
int GetNameSize(const Student&);
```

The above is placed after the code for all operations of the *Student* class.

Code for *GetNameSize()*:

```
//************************************
//***      Standard functions      ***
//************************************

//*** Returns number of characters in st.name ***
int GetNameSize(const Student& st)
{
  return strlen(st.GetName());
}
```

The above is placed after the code for `main()`.

Code for *main()*:

```
int main()
{
  Student st1("Madeline Bray", 'F');
  Student st2("Nicholas Nickleby",'M');
  Student st3("Ada Clare", 'F', false);
  Student st4("Richard Carstone", 'M');
  //*** Assign random marks ***
  for (int ex = 0; ex < 6; ex++)
    st1.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st2.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st3.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st4.SetMark(rand() % 101, ex);
  //*** Display number of chars in st1's name ***
  cout << st1.GetName() << " contains "
  ↳<< GetNameSize(st1) << " characters\n";
}
```

The code fails to compile because the parameter to *GetNameSize()* is defined as a constant.

Activity 16.49

Modified code in *StudentClass*:

```
#include <iostream>
using namespace std;

class Student
{
  private:
    char* name{ nullptr };
    char sex{ '?' };
    bool attending{ true };
    int marks[6]{ 0 };
  public:
    Student(const char*, char, bool = true);
    Student(const Student&);
    void SetName(const char*);
    void SetSex(char);
    void SetAttending(bool);
    void SetMark(int, int);
    char* GetName() const;
    char GetSex() const;
    bool GetAttending() const;
    int GetMark(int) const ;
    double GetAverageMark() const;
    Student& operator=(const Student&);
    ~Student();

    friend ostream& operator<<(ostream&, Student&);
};

Student::Student(const char* n, char s, bool at)
{
  name = new char[strlen(n) + 1]; //Extra byte for
                                  ↳final null
  strcpy_s(name,strlen(n)+1,n);
  s = toupper(s);
  if (s == 'M' || s == 'F')
    sex = s;
  attending = at;
}

Student::Student(const Student& st)
{
  //*** Delete any space already being used ***
  if (name != nullptr)
    delete[]name;
  //*** Set up space for copy of name ***
  int size = strlen(st.name) + 1;
  name = new char[size];
  //*** Copy name ***
  strcpy_s(name,size,st.name);
  //*** Copy sex ***
  sex = st.sex;
  //*** Copy attending ***
```

```
  attending = st.attending;
  //*** Copy marks ***
  for (int c = 0; c < 6; c++)
    marks[c] = st.marks[c];
}

//*** Set the student's name to n ***
void Student::SetName(const char * n)
{
  //*** If no new name, exit ***
  if (n == nullptr)
    return;
  //*** Delete any space already being used ***
  if (name != nullptr)
    delete[]name;
  //*** Create enough space for the new name ***
  name = new char[strlen(n) + 1]; //Extra byte for
                                  ↳final null
  //*** Copy n into this new space ***
  strcpy_s(name,strlen(n)+1,n);

}

//*** Set student's sex to s ***
void Student::SetSex(char s)
{
  s = toupper(s);
  if (s == 'M' || s == 'F')
    sex = s;
}

//*** Set student's attendence status ***
void Student::SetAttending(bool at)
{
  attending = at;
}

//*** Set student's mark for specified exam ***
void Student::SetMark(int mrk, int ex)
{
  //*** If not valid exam, return ***
  if (ex < 0 || ex > 5)
    return;
  //*** If not valid mark, return ***
  if (mrk < 0 || mrk > 100)
    return;
  //*** Store mark ***
  marks[ex] = mrk;
}

//*** Return student's name ***
char* Student::GetName() const
{
  return name;
}

//*** Return student's sex ***
char Student::GetSex() const
{
  return sex;
}

//*** Return student's attendence status ***
bool Student::GetAttending() const
{
  return attending;
}

//*** Return student's exam mark ***
int Student::GetMark(int ex) const
{
  if (ex < 0 || ex > 5)
    return -1;
  return marks[ex];
}

//*** Return student's average mark ***
double Student::GetAverageMark() const
{
  double total{ 0.0 };
  for (auto v : marks)
    total += v;
  return total / 6.0;
}

//*** Copies a Details object to current Details
object ***
Student& Student::operator=(const Student& st)
{
  //*** Delete any space already being used ***
  if (name != nullptr)
    delete[]name;
  //*** Set up space for copy of name ***
  int size = strlen(st.name) + 1;
  name = new char[size];
  //*** Copy name ***
  strcpy_s(name,size,st.name);
  //*** Copy sex ***
  sex = st.sex;
  //*** Copy attending ***
  attending = st.attending;
  //*** Copy marks ***
  for (int c = 0; c < 6; c++)
    marks[c] = st.marks[c];
  //*** Return reference to updated object ***
  return *this;
}


Student::~Student()
{
  if (name != nullptr)
    delete[] name;
}


//*** Display contents of student object ***
ostream& operator<<(ostream& ct, Student& st)
{
  ct << st.name << "  " << st.sex << "  ";
  if (st.attending)
    ct << "attending  ";
  else
    ct << "not attending  ";
  for (int ex = 0; ex < 6; ex++)
    ct << st.marks[ex] << "  ";
  ct << st.GetAverageMark() << endl;
  return ct;
}

//*** Standard function prototypes ***
int GetNameSize(const Student&);


int main()
{
  Student st1("Madeline Bray", 'F');
  Student st2("Nicholas Nickleby",'M');
  Student st3("Ada Clare", 'F', false);
  Student st4("Richard Carstone", 'M');
  //*** Assign random marks ***
  for (int ex = 0; ex < 6; ex++)
    st1.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st2.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st3.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st4.SetMark(rand() % 101, ex);
  //*** Display number of chars in st1's name ***
  cout << st1.GetName() << " contains "
  ↳<< GetNameSize(st1) << " characters\n";
}


//*** Returns number of characters in st.name ***
int GetNameSize(const Student& st)
{
  return strlen(st.GetName());
}
```

Activity 16.50

The compiler objects to the code trying to return a `char*` value from *Trivial*::*GetName()*.

Activity 16.51

The program now works as expected.

Activity 16.52

Changes to *StudentClass*:

Modified code for *main()*:

```
int main()
{
  Student *st1{ new Student("Madeline Bray", 'F') };
  Student * st2{ new Student("Nicholas Nickleby",
  'M') };
  Student st3("Ada Clare", 'F', false);
  Student st4("Richard Carstone", 'M');
  //*** Assign random marks ***
  for (int ex = 0; ex < 6; ex++)
    st1->SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st2->SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st3.SetMark(rand() % 101, ex);
  for (int ex = 0; ex < 6; ex++)
    st4.SetMark(rand() % 101, ex);
  //*** Display object contents ***
  cout << *st1 << endl;
  cout << *st2 << endl;
  cout << st3 << endl;
  cout << st4 << endl;
}
```

Activity 16.53

Changes to *StudentClass*:

Modified code for *main()*:

```
int main()
{
  Student group[4] {Student("Madeline Bray", 'F'),
  Student("Nicholas Nickleby",'M'),
  Student("Ada Clare", 'F', false),
  Student("Richard Carstone", 'M') };
  //*** Assign random marks ***
  for(int idx = 0; idx < 4; idx ++)
    for (int ex = 0; ex < 6; ex++)
      group[idx].SetMark(rand() % 101, ex);
  //*** Display objects' contents ***
  for (auto v: group)
    cout << v << endl;
}
```

Activity 16.54

Changes to *StudentClass*:

Modified code for *main()*:

```
int main()
{

  Student *group[4]
  {
    new Student("Madeline Bray", 'F'),
    new Student("Nicholas Nickleby",'M'),

    new Student("Ada Clare", 'F', false),
    new Student("Richard Carstone", 'M')
  };
  //*** Assign random marks ***
  for(int idx = 0; idx < 4; idx ++)
    for (int ex = 0; ex < 6; ex++)
      group[idx]->SetMark(rand() % 101, ex);
  //*** Display objects' contents ***
  for (auto v: group)
    cout << *v << endl;
}
```

Activity 16.55

Code for *TimeClass*'s definition of *Time*:

```
#include <iostream>
using namespace std;


class Time
{
  private:
    int hour;
    int minute;
    int second;

    int ToSeconds() const;
    Time FromSeconds(int) const;

  public:
    Time(int = 0, int = 0, int = 1);
    void SetTime(int, int, int);
    int GetHour() const;
    int GetMinute() const;
    int GetSecond() const;
    Time operator+(const Time&) const;
    Time& operator+=(const Time&);
    Time operator-(const Time&) const;
    Time& operator-=(const Time&);
    Time& operator++();
    Time operator++(int);
    Time& operator--();
    Time operator--(int);
    bool operator== (const Time&)const;
    bool operator!= (const Time&)const;
    bool operator<(const Time&)const;
    bool operator<=(const Time&)const;
    bool operator>(const Time&)const;
    bool operator>=(const Time&)const;

    friend ostream& operator<<(ostream&,
    const Time&);
};
```

Activity 16.56

Complete code for *TimeClass*:

```
#include <iostream>
using namespace std;


class Time
{
  private:
    int hour;
    int minute;
    int second;

    int ToSeconds() const;
    Time FromSeconds(int) const;

  public:
    Time(int = 0, int = 0, int = 1);
    void SetTime(int, int, int);
    int GetHour() const;
    int GetMinute() const;
    int GetSecond() const;
    Time operator+(const Time&) const;
    Time& operator+=(const Time&);
    Time operator-(const Time&) const;
    Time& operator-=(const Time&);
    Time& operator++();
    Time operator++(int);
    Time& operator--();
    Time operator--(int);
    bool operator== (const Time&)const;
    bool operator!= (const Time&)const;
    bool operator<(const Time&)const;
    bool operator<=(const Time&)const;
    bool operator>(const Time&)const;
    bool operator>=(const Time&)const;

    friend ostream& operator<<(ostream&,
    const Time&);
};

//************************************************
//***        Time Class Implementation         ***
//************************************************

//*********************************
//***      Public Methods       ***
//*********************************
//*** Initalise time to h:m:s if valid else 0:0:0***
Time::Time(int h, int m, int s)
{
  if (h < 0 || m < 0 || m > 59 || s < 0 || s > 59)
  {
    hour = 0;
    minute = 0;
    second = 0;
  }
  else
  {
    hour = h;
    minute = m;
    second = s;
  }
```

```
}


//*** Sets time to h:m:s if this is valid ***
void Time::SetTime(int h, int m, int s)
{
  if (h < 0 || m < 0 || m > 59 || s < 0 ||
  ↪s > 59)
  {
    return;
  }
  else
  {
    hour = h;
    minute = m;
    second = s;
  }
}


//*** Returns hour value ***
int Time::GetHour() const
{
  return hour;
}


//*** Returns hour value ***
int Time::GetMinute() const
{
  return minute;
}


//*** Returns hour value ***
int Time::GetSecond() const
{
  return second;
}


//*** Returns time of t + current time ***
Time Time::operator+(const Time& t) const
{
  return FromSeconds(ToSeconds() + t.ToSeconds());
}


//*** Adds time t to current time ***
Time& Time::operator+=(const Time& t)
{
  return *this = FromSeconds(ToSeconds() +
t.ToSeconds());
  return *this;
}


//*** Returns current time  - t ***
Time Time::operator-(const Time& t) const
{
  return FromSeconds(ToSeconds() - t.ToSeconds());
}


//*** Subtracts time t from current time ***
Time& Time::operator-=(const Time& t)
{
  return *this = FromSeconds(ToSeconds() -
  ↪t.ToSeconds());
  return *this;
}


//*** Adds 1 sec to time (pre) ***
Time& Time::operator++()
{
  *this = FromSeconds(ToSeconds() + 1);
  return *this;
}


//*** Adds 1 sec to time (post) ***
Time Time::operator++(int)
{
  Time result{ *this };
  *this = FromSeconds(ToSeconds() + 1);
  return result;
}


//*** Subtracts 1 sec from time (pre) ***
Time& Time::operator--()
{
  *this = FromSeconds(ToSeconds() - 1);
  return *this;
}


//*** Subtracts 1 sec from time (post) ***
Time Time::operator--(int)
{
  Time result{ *this };
  *this = FromSeconds(ToSeconds() - 1);
  return result;
}


//*** Returns true if times are equal ***
bool Time::operator==(const Time& t) const
{
  return (ToSeconds() == t.ToSeconds());
}


//*** Returns true if times are not equal ***
bool Time::operator!=(const Time& t) const
{
  return (ToSeconds() != t.ToSeconds());
}


//*** Returns true if time < t ***
bool Time::operator<(const Time& t) const
{
  return (ToSeconds() < t.ToSeconds());
}


//*** Returns true if time <= t ***
bool Time::operator<=(const Time& t) const
{
  return (ToSeconds() <= t.ToSeconds());
}


//*** Returns true if time > t ***
bool Time::operator>(const Time& t) const
{
  return (ToSeconds() > t.ToSeconds());
}


//*** Returns true if time >= t ***
bool Time::operator>=(const Time& t) const
{
  return (ToSeconds() >= t.ToSeconds());
}


//*********************************
//***     Private Methods       ***
//*********************************

//*** Converts current time to seconds ***
int Time::ToSeconds() const
{
  return hour * 3600 + minute * 60 + second;
}

//*** Converts seconds to hrs, mins, secs ***
Time Time::FromSeconds(int secs) const
{
  if (secs < 0)
    return Time(0, 0, 0);
  int h = secs / 3600;
  int m = (secs - h * 3600) / 60;
  int s = secs % 60;

  return Time(h, m, s);
}


//*********************************
//***         Friends           ***
//*********************************

//*** Displays the value of time t ***
ostream& operator<<(ostream& ct, const Time& t)
{
  char timestr[9];
```

```
    sprintf(timestr, "%02d:%02d:%02d", t.hour,
    ↳t.minute, t.second);
    ct << timestr;
    return ct;
}


int main()
{
    Time tm(2, 9, 6);
    Time tm2(6, 12, 57);
    tm += tm2;
    cout << tm << endl;
    if (tm < Time(9, 0, 0))
        cout << "Total time is less than nine hours\n";
    else
        cout << "Total time is nine hours or more\n";
}
```

The above code was executed twice with a total time less than 9 hours and again with a total time greater than 9 hours.

17

Creating Static Libraries

In this Chapter:

- ❑ Organising Code
- ❑ Header and Implementation Files
- ❑ Static and Dynamic Libraries
- ❑ Static Libraries in Visual Studio
- ❑ Using a Static Library
- ❑ Namespace

Organising Code

Introduction

When we create a set of functions or define a class that we want to make use of in future projects, it would be very inefficient if we had to copy and paste the existing code into these projects. To demonstrate the problem – and the solution – let's look at a trivial example.

Our original program contains the definition of a type, *CoordsType* and a function *SetCoords()* as well as a `main()` function. The code is shown in FIG-17.1

FIG-17.1

Defining CoordsType

```
#include <iostream>

struct CoordsType
{
   double x;
   double y;
};

CoordsType SetCoords(const double, const double);

int main()
{
   CoordsType point;
   point = SetCoords(3, 9);
   std::cout << '('<<point.x <<','<< point.y <<')' << std::endl;
}

CoordsType SetCoords(const double xv, const double yv)
{
   CoordsType result;
   result.x = xv;
   result.y = yv;
   return result;
}
```

Notice that we have not included the usual `using namespace std;` line and therefore have had to add the term `std::` as a prefix to `cout` and `endl` in `main()`.

Activity 17.1

Start a new project called *UsingHeaders* and implement the code given above.

Now let us assume we want to make use of the code we have written for *CoordsType* in another program. Rather than simply copy and paste the code to our new project we set up two files to contain that code.

Header Files

The first of these files is known as a **header file** and is named with a *.h* extension. This file contains only the `struct` definition and any relevant function prototypes. So in this case, that means it will contain the code shown in FIG-17.2.

FIG-17.2

Contents of *CoordsType.h*

```
struct CoordsType
{
   double x;
   double y;
};

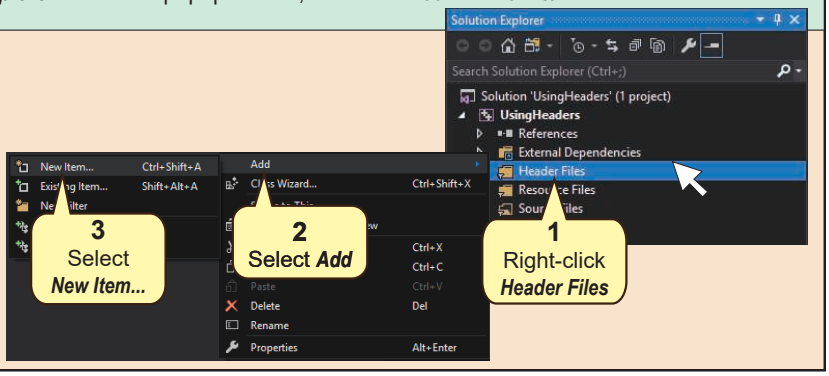
CoordsType SetCoords(const double, const double);
```

Typically, the file will be named after the struct name, giving us the name *CoordsType.h*. The steps needed to create our header file are shown in FIG-17.3.

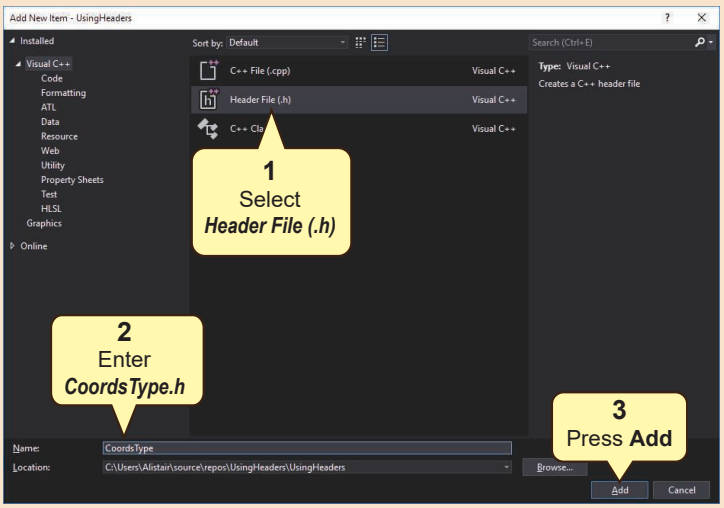
FIG-17.3

Creating a Header File for the *CoordsType*

With the project *UsingHeaders* open, we right click on ***Header Files*** in the *Project Explorer*. From the popup menus, we select ***Add*** and ***New Item...***



In the dialog box that appears, select ***Header File (.h)*** and in the *Name:* edit box, type the name of the struct and "*.h*" – in this case *CoordsType.h* – then hit **Add**.



Following the above steps will result in a new source file appearing in the editor area. The file will contain a single line of code:

```
#pragma once
```

Since we'll have more to say on the purpose of this line later, we'll comment it out for the moment:

```
//#pragma once
```

Activity 17.2

In *UsingHeaders* create the new header file (calling it *CoordsType.h*) using the steps described above and comment out the `#pragma` line.

Cut the lines of code shown in FIG-17.2 from *Source.cpp* and place them in the header file.

Implementation Files

A second file (known as an **implementation file**) will contain the actual code for the *CoordsType*-related function(s). In this case its contents will be as shown in FIG-17.4.

FIG-17.4

Contents of *CoordsType.cpp*

```
CoordsType SetCoords(const double xv, const double yv)
{
   CoordsType result;
   result.x = xv;
   result.y = yv;
   return result;
}
```

The name of this file will again be derived from the struct name, but this time it will have a standard *.cpp* extension.

Activity 17.3

In Project Explorer, right-click on **Resource Files** and select **Add** and **New Item** from the pop-up menus.

In the dialog box, select **C++ File (.cpp)** and in the *Name:* edit box, enter *CoordsType.cpp*. Now click on the **Add** button.

Move all of the code shown above from *Source.cpp* to *CoordsType.cpp*.

This leaves our original file, *Source.cpp* with only the code for `main()` (see FIG-17.5).

FIG-17.5

Remaining Contents of *Source.cpp*

```
#include <iostream>

int main()
{
   CoordsType point;
   point = SetCoords(3, 9);
   std::cout << '('<<point.x <<','<< point.y <<')' << std::endl;
}
```

But if we look at the contents of the two *.cpp* files, we'll see that the compiler is ready to complain that it does not recognise the term *CoordsType*.

The trouble here is that neither *.cpp* file recognises the existence of the header file. In order for them to do that we must add the line

```
#include "CoordsType.h"
```

at the start of the code in *CoordsType.cpp* and *Source.cpp*. This causes the preprocessor to copy and paste the contents of the named file into the *.cpp* files before compilation begins.

Activity 17.4

Add the `#include` statement given above as the first line of code in the two *.cpp* files. What happens to the red underlines in the code?

Check that the program executes correctly.

But wait! All the other `#include` statements we have used in previous programs use angled brackets, so why are we not writing

```
#include <CoordsType.h>
```

instead of

```
#include "CoordsType.h"
```



How the preprocessor treats < > and " " may differ from compiler to compiler.

In fact, the preprocessor treats these two statements somewhat differently. In the case of a `#include` statement which uses angled brackets, it searches for the specified file in pre-defined folders where it expects to find the standard library files – that's where it finds the other header files such as *ctime.h*. Using quotes it searches the pre-defined folders and the folder of the current project.

Activity 17.5

Try changing the `#include` statement to use angled brackets. What is the result?

Change the `#include` statement back to using double quotes.

In the next stage, we're going to add a new type in *Source.cpp*, *LineType*, and two associated functions, *SetLine()* and *LineLength()*. With a test for the new code in `main()` this changes the contents of *Source.cpp* to that shown in FIG-17.6.

FIG-17.6

New code for *Source.cpp*

```
#include <iostream>
#include "CoordsType.h"

struct LineType
{
   CoordsType start;
   CoordsType end;
};

LineType SetLine(CoordsType, CoordsType);
double LineLength(LineType);


int main()
{
   CoordsType st{ 0,0 };
   CoordsType en{ 10,0 };
   LineType line{st, en};
   double length = LineLength(line);
   std::cout << length << std::endl;
}
```



FIG-17.6
(continued)

New code for *Source.cpp*

```
LineType SetLine(CoordsType s, CoordsType e)
{
   LineType result;
   result.start = s;
   result.end = e;
   return result;
}

double LineLength(LineType ln)
{
   double result = sqrt((ln.end.x - ln.start.x)*
   ↳(ln.end.x - ln.start.x) + (ln.end.y - ln.start.y)*
   ↳(ln.end.y - ln.start.y));
   return result;
}
```

Activity 17.6

Repeat the steps you did previously, this time creating a header file called *LineType.h* and a source file called *LineType.cpp* each containing the appropriate parts of the code from that shown above.

Comment out the `#pragma once` line in *LineType.h*.

In order to have the compiler recognise the various struct types and functions, you'll need to add the following lines:

Add `#include "CoordsType.h"` to *LineType.h*.

Add `#include "LineType.h"` and `#include <iostream>` to *LineType.cpp* (you need the second one of these so that `sqrt()` is recognised.

Add `#include "LineType.h"` to *Source.cpp* and modify the contents of `main()` to match that given above.

What happens when you try to compile the program?

The first error displayed by the compiler as a result of Activity 17.6 is

```
Error   C2011  'CoordsType': 'struct' type redefinition
```

#pragma once

Taking a step back for a moment, we know that trying to define the same variable twice in the same block will create an error:

```
int main()
{
   int x;
   int x;   //ERROR
```

And this is exactly what's happening in our latest program. *Source.cpp* contains the lines

```
#include "CoordsType.h"
#include "LineType.h"
```

which tells the preprocessor to copy the contents of the two header files into *Source.cpp*. This means that *Source.cpp* ends up with the contents shown in FIG-17.7.

FIG-17.7

Preprocessor First Step

```
struct CoordsType
{
        double x;
        double y;
};

CoordsType SetCoords(const double, const double);
```

Copied from CoordsType.h

```
#include "CoordsType.h"

struct LineType
{
        CoordsType start;
        CoordsType end;
};

LineType SetLine(CoordsType, CoordsType);
double LineLength(LineType);
```

Copied from LineType.h

```
int main()
{
        CoordsType st{ 0,0 };
        CoordsType en{ 10,0 };
        LineType line{st, en};
        double length = LineLength(line);
        std::cout << length << std::endl;
}
```

Other code in Source.cpp

But notice that code coming from *LineType.h* contains the line

```
#include "CoordsType.h"
```

and this causes the preprocessor to paste yet another copy of *CoordsType.h* into *Source.cpp* (see FIG-17.8).

FIG-17.8

Preprocessor Second Step

```
struct CoordsType
{
        double x;
        double y;
};

CoordsType SetCoords(const double, const double);
```

Copied from CoordsType.h

```
struct CoordsType
{
        double x;
        double y;
};

CoordsType SetCoords(const double, const double);

struct LineType
{
        CoordsType start;
        CoordsType end;
};

LineType SetLine(CoordsType, CoordsType);
double LineLength(LineType);
```

Copied from CoordsType.h

Copied from LineType.h

```
int main()
{
        CoordsType st{ 0,0 };
        CoordsType en{ 10,0 };
        LineType line{st, en};
        double length = LineLength(line);
        std::cout << length << std::endl;
}
```

Other code in Source.cpp

Now we've ended up including the definition of *CoordsType* twice as well as the prototype for *SetCoords()*.

And this brings us back to the line

```
#pragma once
```

This is a command that tells the preprocessor that it must never paste that file more than once into any single *.cpp* file.

Activity 17.7

Return to the files *CoordsType.h* and *LineType.h* and remove the comment markers from the `#pragma once` lines.

What happens now when you attempt to compile and run the program?

Include Guard Alternatives

The `#pragma once` command is known as an **include guard** since it guards against multiple inclusions of a header file.

The `#pragma once` statement is not a defined standard and although implemented by a wide range of compilers, there's no guarantee it will work on your compiler.

An alternative way of achieving the same effect makes use of three further pre-processor commands and takes the form

```
#ifndef GUARD_NAME
#define GUARD_NAME

    C++ code goes here

#endif
```

In structured English the commands are equivalent to:

```
IF GUARD_NAME is not defined THEN
    Define GUARD_NAME
    Copy all of the C++ code given here to the file with the #include statement
ENDIF
```

The actual guard name is traditionally uppercase and is selected by the programmer but must be different for each header file. Normally it will be based on the file name. For example, if we were to use this approach in *LineType.h* its contents would be as shown in FIG-17.9.

FIG-17.9

Alternative Include Guard Coding

```
#ifndef LINETYPE
#define LINETYPE

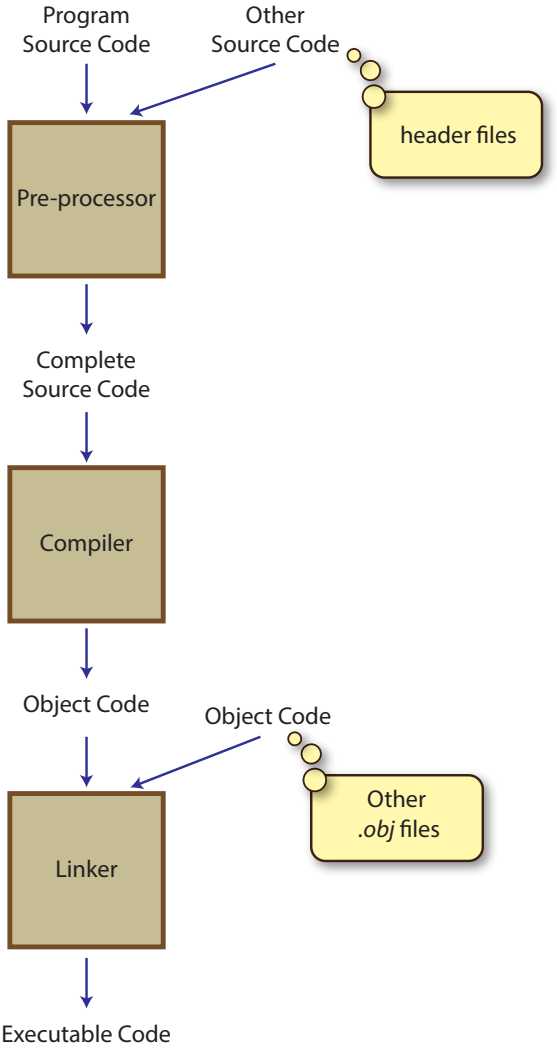
#include "CoordsType.h"
struct LineType
{
   CoordsType start;
   CoordsType end;
};
LineType SetLine(CoordsType, CoordsType);
double LineLength(LineType);

#endif
```

When we write a program that uses header files and multiple implementation files, building the program requires the following steps (see FIG-17.10).

FIG-17.10

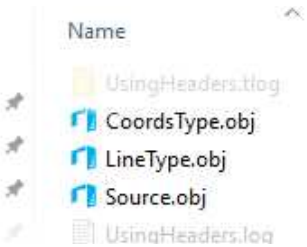
The Compilation Process.



If we use File Explorer to examine the contents of project *UsingHeaders*'s *Debug* folder, we can see the three *.obj* files brought together by the linker to create the final executable code (see FIG-17.11).

FIG-17.11

The *.obj* File Created by the Compiler



Activity 17.8

Use File Explorer to examine the contents of *UsingHeader*'s *Debug* folder and check that it contains the three *.obj* files shown above.

Header and Implementation Files for a Class

What we've just done with traditional structs and functions code can also be performed on classes.

We can place a class's declaration in an appropriate header file and the code for the class's methods in the corresponding implementation file.

Activity 17.9

Load your *DateClass* project and create the new header file (calling it *Date.h*).

Now we have to move the lines containing declaration of the *Date* class from *source.cpp* (our original file) to our new header file. The lines to be moved are:

```
//*** Class declaration ***
class Date
{
private:
   int day;
   int month;
   int year;
   long ToJDN();
   Date FromJDN(long);

public:
   static const int DAYSINYEAR = 365;
   static int count;
   explicit Date(int = 1, int = 1, int = 2001);
   void SetDate(int, int, int);
   int GetDayOfWeek();
   bool IsLeapYear();
   int GetDay();
   int GetMonth();
   int GetYear();
   Date operator+(int);
   int operator-(Date);
   Date operator-(int);
   bool operator==(Date);
   void operator+=(int);
   Date operator++();
   Date operator++(int);
   Date operator--();
   Date operator--(int);
   int GetDaysInYear();
   static int GetDAYSINYEARconst();
   static int GetCount();
   ~Date();

   friend ostream& operator<<(ostream&, Date&);
   friend istream& operator>>(istream&, Date&);
   friend Date operator+(int, Date&);
};
```

Activity 17.10

Remove the lines shown above from *DateClass*'s *source.cpp* and paste them into *Date.h*.

At this point we can see that, in our new *Date.h* file, the compiler has a problem with the terms `istream&` and `ostream&` in the friend prototypes.

These classes are defined within `iostream`, so we need to add the line

```
#include<iostream>
```

after the `#pragma once` line.

Also, because we have not used the line

```
using namespace std;
```

(which would not be considered a good approach when creating separate files for a class) we need to prefix the term `std::` to any reference to `istream` and `ostream`.

This means our `Date` class declaration now ends with the lines

```
    friend std::ostream& operator<<(std::ostream&, Date&);
    friend std::istream& operator>>(std::istream&, Date&);
    friend Date operator+(int, Date&);
};
```

Activity 17.11

Add the `#include<iostream>` line to *Date.h* and prefix any reference to `ostream` or `istream` with the expression `std::`.

We now need to move the code for the initialisation of *count* (`int Date::count = 0;`) and the methods of the *Date* class to an implementation file (*Date.cpp*).

Activity 17.12

In Project Explorer, right-click on **Resource Files** and select **Add** and **New Item** from the pop-up menus.

In the dialog box, select **C++ File (.cpp)** and in the *Name:* edit box, enter *Date.cpp*. Now click on the **Add** button.

Move all of the code relating to the *Date* class *count* initialisation, methods and friends from *Source.cpp* to *Date.cpp*.

Add the line

```
#include "Date.h"
```

as the first line in *Date.cpp*.

Activity 17.13

Make any changes necessary to have *Source.cpp* compile and check that the program executes as before.

Using Header and Obj Files in Other Projects

At the start of this chapter we talked about using existing code in new projects. But so far we've been cheating since all our header and implementation files have been used in the same project that created them!

So let's see how we can use our *Date* class code in an entirely new project in a new solution.

FIG-17.12 shows the code for new project called *UsingDateClass*.

FIG-17.12

Using the Date Class in Another Project

```
#include <iostream>
#include "Date.h"
using namespace std;

int main()
{
   Date today{ 26,2,2019 };
   cout << "Today is : " << today << endl;
}
```

As we can see the project itself is trivial. Our only interest is exploring how to make use of the *Date* class within this project.

Activity 17.14

Start a new project called *UsingDateClass* containing the code given above. Does the program compile?

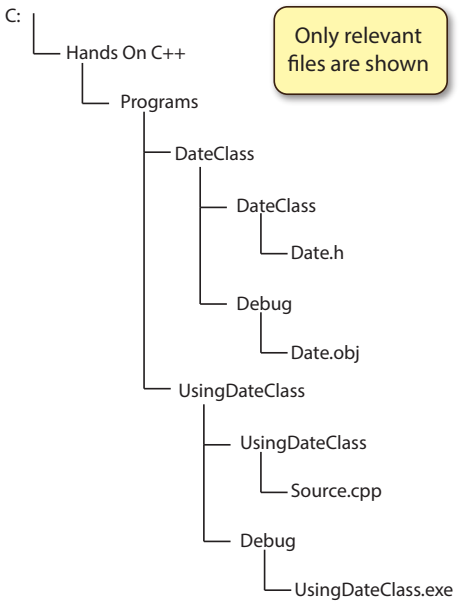
Our attempt at building *UsingDateClass* results in the following error:

```
Error   C1083  Cannot open include file: 'Date.h': No such file or directory
```

Using File Explorer we can see that our *DateClass* and *UsingDateClass* projects have the paths shown in FIG-17.13.

FIG-17.13

Path Information for *DateClass* and *UsingDateClass* Projects



The pre-processor can't find *Date.h* because it is in a different folder from *UsingDateClass*'s *Source.cpp* file. To solve this problem, we need to include path information in our `#include` statement. The path can be either relative or absolute. From the diagram we can see that the relative path would be

```
..\..\DateClass\DateClass\Date.h
```

Activity 17.15

In your program, modify the `#include` statement for *Date.h* giving the path shown above. Does the program compile and build?

This time we have a linker error:

```
Error    LNK1104    cannot open file 'Date.obj'
```

In fact, we have exactly the same type of problem as we had with the header file. This time it's the linker that can't locate *Date.obj*.

To supply the linker with the information it needs to find *Date.obj*, we need to follow the instructions shown in FIG-17.14.

FIG-17.14

Supplying the Linker with Path Details

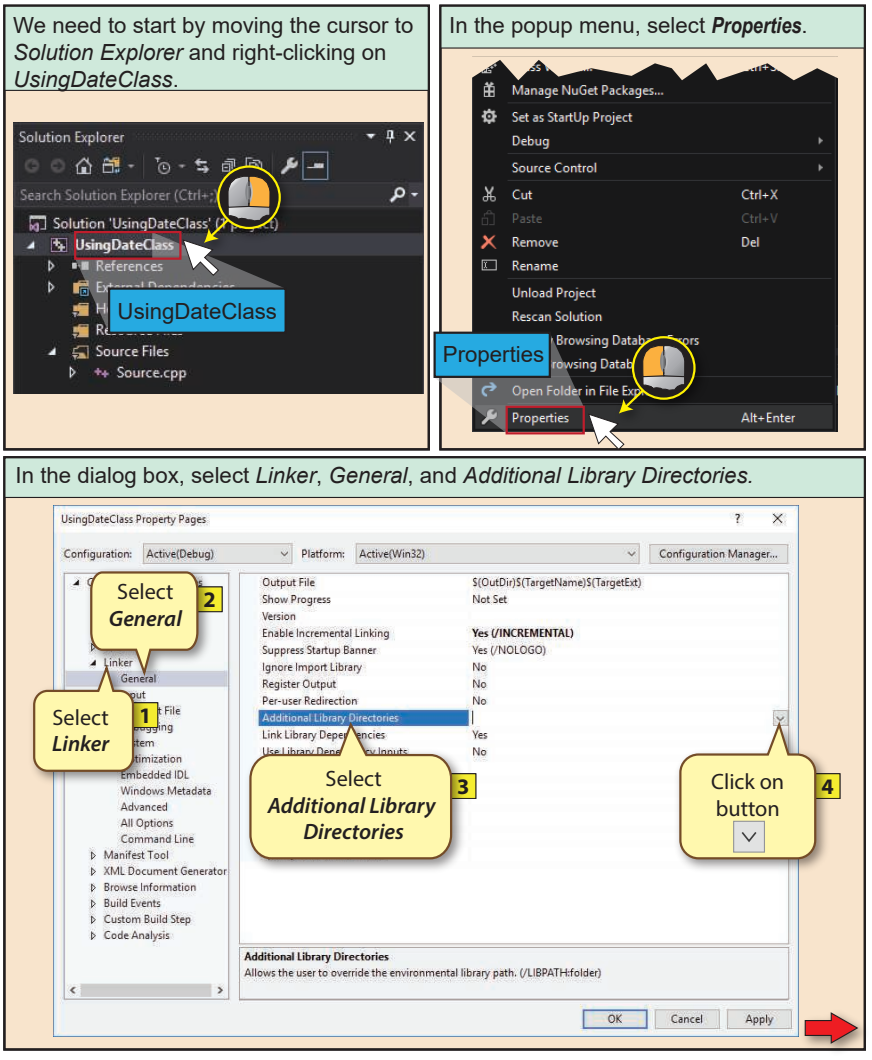
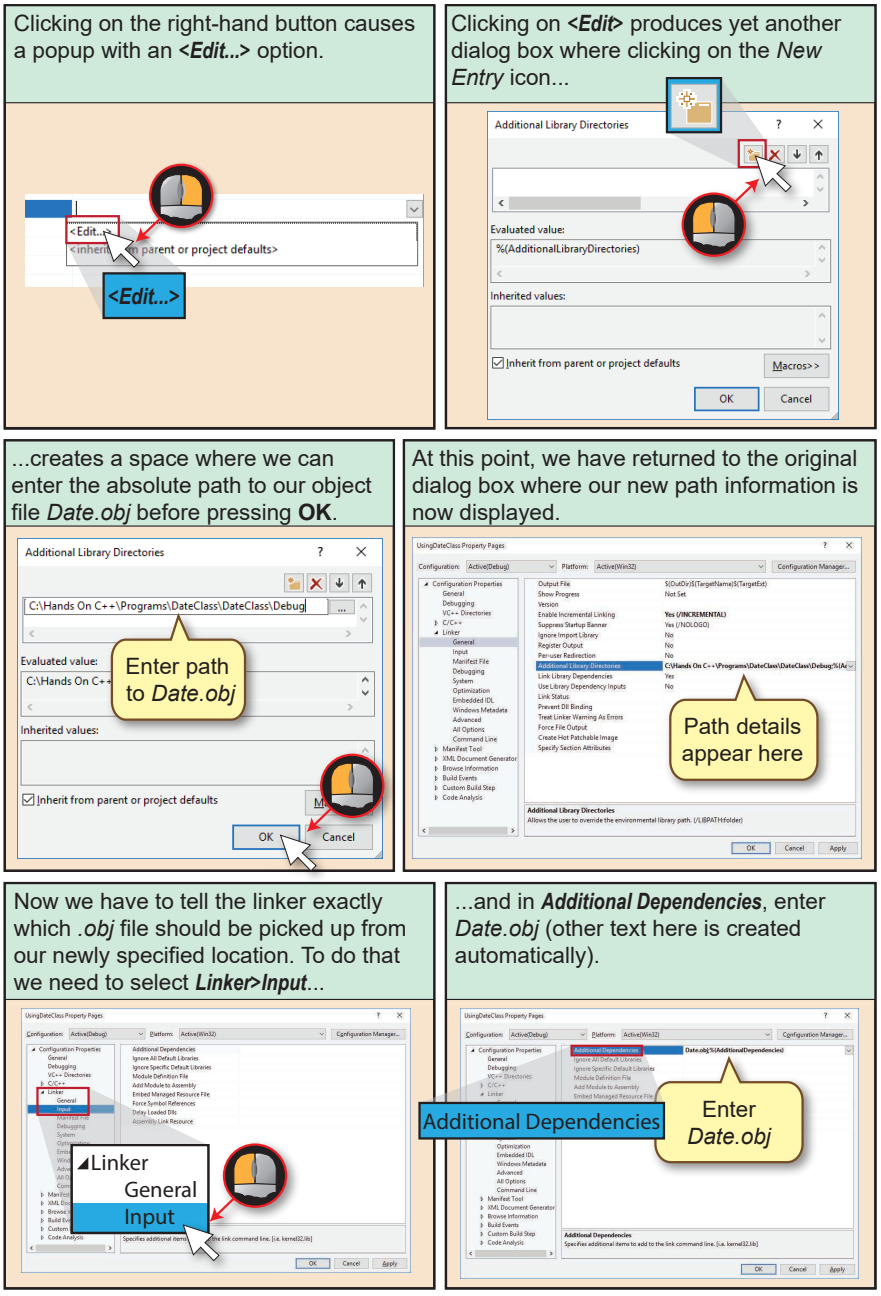


FIG-17.14
(continued)

Supplying the Linker with Path Details



With the path to *Date.obj* added, we can now run the program.

Activity 17.16

Using the steps described above, add the path C:\Hands On C++\Programs\DateClass\DateClass\Debug to the linker's *Additional Library Directories* option and *Date.obj* to *Additional Dependencies*.

Check that the program runs.

Although we now have our program running, there's one last thing we can do which will allow us to remove the path information given in the line

```
#include "..\..\DateClass\DateClass\Date.h"
```

reducing it to the more standard

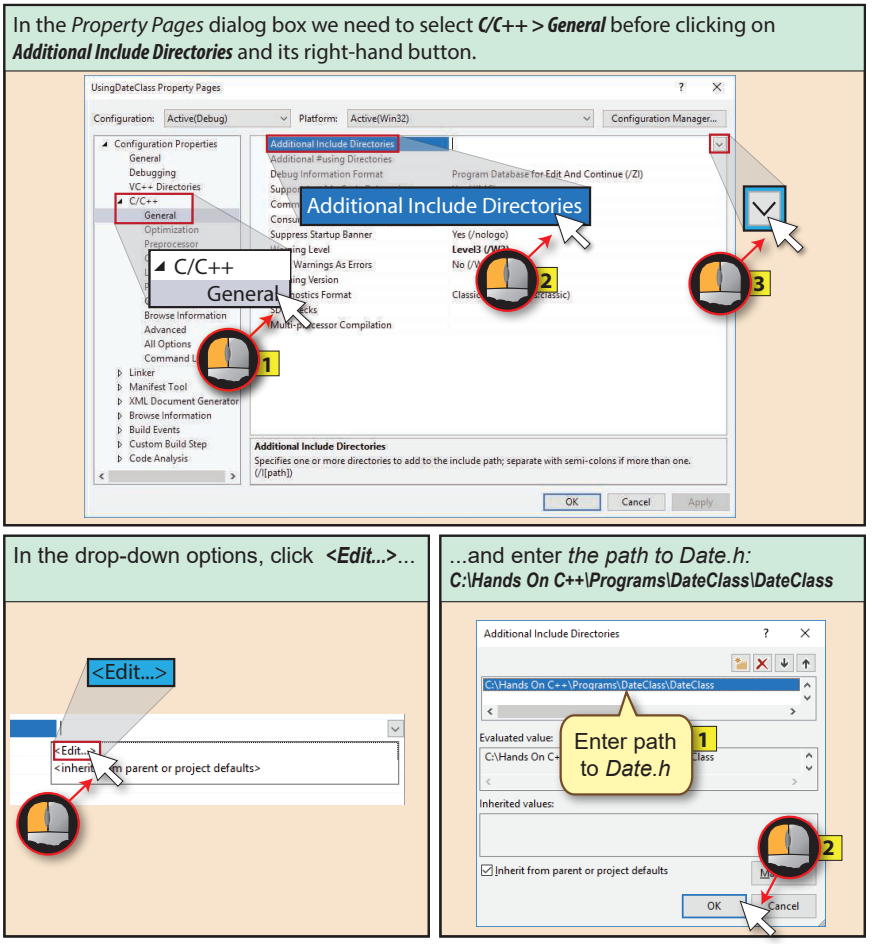
```
#include "Date.h"
```

To do this we must return to the *Properties* option for *UsingDateClass* by right-clicking at the appropriate point in *Solution Explorer*.

The remaining steps are shown in FIG-17.15.

FIG-17.15

Setting Additional Include Directories



Activity 17.17

Using the steps described above, add the path C:\Hands On C++\Programs\DateClass\DateClass to the C/C++'s *Additional Include Directories* option.

Reduce the second `#include` statement in *UsingDateClass* to read

```
#include "Date.h".
```

Check that the program runs correctly.

Summary

- When using previously coded structs, functions or classes in an application we need to separate those elements into two files: a header file and an implementation file.
- A header file contains the struct declaration and function prototypes, or class declarations.
- Header file names have a *.h* extension.
- Header file contents are merged with an app's own code by the preprocessor.
- To ensure only one copy of a header file is included, many preprocessors recognise the command

```
#pragma once
```

  at the start of a header file as an instruction to insert only once.
- Where the `#pragma` command is not recognised, the standard way of ensuring a header file's contents are included only once is to construct the header file in the following format

```
#ifndef GUARD_NAME
        #define GUARD_NAME
        C++ code goes here
#endif
```

- An implementation file contains the actual code for functions or a class's methods.
- Implementation file names have the extension *.cpp* (this may differ on some compilers).
- The implementation file must be compiled to create a *.obj* file containing executable code.
- When dealing with a single class, the header file and implementation files should be named after the class name (e.g. *Date.h* and *Date.cpp*).
- To instruct the preprocessor to insert the contents of a header file into an app, the app must use an `#include` instruction giving the name of the header file to be inserted.
- Header file names in an `#include` statement can be enclosed in angle brackets or double quotes.
- When a `#include` statement uses angled brackets to enclose the header file name, the preprocessor searches for the header file at the locations set by the compiler.
- When a `#include` statement uses angled brackets to enclose the header file name, the preprocessor searches for the header file in the current folder and the locations set by the compiler.
- When we build an application the preprocessor is responsible for adding the contents of included header files and the linker is responsible for adding additional *.obj* files.

Visual Studio Settings

- If the header file to be included is in a different solution, one option is to

include the path information (relative or absolute) within the double quotes of the `#include` statement.

- An alternative to adding path information to the `#include` statement is to set up that path information in the project's **Properties>C/C++>General>Additional Include Directories** entry.
- To inform the linker on where to find additional *.obj* files, we need to set up the path information in **Properties>C/C++>Linker>Additional Library Directories** and then, in **Properties>C/C++>Linker>Input>Additional Dependencies,** enter the full name of any *.obj* files to be used.

Creating a Static Library

Introduction

So far we've explored how to separate the code for a class into a header file and an implementation file and how to use those files in the same project or in other projects. But if we want to be a bit more organised, we'll want to create C++ library files.

A library file (*.lib*) is a collection of compiled implementation files (*.obj* files). This is used in conjunction with a corresponding set of header files which declare the classes or functions implemented by the *.obj* files. When using items from a library in another project, we write a `#include` instruction for each header file we need and the linker selects the appropriate *.obj* files from the library file.

Visual Studio can be used to create two types of libraries. These are:

Dynamic Link Libraries (DLL) which are loaded at run time and can be used by several programs simultaneously.

Static Libraries which contain code that is added to the program at compile time to produce the final executable program.

Although DLLs have some advantages over static libraries, not all platforms support DLLs so for maximum usage it is best to make use of static libraries when building a set of reusable classes.

Creating a Library

Over the next few pages we'll create two static library files containing our *Date* class, *Time* class (in the first library) and *Student* class (in the second library). Both libraries will be held in a single solution.

How to begin creating a library by first adding the *Student* class is described in FIG-17.16.

FIG-17.16

How to Create a Library File

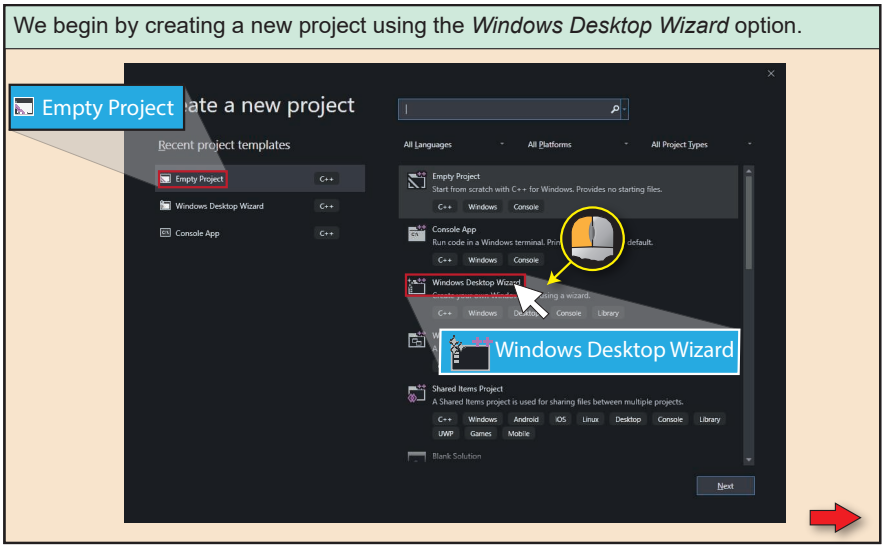
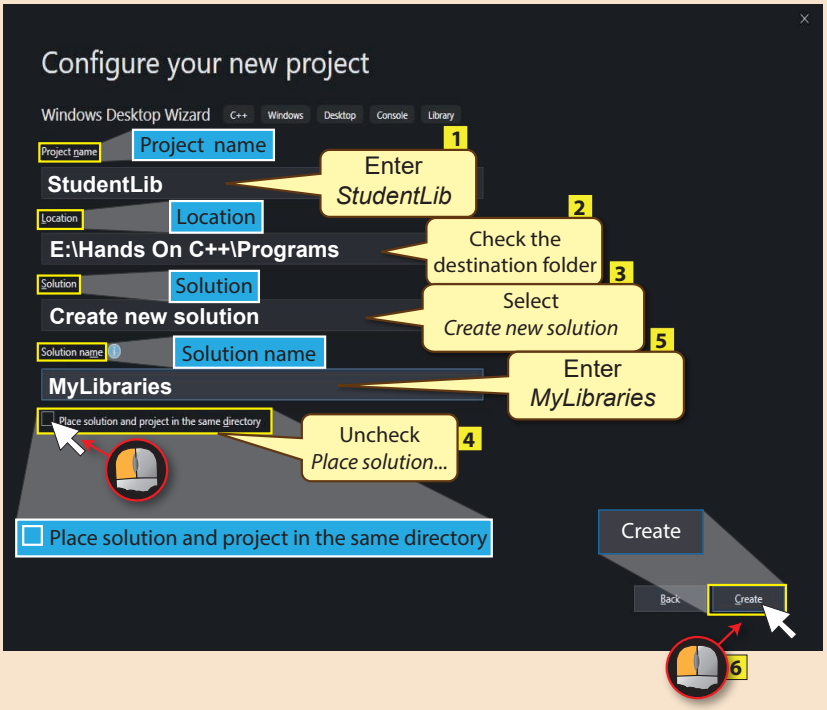


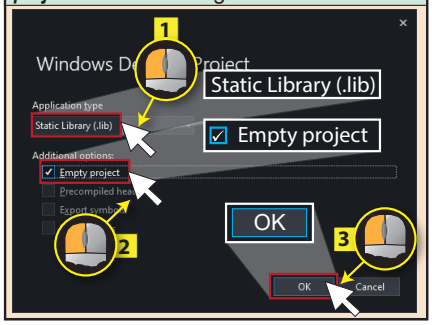
FIG-17.16
(continued)

How to Create a Library File

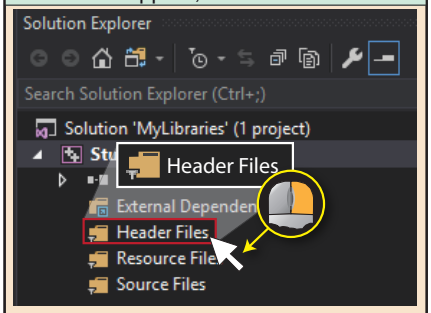
The next step is a little different since we'll name the solution differently from the project.



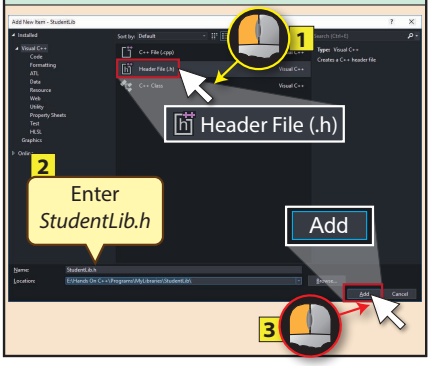
In the dialog box that appears, we need to select ***Static Library*** and check ***Empty project*** before clicking on **OK**.



Now, in *Solution Explorer*, we need to right-click on ***Header Files***, then, in the sub-menus that appear, ***Add*** and ***New Item...***



The first of the new files is the header file for our *Student* class. The header file's name should match that of the library.



We now copy the code defining the *Student* class into our header file immediately after the automatically generated `#pragma once` statement.

```
#pragma once
#include <iostream>

class Student
{
    private:
          char* name{ nullptr };
          char sex{ '?' };
          bool attending{ true };
          int marks[6]{ 0 };
    public:
          Student(const char*, char, bool = true);
          Student(const Student&);
          void SetName(const char*);
          void SetSex(char);
          void SetAttending(bool);
          void SetMark(int, int);
          char* GetName() const;
          char GetSex() const;
          bool GetAttending() const;
          int GetMark(int) const;
          double GetAverageMark() const;
          Student& operator=(const Student&);
          ~Student();

          friend std::ostream& operator<<(std::ostream&, Student&);
};
```

Activity 17.18

Start a new project called *StudentLib* within a solution called *MyLibraries* defining the project as a **Static Library**.

In Solution Explorer, right-click on project *StudentLib*'s **Header Files** option and create a header file called *StudentLib.h.*

Copy the code defining the *Student* class structure into *StudentLib.h*. Include the `#include<iostream>` line but remove `using namespace std;`.

To compensate for the removal of the `using namespace` line, add the prefix `std::` at the appropriate points in the friend function prototype.

We now need to follow a similar sequence of steps to create *Student.cpp*:

1. In Solution Explorer, right click **Source Files**.
2. Select **Add** and **New Item...**
3. Select **CPP File (.cpp)** and name the file *Student.cpp*.
4. Copy the implementation code for the methods of *Student* class.
5. At the start of the code, add the lines

```
#include "StudentLib.h"
#include <iostream>
```

6. In the code for `operator<<()` add the prefix `std::` at the appropriate points.

Activity 17.19

Using the instructions given above, add the file *Student.cpp* to your *StudentLib* project and ensure that the code compiles (**Ctrl+F7**).



Using VS, red underscores in the code indicate points where the compiler would highlight errors.

Adding another Library

Up to now our library hasn't strayed far from the approach we used earlier on the code for *Date* where we separated it into a header and implementation file within the *DateClass* project.

So to begin highlighting the advantages of a set of library files, let's build a second library, called *ChronoLib*, which contains header and implementation files for both *Date* class and *Time* class. The steps involved in the first stage in achieving this goal are shown in FIG-17.17.

FIG-17.17

Adding to a Library

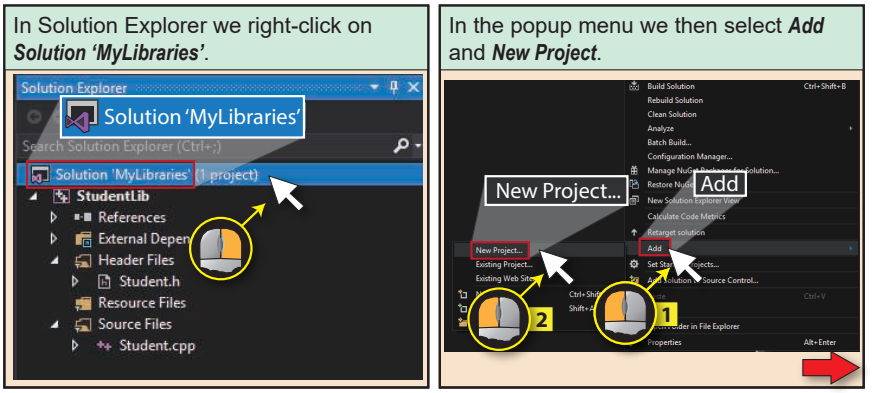
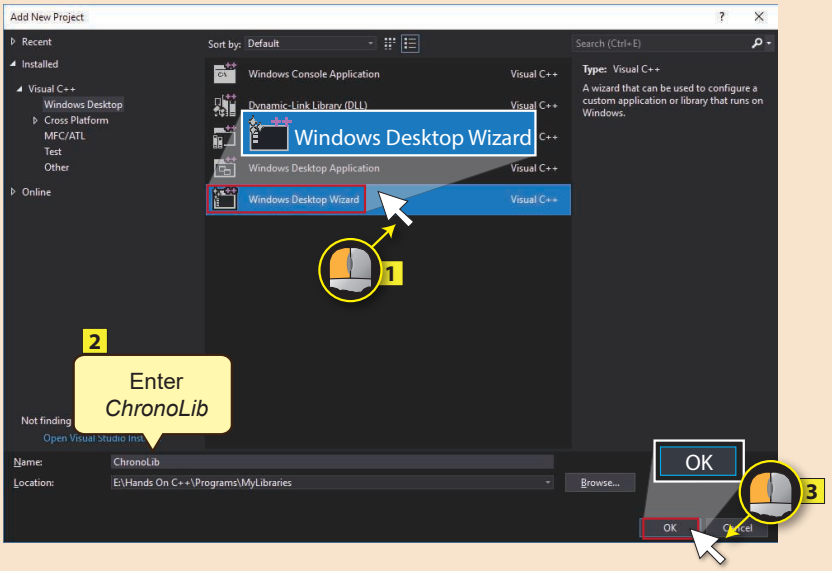


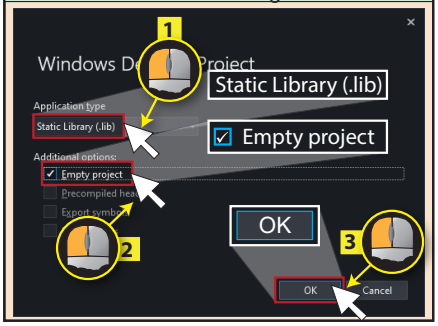
FIG-17.17
(continued)

Adding to a Library

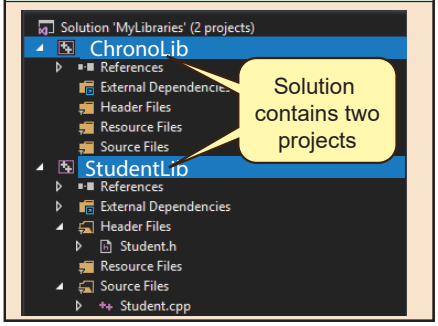
Despite the fact that there is a *Static Library* option in the dialog box, it's still ***Windows Desktop Wizard***, we need to select. The project name is *ChronoLib*.



In the Wizard's dialog box, we select ***Static Library*** and deselect the usual checkboxes before clicking **OK**.



Returning to *Solution Explorer*, we can see that there are now two projects within the solution.



Activity 17.20

Using the instructions given above, add the new project *ChronoLib* to your *MyLibraries* solution.

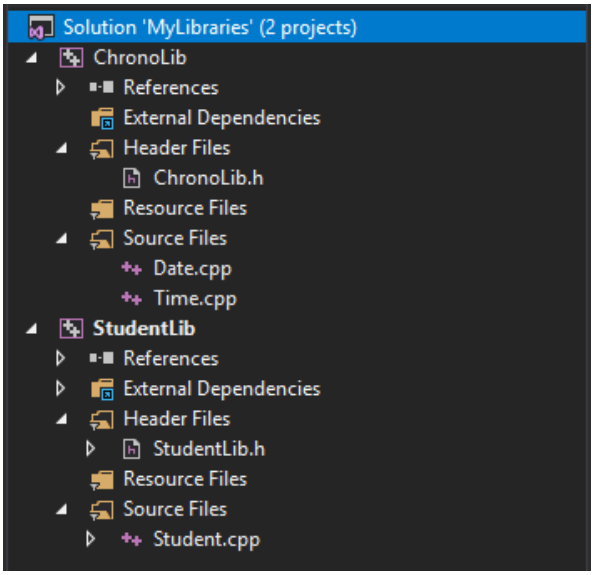
Adding Two Classes to One Library

With our new *ChronoLib* project set up within *MyLibraries* solution, we now need to add an implementation file for *Date* class and another for *Time* class. However, rather than create a header file for each of these classes, we will create only a single header file called *ChronoLib.h*.

At the end of the file-creation process, the *Solution Explorer* will show the details seen in FIG-17.18.

FIG-17.18

Library Structure



When a library contains multiple definitions, the usual approach is to include the appropriate details in a single header file so there is no need for an unrealistic number of #include statements in future application programs. However, the implementation files are kept separate to optimise the size of the final executable file since the linker will only load those *.obj* files related to objects used in the program being created.

Activity 17.21

Add the header file *ChronoLib.h* to project *ChronoLib* as well as the implementation files *Date.cpp* and *Time.cpp*.

In the file *ChronoLib.h*, copy the code for the declarations of the *Date* and *Time* classes.

In the file *Date.cpp* copy the implementation code for the *Date* class.

In the file *Time.cpp*, copy the implementation code for the *Time* class.

Ensure the project builds without errors.

Use File Explorer to find the *.lib* files created by the build.

Using a Static Library

When we are working on our own, creating relatively small applications, it's not unusual to keep all of our projects in the same solution. In larger projects, where we may be working with a team of people, projects may be spread over several solutions.

If our library projects are in the same solution as the application that makes use of them, then the setup is fairly straight forward.

When we have our application in a different solution from the library, then we require a few additional steps in order to compile our program.

Libraries and Application in the Same Solution

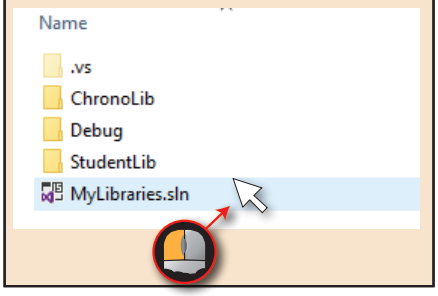
We'll start with the simplest option, and assume our application is in the same solution as our *StudentLib* and *ChronoLib* libraries. That solution was named *MyLibraries*.

The steps needed to create the app and link it to our two libraries are shown in FIG-17.19.

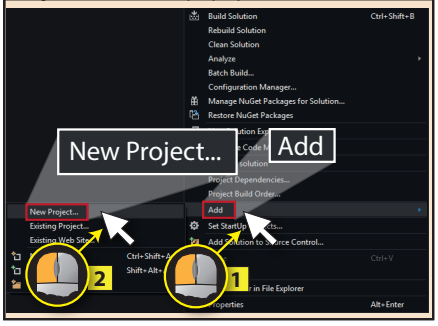
FIG-17.19

Using a Library in the Same Solution

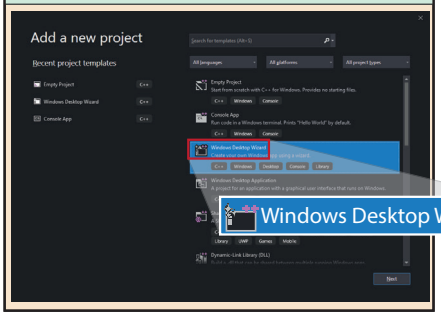
In Visual Studio, we select ***File|Open|Project/Solution*** and choose *MyLibraries.sin*.



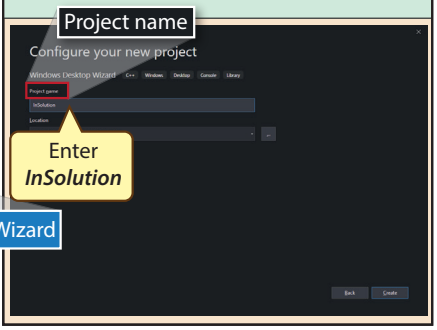
In Solution Explorer, right-click on the solution name and choose ***Add > New Project...*** from the popup menus.



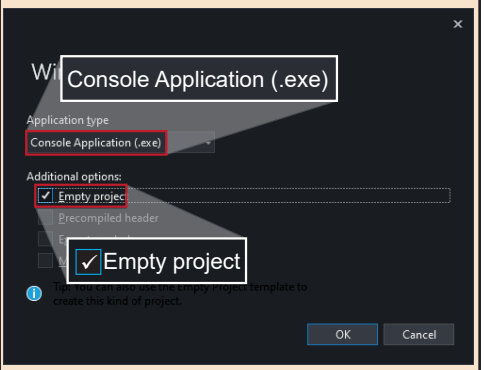
In the *Add New Project* dialog box we pick the usual *Windows Desktop Wizard*



The new project is named *InSolution*.



The next dialog box selections are *Console Application*, check *Empty Project*.



The new project will now be listed in *Solution Explorer*.

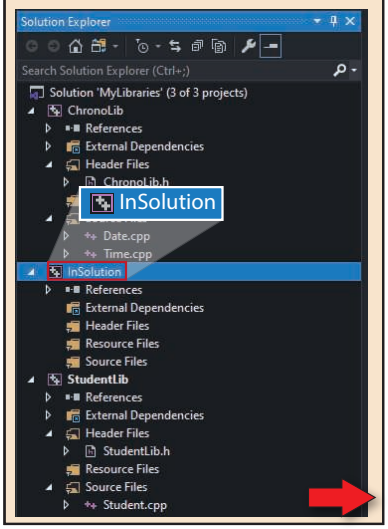
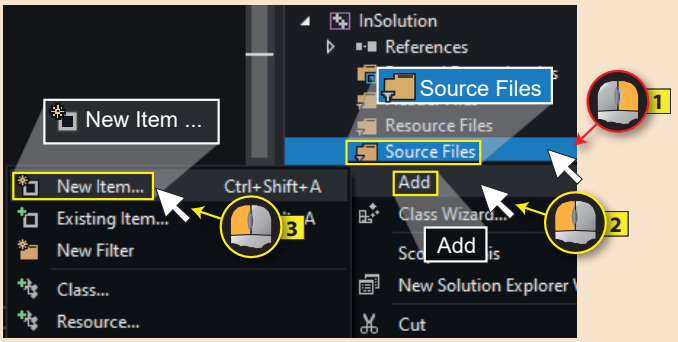


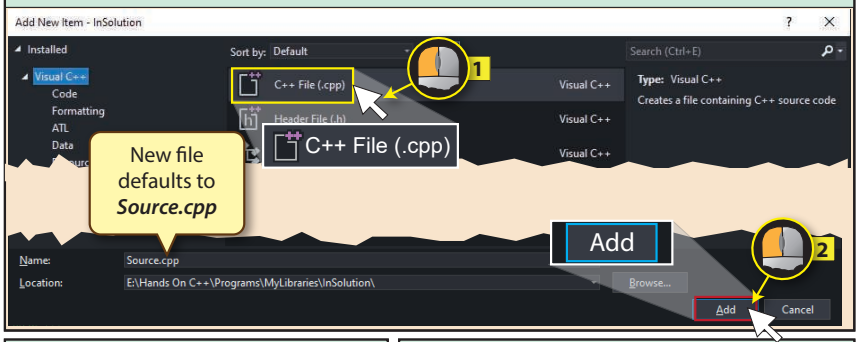
FIG-17.19
(continued)

Using a Library in the Same Solution

Next we follow the usual steps to add a new item to the *InSolution* project.



In the *Add New Item* dialog box we need to select ***C++ File (.cpp)*** and click **Add**.



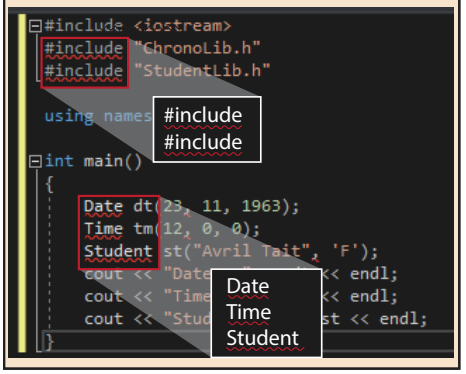
In the new *Source.cpp* file we need to enter our test code which creates an object of each class.

```
#include <iostream>
#include "ChronoLib.h"
#include "StudentLib.h"

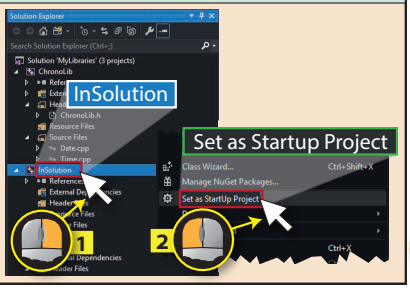
using namespace std;

int main()
{
        Date dt(23, 11, 1963);
        Time tm(12, 0, 0);
        Student st("Avril Tait", 'F');
        cout << "Date : " << dt << endl;
        cout << "Time : " << tm << endl;
        cout << "Student : " << st << endl;
}
```

However, at this point, the compiler complains about much of our code (shown as wavey underlines).



Since there are several projects in our solution, we need to right-click on ***InSolution*** in *Solution Explorer* and pick ***Set as StartUp Project*** from the popup menu.



To ensure the linker can find the library files, we now need to right-click on *InSolution* for a second time and pick ***Add>Reference...*** from the popup menu.

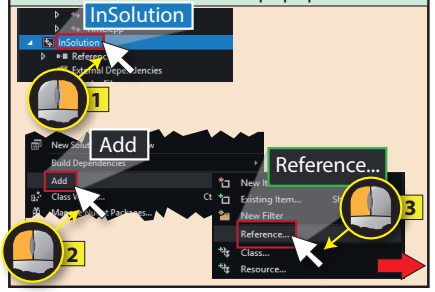
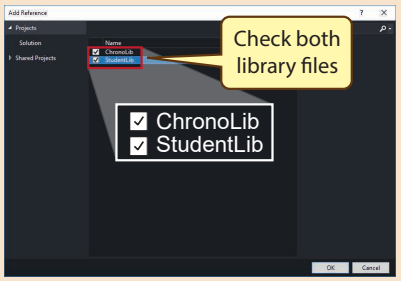


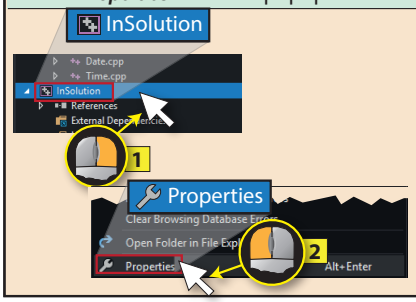
FIG-17.19
(continued)

Using a Library in the Same Solution

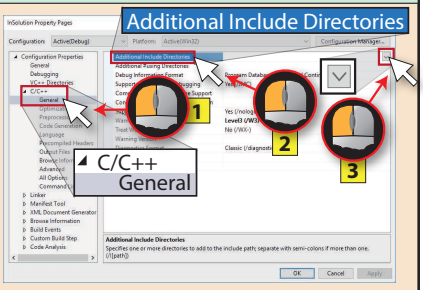
In the *Add Reference* dialog box we want to check both libraries since we will be creating objects from the classes defined in the two libraries.



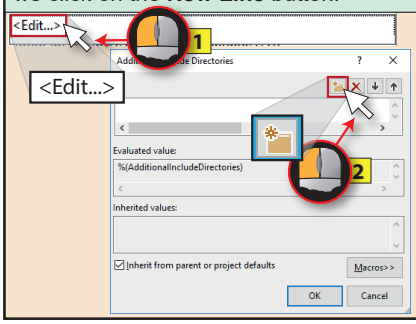
Now we must specify the location of any header files used. To do this we right-click on *InSolution* in the *Solution Explorer* and select ***Properties*** from the popup menu.



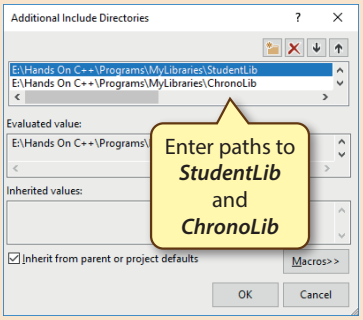
In the dialog box we need to select ***C/C++>General***, *Additional Include Directories*, then click on the button that appears on the extreme right of that line.



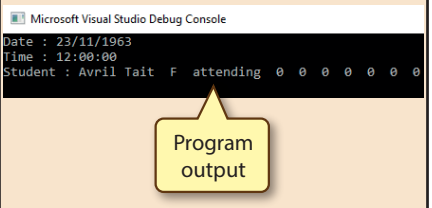
Clicking the button produces a drop-down option,***<Edit...>*** and clicking on this displays yet another dialog box where we click on the **New Line** button.



The button press creates an edit line where we enter the path to a library. Repeating the button press allows the second library path to be entered.



Returning to the source code for the main program, the red underline for the include files and class names will be gone and the program is ready to run.



Activity 17.22

In your *MyLibraries* solution add a new console application project called *InSolution* and add to it a source file containing the code shown in frame 3 of the previous page.

Follow the remaining steps given above to test the use of your static libraries.

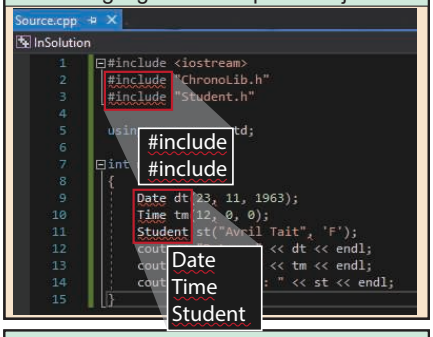
Libraries and Application in Different Solutions

When the project using the libraries is in a different solution from those libraries, then we need to do things a little differently to get our app working. In the steps shown below (see FIG-17.20) we assume that our test program is in a project called *OutSolution* in a solution also called *OutSolution* and that the test program contains the same code as the previous example.

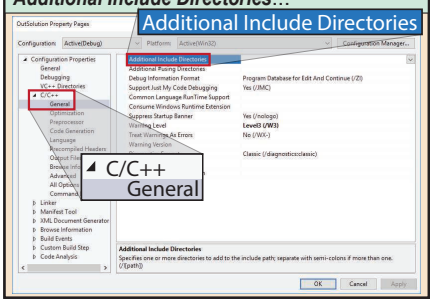
FIG-17.20

Using a Library from a Different Solution

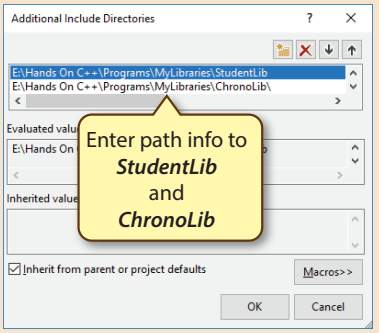
When we first enter the code for our new program, the usual wavey under-scores highlight the compiler's objections.



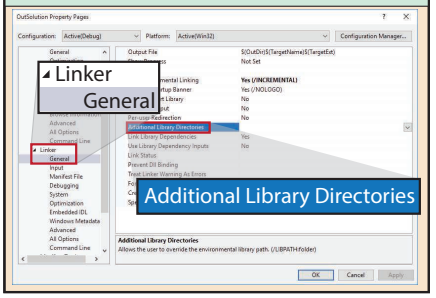
Right-clicking on the project name in the *Solution Explorer* and selecting ***Properties*** then ***C/C++ General*** and ***Additional Include Directories***...



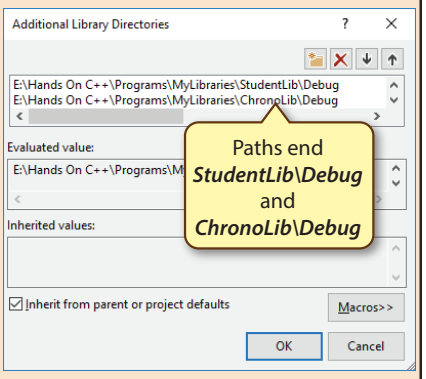
...we can enter the path details for our two libraries.



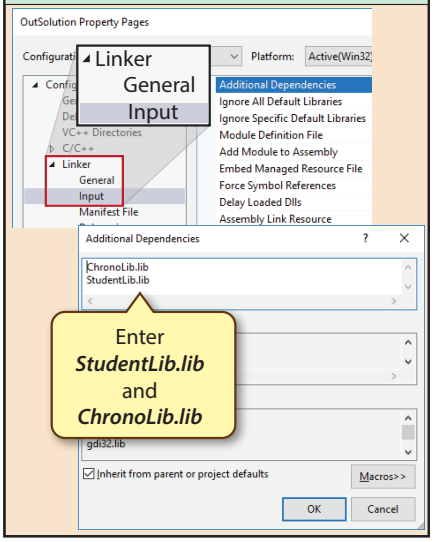
We now need to supply similar details for the linker. To do this we select ***C/C++ > Linker > General*** and ***Additional Library Directories***.



In this area, we enter the paths to the compiled library files.



We must then select ***Linker>Input*** and ***Additional Dependencies*** and add the names of the *.lib* files needed.



Activity 17.23

Start a new project called *OutSolution* in a solution of the same name.

Create a source file containing the same test code as we used in the previous Activity.

In the project's *Property* panel, modify the *Additional Include Directories*, *Additional Library Directories* and *Additional Dependencies* as explained above.

Does the program execute correctly?

Namespace

We've been using namespaces right from our first program but now we get to see its exact purpose. A namespace is used to ensure that there is never any ambiguity when using identifiers which may be unknowingly duplicated elsewhere. Ambiguity can occur if two or more classes, functions, constants, or global variables, defined in different places, have exactly the same name (and, in the case of functions, the same parameter list). For example, we might write a function with the prototype

```
double sin(double);
```

Unfortunately, a function with this prototype already exists in the C++ math library. If we have enclosed our own function in a namespace called *dsk*, then we can differentiate between the two versions of *sin()* by including namespace details when the function is called:

```
double answer = dsk::sin(1.2);
```

Note the use of the scope resolution operator to show that we are using the *sin()* function defined within *dsk* namespace.

Defining a Namespace

To place our code within a namespace we use a line of code such as

```
namespace dsk
```

and then enclose all of the code in a set of braces:

```
namespace dsk
{
      //code within namespace
}
```

For example, let's assume we want to place the code for Student class within the namespace *dsk*. First we would change the header file (*Student.h*) to read:

```
#pragma once
#include <iostream>

namespace dsk
{
   class Student
   {
      private:
         char* name{ nullptr };
```

```
            char sex{ '?' };
            bool attending{ true };
            int marks[6]{ 0 };
        public:
            Student(const char*, char, bool = true);
            Student(const Student&);
            void SetName(const char*);
            void SetSex(char);
            void SetAttending(bool);
            void SetMark(int, int);
            char* GetName() const;
            char GetSex() const;
            bool GetAttending() const;
            int GetMark(int) const;
            double GetAverageMark() const;
            Student& operator=(const Student&);
            ~Student();

            friend std::ostream& operator<<(std::ostream&,
            ↳Student&);
    };
}
```

We can place as many separate sections of code – even code in different files – in the same namespace. And, in the case of the *Student* class, we'll also need to place the contents of *Student.cpp* in our *dsk* namespace:

```
#include "Student.h"
#include <iostream>

namespace dsk
{
    Student::Student(const char* n, char s, bool at)
    {
        name = new char[strlen(n) + 1]; //Extra byte for final
        ↳null
        strcpy_s(name, strlen(n)+1,n);
        s = toupper(s);
        if (s == 'M' || s == 'F')
            sex = s;
        attending = at;
    }

    //Code for other operations goes here

    //*** Display contents of student object ***
    std::ostream& operator<<(std::ostream& ct, Student& st)
    {
        ct << st.name << "  " << st.sex << "  ";
        if (st.attending)
            ct << "attending  ";
        else
            ct << "not attending  ";
        for (int ex = 0; ex < 6; ex++)
            ct << st.marks[ex] << "  ";
        ct << st.GetAverageMark() << std::endl;
        return ct;
    }
}
```

It is possible to specify that the methods of a class belong to a namespace without enclosing their code in braces. As an alternative to the above, we could prefix the first

line of each method's code with the namespace id and the scope resolution operator:

```
#include "Student.h"
#include <iostream>

dsk::Student::Student(const char* n, char s, bool at)
{
   ...
}
   ...
dsk::std::ostream& operator<<(std::ostream& ct, Student& st)
{
   ...
}
```

Generally, the first approach is preferred.

Using Items from a Namespace

When we come to writing an application in which we wish to use *Student* class objects, we now need to specify the namespace when creating our object.

One approach is to use the line

```
using namespace dsk;
```

at the start of our program but this approach negates the point of using namespaces. Instead, we need to add the prefix `dsk::` when declaring our *Student* class object:

```
dsk::Student st1("Margaret Adams",'F');
```

But once we have our object, the namespace details may be ignored:

```
cout << st1.GetMark(2) << endl;
cout << st1 << endl;
```

Activity 17.24

Modify the *ChronoLib* and *StudentLib* source code files so that all of the classes are defined within the *dsk* namespace.

Make the necessary changes to the *InSolution* and *OutSolution* projects so that they compile and run correctly.

Summary

- There are two types of libraries: static libraries and dynamic link libraries.
- Static library code is inserted into the final executable code by the linker.
- Dynamic link libraries are linked into a program during execution.
- A simple static library will normally consist of:
  1) a single header file which contains the declaration of all classes.
  2) multiple source code (*.cpp*) files – one for each class.
  2) multiple *.obj* files – one for each class within the library.

Creating a Library in Visual Studio

- Create a library file using Windows Desktop Wizard.
- Create a single solution to contain all your library files.
- Create a header file for each project in the solution declaring all the classes in that project.
- Create a implementation file for each class in the project.
- Build each class to create the *.obj* files.

Visual Studio Applications in the Library Solution

- Make sure the required library solution is loaded.
- From Solution Explorer select **Add > New Project**.
- Set up the new project as a *Console Application*.
- Create a source file and enter the app's code.
- From the app's popup menu, set as the start up project.
- In Solution Explorer, select the app's **Add** references and check the names of all libraries being used.
- Set app's **Properties>C/C++>General>Additional Include Directories** and add the paths to the libraries used.
- Build and run the app.

Visual Studio Applications not in the Library Solution

- Create a new *Console Application* project in its own solution.
- Create a source file and enter the app's code.
- Set app's **Properties>C/C++>General>Additional Include Directories** add the paths to the libraries used.
- Set app's **Properties>Linker>General>Additional Library Directories** and add the paths to the *.obj* files used.
- Add to the app's **Properties>Linker>Input>Additional Dependencies** the names of each *.lib* files used.
- Build and run the app.

Solutions

Activity 17.1

No solution required.

Activity 17.2

Code for *CoordsType.h*:

```
//#pragma once
struct CoordsType
{
  double x;
  double y;
};

CoordsType SetCoords(const double, const double);
```

Activity 17.3

Code for *CoordsType.cpp*:

```
CoordsType SetCoords(const double xv,
↳const double yv)
{
  CoordsType result;
  result.x = xv;
  result.y = yv;
  return result;
}
```

The main program file (*Source.cpp*) should now contain the following code:

```
#include <iostream>


int main()
{
  CoordsType point;
  point = SetCoords(3, 9);
  std::cout << '(' << point.x << ',' << point.y <<
  ↳')' << std::endl;
}
```

Activity 17.4

Modified code for *CoordsType.cpp*:

```
#include "CoordsType.h"

CoordsType SetCoords(const double xv,
↳const double yv)
{
  CoordsType result;
  result.x = xv;
  result.y = yv;
  return result;
}
```

Modified code for *Source.cpp*:

```
#include <iostream>
#include "CoordsType.h"

int main()
{
  CoordsType point;
  point = SetCoords(3, 9);
  std::cout << '(' << point.x << ',' << point.y <<
  ↳')' << std::endl;
}
```

The red underlines (when using Visual Studio) now disappear and the program executes correctly, displaying

(3,9)

Activity 17.5

Using angled brackets in the `#include` statement for *CoordsType.h* causes the compilation to fail since the preprocessor will no longer look in the current folder for that header file.

Activity 17.6

Code for *LineType.h*:

```
//#pragma once
#include "CoordsType.h"

struct LineType
{
  CoordsType start;
  CoordsType end;
};

LineType SetLine(CoordsType, CoordsType);
double LineLength(LineType);
```

Code for *LineType.cpp*:

```
#include <iostream>
#include "LineType.h"

LineType SetLine(CoordsType s, CoordsType e)
{
  LineType result;
  result.start = s;
  result.end = e;
  return result;
}


double LineLength(LineType ln)
{
  double result = sqrt((ln.end.x - ln.start.x)*(
  ↳ln.end.x - ln.start.x) + (ln.end.y - ln.start.y)
  ↳*(ln.end.y - ln.start.y));
  return result;
}
```

Code for *Source.cpp*:

```
#include <iostream>
#include "CoordsType.h"
#include "LineType.h"

int main()
{
  CoordsType st{ 0,0 };
  CoordsType en{ 10,0 };
  LineType line{ st, en };
  double length = LineLength(line);
  std::cout << length << std::endl;
}
```

The program fails to compile and produces several error messages.

Activity 17.7

With the `#pragma once` lines reinstated, the program now compiles and runs.

Activity 17.8

There are three *.obj* files in the project's *Debug* folder.

Activity 17.9

No solution required.

Activity 17.10

No solution required.

Activity 17.11

Modified contents for *Date.h*:

```
#pragma once
#include <iostream>
//*** Class declaration ***
class Date
{
private:
   int day;
   int month;
   int year;


   long ToJDN();
   Date FromJDN(long);

public:
   static const int DAYSINYEAR = 365;
   static int count;
   explicit Date(int = 1, int = 1, int = 2001);
   void SetDate(int, int, int);
   int GetDayOfWeek();
   bool IsLeapYear();
   int GetDay();
   int GetMonth();
   int GetYear();
   Date operator+(int);
   int operator-(Date);
   Date operator-(int);
   bool operator==(Date);
   void operator+=(int);
   Date operator++();
   Date operator++(int);
   Date operator--();
   Date operator--(int);
   int GetDaysInYear();
   static int GetDAYSINYEARconst();
   static int GetCount();
   ~Date();

   friend std::ostream& operator<<(std::ostream&,
   ↳Date&);
   friend std::istream& operator>>(std::istream&,
   ↳Date&);
   friend Date operator+(int, Date&);
};
```

Activity 17.12

Code for *Date.cpp*:

```
#include "Date.h"

int Date::count = 0;

//**************************************
//***        Date Class Methods      ***
//**************************************

//*** Initialise new Date objects to d/m/y ***
Date::Date(int d, int m, int y)
{
   // *** Days in each month of year ***
   int daysinmonth[] = { 0, 31,28,31, 30,31,30,31,
   ↳31,30,31,30,31 };

   bool valid = true;  // Parameters valid

   //*** If month invalid, invalid parameters ***
   if (m < 1 || m > 12)
      valid = false;
   //*** Add 1 to days in February if leap year ***
   daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0 &&
   ↳y % 100 != 0);
   //*** If days invalid, invalid parameters ***
   if (d <1 || d > daysinmonth[m])
      valid = false;
   //*** If year less than 1, invalid parameters ***
   if (y < 1)
      valid = false;
   //*** Assign date ***
   if (valid)
   {
      day = d;
      month = m;
      year = y;
   }
   else
   {
      day = 1;
      month = 1;
      year = 2001;
   }
   //*** Add one to count of Date objects **
   count++;
}


// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
   // *** Days in each month of year ***
   int daysinmonth[] = { 0, 31,28,31, 30,31,30, 31,
   ↳31,30, 31,30,31 };
   //*** If month invalid, exit ***
   if (m < 1 || m > 12)
      return;
   //*** Add 1 to days in February if leap year ***
   daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0 &&
   ↳y % 100 != 0);
   //*** If days in month invalid, exit ***
   if (d <1 || d > daysinmonth[m])
      return;
   //*** If year less than 1, exit ***
   if (y < 1)
      return;
   //*** Assign date ***
   day = d;
   month = m;
   year = y;
}


// *** Returns the day of the week of a date ***
int Date::GetDayOfWeek()
{
   int M, modifiedyear, C, Y;
   //*** Calculate M ***
   M = (month + 9) % 12 + 1;
   //*** Calculate modified year ***
   modifiedyear = year - M / 11;
   //*** Calculate C ***
   C = modifiedyear / 100;
   //*** Calculate Y ***
   Y = modifiedyear % 100;
   //*** Calculate day of week ***
   int dayofweek = ((static_cast<int>(2.6 * M - 0.2)
   ↳+ day + Y + Y / 4 + C / 4 - 2 * C) % 7 + 7) % 7;
   return dayofweek;
}


// *** Returns true if the date's year is a leap
year ***
bool Date::IsLeapYear()
{
   return ((year % 400 == 0) || (year % 4 == 0 &&
year % 100 != 0));
}


//*** Returns the value in attribute day ***
int Date::GetDay()
{
   return day;
}


//*** Returns the value in attribute month ***
int Date::GetMonth()
{
   return month;
}


//*** Returns the value in attribute year ***
int Date::GetYear()
{
   return year;
}


// *** Returns the JDN of the date ***
long Date::ToJDN()
{
   int a = (14 - month) / 12;
   int m = month + 12 * a - 3;
   int y = year + 4800 - a;
```

```
  long result = day + (153 * m + 2) / 5 + 365 * y +
  ↳y / 4 - y / 100 + y / 400 - 32045;
  return result;
}


// *** Sets date to eqivalent of JDN ***
Date Date::FromJDN(long jdn)
{
  int f = jdn + 1401 + (((4 * jdn + 274277) /
  ↳146097) * 3) / 4 - 38;
  int e = 4 * f + 3;
  int g = (e % 1461) / 4;
  int h = 5 * g + 2;
  Date d;
  d.day = (h % 153) / 5 + 1;
  d.month = (h / 153 + 2) % 12 + 1;
  d.year = e / 1461 - 4716 + (12 + 2 - d.month)/12;
  return d;
}


//*** Returns a Date calculated by adding d days to
↳date ***
Date Date::operator+(int d)
{
  Date dt = FromJDN(ToJDN() + d);
  return dt;
}


//*** Returns the difference in days between two
↳dates ***
int Date::operator-(Date dt)
{
  int result = ToJDN() - dt.ToJDN();
  return result;
}


//*** Returns a Date calculated by subtracting d
↳days to date ***
Date Date::operator-(int d)
{
  Date dt = FromJDN(ToJDN() - d);
  return dt;
}


//*** Returns true if two dates are equal ***
bool Date::operator==(Date dt)
{
  return (ToJDN() == dt.ToJDN());
}


//*** Adds a given number of days to the date ***
void Date::operator+=(int d)
{
  *this = FromJDN(ToJDN() + d);
}


//*** Adds one day to date, returns new date ***
Date Date::operator++()
{
  *this = FromJDN(ToJDN() + 1);
  return *this;
}


//*** Adds one day to date, returns original date
↳***
Date Date::operator++(int)
{
  Date result = *this;
  *this = FromJDN(ToJDN() + 1);
  return result;
}


//*** Subtracts one day from date, returns new date
↳***
Date Date::operator--()
{
  *this = FromJDN(ToJDN() - 1);
  return *this;
}


//*** Subtracts one day from date, returns original
↳date ***
Date Date::operator--(int)
{
  Date result = *this;
  *this = FromJDN(ToJDN() - 1);
  return result;
}


//*** Returns number of days in year ***
int Date::GetDaysInYear()
{
  return DAYSINYEAR + IsLeapYear();
}


//*** Returns the value of DAYSINYEAR ***
int Date::GetDAYSINYEARconst()
{
  return DAYSINYEAR;
}


//*** Returns the number of Date object existing ***
int Date::GetCount()
{
  return count;
}


Date::~Date()
{
  //*** Decrement count of Date objects ***
  count--;
}


//*** Displays a Date ***
std::ostream& operator<<(std::ostream& ct, Date& dt)
{
  ct << dt.day << '/' << dt.month << '/' << dt.year;
  return ct;
}


//*** Accepts a Date value from the keyboard ***
std::istream& operator>>(std::istream& cn, Date& dt)
{
  int d, m, y;
  //*** Read date ***
  scanf("%d / %d / %d", &d, &m, &y);
  //*** Transfer values to date object ***
  dt.SetDate(d, m, y);
  return cn;
}


Date operator+(int days, Date& d)
{
  Date dt = d.FromJDN(d.ToJDN() + days);
  return dt;
}
```

Activity 17.13

Modified code for *Source.cpp*:

```
#include <iostream>
#include "Date.h"
using namespace std;


int main()
{
  Date d1;
  cout << "Enter date : ";
  cin >> d1;
  cout << d1 << endl;
}
```

Activity 17.14

The new program does not compile. The file *Date.h* cannot be found by the preprocessor.

Activity 17.15

The program now compiles (though it still will not build or run).

Activity 17.16

With the linker details added, the program now builds and runs.

Activity 17.17

With the additional path details added, and the `#include` statement reduced to `#include "Date.h"`, the program still executes correctly.

Activity 17.18

Code for *StudentLib.h*:

```
#pragma once
#include <iostream>

  class Student
  {
    private:
      char* name{ nullptr };
      char sex{ '?' };
      bool attending{ true };
      int marks[6]{ 0 };
    public:
      Student(const char*, char, bool = true);
      Student(const Student&);
      void SetName(const char*);
      void SetSex(char);
      void SetAttending(bool);
      void SetMark(int, int);
      char* GetName() const;
      char GetSex() const;
      bool GetAttending() const;
      int GetMark(int) const;
      double GetAverageMark() const;
      Student& operator=(const Student&);
      ~Student();

      friend std::ostream& operator<<(
      ↳std::ostream&, Student&);
  };
```

Activity 17.19

Code for *Student.cpp*:

```
#include "StudentLib.h"
#include <iostream>

Student::Student(const char* n, char s, bool at)
{
  name = new char[strlen(n) + 1]; //Extra byte for
  ↳final null
  strcpy_s(name,strlen(n)+1,n);
  s = toupper(s);
  if (s == 'M' || s == 'F')
    sex = s;
  attending = at;
}

Student::Student(const Student& st)
{
  //*** Delete any space already being used ***
  if (name != nullptr)
    delete[]name;
  //*** Set up space for copy of name ***
  int size = strlen(st.name) + 1;
  name = new char[size];
  //*** Copy name ***
  strcpy_s(name,size,st.name);
  //*** Copy sex ***
  sex = st.sex;
  //*** Copy attending ***
  attending = st.attending;
  //*** Copy marks ***
  for (int c = 0; c < 6; c++)
    marks[c] = st.marks[c];
}

//*** Set the student's name to n ***
void Student::SetName(const char * n)
{
  //*** If no new name, exit ***
  if (n == nullptr)
    return;
  //*** Delete any space already being used ***
  if (name != nullptr)
    delete[]name;
  //*** Create enough space for the new name ***
  name = new char[strlen(n) + 1]; //Extra byte for
  ↳final null
  //*** Copy n into this new space ***
  strcpy_s(name,strlen(n)+1,n);

}


//*** Set student's sex to s ***
void Student::SetSex(char s)
{
  s = toupper(s);
  if (s == 'M' || s == 'F')
    sex = s;
}


//*** Set student's attendence status ***
void Student::SetAttending(bool at)
{
  attending = at;
}


//*** Set student's mark for specified exam ***
void Student::SetMark(int mrk, int ex)
{
  //*** If not valid exam, return ***
  if (ex < 0 || ex > 5)
    return;
  //*** If not valid mark, return ***
  if (mrk < 0 || mrk > 100)
    return;
  //*** Store mark ***
  marks[ex] = mrk;
}


//*** Return student's name ***
char* Student::GetName() const
{
  return name;
}


//*** Return student's sex ***
char Student::GetSex() const
{
  return sex;
}


//*** Return student's attendence status ***
bool Student::GetAttending() const
{
  return attending;
}


//*** Return student's exam mark ***
int Student::GetMark(int ex) const
{
  if (ex < 0 || ex > 5)
    return -1;
  return marks[ex];
}


//*** Return student's average mark ***
double Student::GetAverageMark() const
{
  double total{ 0.0 };
  for (auto v : marks)
    total += v;
  return total / 6.0;
}


//*** Copies a Details object to current Details
↳object ***
```

```
Student& Student::operator=(const Student& st)
{
  //*** Delete any space already being used ***
  if (name != nullptr)
    delete[]name;
  //*** Set up space for copy of name ***
  int size = strlen(st.name) + 1;
  name = new char[size];
  //*** Copy name ***
  strcpy_s(name,size,st.name);
  //*** Copy sex ***
  sex = st.sex;
  //*** Copy attending ***
  attending = st.attending;
  //*** Copy marks ***
  for (int c = 0; c < 6; c++)
    marks[c] = st.marks[c];
  //*** Return reference to updated object ***
  return *this;
}


Student::~Student()
{
  if (name != nullptr)
    delete[] name;
}


//*** Display contents of student object ***
std::ostream& operator<<(std::ostream& ct, Student&
st)
{
  ct << st.name << "  " << st.sex << "  ";
  if (st.attending)
    ct << "attending  ";
  else
    ct << "not attending  ";
  for (int ex = 0; ex < 6; ex++)
    ct << st.marks[ex] << "  ";
  ct << st.GetAverageMark() << std::endl;
  return ct;
}
```

Activity 17.20

No solution required.

Activity 17.21

Code in *ChronoLib.h*:

```
#pragma once
#include <iostream>
//*** Date class declaration ***
class Date
{
private:
  int day;
  int month;
  int year;

  long ToJDN();
  Date FromJDN(long);

public:
  static const int DAYSINYEAR = 365;
  static int count;
  explicit Date(int = 1, int = 1, int = 2001);
  void SetDate(int, int, int);
  int GetDayOfWeek();
  bool IsLeapYear();
  int GetDay();
  int GetMonth();
  int GetYear();
  Date operator+(int);
  int operator-(Date);
  Date operator-(int);
  bool operator==(Date);
  void operator+=(int);
  Date operator++();
  Date operator++(int);
  Date operator--();
  Date operator--(int);
  int GetDaysInYear();
  static int GetDAYSINYEARconst();
  static int GetCount();
  ~Date();
  friend std::ostream& operator<<(std::ostream&,
  ↳Date&);
  friend std::istream& operator>>(std::istream&,
  ↳Date&);
  friend Date operator+(int, Date&);
};


//*** Time class declaration ***
class Time
{
private:
  int hour;
  int minute;
  int second;

  int ToSeconds() const;
  Time FromSeconds(int) const;

public:
  Time(int = 0, int = 0, int = 1);
  void SetTime(int, int, int);
  int GetHour() const;
  int GetMinute() const;
  int GetSecond() const;
  Time operator+(const Time&) const;
  Time& operator+=(const Time&);
  Time operator-(const Time&) const;
  Time& operator-=(const Time&);
  Time& operator++();
  Time operator++(int);
  Time& operator--();
  Time operator--(int);
  bool operator== (const Time&)const;
  bool operator!= (const Time&)const;
  bool operator<(const Time&)const;
  bool operator<=(const Time&)const;
  bool operator>(const Time&)const;
  bool operator>=(const Time&)const;

  friend std::ostream& operator<<(std::ostream&,
  ↳const Time&);
};
```

Code in *Date.cpp*:

```
#include "ChronoLib.h"

int Date::count = 0;

//**************************************
//***        Date Class Methods      ***
//**************************************

//*** Initialise new Date objects to d/m/y ***
Date::Date(int d, int m, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = { 0, 31,28,31, 30,31,30,31,
  ↳31,30,31,30,31 };

  bool valid = true;  // Parameters valid

  //*** If month invalid, invalid parameters ***
  if (m < 1 || m > 12)
    valid = false;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0 &&
  ↳y % 100 != 0);
  //*** If days invalid, invalid parameters ***
  if (d <1 || d > daysinmonth[m])
    valid = false;
  //*** If year less than 1, invalid parameters ***
  if (y < 1)
    valid = false;
  //*** Assign date ***
  if (valid)
  {
    day = d;
    month = m;
    year = y;
  }
  else
  {
    day = 1;
    month = 1;
    year = 2001;
  }
  //*** Add one to count of Date objects **
```

```
  count++;
}


// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
  // *** Days in each month of year ***
  int daysinmonth[] = { 0, 31,28,31, 30,31,30, 31,
  ↳31,30, 31,30,31 };
  //*** If month invalid, exit ***
  if (m < 1 || m > 12)
    return;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0 &&
  ↳y % 100 != 0);
  //*** If days in month invalid, exit ***
  if (d <1 || d > daysinmonth[m])
    return;
  //*** If year less than 1, exit ***
  if (y < 1)
    return;
  //*** Assign date ***
  day = d;
  month = m;
  year = y;
}


// *** Returns the day of the week of a date ***
int Date::GetDayOfWeek()
{
  int M, modifiedyear, C, Y;
  //*** Calculate M ***
  M = (month + 9) % 12 + 1;
  //*** Calculate modified year ***
  modifiedyear = year - M / 11;
  //*** Calculate C ***
  C = modifiedyear / 100;
  //*** Calculate Y ***
  Y = modifiedyear % 100;
  //*** Calculate day of week ***
  int dayofweek = ((static_cast<int>(2.6 * M - 0.2)
  ↳+ day + Y + Y / 4 + C / 4 - 2 * C) % 7 + 7) % 7;
  return dayofweek;
}


// *** Returns true if the date's year is a leap
↳year ***
bool Date::IsLeapYear()
{
  return ((year % 400 == 0) || (year % 4 == 0 &&
  ↳year % 100 != 0));
}

//*** Returns the value in attribute day ***
int Date::GetDay()
{
  return day;
}

//*** Returns the value in attribute month ***
int Date::GetMonth()
{
  return month;
}

//*** Returns the value in attribute year ***
int Date::GetYear()
{
  return year;
}

// *** Returns the JDN of the date ***
long Date::ToJDN()
{
  int a = (14 - month) / 12;
  int m = month + 12 * a - 3;
  int y = year + 4800 - a;
  long result = day + (153 * m + 2) / 5 + 365 * y +
  ↳y / 4 - y / 100 + y / 400 - 32045;
  return result;
}

// *** Sets date to eqivalent of JDN ***
Date Date::FromJDN(long jdn)
{
  int f = jdn + 1401 + (((4 * jdn + 274277) /
  ↳146097) * 3) / 4 - 38;
  int e = 4 * f + 3;
  int g = (e % 1461) / 4;
  int h = 5 * g + 2;
  Date d;
  d.day = (h % 153) / 5 + 1;
  d.month = (h / 153 + 2) % 12 + 1;
  d.year = e / 1461 - 4716 + (12 + 2 - d.month)/12;
  return d;
}

//*** Returns a Date calculated by adding d days to
↳date ***
Date Date::operator+(int d)
{
  Date dt = FromJDN(ToJDN() + d);
  return dt;
}

//*** Returns the difference in days between two
↳dates ***
int Date::operator-(Date dt)
{
  int result = ToJDN() - dt.ToJDN();
  return result;
}

//*** Returns a Date calculated by subtracting d
↳days to date ***
Date Date::operator-(int d)
{
  Date dt = FromJDN(ToJDN() - d);
  return dt;
}


//*** Returns true if two dates are equal ***
bool Date::operator==(Date dt)
{
  return (ToJDN() == dt.ToJDN());
}

//*** Adds a given number of days to the date ***
void Date::operator+=(int d)
{
  *this = FromJDN(ToJDN() + d);
}


//*** Adds one day to date, returns new date ***
Date Date::operator++()
{
  *this = FromJDN(ToJDN() + 1);
  return *this;
}

//*** Adds one day to date, returns original date
↳***
Date Date::operator++(int)
{
  Date result = *this;
  *this = FromJDN(ToJDN() + 1);
  return result;
}

//*** Subtracts one day from date, returns new date
↳***
Date Date::operator--()
{
  *this = FromJDN(ToJDN() - 1);
  return *this;
}

//*** Subtracts one day from date, returns original
↳date ***
Date Date::operator--(int)
{
  Date result = *this;
  *this = FromJDN(ToJDN() - 1);
  return result;
}


//*** Returns number of days in year ***
int Date::GetDaysInYear()
{
  return DAYSINYEAR + IsLeapYear();
}

//*** Returns the value of DAYSINYEAR ***
int Date::GetDAYSINYEARconst()
```

```
{
  return DAYSINYEAR;
}

//*** Returns the number of Date object existing ***
int Date::GetCount()
{
  return count;
}

Date::~Date()
{
  //*** Decrement count of Date objects ***
  count--;
}

//*** Displays a Date ***
std::ostream& operator<<(std::ostream& ct, Date& dt)
{
  ct << dt.day << '/' << dt.month << '/' << dt.year;
  return ct;
}

//*** Accepts a Date value from the keyboard ***
std::istream& operator>>(std::istream& cn, Date& dt)
{
  int d, m, y;
  //*** Read date ***
  scanf("%d / %d / %d", &d, &m, &y);
  //*** Transfer values to date object ***
  dt.SetDate(d, m, y);
  return cn;
}

Date operator+(int days, Date& d)
{
  Date dt = d.FromJDN(d.ToJDN() + days);
  return dt;
}
```

Code in *Time.cpp*:

```
#include "ChronoLib.h"

//************************************************
//***        Time Class Implementation         ***
//************************************************
***

//*********************************
//***      Public Methods       ***
//*********************************
//*** Initalise time to h:m:s if valid else 0:0:0
↳***
Time::Time(int h, int m, int s)
{
  if (h < 0 || h > 23 || m < 0 || m > 59 || s < 0 ||
  ↳s > 59)
  {
    hour = 0;
    minute = 0;
    second = 0;
  }
  else
  {
    hour = h;
    minute = m;
    second = s;
  }
}

//*** Sets time to h:m:s if this is valid ***
void Time::SetTime(int h, int m, int s)
{
  if (h < 0 || h > 23 || m < 0 || m > 59 || s < 0 ||
  ↳s > 59)
  {
    return;
  }
  else
  {
    hour = h;
    minute = m;
    second = s;
  }
}

//*** Returns hour value ***
int Time::GetHour() const
{
  return hour;
}


//*** Returns hour value ***
int Time::GetMinute() const
{
  return minute;
}


//*** Returns hour value ***
int Time::GetSecond() const
{
  return second;
}


//*** Returns time of t + current time ***
Time Time::operator+(const Time& t) const
{
  return FromSeconds(ToSeconds() + t.ToSeconds());
}


//*** Adds time t to current time ***
Time& Time::operator+=(const Time& t)
{
  return *this = FromSeconds(ToSeconds() +
  ↳t.ToSeconds());
  return *this;
}

//*** Returns current time  - t ***
Time Time::operator-(const Time& t) const
{
  return FromSeconds(ToSeconds() - t.ToSeconds());
}


//*** Subtracts time t from current time ***
Time& Time::operator-=(const Time& t)
{
  return *this = FromSeconds(ToSeconds() -
  ↳t.ToSeconds());
  return *this;
}


//*** Adds 1 sec to time (pre) ***
Time& Time::operator++()
{
  *this = FromSeconds(ToSeconds() + 1);
  return *this;
}


//*** Adds 1 sec to time (post) ***
Time Time::operator++(int)
{
  Time result{ *this };
  *this = FromSeconds(ToSeconds() + 1);
  return result;
}

//*** Subtracts 1 sec from time (pre) ***
Time& Time::operator--()
{
  *this = FromSeconds(ToSeconds() - 1);
  return *this;
}


//*** Subtracts 1 sec from time (post) ***
Time Time::operator--(int)
{
  Time result{ *this };
  *this = FromSeconds(ToSeconds() - 1);
  return result;
}


//*** Returns true if times are equal ***
bool Time::operator==(const Time& t) const
{
  return (ToSeconds() == t.ToSeconds());
}
```

```
//*** Returns true if times are not equal ***
bool Time::operator!=(const Time& t) const
{
  return (ToSeconds() != t.ToSeconds());
}


//*** Returns true if time < t ***
bool Time::operator<(const Time& t) const
{
  return (ToSeconds() < t.ToSeconds());
}


//*** Returns true if time <= t ***
bool Time::operator<=(const Time& t) const
{
  return (ToSeconds() <= t.ToSeconds());
}


//*** Returns true if time > t ***
bool Time::operator>(const Time& t) const
{
  return (ToSeconds() > t.ToSeconds());
}


//*** Returns true if time >= t ***
bool Time::operator>=(const Time& t) const
{
  return (ToSeconds() >= t.ToSeconds());
}


//*********************************
//***     Private Methods       ***
//*********************************

//*** Converts current time to seconds ***
int Time::ToSeconds() const
{
  return hour * 3600 + minute * 60 + second;
}


//*** Converts seconds to hrs, mins, secs ***
Time Time::FromSeconds(int secs) const
{
  if (secs < 0)
    return Time(0, 0, 0);
  int h = secs / 3600;
  int m = (secs - h * 3600) / 60;
  int s = secs % 60;

  return Time(h, m, s);
}

//*********************************
//***         Friends           ***
//*********************************

//*** Displays the value of time t ***
std::ostream& operator<<(std::ostream& ct,
↳const Time& t)
{
  char timestr[9];
  sprintf(timestr, "%02d:%02d:%02d", t.hour,
  ↳t.minute, t.second);
  ct << timestr;
  return ct;
}
```

The *.lib* files can be found in the folder *MyLibraries/Debug*.

Activity 17.22

The program should execute as expected.

Activity 17.23

The program should execute as expected.

Activity 17.24

Changes to *MyLibraries* header and implementation files:

existing #include lines

```
namespace dsk
{
    existing code
}
```

Changes to *OutSolution Source.cpp*:

```
#include <iostream>
#include "ChronoLib.h"
#include "StudentLib.h"

using namespace std;

int main()
{
  dsk::Date dt(23, 11, 1963);
  dsk::Time tm(12, 0, 0);
  dsk::Student st("Avril Tait", 'F');
  cout << "Date : " << dt << endl;
  cout << "Time : " << tm << endl;
  cout << "Student : " << st << endl;
}
```

18

Error Handling

In this Chapter:

- ❑ Types of Errors
- ❑ Traditional Error Avoidance and Handling
- ❑ Using assert()
- ❑ Exceptions: try, throw and catch
- ❑ catch Block Variables
- ❑ Multiple catch Blocks
- ❑ Throwing an Object
- ❑ A Universal catch Block
- ❑ Rethrowing an Exception
- ❑ Unhandled Exceptions

Error Handling

Introduction

Every line of code that we write introduces another opportunity for an error in a program. There are several types of errors that we are likely to encounter. These are:

syntax errors where we have made a mistake in the grammar of a statement or program structure.
Typically, this type of error will be created by omitting a required character or misspelling a word.
Luckily, these errors are automatically detected by the compiler.

compilation errors where the compiler detects an error such as a variable being declared multiple times or a function call with invalid parameters.

linker errors where the compiler cannot find a required resource such as a library file or a header file.

logic errors where the logic employed by a program is faulty.
Examples of this type of error can occur if a Boolean expression tests the wrong condition or where an arithmetic calculation makes use of the wrong formula.

runtime errors where an error occurs during the execution of a program.
Division by zero, trying to store a number which is too large for the space it has been allocated or running out of storage space are all situations which may cause a runtime error.

Logic and runtime errors can be more difficult to correct, requiring a combination of good debugging skills and **defensive programming** (where we try to anticipate potential problems and ensure that the program continues to operate in a predictable manner when problems occur).

By necessity, the amount of logic and runtime error detection included in this book's examples has been limited. If this had not been the case, the methods and techniques being demonstrated in the examples might have been obscured under a blanket of error detection code. However, we have already covered the main traditional methods of error elimination and detection:

User input validation
By using tightly controlled user-written input functions rather than simply using `cin` or `scanf()` to ensure input data is within the required range, we can reduce the likelihood of accepting invalid data from the user.

Function precondition checks
Placing an `if` statement to check that a function or method's precondition has been met at the start of a routine ensures that parameter values stay within the designed range.

Function result indicators

By returning a value which signifies the success or failure of a function's execution, the program calling the function can use that value to detect any problems that have arisen during the function's execution.

This last approach is used by almost all traditional C functions. Typically, zero is returned to indicate the success when a function is executed, while a non-zero value is used to give coded details of what has gone wrong. For example, the syntax for `fseek()` is

```
int fseek(FILE *stream, long offs, int start);
```

in which the returned value is set to zero if the seek operation is successful and a non-zero value if an error occurs.

Other Options



To use *assert()* a program must contain the line

#include<assert.h>

assert()

The `assert()` function evaluates the parameter specified (usually this will be a Boolean expression). If the result of that evaluation is not zero, then program execution continues; if the result is zero, then the program terminates.

When a program is terminated by an `assert()` call, an error message will be displayed. Exactly what that message is depends on the compiler implementation, but generally it will be of the form

Assertion failed: expression, filename, line number

where:

expression	is the expression given as a parameter to `assert()`.
filename	is the name of the source code file containing the assert command.
line number	is the line number within the program of the `assert()` call that failed.

The program in FIG-18.1 uses the `assert` statement to ensure that an attempt to divide by zero does not occur.

FIG-18.1

Using *assert()*

```
#include <iostream>
#include <assert.h>
using namespace std;

void main()
{
   float no1, no2, result;

   //*** Read in two values ***
   cout << "Enter first number: ";
   cin >> no1;
   cout << "Enter second number: ";
   cin >> no2;
```



FIG-18.1
(continued)

Using *assert()*

```
    assert(no2 != 0);
    //*** Perform division ***
    result = no1 / no2;
    //*** Display result ***
    cout << no1 << '/' << no2 << '=' << result << endl;
}
```

If the second value entered was zero, then the Visual Studio would display the message shown in FIG-18.2 as well as the window shown in FIG-18.3.

FIG-18.2

An *assert()* Fail

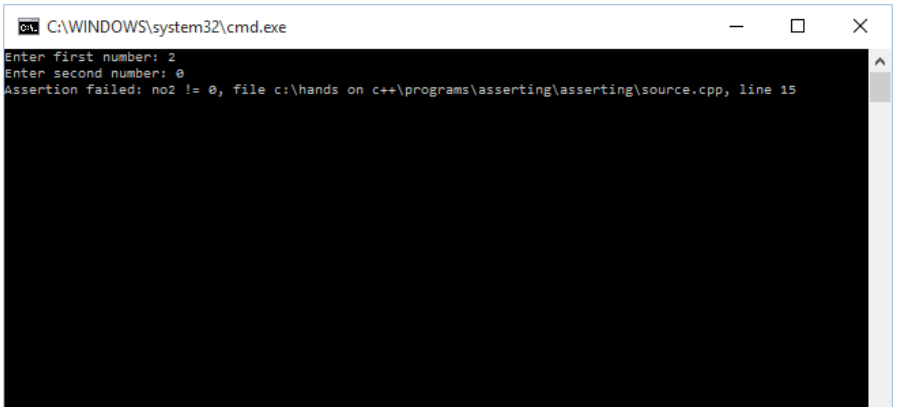
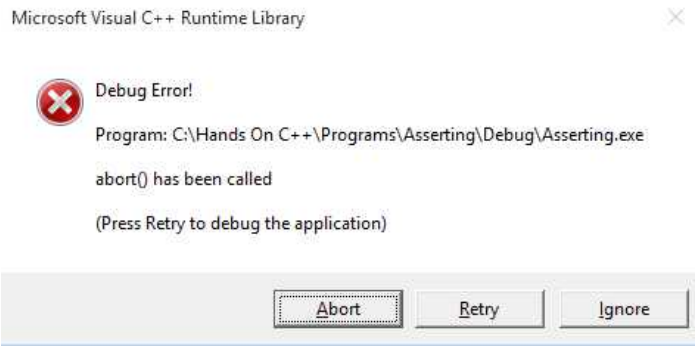


FIG-18.3

An *assert()* Fail Dialog Box



In practice `assert` statements are usually added to a program during debug to help find conditions which have unexpected results. After the problems have been located and corrected, `assert` statements are removed before the program is rebuilt.

Rather than physically removing the `assert` statements from a program, a simpler option is to include the line

```
#define NDEBUG
```

as the first directive in the source code. This will cause the `assert()` calls to be ignored during the compilation process..

Activity 18.1

Create a new project, *UsingAssert*, and implement the code given in FIG-18.1 and test it using the values 12 and 0.

Add the `#define NDEBUG` directive and check that the error is not caught in the same way when the values 12 and 0 are used for a second time.

Maintaining an Error Flag

As already stated, a function often returns an `int` value to indicate how successful it has been. Sometimes, however, it is not possible to do this. For example, when overloading a C++ operator such as `+` or `==`, we cannot return an error indicator since the overloaded function will have to return the result produced by the operation.

As an alternative approach, we can create an error status attribute within a `struct` or a `class` declaration. For example,

```
class MyClass
{
   private:
      int error_status;
         ...
   public:
      int GetStatus();
         ...
```

This can then be set by the methods of the class and subsequently interrogated by the application programmer.



We'll learn more about the *iostream* class in a later chapter.

In fact, this is exactly the approach used by the `iostream` class which maintains a `status` attribute whose bits are set by various methods in the class and can be interrogated by other methods such as `bad()`, `good()`, `eof()`, etc.

The main disadvantage of this approach is that it depends on the application programmer interrogating the error status attribute to determine a previous operation's success. As we saw in a previous chapter, it's tempting to miss out such checks. On the other hand, if error checking is comprehensive, the resulting code can be difficult to read, thereby obscuring the core logic of the program.

Summary

- Syntax, compilation and linker errors are relatively easy to fix.
- Logic and runtime errors can be more difficult to correct and require the use of defensive programming and good debugging skills.
- Input validation attempts to ensure that data entered by the user is valid.
- Adding an `if` statement to handle a function's precondition ensures that parameters and other values are within an acceptable range.
- Returning an error indicator from a function allows the calling code to check the success of the function's execution.
- An error status field can be defined within a `struct` or `class` to record the success of a function's execution.
- The `assert` instruction can be used to check the truth of an expression during debugging.
- To use the `assert` instruction, add the line `#include <assert.h>`.
- The `assert` instruction can be ignored during compilation by including the line
  `#define NDEBUG`
  at the start of a program.

Exception Handling

Introduction

Another way of dealing with runtime errors is to use **exception handling**. When an error occurs, exception handling transfers control to a separate part of the program which has been created specifically to handle the error.

Using this approach, when a section of code detects an error (such as division by zero, an out-of-bounds array access, or perhaps a pre-condition not met), it **throws** an **exception**. Control is then transferred to a block of code known as an **exception handler** or **catch block** which should contain the necessary code to deal with the problem that has arisen. If no exception handler is defined for the type of exception that has occurred, the program will terminate.

Exception Handling in C++

C++ uses three key words when coding exception handling. These are `try`, `throw` and `catch`.

The keyword `try` is used to mark the start of a block of code which contains the logic that the programmer knows has the potential to cause an error.

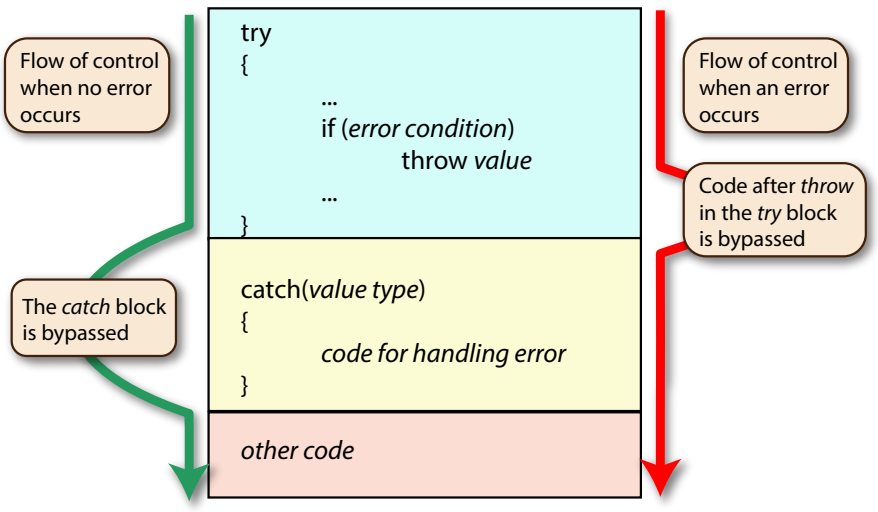
The term `throw` is used to throw the exception when a runtime problem is detected. Typically, this will be placed within an `if` statement. The `throw` statement will return a copy of a specified value. If the tested-for runtime error occurs, the remainder of the try block is ignored and control is transfered to the **catch block**.

The term `catch` is used to mark the start of the block of code designed to handle the exception that has been thrown. The `catch` statement will specify the type of value thrown by the exception. This block of code will be ignored if the exception it is designed to catch does not occur.

FIG-18.4 shows the flow of control through a simple program which employs exception handling.

FIG-18.4

Flow of Control and Exceptions



The program in FIG-18.5 is designed to display the square root of a number entered by the user. The program throws an exception if the number entered is less than zero.

FIG-18.5

Throwing an Exception

```
#include <iostream>
using namespace std;

void main()
{
   double num;

   //*** Get number ***
   cout << "Enter a number : ";
   cin >> num;
   //*** Try to find the square root ***
   try
   {
      //*** Throw an exception if num < 0 ***
      if (num < 0)
         throw -1;
      cout << "Square root of " << num << " is " << sqrt(num) <<
      ↳endl;
   }
   catch (int) //catches any int value thrown
   {
      cout << "Tried to find the square root of a negative value\n";
   }
   //*** Program finished message ***
   cout << "Program complete\n";
}
```



A catch block must be placed immediately after the try block it is designed to work with.

Notice that the value thrown by the exception is an `int` (-1) and that the `catch` statement specifies a type rather than a value.

Activity 18.2

Create a new project called *Except01* and implement the code given in FIG-18.5 and test it using the values 9 and -9.

Activity 18.3

Create a new project, *Except02*, which reads in two floating point numbers and attempts to display the result produced when the first number is divided by the second. Throw an exception if the second number is zero. Test your code with the values 12.6 and 3.2 then 12.6 and 0.0.

Local Values in the try Block

When an exception is thrown, any variables or objects which are local to the try block have their space deallocated and, in the case of objects, their destructor method will be executed.

Program Termination

If an exception is thrown by code that is not enclosed in a try block, the program will terminate. We'll discuss this more at the end of this chapter.

If there is no `catch` block for the type of value thrown by an exception, the program will terminate.

Empty *catch* Block

It is valid for a catch block to be empty:

```
catch(int)
{}
```

And although this would seem pointless, there are times, at the early development stage of a program that this can be useful to ensure that a program does not terminate when an exception of the specified type is thrown.

Throwing an Exception from a Function

Although, as we have seen, an exception can be thrown from within `main()`, this is not normally the way in which an exception is thrown. The most likely practice is to throw an exception from within a function or a class method.

When an exception is thrown from within a function (or method), several things happen:

- The remainder of the function is bypassed.
- All local variables and objects created within the function are destroyed.
- Control returns to the code which called the function.

The program in FIG-18.6 is a variation on that given in FIG-18.5 with the square root calculation and exception being coded within a separate function.

FIG-18.6

Throwing an Exception from a Function

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
double CalcSquareRoot(float);


void main()
{
   double num;
   //*** Get number ***
   cout << "Enter a number : ";
   cin >> num;
   //*** Try to find the square root ***
   try
   {
      cout << "Square root of " << num << " is " <<
      ↳CalcSquareRoot(num) << endl;
   }
   catch (int)
   {
      cout << "Tried to find the square root of a negative value\n";
   }
   //*** Program finished message ***
   cout << "Program complete\n";
}
```



FIG-18.6
(continued)

Throwing an Exception from a Function

```
//*** Calculates square root of v ***
double CalcSquareRoot(double v)
{
   if (v < 0)
      throw -1;
   return sqrt(v);
}
```

Activity 18.4

Modify *Except01* to match the code given in FIG-18.6 and test it using the values 9 and -9.

By pressing the *F11* key (which, in Visual Studio single steps through the program's code) observe which statements are executed when using the two test values.

Another situation which is often handled by throwing an exception is when the pre-condition of a function or class method is not met. For example, our *Date* class has a method called *SetDate()* which is used to change the date stored in a *Date* object. Let's assume we want to throw an integer value of -2 if the parameters to that method do not form a valid date. Our code for the method would be

```
// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
   // *** Days in each month of year ***
   int daysinmonth[] = { 0, 31,28,31, 30,31,30, 31,31,30, 31,
   ↳30,31 };
   //*** If month invalid, throw exception ***
   if (m < 1 || m > 12)
      throw -2;
   //*** Add 1 to days in February if leap year ***
   daysinmonth[2] += (y % 400 == 0)||(y % 4==0 && y % 100!=0);
   //*** If days in month invalid, throw exception ***
   if (d <1 || d > daysinmonth[m])
      throw -2;
   //*** If year less than 1, throw exception ***
   if (y < 1)
      throw -2;
   //*** Assign date ***
   day = d;
   month = m;
   year = y;
}
```

Activity 18.5

Load up the solution *MyLibraries* and in project *ChronoLib* modify the code for the *Date* class's *SetDate()* to match that given above.

In the *Time* class's *SetTime()* method, change the code so that invalid parameters cause the method to throw the value -3.

Rebuild the *MyLibraries* solution.

The program in FIG-18.7 can be used to check that the *SetDate()* exception is thrown appropriately.

FIG-18.7

Handling *SetDate()*'s Exception

```
#include <iostream>
#include <ChronoLib.h>

using namespace std;

int main()
{
   dsk::Date dt1;
   //*** Get date to be stored ***
   int d, m, y;
   cout << "Enter date to be stored : ";
   scanf_s("%d/%d/%d", &d, &m, &y);
   try
   {
      //*** Store and display date ***
      dt1.SetDate(d,m,y);
      cout << dt1 << endl;
   }
   catch (int)
   {
      //*** Display error message and set date to default ***
      cout << "Date assignment invalid\n";
      cout << "Date set to 1/1/2001\n";
      dt1.SetDate(1, 1, 2001);
   }
   //*** Display date ***
   cout << "Date is " << dt1 << endl;
}
```

Activity 18.6

Create a new project called *ClassExceptions* and implement the code given above.

In order to have the new project compile and link, set the following project properties:

C/C++>General>Additional Include Directories to
C:\Hands On C++\Programs\MyLibraries\ChronoLib

Linker>General>Additional Library Directories to
C:\Hands On C++\Programs\MyLibraries\Debug

Linker>Input>Additional Dependencies to
ChronoLib.lib

Run the program and check its response to both a valid and invalid date.

Modify the program to read in the value for three more variables and use these to set the time of a new *Time* object, *tm1*. Do not change the catch block code.

What happens when a valid date but invalid time are entered?

The reason that an invalid time is reported as an invalid date is easy enough to see. For, although *SetDate()* throws the value -2 and *SetTime()* throws -3, both of these values are of type `int` and will therefore be caught by the same catch block.

Using a *catch* Variable

We can, if we wish, add a variable name to the type specified in the `catch` line. For instance, in the previous example, we might start the catch block with the line

```
catch(int no)
```

When we use this approach, the named variable will hold a copy of the value that has been thrown (in this case, -2 or -3).

This would allow us to rewrite the handler so that it displays the type of error that has occurred:

```
catch (int no)
{
   //*** Handle invalid Date ***
   if (no == -2)
   {
      //*** Display error message and set date to default ***
      cout << "Date assignment invalid\n";
      cout << "Date set to 1/1/2001\n";
      dt1.SetDate(1, 1, 2001);
   };
   //*** Handle invalid Time ***
   if (no == -3)
   {
      //*** Display error message and set time to default ***
      cout << "Time assignment invalid\n";
      cout << "Time set to 0:0:0\n";
      tm1.SetTime(0,0,0);
   };
}
```

Activity 18.7

Modify *ClassExceptions* by re-coding the catch block as shown above.

Test the changes by entering first an invalid date and then a valid date and invalid time.

Throwing and Catching Different Types

An alternative approach to handling different exceptions is to have the exceptions throw different types of values. For example, we might rewrite our *SetTime()* method to throw a `double` by changing its `throw` statement to

```
throw -3.0;
```

Since the value thrown by the new version of our function is no longer an `int`, it will not be caught by the existing catch block in `main()`. However, we are allowed to have as many catch blocks as we wish with each block catching a different type of value. So all we need to handle any exception thrown by the latest version of *SetTime()* is to add a `catch(double)` block. The code for the updated version of the `catch(int)` block and the new `catch(double)` block is shown below:

```
catch (int)
{
   //*** Handle invalid Date ***
   //*** Display error message and set date to default ***
   cout << "Date assignment invalid\n";
```

```
      cout << "Date set to 1/1/2001\n";
      dt1.SetDate(1, 1, 2001);
   };
   catch(double)
   {
      //*** Handle invalid Time ***
      //*** Display error message and set time to default ***
      cout << "Time assignment invalid\n";
      cout << "Time set to 0:0:0\n";
      tm1.SetTime(0,0,0);
   };
```

Activity 18.8

In solution *MyLibraries*, modify the code for *SetTime()* in *Time.cpp* in project *ChronoLib*, so that it throws a `double` value.

Modify *ClassExceptions* by coding two catch blocks as shown above.

Test the changes by entering first an invalid date and then a valid date and invalid time.

Throwing Objects

If functions were only to throw standard data type values such as `int` and `double`, we would soon run out of options if we had a number of functions each of which wanted to throw a different type. There's also the problem that existing library functions could be throwing the same exception types as our own routines, making it difficult to ensure all exceptions are correctly handled.

However, we can solve this problem by creating and throwing our own error class objects.

C++ allows us to declare a class which has no properties. For example,

```
class TimeError{};
```

and

```
class DateError{};
```

The code below shows the changes needed to *ChronoLib.h*, *Date.cpp* and *ClassExceptions*' *Source.cpp* in order to throw and catch a *DateError* object.

In *ChronoLib.h* (at start of file):

```
#pragma once
#include <iostream>
namespace dsk
{
   //*** DateError class ***
   class DateError{};
```

In *SetDate()*:

```
// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
   // *** Days in each month of year ***
      int daysinmonth[] = { 0, 31,28,31, 30,31,30, 31,31,30,
      ↳31,30,31 };
```

```
        //*** If month invalid, throw exception ***
        if (m < 1 || m > 12)
           throw DateError();
        //*** Add 1 to days in February if leap year ***
        daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0 && y %
        ↳100 != 0);
        //*** If days in month invalid, throw exception ***
        if (d <1 || d > daysinmonth[m])
           throw DateError();
        //*** If year less than 1, throw exception ***
        if (y < 1)
           throw DateError();
        //*** Assign date ***
        day = d;
        month = m;
        year = y;
     }
```

In *ClassExceptions (Source.cpp)*:



Note the need to add the namespace when using the *DateError* class since it is defined within that namespace.

```
catch (dsk::DateError)
{
   //*** Handle invalid Date ***
   //*** Display error message and set date to default ***
   cout << "Date assignment invalid\n";
   cout << "Date set to 1/1/2001\n";
   dt1.SetDate(1, 1, 2001);
};
```

Activity 18.9

In solution *MyLibraries*, modify the code for *ChronoLib.h* and *Date.cpp* so that *SetDate()* throws a *DateError* object if the parameters are invalid.

Modify the code for *ChronoLib.h* and *Time.cpp* so that *SetTime()* throws a *TimeError* object if the parameters are invalid.

Modify *ClassExceptions' Source.cpp* so that it catches *DateError* and *TimeError* objects.

Test the changes by entering first an invalid date and then a valid date and invalid time.

Classes designed specifically for exception handling don't have to be empty, they can have all the features of a normal class such as attributes, methods, constructors, etc. As an example, we will create a longer version of the *DateError* class which contains fields to hold the invalid date we attempted to assign.

```
class DateError
{
   public:
      int wrongday;
      int wrongmonth;
      int wrongyear;

      DateError(int d, int m, int y)
      {
         wrongday = d;
         wrongmonth = m;
         wrongyear = y;
      }
};
```

Using an object of this class, we can throw an object from *SetDate()* which returns the values of the invalid parameters. All of the `throw` statements becoming:

```
throw DateError(d,m,y);
```

This allows the application program to gain access to more information about the error. For example, *HandleExceptions' Source.cpp* catch block could be written as

```
catch (dsk::DateError de)
   {
      //*** Handle invalid Date ***
      //*** Display error message and set date to default ***
      cout << "Date assignment invalid.\n";
      cout << "Attempted to set date to " << de.wrongday << '/'
      ↳<< de.wrongmonth << '/' << de.wrongyear << endl;
      cout << "Date set to 1 / 1 / 2001\n";
      dt1.SetDate(1, 1, 2001);
   }
```

Activity 18.10

Modify the appropriate files so that *SetDate()* and *SetTime()* can throw objects detailing the invalid values used in the attempted assignment.

In *ClassExceptions*, both catch blocks should display the values thrown.

Multiple throws

We saw in the code for *SetDate()* that a single function may contain multiple `throw` statements. But in that case, all the exceptions were of the same type (initially, `double` then *DateError*) however it is also possible for a function to throw exceptions of differing types. Hence, a function could have a structure such as

```
void f()
{
   ...
   if (condition)
      throw -1;          //Throws an int
   ...
   if (condition)
      throw -2.0;        //Throws a double
   ...
   if (condition)
      throw "Error";     //Throws char*
   ...
}
```

The Universal Handler

If an exception were to be thrown for which there was no matching catch block, the program would terminate.

When a program makes use of existing functions, many of which are capable of throwing various types of exceptions, we might be faced with the problem of writing `catch` blocks for each exception type if we don't want the program to terminate. Luckily, C++ offers a solution to this problem by allowing us to write a single catch block which is capable of catching and dealing with any type of exception.

This catch block is known as a **universal handler**.

The universal handler uses the ellipses as the exception type, meaning it starts with the line

```
catch(...)
```

If a program contains several catch blocks, the universal handler, if included, should always come last.

A program will execute the first catch block that can handle the type of value thrown. This means that the universal handler, if positioned as the first catch block, would make it impossible for any other catch block to be executed when an exception occurs.

We can think of a sequence of catch blocks as being similar in operation to a `switch` construct with the individual catch blocks being equivalent to a `case` option and the universal handler being the equivalent to the `default:` option.

Activity 18.11

Modify *ClassExceptions,* Comment out the two existing catch blocks and add a final universal handler which simply displays the message "*Error detected*". How does the program now react to an invalid date and an invalid time?

Reinstate the *DateError* catch block (but not the *TimeError* one). How does the program react to a date error and a time error?

Reinstate the *TimeError* catch block and test the program with date and time errors.

Move the universal handler so that it appears before the other `catch` blocks. How does this affect the results when using the same test data as in the previous run?

Documenting Functions that Throw Exceptions

In earlier versions of C++, when we first created a function's code, we were at liberty to specify which type of exception the function is capable of handling. This information was then added to the function prototype and heading. For example, if we were to include details of the exception thrown by *CalcSquareRoot()*, then the prototype would be written as

```
double CalcSquareRoot(double) throw(int);
```

However, this feature has now been depreciated and the only option that remains is to state that a function does not throw any exception using the keyword `noexcept`. For example, if we had a function called *Multiply()* whose code did not make use of the `throw` statement, we could code the prototype of that function as

```
double Multiply(double v1, double v2) noexcept
```

and the function itself as:

```
double Multiply(double v1, double v2) noexcept
{
   return v1*v2;
}
```

Although an explicit declaration of this type is not compulsory, it may help the compiler optimise the object code of a program.

In a more complex situation where function *A()* makes a call to function *B()*, function *A()* can only use the `noexcept` option if neither *A()* nor *B()* are coded to throw exceptions. In other words, a function can only use `noexcept` if none of the functions that it calls, nor any lower level functions, throw an exception.

Exceptions Thrown by Lower Level Functions

If a try block in `main()` makes a call to function *A()* and function *A()* calls function *B()* and function *B()* is capable of throwing an exception, then the catch block can be embedded in either function *A()* or in `main()`.

For example, if we look at the code for *Date* class's *FromJDN()* we can see that there's potential for an error situation because we have not ensured that its parameter will end up creating a date after the year 1751. In fact, the date 1/1/1752 has a Julian Date Number of 2360965, so a parameter value of less than this should be rejected by *FromJDN()*.

To enable this we'll change the code for that method to

```
// *** Sets date to eqivalent of JDN ***
Date Date::FromJDN(long jdn)
{
   int f = jdn+1401+(((4*jdn + 274277)/146097)*3)/4 - 38;
   int e = 4 * f + 3;
   int g = (e % 1461) / 4;
   int h = 5 * g + 2;
   Date d;
   d.day = (h % 153) / 5 + 1;
   d.month = (h / 153 + 2) % 12 + 1;
   d.year = e / 1461 - 4716 + (12 + 2 - d.month) / 12;
   if (jdn < 2360965)
      throw DateError(d.day, d.month, d.year);
   return d;
}
```

Now, one of the *Date* class methods that makes a call to *FromJDN()* is `operator--()` which subtracts 1 day from an object's current value:

```
//*** Subtracts one day from date, returns new date ***
Date Date::operator--()
{
   *this = FromJDN(ToJDN() - 1);
   return *this;
}
```

In the following scenario the revised *FromJDN()* method will throw an exception:

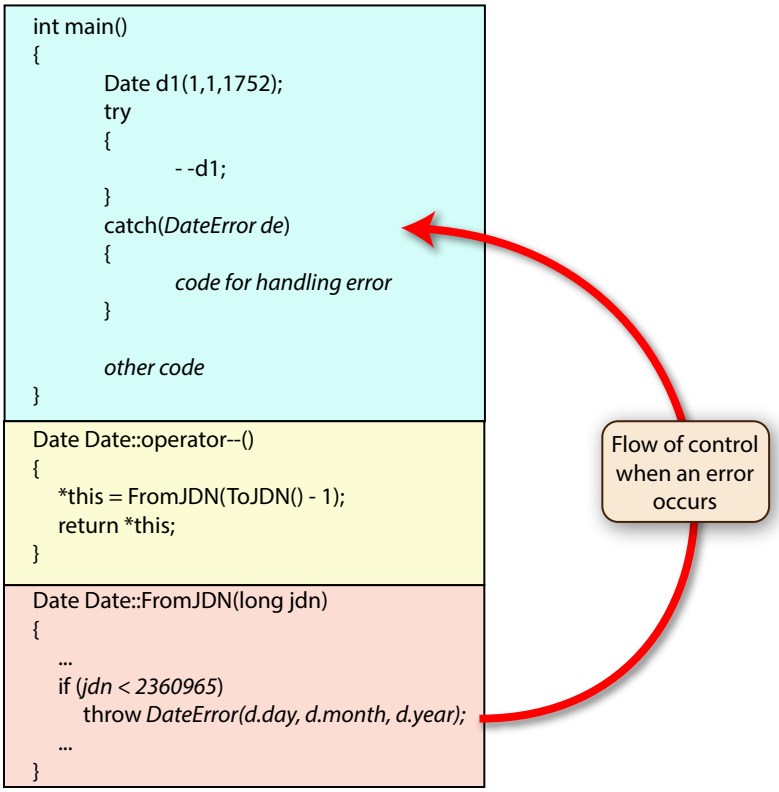
```
Date d1(1,1,1752);
--d1;
```

since we would be attempting to create a date with a *year* value of 1751.

If we catch *DateError* exceptions in *main()* then we create the setup shown in FIG-18.8.

FIG-18.8

Flow of Control From a Deep Exception



In this situation, not only are the local variables and objects created in *FromJDN()* automatically deleted when the throw occurs, but those in *operator--()* are also deleted. Since these local variables are allocated space within a stack data structure, their deallocation is known as **unwinding the stack**.

Of course, we could have placed the `try` block in *operator--()* rather than in *main():*

```
Date Date::operator--()
{
   try
   {
      *this = FromJDN(ToJDN() - 1);
      return *this;
   }
   catch(DateError de)
   {
      cout << "Invalid date: " << de.wrongday << '/'
      ↳<< de.wrongmonth << '/' << de.wrongyear  << endl;
   }
}
```

This approach means that `main()` need no longer use try nor catch blocks in relation to a check on the `--` operator:

```
int main()
{
   dsk::Date d1(1,1,1752);
   --d1;
   ...
```

Rethrowing Exceptions

The problem with catching exceptions within the methods of a class is that the programmer who is coding that class can have no idea how an application programmer using a *Date* object might wish to handle the error.

In this situation, a handler can partially deal with the exception before throwing that same exception to a higher level function. This is known as **rethrowing** the exception. To cause a rethrow, we end the current catch block with the command



No parameter is given when rethrowing an exception.

```
throw;
```

For example, our `operator--()` method could display its error message and then rethrow the exception to *main()*:

```
Date Date::operator--()
{
   try
   {
      *this = FromJDN(ToJDN() - 1);
      return *this;
   }
   catch(DateError de)
   {
      cout << "Invalid date: " << de.wrongday << '/'
      ↳<< de.wrongmonth << '/' << de.wrongyear << endl;
      throw;
   }
}
```

Of course, this means that `main()` will return to using try and catch blocks:

```
int main()
{
   dsk::Date d1(1,1,1752);
   try
   {
      --d1;
      cout << d1 << endl;
   }
   catch (dsk::DateError de)
   {
      cout << "Attempt to set to a date before 1/1/1752\n";
   }
   cout << "Program complete\n";
}
```

Activity 18.12

Make the latest changes suggested above to `operator--()` and *FromJDN()* in the *Date* class.

Modify *ClassExceptions* so that its code involving the *Time* object is deleted, *dt* is set to 1/1/1752 in its constructor (remove `scanf_s()` call) and modify the *DateError* catch block to match that given above.

Test the program to check that both catch blocks (in *main()* and `operator--()`) are executed.

terminate() and *abort()*

When an exception is thrown but not caught, C++ executes a built-in function called `terminate()`. This, in turn, calls a second function, `abort()` which ends the currently executing program.

In Visual C++, executing the code

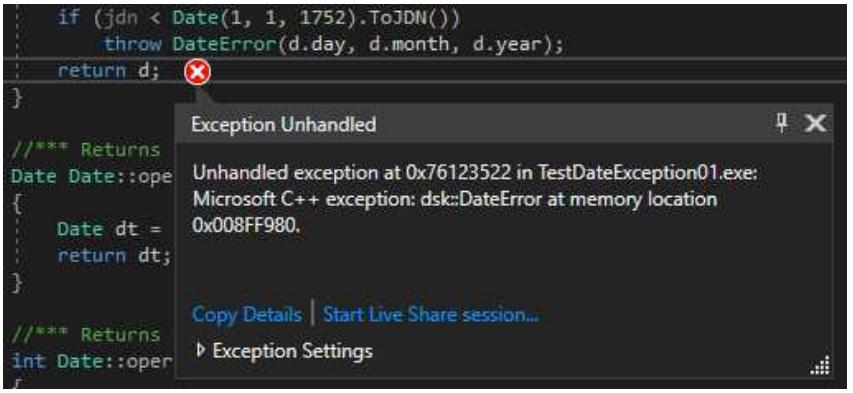
```
int main()
{
   dsk::Date test(1,1,1752);
   --test;
   cout << test << endl;
}
```

causes the *DateError* exception to be thrown from the `operator--()` method since the date created is before 1752.

Since there is no try or catch block, `terminate()` and `abort()` are both executed. In the case of Visual C++ software when running in debug mode (*F5*), this results in the display shown in FIG-18.9.

FIG-18.9

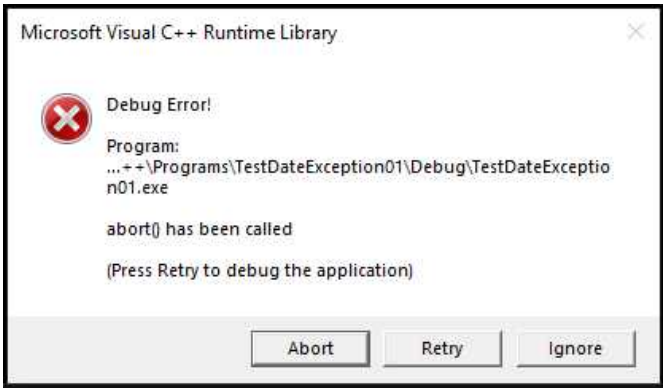
An Unhandled Exception in Debug Mode



When running without debug (*Ctrl+F5*) the program still ends, but the dialog box is somewhat different (see FIG-18.10).

FIG-18.10

An Unhandled Exception in Standard Mode



set_terminate()

We have the option to create our own function to handle a fatal error. By calling the

`set_terminate()` function we can specify the address of an alternative function to be executed when the program terminates. Our new function must return no value and take no parameters. The trivial program in FIG-18.11 shows how `set_terminate()` is used.

FIG-18.11

Replacing *terminate()*

```
#include <iostream>
#include <ChronoLib.h>
using namespace std;

//*** Function prototype ***
void OhDear();

int main()
{
   set_terminate(OhDear);
   dsk::Date test(1, 1, 1752);
   --test;
   cout << test << endl;
}


void OhDear()
{
   cout << "Something really bad has happened\n";
   exit(1);
}
```

Notice that the function replacing `terminate()`, *OhDear()*, calls `exit()` rather than `abort()`. Both `exit()` and `abort()` terminate the program, however, `exit()` will call the destructor of existing objects, `abort()` will not. Also, `exit()` returns a value from the program.

Activity 18.13

Start a new project called *ReplaceTerminate* and implement the code given above. Test the program by running in normal mode (not debug mode). Press *Ctrl+F5* or select **Debug|Start Without Debugging** from the main menu.

Summary

- Exceptions are a method of dealing with unexpected error conditions.
- Using exceptions allows the application programmer to decide how to handle error conditions rather than this being the responsibility of the class programmer.
- Exceptions make use of the keywords `throw`, `try` and `catch`.
- When an unexpected error occurs within a function, the function will throw an exception value.
- A function may contain multiple `throw` statements.
- The value thrown can be an object.
- Often a class will be produced specifically as a type designed to be thrown.
- An exception can travel back through several “layers” of function calls to

reach its catch block.

- A catch block can partially deal with an exception before rethrowing the exception to a higher level.
- Code which calls a function which is capable of throwing an exception is normally coded within a try block.
- When an exception is thrown, the remainder of the function in which it occurs is ignored, local variables are destroyed and control moves to the appropriate catch block bypassing the remainder of the try block.
- The catch block must follow immediately after the try block.
- Several catch blocks can be present, each handling different exception types.
- When an exception occurs, the first catch block which will accept the value thrown is executed.
- Only one catch block is executed when an exception occurs.
- The universal handler – `catch(...)` – will handle any exception.
- The start of a catch block contains the keyword catch and, in parentheses, the type of value the catch block is designed to handle.
- A catch block heading can, optionally, specify a variable name which will hold a copy of the value being thrown.
- When the code within the catch block has been executed, control moves to the first statement not in a catch block.
- If no exception is thrown, code within the catch block is ignored.
- If no catch block exists for an exception, the function `terminate()` is executed, which in turn executes `abort()`.
- Use `set_terminate()` to specify a different function to be executed when an exception is not caught.
- Use `abort()` or `exit()` to terminate a program.
- `abort()` does not call the destructors of any remaining objects.
- `exit()` calls the destructors of remaining objects and returns a value from the program.

Solutions

Activity 18.1

The original version of the code halts at the `assert()` call and produces a dialog box highlighting that the `assert()` failed.

Modified code for *UsingAssert*:

```
#define NDEBUG
#include <iostream>
#include <assert.h>
using namespace std;

void main()
{
  float no1, no2, result;

  //*** Read in two values ***
  cout << "Enter first number : ";
  cin >> no1;
  cout << "Enter second number : ";
  cin >> no2;
  assert(no2 != 0);
  //*** Perform division ***
  result = no1 / no2;
  //*** Display result ***
  cout << no1 << '/' << no2 << '=' << result<< endl;
}
```

NOTE: The `#define` statement must be the first non-comment line of the program.

The program will still fail, but no assert message is displayed.

Activity 18.2

On the first run, using value 9, no exception is thrown so the `catch` block of code is ignored and the program outputs the lines:

```
Square root of 9 is 3
Program complete
```

On the second run, using value -9, an exception is thrown so the `catch` block of code is executed and the program outputs the lines:

```
Tried to find the square root of a negative value
Program complete
```

Activity 18.3

Code for *Except02*:

```
#include <iostream>
using namespace std;

void main()
{
  float no1, no2;

  //*** Read in two values ***
  cout << "Enter first number : ";
  cin >> no1;
  cout << "Enter second number : ";
  cin >> no2;
  try
  {
    if (no2 == 0)
      throw - 1;
    cout << no1 << '/' << no2 << '=' << no1 / no2
    ↳<< endl;
  }
  catch (int)
  {
    cout << "Tried to divide by zero\n";
  }
  cout << "Program complete\n";
}
```

Using 12 and 3 produces the output:

```
12/3=4
Program complete
```

Using 12 and 0 produces the output:

```
Tried to divide by zero
Program complete
```

Activity 18.4

When 9 is entered, single-stepping through each line of code shows us that the exception is not thrown and the catch block is skipped.

When -9 is entered, the test `(v < 0)` in *CalcSquareRoot()* is *true*, so the exception is thrown and the code within the catch block is executed. After executing the catch block, the program continues to execute normally starting with the first line following the catch block.

Activity 18.5

Modified code for *Time* class's *SetTime()*:

```
//*** Sets time to h:m:s if this is valid ***
void Time::SetTime(int h, int m, int s)
{
  if (h < 0 || h > 23 || m < 0 || m > 59 || s < 0 ||
  ↳s > 59)
    throw - 3;
  else
  {
    hour = h;
    minute = m;
    second = s;
  }
}
```

Select **Build|Build Solution** or press *Ctrl+Shift+B* to build the new code.

Remember, the *ChronoLib* has no `main()` function and therefore cannot be executed.

Activity 18.6

Modified code for *ClassExceptions*:

```
#include <iostream>
#include <ChronoLib.h>

using namespace std;

int main()
{
  dsk::Date dt1;
  dsk::Time tm1;
  //*** Get date to be stored ***
  int d, m, y;
  cout << "Enter date to be stored : ";
  scanf("%d/%d/%d", &d, &m, &y);
  //*** Get time to be stored ***
  int h, mn, s;
  cout << "Enter time to be stored : ";
  scanf("%d:%d:%d", &h, &mn, &s);
  try
  {
    //*** Store and display date ***
    dt1.SetDate(d,m,y);
    cout << dt1 << endl;
    //*** Store and display time ***
    tm1.SetTime(h, mn, s);
    cout << tm1 << endl;
  }
  catch (int)
  {
    //*** Display error message and set date to
    ↳default ***
    cout << "Date assignment invalid\n";
    cout << "Date set to 1/1/2001\n";
    dt1.SetDate(1, 1, 2001);
```

```
  }
  //*** Display date and time ***
  cout << "Date is " << dt1 << endl;
  cout << "Time is " << tm1 << endl;
}
```

An invalid time throws a value of -3 which is caught by the `int` catch block and therefore we get a message indicating that it is the date that is invalid.

Activity 18.7

Modified code for *ClassExceptions*:

```
#include <iostream>
#include <ChronoLib.h>

using namespace std;

int main()
{
  dsk::Date dt1;
  dsk::Time tm1;
  //*** Get date to be stored ***
  int d, m, y;
  cout << "Enter date to be stored : ";
  scanf("%d/%d/%d", &d, &m, &y);
  //*** Get time to be stored ***
  int h, mn, s;
  cout << "Enter time to be stored : ";
  scanf("%d:%d:%d", &h, &mn, &s);
  try
  {
    //*** Store and display date ***
    dt1.SetDate(d,m,y);
    cout << dt1 << endl;
    //*** Store and display time ***
    tm1.SetTime(h, mn, s);
    cout << tm1 << endl;
  }
  catch (int no)
  {
    //*** Handle invalid Date ***
    if (no == -2)
    {
      //*** Display error message and set date to d
      ↳default ***
      cout << "Date assignment invalid\n";
      cout << "Date set to 1 / 1 / 2001\n";
      dt1.SetDate(1, 1, 2001);
    };
    //*** Handle invalid time ***
    if (no == -3)
    {
      //*** Display error message and set time to
      ↳default ***
      cout << "Time assignment invalid\n";
      cout << "Time set to 0:0:0\n";
      tm1.SetTime(0, 0, 0);
    };
  }
  //*** Display date and time ***
  cout << "Date is " << dt1 << endl;
  cout << "Time is " << tm1 << endl;
}
```

Now each thrown value is handled separately. Although you may have noticed that if the *Date* is invalid, no attempt is made to set the *Time* value.

Activity 18.8

Modified code for *Time::SetTime()*:

```
//*** Sets time to h:m:s if this is valid ***
  void Time::SetTime(int h, int m, int s)
  {
    if (h < 0 || h > 23 || m < 0 || m > 59 || s < 0
    ↳|| s > 59)
      throw -3.0;
    else
    {
      hour = h;
      minute = m;
      second = s;
    }
  }
```

Modified code for *ClassExceptions*:

```
#include <iostream>
#include <ChronoLib.h>

using namespace std;

int main()
{
  dsk::Date dt1;
  dsk::Time tm1;
  //*** Get date to be stored ***
  int d, m, y;
  cout << "Enter date to be stored : ";
  scanf("%d/%d/%d", &d, &m, &y);
  //*** Get time to be stored ***
  int h, mn, s;
  cout << "Enter time to be stored : ";
  scanf("%d:%d:%d", &h, &mn, &s);
  try
  {
    //*** Store and display date ***
    dt1.SetDate(d, m, y);
    cout << dt1 << endl;
    //*** Store and display time ***
    tm1.SetTime(h, mn, s);
    cout << tm1 << endl;
  }
  catch (int)
  {
    //*** Handle invalid Date ***
    //*** Display error message and set date to
    ↳default ***
    cout << "Date assignment invalid\n";
    cout << "Date set to 1/1/2001\n";
    dt1.SetDate(1, 1, 2001);
  }
  catch (float)
  {
    //*** Handle invalid Time ***
    //*** Display error message and set time to
    ↳default ***
    cout << "Time assignment invalid\n";
    cout << "Time set to 0:0 : 0\n";
    tm1.SetTime(0, 0, 0);
  };
  //*** Display date and time ***
  cout << "Date is " << dt1 << endl;
  cout << "Time is " << tm1 << endl;
}
```

The date error is caught by the `int` catch block while the time error is caught by the `float` catch block.

Activity 18.9

To define our two new error classes (*DateError* and *TimeError*) *ChronoLib.h* should begin with the lines:

```
#pragma once
#include <iostream>
namespace dsk
{
  //*** DateError class ***
  class DateError {};
  //*** TimeError class ***
  class TimeError {};
```

In the file *Date.cpp*, the method *Date::SetDate()* should now be coded as:

```
// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
  int error{ 0 };
  // *** Days in each month of year ***
  int daysinmonth[] = { 0, 31,28,31, 30,31,30,
  ↳31,31,30, 31,30,31 };
  //*** If month invalid, exit ***
  if (m < 1 || m > 12)
    error = -2;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += static_cast<int>(IsLeapYear());
  //daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0
  ↳&& y % 100 != 0);
  //*** If days in month invalid, exit ***
  if (d <1 || d > daysinmonth[m])
    error = -2;
```

```
  //*** If year less than 1, exit ***
  if (y < 1)
    error = -2;
  if (error != 0)
    throw DateError();
  //*** Assign date ***
  day = d;
  month = m;
  year = y;
}
```

In the file *Time.cpp*, the method *Time::SetTime()* should now be coded as:

```
//*** Sets time to h:m:s if this is valid ***
  void Time::SetTime(int h, int m, int s)
  {
    if (h < 0 || h > 23 || m < 0 || m > 59 || s < 0
    ↳|| s > 59)
      throw TimeError();
    else
    {
      hour = h;
      minute = m;
      second = s;
    }
  }
```

Modified code for *ClassExceptions*:

```
#include <iostream>
#include <ChronoLib.h>

using namespace std;

int main()
{
  dsk::Date dt1;
  dsk::Time tm1;
  //*** Get date to be stored ***
  int d, m, y;
  cout << "Enter date to be stored : ";
  scanf_s("%d/%d/%d", &d, &m, &y);
  //*** Get time to be stored ***
  int h, mn, s;
  cout << "Enter time to be stored : ";
  scanf_s("%d:%d:%d", &h, &mn, &s);
  try
  {
    //*** Store and display date ***
    dt1.SetDate(d, m, y);
    cout << dt1 << endl;
    //*** Store and display time ***
    tm1.SetTime(h, mn, s);
    cout << tm1 << endl;
  }
  catch (dsk::DateError)
  {
    //*** Handle invalid Date ***
    //*** Display error message and set date to
    ↳default ***
    cout << "Date assignment invalid\n";
    cout << "Date set to 1/1/2001\n";
    dt1.SetDate(1, 1, 2001);
  }
  catch (dsk::TimeError)
  {
    //*** Handle invalid Time ***
    //*** Display error message and set time to
    default ***
    cout << "Time assignment invalid\n";
    cout << "Time set to 0:0 : 0\n";
    tm1.SetTime(0, 0, 0);
  };
  //*** Display date and time ***
  cout << "Date is " << dt1 << endl;
  cout << "Time is " << tm1 << endl;
}
```

The program will react exactly as before to invalid dates and times.

Activity 18.10

Modified start of *ChronoLib.h:*

```
#pragma once
#include <iostream>
namespace dsk
{
  //*** DateError class ***
  class DateError
  {
    public:
      int wrongday;
      int wrongmonth;
      int wrongyear;

      DateError(int d, int m, int y)
      {
        wrongday = d;
        wrongmonth = m;
        wrongyear = y;
      }
  };


  //*** TimeError class ***
  class TimeError
  {
    public:
      int wronghour;
      int wrongminute;
      int wrongsecond;

      TimeError(int h, int m, int s)
      {
        wronghour = h;
        wrongminute = m;
        wrongsecond = s;
      }
  };


  //*** Date class declaration ***
  class Date
  ...
```

In the file *Date.cpp*, the method *Date::SetDate()* should now be coded as:

```
// *** Sets the date to d/m/y ***
void Date::SetDate(int d, int m, int y)
{
  int error{ 0 };
  // *** Days in each month of year ***
  int daysinmonth[] = { 0, 31,28,31, 30,31,30,
  ↳31,31,30, 31,30,31 };
  //*** If month invalid, throw exception ***
  if (m < 1 || m > 12)
    throw DateError(d,m,y);;
  //*** Add 1 to days in February if leap year ***
  daysinmonth[2] += static_cast<int>(IsLeapYear());
  //daysinmonth[2] += (y % 400 == 0) || (y % 4 == 0
  ↳&& y % 100 != 0);
  //*** If days in month invalid, throw exception***
  if (d <1 || d > daysinmonth[m])
    throw DateError(d,m,y);
  //*** If year less than 1, throw exception ***
  if (y < 1)
    throw DateError(d,m,y);
  //*** Assign date ***
  day = d;
  month = m;
  year = y;
}
```

In the file *Time.cpp*, the method *Time::SetTime()* should now be coded as:

```
//*** Sets time to h:m:s if this is valid ***
void Time::SetTime(int h, int m, int s)
{
  if (h < 0 || h > 23 || m < 0 || m > 59 || s < 0 ||
  ↳s > 59)
    throw TimeError(h,m,s);
  else
  {
    hour = h;
    minute = m;
    second = s;
  }
}
```

Modified code for the catch blocks in *ClassExceptions*:

```
catch (dsk::DateError de)
{
```

```
  //*** Handle invalid Date ***
  //*** Display error message and set date to
  ↳default ***
  cout << "Date assignment invalid\n";
  cout << "Attempted to set date to " << de.wrongday
  ↳<< '/' << de.wrongmonth << '/' << de.wrongyear
  ↳<< endl;
  cout << "Date set to 1/1/2001\n";
  dt1.SetDate(1, 1, 2001);
}

catch (dsk::TimeError te)
{
  //*** Handle invalid Time ***
  //*** Display error message and set time to
  ↳default ***
  cout << "Time assignment invalid\n";
  cout << "Attempted to set time to " <<
  ↳te.wronghour << '/' << te.wrongminute << '/' <<
  ↳te.wrongsecond << endl;
  cout << "Time set to 0:0 : 0\n";
  tm1.SetTime(0, 0, 0);
};
```

Activity 18.11

Modified catch blocks in *ClassExceptions (first setup)*:

```
/*
catch (dsk::DateError de)
{
  //*** Handle invalid Date ***
  //*** Display error message and set date to
  ↳default ***
  cout << "Date assignment invalid\n";
  cout << "Attempted to set date to " << de.wrongday
  ↳<< '/' << de.wrongmonth << '/' << de.wrongyear
  ↳<< endl;
  cout << "Date set to 1/1/2001\n";
  dt1.SetDate(1, 1, 2001);
}
catch (dsk::TimeError te)
{
  //*** Handle invalid Time ***
  //*** Display error message and set time to
  ↳default ***
  cout << "Time assignment invalid\n";
  cout << "Attempted to set time to " <<
  ↳te.wronghour << '/' << te.wrongminute << '/' <<
  ↳te.wrongsecond << endl;
  cout << "Time set to 0:0 : 0\n";
  tm1.SetTime(0, 0, 0);
}
*/
catch (...)
{
  cout << "Error detected\n";
}
```

Both *DateError* and *TimeError* exceptions are handled by the universal handler.

Modified catch blocks in *ClassExceptions (second setup)*:

```
catch (dsk::DateError de)
{
  //*** Handle invalid Date ***
  //*** Display error message and set date to
  ↳default ***
  cout << "Date assignment invalid\n";
  cout << "Attempted to set date to " << de.wrongday
  ↳<< '/' << de.wrongmonth << '/' << de.wrongyear
  ↳<< endl;
  cout << "Date set to 1/1/2001\n";
  dt1.SetDate(1, 1, 2001);
}
/*
catch (dsk::TimeError te)
{
  //*** Handle invalid Time ***
  //*** Display error message and set time to
  ↳default ***
  cout << "Time assignment invalid\n";
  cout << "Attempted to set time to " <<
  ↳te.wronghour << '/' << te.wrongminute << '/' <<
  ↳te.wrongsecond << endl;
  cout << "Time set to 0:0 : 0\n";
  tm1.SetTime(0, 0, 0);
}
*/
catch (...)
{
  cout << "Error detected\n";
}
```

DateError exceptions are handled by *DateError* catch block; *TimeError* exceptions are handled by the universal handler.

Modified catch blocks in *ClassExceptions (third setup)*:

```
catch (dsk::DateError de)
{
  //*** Handle invalid Date ***
  //*** Display error message and set date to
  ↳default ***
  cout << "Date assignment invalid\n";
  cout << "Attempted to set date to " << de.wrongday
  ↳<< '/' << de.wrongmonth << '/' << de.wrongyear
  ↳<< endl;
  cout << "Date set to 1/1/2001\n";
  dt1.SetDate(1, 1, 2001);
}

catch (dsk::TimeError te)
{
  //*** Handle invalid Time ***
  //*** Display error message and set time to
  ↳default ***
  cout << "Time assignment invalid\n";
  cout << "Attempted to set time to " <<
  ↳te.wronghour << '/' << te.wrongminute << '/' <<
  ↳te.wrongsecond << endl;
  cout << "Time set to 0:0 : 0\n";
  tm1.SetTime(0, 0, 0);
}
catch (...)
{
  cout << "Error detected\n";
}
```

DateError and *TimeError* exception are handled by their own catch blocks. The universal handler is not used.

Modified catch blocks in *ClassExceptions (fourth setup)*:

```
catch (...)
{
  cout << "Error detected\n";
}

catch (dsk::DateError de)
{
  //*** Handle invalid Date ***
  //*** Display error message and set date to
  ↳default ***
  cout << "Date assignment invalid\n";
  cout << "Attempted to set date to " << de.wrongday
  ↳<< '/' << de.wrongmonth << '/' << de.wrongyear
  ↳<< endl;
  cout << "Date set to 1/1/2001\n";
  dt1.SetDate(1, 1, 2001);
}

catch (dsk::TimeError te)
{
  //*** Handle invalid Time ***
  //*** Display error message and set time to
  ↳default ***
  cout << "Time assignment invalid\n";
  cout << "Attempted to set time to " <<
  ↳te.wronghour << '/' << te.wrongminute << '/' <<
  ↳te.wrongsecond << endl;
  cout << "Time set to 0:0 : 0\n";
  tm1.SetTime(0, 0, 0);
}
```

Although all three catch blocks are included, both *DateError* and *TimeError* exceptions are handled by the universal handler since it appears first in the list of handlers.

Activity 18.12

In the file *Date.cpp*, the method *Date::FromJDN()* and *Date::operator--()* should now be coded as:

```
// *** Sets date to eqivalent of JDN ***
Date Date::FromJDN(long jdn)
{
  int f = jdn + 1401 + (((4 * jdn + 274277) /
  ↳146097) * 3) / 4 - 38;
  int e = 4 * f + 3;
  int g = (e % 1461) / 4;
  int h = 5 * g + 2;
  Date d;
  d.day = (h % 153) / 5 + 1;
  d.month = (h / 153 + 2) % 12 + 1;
  d.year = e / 1461 - 4716 + (12 + 2 - d.month) /
  ↳12;
  if (jdn < 2360965)
    throw DateError(d.day, d.month, d.year);
  return d;
}

//*** Subtracts one day from date, returns new date
↳***
Date Date::operator--()
{
  try
  {
    *this = FromJDN(ToJDN() - 1);
    return *this;
  }
  catch (DateError de)
  {
    cout << "Invalid date : " << de.wrongday << '/'
    ↳<< de.wrongmonth << '/' << de.wrongyear <<
    ↳endl;
    throw;
  }
}
```

Modified code *ClassExceptions's Source.cpp:*

```
int main()
{
  dsk::Date d1(1,1,1752);
  try
  {
    --d1;
    cout << d1 << endl;
  }
  catch (dsk::DateError de)
  {
    cout << "Attempt to set to a date before
    ↳1/1/1752\n";
  }
  cout << "Program terminated\n";
}
```

Activity 18.13

The program should output the message *"Something really bad has happened"* before exiting..

19

Container Classes

In this Chapter:

- ❑ Creating a Container Class
- ❑ The *array* Class
- ❑ Multi-dimensional *array* Objects
- ❑ The *vector* Class
- ❑ Multi-dimensional *vector* Objects

Creating a Container Class

Introduction

A **container class** is one designed to hold a group of values in an organised structure.

We have already met the standard array which, although not a class, is an example of a container (usually of numbers or characters).

Typically, a container class will have operations to:

- Create a new container.
- Add a new value.
- Delete a value.
- Delete all the current contents.
- Allow access to individual values.
- Find the number of values in the container.

A Set Container Class

In this section we will experience the ideas behind a container by creating a container class of our own – **the set**.

The Maths

In mathematics a **set** is a collection of related **elements** held in no particular order.

Typically, a set is named using a single capital letter and the values within the set are shown enclosed in braces. For example, if A was the set of odd integer values between 1 and 10, we would write this as

$A = \{1,3,5,7,9\}$

The order of the values is irrelevant, so if $B = \{3,1,9,7,5\}$ then

$B = A$



In practice, we never show duplicate values.

Also duplicate values are ignored, so if $C = \{1,1,3,3,5,7,9\}$ then

$C = A$

The symbol $\in$ is used to mean "**is a member of**", so we can write

$5 \in A$

The symbol $\notin$ means "**is not a member of**", hence

$8 \notin A$

The **cardinality** of a set is simply the number of different members it contains (duplicates are not counted). The commonest way of showing cardinality in maths is to enclose the set (or its identifying letter) between vertical lines. For example,

$|A| = 5$

The set A is a **subset** of set B ($A \subseteq B$) if all the members of A are also members of B. So, if

$A = \{1,3,5,7,9\}$

and
B = {1,2,3,4,5,6,7,8,9}
then
$A \subseteq B$

B is a superset of A ($B \supseteq A$) if B contains all the members of A (and possibly additional members). Hence from the example above,

$B \supseteq A$

Union ($\cup$) is the set created by joining two sets:



Duplicate values are not included in the new set.

{1,2,3,4} $\cup$ {3,4,5} = {1,2,3,4,5}

Intersection ($\cap$) is the set created by the members which are common to two sets:

{1,2,3,4} $\cap$ {3,4,5} = {3,4}

Difference (-) is the set created by the members which are part of a first set, but not members of a second set:

{1,2,3,4} – {3,4,5} = {1,2}

In practice, it is often useful to check if a set is empty (represented by a set of empty braces {}). To state that a set is empty we would write something along the lines of

A = { }

Implementing a *Set* Class

The only data type available to us at this point in our knowledge that can store multiple values is the array, so we'll need to use that as our storage for the members of a set. And assuming these members are `int` values, we can start our *Set* class definition with the code:

```
class Set
{
   private:
      int store[100];
      int count;
```

where *store* will hold the member values and *count* will keep a count of how many elements of the array are occupied.

Next come the prototypes of all the methods of the *Set* class. This includes a friend function which will allow us to easily display the contents of the set:

```
   public:
      Set();
      bool IsEmptySet() const;
      bool IsMember(int) const;
      bool IsSubSet(const Set&) const;
      int Card() const;
      void AddToSet(int);
      void RemoveFromSet(int);
      Set Union(const Set&) const;
      Set Intersection(const Set&) const;
      Set Difference(const Set&) const;

      friend ostream& operator<<(ostream&,const Set&);
};
```

Finally comes the coding for each method (we'll keep everything in a single file for this project).

The *Set* constructor only needs to set *count* to zero:

```
//*** Creates an empty set ***
Set::Set()
{
   count = 0;
}
```

IsEmptySet() only has to check the value of *count* to discover if the set is empty:

```
//*** Return true if set is empty ***
bool Set::IsEmptySet() const
{
   bool result = false;
   if (count == 0)
      result = true;
   return result;
}
```

IsMember() has to search the contents of *store* for a match to the parameter value:

```
//*** Return true if v is a member ***
bool Set::IsMember(int v) const
{
   bool result = false;
   for (auto x : store)
      if (x == v)
         result = true;
   return result;
}
```

IsSubSet() has to check every member of set *s* against the contents of the current set, returning *true* only if every member of *s* is also a member of the current set:

```
//*** Returns true if s is a subset ***
bool Set::IsSubSet(const Set& s) const
{
   bool result = true;
   for (auto x : s.store)
      if (!IsMember(x))
      {
         result = false;
         break;
      }
   return result;
}
```

Card() (short for cardinality) returns the value of *count* since this tells us the number of members in the current set:

```
//*** Returns the number of members ***
int Set::Card() const
{
   return count;
}
```

AddToSet() adds a new value to the set but only if that value is not already a member of the set:

```
//*** Adds v to the set ***
void Set::AddToSet(int v)
```

```
{
   if (!IsMember(v))
   {
      store[count] = v;
      count++;
   }
}
```

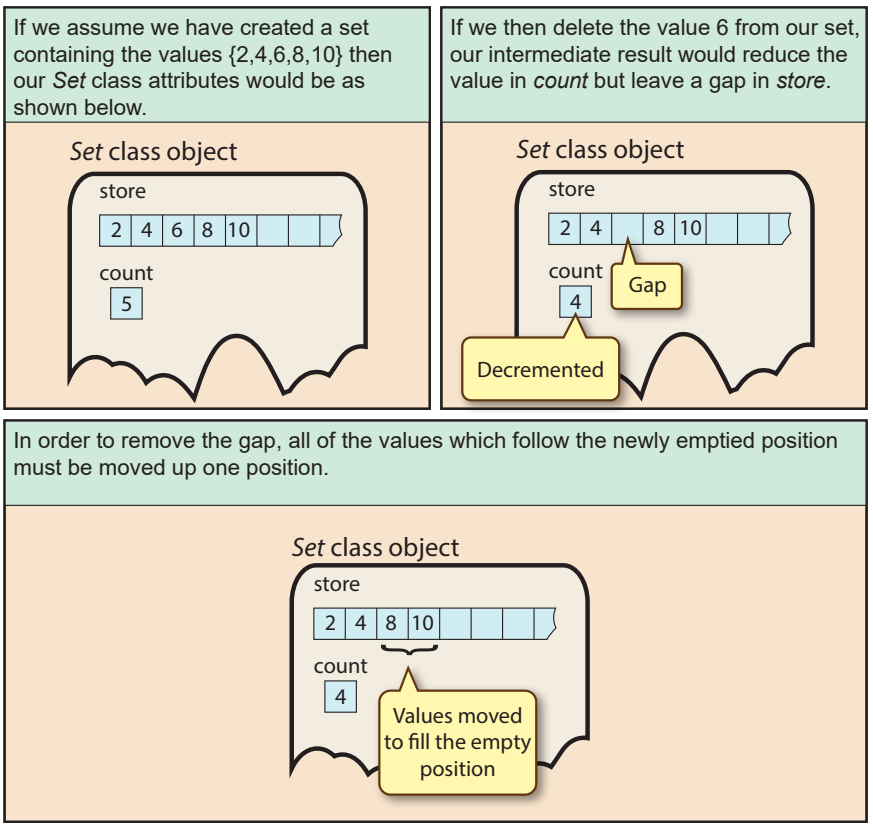
RemoveFromSet() removes a specified value from the set. Of course, if the specified value is not a member of the set, this operation has no effect:

```
//*** Removes v from the set ***
void Set::RemoveFromSet(int v)
{
   if (IsEmptySet())
      return;
   int c = 0;
   while (store[c] != v && c < count-1)
      c++;
   if (store[c] == v)
   {
      for (int k = c; k < count - 1; k++)
         store[k] = store[k + 1];
      count--;
   }
}
```

This is the most complex method in the class since the code must adjust the position of any values which follow the one being deleted (see FIG-19.1).

FIG-19.1

Deleting from an Array



Union() joins the current set with a second, specified set to create a new set, making sure that no duplicate values are created in the new set.

```
//*** Join two sets ***
Set Set::Union(const Set& s) const
{
   Set result = *this;
   for (auto v : s.store)
      if (!result.IsMember(v))
         result.AddToSet(v);
   return result;
}
```

Intersection() creates a new set from the members that are common to the current set and a second, specified set:

```
//*** Form a set from the members common to two sets ***
Set Set::Intersection(const Set& s) const
{
   Set result;
   for (auto v : s.store)
      if (IsMember(v))
         result.AddToSet(v);
   return result;
}
```

Difference() creates a new set from those members that are part of the current set but not part of a second, specified set:

```
//*** Form a set from members in the current set not in s ***
Set Set::Difference(const Set& s) const
{
   Set result;
   for (auto v : store)
      if (!s.IsMember(v))
         result.AddToSet(v);
   return result;
}
```

Finally, we come to the friend function which allows us to display the contents of a set object:

```
//*** Display the contents of a set ***
ostream& operator<<(ostream& os, const Set& s)
{
   for (int c = 0; c < s.count; c++)
      os << s.store[c] << "  ";
   return os;
}
```

All that is required now is a main program to test the various methods:

```
int main()
{
   Set A;
   Set B;
   for (int c = 2; c < 11; c+=2)
      A.AddToSet(c);
   for (int c = 2; c < 6; c ++)
      B.AddToSet(c);
   cout << A.Card() << endl;
   cout << A.IsMember(2) << endl;
   cout << A << endl;
   cout << B << endl;
   cout << A.Union(B) << endl;
   cout << A.Intersection(B) << endl;
   cout << A.Difference(B) << endl;
}
```

Activity 19.1

Start a new project called *SetContainer* and use the code given on the previous pages to implement and test a *Set* class.

The tests given are not complete. Add new statements to your program to test deleting 3 and 5 from B, if B is a subset of A, and to check if set A is empty.

Template Classes

The trouble with our *Set* class is that it is only capable of holding `int` values but it will only be of real use if we can have it store values of any type.

Back in Chapter 12 we saw how templating could be used to produce a function capable of accepting a wide range of data types. This same approach can also be used to expand the capability of any container class.

To create a class template we need to start the class definition with the line

```
template <class T>
```

If you remember, back in Chapter 12 we used the line

```
template <typename T>
```

In fact, the terms `typename` and `class` are interchangeable here. But since it is likely that our container will often hold class objects, the term `class` seems more appropriate in this case.

Next comes the usual class definition but with *T* replacing any type that is dependent on the type of values held within the container:

```
class Set
{
   private:
      T store[100];
      int count;
   public:
      Set();
      bool IsEmptySet() const;
      bool IsMember(T) const;
      bool IsSubSet(const Set<T>&) const;
      int Card() const;
      void AddToSet(T);
      void RemoveFromSet(T);
      Set<T> Union(const Set<T>&) const;
      Set<T> Intersection(const Set<T>&) const;
      Set<T> Difference(const Set<T>&) const;

      template <class T>
      friend ostream& operator<<(ostream&, const Set<T>&);
};
```

Notice that where the method's parameter or return type is a *Set*, we need to use the term `Set<T>` to state that the set will be containing values of type *T*.

Notice also that the friend function must be given its own `template <class T>` line preceding the function prototype.

Next comes the code for each of the methods of the class as well as the code for the friend function. Again, each of these must be preceded by the line

```
template<class T>
```

For example, the constructor is coded as:

```
//*** Creates an empty set ***
template <class T>
Set<T>::Set()
{
   count = 0;
}
```

Notice that the class name is now given as `Set<T>` rather than just `Set` as it was previously.

The method `IsMember()` is coded as

```
//*** Return true if v is a member ***
template <class T>
bool Set<T>::IsMember(T v) const
{
   bool result = false;
   for (auto x : store)
      if (x == v)
         result = true;
   return result;
}
```

The friend function is coded as:

```
//*** Display the contents of a set ***
template <class T>
ostream& operator<<(ostream& os, const Set<T>& s)
{
   for (int c = 0; c < s.count; c++)
      os << s.store[c] << "  ";
   return os;
}
```

When all the methods have been converted, our final modifications are in `main()` which only has to define the type of values to be stored in our *Set* objects. For example, when storing integers we would declare our objects with the lines

```
Set<int> A;
Set<int> B;
```

Activity 19.2

Modify the code in *SetContainer* so that the *Set* class becomes a class template.

Check that the program still operates correctly by defining A and B as sets containing `int` values.

Date Sets

Rather than storing `int` values in our *Set* containers, let's try something a bit more interesting by creating *A* and *B* as *Set* objects which contain collections of *Date* objects.

We created the *Date* class back in Chapter 16 and then created a static library

(*ChronoLib*) in Chapter 17 which contained both *Date* and *Time* classes.

So before we can create a set of dates, we need to tell the compiler how to access our static library.

First we need to add the line

```
#include <ChronoLib.h>
```

to the start of our *SetContainer* source code.

Next we need to supply the compiler and linker with the necessary information to access the required files.

In *Solution Explorer*, we need to right-click on the *SetClass* project name and then select *Properties* from the popup menu. We then need to set the following *Configuration Properties* options:



If you have stored your library elsewhere, then these values will be different.

C/C++ > General > Additional Include Directories :
E:\Hands On C++\Programs\MyLibraries\ChronoLib

Linker > General > Additional Library Directories :
E:\Hands On C++\Programs\MyLibraries\Debug

Linker > Input > Additional Dependencies :
ChronoLib.lib



Date and *Time* classes are defined within the *dsk* namespace.

With those changes in place, we can now create as set of *Date* objects:

```
Set<dsk::Date> A;
Set<dsk::Date> B;
```

Obviously, how we store a value in these sets will have to change, removing the previous `for` loops and replacing them with statements such as:

```
A.AddToSet(dsk::Date(1, 12, 2001));
A.AddToSet(dsk::Date(23, 11, 1963));
A.AddToSet(dsk::Date(20, 7, 1969));
B.AddToSet(dsk::Date(23, 11, 1963));
B.AddToSet(dsk::Date(6, 11, 1999));
```

Activity 19.3

Modify *SetContainer* so that it can access the *ChronoLib* library.

Modify sets *A* and *B* to contain *Date* objects.

Does the program compile?

What Went Wrong?

Activity 19.3 produces a compilation error. Clicking on the message takes us to the *Set* class's *IsMember()* function:

```
//*** Return true if v is a member ***
template <class T>
bool Set<T>::IsMember(T v) const
{
   bool result = false;
```

```
   for (int c = 0; c < count; c++)
      if (store[c] == v)
         result = true;
   return result;
}
```

More specifically, the problem is with the expression

```
store[c] == v
```

Since we are attempting to store *Date* objects in our *Set* objects, the `==` operator in the above expression is comparing two *Date* objects held in *store[c]* and *v*. The *Date* class has defined a method, `operator==()`, so the comparison of two *Date* objects shouldn't be a problem. However, if we look at the code for that operation:

```
//*** Returns true if two dates are equal ***
bool Date::operator==(Date dt)
{
   return (ToJDN() == dt.ToJDN());
}
```

we can see that the method's heading does not end with the `const` term. In other words, the method is not promising to leave the calling *Date* object unchanged after its code has been executed.

On the other hand the *Set* class's *IsMember()* does end with `const` and therefore cannot change the contents of the object calling this method.

And, because *IsMember()* cannot change the contents of the object to which it belongs, it's unwilling to hand part of that object (*store[c]*) to an operation (`==`) which has no such restriction.

To solve this problem, we need to return to the *Date* class and add `const` to the end of all those methods that do not modify the *Date* object. And, while we're there, we might as well add `const` and reference(`&`) to any parameters that remain unchanged. This gives us the following updated header for the *Date* class:

```
//*** Date class declaration ***
class Date
{
   private:
      int day;
      int month;
      int year;

      long ToJDN() const;
      Date FromJDN(long) const;

   public:
      static const int DAYSINYEAR = 365;
      static int count;
      explicit Date(int = 1, int = 1, int = 2001);
      void SetDate(int, int, int);
      int GetDayOfWeek() const;
      bool IsLeapYear() const;
      int GetDay() const;
      int GetMonth() const;
      int GetYear() const;
      Date operator+(int) const;
      int operator-(const Date&) const;
      Date operator-(int) const;
      bool operator==(const Date&) const;
      bool operator!=(const Date&) const;
```

```
        void operator+=(int);
        Date operator++();
        Date operator++(int);
        Date operator--();
        Date operator--(int);
        int GetDaysInYear() const;
        static int GetDAYSINYEARconst();
        static int GetCount();
        ~Date();

        friend std::ostream& operator<<(std::ostream&, const
        ↳Date&);
        friend std::istream& operator>>(std::istream&, Date&);
        friend Date operator+(int, Date&);
};
```

We then need to change the headings in the code for each of these methods. For example, the `operator-()` methods would now have the coding:

```
//*** Returns the difference in days between two dates ***
int Date::operator-(const Date& dt) const
{
   int result = ToJDN() - dt.ToJDN();
   return result;
}

//*** Returns a Date calculated by subtracting d days from date
↳***
Date Date::operator-(int d) const
{
   Date dt = FromJDN(ToJDN() - d);
   return dt;
}
```

Activity 19.4

Load your *MyLibraries* project and modify the *ChronoLib.h* and *Date.cpp* files so that the changes described above are implemented.

Recompile *MyLibraries*.

Reload *SetContainer* and check that it now compiles and runs correctly.

Storing Other Types of Objects in a Set

So if we can store *Date* objects in our *Set* container, surely we can store any type of object we wish? Unfortunately, it's not that simple. If we look at the code for the methods in the *Set* class we'll find expressions such as

```
store[c] == v
store[c] != v
```

These expressions are testing the equality of objects of type *T*, so if *T* is instantiated as type *Date*, then the *Date* class must define the `==` and `!=` operations. Luckily, the *Date* class does define those operators. But if we were to try using another class that did not define these operations, our attempt to store such objects within sets would fail.

In addition, the *Set* class's friend function uses the << operator to output the elements of the set, so this operator must also be defined for any class that is placed within a *Set* object. Again, this is true for the *Date* class.

Also, as we saw with the *Date* class, having the necessary operations defined in a class still doesn't guarantee success if we do not have the appropriate `const` declarations.

Activity 19.5

Could objects of the *Time* class be stored in a *Set* object?
HINT: Look at the code in *ChronoLib.h*.

Summary

- Container classes, like arrays, can contain multiple values.
- By defining a container class template, a container class object can hold a type defined by the application programmer.
- In order to hold objects of a given class, that class must define any operations or methods used within the container's methods.

STL Container Classes

Introduction

Luckily, we don't have to create our own container class very often since C++ comes equipped with several container classes (including a `set` class) in its Standard Template Library (STL).

These container classes share many operations. Typically, a container class will contain operations to:

- Create a new container.
- Add a new object.
- Delete an object.
- Delete all the current contents.
- Check if the container is empty.
- Allow access to individual objects within the container.

In the remainder of this chapter we'll look at just two STL container classes: `array` and `vector`.

As might be guessed from its name, the `array` class is an object-oriented replacement for the array structure we've been using up to this point in the text.

In fact, the `vector` class is also used as a substitute for traditional arrays but has added benefits over the `array` class in some situations as its size can be varied during runtime.

Iterators

An **iterator** can be thought of as a form of pointer that references an element of a container. Typically, an iterator object is used to move through the values within a container. Also many container classes will offer methods which return an iterator value to a specific position within a container. For example, the method `begin()` will return an iterator value referencing the first value in an `array` or `vector` object.

Like variables and objects, an iterator must be declared and named.

Iterators can be one of five types:

input	an input iterator can only move forward through the values in a container, giving access to the current value referenced.
output	like the input iterator, the output iterator can only move forward through a container, and is used to assign a new value to the accessed element in the container.
forward	a forward iterator can only move forward through the elements of a container, but can be used to both access or change the current value.

bidirectional	a bidirectional iterator can move forward or backward (one position at a time) through the values in a container, allowing those values to be accessed or changed.
random-access	a random-access iterator can reference any position in a container and can be used to access or change the value at the currently accessed location.

The type of iterator used by a container is determined by the container, not the programmer. For example, both `array` and `vector` containers use random-access iterators.

We'll get practical experience of iterators in the next few pages.

The array Container Class

Introduction

Although C++, like almost every other programming language, allows us to create standard arrays for storing a collection of similar values, it also offers an **array container class** which has several advantages over regular arrays.

Declaration

To use `array` class objects, a program must include the line

```
#include <array>
```

The `array` container class, like all other built-in container classes, is coded using templates so that we are then free to declare an array object capable of storing any type we require.

The basic format of the declaration is shown in FIG-19.2.

FIG-19.2

Declaring an ***array*** Object



where:

T	is the type of value the array is to contain.
size	is the number of elements in the array.
identifier	is the name to be assigned to the object being created.

If the program in which the `array` object is declared contains the statement

```
using namespace std;
```

then the initial tokens `std` and `::` may be omitted.

So, if we wish to declare an `array` object called *numbers* capable of containing 20 `int` values then we would write

```
std::array<int, 20> numbers;
```

Like a standard array, the elements of our array object will not be initialised to any specific value and therefore will contain apparently random values on startup.

Initialising an *array* Object

`array` objects are designed to closely mimic the behaviour of standard arrays, so they can be initialised in the same way. Examples of initialisations are given below:

```
std::array< int, 20 > numbers {0}; //All elements set to zero

std::array< int, 20 > numbers {};  //All elements set to zero

std::array< int, 20 > numbers {2,4,6} //First three cells set
         //to 2, 4 and 6 respectively, others cells set to zero
```

Accessing *array* Object Elements



The first element of an array object, like a standard array, has an index of zero.

Array object elements can be accessed in exactly the same way as regular arrays, using an index enclosed in brackets. For example, the following line assigns the value 12 to the fourth element of *numbers*:

```
numbers[3] = 12;
```

However, also like normal arrays, C++ won't check that the index value is valid meaning that it will not object to a line such as

```
numbers[33] = 2;
```

The range-based `for` loop can also be used with an array object. For example, we could display every element in *numbers* using the lines

```
for (auto v: numbers)
   cout << v << endl;
```

Or we could increment every value using a reference variable:

```
for (auto &v: numbers)
   v++;
```

array Class Methods

So far the `array` class seems to offer no advantages over the traditional array. But those advantages come with the methods defined within the class – as described below:

fill()

The `fill()` method allows us to fill every cell within an array object with the same, specified value. For example,

```
numbers.fill(2);
```

would place the value 2 in every cell of *numbers*.

size() and *max_size()*

Both of these methods return the number of cells in the array object. For example, we might use `size()` within a `for` loop to ensure we access every cell:

```
for(int c = 0; c < numbers.size(); c++)
   cout << numbers[c] << endl;
```

empty()

The `empty()` method returns *true* if the array contains zero elements. To achieve this result, a program would have to create the array with a line such as

```
array<int, 0> numbers;
```

at()

The `at()` method allows us to access a specified cell within the array. In effect, this is an alternative to using brackets. So, rather than using the expression

```
numbers[7]
```

we could write

```
numbers.at(7)
```

The advantage of using `at()` is that it will throw an exception if the index value given is invalid. The exception type is `std::out_of_range`.

front() and *back()*

The contents of the first and last elements of an array object can also be accessed using the methods `front()` and `back()`. Both methods return a reference to the appropriate value. Hence, the expression

```
numbers.front()
```

is equivalent to

```
numbers[0]
```

while

```
numbers.back()
```

is equivalent to

```
numbers[numbers.size()-1]
```

We can use these two methods to either access an element's contents,

```
cout << numbers.back() << endl;
```

or to assign a value to it,

```
numbers.back() = 123;
```

Assignment Operator (=)

The `array` class defines the assignment operator to allow us to copy the contents of one array object to another with a single statement. For example, if our program contains the declarations

```
std::array<int,4> seq1 {1,2,3,4};
std::array<int,4> seq2 {};
```

we could copy the contents of *seq1* to *seq2* using the single line



For the assignment operator to work, the array objects must be of the same type and size.

```
seq2 = seq1;
```

If we had used a standard array for *seq1* and *seq2*, we would have to copy the values one cell at a time with code such as

```
for(int c = 0; c < size(seq1); c++)
   seq2[c] = seq1[c];
```

swap()

The contents of two identical (in size and type) array objects can be swapped using the `swap()` method. For example,

```
seq1.swap(seq2);
```

would swap the contents of *seq1* and *seq2*.

Of course, using the line

```
seq2.swap(seq1);
```

would have exactly the same effect.

Relational Operators (==, !=, <, <=, >, >=)

All of the relational operators are defined to work on complete array objects. For example, while testing to find out if two traditional arrays contain exactly the same value would require us to test each corresponding element in turn, when dealing with two array objects (*seq1* and *seq2*), we need merely write

```
if ( seq1 == seq2 )
```

This will return *true* only if the contents of the two array objects are an exact match.

On the other hand, the expression

```
seq1 != seq2
```

would return *true* if even a single difference exists between the arrays.

When testing for "greater" or "less than" with an expression such as

```
seq1 < seq2
```

the contents of the two arrays are compared cell-by-cell until a difference is found. At that point, if *seq1* contains the lesser of the two values, then the expression would return *true*. Similar logic is used for the other comparison operators.

The program in FIG-19.3 demonstrates the use of the `array` class by creating a 5-element `int` array containing 5 random values. The program also demonstrates the use of various methods of the `array` class.

FIG-19.3

Using an *array* Object

```
#include <iostream>
#include <array>
using namespace std;

int main()
{
   array<int, 5> values;

   //*** Assign random values ***
   for (int c = 0; c < values.size(); c++)
      values[c] = rand();

   //*** Display contents ***
   cout << "Contents of values:\n";
   for (int c = 0; c < values.size(); c++)
      cout << values.at(c) << endl;

   //*** Display first and last values ***
   cout << "\nFirst value : " << values.front() << endl;
   cout << "Last value : " << values.back() << endl;

   //*** Create a new array, elements set to -1 ***
   array<int, 5> copy;
   copy.fill(-1);

   //*** Display the contents of copy ***
   cout << "\n\nContents of copy:\n";
   for (int c = 0; c < copy.size(); c++)
      cout << copy[c] << endl;

   //*** Duplicate values in copy ***
   copy = values;
```



FIG-19.3
(continued)

Using an *array* Object

```
   //*** Display latest values in copy ***
   cout << "\nContents of copy:\n";
   for (int c = 0; c < copy.size(); c++)
      cout << copy[c] << endl;

   //*** Check if both array objects are identical ***
   if (values == copy)
      cout << "\nBoth arrays contain the same values\n";
   else
      cout << "\nContents of both arrays are different\n";
}
```

Activity 19.6

Start a new project called *UsingArrayObjects01* and implement the code given above.

At the end of the existing code, add instructions to change the contents of the final element in *values* to -8 and then display the full contents of *values*.

Use the `swap()` method to exchange the contents of the two arrays then display the contents of the two arrays, using the `at()` method to access the value in each element.

data()

If we want to call a function which has been designed to take a standard array as a parameter – which as we know from Chapter 10, is nothing more than a pointer to the start of the array's data – then we can pass the values held within an `array` object to such a function by using the `data()` method which returns the address of the first value held within the array object. For example, let's assume we have a function called `average()` which takes a standard array and returns the average of the values held:

```
double average(int list[], int n)
{
   int total{ 0 };   //Total of all values

   //*** Sum values ***
   for (int c = 0; c < n; c++)
      total += list[c];
   //*** Calculate and return average ***
   return total / static_cast<float>(n);
}
```

If the main program contains both a traditional array (*list*) and an array object (*seq*):

```
int list[] { 1,2,3,4,5 };
array<int, 5> seq{ 2,4,6,8,10 };
```

then we could display the average value of each with the lines

```
cout << average(list, 5) << endl;
cout << average(seq.data(), 5) << endl;
```

Of course, if we design the *average()* function to accept `array` objects, then we no longer need the second parameter specifying the number of elements in the array since we have the `size()` method available to supply this information.

```
template<class T>
double average(T list)
{
   int total{ 0 };   //Total of all values
   //*** Sum values ***
   for (auto v: list)
      total += v;
   //*** Calculate and return average ***
   return total / static_cast<float>(list.size());
}
```

Now we can use the call

```
cout << average(seq) << endl;
```

to display the average of the values in the array object.

Activity 19.7

Start a new project called *UsingArrayObjects02* and create a program containing the two versions of *average()* as shown above.

In the `main()` section, create an `array` object, *values*, assigning it the values 2,4,6,8,10.

Using the `array` object (or its data), call each function in turn and display the results produced.

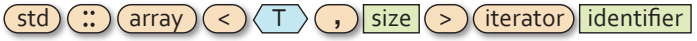
array Class Iterators

So far we've used standard `int` variables to specify the subscript of any element we've wanted to access in our array objects. But we also have the option to declare and use an iterator.

`array` class iterators are declared using the syntax shown in FIG-19.4.

FIG-19.4

Declaring an *array* Iterator



where:

T	is the type of value held in the array whose elements are referenced by the iterator.
size	is the number of elements in the array.
identifier	is the name to be assigned to the iterator variable.

As usual, if we are using the

```
using namespace std;
```

line in our program, we can drop the initial `std::` from the declaration.

If we wanted to define an iterator for an array object in our *UsingArrayObjects* project, we would use a line such as

```
array<int,5>::iterator iter;
```

where *iter* is the name of our iterator variable.

Since *iter* is an iterator for an `array` object, it will automatically become a random-access iterator, meaning it can be assigned a reference to any location within one of our arrays and the contents of that location can be read or written to.

Since an iterator is really a form of pointer, we need to assign *iter* the address of an element within an `array` object. However, we cannot use the address of operator (`&`). Instead, container classes all contain methods which are designed to return a value which can be assigned to an iterator.

begin()

The `begin()` method is defined within the `array` class.

The `begin()` method returns an iterator value giving the address of the first value within the `array` object. So, for example, the line

```
iter = values.begin();
```

would assign to *iter* the address of the first element in *values*.

If we want to access the value referenced by an iterator, we need to dereference the iterator, just as we would a standard variable. Hence, having made *iter* reference the first element in *values*, we can display the contents of that element using the line

```
cout << *iter << endl;
```

Activity 19.8

Modify *UsingArrayObjects01* so that it contains only the initial code for setting up the *values* object, assigning it random values and displaying its contents.

Add code to declare the iterator *iter*, have it reference the first cell in *values* and use the iterator to display that cell's contents.

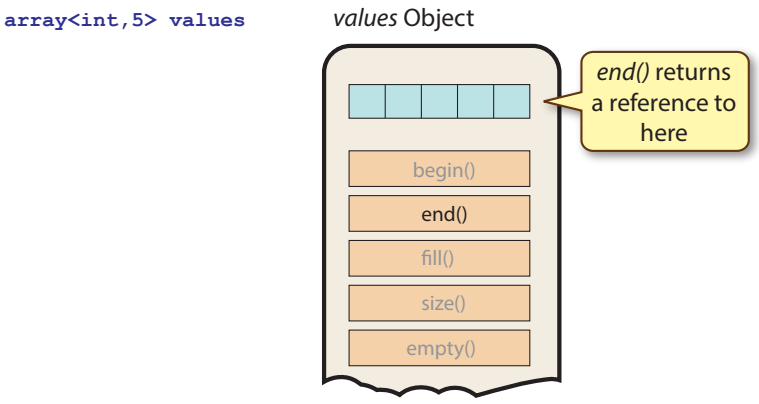
end()

Since `begin()` references the first element in an `array` object, we might assume that the `end()` method would reference the final element. Unfortunately, we would be wrong!

The `end()` method references an address immediately AFTER the final element of the array (see FIG-19.5).

FIG-19.5

end() Return Value



To access the address of the final element in the array *values*, we could use the line

```
iter = values.end()-1;
```

Activity 19.9

Add to *UsingArrayObjects01* so that *iter* references the last cell in *values,* and display the contents of that cell.

Using an Iterator to Access Every Element of an Array

By far the commonest use of an iterator is to move through each element in a container in turn. Often this will be to display the contents of the whole array or to search the array for a specific value.

To display the contents of values using an iterator, we would create the following code:

```
for (auto iter = values.begin(); iter != values.end(); iter++)
      cout << *iter << endl;
```

Notice that the code uses `auto` to declare *iter* (the earlier declaration of *iter* would have to be removed).

Activity 19.10

Modify *UsingArrayObjects01* so that *iter* is used in the `for` loop displaying the contents of *values*.

When we use `begin()` to assign a value to an iterator, that iterator can then be used to modify the value that is being referenced. For example, the code

```
for (auto iter = values.begin(); iter != values.end(); iter++)
      (*iter)++;
```

will add 1 to every value in the array.

cbegin() and *cend()*

If we want to ensure that an iterator cannot be used to change the value it references, we can use the `cbegin()` and `cend()` functions when assigning the iterator a value. These methods return `const iterator` values which means that when the returned values are assigned to an iterator, that iterator cannot be used to modify the value stored at the referenced address. Hence, the code

```
for (auto iter = values.cbegin();iter != values.cend();iter++)
      (*iter)++;
```

would fail to compile.

rbegin()

Should we need to travel through an `array` object in reverse order (starting at the last element and ending up at the first), then we can intialise our iterator using the `rbegin()` method (short for reverse-begin). Hence, the line

```
auto iter = values.rbegin();
```

will store the address of the final (5th) element of *values* in *iter*.

If we were dealing with a standard pointer, we would quite reasonably expect to decrement *iter* to set it to the address of the fourth element of *values*. However, since we now have a reverse iterator, we actually use the increment operator to have the iterator move to the previous location!

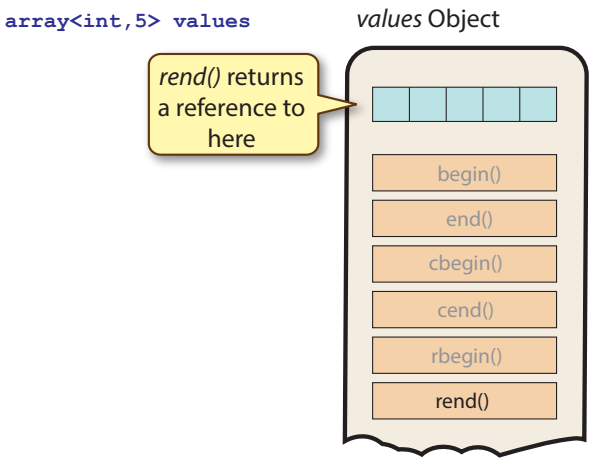
```
iter++;  //Move to previous element
```

rend()

The `rend()` method returns a reference to an address before that of the first element in an `array` object (see FIG-19.6).

FIG-19.6

rend() Return Value



To display the contents of values in reverse order, we would use the lines

```
for (auto iter = values.rbegin();iter != values.rend();iter++)
      cout << *iter << endl;
```

crbegin() and *crend()*

If when working through our array in reverse order, we want to ensure that it is impossible to change the contents of that array, we can use the methods `crbegin()` and `crend()` to assign a `const iterator` value to our iterator.

Iterator Operators

In our examples we've seen that an iterator variable can be incremented (`++`) – though we must remember in the case of a reverse iterator, this operator takes us to the previous element rather than the next one.

To access the referenced value, we have to dereference the iterator (*).

Other operators normally used with standard pointers are also available for a random-access iterator. These are:

`--`	Decrement the iterator (previous address for forward-operating iterators and next address of reverse-operating iterators).
`+=`	Add to address.

`-=`	Subtract from address.
+	Add two iterators.
-	Subtract two iterators.

All the relational operators are also available. So, for example, we can compare two iterators (assuming they reference exactly the same type of array object) using `==` , `!=` , `<=` , `<` , `>` or `>=` .

Multi-Dimensional *array* Objects

We've seen the advantages that an `array` object offers over a traditional array, but how might we create a two, three, or four dimensional array using this class?

In fact, it is possible to create two (and higher) dimensional arrays using the `array` class. The trick to achieving this is to define the type of the array as another array! For example, to create a five-row, three-column `int` array we would use the line.

```
array<array<int, 3>, 5> tt;
```



At the time of writing VS2019 will not accept nested braces for the initialisation.

To initialise the array we could expand the line to

```
array<array<int,3>,5>tt{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
```

To access the elements of the array we use the standard two sets of brackets as in the line

```
cout << tt[0][2] << endl;
```

If we'd rather use the `at()` method to access an element, then the above line would change to

```
cout << tt.at(0).at(2) << endl;
```

Activity 19.11

Start a new project called *MultiArray* and create a 2D 3 by 2 array containing the values 2,4,6, 8,10,12. Multiply each value in the array by 3 and display its new contents.

To create a 3D array, we only have to add another layer of nesting to the declaration. For example,

```
array<array<array<int,2>, 3>, 5> cb;
```

Summary

- An `array` object is an object capable of storing a fixed number of values and has many of the features of a standard array.
- Add `#include <array>` to use the `array` class in a program.
- Use `std::array<type,size> name;` to set up an `array` object.
- An array object is created with undefined values within its elements.
- An `array` object declaration can also include the set of values to be stored in the array.
- A value within an array object can be accessed using the object name, and an

index enclosed in square brackets (e.g. `values[2]`).

- To create multi-dimensional arrays, nest the type and size details.
- Methods:

fill()	Fills every element within the array with a specified value.
size()	Returns the number of elements in the `array` object.
max_size()	Like `size()`, returns the number of elements in the `array` object.
empty()	Returns *true* if the array contains no elements.
at()	An alternative to [ ] but also checks that the given subscript is within range.
front()	Gives access to the first value in the array.
back()	Gives access to the last value in the array.
swap()	Swaps the contents of two `array` objects. The arrays must store identical value types and be of the same size.
data()	Returns the contents of the `array` object in the form of a traditional array.

- Iterator Methods

begin()	Returns a reference to the first value.
end()	Returns a reference to an address following the last value.
cbegin()	Returns a `const` reference to the first value.
cend()	Returns a `const` reference to an address following the last value.
rbegin()	Returns a reference to the last value and sets up the iterator to work in reverse.
rend()	Returns a reference to an address before the first value.
crbegin()	Returns a `const` reference to the last value and sets up the iterator to work in reverse.
crend()	Returns a `const` reference to an address before the first value.

- Iterator Operators

++	Increment iterator.
--	Decrement iterator.
*	Dereference iterator.

- Relational Operators (the two array objects being compared must contain the same number of elements)

==	Returns *true* if the arrays contain identical sequences of values.
!=	Returns *true* if the arrays do not contain identical sequences of values.
<	Returns *true* if left-hand array contains a lesser sequence than the right-hand array.
<=	Returns *true* if left-hand array contains a lesser sequence than the right-hand array or the two sequences are identical.
>	Returns *true* if left-hand array contains a greater sequence than the right-hand array.
>=	Returns *true* if left-hand array contains a greater sequence than the right-hand array or the two sequences are identical.

The vector Container Class

Introduction

A major shortcoming of the `array` class is that the container must be of a fixed size. There is no opportunity to expand or contract the number of cells depending on the amount of data we wish to store at a given time. The `vector` class overcomes this restriction allowing us to optimise the use of memory space.

However, it should be noted that a vector uses contiguous memory space (just like a standard array) and when the allocated space needs to be expanded there is a possibility that the vector's data will need to be moved to new locations. This can lead to slower execution times and make the address contained within any existing iterator variable invalid.

Declaration

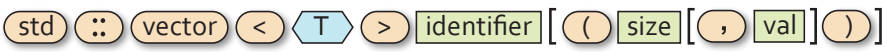
To use v`ector` class objects, a program must include the line

```
#include <vector>
```

The basic format of the declaration is shown in FIG-19.7.

FIG-19.7

Declaring a *vector* Object



where:

T	is the type of value the vector is to contain.
identifier	is the name to be assigned to the object being created.
size	is an integer value giving the number of elements in the vector.
val	is a type compatible with T specifying the initial value to be stored in every element.

A typical declaration would be

```
std::vector< int > vec;
```

This creates an empty vector with no elements in which `int` values can be stored.

If the program in which the `vector` object is declared contains the statement

```
using namespace std;
```

then the initial tokens `std` and `::` may be omitted.

However, it is also possible to specify an initial number of elements within the vector with a statement such as

```
vector<int> vec(12); // 12 cells
```

Unlike arrays (and `array` objects), the elements within a numeric vector are automatically set to zero.

If we want our vector to start with all its cells set to some other value, that can be

specified too. In the line below all the cells contain the value -1:

```
vector<int> vec(12, -1); // 12 cells all set to -1
```

To set up differing values within the cells, we can use braces as in the line

```
vector<int> vec {2,4,6,8,10,12};
```

In this case the vector will contain exactly the number of cells required to store the specified values.

Accessing *vector* Object Elements

`vector` object elements can be accessed in exactly the same way as regular arrays and `array` objects, using an index enclosed in brackets (index values start at zero). For example, the following line assigns the value 12 to the fourth element of *vec*:

```
vec[3] = 12;
```

However, as we saw with arrays and `array` objects, C++ won't check that the index value is valid, meaning that it will not complain to a line such as

```
vec[13] = 20;
```

which will then cause a runtime error. In Visual Studio, this causes an `assert()` fail.

The range-based `for` loop can also be used with a vector object. For example, we could display every element in *vec* using the lines

```
for (auto v: vec)
   cout << v << endl;
```

Activity 19.12

Create a new project called *UsingVectors01*.

In `main()` create a `vector` object, *vec*, containing 5 elements all set to -3.

Change the contents of the fourth element to 21 then display the contents of the vector.

vector Class Methods

As we will see, the `vector` shares many of the methods we saw previously in the `array` class.

at()

The `at()` method allows us to access a specified cell within the vector. In effect, this is an alternative to using brackets. So rather than write the expression

```
vec[3]
```

we could write

```
vec.at(3)
```

The advantage of using `at()` is that it will throw an exception if the index value given is invalid. The exception type is `std::out_of_range`.

size()

The `size()` method returns the number of values currently stored in the vector object. This will normally be equal to the number of elements in the vector.

data()

The `data()` method returns the contents of the vector object as a standard array. This can be useful when making a call to a function designed to take a standard array as a parameter. For example, if a function, *smallest()*, takes an array and its size as parameters, then we might use the contents of a vector object (*vec*) as the argument for this function using the line

```
cout << smallest(vec.data(), vec.size()) << endl;
```

empty()

The `empty()` method returns *true* if the vector contains no elements.

front() and *back()*

The contents of the first and last values in a `vector` object can also be accessed using the methods `front()` and `back()`:

```
vec.front()
vec.back()
```

These methods return a reference to the value and therefore can be used to find or change the value at that location:

```
cout << vec.front() << endl;
vec.front() = 12;
```

swap()

The contents of two `vector` objects containing the same type (though they may be of differing sizes) can be swapped using the `swap()` method. For example,

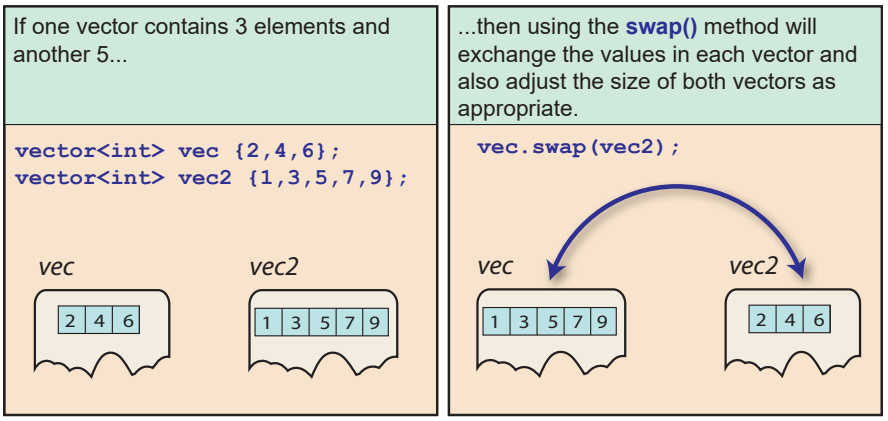
```
vec1.swap(vec2);
```

would swap the contents of *vec1* and *vec2*.

The `swap()` operation adjusts the number of cells within the two vectors where necessary. A typical example is shown in FIG-19.8.

FIG-19.8

Swapping Vectors



Activity 19.13

Modify *UsingVectors01* so that it contains a second vector, *vec2*, with 10 elements all set to 7.

Use the `swap()` method to swap the contents of *vec* and *vec2*, then display the contents of each vector. Use a separate line for the contents of each vector.

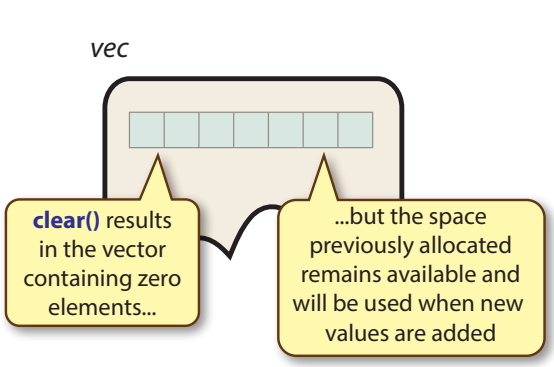
Other methods of the `vector` class are not available in the `array` class. These are:

clear()

The function `clear()` removes all of the elements from the vector, leaving it with a size of zero. Note that the space allocated for a vector's data is not freed by this operation but remains available for new values to be added (see FIG-19.9).

FIG-19.9

The Effect of *clear()*



max_size()

The `max_size()` method returns the maximum size a vector can become. The main (but not only) factor determining the value returned is the memory available.

resize()

The `resize()` changes the number of values within the vector to a specified amount. If this increases the number of values held, new cells in numeric vectors are normally zero filled. For instance, the following code

```
vector<int> vec{ 2,4,6 };  //Create list containing 2,4,6
vec.resize(10);            //Adds 7 cells (all set to zero)
//*** Display contents ***
for (int v : vec)
   cout << v << ' ';
cout << endl;
```

would display the values

```
2 4 6 0 0 0 0 0 0 0
```

Optionally, the value given to any new cells can also be specified. For example, changing the previous code call to `resize()` to

```
vec.resize(10,-2);
```

would resize *vec* to contain 10 cells, with the value held within the seven new cells

set to -2 which would result in the output

```
2 4 6 -2 -2 -2 -2 -2 -2 -2
```

Using `resize()` to reduce the number of values in the vector will delete cells starting with the highest index cell. For example, executing the following code (as part of a complete program)

```
std::vector<int> vec { 2, 4, 6, 8, 10, 12, 14, 16 };
//*** Reduce size ***
vec.resize(5);
//*** Display contents ***
for (int v : vec)
   cout << v << ' ';
cout << endl;
```

would display the values

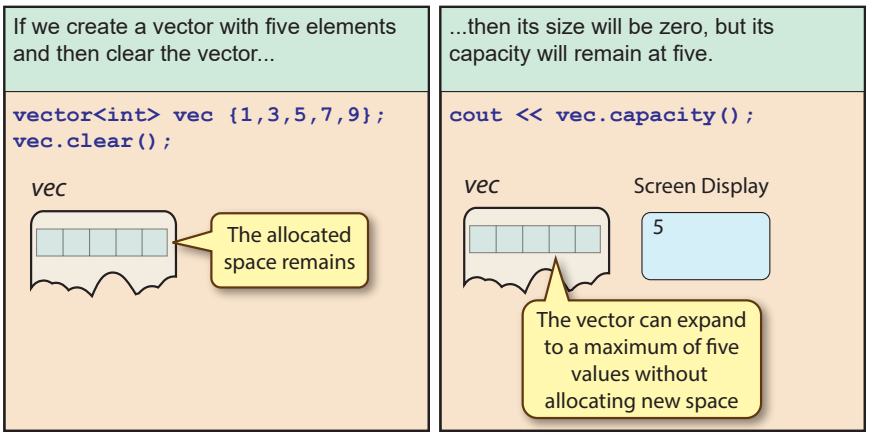
```
2 4 6 8 10
```

capacity()

The number of values stored in a vector does not necessarily relate to the amount of space allocated to that vector. While we can use `size()` to discover the number of values, we can use `capacity()` to discover how many cells are currently allocated to a vector object (see FIG-19.10).

FIG-19.10

How *capacity()* Works



The program in FIG-19.11 creates a 100 cell vector, empties it and then displays the results obtained from calls to `size()`, `capacity()` and `empty()`.

FIG-19.11

Using a *vector* Object

```
#include <iostream>
#include <vector>
using namespace std;


//*** Demonstrates use of clear(), size(), capacity() ***
//*** and empty()                                       ***
int main()
{
   vector<int> vec(100);
   //*** Clear the vector ***
   vec.clear();
   //*** Display the number of values and the number of cells ***
   cout << vec.size() << endl;
```

```
    cout << vec.capacity() << endl;
    //*** Indicate if vector contains no values ***
    if (vec.empty())
       cout << "Vector is empty" << endl;
    else
       cout << "Vector is not empty" << endl;
}
```

Activity 19.14

Start a new project called *UsingVectors02* and implement the code given above.

Check out the results produced when the program is run.

reserve()

Although a vector will, when necessary, automatically create new cells for any values we add, we can force the allocation of new cells by using the `reserve()` method. This increases the number of cells in the vector to a specified number. For example, we could increase the cells in *vec* to 200 using the line

```
vec.reserve(200);
```

Note that the value given is the total number of cells, not the increase in the number of cells.

If the value given is less than the current number of cells (in other words, the value given by `capacity()`), the command is ignored.

Of course, it's most efficient to create a vector of an appropriate size during the object's declaration. Where it's not possible to do that, increasing the size by a block at a time means that there is less chance that there will be a need to move all of the data to a new block. In order to reserve additional space, the existing contents of the vector may have to be moved to new locations.

shrink_to_fit()

The function `shrink_to_fit()` when called is, in effect, a request to reduce the number of cells reserved by the vector to match exactly the number of values in that vector. However, this reduction is not guaranteed and is dependent on the optimal use of memory.

In the code below we first create a vector containing the values 2,4, and 6 before increasing the number of cells to 10. We then call `shrink_to_fit()`, displaying the vector's capacity before and after the call.

```
std::vector<int> vec{ 2,4,6 };   //*** Initial vector contents
//*** Increase number of cells to 200
vec.reserve(200);
//*** Display the capacity ***
cout << "Initial number of cells " << vec.capacity() << endl;
//*** Shrink to number of values held ***
vec.shrink_to_fit();
//*** Display capacity again ***
cout << "Final number of cells " << vec.capacity() << endl;
```

This code outputs

```
Initial number of cells 200
Final number of cells 3
```

Assignment Operator (=)

The `vector` class defines the assignment operator to allow us to copy the contents of one vector object to another with a single statement. For example, the following code copies the contents of vector *list1* to vector *list2* and displays the result:

```
std::vector<int> vec1{ 2,4,6 };  //Vector with three values
std::vector<int> vec2;           //Vector with no values
//*** Copy vec1 to vec2 ***
vec2 = vec1;
//*** Display the contents of vec2 ***
for (int v : vec2)
   cout << v << ' ';
cout << endl;
```

The size of the receiving vector is automatically adjusted (up or down) to accommodate the new data.

push_back()

The `push_back()` method allows us to add a new value to the "end" of the vector, increasing its size by 1. For example,

```
vector<int> vec{ 2,4,6,8,10 };
vec.push_back(12);
```

Adds the value 12 to the end of *vec1* changing its contents to 2,4,6,8,10,12. If adding the value has increased the vector's size beyond its current capacity, the reserved space will be increased. It is possible that this will involve moving all existing data to a new location.

Activity 19.15

Start a new project called *UsingVectors03* and set up a vector, *vec*, containing the values 2,4,6,8,10.

Display the size and capacity of the vector.

Use `push_back()` to add the value 12 to the vector and then display the size and capacity again. Are the size and capacity equal?

pop_back()

The `pop-back()` method removes the "end" element from the vector. For example,

```
vector<int> vec1{ 2,4,6,8,10 };
vec1.pop_back();
```

would reduce the contents of *vec1* to 2,4,6,8. This effects the size but not the capacity.

Attempting to remove a value from an empty vector will cause a runtime error.

Iterator-Based Methods

Like the `array` class, the `vector` class makes use of random-access iterators to access elements within the `vector` object.

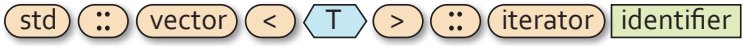
We can declare a vector iterator with a line such as:

```
std::vector<int>::iterator iter;
```

The general form of the declaration is shown in FIG-19.12.

FIG-19.12

Declaring a *vector* Iterator



where:

T	is the type of value the vector which the iterator is to reference contains.
identifier	is the name to be assigned to the iterator variable.

However, it is much more likely that `auto` will be used when declaring an iterator.

The `vector` class has several iterator-related methods in common with the `array` class:

begin()

Returns an iterator value referencing the first value in the vector.

end()

Returns an iterator reference to an address following the last value in the vector. For a vector, *vec*, use the term

```
vec.end()-1
```

to access the final element.

Typically, `begin()` and `end()` will be used together in a loop structure to access every value in the vector:

```
for(auto iter = vec.begin(); iter != vec.end(); iter++)
```

rbegin() and *rend*()

To move through the values in a vector in reverse order, use `rbegin()` to assign an iterator a reference to the last value in the vector and `rend()` to give the address of a location preceding the location of the first value in the vector.

cbegin() and *cend()*

These are identical in purpose to `begin()` and `end()`, but return a `const` iterator. This means that the iterator cannot be used to modify the value at the location to which it points. For example, while the line

```
vec.begin() = 12;
```

is legal, the line

```
vec.cbegin() = 12;
```

will fail.

crbegin() and *crend()*

These are identical in purpose to `rbegin()` and `rend()`, but return a `const` iterator.

insert()

The `insert()` method allows us to place one or more new values at any position in the vector. The method also returns an iterator value which references the newly inserted value – or first of the newly inserted values when several values are being added. The method is overloaded to make several options available.

If we assume we have executed the statement

```
vector<int> vec1{ 2,4,6,8,10 };
```

the first option is to insert only a single new value at the position specified. However, the position must be specified using an iterator value. For example,

```
vec1.insert(vec1.begin()+2, 27);
```

This would insert 27 as the third value in *vec1* giving:

2,4,27,6,8,10

As a result, the number of values in *vec1* would increase by one. Since the method always returns an iterator, we could have saved this return value by expanding our code to

```
auto iter = vec1.insert(vec1.begin()+2, 27);
```

A second option is to insert multiple copies of a single value. For example, let's say we wanted to insert three copies of the value -1 at the start of our vector, then we could write:

```
vec1.insert(vec1.begin(),3,-1);
```

Here the parameters are: position, number of copies, value and the result would have *vec1* containing the values

-1, -1, -1, 2,4,6,8,10

The third option allows a set of values from another container to be inserted into the vector. For example, if we assume the existence of a second vector

```
vector<int> vec2 {1,3,5,7,9,11};
```

then we can copy values from here into our original vector with a line such as

```
vec1.insert(vec1.begin()+1,vec2.begin()+2, vec2.begin()+4);
```

In this example, the values 5 and 7 would be copied giving *vec1* the contents:

2,5,7,4,6,8,10

The parameters for this version of *insert()* are:

destination insert address, source start address, source end address

However, notice that, as with `erase()`, the value held at *source end address* is not included in the operation.

The container from which values are being copied doesn't have to be a vector; it could be an `array` object:

```
array<int, 5> seq{ 7,14,21,28,31 };
vec1.insert(vec1.begin()+1,seq.begin(),seq.begin()+3);
```

The result of this being:

2,7,14,21,4,6,8,10

We can even copy from a standard array (remembering that the array name acts as the start address of that array):

```
int arr[]{ 5,10,15,20,25,30 };
vec1.insert(vec1.begin()+1,arr,arr+2);
```

Result:

2,5,10,4,6,8,10

Yet another option is to specify the actual values that are to be inserted, enclosing them in a set of braces:

```
vec1.insert(vec1.begin(),{-1,-3,-7});
```

giving the result

-1,-3,-7,2,4,6,8,10

assign()

The `assign()` method allows us to assign a new set of values to the vector – previously held values are deleted. There are three variations of this operation. In the simplest, we specify the number of cells and the single value to be copied into each. For example,

```
vec1.assign(3, -1);
```

changes *vec1* to contain exactly three elements each containing the value -1.

The second option allows us to specify the list of values to be placed in the vector:

```
vec1.assign({ 1,2,3,4,5,6 });
```

The last option copies values from an existing sequence to the vector. The parameters are two iterators specifying the start and end positions to be copied. For example,

```
vector<int> vec1{ 2,4,6,8,10 };
vector<int> vec2;          //Vector with no values
//*** Copy part of vec1 to vec2 ***
list2.assign(vec1.begin()+1, vec1.end()-1);
```

copies the second to fourth value from *vec1* to *vec2*. In other words, *vec2* ends up containing the values 4,6,8.

emplace()

The `emplace()` method is similar to the simplest version `insert()` in as much as it allows a single value to be added to the vector and returns an iterator which references the value that has been added. For example, the line

```
vec1.emplace(vec1.begin()+1,33);
```

will insert the value 33 into the second location in *vec1* while the line

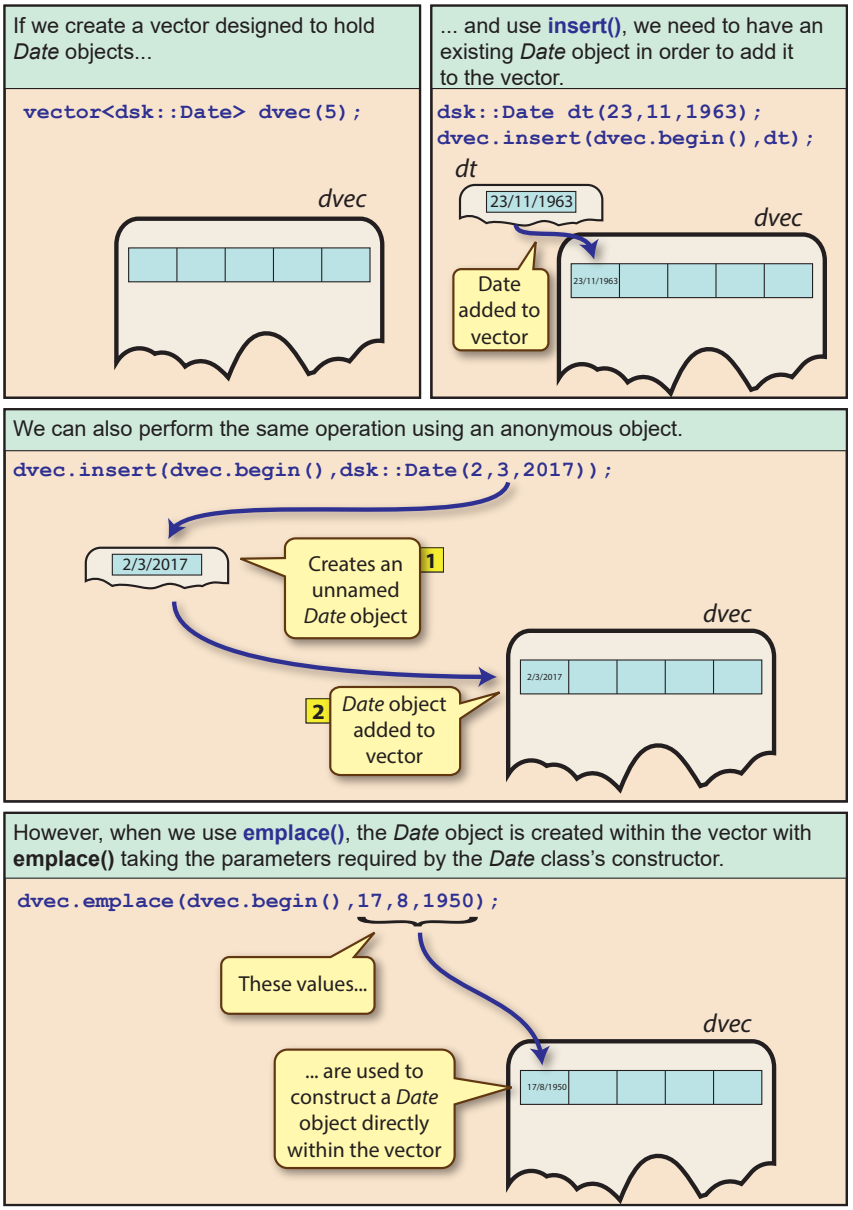
```
vec1.insert(vec1.begin()+1,33);
```

produces exactly the same result.

It's only when we create a vector containing objects that the difference becomes apparent. FIG-19.13 highlights the differences.

FIG-19.13

Using *emplace()*



emplace_back()

The method `emplace_back()` is similar to `emplace()` but always places the new value at the end of the vector. This means that no iterator parameter is required. For example, we could add a *Date* class object with the value 23/11/1963 to the end of vector *dvec* using the line

```
dvec.emplace_back(23,11,1963);
```

erase()

The `erase()` method allows us to delete a single value from a given position in a vector or to delete a sequence of values by specifying the start and end positions of those values to be deleted. Positions must be given using iterators.

For example, let's assume we have the following vector

```
vector<int> vec1{ 2,4,6,8,10,12,14 };
```

then we could delete the third value (6) using the line

```
vec1.erase(vec1.begin()+2);
```

Or we could delete the 2nd, 3rd and 4th values using the line

```
vec1.erase(vec1.begin()+1,vec1.begin()+4);
```

Notice the second parameter in the line above. The `erase()` method will delete the value given by the first parameter, but stops at the value preceding the address given by the second parameter.

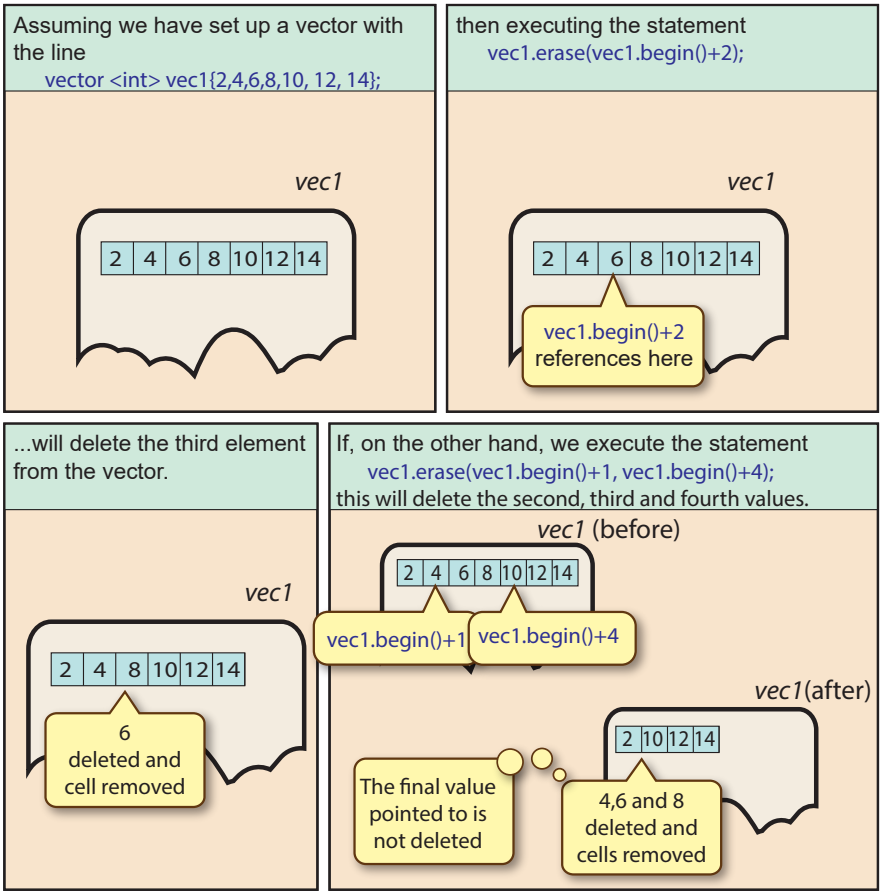
Although this may seem strange, it does mean we can use `end()` to delete from a given position to the end of the vector:

```
vec1.erase(vec1.begin()+1,vec1.end());
```

FIG-19.14 shows the effect of using a call to `erase()`.

FIG-19.14

Using *erase()*



vector Objects as Function Parameters

In the previous section covering the `array` class, we saw how the function *average()* could be defined to accept an object of that type, with the function being coded as:

```
template<class T>
```

```
double average(T list)
{
   int total{ 0 };   //Total of all values
   //*** Sum values ***
   for (auto v: list)
      total += v;
   //*** Calculate and return average ***
   return total / static_cast<float>(list.size());
}
```

And since the function is templated, we can, without any changes to its code, pass a `vector` object to the same function to display its average:

```
vector<int> numbers{ 2,4,6,8,10 };
cout << average(numbers) << endl;
```

Using a Fold Expression to Add to a *vector*

Back in Chapter 12 we had a brief look at fold expressions which proved to be a neat device for performing a basic operation on an list of parameters of unknown length.

Using a function containing a fold expression is also an efficient way of allowing several values to be added to a container. The program in FIG-19.15 creates the function *Append()* which takes a reference to a vector and a parameter pack as parameters and adds the contents of the parameter pack to the vector. The main program then calls the *Append()* function and displays the final contents of the vector.

FIG-19.15

Using a Fold Expression to Add to a Vector

```
#include <iostream>
#include <vector>
using namespace std;

template <typename T, typename... Types>
void Append(vector<T>&list, Types ...vs)
{
   (list.push_back(vs),...);
}



int main()
{
   vector<int> numbers;
   Append(numbers, 2, 4, 6, 8, 10);
   for (auto v : numbers)
      cout << v << "  ";
   cout << endl;
}
```

Activity 19.16

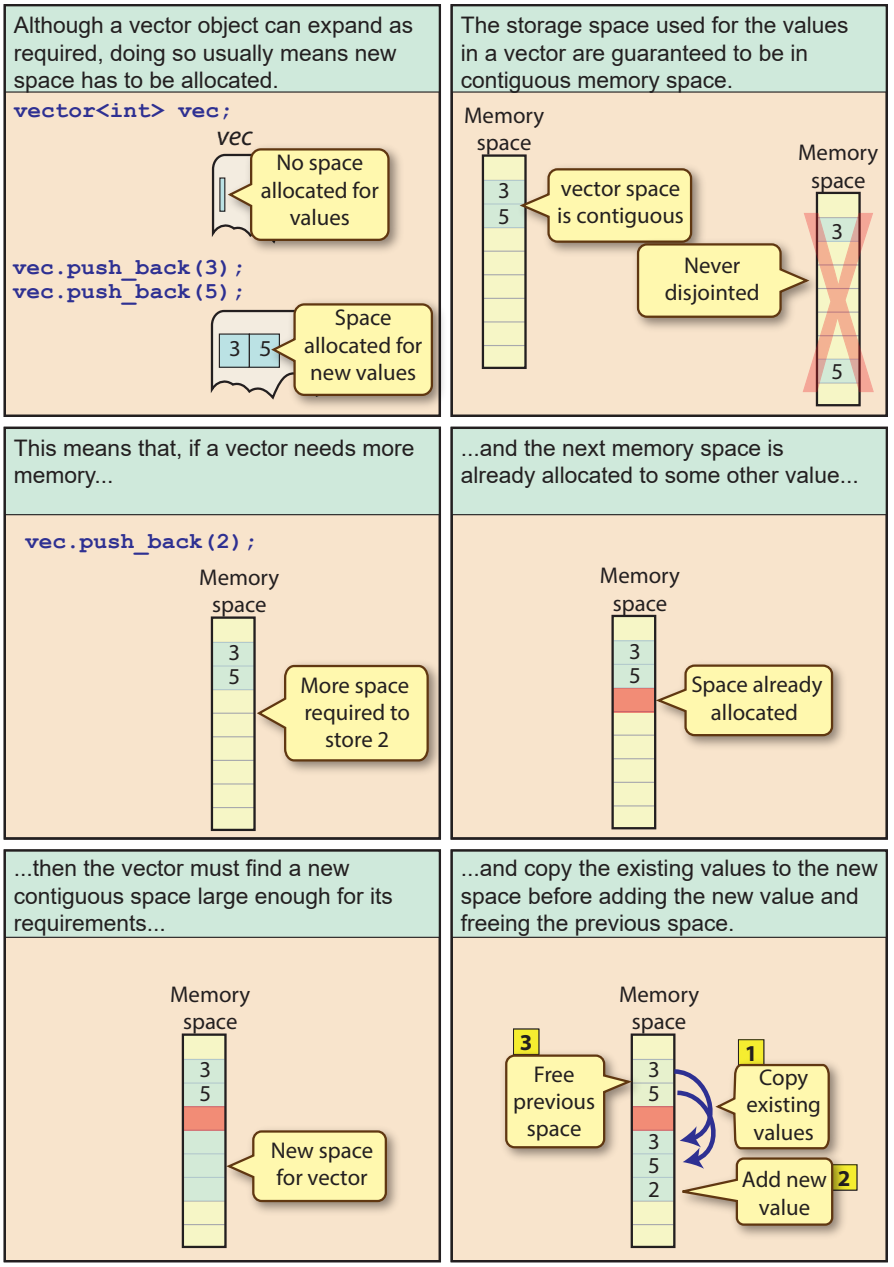
Start a new project called *UsingVectors04* and implement the code given above. Check out the results produced when the program is run.

Vector Dangers

In many ways using `vector` objects is safer and more flexible than using standard arrays, but there are a few points that we should watch out for. To understand what these problems are we must first appreciate how a vector operates (see FIG-19.16).

FIG-19.16

How a *vector* Object Operates



The first and most obvious problem with vectors is that they can be slow to execute a `push_back()` operation if different memory space has to be allocated and the existing values copied to this new space.

A second and less obvious problem occurs if an iterator value has been stored in a variable. For example, consider the following situation:

A statement assigns the start address of a vector's data to an iterator with the line

```
std::vector<int>::iterator iter = vec.begin();
```

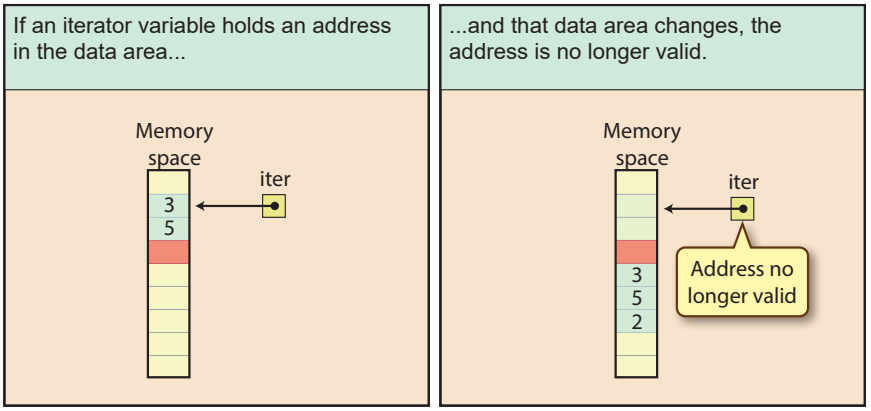
Next, a new value is added to the vector:

```
vec.push_back(13);
```

If this new value results in a need to allocate a different block of memory for *vec*'s stored values, then the address stored in *iter* is no longer valid (see FIG-19.17).

FIG-19.17

Iterator Variable Problem



We can guard against this problem by always reassigning a value to the iterator variable whenever the vector's capacity is increased.

Multi-Dimensional Vectors

We can employ the same declaration strategy that we used with `array` objects to create a multi-dimensional vector with a line such as

```
vector <vector<int>> twv;
```

But, without the number of cells being specified we have an empty structure. To fix a size, the easiest option is simply to state what values are to be assigned as part of the declaration:

```
vector <vector<int>> twv{ {1,2,3},{4,5,6} };
```

Being vectors, each "row" can vary in the number of values it contains:

```
vector <vector<int>> twv{ {1,2,3},{4,5,6,7,8,9} };
```

We can also create our two-dimensional vector with a line such as

```
vector <vector<int>> twv(3, vector<int>(5));
```

which would create a 3 by 5 structure. Like the first example, this creates a 2D vector where, initially at least, each row has the same number of cells.

If we don't assign initial values (as with the first declaration given above), then our first assignment must be a vector (see FIG-19.18).

FIG-19.18

Creating a 2D Vector

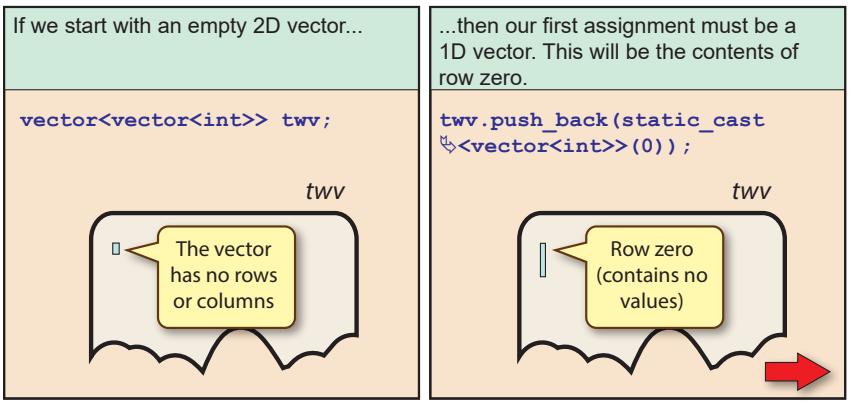
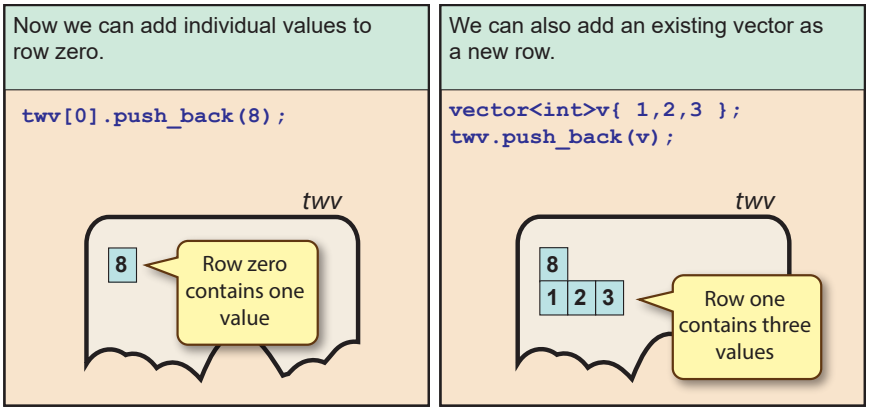


FIG-19.18
(continued)

Creating a 2D Vector



Activity 19.17

Start a new project called *Vectors2D* and create the 2D vector shown in the last frame of FIG-19.18. Display the contents of each row on a separate line.

Higher dimensional vectors simply need deeper nesting:

```
vector<vector<vector<int>>> thrv;
```

Summary

- A `vector` object is an object capable of storing a variable number of values.
- Add `#include <vector>` to use the vector class in a program.
- Use `std::vector<type,size> name` to set up a `vector` object.
- A `vector` object is created with undefined values within its elements.
- A `vector` object declaration can also include the set of values to be stored in the vector.
- A value within a `vector` object can be accessed using the object name, and an index enclosed in square brackets (e.g. `vec[2]`).
- Methods:

assign()	*option 1*: parameters *it1*, *it2*. Copies the existing value between iterator addresses *it1* and *it2*-1 to the end of the vector. *option 2*: parameters *n*, *v*. Adds *n* copies of value *v* to the end of the vector. *option 3*: parameters *list*. Adds contents of *list* to the end of the vector
at()	An alternative to [ ] but also checks that the given subscript is within range.
back()	Gives access to the last value in the vector.
capacity()	Returns the number of elements currently allocated to the vector.

clear()	Removes all values from the vector. Allocated space remains.
data()	Returns the contents of the `vector` object in the form of a traditional array.
emplace()	Adds a value at a specified address within the vector. The parameter list can be the values required by the constructor of the value being added.
emplace_back()	Adds a value to the end of the vector. The parameter list can be the values required by the constructor of the value being added.
empty()	Returns *true* if the vector contains no elements.
erase()	*option 1*: parameters *addr*. Deletes a single value at address *addr*. *option 2*: parameters *staddr*, *endaddr*. Deletes the set of values between addresses *staddr* and *endaddr*-1.
fill()	Fills every element within the vector with a specified value.
front()	Gives access to the first value in the vector.
insert()	*option 1*: parameters *addr*, *v*. Adds value *v* at address *addr*. *option 2*: parameters *addr*, *n*, *v*. Adds *n* copies of value *v* starting at address *addr*. *option 3*: parameters *addr*, *staddr*, *endaddr*. Adds the sequence of values between locations *staddr* and *endaddr*-1 starting at location *addr*. *option 4*: parameters *addr*, *list*. Adds the contents of *list* starting at address *addr*.
max_size()	Returns the maximum size a vector can become.
pop_back()	Removes the final value from the vector.
push_back()	Adds a specified value to the end of the vector.
reserve()	Reserves a specified amount of space for the vector.
resize()	Adjusts the number of values in the vector.
shrink_to_fit()	Adjust the capacity of the vector to match the actual number of values held.
size()	Returns the number of elements in the `vector` object.
swap()	Swaps the contents of two `vector` objects. The vectors must store identical value types but can be of differing size.

- Iterator Methods

begin()	Returns a reference to the first value.
end()	Returns a reference to an address following the last value.
cbegin()	Returns a `const` reference to the first value.
cend()	Returns a `const` reference to an address following the last value.
rbegin()	Returns a reference to the last value and sets up the iterator to work in reverse.
rend()	Returns a reference to an address before the first value.
crbegin()	Returns a `const` reference to the last value and sets up the iterator to work in reverse.
crend()	Returns a `const` reference to an address before the first value.

- Iterator Operators

++	Increment iterator.
--	Decrement iterator.
*	Dereference iterator.

Solutions

Activity 19.1

Code for *SetContainer*:

```
#include <iostream>
using namespace std;

class Set
{
  private:
    int store[100];
    int count;
  public:
    Set();
    bool IsEmptySet() const;
    bool IsMember(int) const;
    bool IsSubSet(const Set&) const;
    int Card() const;
    void AddToSet(int);
    void RemoveFromSet(int);
    Set Union(const Set&) const;
    Set Intersection(const Set&) const;
    Set Difference(const Set&) const;

    friend ostream& operator<<(ostream&,const Set&)
};

//*** Creates an empty set ***
Set::Set()
{
  count = 0;
}

//*** Return true if set is empty ***
bool Set::IsEmptySet() const
{
  bool result = false;
  if (count == 0)
    result = true;
  return result;
}

//*** Return true if v is a member ***
bool Set::IsMember(int v) const
{
  bool result = false;
  for (auto x : store)
    if (x == v)
      result = true;
  return result;
}

//*** Returns true if s is a subset ***
bool Set::IsSubSet(const Set& s) const
{
  bool result = true;
  for (auto x : s.store)
    if (!IsMember(x))
    {
      result = false;
      break;
    }
  return result;
}

//*** Returns the number of members ***
int Set::Card() const
{
  return count;
}

//*** Adds v to the set ***
void Set::AddToSet(int v)
{
  if (!IsMember(v))
  {
    store[count] = v;
    count++;
  }
}

//*** Removes v from the set ***
void Set::RemoveFromSet(int v)
{
  if (IsEmptySet())
    return;
  int c = 0;
  while (store[c] != v && c < count-1)
    c++;
  if (store[c] == v)
  {
    for (int k = c; k < count - 1; k++)
      store[k] = store[k + 1];
    count--;
  }
}

//*** Join two sets ***
Set Set::Union(const Set& s) const
{
  Set result = *this;
  for (auto v : s.store)
    if (!result.IsMember(v))
      result.AddToSet(v);
  return result;
}

//*** Form a set from the members common to two sets
↳***
Set Set::Intersection(const Set& s) const
{
  Set result;
  for (auto v : s.store)
    if (IsMember(v))
      result.AddToSet(v);
  return result;
}

//*** Form a set from members in the current set not
↳in s ***
Set Set::Difference(const Set& s) const
{
  Set result;
  for (auto v : store)
    if (!s.IsMember(v))
      result.AddToSet(v);
  return result;
}

//*** Display the contents of a set ***
ostream& operator<<(ostream& os, const Set& s)
{
  for (int c = 0; c < s.count; c++)
    os << s.store[c] << "  ";
  return os;
}

int main()
{
  Set A;
  Set B;
  for (int c = 2; c < 11; c+=2)
    A.AddToSet(c);
  for (int c = 2; c < 6; c ++)
    B.AddToSet(c);
  cout << A.Card() << endl;
  cout << A.IsMember(2) << endl;
  cout << A << endl;
  cout << B << endl;
  cout << A.Union(B) << endl;
  cout << A.Intersection(B) << endl;
  cout << A.Difference(B) << endl;
  //*** Additional tests ***
  B.RemoveFromSet(3);
  B.RemoveFromSet(5);
  cout << "Contents of B:\n";
  cout << B << endl;
  cout << "B subset of A? " << B.IsSubset(A)<<endl;
  cout << "A is empty? " << A.IsEmptySet() << endl;
}
```

Activity 19.2

Modified code for *SetContainer*:

```
#include <iostream>
using namespace std;

template <class T>
class Set
{
private:
  T store[100];
  int count;
public:
```

```
  Set();
  bool IsEmptySet() const;
  bool IsMember(T) const;
  bool IsSubSet(const Set<T>&) const;
  int Card() const;
  void AddToSet(T);
  void RemoveFromSet(T);
  Set<T> Union(const Set<T>&) const;
  Set<T> Intersection(const Set<T>&) const;
  Set<T> Difference(const Set<T>&) const;
  template <class T>
  friend ostream& operator<<(ostream&, const
  Set<T>&);
};

//*** Creates an empty set ***
template <class T>
Set<T>::Set()
{
  count = 0;
}

//*** Return true if set is empty ***
template <class T>
bool Set<T>::IsEmptySet() const
{
  bool result = false;
  if (count == 0)
    result = true;
  return result;
}

//*** Return true if v is a member ***
template <class T>
bool Set<T>::IsMember(T v) const
{
  bool result = false;
  for (auto x : store)
    if (x == v)
      result = true;
  return result;
}

//*** Returns true if s is a subset ***
template <class T>
bool Set<T>::IsSubSet(const Set<T>& s) const
{
  bool result = true;
  for (auto x : s.store)
    if (!IsMember(x))
    {
      result = false;
      break;
    }
  return result;
}

//*** Returns the number of members ***
template <class T>
int Set<T>::Card() const
{
  return count;
}

//*** Adds v to the set ***
template <class T>
void Set<T>::AddToSet(T v)
{
  if (!IsMember(v))
  {
    store[count] = v;
    count++;
  }
}

//*** Removes v from the set ***
template <class T>
void Set<T>::RemoveFromSet(T v)
{
  if (IsEmptySet())
    return;
  int c = 0;
  while (store[c] != v && c < count-1)
    c++;
  if (store[c] == v)
  {
    for (int k = c; k < count - 1; k++)
      store[k] = store[k + 1];
    count--;
  }
}

template <class T>
Set<T> Set<T>::Union(const Set<T>& s) const
{
  Set result = *this;
  for (auto v : s.store)
    if (!result.IsMember(v))
      result.AddToSet(v);
  return result;
}

//*** Form a set from the members common to two sets
***
template <class T>
Set<T> Set<T>::Intersection(const Set<T>& s) const
{
  Set result;
  for (auto v : s.store)
    if (IsMember(v))
      result.AddToSet(v);
  return result;
}

//*** Form a set from members in the current set not
in s ***
template <class T>
Set<T> Set<T>::Difference(const Set<T>& s) const
{
  Set result;
  for (auto v : store)
    if (!s.IsMember(v))
      result.AddToSet(v);
  return result;
}

//*** Display the contents of a set ***
template <class T>
ostream& operator<<(ostream& os, const Set<T>& s)
{
  for (int c = 0; c < s.count; c++)
    os << s.store[c] << "  ";
  return os;
}


int main()
{
  Set<int> A;
  Set<int> B;
  for (int c = 2; c < 11; c+=2)
    A.AddToSet(c);
  for (int c = 2; c < 6; c ++)
    B.AddToSet(c);
  cout << A.Card() << endl;
  cout << A.IsMember(2) << endl;
  cout << A << endl;
  cout << B << endl;
  cout << A.Union(B) << endl;
  cout << A.Intersection(B) << endl;
  cout << A.Difference(B) << endl;
  //*** Additional tests ***
  B.RemoveFromSet(3);
  B.RemoveFromSet(5);
  cout << "Contents of B:\n";
  cout << B << endl;
  cout << "B subset of A? " << B.IsSubset(A)<<endl;
  cout << "A is empty? " << A.IsEmptySet() << endl;
}
```

Activity 19.3

Changes to *SetContainer*:

At the start of the code:

```
#include <iostream>
#include <ChronoLib.h>

using namespace std;
```

In `main()`:

```
Set<dsk::Date> A;
Set<dsk::Date> B;
//*** Store dates in each set ***
A.AddToSet(dsk::Date(1, 12, 2001));
A.AddToSet(dsk::Date(23, 11, 1963));
A.AddToSet(dsk::Date(20, 7, 1969));
```

```
B.AddToSet(dsk::Date(23, 11, 1963));
B.AddToSet(dsk::Date(6, 11, 1999));
```

Unfortunately, the compiler identifies an error in the code:

```
Error  C2678   binary '==': no operator found which
takes a left-hand operand of type 'const T' (or
there is no acceptable conversion)
```

Activity 19.4

In *MyLibraries* project:

Changes to *Date* in *ChronoLib.h*:

```
//*** Date class declaration ***
class Date
{
  private:
    int day;
    int month;
    int year;

    long ToJDN() const;
    Date FromJDN(long) const;

  public:
    static const int DAYSINYEAR = 365;
    static int count;
    explicit Date(int = 1, int = 1, int = 2001);
    void SetDate(int, int, int);
    int GetDayOfWeek() const;
    bool IsLeapYear() const;
    int GetDay() const;
    int GetMonth() const;
    int GetYear() const;
    Date operator+(int) const;
    int operator-(const Date&) const;
    Date operator-(int) const;
    bool operator==(const Date&) const;
    bool operator!=(const Date&) const;
    void operator+=(int);
    Date operator++();
    Date operator++(int);
    Date operator--();
    Date operator--(int);
    int GetDaysInYear() const;
    static int GetDAYSINYEARconst();
    static int GetCount();
    ~Date();

    friend std::ostream& operator<<(std::ostream&,
    ↳const Date&);
    friend std::istream& operator>>(std::istream&,
    ↳Date&);
    friend Date operator+(int, Date&);
};
```

Changes to method headings *Date.cpp*:

```
long Date::ToJDN() const
Date Date::FromJDN(long jdn) const

int Date::GetDayOfWeek() const
bool Date::IsLeapYear() const
int Date::GetDay() const
int Date::GetMonth() const
int Date::GetYear() const
Date Date::operator+(int d) const
int Date::operator-(const Date& dt) const
Date Date::operator-(int d) const
bool Date::operator==(const Date& dt) const
bool Date::operator!=( const Date& dt) const
int Date::GetDaysInYear() const
```

With these changes, the *SetContainer* project's code should now compile and run.

Activity 19.5

Sets containing *Time* objects can be created since the *Time* class defines both the == and != operations as well as using the `const` term where appropriate.

For example, *SetContainer*'s `main()` code could be:

```
int main()
{
  Set<dsk::Time> A;
  Set<dsk::Time> B;
  //*** Store dates in each set ***
  A.AddToSet(dsk::Time(1, 12, 20));
  A.AddToSet(dsk::Time(23, 11, 19));
  A.AddToSet(dsk::Time(8, 7, 16));
  B.AddToSet(dsk::Time(23, 11, 19));
  B.AddToSet(dsk::Time(8, 7, 16));
  cout << A.Card() << endl;
  cout << A << endl;
  cout << B << endl;
  cout << A.Union(B) << endl;
  cout << A.Intersection(B) << endl;
  cout << A.Difference(B) << endl;
  //*** Additional tests ***
  B.RemoveFromSet(dsk::Time(8, 7, 16));
  cout << "\n\nContents of B : \n";
  cout << B << endl;
  cout << "\nB subset of A ? " << A.IsSubSet(B) <<
  ↳endl;
  cout << "\nA is empty ? " << A.IsEmptySet() <<
  ↳endl;
}
```

Activity 19.6

Modified code for *UsingArrayObjects01*:

```
#include <iostream>
#include <array>
using namespace std;

int main()
{
  array<int, 5> values;
  //*** Assign random values ***
  for (int c = 0; c < values.size(); c++)
    values[c] = rand();
  //*** Display contents ***
  cout << "Contents of value:\n";
  for (int c = 0; c < values.size(); c++)
    cout << values.at(c) << endl;
  //*** Display first and last values ***
  cout << "\nFirst values : " << values.front() <<
  ↳endl;
  cout << "Last value : " << values.back() << endl;
  //*** Create a new array, elements set to -1 ***
  array<int, 5> copy;
  copy.fill(-1);
  //*** Display the contents of copy ***
  cout << "\n\nContents of copy:\n";
  for (int c = 0; c < copy.size(); c++)
    cout << copy[c] << endl;
  //*** Duplicate values in copy ***
  copy = values;
  //*** Display latest values in copy ***
  cout << "\nContents of copy:\n";
  for (int c = 0; c < copy.size(); c++)
    cout << copy[c] << endl;
  if (values == copy)
    cout << "\nBoth arrays contain the same
    ↳values\n";
  else
    cout << "\nContents of both arrays are
    ↳different\n";
  //*** Change last element in values to -8 ***
  values.back() = -8;
  cout << "Contents of values:\n";
  for (int c = 0; c < values.size(); c++)
    cout << values.at(c) << endl;
  //*** Swap the contents of the arrays ***
  values.swap(copy);
  //*** Display the contents of both arrays ***
  cout << "\n\nContents of value:\n";
  for (int c = 0; c < values.size(); c++)
    cout << values.at(c) << endl;
  //*** Display the contents of copy ***
  cout << "\n\nContents of copy:\n";
  for (int c = 0; c < copy.size(); c++)
    cout << copy.at(c) << endl;
}
```

Activity 19.7

Code for *UsingArrayObjects2*:

```
#include <iostream>
#include <array>
```

```
using namespace std;

//*** Function prototypes ***
double average(int list[], int n);
template<class T>double average(T list);


int main()
{
   array<int, 5> values{ 2,4,6,8,10 };
   cout << average(values.data(), values.size()) <<
   ↳endl;
   cout << average(values) << endl;

}

double average(int list[], int n)
{
   int total{ 0 };  //Total of all values
   //*** Sum values ***
   for (int c = 0; c < n; c++)
      total += list[c];
   //*** Calculate and return average ***
   return total / static_cast<float>(n);
}

template<class T>
double average(T list)
{
   int total{ 0 };  //Total of all values
   //*** Sum values ***
   for (auto v : list)
      total += v;
   //*** Calculate and return average ***
   return total / static_cast<float>(list.size());
}
```

Activity 19.8

Modified code for *UsingArrayObjects01*:

```
int main()
{
   array<int, 5> values;
   //*** Assign random values ***
   for (int c = 0; c < values.size(); c++)
      values[c] = rand();
   //*** Display contents ***
   cout << "Contents of value:\n";
   for (int c = 0; c < values.size(); c++)
      cout << values.at(c) << endl;
   //*** Create iterator ***
   array<int,5>::iterator iter;
   //*** Reference and display the first value ***
   iter = values.begin();
   cout << *iter << endl;
}
```

Activity 19.9

Modified code for *UsingArrayObjects01*:

```
int main()
{
   array<int, 5> values;
   //*** Assign random values ***
   for (int c = 0; c < values.size(); c++)
      values[c] = rand();
   //*** Display contents ***
   cout << "Contents of value:\n";
   for (int c = 0; c < values.size(); c++)
      cout << values.at(c) << endl;
   //*** Create iterator ***
   array<int,5>::iterator iter;
   //*** Reference and display the first value ***
   iter = values.begin();
   cout << *iter << endl;
   //*** Reference and display the last value ***
   iter = values.end()-1;
   cout << *iter << endl;
}
```

Remember, `end()` returns an address after the final value, so we need to subtract 1 to get back to the address of the final value in the array.

Activity 19.10

Modified code for *UsingArrayObjects01*:

```
int main()
{
   array<int, 5> values;
   //*** Assign random values ***
   for (int c = 0; c < values.size(); c++)
      values[c] = rand();
   //*** Display contents using an iterator ***
   cout << "Contents of values:\n";
   for (auto iter = values.begin(); iter !=
   ↳values.end(); iter++)
      cout << *iter << endl;
}
```

Activity 19.11

Code for *MultiArrays*:

```
#include <iostream>
#include <array>
using namespace std;

int main()
{
   array<array<int, 2>, 3> arr{ 2,4,6,8,10,12 };
   //*** Multiply each element by 3 ***
   for(auto& row: arr)
      for (auto& v : row)
      v *= 3;
   //*** Display the contents of array ***
   for (auto row : arr)
   {
      for (auto v : row)
         cout << v << " ";
      cout << endl;
   }
}
```

Activity 19.12

Code for *UsingVectors01*:

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
   vector<int> vec(5, -3);
   vec[3] = 21;
   cout << "Contents of vec : \n";
   for (int c = 0; c < 5; c++)
      cout << vec[c] << endl;
}
```

Activity 19.13

Modified code for *UsingVector01*:

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
   vector<int> vec(5, -3);
   vector<int> vec2(10, 7);
   vec[3] = 21;
   vec.swap(vec2);
   cout << "Contents of vec : \n";
   for (int c = 0; c < vec.size(); c++)
      cout << vec[c] << "  ";
   cout << "\n\nContents of vec2 : \n";
   for (int c = 0; c < vec2.size(); c++)
      cout << vec2[c] << "  ";
   cout << "\n\n";
}
```

Activity 19.14

The output from the program is

```
0
100
Vector is empty
```

Having executed `vec.clear()` the vector contains no values but still has the reserved space for 100 entries.

Activity 19.15

Code for *UsingVector03*:

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
  vector<int> vec{ 2,4,6,8,10 };

  cout << vec.size() << endl;
  cout << vec.capacity() << endl;
  vec.push_back(12);
  cout << vec.size() << endl;
  cout << vec.capacity() << endl;
}
```

When testing this code the results were

```
5
5
```

and

```
6
7
```

Your own results may give different capacity values.

Activity 19.16

The output produced is

```
2 4 6 8 10
```

Activity 19.17

Code for *Vectors2D*:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  vector <vector<int>> twv;
  //*** Create empty first row (row zero) ***
  twv.push_back(static_cast<vector<int>>(0));
  //*** Add a value to row zero ***
  twv[0].push_back(8);
  //*** Create a 1D vector ***
  vector<int>v{ 1,2,3 };
  //*** Add 1D vector as second row 2D vector ***
  twv.push_back(v);
  //*** Display the contents of 2D vector ***
  for (auto row : twv)
  {
    for (auto v : row)
      cout << v << " ";
    cout << endl;
  }
}
```

20

Container & Iterator Functions

In this Chapter:

- ❑ Algorithm Package Functions
- ❑ Iterator Library Functions

The Algorithm Package

Introduction

The Algorithm Package contains a set of function templates that can be used to perform operations on data within a container. For example, there are functions which allow us to search, sort, or count container data.

To make use of these functions a program must contain the line

```
#include <algorithm>
```

Creating a Test Data Container

Before we look at some of the many functions available in this library, we'll start by creating a basic program to set up a `vector` object containing random integer values (see FIG-20.1). This set of values will be our test data for many of the various functions that follow.

FIG-20.1

Creating Test Data

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
   vector<int> testdata(20);

   //*** Store random values between 0 and 50 ***
   for (int c = 0; c < 20; c++)
      testdata[c] = rand() % 51;
   //*** Display the values stored ***
   for (int v : testdata)
      cout << v << " ";
   cout << endl;
}
```

Activity 20.1

Start a new project called *TestAlgorithmLibrary* and implement the code given above.

Note that since no seed value is assigned to the random number generator, the same set of values is generated each time the program is run.

Algorithm Package Non-Modifying Functions

all_of()



Elements *start* to *finish-1* are examined.

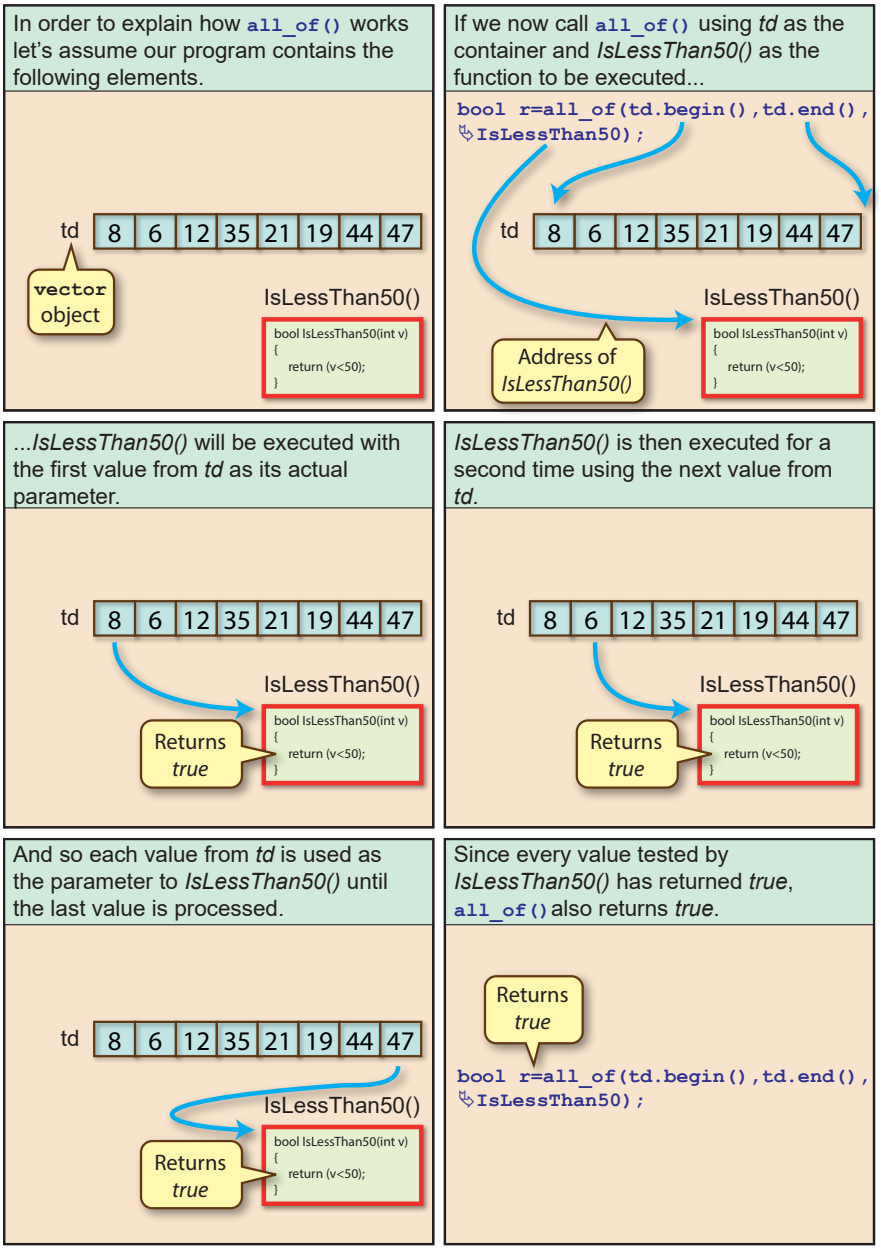
The `all_of()` function uses its first two parameters, *start* and *finish* iterators, to determine which section of the container is to be examined. The third parameter, *function address,* is the address of a function which takes exactly one parameter – the value being examined – and returns a `bool` value. The `all_of()` function passes every value in the selected area of the container to the addressed function (one at a

IsLessThan50() returns *true* if its parameter is < 50.

time) and if every value returns *true*, then `all_of()` also returns *true*. FIG-20.2 visualises the process with the *IsLessThan50()* function being used as the final parameter to `all_of()`.

FIG-20.2

How *all_of()* Works



To test `all_of()`, we'll start by adding two functions to our program:

```
bool IsLessThan50(int v)
{
   return (v < 50);
}

bool IsLessThan25(int v)
{
   return (v < 25);
}
```

The first returns *true* if parameter $v < 50$; the second returns *true* if $v < 25$.

To test `all_of()` using the *IsLessThan50()* function, we'll add the following code to `main()`:



Remember, `testdata.end()` gives an address which is after the last value held in the vector.

```
if (all_of(testdata.begin(), testdata.end(), IsLessThan50))
   cout << "All values in testdata are less than 50\n";
else
   cout << "Not all value in testdata are less than 50\n";
```

Activity 20.2

Modify *TestAlgorithmLibrary* to include the *IsLessThan50()* and *IsLessThan25()* functions (don't forget to add their prototypes).

Add the code which calls `all_of()` and run the program to check that the correct message is displayed.

Modify the newly added `if` statement to use *IsLessThan25()* rather than *IsLessThan50()* (also change the message displayed) and check the result produced.

Activity 20.3

Add a new function to *TestAlgorithmLibrary.* The function is called *IsEven()* and returns *true* if a specified value is even.

Use the address of this function in the call to `all_of()` to check if the first four numbers in *testdata* are even. Remove any other calls to `all_of()`.

This is all very well, but it's frustrating that we have to write two separate functions in order to check for values under 25 and values under 50. Unfortunately, we can't have the supplied function address give access to a function with more than one parameter.

To solve this problem, we can start by defining a functor (see Chapter 16) whose only attribute is the value we are testing for (25 and 50 in our previous examples). A constructor will set this value and a method for the parentheses operator, `()`, will be defined to perform the necessary test. This gives us the following code:

```
class IsLessThan
{
   public:
      int limit;
      IsLessThan(int lim) { limit = lim; }
      bool operator()(int v) const
      {
         return (v < limit);
      }
   };
```

Note that we have made the class's only attribute, *limit*, public to save us having to write a *GetLimit()* method. Also, the code for `operator()` is an inline function.

Next, in `main()`, we can define an object of the class, using its constructor to set the limit we require:

```
IsLessThan mylimit(50);
```

And now we can use the object name in our call to `all_of()`:

```
if (all_of(testdata.begin(), testdata.end(), mylimit))
```

But wait! Weren't we meant to supply the address of a function as the final parameter in the line above?

When we code a method for `()` in a class, the name of an object then acts as the address to the object's `operator()` method and therefore satisfies the requirements of `all_of()`'s final parameter.

So we can finish our `if` statement with the lines

```
if (all_of(testdata.begin(), testdata.begin() + 4, mylimit))
   cout << "All values are less than " <<mylimit.limit << endl;
else
   cout<<"Not all values are less than "<<mylimit.limit<<endl;
```

Activity 20.4

Modify *TestAlgorithmLibrary* by adding the *IsLessThan* class defined above.

Remove any previous calls to `all_of()` from `main()` and replace them with a check for all values in *testdata* being less than 50 followed by a test for the values being less than 25.

any_of() and *none_of()*

These two functions have an identical signature to `all_of()`, returning a `bool` value and taking two iterators and a function address as parameters.

`any_of()`	returns *true* is any of the values examined return *true* from the specified function.
`none_of()`	returns *true* if none of the values examined return *true* from the specified function.

For example, to check for at least one even number in the first four elements of *testdata* our code would be:

```
if (any_of(testdata.begin(), testdata.begin() + 4, IsEven))
   cout << "At least one of the first four values is even\n";
else
   cout << "The first four values are all odd\n";
```

To check for no even numbers in the first four elements of *testdata* our code would be:

```
if (none_of(testdata.begin(), testdata.begin() + 4, IsEven))
   cout << "Not one of the first four values is even\n";
else
   cout << "At least one of the first four values is even\n";
```

Activity 20.5

Modify *TestAlgorithmLibrary* by adding the call to `any_of()` and `none_of()` as coded above. Remove the code which calls `all_of()`.

count()

The `count()` function returns the number of values in a container that match a specified value.

This function requires start and finish iterators just like the previous functions we've looked at.

The third parameter is the actual value we are looking for in the container. For example, the program in FIG-20.3 displays the number of 5's in vector *testdata*.

FIG-20.3

Using *count()*

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
   vector<int> testdata(20, 0);

   //*** Store random values between 0 and 10 ***
   for (int c = 0; c < 20; c++)
      testdata[c] = rand() % 11;
   //*** Display the values stored ***
   for (int v : testdata)
      cout << v << " ";
   cout << endl;
   //*** Display the number of 5s ***
   cout << "There are " << count(testdata.begin(), testdata.end(),
   ↳5) << " 5s in the vector\n";
}
```

Activity 20.6

Start a new project called *TestAlgorithmLibrary02* and implement the code given above. Check that the displayed count agrees with the vector contents shown.

count_if()

The `count_if()` routine takes the usual start and finish iterators but the third parameter is a function address. This function must return a `bool` value. The `count_if()` routine returns the number of times the function specified in the parameter list has returned *true* when traversing the values held within the container being examined. For example, assuming the existence of the function *IsEven()* – as described earlier – then we can count the number of even values in *testdata* with the line

```
cout << "There are "<< count_if(testdata.begin(),testdata.end()
↳, IsEven) << " even numbers in the vector\n";
```

Activity 20.7

Modify *TestAlgorithmLibrary02* by adding the code for *IsEven()* (as created in project *TestAlgorithmLibrary)* .

Replace the call to `count()` with a call to `count_if()` so that the number of even values in *testdata* is displayed.

find()

The parameters for `find()` are the same as those for `count()`: start and finish iterators and a specific value.

The function returns an iterator value giving the point in the container where a match was found. If no match is found, the returned value is the finish iterator.

For example, we can check if *testdata* contains the value 7 at least once, using the line

```
if (find(testdata.begin(), testdata.end(), 7)!= testdata.end())
```

find_if()

The `find_if()` returns an iterator to the first value in a container which meets a given condition. As with `count_if()`, the parameters are the start and finish iterators for the search area and a function address for a routine that takes a single parameter and returns a Boolean value. For example, we could find the location of the first even number in *testdata* using a line such as

```
auto iter = find_if(testdata.begin(),testdata.end(),IsEven);
```

find_if_not()

This function, which takes the same three parameters as `find_if()`, returns an iterator which references the first value in a container which does not meet a specified condition. For example, we could find the location of the first odd number in *testdata* with the line

```
auto iter2 =find_if_not(testdata.begin(),testdata.end(),IsEven);
```

find_first_of()

This function finds the first value in a container which matches any value in a second container.

For example, let's say we have set up two vectors with the code

```
vector<int> src(10);
vector<int> wnt{ 1, 4, 7 };
```

and, assuming vector *src* is subsequently assigned a set of varying values, we can find a reference to the first value in *src* which is a 1, 4, or 7 (a value from *wnt*) using the line

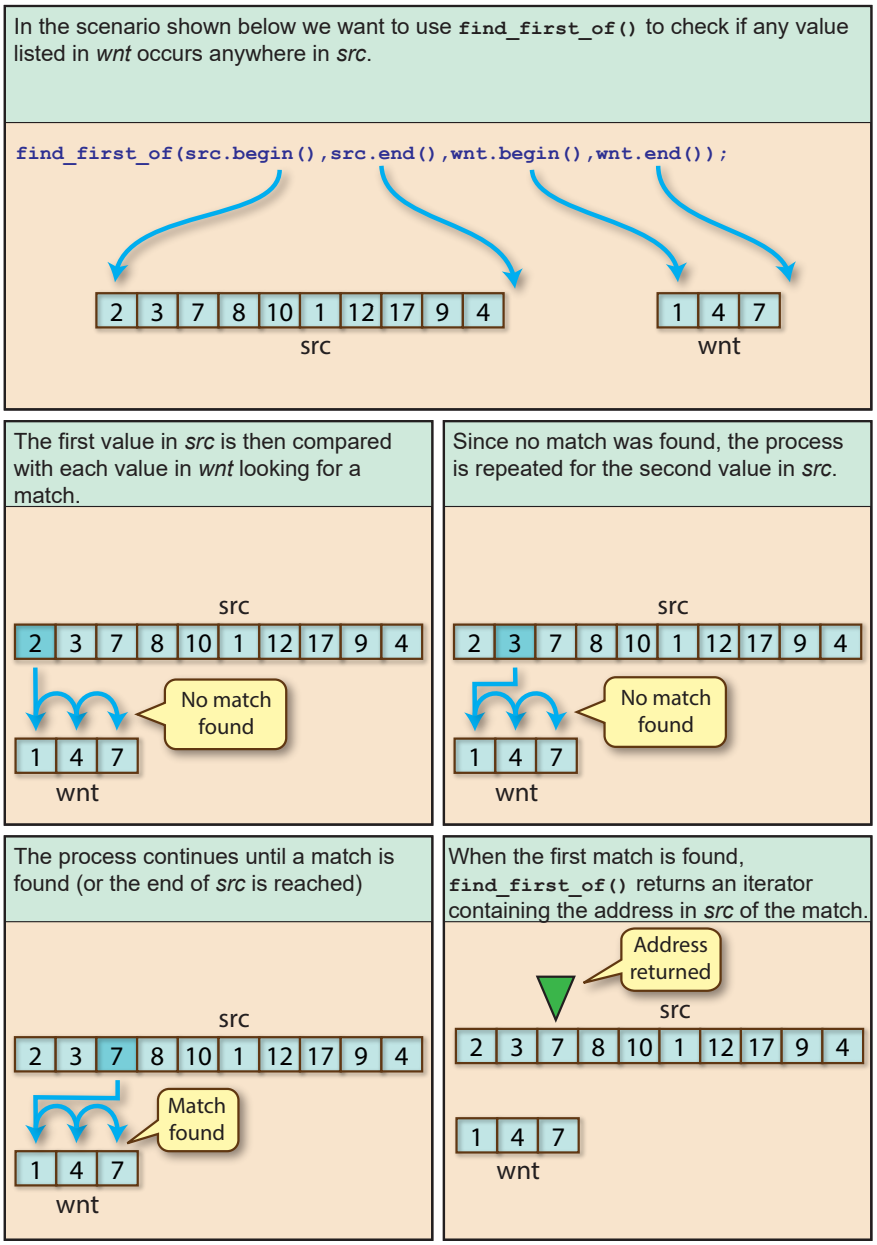
```
auto iter = find_first_of(src.begin(), src.end(), wnt.begin(),
↳wnt.end());
```

Notice that the parameters are different from what's gone before. This time we have the start and finish iterators for the container to be searched (*src*) followed by the start and finish iterators for the container holding the values to be searched for (*wnt*).

A visual representation of how `find_first_of()` operates on this data is given in FIG-20.4.

FIG-20.4

How *find_first_of()* Works



A second version of the `find_first_of()` function exists. This version takes a final, fifth parameter which is the address of a function which returns a `bool` value. Unlike our previous `bool`-returning functions this must have two parameters (it will be picking up a value from both containers used in the `find_first_of()` parameter list).

This function is used to modify what is considered a match between the values in the two containers. For example, let's say we wanted any value close to 1, 4 or 7 (as listed in *wnt*) which is found in *src* to be considered a match, then we might write the following function

```
bool Near(int v1, int v2)
{
   return (abs(v1 - v2) <= 2);
}
```

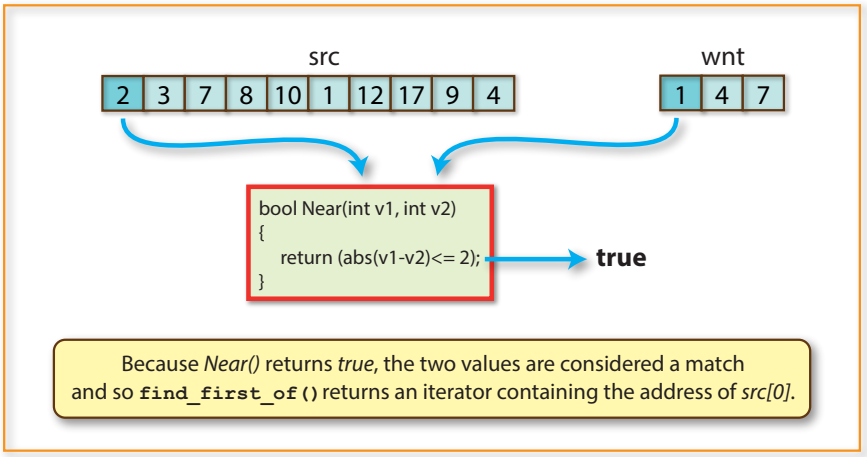
Using this routine in our call to any value `find_first_of()` would match on any value which had a difference of 2 or less from 1, 4, or 7. The call to `find_first_of()` would be

```
auto iter2 = find_first_of(src.begin(), src.end(), src.begin(),
↳src.end(), Near);
```

In our example, *src.begin()* would contain the address of *src[0]*. Since the value at that location is 2, which has an absolute difference of 2 or less when compared to the first value in *wnt,* the function *Near()* will return *true*. FIG-20.5 shows the logic involved.

FIG-20.5

How *find_first_of()* Works When a Function Address is Included



adjacent_find()

This function returns an iterator containing the address of the first matching of values in a container. For example, if a program contained the line

```
vector<int> td{ 1, 3, 5, 5, 7 };
```

searching this data using `adjacent_find()` would return a reference to *td[2]* since *td[2]* and *td[3]* contain identical values.

The parameters for `adjacent_find()` are the start and finish locations of the search area within the container. For example, we could write

```
auto iter3 = adjacent_find(testdata.begin(), testdata.end());
```

A second version of the function takes a third parameter which gives the address of a `bool`-returning function with two parameters. We can use this function to adjust what is considered "equal". For example, we could make two adjacent values "equal" if they are within a value of 2.

```
auto iter4 =adjacent_find(testdata.begin(),testdata.end(),Near);
```

The program in FIG-20.6 demonstrates the use of all the `find()` functions.

FIG-20.6

Using the *find()* Functions

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>

using namespace std;
```

FIG-20.6
(continued)

Using the *find()* Functions

```
//*** Function prototypes ***
bool IsEven(int v);
bool Near(int v1, int v2);



int main()
{
   vector<int> testdata(25);
   vector <int> searchdata{ 1, 3, 5 };

   //*** Store random values between 0 and 71 ***
   for (int c = 0; c < 25; c++)
      testdata[c] = rand() % 71;
   //*** Display the values stored ***
   cout << "Testdata : ";
   for (int v : testdata)
      cout << setw(2) << v << ' ';
   cout << "\n\n\n";

   //*** Use find() to find location of a 7 ***
   auto found = find(testdata.begin(), testdata.end(), 7) !=
   ↳testdata.end();
   if (found)
      cout << "testdata contains at least one 7\n";
   else
      cout << "testdata contains no 7s\n";

   //*** Use find_if() to find first even number ***
   auto it2 = find_if(testdata.begin(), testdata.end(), IsEven);
   if (it2 !=testdata.end())
      cout << "First even number found was " << *it2 << endl;
   else
      cout << "testdata contains no even numbers\n";

   //*** Use find_if_not() to find first odd number ***
   auto it3 = find_if_not(testdata.begin(), testdata.end(), IsEven);
   if (it3 != testdata.end())
      cout << "First odd number found was " << *it3 << endl;
   else
      cout << "testdata contains no odd numbers\n";

   //*** Use find_first_of() to look for 1, 3, or 5 ***
   auto it4 = find_first_of(testdata.begin(), testdata.end(),
   ↳searchdata.begin(), searchdata.end());
   if (it4 != testdata.end())
      cout << "1, 3, or 5 found in testdata\n";
   else
      cout << "testdata does not contain a 1, 3, or 5\n";

   //*** Use find_first_of() to find position of any value within a
   ↳value of 2 from 1, 3, or 5 * **
   auto it5 = find_first_of(testdata.begin(), testdata.end(),
   ↳searchdata.begin(), searchdata.end(), Near);
   if (it5 != testdata.end())
      cout << "A value close to 1, 3, or 5 found in testdata\n";
   else
      cout << "testdata does not contain a value close to 1, 3, or
      ↳5\n";
```



```
   //*** Find matching consecutive pair ***
   auto it6 = adjacent_find(testdata.begin(), testdata.end());
   if (it6 != testdata.end())
      cout << "At least one matching pair exists\n";
   else
      cout << "There are no matching pairs\n";

   //*** Find a consecutive pair within 2 of each other ***
   auto it7 = adjacent_find(testdata.begin(), testdata.end(),Near);
   if (it7 != testdata.end())
      cout << "At least one adjacent pair are with a value of 2 of
      ↳each other\n";
   else
      cout << "There are no matching pairs\n";
}


bool IsEven(int v)
{
   return (v % 2 == 0);
}


bool Near(int v1, int v2)
{
   return (abs(v1 - v2) <= 2);
}
```

Activity 20.8

Start a new project called *TestAlgorithmLibrary03* and implement the code given above .

Try changing the range of numbers generated to 0 to 15 and see how this affects the messages displayed.

search()

This function searches for the first occurrence of a specified sequence of values in a container and returns an iterator containing the address of the first occurrence of that sequence or, if no match is found, the address immediately following the last element of the container.

Parameters are the start and finish iterators of the area of the container to be searched, and the start and finish iterators of the area of the container holding the sequence being looked for.

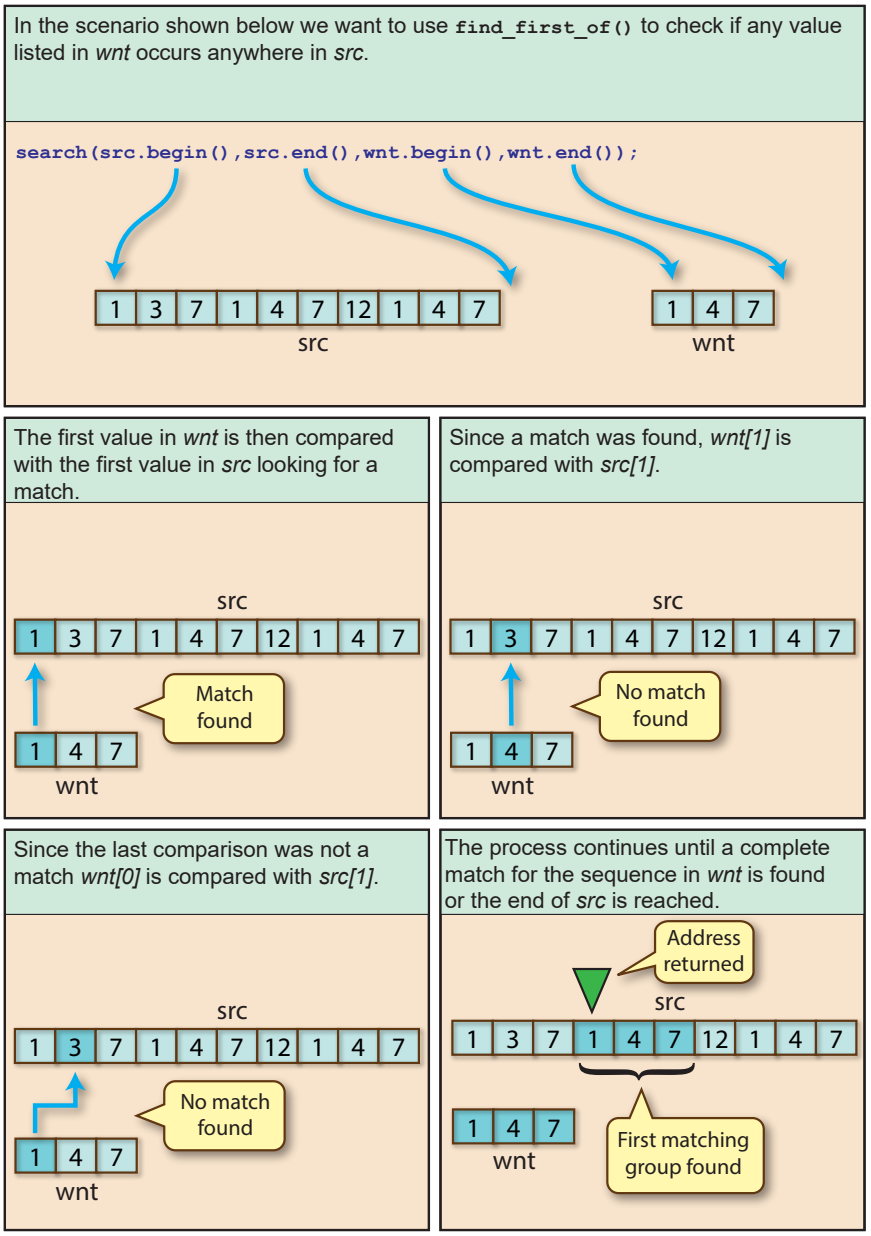
For example, we can try and find the first occurrence of the sequence 1, 4, 7 (as stored in *wnt*) in *src* using the following line of code:

```
auto it = search(src.begin(),src.end(),wnt.begin(),wnt.end());
```

A visual representation of how `find_first_of()` operates is given in FIG-20.7.

FIG-20.7

How *search()* Works



A second version of `search()` takes an additional parameter giving the address of a two-parameter function which allows us to modify what we consider equal. So, we can consider values a match if their absolute difference is 2 or less by writing

```
auto it = search(src.begin(),src.end(),wnt.begin(),wnt.begin(),
↳Near);
```

Activity 20.9

Start a new project called *TestAlgorithmLibrary04* use the values for *src* and *wnt* given in the diagram above to call both versions of `search()`.

Display the contents of the cell whose address is returned by these functions.

find_end()

Whereas `search()` finds the first occurrence of a required sequence, `find_end()` returns an iterator to the last occurrence of a sequence.

The parameters required by `find_end()` are identical to those for `search()`. So, assuming the following containers

```
vector<int> src{ 1,3,7,1,4,7,12,1,4,7};
vector<int> wnt{ 1,4,7 };
```

to find the last occurrence of 1,4,7 we would use the line

```
auto it = find_end(src.begin(),src.end(),wnt.begin(),wnt.end());
```

After the above line has been executed, it would contain the address of *src[7]* since the last occurrence of the sequence 1,4,7 occurs in elements *src[7], src[8],* and *src[9]*.

A second version of the function takes a function address as a final parameter. As with the similar option for `find_first_of()`, the function specified must return a `bool` and take two parameters. For example, we could write

```
auto it2 = find_end(src.begin(),src.end(),src.begin(),src.end(),
↳Near);
```

to check for the last three-value sequence in *src* where each of the numbers are with an absolute value of 2 of the sequence in *wnt*.

The function returns an iterator containing the address of the value in the discovered sequence.

If we were to add a call to `find_end()` to our previous program, how would we know that the value it returned referenced a different location than that found by `search()`?

We could test the two iterator values returned and check if they both contained the same value, but an even better option would be if we could find the index value corresponding to the returned address.

Luckily, the index can be easily found by the simple expression

```
int index = it - src.begin();
```

where it is the address returned by `search()`, or `find_end()` (in fact, this works for any iterator which contains the address of a cell within a vector or array object).

Activity 20.10

Modify *TestAlgorithmLibrary04* so that, in addition to the calls to `search()`, both versions of `find_end()` are also executed, searching for the sequence in *wnt*.

Modify the existing output statements so that the index corresponding to the addresses returned by the calls to `search()` are displayed rather than the value at those locations.

Also, add output statements to display the indexes of the addresses returned by the calls to `find_end()`.

search_n()

This function searches for a value *v* (fourth parameter) in a container to check if it occurs at least *n* times (third parameter) in a contiguous sequence. The part of the container being searched is given by the iterators forming the first two parameters.

The function returns an iterator to the location where the first such sequence found starts. For example, if we wanted to find two zeros in a row in the vector *src*, our call to `search_n()` would be

```
auto it = search_n(src.begin(),src.end(), 2, 0);
```

As we might expect by now, a second version of the `search_n()` function allows us to add a function address. Again, the function should return a `bool` and accept two parameters and control what values are accepted as a match. So, to search for a sequence of eight values in the range 2 to 6 we could write

```
auto it = search_n(src.begin(),src.end(),8,4,Near);
```

for_each()

The `for_each()` function allows us to have each value in a container processed by a given function. As usual the parameters begin with the start and finish iterators but the function whose address is supplied as the third parameter should not return a value (if it does, that value is ignored) and must take a single parameter. For example, let's assume we have a function *DisplayOdd()* which will only display its parameter if that value is odd. The function's code would be

```
void DisplayOdd(int v)
{
   if (v % 2 != 0)
      cout << v << "  ";
}
```

Using that function, we can display all of the odd numbers in *testdata* with the line

```
for_each(testdata.begin(), testdata.end(), DisplayOdd);
```

We can actually use the `for_each()` function to modify the contents of the container object (in this case, it's *testdata*) if we make the parameter to the function a reference value. For example, the function *Cube()*, given below, returns the cube of the parameter passed to it:

```
void Cube(int& v)
{
   v*=v*v;
}
```

and we can use this to cube all the values in *testdata* with the line

```
for_each(testdata.begin(), testdata.end(), Cube);
```

The program in FIG-20.8 demonstrates the use of `for_each()` used in association with, first *DisplayOdd()* and then *Cube()*.

FIG-20.8

Using the *for_each()* Functions

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
using namespace std;
```



FIG-20.8
(continued)

Using *for_each()* Functions

```
//*** Function prototypes ***
void DisplayOdd(int v);
void Cube(int& v);

int main()
{
   vector<int> testdata(20);

   //*** Store random values between 0 and 10 ***
   for (int c = 0; c < 20; c++)
      testdata[c] = rand() % 11;
   //*** Display the values stored ***
   for (int v : testdata)
      cout << setw(2)<< v << ' ';
   cout << endl;
   //*** Display only odd values ***
   cout << "Odd numbers in testdata: ";
   for_each(testdata.begin(), testdata.end(), DisplayOdd);
   cout << endl;
   //*** Cube every value ***
   for_each(testdata.begin(), testdata.end(), Cube);
   //*** Display the values stored ***
   cout << "testdata after cubing : ";
   for (int v : testdata)
      cout << v << "  ";
   cout << endl;
}

void DisplayOdd(int v)
{
   if (v % 2 == 1)
      cout << v << "  ";
}

void Cube(int& v)
{
   v*=v*v;
}
```

Activity 20.11

Start a new project called *TestAlgorithmLibrary05* and implement the code given above.

for_each_n()

This function is a slight modification to `for_each()` with the finish iterator parameter being replaced by the number of values to be processed by the function. For example, to display only the odd numbers in the first ten elements of *testdata*, the call to `for_each_n()` would be

```
for_each_n(testdata.begin(), 10, DisplayOdd);
```



We'll use this project again in Chapter 26.

Activity 20.12

Modify *TestAlgorithmLibrary05* by using `for_each_n()` so that only the first 5 values in *testdata* are cubed.

With the exception of `for_each()` and `for_each_n()` all of the previous functions were designed to examine and not modify the contents of the container being operated on. The following functions are designed to modify the container they access (or copy the results to a second container).

For the examples that follow, we will assume the following declarations have already been made:

```
vector<double> src = { 2,3,7,8,10,0,0,0,0,0 };
vector<double> dest(10);
```

copy(), copy_n(), copy_if(), and *copy_backward()*

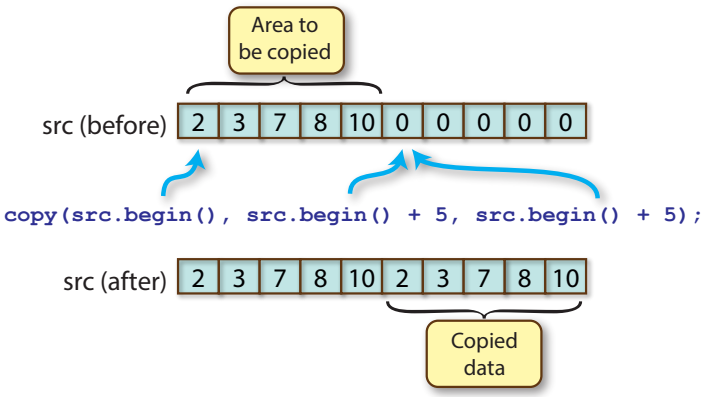
`copy()` allows us to copy a section of a container to another area in the same container or to a separate container. For example, the code

```
copy(src.begin(), src.begin() + 5, src.begin() + 5);
```

would copy the values in the first five cells of *src* to the second five cells of *src* giving the result shown in FIG-20.9.

FIG-20.9

How *copy()* Works

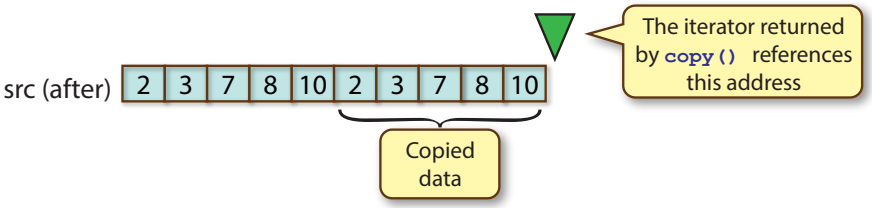


The function returns an iterator to the address immediately following the final destination cell (see FIG-20.10).

FIG-20.10

copy()'s Return Value

```
copy(src.begin(), src.begin() + 5, src.begin() + 5);
```



The function will fail if the destination area falls within the source area.

`copy_n()` replaces the second parameter with the number of cells to be copied. So, the equivalent of the previous statement would be

```
copy_n(src.begin(), 5, src.begin() + 5);
```

The function returns an iterator containing an address immediately after the final copied value.

`copy_if()` allows us to copy from a section of a container those values which meet a specified criteria (tested using a separate function) to another area or separate container. The parameters are: start iterator, finish iterator, start iterator for destination and function address. For example, if we only wanted to copy odd values to the new destination our call would be

```
copy_if(src.begin(), src.begin() + 5, src.begin() + 5, IsOdd);
```



This assumes *IsOdd()* is a coded function with the appropriate signature.

This would result in only the values 3 and 7 being copied and these would be placed in cells with the indexes 5 and 6.

The function returns an iterator which contains the address immediately after the final value to be copied.

`copy_backward()` reverses the order in which values are copied meaning that copying starts with the last value in the source container. However, this last value is copied to the last position in the destination container, so the order of the values remains the same in the destination as in the source. The function requires the start and finish iterators for the sources container and the finish iterator for the destination container. For example, the line

```
copy_backward(src.begin(), src.begin() + 5, src.end());
```

produces exactly the same result as the previous example:

```
copy(src.begin(), src.begin() + 5, src.begin() + 5);
```

The function returns an iterator containing the start of the destination area.

The program in FIG-20.11 demonstrates the use of each `copy...()` function.

FIG-20.11

Using the *find()* Functions

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

//*** Function prototypes ***
bool IsOdd(int v);

int main()
{
   vector<int> src = { 2,3,6,7,10,0,0,0,0,0 };
   vector<int> dest1(10),dest2(10),dest3(10);

   //*** Display contents of src ***
   cout << "src Original    : ";
   for (auto v : src)
      cout << v << "  ";
   cout << endl;

   //*** Execute various copy commands ***
   copy(src.begin(), src.begin() + 5, src.begin() + 5);
   copy_n(src.begin(), 5, dest1.begin());
   copy_if(src.begin(), src.begin()+5, dest2.begin(), IsOdd);
   copy_backward(src.begin(), src.begin()+5, dest3.end());

   //*** Display contents of each container ***
   cout << "src Modified    : ";
   for (auto v : src)
      cout << v << "  ";
   cout << endl;
```



FIG-20.11
(continued)

Using the *find()* Functions

```
   cout << "dest1 Contents : ";
   for (auto v : dest1)
      cout << v << "  ";
   cout << endl;
   cout << "dest2 Contents : ";
   for (auto v : dest2)
      cout << v << "  ";
   cout << endl;
   cout << "dest3 Contents : ";
   for (auto v : dest3)
      cout << v << "  ";
   cout << endl;
};

bool IsOdd(int v)
{
   return(v % 2 != 0);
}
```

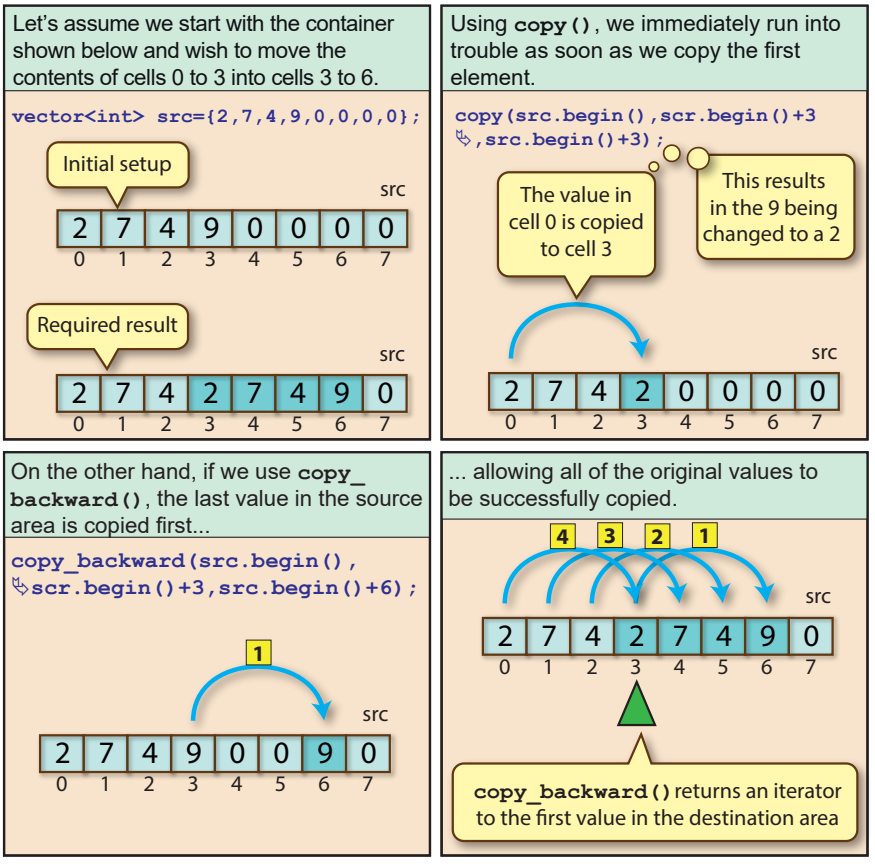
Activity 20.13

Start a new project called *TestAlgorithmLibrary06* and implement the code given above observing how the copying affects the vectors.

Before we move on it's worth understanding why we have both a `copy()` and `copy_backward()` function when they both seem to produce the same result. FIG-20. 12 describes the problem `copy_backward()` is designed to solve.

FIG-20.12

How *copy_backward()* Works



move(), move_backward()

`move()` is similar to `copy()` in that a set of values within a container are moved to a new location. The parameters are also the same as `copy()`. The return value is an iterator to the location immediately after the position of the last destination cell. However, unlike `copy()`, `move()` does not guarantee that source locations will still contain the values that were moved (though those source locations will still contain valid values – just not necessarily the original values).

`move_backward()` solves the same problem as we saw highlighted above for `copy_backward()` and uses the same parameters but, like `move()`, does not guarantee that the source area will still contain the same values after the move has been performed. We only need to use this option if the source area overlaps the destination area. The function returns an iterator to the first value in the destination area.

fill(), fill_n()

`fill()` fills a given part of a container with a specified value. For example, to fill every element in *src* with the value 11 we would write

```
fill(src.begin(), src.end(), 11);
```

`fill_n()` takes a start iterator, a quantity and a value as parameters. Starting at the iterator position, the specified number of cells are assigned the given value. To fill the first five cells of *src* with the value 42 we would write

```
fill_n(src.begin(), 5, 42);
```

transform()

The `transform()` function comes in two versions. The first of these applies another function's code to an area of values from a container and stores the results in a second specified area of the same or different container. For example, the line

```
transform(src.begin(), src.begin()+3, dest.begin(), AddTwo);
```

will apply the code in *AddTwo()* to the values held in *src[0]* to *src[3]* and store the values returned by *AddTwo()* in *dest[0]* to *dest[3]*.

To satisfy the requirements of this version of `transform()`, *AddTwo()* will need to take a single value parameter and return a result. FIG-20.13 visualises the operation.

FIG-20.13

How *transform()* (version 1) Works

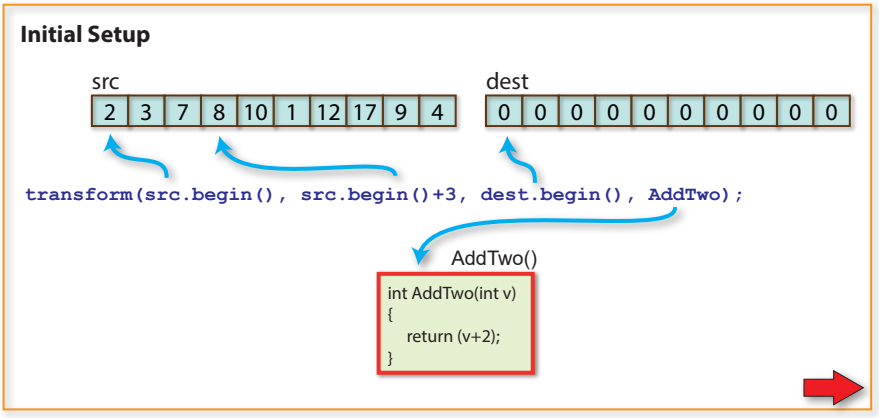
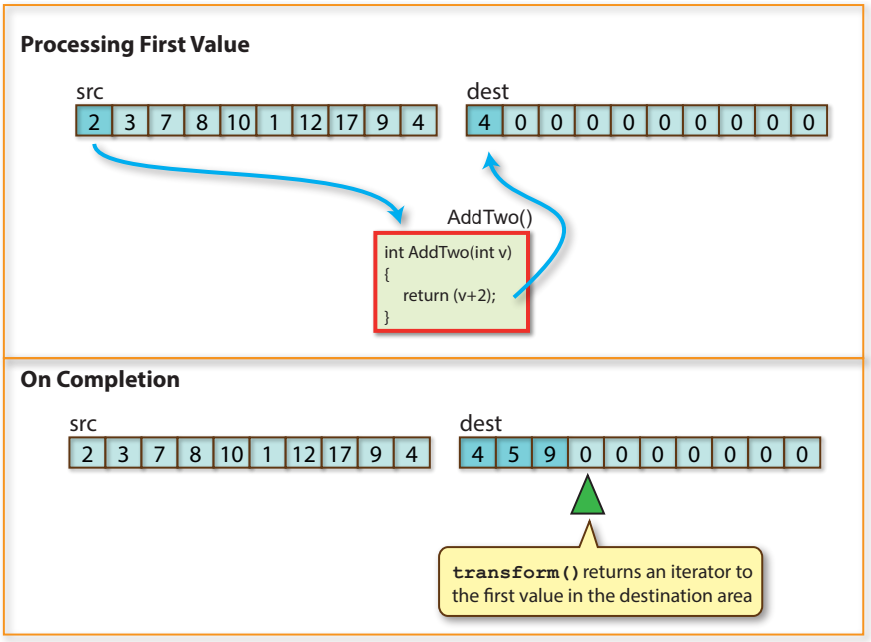


FIG-20.13
(continued)

How *transform()* (version 1) Works



The second version of `transform()` uses the elements of two areas for source data.

Its first three parameters are the start and end iterators for the first source area and the start iterator for the second source area. Next comes the start iterator for the destination container and finally, the address of the function that is to work on the source data. This time the routine must be designed to take two parameters (one from each source area) and return a single value. For example, multiplying together the elements in *src* and *src2* storing the result in *dest* and using function *Mult()*, the call to `transform()` would be

```
transform(src.begin(), src.end(), src2.begin(), dest.begin(),
Mult);
```

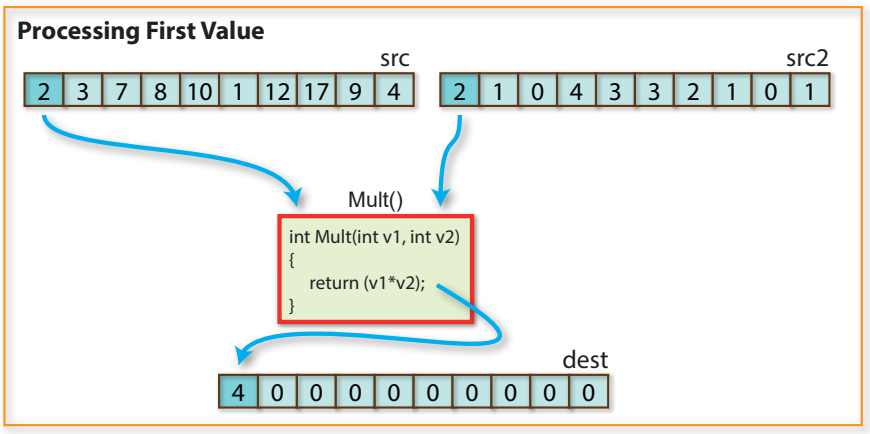
where *Mult()* is coded as

```
int Mult(int v1, int v2)
{
   return(v1 * v2);
}
```

FIG-20.14 shows how the first value to be stored in *dest* is calculated.

FIG-20.14

How *transform()* (version 2) Works



The function returns an iterator containing the address following the final value in the destination area.

The routine being called by `transform()` doesn't have to be one that we've created ourselves; it's quite possible to call a function that already exists as part of C++'s extensive library. However, attempting to call any function which is overloaded (for example, `sqrt()` which is overloaded to except `float`, `double` and `long double` arguments) will cause a compilation error.

generate(), generate_n()

`generate()` takes a start and finish iterator and overwrites the original data in that area with the results produced by a function. Although the function must have a compatible return type it has no parameters. Since the lack of a parameter restricts exactly what the function can achieve, typical usage is to return a random value. For example, the line

```
generate(src.begin(), src.begin()+3, rand);
```

will overwrite the values in *src[0]* to *src[3]* with random values generated by C++'s own `rand()` function.

If we wanted to control the range of values generated we could create our own random-generating routine

```
int Random20()
{
   return (rand()%20+1);
}
```

and use this in our call to `generate()`:

```
generate(src.begin(), src.begin()+3, Random20);
```

An even better approach would be to create a functor whose objects could be used to generate any range of value required.

Activity 20.14

Start a new project called *TestGenerate* and define a functor class called *Random* with attributes, *min* and *max*. These will define the range of integer random values that the class's only method will create.
The class constructor should set the values of *min* and *max*.
Define a method, `operator()`, which generates and returns a random number between min and max.

In `main()` create a 15 element vector object, *src*, and use a *Random* object as an argument in a call to `generate()` which is used to fill *src* with initial values.

Display the set of values in *src*.



We took a similar approach in Activity 20.4

`generate_n()` replaces the finish iterator parameter of `generate()` with the number of elements to be processed but is otherwise identical in performance to `generate()`. So, using this function we could replace the contents of *src[0]* to *src[3]* with random values using the line

```
generate_n(src.begin(), 4, rand);
```

Neither `generate()` nor `generate_n()` return a value.

remove(), remove_if(), remove_copy(), remove_copy_if()

`remove()` takes an area of a container and removes all occurrences of a specified value. For example,

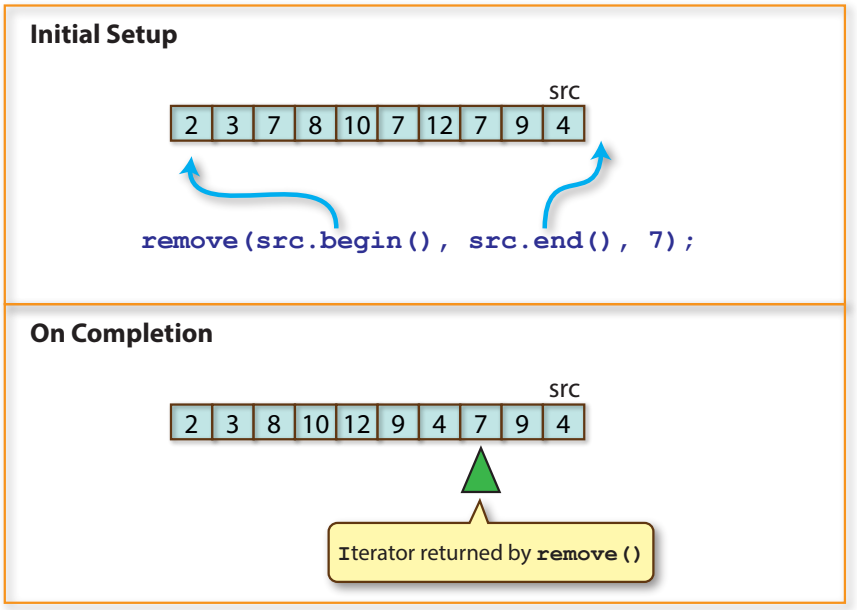
```
remove(src.begin(), src.end(), 7);
```

removes all 7's stored in *src*.

When a value is removed later values are moved down. When complete, `remove()` returns an iterator to the address immediately after the last value of the container not to have been deleted. (see FIG-20.15).

FIG-20.15

How *remove()* Works



As we can see from the diagram above, the cells outside the final area are left unchanged.

`remove_if()` examines an area of a container and removes values which meet a given criteria. That criteria is determined by a separate Boolean-returning function which takes the value under consideration as a parameter. For example,

```
remove_if(src.begin(), src.end(), IsOdd);
```

would remove all odd values from *src*. The function returns an iterator to the address immediately after the last value of the container not to have been deleted.

`remove_copy()` copies an area of a container to a second container or a different area in the original container but does not copy to the destination area entries of a specific value. For example, the line

```
remove_copy(src.begin(), src.end(), dest.begin(), 7);
```

will copy all the values in *src* to *dest* except for any entry in *src* containing the value 7. The function returns an iterator to the address immediately after the last value copied to the destination container.

`remove_copy_if()` copies an area of a container to a second container or a different area of the original container but does not copy values which meet a specific condition. For example,

```
remove_copy_if(src.begin(), src.end(), dest.begin() IsOdd);
```

will copy all even numbers from *src* to *dest*. The function returns an iterator to the address immediately after the last value copied to the destination container.

replace(), replace_if(), replace_copy(), replace_copy_if()

`replace()` takes an area of a container and replaces all occurrences of a specified value with a given new value. For example,

```
replace(src.begin(), src.end(), 7, 8);
```

replaces all 7's in *src* by 8's. `replace()` returns no value.

`replace_if()` examines an area of a container and replaces values which meet a given criteria with a new value. For example,

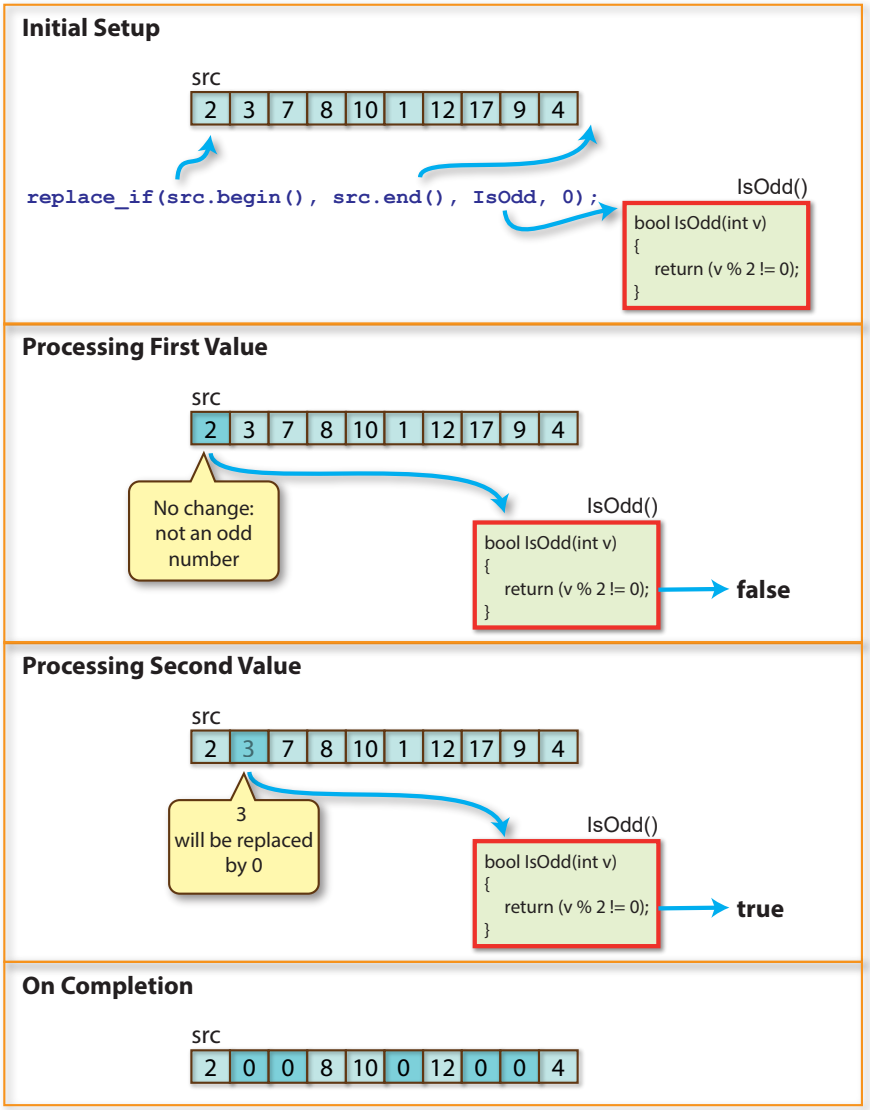
```
replace_if(src.begin(), src.end(), IsOdd, 0);
```

replaces all odd values in *src* by zero. `replace_if()` returns no value.

FIG-20.16 visualises the above `replace_if()` statement.

FIG-20.16

How *replace_if()* Works



`replace_copy()` copies an area of a container to a second container or a different area in the original container but replaces all occurrences of a specified value in the original area with a new, given value in the destination area. For example,

```
replace_copy(src.begin(), src.end(), dest.begin(), 7, 8);
```

copies the contents of *src* to *dest*, replacing any entries containing the value 7 in *src* with an entry containing the value 8 in *dest*.
`replace_copy()` returns an iterator to the address immediately after the last value added to the destination container.

`replace_copy_if()` copies an area of a container to a second container or a different area of the original container changing all occurrences of values which meet a given criteria with a new value. For example,

```
remove_copy_if(src.begin(), src.end(), dest.begin() IsOdd, 0);
```

will copy all even numbers from *src* to *dest*.
`replace_copy_if()` returns an iterator to the address immediately after the last value added to the destination container.

swap(), swap_ranges(), iter_swap()

`swap()` swaps the contents of two containers. For example,

```
swap(src, dest);
```

swaps the contents of *src* and *dest*.

`swap_ranges()` swaps part of a container with the equivalent sized area in a second container or a different area of the original container. For example,

```
swap_ranges(src.begin(), src.begin()+4, src.end()-5);
```

swaps the contents of the area *src[0]* to *src[3]* with the *src[5]* to *src[8]*.

`iter_swap()` swaps the contents of two cells. The cells' locations are given using iterators. For example,

```
iter_swap(src.begin(), src.begin()+1);
```

swaps the contents of *src[0]* and *src[1]*.

reverse(), reverse_copy()

`reverse()` reverses the order of an area of a container. For example,

```
reverse(src.begin(), src.begin()+6);
```

reverses the order of the values in *src[0]* to *src[5]*. No value is returned by `reverse()`.

`reverse_copy()` copies an area of a container to a different location and reversing the order of the values in the process. For example,

```
reverse_copy(src.begin(), src.end(), dest.begin());
```

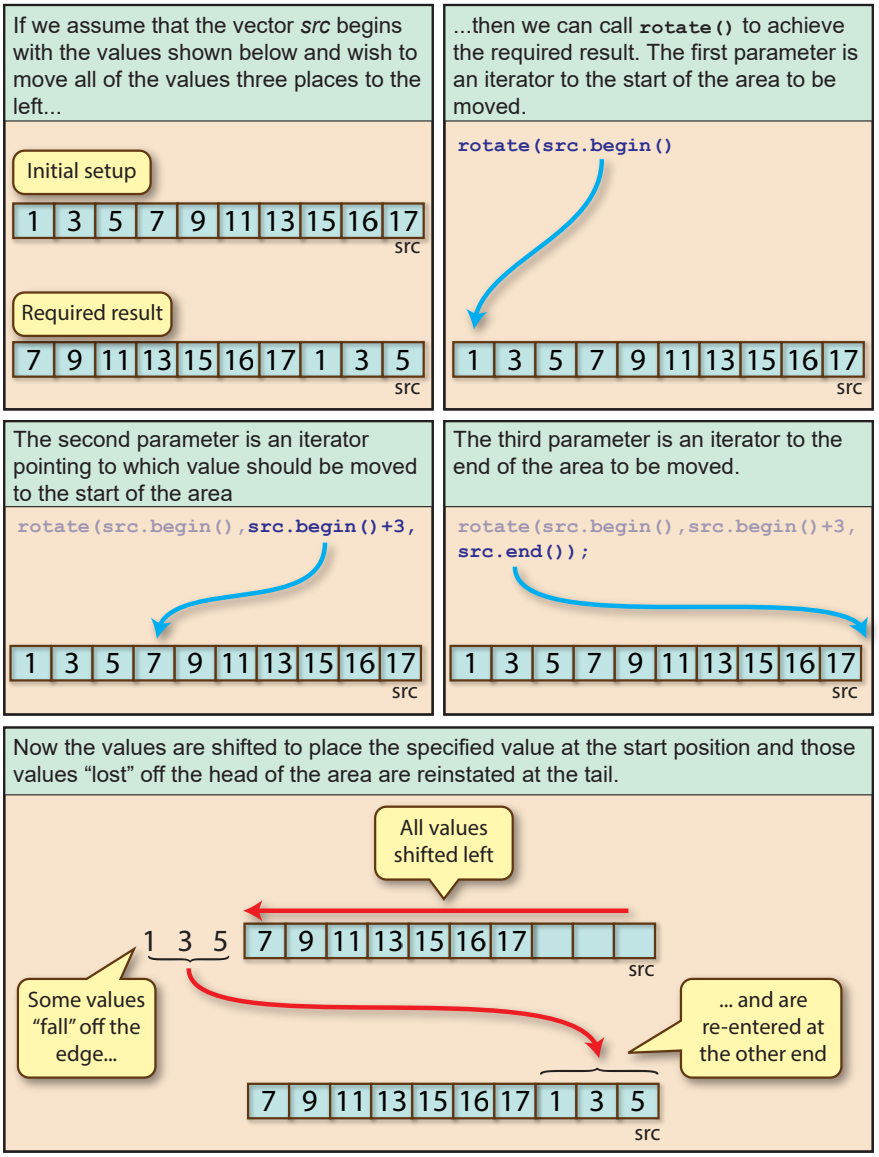
copies the contents of *src* to *dest*, reversing the order of the values. `reverse_copy()` returns an iterator to the address immediately following the last value to be copied.

rotate(), rotate_copy()

`rotate()` shifts a sequence of values within a container left or right with values that shifted outside the range looping back to the other "end" of the sequence (see FIG-20.17).

FIG-20.17

How *rotate()* Works



No value is returned by `rotate()`.

`rotate_copy()` takes an final, additional parameter over those used by `rotate()`. This final parameter is an iterator point to an area where the rotated data is to be stored.

In effect, `rotate_copy()` performs the same type of operation as `rotate()` but, instead of modifying the original container area, it stores the result of the rotation in a different area. For example,

```
rotate_copy(src.begin(), src.begin()+4,src.end(),dest.begin());
```

would store the rotated data in container *dest* without affecting the contents of *src*. `rotate_copy()` returns an iterator containing the address immediately after the final copied value.

shuffle()

Back in Chapter 8 we created an algorithm to shuffle a set of values. However, there is, in fact, an existing function that will perform that task for us when using containers.

`shuffle()` takes the start and finish position iterators and a random number generator object as parameters and shuffles the values in that part of the container.

We will not go into the details of random number classes in this text, but simply satisfy ourselves that such an object can be created with an expression such as

```
default_random_engine(time(0))
```



To use a random object, add the line `#include <random>` to your program.

So, we could shuffle the values in *src* using the line

```
shuffle(src.begin(), src.end(), default_random_engine(time(0)));
```

No value is returned by `shuffle()`.

unique(), unique_copy()

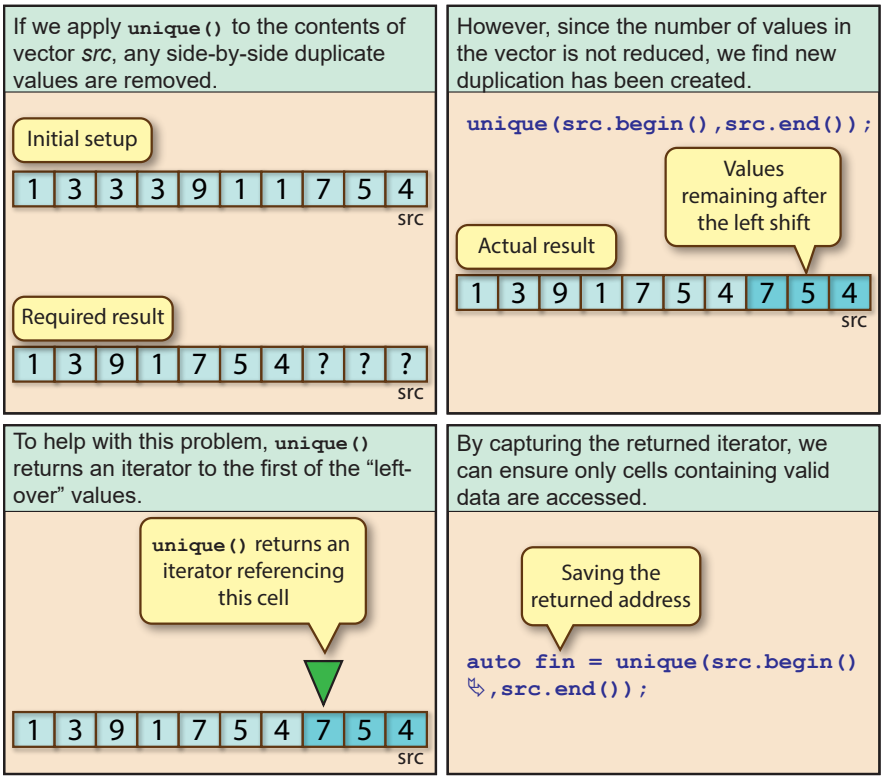
`unique()` version 1 takes a start and finish iterator and removes all contiguous duplicates. For example, to remove the duplicates from *src* we would write

```
unique(src.begin(), src.end());
```

The only problem is that the number of values in the vector are not reduced so we may find duplicate values at the end of the search area (see FIG-20.18).

FIG-20.18

How *unique()* Works



With this iterator saved, we can easily restrict access to the valid data. For example, displaying the valid data would be achieved with the code

```
for (auto iter = src.begin(); iter != fin; iter++)
   cout << *iter << "  ";
cout << endl;
```

`unique()` version 2 takes a third parameter. This is the address of a function which returns a `bool` value. The function must take two parameters of a type appropriate to those held in the container. If the function returns *true*, the second of these two values is removed from the container. This version of `unique()` also returns an iterator containing the address of the first "left-over" values.

The program in FIG-20.19 demonstrates the use of both `shuffle()` and the second version of `unique()`. The function called by `unique()`, *Approx()*, returns *true* if the absolute difference between its two parameters is <= 1.

FIG-20.19

Using *shuffle()* and *unique()*

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
#include <random>
using namespace std;

//*** Function prototypes ***
bool Approx(int v1, int v2);

int main()
{
   vector<double> src(15);

   //*** Store random value between 1 and 10 ***
   for (auto& v : src)
      v = rand() % 10 + 1;
   //*** Display original contents ***
   cout << "src (original)       : ";
   for (auto v : src)
      cout << setw(2) << v << "  ";
   cout << endl;

   //*** Shuffle src ***
   shuffle(src.begin(), src.end(), default_random_engine(time(0)));
   //*** Display shuffled contents ***
   cout << "src (shuffle)  : ";
   for (auto v : src)
      cout << setw(2) << v << "  ";
   cout << endl;

   //*** Remove adjacent values with diff <= 1 ***
   auto iter = unique(src.begin(), src.end(), Approx);
   //*** Display final contents ***
   cout << "src (unique)   : ";
   for (auto it = src.begin(); it < iter; it++)
      cout << setw(2) << *it << "  ";
   cout << endl;
}


bool Approx(int v1, int v2)
{
   return (abs(v1-v2) <= 1);
}
```

Activity 20.15

Start a new project called *TestAlgorithmLibrary07* and implement the code given above.

`unique_copy()` performs a similar task to `unique()` but copies the processed data to a new location and leaves the original untouched.

The first version of `unique_copy()` takes three parameters: start and finish iterators for the source area and a start iterator for the destination area. The function returns an iterator containing the address immediately after the last element to be copied to the destination container. For example, we could remove duplicates from *src* and store the result in *dest* using the line

```
unique_copy(src.begin(), src.end(), dest.begin());
```

The second version of `unique_copy()` takes four parameters: start and finish iterators for the source area, a start iterator for the destination area and the address of a function which returns a `bool` value and requires two parameters. For example, we could eliminate *src*'c adjacent values with a difference of 1 or less and store the result in *dest* using the line



Approx() was defined as a function in the previous program.

```
unique_copy(src.begin(), src.end(), dest.begin(), Approx);
```

Activity 20.16

Modify *TestAlgorithmLibrary07* by first shuffling the values that remain in *src* at the end of the existing code and then eliminating all adjacent values with a difference of one or less, storing the result in vector *dest*.

Display *src* after the second shuffle and *dest* after the call to `unique_copy()`.

Make sure only the part of *src* containing meaningful data is processed.

Sorting and Searching

An important aspect of any program that processes large amounts of data is its sorting and searching algorithms. These topics are worthy of a complete book on their own, but we touch on them here only to cover the relevant Algorithm Package functions.

sort()

The `sort()` function, which re-arranges the data of a container into ascending order, comes in two versions. The first of these requires only start and finish iterators used to specify the area of the container that is to be sorted. For example, the line

```
sort(src.begin(), src.end());
```

will sort the contents of *src* into ascending order.

Activity 20.17

Modify *TestAlgorithmLibrary07* by sorting and displaying the contents of *dest*.

A second version of `sort()` takes an additional parameter giving the address of a

function. The function must return a `bool` value and take two parameters of a type appropriate to those in the container being sorted. The function should return *true* if the value of the first parameter is less than that of the second.

For example, when sorting a set of characters we would want upper and lower cases to be treated as identical by the sort process. A function for this which we could use in a call to `sort()` would be

```
bool IsLess(char v1, char v2)
{
   return (tolower(v1) < tolower(v2));
}
```

Activity 20.18

Start a new project called *TestAlgorithmLibrary08* and call `sort()` using *IsLess()* to sort the contents of the vector *letters* which is defined as:

```
vector<char>letters{ 'A','B','C','a','b','c','D' };
```

binary_search()

We have already covered the `search()` method which will search a container for a specific value. However, that function assumes that the container's data is not in any specific order and hence the search technique employed is not as efficient as it could be when the data is sorted.

If we know a container's data is sorted, we can use the `binary_search()` function to perform a much more efficient search for a required value.

The `binary_search()` function comes with two options. The first of these takes the usual start and finish iterators and the value being searched for. For example,

```
bool found = binary_search(src.begin(), src.end(), 11);
```

would search *src* for the value 11 – but the contents of *src* must have been sorted previously.

The function returns a `bool` value – *true* if the required value is found, otherwise *false*. It gives no indication of where in the container a match was found!

The second version of `binary_search()` adds a final value parameter which is the address of a function. This function must return a `bool` value and take two parameters. The function's logic should return *true* if the first parameter's value is less than that of the second. For example, looking for 'd' in letters would be coded as

```
bool found = binary_search(src.begin(),src.end(),'d',IsLess);
```

Activity 20.19

Modify *TestAlgorithmLibrary08* by adding a call to the second version of `binary_search()` to search for 'd' in *letters*. Display the message "*Value found*" or "*Value not found*" as appropriate.

Summary

- To use any of the following functions, a program must contain the line
  ```
  #include <algorithm>
  ```
- Use `all_of()` to find if all of the values in a specified area of a container meet a given criteria.
- Use `any_of()` to find if any value in a specified area of a container meets a given criteria.
- Use `none_of()` to find if no value in a specified area of a container meets a given criteria.
- Use `count()` to find how many copies of a given value are in a specified area of a container.
- Use `count_if()` to find how many values in a specified area of a container meet a given criteria.
- Use `find()` to produce an iterator which references the first entry in a specified area of a container which is of a given value.
- Use `find_if()` to produce an iterator which references the first entry in a specified area of a container which meets a given criteria.
- Use `find_if_not()` to produce an iterator which references the first entry in a specified area of a container which does not meet a given criteria.
- Use `find_first_of()` to produce an iterator which references the first entry in a specified area of a container which matches any one of the values in a specified area of a second container.
- Use `adjacent_find()` to return an iterator to the first entry that is immediately followed by the identical value or a value which also meets a given criteria.
- Use `search()` to return an iterator to the first occurrence of a specified sequence of values or match the same criteria as that searched for sequence.
- Use `find_end()` to return an iterator to the last occurrence of a specified sequence of values or match the same criteria as that searched for sequence.
- Use `search_n()` to return an iterator to the first occurrence of a specified sequence of values within a number-specified area of a container or match the same criteria as that searched for sequence.
- Use `for_each()` to apply a function to a specified section of a container.
- Use `for_each_n()`to apply a function to a specified number of entries in a container.
- Use `copy()` to copy a section of a container to a new area. The two areas must not overlap.
- Use `copy_n()` to copy a number of entries in a container to a new area. The two areas must not overlap.
- Use `copy_if()` to copy the values in a section of a container which meet a given criteria to a new area. The two areas must not overlap.
- Use `copy_backward()` to copy a section of a container to a new area. The two areas may overlap.

- Use `move()` to move a section of values to a new location. The values at the original location may be changed by this operation. The two areas cannot overlap.
- Use `move_backward()` to move a section of values to a new location when the two areas overlap.
- Use `fill()` to fill an area of a container with a specific value.
- Use `fill_n()` to fill a given number of elements of a container with a specific value.
- Use `transform()` to apply an operation to an area of a container.
- Use `generate()` to create values produced by a function and store them in a container.
- Use `generate_n()` to create values produced by a function and store them in a specified number of cells in a container.
- Use `remove()` to remove a specified value from an area of a container.
- Use `remove_n()` to remove a specified value from a number of cells in a container.
- Use `remove_if()` to remove values from an area of a container if they meet a specified condition.
- Use `remove_copy()` to make a copy of an area of a container to another location but omitting any entries of a specified value.
- Use `remove_copy_if()` to make a copy of an area of a container to another location but omitting any entries that meet a specified condition.
- Use `replace()` to replace all occurrences of an old value with a new value within a specified area of a container.
- Use `replace_if()` to replace all values which meet a given condition with a new value within a specified area of a container.
- Use `replace_copy()` to copy an area of a container to a new location, replacing all occurrences of an old value with a new value.
- Use `replace_copy_if()` to copy an area of a container to a new location, replacing all values which meet a specified condition with a new value.
- Use `swap()` to swap the complete contents of two appropriate containers.
- Use `swap_ranges()` to swap an area of a container with another area. The two areas must not overlap, if using the same container.
- Use `iter_swap()` to swap the values of exactly two container cells.
- Use `reverse()` to reverse the order of the values in a given area of a container.
- Use `reverse_copy()` to copy an area of a container to a new location, reversing the original order of the data being copied to the destination.
- Use `rotate()` to shift the contents of an area of a container to the left so that a specified cell's value becomes the first value in that area. Values "falling off" the front of the area are copied back to the rear of the area.
- Use `rotate_copy()` to copy an area of a container so that the copied version is

shifted to the left so that a specified cell's value becomes the first value in that area. Values "falling off" the front of the area are copied back to the rear of the area.

- Use `shuffle()` to shuffle the order of the values in a given area of a container.
- Use `unique()` to remove all but the first of adjacent copies of identical values from an area of a container.
- Use `unique_copy()` to copy a given area of a container to a new location, removing all but the first of adjacent copies of identical values during the copy process.
- Use `sort()` to sort an area of a container into ascending order.
- Use `binary_search()` to search for a specified value within an area of a sorted container.

Iterator-Based Functions

Introduction

The *iterator* header file contains the definition of several iterator-related functions which may prove helpful when dealing with containers. To use the following functions a program must contain the line

```
#include <iterator>
```

The functions are defined within the `std` namespace.

Functions

As we saw earlier, when dealing with a container that makes use of a random-access iterator (both `array` and `vector` objects use this type of iterator) converting the address in an iterator to the equivalent index is quite simple. All that is required is to subtract the container's `begin()` iterator from the iterator to be converted. A trivial example is shown in FIG-20.20.

FIG-20.20

Iterator Subtraction

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

int main()
{
   vector <int> src{ 1,2,3,4,5,6 };
   auto iter = find(src.begin(), src.end(), 4);
   cout << "Found in cell " << iter - src.begin() << endl;
}
```

distance()

Unfortunately, the above approach won't work if the container type being used does not employ random-access iterators. In this case we can make use of the `distance()` function to convert the iterator to an index value. This function takes two iterators and returns the difference between the two. Note, however, that the iterators are in the reverse order from how they are used with the subtraction operator. For example, we could replace the `cout` line in the program above with

```
cout << "Found in cell " << distance(src.begin(),iter) << endl;
```

begin()

The `begin()` function (not to be confused with the `begin()` method in most container classes) returns an iterator. When used on a normal container object such as a vector, it returns an iterator to the first element of the container (in other words, it gives the same result as the `begin()` method).

But the normal use of this function is to apply it to a standard array (not an `array` class object). Standard arrays do not use iterators and so the various functions we covered in the previous section cannot be directly applied to arrays. To solve this problem, we can call the `begin()` function to obtain an iterator which references the first cell in a standard array. For example,

```
int list[]{ 2,4,6,8,10 };
auto startit = begin(list);
```

will set iterator *startit* to reference the first element in *list*.

end()

Of course,the `end()` function is generally used on standard arrays and returns an iterator to an address just past the last value in the array. For example,

```
auto endit = end(list);
```

With the necessary iterators, we can now apply any iterator-based function such as `find()` on our standard array. For example, we can find the index of the cell containing the value 8 with the following code:

```
auto postit = find(startit, endit, 8);
cout << "8 found in element "<<distance(startit,postit)<< endl;
```

next()

The `next()` function returns an iterator to the address of the next location in a collection (or standard array, if we've set up the appropriate iterator beforehand) . For example, if *startit* contains the address of *list[0]*, we can have *iter2* contain an iterator to *list[1]* using the line

```
auto iter2 = next(startit);
```

If we add a second parameter, we can have the returned value reference more than one cell ahead. For example,

```
auto iter2 = next(startit,3);
```

will return an iterator to *list[3]*;

advance()

This function has a similar effect to `next()`, but this time there is no return value and it is the iterator parameter itself that is modified. Also, the second parameter, giving the number of elements the iterator is to be adjusted by must be included. For example, we can have *iter2* reference *list[4]* (assuming it currently references *list[3]*) using the line

```
advance(iter2,1);
```

prev()

This function returns an iterator to the previous element in a container. For example,

```
auto iter3 = prev(endit);
```

will return an iterator containing the address of *list[4]* (remember, *endit* contains an address after the last element).

Optionally, a second parameter can be added to return an iterator which is further back. For example,

```
auto iter3 = prev(endit,2);
```

will return an iterator to *list[3]*. Note that it is a positive number that is used as the second parameter to *prev()*.

The program in FIG-20.21 creates iterators for the first and last elements in an array

list, and then uses these iterators as the parameters for `generate()` and `sort()` to sort the randomly generated values in *list*.

FIG-20.21

Using Iterators with an Array

```
#include <vector>
#include <algorithm>
#include <iomanip>
using namespace std;

class Random
{
private:
   int min;
   int max;
public:
   Random(int v1, int v2)
   {
      min = v1;
      max = v2;
   }

   int operator()()
   {
      return(rand() % (max - min) + min);
   }
};


int main()
{
   int list[10];
   //*** create iterators for start and end ***
   auto its = begin(list);
   auto itf = end(list);

   //*** Generate a set of values in list ***
   generate(its, itf, Random(1, 12));

   //*** Display the values in list ***
   cout << "Unsorted : ";
   for (int v : list)
      cout << setw(2) << v << ' ';
   cout << endl;

   //*** Sort the values in list ***
   sort(its, itf);

   //*** Display the values in list ***
   cout << "Sorted   : ";
   for (int v : list)
      cout << setw(2) << v << ' ';
   cout << endl;
}
```

Activity 20.20

Start a new project called *TestArrayIterators* and implement the code given above.

Summary

- Use `difference()` to find the number of data elements between two iterators.
- Use `begin()` to create an iterator to the start of an array.
- Use `end()` to create an iterator to the address immediately after the end of an array.
- Use `next()` to return an address immediately after the given iterator's address or return an address a given number of locations in advance.
- Use `advance()` to change the contents of an iterator to the next element's address or to an address several data elements away.
- Use `prev()` to return an address immediately before the given iterator's address or to return an address a given number of locations in back from the given iterator.

Solutions

Activity 20.1

No solution required.

Activity 20.2

Modified code for *TestAlgorithmLibrary*:

Version 1:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

//*** Function prototypes ***
bool IsLessThan50(int v);
bool IsLessThan25(int v);

int main()
{
  vector<int> testdata(20, 0);

  //*** Store random values between 0 and 50 ***
  for (int c = 0; c < 20; c++)
    testdata[c] = rand() % 51;
  //*** Display the values stored ***
  for (int v : testdata)
    cout << v << " ";
  cout << endl;

  if (all_of(testdata.begin(), testdata.end(),
  ↳IsLessThan50))
    cout << "All values in testdata are less than
    ↳50\n";
  else
    cout << "Not all value in testdata are less
    ↳than 50\n";
}

bool IsLessThan50(int v)
{
  return (v < 50);
}

bool IsLessThan25(int v)
{
  return (v < 25);
}
```

The output is:

```
All values in testdata are less than 50
```

There's a possibility that your own test data may contain the value 50!

Version 2:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

//*** Function prototypes ***
bool IsLessThan50(int v);
bool IsLessThan25(int v);

int main()
{
  vector<int> testdata(20, 0);

  //*** Store random values between 0 and 50 ***
  for (int c = 0; c < 20; c++)
    testdata[c] = rand() % 51;
  //*** Display the values stored ***
  for (int v : testdata)
    cout << v << " ";
  cout << endl;

  if (all_of(testdata.begin(), testdata.end(),
  ↳IsLessThan25))
    cout << "All values in testdata are less than
    ↳25\n";
  else
    cout << "Not all value in testdata are less
    ↳than 25\n";
}

bool IsLessThan50(int v)
{
  return (v < 50);
}

bool IsLessThan25(int v)
{
  return (v < 25);
}
```

The output is:

```
Not all values in testdata are less than 25
```

Activity 20.3

Modified code for *TestAlgorithmLibrary*:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

//*** Function prototypes ***
bool IsLessThan50(int v);
bool IsLessThan25(int v);
bool IsEven(int v);

int main()
{
  vector<int> testdata(20, 0);

  //*** Store random values between 0 and 50 ***
  for (int c = 0; c < 20; c++)
    testdata[c] = rand() % 51;


  //*** Display the values stored ***
  for (int v : testdata)
    cout << v << " ";
  cout << endl;

  if (all_of(testdata.begin(), testdata.begin()+4,
  ↳IsEven))
    cout << "The first four values in testdata are
    ↳even\n";
  else
    cout << "At least one of the first four values
    ↳in testdata is not even\n";
}


bool IsLessThan50(int v)
{
  return (v < 50);
}

bool IsLessThan25(int v)
{
  return (v < 25);
}

bool IsEven(int v)
{
  return (v % 2 == 0);
}
```

Notice that the end position i-

s given as `testdata.begin()+4` and not `testdata.begin()+3` since the last address is not included in the section being examined.

Activity 20.4

Modified code for *TestAlgorithmLibrary*:

```
#include <iostream>
```

```
#include <vector>
#include <algorithm>
using namespace std;

//*** Function prototypes ***
bool IsLessThan50(int v);
bool IsLessThan25(int v);
bool IsEven(int v);

//*** Class definition ***
class IsLessThan
{

  public:
     int limit;
     IsLessThan(int lim) { limit = lim; }
     bool operator()(int v) const
     {
        return (v < limit);
     }
};


int main()
{
  vector<int> testdata(20, 0);

  //*** Store random values between 0 and 50 ***
  for (int c = 0; c < 20; c++)
     testdata[c] = rand() % 51;

  //*** Display the values stored ***
  for (int v : testdata)
     cout << v << " ";
  cout << endl;

  //*** Check for all values being less than 50 then
  ↳25 ***
  IsLessThan mylimit(50);
  for (int c = 1; c <= 2; c++)
  {
     if (all_of(testdata.begin(), testdata.end(),
     ↳mylimit))
        cout << "All values are less than " <<
        ↳mylimit.limit << endl;
     else
        cout << "Not all values are less than " <<
        ↳mylimit.limit << endl;
     mylimit.limit = 25;
  }
}


bool IsLessThan50(int v)
{
  return (v < 50);
}

bool IsLessThan25(int v)
{
  return (v < 25);
}

bool IsEven(int v)
{
  return (v % 2 == 0);
}
```

Activity 20.5

Modified code for *TestAlgorithmLibrary*:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

//*** Function prototypes ***
bool IsLessThan50(int v);
bool IsLessThan25(int v);
bool IsEven(int v);

//*** Class definition ***
class IsLessThan
{

  public:
     int limit;
     IsLessThan(int lim) { limit = lim; }
     bool operator()(int v) const
     {
        return (v < limit);
     }
};


int main()
{
  vector<int> testdata(20, 0);

  //*** Store random values between 0 and 50 ***
  for (int c = 0; c < 20; c++)
     testdata[c] = rand() % 51;

  //*** Display the values stored ***
  for (int v : testdata)
     cout << v << " ";
  cout << endl;

  //*** Perform even test on elements 0 to 3 ***
  if (any_of(testdata.begin(), testdata.begin() + 4,
  ↳IsEven))
     cout << "At least one of the first four values
     ↳is even\n";
  else
     cout << "The first four values are all odd\n";
  if (none_of(testdata.begin(), testdata.begin() +
  ↳4, IsEven))
     cout << "Not one of the first four values is
     ↳even\n";
  else
     cout << "At least one of the first four values
     ↳is even\n";
}


bool IsLessThan50(int v)
{
  return (v < 50);
}

bool IsLessThan25(int v)
{
  return (v < 25);
}

bool IsEven(int v)
{
  return (v % 2 == 0);
}
```

Activity 20.6

No solution required.

Activity 20.7

Modified code for *TestAlgorithmLibrary02*.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

bool IsEven(int v);

int main()
{
  vector<int> testdata(20, 0);

  //*** Store random values between 0 and 10 ***
  for (int c = 0; c < 20; c++)
     testdata[c] = rand() % 11;
  //*** Display the values stored ***
  for (int v : testdata)
     cout << v << "  ";
  cout << endl;
  //*** Display the number of even numbers ***
  cout << "There are " << count_if(testdata.begin(),
  ↳testdata.end(), IsEven) << " even numbers in the
  ↳vector\n";
}

bool IsEven(int v)
{
  return (v % 2 == 0);
}
```

Activity 20.8

No solution required.

Activity 20.9

Code for *TestAlgorithmLibrary04*:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iomanip>
using namespace std;

//*** Function prototypes ***
bool Near(int v1, int v2);

int main()
{
  vector<int> src{ 1,3,7,1,4,7,12,1,4,7};
  vector<int> wnt{ 1,4,7 };

  //*** Display src ***
  cout << "src : ";
  for (auto v : src)
    cout << setw(2) << v << ' ';
  cout << "\n\n\n";

  //*** Search for first 1,4,7 ***
  auto it = search(src.begin(), src.end(),
  ↳wnt.begin(), wnt.end());

  //*** Search for first close to 1,4,7 ***
  auto it2 = search(src.begin(), src.end(),
  ↳wnt.begin(), wnt.end(), Near);

  //*** Display values found ***
  cout << "Value found by find_end() (ver 1) : " <<
  ↳*it << endl;
  cout << "Value found by find_end() (ver 2) : " <<
  ↳*it2 << endl;
}


bool Near(int v1, int v2)
{
  return (abs(v1 - v2) <= 2);
}
```

Activity 20.10

Modified code for *TestAlgorithmLibrary04:*

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iomanip>
using namespace std;

//*** Function prototypes ***
bool Near(int v1, int v2);

int main()
{
  vector<int> src{ 1,3,7,1,4,7,12,1,4,7};
  vector<int> wnt{ 1,4,7 };

  //*** Display src ***
  cout << "src : ";
  for (auto v : src)
    cout << setw(2) << v << ' ';
  cout << "\n\n\n";

  //*** Search for first 1,4,7 ***
  auto it = search(src.begin(), src.end(),
  ↳wnt.begin(), wnt.end());
  //*** Search for first close to 1,4,7 ***
  auto it2 = search(src.begin(), src.end(),
  ↳wnt.begin(), wnt.end(), Near);
  //*** Search for last 1,4,7 ***
  auto it3 = find_end(src.begin(), src.end(),
  ↳wnt.begin(), wnt.end());
  //*** Search for last close to 1,4,7 ***
  auto it4 = find_end(src.begin(), src.end(),
  ↳wnt.begin(), wnt.end(), Near);

  //*** Display indexes of where is match was found
  ↳***
  cout << "Value found by search() (ver 1) in cell
  ↳: " << it - src.begin() << endl;
  cout << "Value found by search() (ver 2) in cell
  ↳: " << it2 - src.begin() << endl;
  cout << "Value found by find_end() (ver 1) in cell
  ↳: " << it3 - src.begin() << endl;
  cout << "Value found by find_end()(ver 2) in cell
  ↳: " << it4 - src.begin() << endl;
}

bool Near(int v1, int v2)
{
  return (abs(v1 - v2) <= 2);
}
```

Activity 20.11

No solution required.

Activity 20.12

Modified code for *TestAlgorithmLibrary05:*

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void DisplayOdd(int v);
void Cube(int& v);


int main()
{
  vector<int> testdata(20);

  //*** Store random values between 0 and 10 ***
  for (int c = 0; c < 20; c++)
    testdata[c] = rand() % 11;
  //*** Display the values stored ***
  for (int v : testdata)
    cout << v << "  ";
  cout << endl;
  //*** Display only odd values ***
  cout << "Odd numbers in testdata: ";
  for_each(testdata.begin(), testdata.end(),
  ↳DisplayOdd);
  cout << endl;
  //*** Cube first 5 values ***
  for_each_n(testdata.begin(), 5, Cube);
  //*** Display the values stored ***
  cout << "testdata after cubing : ";
  for (int v : testdata)
    cout << v << "  ";
  cout << endl;
}


void DisplayOdd(int v)
{
  if (v % 2 != 0)
    cout << v << "  ";
}

void Cube(int& v)
{
  v*=v*v;
}
```

Activity 20.13

No solution required.

Activity 20.14

Code for *TestGenerate*:

```
#include <vector>
#include <algorithm>
#include <iomanip>
using namespace std;

class Random
{
private:
  int min;
  int max;
public:
```

```
  Random(int v1, int v2)
  {
    min = v1;
    max = v2;
  }

  int operator()()
  {
    return(rand() % (max - min) + min);
  }
};


int main()
{
  vector<int> src(15);
  //*** Store random values between 10 and 21 ***
  generate(src.begin(), src.end(), Random(10, 21));
  //*** Display the values stored ***
  for (int v : src)
    cout << setw(2) << v << ' ';
  cout << endl;
}
```

Activity 20.15

A typical output would be

```
src (original)    : 2  8  5  1 10  5  9  9  3  5  6  6  2  8  2
src (shuffle)     : 5  5  9  8  2  1  8  5  2  6  2  6 10  3  9
src (unique)      : 5  9  2  8  5  2  6  2  6 10  3  9
```

Activity 20.16

Modified code for *TestAlgorithmLibrary07*:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
#include <random>
using namespace std;

//*** Function prototypes ***
bool Approx(int v1, int v2);

int main()
{
  vector<double> src(15);
  vector<double> dest(15);

  //*** Store random value between 1 and 10 ***
  for (auto& v : src)
    v = rand() % 10 + 1;
  //*** Display original contents ***
  cout << "src (original) : ";
  for (auto v : src)
    cout << setw(2) << v << "  ";
  cout << endl;

  //*** Shuffle src ***
  shuffle(src.begin(), src.end(),
  default_random_engine(time(0)));
  //*** Display shuffled contents ***
  cout << "src (shuffle)  : ";
  for (auto v : src)
    cout << setw(2) << v << "  ";
  cout << endl;

  //*** Remove adjacent values with diff <= 1 ***
  auto iter = unique(src.begin(), src.end(),
  Approx);
  //*** Display final contents ***
  cout << "src (unique)   : ";
  for (auto it = src.begin(); it < iter; it++)
    cout << setw(2) << *it << "  ";
  cout << endl;

  //*** //*** Shuffle src ***
  shuffle(src.begin(), iter,
  default_random_engine(time(0)));
  //*** Display shuffled contents ***
  cout << "\n\nsrc (2nd shuffle)  : ";
  for (auto it = src.begin(); it < iter; it++)
    cout << setw(2) << *it << "  ";
  cout << endl;

  //*** Remove adjacent duplicate values  ***
  auto iter2 = unique_copy(src.begin(), iter,
  dest.begin(), Approx);
  //*** Display dest contents ***
  cout << "dest (unique_copy) : ";
  for (auto it = dest.begin(); it < iter2; it++)
    cout << setw(2) << *it << "  ";
  cout << endl;
}


bool Approx(int v1, int v2)
{
  return (abs(v1-v2) <= 1);
}
```

A typical output:

```
src (original) : 2  8  5  1 10  5  9  9  3  5  6  6  2  8  2
src (shuffle)  : 2  6  5  2  5 10  8  5  3  9  9  6  8  2  1
src (unique)   : 2  6  2  5 10  8  5  3  9  6  8  2

src (2nd shuffle)    : 2  8  8  6  2 10  5  6  9  3  5  2
dest (unique_copy) : 2  8  6  2 10  5  9  3  5  2
```

Activity 20.17

Modified code for *TestAlgorithmLibrary07*:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
#include <random>
using namespace std;

//*** Function prototypes ***
bool Approx(int v1, int v2);

int main()
{
  vector<double> src(15);
  vector<double> dest(15);

  //*** Store random value between 1 and 10 ***
  for (auto& v : src)
    v = rand() % 10 + 1;
  //*** Display original contents ***
  cout << "src (original) : ";
  for (auto v : src)
    cout << setw(2) << v << "  ";
  cout << endl;

  //*** Shuffle src ***
  shuffle(src.begin(), src.end(),
  default_random_engine(time(0)));
  //*** Display shuffled contents ***
  cout << "src (shuffle)  : ";
  for (auto v : src)
    cout << setw(2) << v << "  ";
  cout << endl;

  //*** Remove adjacent values with diff <= 1 ***
  auto iter = unique(src.begin(), src.end(),
  Approx);
  //*** Display final contents ***
  cout << "src (unique)   : ";
  for (auto it = src.begin(); it < iter; it++)
    cout << setw(2) << *it << "  ";
  cout << endl;

  //*** //*** Shuffle src ***
  shuffle(src.begin(), iter,
  default_random_engine(time(0)));
  //*** Display shuffled contents ***
  cout << "\n\nsrc (2nd shuffle)  : ";
  for (auto it = src.begin(); it < iter; it++)
    cout << setw(2) << *it << "  ";
  cout << endl;

  //*** Remove adjacent duplicate values  ***
  auto iter2 = unique_copy(src.begin(), iter,
  dest.begin(), Approx);
  //*** Display dest contents ***
  cout << "dest (unique_copy) : ";
  for (auto it = dest.begin(); it < iter2; it++)
    cout << setw(2) << *it << "  ";
  cout << endl;
```

```
  //*** Sort dest ***
  sort(dest.begin(), iter2);
  //*** Display sorted dest ***
  cout << "dest (sort)        : ";
  for (auto it = dest.begin(); it < iter2; it++)
    cout << setw(2) << *it << "  ";
  cout << endl;
}


bool Approx(int v1, int v2)
{
  return (abs(v1-v2) <= 1);
}
```

A typical output:

```
src (original) : 2  8  5  1 10  5  9  9  3  5  6  6  2  8  2
src (shuffle)  : 8 10  5  8  1  9  6  2  5  2  6  9  3  2  5
src (unique)   : 8 10  5  8  1  9  6  2  5  2  6  9  3  5

src (2nd shuffle)    : 10  1  5  5  8  2  6  3  9  8  9  6  5  2
dest (unique_copy): 10  1  5  8  2  6  3  9  6  2
dest (sort)          :  1  2  2  3  5  6  6  8  9 10
```

Activity 20.18

Code for *TestAlgorithmLibrary08*:

```
#include <iostream>
#include <algorithm>
#include<vector>
using namespace std;

//*** Function prototypes ***
bool IsLess(char v1, char v2);

int main()
{
  vector<char>letters{'A','B','C','a','b','c','D'};

  //*** Display letters ***
  cout << "Unsorted : ";
  for (auto v : letters)
    cout << v << "  ";
  cout << endl;
  //*** sort letters ***
  sort(letters.begin(), letters.end(), IsLess);
  cout << "Sorted   : ";
  for (auto v : letters)
    cout << v << "  ";
  cout << endl;
}

bool IsLess(char v1, char v2)
{
  return (tolower(v1) < tolower(v2));
}
```

Activity 20.19

Modified code for *TestAlgorithmLibrary08*:

```
#include <iostream>
#include <algorithm>
#include<vector>
using namespace std;

//*** Function prototypes ***
bool IsLess(char v1, char v2);

int main()
{
  vector<char>letters{'A','B','C','a','b','c','D'};

  //*** Display letters ***
  cout << "Unsorted : ";
  for (auto v : letters)
    cout << v << "  ";
  cout << endl;
  //*** sort letters ***
  sort(letters.begin(), letters.end(), IsLess);
  cout << "Sorted   : ";
  for (auto v : letters)
  cout << v << "  ";
  cout << endl;
  bool found = binary_search(letters.begin(),
  ↳letters.end(), 'd', IsLess);
  if (found)
    cout << "Value found\n";
  else
    cout << "Value not found\n";
}

bool IsLess(char v1, char v2)
{
  return (tolower(v1) < tolower(v2));
}
```

Activity 20.20

No solution required.

21

The string Class

In this Chapter:

- ❑ Creating a *string* Object
- ❑ *string* Operators
- ❑ Comparing Strings
- ❑ Sub-Strings
- ❑ Accessing Individual Characters
- ❑ Standard *string* Methods
- ❑ Container-Related *string* Methods
- ❑ Iterator-Related *string* Methods
- ❑ The *stringstream* Class

The string Class

Introduction

Up to this point we have been using `char` arrays (known as **C-style** strings) to handle string values. Although this has been the standard method of string handling in C and even in many C++ programs, there is now a better and simpler way – using objects of C++'s `string` class.

In reality the `string` class is just another type of container and because of this, we'll meet many of the same methods we saw in the previous chapter. But unlike the `array` and `vector` classes, the `string` class is designed to contain only characters.

Creating a *string* Object

The `string` class is defined in the `std` namespace, so we need to use the term

```
std::string
```

unless our program contains the line

```
using namespace std;
```

and then we can, as usual, reduce the name to

```
string
```

So, if we wanted to create a variable in which we intended to store a name, we might use one of the following lines

```
string name;
string name{};
```

Each of the above statements will call the `string` class's constructor which will store an empty string in *name*.

To assign an initial value we can use any of the normal formats:

```
string name1{"Susan Foreman"};
string name2 = "Barbara Wright";
string teachers ("TBD");
string science_teacher;
```

The string constants *"Susan Foreman"*, *"Barbara Wright"* and *"TBD"* in the above examples will be held as `const` arrays of type `char`, and the initialisation (which will be using the `string` class's constructor) will convert that value specified to a string object during the assignment process.

We can also create a `string` object from a substring. For example,

```
string staff ("Elizabeth Heron",5,4);
```

would store *"beth"* in *staff*.

The numeric parameters above define the starting index in the `char` array (the string's index starts at zero) and the number of characters to be copied. These are 5 and 4 respectively.

Specifying a size that is too large will copy the remainder of the `char` array's contents

```
string staff ("Elizabeth Heron",5,40);// colleague contains beth
                                      //Heron
```

and if only a single numeric value is given, this is taken as the number of characters to be copied starting at index zero.

```
string staff ("Elizabeth Heron",5);//staff contains Eliza
```

We can also create a string object whose initial value is another string object:

```
string history_teacher {name2};
```

Or we can copy only part of the initialising string object:

```
string history_teacher {name2,0,7}; //7 chars starting at first
```

The above would give *history_teacher* the value

```
Barbara
```

since we are copying 7 characters starting at the first character.

Although it is not possible to initialise a string object with a `char` value – for example the line

```
string ones {'1'};   //INVALID
```

won't compile – it is possible to copy a `char` value multiple times when creating a string object:

```
string ones(3,'1'); //Sets ones to "111"
```

Since this is a constructor call, we need to use parentheses – braces won't work.

string-Style Constants

Although normal double-quoted string constants such as `"Hello world"` are stored as C-style strings, we can turn this into a `string` class constant by adding the letter `s` after the closing double quotes as in the term

```
"Hello world"s
```

I/O Operations

Both the `<<` and `>>` operators can be used with the `string` class but, in order to do that, we must add the line

```
#include <string>
```

to the start of our program.

With this line added to our program we can then use statements such as

```
cout << teachers << endl;
```

or

```
cin >> name1;
```

getline()

When using `cin` with a `string` object, we have the same problem as we did when inputting C-style strings in earlier chapters: any whitespace character in the input stream will be assumed to mark the end of the input.

Luckily, the `getline()` function is also defined for the `string` class, meaning we can use a line such as

```
getline(cin,name);
```

to ensure that all characters entered up to the newline character (created by pressing the *Enter* key) are read into the `string` object.

The `getline()` function accepts a third parameter which allows us to specify a different terminating character for the input. For example, we could write

```
getline(cin,name,'.');
```

and have the value stored in *name* be all the characters up to but excluding a full stop.

Using the line given above, if we were to enter

```
ABC.DEF <enter>
```

then the string *name* would contain the value *"ABC"*.

The program in FIG-21.1 makes use of most of the `string` class features we've covered so far.

FIG-21.1

Using *string* Objects

```
#include <iostream>
#include <string>

using namespace std;

void main()
{
   string name1{};
   string name2{};
   string history_teacher;
   string science_teacher;
   string teachers;
   //*** Get names ***
   cout << "Enter name of male teacher: ";
   getline(cin, name1);
   cout << "Enter name of female teacher: ";
   getline(cin, name2);
   //*** Display names ***
   cout << "Male teacher: " << name1 << endl;
   cout << "Female teacher: " << name2 << endl;
}
```

Activity 21.1

Create a new project called *StringClass01* and implement the code given above.

Enter the names as *Ian Chesterton* and *Barbara Wright*.

Interrogating a String

We can discover various details about a `string` object such as the number of characters it contains using the following `string` methods.

size() and *length()*

These methods are identical in operation, with both returning the number of bytes occupied by the characters within the string. Typical lines might be:

```
cout << name1.size() << endl;
cout << name1.length() << endl;
```

The value returned does not include the terminating null character which is stored as part of the string (just like a C-style string).

empty()

This method returns *true* if the string contains no characters, otherwise *false* is returned. Example,

```
if (name1.empty())
   ...
```

Of course, we can achieve the same result using `size()` or `length()` :

```
if (name2.size() == 0)
   ...
```

Since `string` constants (with the suffix `s`) are also fully-fledged `string` objects, we can also use any of the methods and operators defined within the `string` class on those constants. For example it is quite valid to create the line

```
cout << "Hello world"s.size() << endl;
```

But, of course, it is not appropriate to attempt to modify the contents of a constant.

string Class Operators

Several standard C++ operators have been redefined within the `string` class to allow us to easily perform operations such as assignment, concatenation and comparison. These are likely to be sufficient for most simple string-handling features within a program.

These string operators are described below.

Assignment Operator (=)

After the variable has been created, a new value can be assigned to the string variable using the normal assignment operator:

```
name1 = "Ian Chesterton";  // C-style string converted to
                           // string object and stored in name1

teachers = "Robert Smith"s;//string object constant stored
                           //in teachers

science_teacher = name1;   // string in object name1 copied to
                           // science_teacher
```

A traditional C-style string stored within a `char` array can also be assigned to a string object. For example,

```
char txt[]{ "Coalhill" };
name1 = txt;
```

The Concatenation Operator (+)

We can use the concatenation operator to join strings:

```
string teachers = name1 + name2;
```

However, we must remember that the + operator is defined within the `string` class and therefore designed to work when at least one of the two operands is a `string` object, meaning statements such as

```
teachers = "Mr. " + name1;
teachers = name2 + ", Miss";
```

are acceptable.

However, a line such as

```
teachers = "Ian Chesterton" + "Barbara Wright";
```

will fail since neither operand is a `string` class object. But, we can always create an anonymous string object to solve this problem using the line

```
teachers = string("Ian Chesterton") + "Barbara Wright";
```

or simply by adding an `s` at the end of one of the strings:

```
teachers = "Ian Chesterton"s + "Barbara Wright";
```

Multiple `+` operators can be used in a single expression:

```
teachers = "Mr. " + name1 + "Miss " + name2;
```

We can also use the concatenation operator to add a single character.

```
teachers = name1 + '&' + name2;
```

The Append-Assignment Operator (+=)

The `+=` operator is also defined for the `string` class allowing new characters to be added to the end of an existing string. There are options to use another `string` object, a C-style string or a character:

```
name1 += name2;   //Add string object's value
name1 += "Mr. "s; //Add string constant's value
name1 += "Mr. ";  //Add C-style string constant
name1 += 'H';     //Add character
```

The C-style string can also be stored in a `char` array:

```
char txt[]{ "Coalhill" };
name1 += txt;        //Add contents of C-style char array
```

Activity 21.2

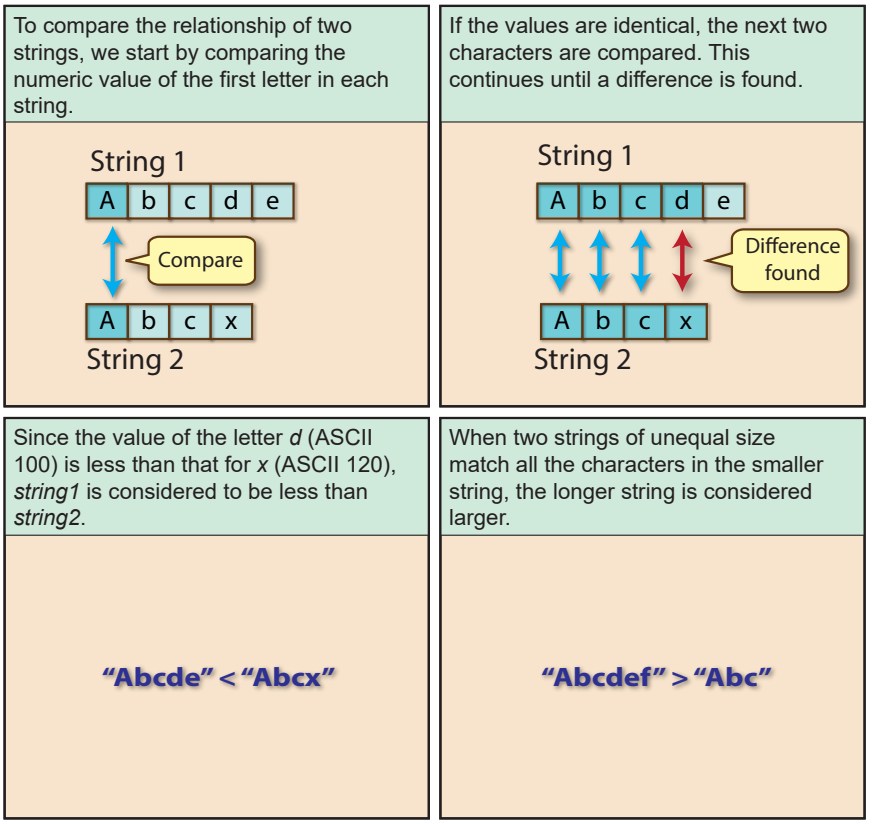
Modify *StringClass01* and use the + operator to add the title "*Mr.* " *name1* and use += to add "*, Miss*" to the end of *name2*.

The Relational Operators

When deciding if two strings are equal, the numeric values of each character in the corresponding positions of the strings are compared. The first position where there is not an exact match determines the relationship between the two strings (see FIG-21.2).

FIG-21.2

How Strings are Compared



Like C-style strings, *string* objects store a string with a terminating null character. But, because that null character is always handled automatically, it is not shown here.

All of the relational operators are defined for the `string` class, so Boolean expressions such as

```
name1 == name2
name1 != name2
name1 < name2
name2 > "W"
name1 <= teachers
name2 >= ""
```

are all valid.

Activity 21.3

Modify *StringClass01* to check if the final value in *name1* is greater than the final value in *name2*.

Display the value of *name1* along with either " *is greater than* " or " *is less than or equal to* " as appropriate followed by the value of *name2*.

string Class Comparisons

compare()

Although we would probably use the relational operators for most string comparisons. we also have the `string` class's `compare()` method which allows us to compare the

selected string object with another `string` object, a substring, or a C-style string.

The method returns an integer value. The value returned is positive (usually +1) if the `string` object calling the method is greater than the argument, zero if the two strings are exactly identical, and a negative value (usually -1) if the argument string is greater than the calling string.

For example, the following tests compare strings against the contents of *name1*:

```
char other[]{"Foreman"};
int result;
result = name1.compare(name2);   //string object
result = name1.compare(other);   //C-style string variable
result = name1.compare("Susan"); //C-style string constant
```

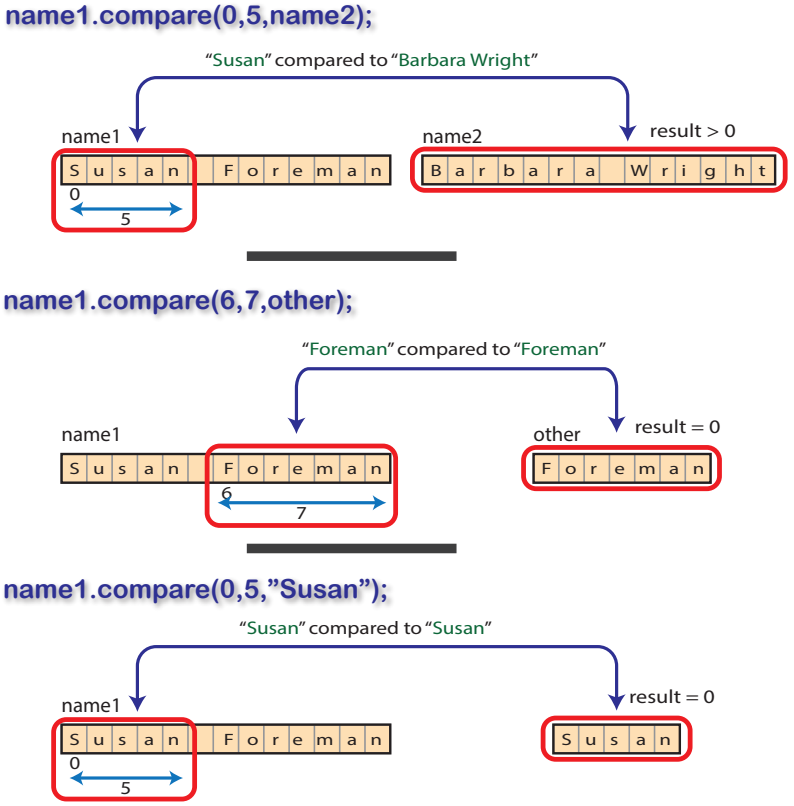
If necessary, we can compare only a substring from our object (by specifying the start position and length of the substring) with the second string as shown in the following examples:

```
result = name1.compare(0,5,name2);  //string object
result = name1.compare(6,7,other);  //C-style string variable
result = name1.compare(0,5,"Susan"); //C-style string constant
```

The comparisons made by the above three statements are visualised in FIG-21.3.

FIG-21.3

Comparing a String to a Sub-String



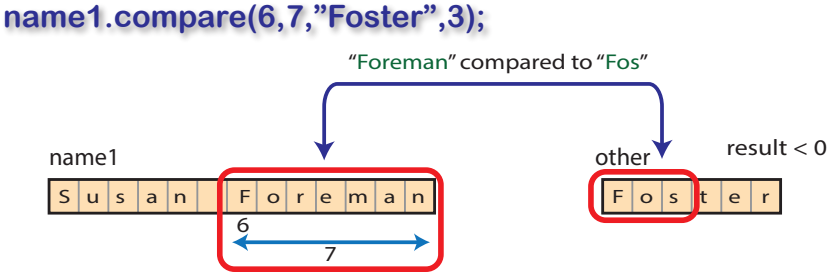
We can even select a substring from the string parameter by adding a final parameter which specifies the number of characters to be used in the comparison. Note, however, that the substring always starts from the first character. For example, the line

```
result = name1.compare(6,7,"Foster,3);
```

compares the substring shown in FIG-21.4.

FIG-21.4

Comparing Two Sub-Strings



Activity 21.4

Modify *StringClass01* removing the current `if` structure and replacing it with a test to check if the fifth, sixth and seventh characters in *name1* are equal to "*Ian*". Display the message "*First name is Ian*" or "*First name is not Ian*" as appropriate.

substr()

We can extract a substring from our `string` object by using the `substr()` method. For example, in the code

```
string name2 = "Barbara Wright";
string forename;
forename = name2.substr(0, 7);
cout << forename << endl;
```

the output would be

```
Barbara
```

Note that the method requires the start position of the substring and the number of characters to be copied.

If the value given for the first parameter is not a valid position, a runtime error will occur.

If the second parameter is larger than the number of characters remaining, the substring returned consists of all the characters from the starting point to the end of the string. The same effect can be achieved by omitting the second parameter altogether. So the two lines,

```
cout << name2.substr(8,99);
cout << name2.substr(8);
```

both achieve the same result:

```
Wright
```

Using `substr()` and standard relational operators, we can achieve the same results as we did when using `compare()` to compare the current string to a substring. For example, the line

```
if (name1.substr(4,3) == "Ian")
```

achieves the same result as

```
if (name1.compare(4,3,"Ian") == 0)
```

Activity 21.5

Modify *StringClass01*, replacing the call to `compare()` with the equivalent call to `substr()`.

Accessing Characters

[ ]

We can access the individual characters in a `string` object by enclosing the index of the required character (index values start at zero) in brackets. For example, the lines

```
string text{ "ABCDEF" };
cout << text[0] << endl;
```

would output

```
A
```

If we want to access each character in a string individually we have the option of using a standard `for` loop:

```
for (int c = 0; c < text.size(); c++)
   cout << text[c] << endl;
```

or a range-based `for` loop:

```
for(auto v: text)
   cout << v << endl;
```

We can also use a reference variable to allow us to modify the characters in a string:

```
for(auto &v: text)
   v = tolower(v);
```

at()

Rather than use brackets to specify the position of the character to be accessed we can use the `at()` method. For example, the lines

```
cout << text[2] << endl;
```

and

```
cout << text.at(2) << endl;
```

have exactly the same effect.

The `at()` method checks that its parameter's value is within the length of the string. If the parameter is outside that range, an exception of type `out_of_range` is thrown. An invalid index within brackets, on the other hand, creates a runtime error.

front()

This method returns a reference to the first character in the string. For example,

```
string text{ "ABCDEF" };
cout << text.front() << endl;
```

displays

```
A
```

back()

This method returns a reference to the last character in the string. For example,

```
cout << text.back() << endl;
```

displays

```
F
```

Having accessed a specific position within a string, we are free to change the character at that position. For example, any of the following statements have the same effect, changing the first letter in *text* from A to J:

```
text[0] = 'J';
text.at(0) = 'J';
text.front() = 'J';
```

Changing the final letter in *text* from *F* to *P* can be done by any of the following statements:

```
text[text.length()-1] = 'P';
text.at(text.length()-1) = 'P';
text.back() = 'P';
```

Iterators

A standard `string` class iterator can be created with a statement such as:

```
std::string::iterator iter; //Iterator named iter
```

And assuming our program contains the line `using namespace std;` we can reduce the declaration to

```
string::iterator iter;
```

A `string` class has the same iterator-accessing methods we saw previously in the `array` and `vector` classes:

begin()	Returns an iterator reference to the first character in the string.
end()	Returns an iterator reference to the address immediately following the last character in the string.
cbegin()	Returns a `const` iterator referencing the first character of the string.
cend()	Returns a `const` iterator reference to the address immediately following the last character in the string.
rbegin()	Returns a reverse iterator reference to the address of the last character in the string.
rend()	Returns a reverse iterator reference to the address before the first character in the string.
crbegin()	Returns a `const` reverse iterator reference to the address of the last character in the string.
crend()	Returns a `const` reverse iterator reference to the address before the first character in the string.

The program in FIG-21.5 uses a `string` iterator to change all the characters in a string to uppercase.

FIG-21.5

Using an Iterator on a string Object

```
#include <iostream>
#include <string>
using namespace std;

int main()
{

   string name2{ "Barbara Wright" };
   string::iterator iter;
   //*** Change string to uppercase ***
   for (iter = name2.begin(); iter < name2.end(); iter++)
      *iter = toupper(*iter);
   //*** Display string ***
   cout << name2 << endl;
}
```

Activity 21.6

Start a new project called *StringClass02*, and implement the code given above, checking that the contents of *name2* are changed to uppercase.

Of course, if we were to use `cbegin()` and `cend()` in the above code, we would be presented with a compilation error since we are attempting to change the value referenced by a `const` iterator:

```
string name2{ "Barbara Wright" };
string::const_iterator iter;  //Different iterator type
for (iter = name2.cbegin(); iter < name2.cend(); iter++)
   *iter = toupper(*iter);    //ERROR
cout << name2 << endl;
```

Search and Replace Operations

We can search the current string for a specific character of substring using one of several search-related methods. In the examples below we'll assume the following declarations:

```
string text{ "ABCDEFABCDEF" };
string lookfor{ "CDE" };
char  letter{'C'};
int foundat;
```

find()



string::npos represents the largest possible unsigned positive value. When stored in a signed variable it will display as -1.

This method allows us to search for a character or substring in all or part of the current string and returns the position at which the match was found (the first character in the string is at position zero). If no match is found, the named constant `string::npos` is returned – a value which is returned by all find-related methods in the `string` class in response to an unsuccessful search.



All comparisons are case-sensitive, so a search for 'e' would return ***string::npos***.

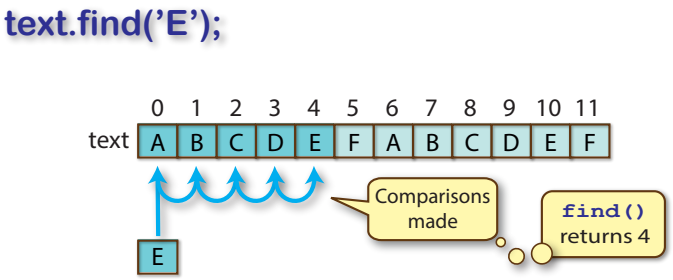
The simplest option is to specify a single character to be searched for, as in the lines

```
foundat = text.find('E');
foundat = text.find(letter);
foundat = text.find('X');
```

These lines would set *foundat* to 4, 2, and -1 (`string::npos` stored in the signed integer, *foundat*) respectively. Note that when the searched for item occurs more than once in the current string, it is the position of the first occurrence that is returned. FIG-21.6 shows how the method searches for 'E'.

FIG-21.6

Searching a String



When searching for a character, we have the option to specify at which position in the current string we should start the search. For example, in the lines

```
foundat = text.find('E',3);
foundat = text.find(letter,3);
```

the search starts at position 3 (the fourth character), meaning that only the substring "DEFABCDEF" is examined. This would result in *foundat* being set to 4 and 8 respectively in the two examples above. Notice that the returned value still counts from the start of the whole string.

Activity 21.7

Returning to *StringClass01*, modify the code, replacing the `if` statement with code which displays the number of times the letter *e* occurs in *name1*.

Rather than search for a single character, we can also search for a matching substring using either another `string` object or a C-style string as shown in the lines below:

```
foundat = text.find(lookfor);
foundat = text.find("EF");
```

These examples set *foundat* to 2 and 4 respectively.

Again we can specify where in the current string the search should begin as in the lines:

```
foundat = text.find(lookfor,4);
foundat = text.find("EF",4);
```

In these cases, *foundat* would be set to 8 and 4 respectively.

When searching for a value specified in the form of a C-style string, we can also search for part of that value. For example, in the line

```
foundat = text.find("FAX",3,2);
```

the second parameter, 3, specifies where to start the search in *text* and the third parameter, 2, specifies how many characters (starting at the first) in "FAX" are to be searched for (only "FA"). Hence, *foundat* would be assigned the value 5.

Activity 21.8

Modify *StringClass01*, replacing the count of e's in *name1* to a count of *"ar"*s in *name2*.

rfind()

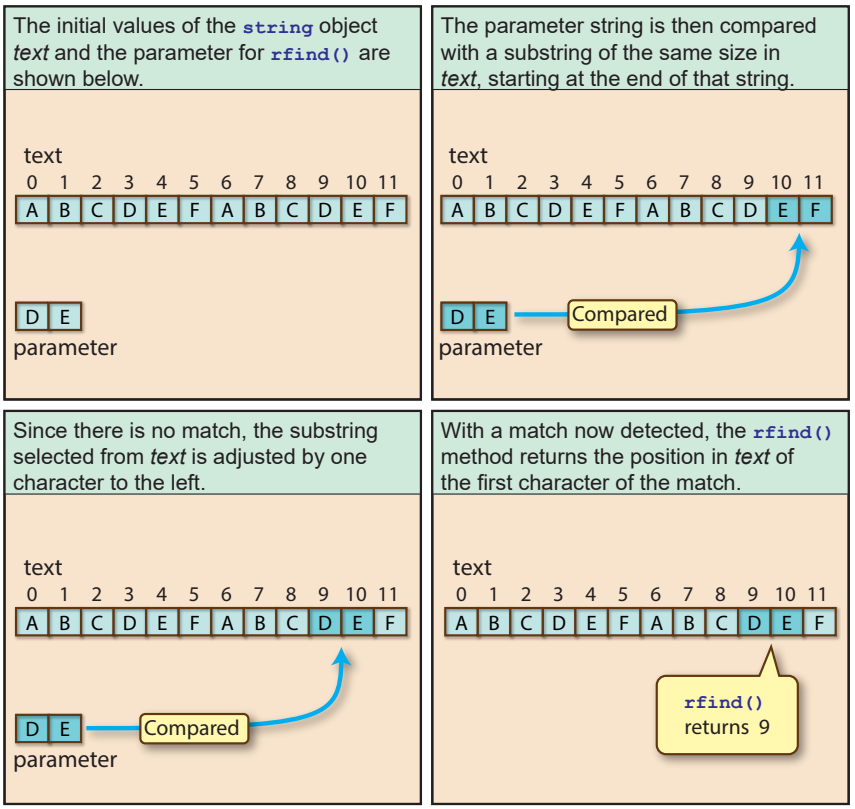
Whereas `find()` searches from the start of the current string, `rfind()` starts from the end of the string and so finds the last occurrence of the value being searched for rather than the first. Otherwise, it has exactly the same options when it comes to parameters. In the lines below,

```
foundat = text.rfind("FAX", 11, 2);
foundat = text.rfind("DE");
foundat = text.rfind('C');
foundat = text.rfind(lookfor);
```

findat has the values 5, 9, 8 and 8 respectively. FIG-21.7 shows how the search for "DE" is performed.

FIG-21.7

Searching Backwards Through a String



Activity 21.9

Modify *StringClass01*, replacing the count of *ar*'s with a display of the position of the last 'a' in *name2*.

find_first_of()

When using `find()` and `rfind()` a match is only found when the complete string being searched for occurs within the current string object. The method `find_first_of()`, however, will return a found position when ANY of the characters in the searched-for string are encountered.

It is the position of the first occurrence of a matching character which is returned, or

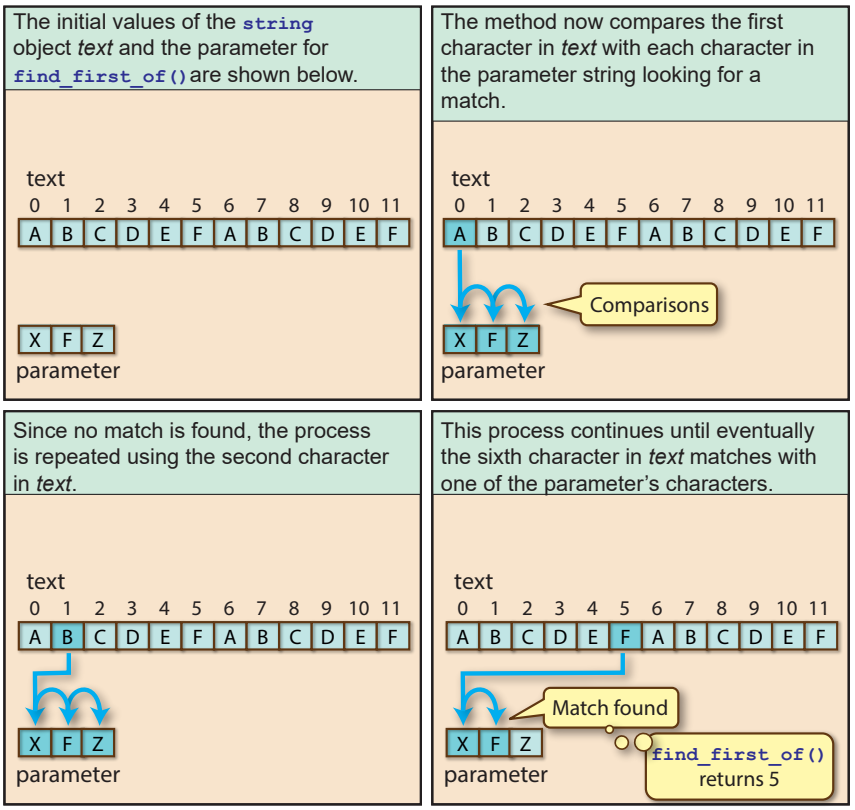
`string::npos` if no match is found. For example, the line

```
foundat = text.find_first_of("XFZ");
```

will set *foundat* to 5 since 'F' which is in the string "XFZ" can be found first at position 5 in the *text* `string` object. FIG-21.8 shows the steps executed by the above example.

FIG-21.8

How *find_first_of()* Works



This method has the same parameter options as `find()` and `rfind()`, so all of the following are valid calls:

```
//*** Look for 'Z' in "ABCDEF" ***
foundat = text.find_first_of("XFZ",6,1);          //foundat = -1
//*** Look for any of "CDE" in "DEFABCDEF" ***
foundat = text.find_first_of(lookfor,3);          //foundat = 3
//*** Look for 'C' in "ABCDEFABCDEF"
foundat = text.find_first_of('C');                //foundat = 2
```

Activity 21.10

Modify *StringClass01*, replacing the display of the position of the last 'a' in *name2* with the position of the first character in *name1* matching any character in the string ". ▽ ," (full stop, space, comma) and also displaying which of the characters has been found.

Allow for none of the characters being found with the message "There are no full stops, spaces or commas in " along with the contents of *name1*.

find_last_of()

The `find_last_of()` searches for any character in a specified string working from right to left and so finds the last occurrence of a specified character. All the standard parameter options shown in `find_first_of()` are available. For example,

```
foundat = "DEDE"s.find_last_of("DA");
```

would set *foundat* to 2 since the letter 'D' from the string "DA", is found at position 2 (the third character).

find_first_not_of()

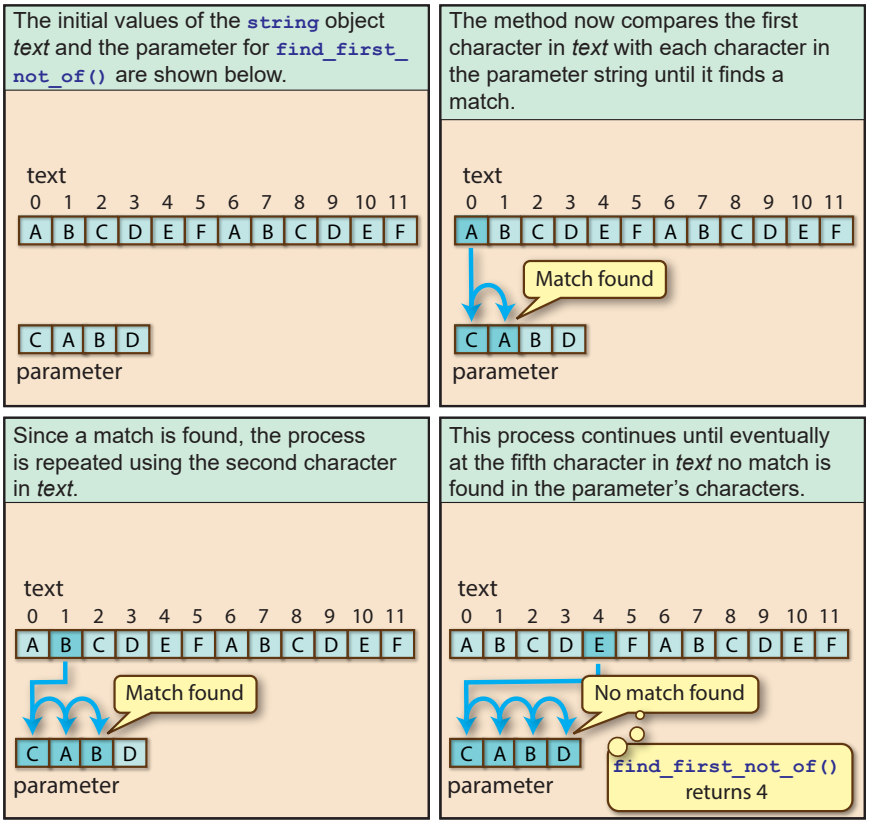
To find within the current string the first character NOT in a specified string we can use the `find_first_not_of()` method. For example,

```
foundat = text.find_first_not_of("CABD");   //foundat = 4
```

the first character in *text* which is not in the specified string is E at position 4. The process involved is shown in FIG-21.9.

FIG-21.9

How *find_first_not_of()* Works



All the parameter options seen before in `find_first_of()` are available here.

find_last_not_of()

To find within the current string the last character NOT in a specified string we can use the `find_last_not_of()` method. For example,

```
foundat = text.find_last_not_of("CABD");   //foundat = 10
```

the last character in *text* which is not in the specified string is E at position 10.

All the parameter options seen before in `find_last_of()` are available here.

replace()

Part of the existing string can be deleted and replaced by new contents using the `replace()` method. All variations of this method require the start position and the number of characters being deleted. The simplest option replaces the specified section with a completely new string. So, starting with the lines

```
string sentence { "Never fire a gun at a mirror" };
string extra{ "throw" };
```

executing the statement

```
sentence.replace(6, 4, extra);
```

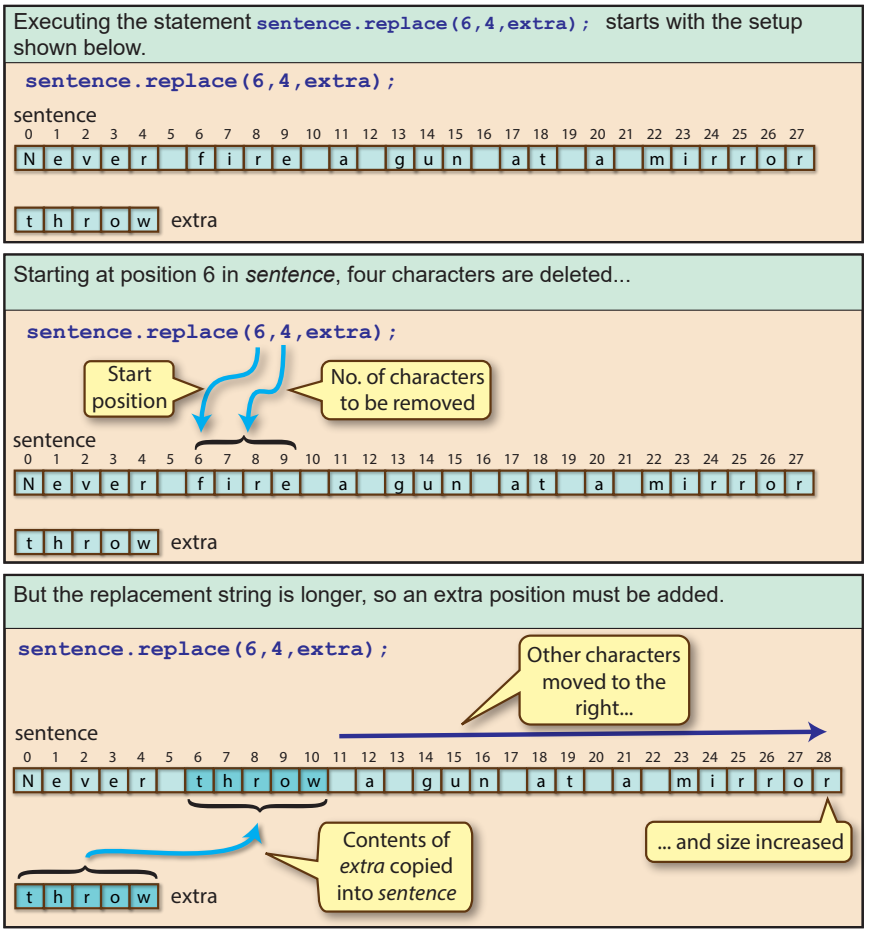
changes the contents of *sentence* to

```
Never throw a gun at a mirror
```

The steps involved are shown in FIG-21.10.

FIG-21.10

How *replace()* Works



Note that if the start position is invalid, an `out_of_range` exception is thrown. If the number of characters to be removed is larger than the number of characters available for removal, all characters to the end of the string are removed.

The new text can also be given as a C-style string as in the line

```
sentence.replace(13,3,"laser");
```

which gives us the result

```
Never fire a laser at a mirror
```

having removed three characters from the original string starting at position 13 ("*gun*") and then replacing those characters with the string "*laser*".

Alternatively, we can copy just part of the second string into the original by specifying the start position in the second string and the number of characters to copy:

```
sentence.replace(13,3,"laser",2,3);
```

This gives the following result:

```
Never fire a ser at a mirror
```

If the part being copied is from the start of the second string, then only the number of characters to be copied needs to be specified:

```
sentence.replace(13,3,"laser",2)
```

gives:

```
Never fire a la at a mirror
```

Another option is to replace the selected text with a single character repeated a number of times. For example,

```
sentence.replace(13,3,5,'-')
```

gives us:

```
Never fire a ----- at a mirror
```

Rather than specify the start position and number of characters to be removed from the original string, we can use iterators to specify the start and end positions of the substring being deleted. For example,

```
sentence.replace(sentence.begin(),sentence.begin()+5,"Don\'t");
```

gives us

```
Don't fire a gun at a mirror
```

Note that the second iterator value points to the character AFTER the last one to be deleted.

The above iterator option can also be used with the third parameter being given as another `string` object as in the line

```
sentence.replace(sentence.begin()+6,sentence.begin()+10,extra);
```

Iterators can also be used when using only a substring of the third argument (which must in this case be a C-style string) and when we want to replace text with a repeated single character.

Activity 21.11

Modify *StringClass01*, removing the code which follows the display of the contents of *name1* and *name2*. Now use the `replace()` method to change the term *Miss* in *name2* to *Ms*.

Activity 21.12

Modify *StringClass01*, removing the code which follows the display of the contents of *name1* and *name2*. Now use the `find()` and `replace()` methods to change all a's in *name2* to e's.

swap()

The function `swap()` (which we met previously in the `array` and `vector` container classes) is designed to swap the contents of two `string` objects, For example the line

```
swap(name1,name2);
```

will swap the contents of objects *name1* and *name2*.

Adding to a String

We already have the + and += operators which allow us to add contents to the start or end of a string. However, there are two methods in the `string` class which also allow us to add new characters to an existing string.

One method allows the new characters to be added either to the end of the existing string while the second method allows us to insert additional text at any position within the string.

append()

The `append()` method's simplest option mimics the `+=` allowing us to add a specified string to the end of the current string object. For example,



C-style strings or string objects can be used as the parameter in most options.

```
string name2{"Barbara Wright"};
name2.append(", Miss");
cout << name2 << endl;
```

would output

```
Barbara Wright, Miss
```

The function becomes more useful if only part of the new string is to be copied into the existing one. In this situation, we can specify the number of characters to be copied:

```
name2.append(", Miss",3); //Adds ", M"
```

which would assign *name2* the new value

```
Barbara Wright, M
```

If the section of string to be added does not start at the first character, the start position and number of the characters to be copied can be specified:

```
name2.append(", Miss",3,2);   //Adds "is"
```

This would assign *name2* the value

```
Barbara Wrightis
```

Yet another option is to append a specified character a given number of times:

```
name2.append(4,'X');
```

results in *name2* having the value

```
Barbara WrightXXXX
```

If the string to be appended is part of one held in a `string` object, then we have to use the iterator options to access the substring required For example,

```
string name1 = "Ian Chesterton";
string name2 = "Barbara Wright";
name.append(name2.begin()+12,name2.end());
cout << name << endl;
```

produces the output

```
Ian Chesterton Wright
```

insert()

When the text to be added to an existing `string` object is not to be placed at the end of the current contents, we can use the `insert()` method.

As usual there are several forms of the method available. The simplest takes the insert point and the text to be added:

```
string command = "Klaatu nikto";
command.insert(6," barada");
cout << command << endl;
```

This code outputs

```
Klaatu barada nikto
```

The next option is to insert only part of the given string, specifying the position of the first character to be copied and the number of the characters to be copied:

```
command.insert(6," barada",2,3);
```

This sets *command* to

```
Klaatuara nikto
```

Like most `string` methods, `insert()` can handle both C-style strings and `string` class objects. For the most part, there is no difference in the result created, but when using `insert()`, there is one option where the result depends on the type of string used as a parameter. In the line

```
command.insert(6," barada",3);
```

the use of a C-type string results in the parameters being interpreted as: insert the first three characters of *" barada"* into *command*'s existing string starting at position 6 giving us:

```
Klaatu ba nikto
```

But when we use a `string` object,

```
command.insert(6, " barada"s,3);
```

the method starts at position 3 (the fourth character) of the `string` parameter and inserts the remainder of the string:

```
Klaaturada nikto
```

Another option is to insert a specified character a number of times as exampled in the line

```
command.insert(6,4,'@');
```

which produces the result

```
Klaatu@@@@ nikto
```

Activity 21.13

Start a new project called *StringClass03* which reads in a string value from the keyboard and inserts an 'X' after every 'e'.

The modified string should be displayed.

Deleting from a String

clear()

This method removes all the characters currently held, creating an empty string. For example,

```
name1.clear();
```

would result in *name1* containing zero characters.

Of course, the simplest way to clear a string is simply to write

```
name1 = "";
```

erase()

The `erase()` method can erase one or more characters from a string. So, starting with the code

```
string name1 {"Ian Chesterton"};
```

the line

```
name1.erase();
```

will delete every character in *name1*.

```
name1.erase(3);
```

will delete all characters from position 3 onwards giving:

```
Ian
```

We can also supply a starting position and the number of characters to be deleted:

```
name1.erase(3,8);
```

gives us

```
Ianton
```

Using an iterator value, we can delete a single character:

```
name1.erase(name.begin()+4);
```

This gives us:

```
Ian hesterton
```

Iterators can also be used to define the substring to be deleted:

```
name.erase(name.begin()+8, name.end()-3);
```

name now contains the string

```
Ian Cheston
```

Activity 21.14

Modify *StringClass03*, removing the code to insert 'X's and modify the program to remove all lowercase vowels. The modified string should be displayed.

Converting to a C-Style String

We can convert the contents of a string object to a C-style string using one of two methods from the `string` class.

c_str()

We can convert the `string` object's contents to a `const char*` type string using the `c_str()` method. For example,

```
const char* teacher1{ name1.c_str() };
```

Notice the requirement to define *teacher1* as a `const char*` type and not simply `char*`.

Of course, if *teacher1* was defined earlier, then we would have to use `strcpy_s()` to assign the value returned by `c_str()`:

```
char pupil[20];
strcpy_s(pupil,20,name1.c_str());
```

data()

The `data()` method also returns a C-style string from the object, but this time the string is not returned as a constant so we can assign it to a standard `char` pointer:

```
char* teacher1{ name1.data() };
```

Since it's not a constant, we can modify the contents of the C-style string. For example, we could make the first character lowercase using the line

```
teacher1[0] = tolower(teacher1[0]);
```

As a general rule, we'll only want to convert our `string` class object to a C-style string if it has to be passed to a function which expects a `char*` type parameter.

Container-Related Methods

When looking at the `array` and `vector` container classes in a previous chapter we saw that they had several methods in common that related to investigating or modifying the contents of the container.

The `string` class is really a specialised container since it holds a sequence of characters and so it should come as no surprise that it too has some container-related operations. However, in the `string` class, most of the following methods have easier alternatives for achieving the same effect.

assign()

The `assign()` method allows us to assign a new value to the string. There are three variations on this operation. In the simplest, we specify the new string to be assigned.

This can be another `string` object or a C-style string. The following are valid examples:

```
name1.assign("Ian Chesterton");  //C-style string
name1.assign("Ian Chesterton"s); //anonymous string object
science_teacher.assign(name1);   //named string object
```

When using a C-style string, we can assign just part of the string. For example,

```
name1.assign("Ian Chesterton",3);   //Copies Ian
```

assigns the first three characters of "*Ian Chesterton*" to *name1*.

We have even more control when using a string object parameter since we can specify the start position and number of characters to be copied as in the line

```
name1.assign("Ian Chesterton"s,4,10);  //Copies Chesterton
```

which assigns 10 characters starting at the fifth character of "*Ian Chesterton*" to *name1*.

Another option is to fill the string with a specified number of copies of a given character. For example,

```
name1.assign(3,'?'); //Copies ???
```

We can also use iterators to determine which characters are copied. For example, using the lines

```
string name1{"Ian Chesterton"};
science_teacher.assign(name1.rbegin()+1, name1.rend(-3));
```

assigns the value "*otretsehC*" to *science_teacher*.

pop_back()

This method removes the last character from the string. Hence, the code

```
string text{ "ABCDEF" };
text.pop_back();
cout << text() << endl;
```

displays

```
ABCDE
```

having removed the final character from the string *text*.

push_back()

This method adds a specified character to the end of the string. For example,

```
string text{ "ABCDEF" };
text.push_back('Z');
cout << text() << endl;
```

displays

```
ABCDEFZ
```

having added the *Z* to the string *text*.

Note that, generally, it's easier to use the `+=` operator.

capacity()

This method returns the current capacity of a string. That is the number of bytes available to hold the characters of the string. Although this will usually be the same value as that given by `size()` that may not always be the case. Sometimes the string is assigned more bytes than that required.

Typically, the capacity will be given as one byte longer than the length since internally the `string` object still retains a null character at the end of the stored sequence of characters.

reserve()

Should we wish to change the capacity of a string we can do this using the `reserve()` method. This can increase or decrease the capacity but C++ will ensure that the capacity remains large enough to contain the currently held sequence of characters. For example,

```
name.reserve(20); //Reserve space for 20 characters in total
```

In some ways this may seem like a pointless option since `string` objects will automatically change capacity to cope with any changes to the size of the string stored within it. However, when a string changes size, it may have to release the memory space currently being used to hold the string and find a larger area elsewhere in memory.

If we use the `reserve()` method we can help ensure that the space will be large enough to accommodate any possible value that may be assigned to the string and therefore eliminate the time delay caused by having to allocate a different space.

shrink_to_fit()

This method is a request to reduce the capacity of the string so that it is an exact match for the number of characters currently held within the string. In FIG-21.11 below, the `string` object *name2* has its size and capacity displayed, before and after a change of contents and after a call to `shrink_to_fit()`.

FIG-21.11

Using *size()* and *capacity()*

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string name2{ "Barbara Wright" };
   //*** Display original contents, size and capacity ***
   cout << "name2: " << name2 << endl;
   cout << "Size: " << name2.size() << endl;
   cout << "Capacity: " << name2.capacity() << endl;
   //*** Add text to end of string                 ***
   //*** then display contents, size and capacity ***
   name2 += ", Miss";
   cout << "name2: " << name2 << endl;
   cout << "Size: " << name2.size() << endl;
   cout << "Capacity: " << name2.capacity() << endl;
   //*** Shrink the display size and capacity ***
   name2.shrink_to_fit();
   cout << "Size: " << name2.size() << endl;
   cout << "Capacity: " << name2.capacity() << endl;
}
```

When run on a PC using Visual C++ 2019, the output was:

```
name2: Barbara Wright
Size: 14
Capacity: 15
name2: Barbara Wright, Miss
Size: 20
Capacity: 31
Size: 20
Capacity: 31
```

Note that the call to `shrink_to_fit()` had no effect on the capacity of the string. This highlights the fact that the method is only a request – not a command!

max_size()

This method returns the maximum size (in bytes) that a string can become. This will be limited by the way in which strings are implemented by the compiler or by the amount of memory available. Example,

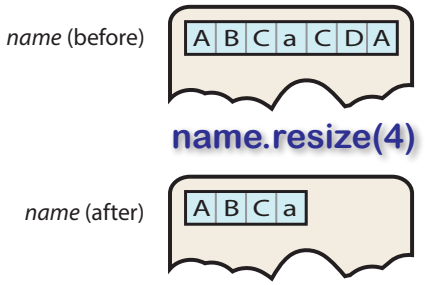
```
cout << name.max_size() << " bytes\n";
```

resize()

This method resizes a string. If the new size is less than the string's current length, the characters to the right of the string will be deleted (see FIG-21.12).

FIG-21.12

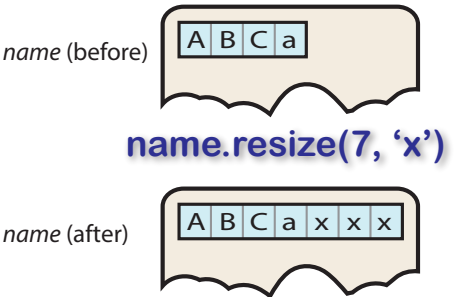
Using *resize()* to Shorten a String



It is also possible to increase the size of the string, in which case the character used to fill the new positions can be specified (see FIG-21.13).

FIG-21.13

Using *resize()* to Lengthen a String



If no character is specified when expanding a string, the space character is used.

Iterator Functions and Strings

In the previous chapter we looked at many functions which operate on containers and make use of iterators as parameters. Because the `string` class is nothing more than a container that is restricted to holding `char` elements, `string` objects can make use of those same functions.

For example, many chapters ago we came across the `tolower()` function which returns the lowercase version of a given character. But there is no function to convert a complete string to lowercase. However, this is a problem that is easily overcome using the `transform()` function declared in the *algorithm* header file. All that is required is the line



If the program does not contain the line `using namespace std;` then the last parameter would be `std::tolower`

```
transform(txt.begin(), txt.end(), txt.begin(), tolower);
```

where *txt* is the string object to be converted.

Activity 21.15

Start a new project called *TestStringManipulation* and write a program to count every vowel in a string and to display each word in the string (this assumes the string contains a sentence).

Converting to/from a Numeric Value

to_string()

This function is not a member of the `string` class. Rather it is a stand-alone function which takes any type of numeric value and returns the equivalent-valued `string` object. For example,

```
string pi {to_string(3.14159265);
```

sets *pi* to the string "3.14159265". Your compiler may round the value to fewer digits.

stoi(), stol(), stoll(), stoul(), stoull(), stof(), stod(), stold()

If we want to reverse the process, converting a string to a numeric, then we must select one of the functions listed above. Which function we choose depends on the type of numeric we want to end up with (`int`, `long`, `long long`, `unsigned long`, `unsigned long long`, `float`, `double`, `long double`). The function ending gives us a clue to the type returned by the function, for example, `stoi()` returns an `int`.

The lines

```
string snum{ "123" };
int num = stoi(snum);
cout << num << endl;
```

would output

```
123
```

The functions also accept C-style parameters since these are automatically converted to `string` objects by the `string` class constructor.

If the string starts with a non-numeric character (other than '-' or '+') an `invalid_argument` exception is thrown.

If non-numeric characters follow the initial numeric characters, as in the string "*456ABC*", the function will return a numeric value equivalent to the initial numeric digits. In this example, that would be 456.

But wait! There's more!

These functions also accept optional second and third parameters. The second parameter, if used, is a pointer to a value of type `size_t` and is used to return the index of the first non-numeric character in the string being converted. For example, using the code

```
string snum{ "456ABC" };
size_t first_non_num{};
int num = stoi(snum, &first_non_num);
```

first_non_num would be set to 3 since position three in the string "456ABC" contains the first non-numeric character.

If the string is fully numeric, the second parameter returns the index of the terminating null character.

If we do not wish to have the value returned, but still require to enter a value for this parameter, any null value can be used (0, `null`, or `nullptr`) in which case no value is returned.

The third and final parameter specifies the number base of the value within the string. The assumed value for this is 10, we are free to specify any base value we wish. For example, if *snum* contained the string "FAFA" we could specify that we were dealing with a hexadecimal value by using the line

```
int num = stoi(snum,0,16);
```

which would assign *num* the value 64250.

Of course, when using a number base other than 10, what is considered to be a numeric character is automatically adjusted. For example, in the line

```
int num = stoi("1239", &first_non_num, 8);
```

9 would be considered a non-numeric character since octal values can only contain the digits 0 to 7.

Other *String* Classes

The *string_view* Class

A function (or method) that takes a `string` class parameter may find it has to make a copy of that string. For example, we might write a function to return the number of spaces in a string, giving the function heading

```
int CountSpaces(string s)
```

This will result in a local copy of the actual parameter's value in a `string` object called *s* within the function.

Perhaps we could do better by using a reference variable:

```
int CountSpaces(string& s)
```

Now the parameter's local variable *s* will contain only a reference to the actual parameter.

But the above statement is only true if the actual parameter is a string object. If we had used a C-style string as the actual parameter, say

```
int count = CountSpaces("Ask no questions; be told no lies.");
```

then the compiler will convert this to a local string object before executing the

function with the actual parameter containing the address of the newly created object.

The `string_view` class (defined within the `std` namespace) creates an object which contains a pointer to the start of any string value (a `string` object's text or C-style text) and a count of the number of characters in the string. If we create our original function's parameter to be of this type,

```
int CountSpaces(string_view s)
```

then the function's local variable will only ever contain a pointer to the actual parameter, be that a string object or a C-style string. This guarantees that no copy need be made of the actual parameter – the cause of a significant delay if the string is a long one.

However, when using a `string_view` object, the contents of the original object are treated as a constant and its string cannot be changed.

The *wstring* Class

Whereas the `string` class is based on the `char` type for storing its sequence of characters, the `wstring` class uses the `wchar_t` type (defined within the `std` namespace) for storing its characters. The same methods as those in the `string` class also exist within the `wstring` class, but with names that are prefixed by a `w` (for example, `wreplace()`).

There are also options to create Unicode 16 strings (`std::u16string` class) and Unicode 32 strings (`std::u32string` class).

Summary

- Most of the operations that follow work with parameters that are either `string` class objects or C-style strings.
- C-style string constants are enclosed in double quotes (e.g. `"Hello world"`).
- `string` class constants are defined by adding the character 's' after the closing double quote (e.g. `"Hello world"s`).

Operators

- `=` to assign a string to the current `string` object.
- `+` to join two strings (at least one must be a `string` class object).
- `+=` to join a string to the current string object.
- `==`, `!=`, `<`, `<=`, `>`, `>=` to compare two strings (at least one must be a `string` class object).
- `[ ]` to access a character in a string.

I/O Operations

- `>>` to input a string using `cin` (whitespace terminated).
- `<<` to output a string using `cout`.

- `getline()` to input a string (*Enter* or specified character terminated).

Iterators

- `begin()` returns address of the first character.
- `end()` returns the address immediately after the last character.
- `cbegin()` returns the address of the first character as the address of a constant.
- `cend()` returns the address immediately after the last character as the address of a constant.
- `rbegin()` returns the address of the final character in the string for use with a reverse iterator.
- `rend()` returns the address immediately before the first character.
- `crbegin()` returns the address of the last character as a `const` iterator address.
- `crend()` returns the address immediately before the first character as a `const` iterator address.

Methods

- `compare()` returns integer based on comparison with other string.
- `substr()` returns a substring.
- `at()` returns character at a specified position.
- `front()` returns first character.
- `back()` returns last character.
- `find()` returns start position of first occurrence of specified substring.
- `rfind()` returns start position of last occurrence of specified substring.
- `find_first_of()` returns position of first character matching any character in specified substring.
- `find_last_of()` returns position of last character matching any character in specified substring.
- `find_first_not_of()` returns position of first character not matching any character in specified substring.
- `find_last_not_of()` returns position of last character not matching any character in specified substring.
- `replace()` replaces a specified number of characters starting at a given position with substring or repeating character.
- `append()` adds a substring to the end of the current string.
- `insert()` inserts a given substring at a specified position.

- `clear()` empties the string.
- `erase()` deletes all or part of a string.
- `c_str()` returns a C-style string constant equivalent to the current string object's contents.
- `data()` returns a C-style string variable equivalent to the current string object's contents.

Container-Related Methods

- `assign()` assigns a given value to the string object.
- `pop_back()` removes the last character in the string.
- `push_back()` adds a character to the end of the string.
- `capacity()` returns the space available for the string's contents.
- `reserve()` sets the space available for the string's contents.
- `shrink_to_fit()` adjusts the space available to match the current contents of the string.
- `max_size()` returns the maximum space available to the string.
- `resize()` changes the number of characters in the string. This may delete existing characters or add new ones.

Related Functions

- `tostring()` returns the string equivalent of a numeric value.
- `stoi()`, `stol()`, `stoll()`, `stoul()`, `stoull()`, `stof()`, `stod()`, `stold()` converts a string containing numeric characters to an `int`, `long`, `long long`, `unsigned long`, etc.

Container Related Functions

- All of the functions discussed in Chapter 20 such as `fill()`, `generate()` and `shuffle()` can be applied to string objects.

Solutions

Activity 21.1

No solution required.

Activity 21.2

Modified code for *StringClass01*:

```
#include <iostream>
#include <string>
using namespace std;

void main()
{
  string name1{};
  string name2{};
  string history_teacher;
  string science_teacher;
  string teachers;
  //*** Get names ***
  cout << "Enter name of male teacher : ";
  getline(cin, name1);
  cout << "Enter name of female teacher : ";
  getline(cin, name2);
  //*** Add titles ***
  name1 = "Mr. " + name1;
  name2 += ", Miss";
  //*** Display names ***
  cout << "Male teacher : " << name1 << endl;
  cout << "Female teacher : " << name2 << endl;
}
```

Activity 21.3

Modified code for *StringClass01*:

```
#include <iostream>
#include <string>
using namespace std;

void main()
{
  string name1{};
  string name2{};
  string history_teacher;
  string science_teacher;
  string teachers;
  //*** Get names ***
  cout << "Enter name of male teacher : ";
  getline(cin, name1);
  cout << "Enter name of female teacher : ";
  getline(cin, name2);
  //*** Add titles ***
  name1 = "Mr. " + name1;
  name2 += ", Miss";
  //*** Display names ***
  cout << "Male teacher : " << name1 << endl;
  cout << "Female teacher : " << name2 << endl;
  //*** Display result of comparison ***
  if (name1 > name2)
    cout << name1 << " is greater than " << name2
    ↳<< endl;
  else
    cout << name1 << " is less than or equal to "
    ↳<< name2 << endl;
}
```

Activity 21.4

Modified code for *StringClass01*:

```
#include <iostream>
#include <string>
using namespace std;

void main()
{
  string name1{};
  string name2{};
  string history_teacher;
  string science_teacher;
  string teachers;
  //*** Get names ***
  cout << "Enter name of male teacher : ";
  getline(cin, name1);
  cout << "Enter name of female teacher : ";
  getline(cin, name2);
  //*** Add titles ***
  name1 = "Mr. " + name1;
  name2 += ", Miss";
  //*** Display names ***
  cout << "Male teacher : " << name1 << endl;
  cout << "Female teacher : " << name2 << endl;
  //*** Display result of comparison ***
  if (name1.compare(4, 3, "Ian") == 0)
    cout << "First name is Ian\n";
  else
    cout << "First name is not Ian\n";
}
```

Activity 21.5

Modified code for *StringClass01*:

```
#include <iostream>
#include <string>
using namespace std;

void main()
{
  string name1{};
  string name2{};
  string history_teacher;
  string science_teacher;
  string teachers;
  //*** Get names ***
  cout << "Enter name of male teacher : ";
  getline(cin, name1);
  cout << "Enter name of female teacher : ";
  getline(cin, name2);
  //*** Add titles ***
  name1 = "Mr. " + name1;
  name2 += ", Miss";
  //*** Display names ***
  cout << "Male teacher : " << name1 << endl;
  cout << "Female teacher : " << name2 << endl;
  //*** Display result of comparison ***
  if (name1.substr(4, 3) == "Ian")
    cout << "First name is Ian\n";
  else
    cout << "First name is not Ian\n";
}
```

Activity 21.6

No solution required.

Activity 21.7

Modified code for *StringClass01*:

```
#include <iostream>
#include <string>
using namespace std;

void main()
{
  string name1{};
  string name2{};
  string history_teacher;
  string science_teacher;
  string teachers;
  //*** Get names ***
  cout << "Enter name of male teacher : ";
  getline(cin, name1);
  cout << "Enter name of female teacher : ";
  getline(cin, name2);
  //*** Add titles ***
  name1 = "Mr. " + name1;
  name2 += ", Miss";
  //*** Display names ***
  cout << "Male teacher : " << name1 << endl;
  cout << "Female teacher : " << name2 << endl;
  //*** Display number of e's in name1 ***
  int ecount{};    //Num of e's
  int lastfound{-1}; //Position last e found
  while ((lastfound = name1.find('e', lastfound+1))!=
  ↳string::npos)
    ecount++;
  cout << "The letter \'e\' appears " << ecount <<
  ↳" times in "<< name1 << endl;
}
```

Activity 21.8

Modified code for *StringClass01*:

```
#include <iostream>
#include <string>
using namespace std;

void main()
{
   string name1{};
   string name2{};
   string history_teacher;
   string science_teacher;
   string teachers;
   //*** Get names ***
   cout << "Enter name of male teacher : ";
   getline(cin, name1);
   cout << "Enter name of female teacher : ";
   getline(cin, name2);
   //*** Add titles ***
   name1 = "Mr. " + name1;
   name2 += ", Miss";
   //*** Display names ***
   cout << "Male teacher : " << name1 << endl;
   cout << "Female teacher : " << name2 << endl;
   //*** Display number of "ar"'s in name2 ***
   int ecount{};      //Num of ar's
   int lastfound{-1}; //Position of last ar found
   while ((lastfound = name2.find("ar", lastfound+1 ))
   ↳!= string::npos)
      ecount++;
   cout << "The sequence \"ar\" appears " << ecount
   ↳<<" times in "<< name2 << endl;
}
```

Activity 21.9

Modified code for *StringClass01*:

```
#include <iostream>
#include <string>
using namespace std;

void main()
{
   string name1{};
   string name2{};
   string history_teacher;
   string science_teacher;
   string teachers;
   //*** Get names ***
   cout << "Enter name of male teacher : ";
   getline(cin, name1);
   cout << "Enter name of female teacher : ";
   getline(cin, name2);
   //*** Add titles ***
   name1 = "Mr. " + name1;
   name2 += ", Miss";
   //*** Display names ***
   cout << "Male teacher : " << name1 << endl;
   cout << "Female teacher : " << name2 << endl;
   //*** Display position of last 'a' in name2 ***
   if (int post{}; (post = name2.rfind('a')) !=
   ↳string::npos)
      cout << "The last \'a\' in " << name2 <<
      ↳" appears at position " << post << endl;
   else
      cout << "There is no \'a\' in " << name2 <<
      ↳endl;
}
```

Activity 21.10

Modified code for *StringClass01*:

```
#include <iostream>
#include <string>
using namespace std;

void main()
{
   string name1{};
   string name2{};
   string history_teacher;
   string science_teacher;
   string teachers;
   //*** Get names ***
   cout << "Enter name of male teacher : ";
   getline(cin, name1);
   cout << "Enter name of female teacher : ";
   getline(cin, name2);
   //*** Add titles ***
   name1 = "Mr. " + name1;
   name2 += ", Miss";
   //*** Display names ***
   cout << "Male teacher : " << name1 << endl;
   cout << "Female teacher : " << name2 << endl;
   //*** Find full stop, space or comma in name1 ***
   int foundat = name1.find_first_of(". ,");
   if (foundat != string::npos)
      cout << name1[foundat] << " found at position "
      ↳<< foundat << " in " << name1 << endl;
   else
      cout << "There are no full stops, spaces or
      ↳commas in " << name1 << endl;
}
```

Activity 21.11

Modified code for *StringClass01*:

```
#include <iostream>
#include <string>
using namespace std;

void main()
{
   string name1{};
   string name2{};
   string history_teacher;
   string science_teacher;
   string teachers;
   //*** Get names ***
   cout << "Enter name of male teacher : ";
   getline(cin, name1);
   cout << "Enter name of female teacher : ";
   getline(cin, name2);
   //*** Add titles ***
   name1 = "Mr. " + name1;
   name2 += ", Miss";
   //*** Display names ***
   cout << "Male teacher : " << name1 << endl;
   cout << "Female teacher : " << name2 << endl;
   //*** Replace Miss with Ms ***
   name2.replace(name2.end() - 4, name2.end(), "Ms");
   cout << "New value for female teacher : " << name2
   ↳<< endl;
}
```

Activity 21.12

Modified code for *StringClass01*:

```
#include <iostream>
#include <string>
using namespace std;

void main()
{
   string name1{};
   string name2{};
   string history_teacher;
   string science_teacher;
   string teachers;
   //*** Get names ***
   cout << "Enter name of male teacher : ";
   getline(cin, name1);
   cout << "Enter name of female teacher : ";
   getline(cin, name2);
   //*** Add titles ***
   name1 = "Mr. " + name1;
   name2 += ", Miss";
   //*** Display names ***
   cout << "Male teacher : " << name1 << endl;
   cout << "Female teacher : " << name2 << endl;
   //*** Replace a's with e's in name2 ***
   int lastfound{ -1 }; //Position of last a found
   while ((lastfound = name2.find("a", lastfound +
   ↳1)) != string::npos)
      name2.replace(lastfound,1,"e");
   cout << name2 << endl;
}
```

Activity 21.13

Code for *StringClass03*:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string txt{};
   //*** Get string value ***
   cout << "Enter string : ";
   getline(cin, txt);
   //*** Find e's and insert X's ***
   int lastfound{ -1 }; //Position of last e found
   while ((lastfound = txt.find("e", lastfound + 1))
   ↳!= string::npos)
      txt.insert(txt.begin() + lastfound+1, 'X');
   //*** Display modified string ***
   cout << txt << endl;
}
```

Activity 21.14

Modified code for *StringClass03*:

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string txt{};
   //*** Get string value ***
   cout << "Enter string : ";
   getline(cin, txt);
   //*** Find e's and insert X's ***
   int lastfound{ 0 };  //Position of last e found
   //*** Erase vowels ***
   while ((lastfound = txt.find_first_of("aeiou",
   ↳lastfound)) != string::npos)
      txt.erase(lastfound, 1);
   //*** Display modified string ***
   cout << txt << endl;
}
```

Note that *lastfound* now starts at zero and that each iteration starts at the position where the previous match was found. We start at the same position because, having deleted a vowel, that position now contains a new character.

Activity 21.15

Code for *TestStringManipulation*:

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;

bool IsVowel(char ch);
void DisplayEachWord(string s);

int main()
{
   string txt{ "Open the pod bay door, Hal." };
   //*** Count the number of vowels ***
   int count = count_if(txt.begin(), txt.end(),
   ↳IsVowel);
   cout << "String is : " << txt << endl;
   cout << "There are " << count << " vowels\n";
   //*** Display each word separately ***
   DisplayEachWord(txt);
}

bool IsVowel(char ch)
{
   static char vowels[]{ "aeiou" };
   //*** Make sure ch is lowercase ***
   char tmp = tolower(ch);
   //*** If ch matches a letter in vowels ***
   //*** return true                      ***
   for (char v : vowels)
      if (v == tmp)
         return true;
   //*** Not a vowel, return false ***
   return false;
}

void DisplayEachWord(string s)
{
   string word;
   cout << "\n\nWords in sentence :\n";
   //*** WHILE text remaining DO***
   while (s.size() > 0)
   {
      //*** Find word separator character ***
      int end = s.find_first_of(" ,.?!");
      //*** Extract the word from sting ***
      word = s.substr(0, end);
      //*** If at least one char in word, display it
      ↳***
      if (word != "")
         cout << word << endl;
      //*** Remove word and it separator from the
      ↳text ***
      s = s.substr(end+1, s.size() - end);
   }
}
```

22

Class Relationships - 1

In this Chapter:

- ❑ Inheritance
- ❑ Constructors and Destructors in Derived Classes
- ❑ Hiding Inherited Methods
- ❑ Overloading Friend Functions
- ❑ Protected Properties
- ❑ Using *using* to Change Access Settings
- ❑ Using *destroy* to Delete Inherited Methods
- ❑ Static Features in Derived Classes
- ❑ Setting the Inheritance Access Specifier
- ❑ Multiple Inheritance

Relationships Between Classes

Introduction

We all have relatives. It's from these relatives that we inherit our basic characteristics like hair colour and facial structure. Most manufactured objects also have an ancestry. Our flat screen TV has inherited features from the old colour CRT TVs, colour TVs inherited from black and white TVs, and TVs themselves inherited features from radios.

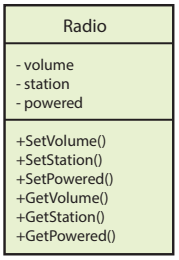
This approach to building on what has gone before is one of the main goals of object-oriented programming. By adopting this approach, rather than write a new class from scratch we can just inherit all the features from an existing class then add to or modify those features to create the new class needed by our latest project.

Class Relationships

To see how this relationship between a new, derived class and an existing class is represented, let's have a look at a possible class diagram for radio (see FIG-22.1).

FIG-22.1

Radio Class



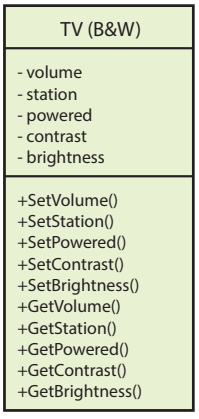
The attributes are:

volume	the volume to which the sound is set
station	the frequency to which the radio is tuned
powered	the radio's on/off status

A black and white TV has similar properties to the radio, but in addition there will be brightness and contrast settings (see FIG-22.2).

FIG-22.2

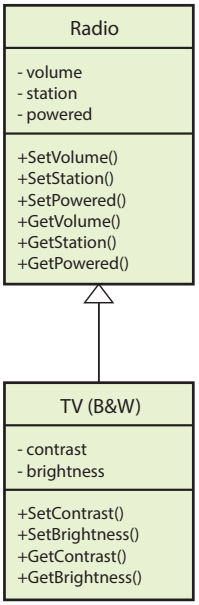
TV Class



We can show the relationship between the two classes in a diagram by linking them together to show that the TV is derived from the radio (see FIG-22.3).

FIG-22.3

Designing a Derived Class



Notice that the derived class (*TV(B&W)*) shows only the features that are new to it. The assumption is that *TV(B&W)* has automatically inherited all the properties of its ancestor (*Radio*).

In programming, when we create a new class based on an existing class, we introduce a few new terms to object-oriented design and programming. These are:

base class this is the class from which a new class is derived. In the example above, *Radio* is the base class for the *TV(B&W)* class. The terms **superclass** and **parent class** are also used.

derived class this is the new class which is created from the base class. The terms **subclass** and **child class** are also used.

There are no restrictions on the number of generations of classes that can be created. For example, we could create a derived class using *TV(B&W)* as the base class. This new class would then inherit all the features of the *TV(B&W)* class (including those which the *TV(B&W)* class itself inherited from the *Radio* class).

Activity 22.1

Draw a class diagram for a new class, *ColourTV*, which is derived from the *TV(B&W)* class.

The new class should have the following attributes:

volume, station, powered, contrast, brightness, coloursaturation

with operations:

SetVolume(), SetStation(), SetPowered(), GetVolume(), GetStation(), GetPowered(), SetContrast(), SetBrightness(), GetContrast(), GetBrightness(), SetColour(), GetColour()

We can think of the diagram as akin to a family tree (albeit one in which children have a single parent) where offspring inherit all the features of its parent. And, like a family tree, a parent class can have more than one child class.

Activity 22.2

Add a new class to your diagram called *StereoRadio*. The new class has the same features as the *Radio* class with the additional attribute, *balance*, and new operations *SetBalance()* and *GetBalance()*.

This feature of object-oriented design, in which a new class automatically contains the features of an existing class is known as **inheritance**.

Implementing Derived Classes

C++ allows us to implement derived classes in our code. We'll see how this is done by working through an example.

Previously, we created a *Time* class which we used to store a time in hours, minutes and seconds. Let's assume we now want to include milliseconds.

One option, of course, is simply to edit the code for the original *Time* class, but this may not be possible for several reasons:

- The original class is still required for other projects and should not be changed.
- The original class was written by another programmer and we are not familiar with the code's logic.
- We only have a compiled version of the original class and so cannot edit its source code.

If we cannot modify the *Time* class, the new class will have to be derived from it. We'll call the new class *MoreTime* giving it a new attribute, *millisecond*, and two new operations *SetMillisecond()* and *GetMillisecond()*.

The original *Time* class has been coded in project *ChronoLib* in the files *ChronoLib.h* and *Time.cpp* within the solution *MyLibraries* and held in a namespace called *dsk*.

We have three choices about where to create our new class:

- Add the code for *MoreTime* to the same project by extending *ChronoLib.h* and creating a new file, *MoreTime.cpp*.
- Create the new class in a different project in the *MyLibraries* solution.
- Create the new class in a new project in a new solution.

Which of these options we choose depends on several factors. However, if the source code for the original class is not available to the application programmer, then the new class will have to be placed in a new project in a new solution. It is that option we'll use here.

Our new project/solution will be called *TestMoreTime* and the code for its *Source.cpp* file will start with the lines

```
#include<iostream>
#include<ChronoLib.h>
using namespace std;
```

Also, we'll need to tell the compiler and linker where to find the original *ChronoLib* files. This means we need to select **Project>TestMoreTime Properties** from the main menu and adjust the following:



If you have stored your library elsewhere, then these values will be different.

C/C++ > General > Additional Include Directories :
E:\Hands On C++\Programs\MyLibraries\ChronoLib

Linker > General > Additional Library Directories :
E:\Hands On C++\Programs\MyLibraries\Debug

Linker > Input > Additional Dependencies :
ChronoLib.lib

The next line of code shows that the new class is derived from the *Time* class by beginning with the line

```
class MoreTime : public dsk::Time
```

The term `public` is used to show that the features of the *Time* class are to have the same access classification (`public` or `private`) in the derived class as they would have in an application. That is to say, the methods coded within the *MoreTime* class have access to any public features inherited from the *Time* class but no access to inherited private features. Next comes the remainder of the new class's definition, giving only the features that are new to this class:

```
class MoreTime : public dsk::Time
{
   private:
      int millisecond;
   public:
      void SetMillisecond(int);
      int GetMillisecond() const;
};
```

Finally, we add code for the new methods:

```
//*** Set the milliseconds value ***
void MoreTime::SetMillisecond(int v)
{
   //*** If invalid value, exit ***
   if (v < 0 || v > 999)
      return;
   millisecond = v;
}

//*** Returns the milliseconds value ***
int MoreTime::GetMillisecond() const
{
   return millisecond;
}
```

We are now ready to write a `main()` function to make use of our new class:

```
int main()
{
   MoreTime mt;
   mt.SetTime(1, 2, 3);
   mt.SetMillisecond(500);
   cout << mt.GetHour() << ':' << mt.GetMinute() << ':' <<
   ↳mt.GetSecond() << ':' << mt.GetMillisecond() << endl;
}
```

Notice that the code makes a call to *SetTime()* with the line

```
mt.SetTime(1, 2, 3);
```

This is valid because the *MoreTime* class has inherited this method (and its code) from the *Time* class. However, this only assigns values to the *hour*, *minute* and *second* attributes (also inherited) in our *mt* object, so we need to follow this up with the line

```
mt.SetMillisecond(500);
```

in order to set the *millisecond* attribute which did not exist in the *Time* class.

Activity 22.3

Start a new project called *TestMoreTime* and implement the *MoreTime* class using the code given above.

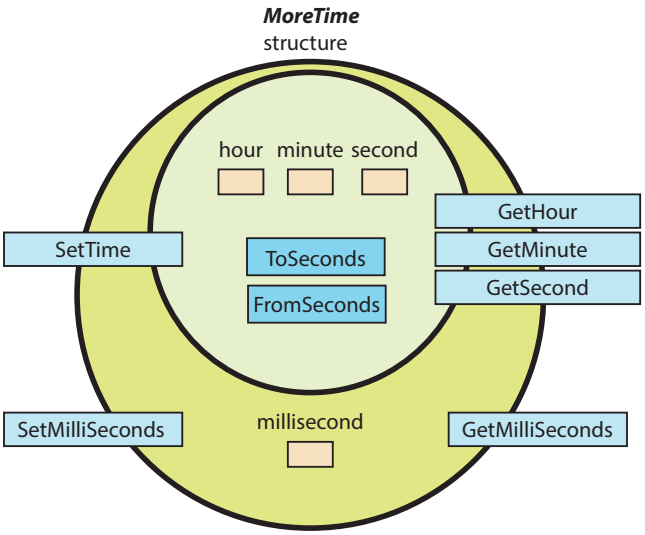
Make the appropriate settings for the compiler and linker.

Add the `main()` program code and test the new class by creating a *MoreTime* object, *mt*, set to 1:02:03:500.

FIG-22.4 visualises the structure created by a *MoreTime* object.

FIG-22.4

Inheritance Visualised



In the diagram we can see the features of the original *Time* class (the inner circle) are embedded in the *MoreTime* design. Elements entirely enclosed in the inner circle cannot be accessed by either the methods of *MoreTime* or by any code which creates a *MoreTime* object.

Any element that extends beyond the inner circle represents a method of the *Time* class that can be accessed by code within the methods of the *MoreTime* class.

Any element that extends beyond the outer circle represents a method coded within the *Time* class or *MoreTime* class that can be accessed by code using *MoreTime* objects.

About Friends and Inherited Methods

Looking back at the code we created for the original *Time* class, we can see that one

of the routines was a friend function rather than a method of the class. This was the overloading of the << operator:

```
//*** Displays the value of time t ***
std::ostream& operator<<(std::ostream& ct, const Time& t)
{
   char timestr[9];
   sprintf(timestr, "%02d:%02d:%02d", t.hour, t.minute,
   ↳t.second);
   ct << timestr;
   return ct;
}
```

This function expects its actual parameters to be an *ostream* object reference and a *Time* object reference. So if `main()` were to contain the line

```
cout << mt;
```

– which is in effect an attempt to call to the above friend function – we'd expect a compilation error since we would be passing the function an `ostream` reference (*cout*) and a *MoreTime* reference (*mt*) rather than an `ostream` reference and a *Time* reference.

But, in fact, the compiler will accept the new code without complaint. And here we meet an important rule regarding derived classes:

If a function or method takes an object reference or pointer of a given class as a parameter it will also accept an object reference or pointer from a class which is derived from that parameter's class.

So, since the `operator<<()` function is coded to accept a reference to a *Time* object it will also accept a reference to a *MoreTime* object since the *MoreTime* class is derived from the *Time* class. This allows us to replace the line

```
cout << mt.GetHour() << ':' << mt.GetMinute() << ':' <<
↳mt.GetSecond() << ':' << mt.GetMillisecond() << endl;
```

with the much shorter

```
cout << mt << ':' << mt.GetMillisecond() << endl;
```

Activity 22.4

In *TestMoreTime* modify the `cout` statement in `main()` to make use of the existing overload on the << operator as shown above and check that the display is identical to that produced by the previous version of the code.

Hiding Inherited Methods

Although a new class may inherit its ancestor's properties, some of the methods it inherits may not be wholly appropriate in the new class. For example, *SetTime()* sets only the *hour*, *minute* and *second* attributes and not *millisecond*.



We saw how to overload functions and operators back in Chapter 12.

However, if the new class has inherited operations which are not appropriate to that new class, we can hide that function by writing a new version. This is known as **hiding an inherited method**.

To do this we simply add the new method's prototype to our new class's definition and then create the code for that new method. So, let's try to create a new *SetTime()* method for the *MoreTime* class. First we have the updated class definition:

```
class MoreTime : public dsk::Time
{
   private:
      int millisecond;
   public:
      void SetMillisecond(int);
      int GetMillisecond() const;
      void SetTime(int, int, int, int);
};
```

And then add the new method's code:

```
//*** Sets the hour, minute, second, millisecond attr ***
void MoreTime::SetTime(int h, int m, int s, int ms)
{
   //*** If any invalid parameter, exit ***
   if (h < 0 || m < 0 || m > 59 || s < 0 || s > 59 || ms < 0 ||
   ↳ms > 999)
      return;
   hour = h;
   minute = m;
   second = s;
   millisecond = ms;
}
```

This allows us to change the code in `main()` to

```
int main()
{
   MoreTime best_time;
   mt.SetTime(1, 2, 3, 500);
   ...
```

Activity 22.5

Add the new *SetTime()* method to *TestMoreTime*. Does the code compile?

The trouble is that the attributes *hour*, *minute* and *second*, although inherited by the new *MoreTime* class, were set as `private` in the original *Time* class and therefore cannot be accessed in the derived class.

But there's another more general problem with the new method's code. This time it's one which is not obvious to us since we also wrote the code for the *Time* base class.

If we had no access to the *Time* classes source code, then we would have no idea that the attributes of that class were called *hour*, *minute* and *second* and therefore we would have no idea how to identify the attributes that should be assigned the parameter values *h*, *m* and *s*.

To try solving both these problems, we have to make use of the inherited version of *SetTime()* by rewriting our new version of *SetTime()* as

```
//*** Sets the hour, minute, second, millisecond attr ***
void MoreTime::SetTime(int h, int m, int s, int ms)
{
   //*** If any invalid parameter, exit ***
   if (h < 0 || m < 0 || m > 59 || s < 0 || s > 59 || ms < 0 ||
   ms > 999)
      return;
   SetTime(h, m, s); //Call the inherited version of SetTime()
   millisecond = ms;
}
```

Activity 22.6

Modify the new *SetTime()* method as shown above. Does the code compile?

And now we find another rule about derived classes:

If an inherited method is rewritten in a derived class, any method defined in the base class using the same function name, is no longer automatically "visible" in the derived class.

This new rule can come as a bit of a surprise. Back in Chapter 12 we saw that, when writing stand-alone functions we could overload the function as often as we wanted (as long as the parameter list differed in each version) and all versions of the function were available. However, that rule does not apply to methods inherited by derived classes.

But we may freely overload a method within a given class (as we did with the - operator in the *Date* class).

Luckily, the *SetTime()* method as defined in the original *Time* class is not entirely lost to us. That original version of the method can be called if we precede the call with its class name and the scope resolution operator :

```
Time::SetTime(h,m,s); //Call the inherited version of SetTime()
```

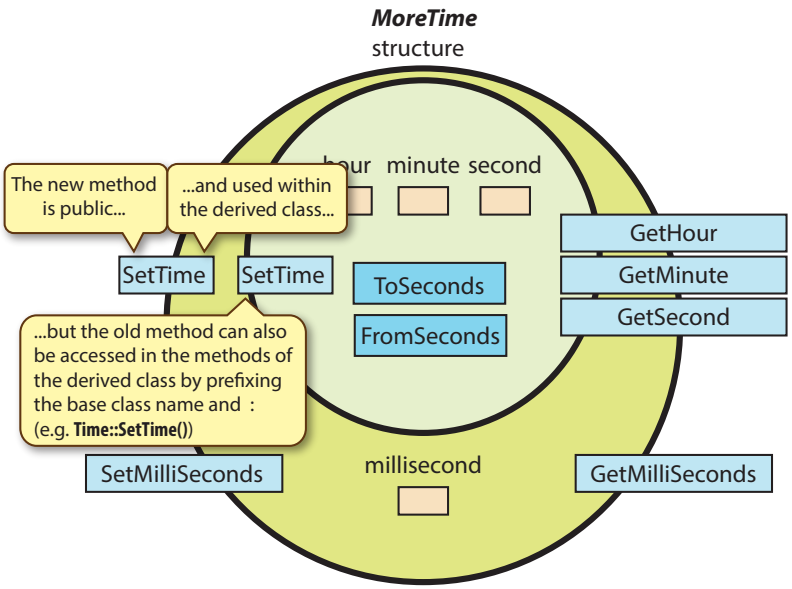
Activity 22.7

Modify the new *SetTime()* method as shown above. Does the code compile?

The effect of creating a new version of an existing method in a derived class is shown in FIG-22.5.

FIG-22.5

Hidden Inherited Methods



An alternative way of calling the base class's *SetTime()* method is to add the line

```
using Time::SetTime;
```

at the start of the definition of *MoreTime*:

```
class MoreTime : public dsk::Time
{

   private:
      int millisecond;
   public:
      using Time::SetTime;
      void SetMillisecond(int);
      int GetMillisecond() const;
      void SetTime(int, int, int, int);
};
```

With this new line in place, we can change *MoreTime*'s *SetTime()* back to its original code:

```
//*** Sets the hour, minute, second, millisecond attr ***
void MoreTime::SetTime(int h, int m, int s, int ms)
{
   //*** If any invalid parameter, exit ***
   if (h < 0 || m < 0 || m > 59 || s < 0 || s > 59 || ms < 0 ||
   ↳ms > 999)
      return;
   SetTime(h, m, s);
   millisecond = ms;
}
```

Of the two options, the first is perhaps less likely to cause confusion.

Many more of the methods inherited by the *MoreTime* class need to be replaced and although it's not our intention to code all of these, let's examine one more method that will need to be replaced – `operator+()`.

Before replacing the method, let's have a look at the code that has been inherited:

```
//*** Returns time of t + current time ***
Time Time::operator+(const Time& t) const
{
   return FromSeconds(ToSeconds() + t.ToSeconds());
}
```

Remember, in reality we may not have access to the code from the base class as we have here. In this case, having that source code is of no great help since *ToSeconds()* and *FromSeconds()* are defined as `private` within the *Time* class, so, like *hour*, *minute* and *second*, we will have no access to them in our replacement method.

It might well be worth the effort to create new `private` methods for the *MoreTime* class called *ToMilliseconds()* and *FromMilliseconds()* (whose functions are, hopefully, obvious). However, in this case, since we don't plan to do a complete coding of the *MoreTime* class, we'll just perform the full calculations within the new `operaor+()` method:

```
MoreTime MoreTime::operator+(const MoreTime& mt) const
{
   MoreTime result;
   int total_millisecs = (GetHour()+mt.GetHour())*3600000
   ↳+ (GetMinute()+mt.GetMinute())*60000 + (GetSecond()+
   ↳mt.GetSecond()) * 1000 + millisecond + mt.millisecond;
   int hours = total_millisecs / 3600000;
   int minutes = (total_millisecs - hours * 3600000) / 60000;
   int seconds = (total_millisecs - (hours * 3600000 + minutes
   ↳* 60000)) / 1000;
```

```
    int millisecs = total_millisecs % 1000;
    result.SetTime(hours, minutes, seconds, millisecs);
    return result;
}
```

And, we'll need the new method's prototype in the class definition:

```
class MoreTime : public dsk::Time
{
   private:
      int millisecond;
   public:
      void SetMillisecond(int);
      int GetMillisecond() const;
      void SetTime(int, int, int, int);
      MoreTime operator+(const MoreTime&) const;
};
```

Activity 22.8

Add the new version of the `+` operator to *MoreTime* and check that the new code operates correctly by calculating *mt3* = *mt1* + *mt2*. (*mt1* = 1:02:03:500, *mt2* = 4:05:06:900)

Another operator that will need to be overridden in the *MoreTime* class is the relational operator `==`. The original version of this returns *true* if the *hour*, *minute* and *second* values are identical in the calling object and a second *Time* object. Since the existing version of `operator==()` does 75% of the testing needed in a new version, we should make use of it in our new code.

We've already used a similar approach when we created a replacement for *Time*'s *SetTime()* in the *MoreTime* class. There we saw that in order to use the older version of *SetTime()* in the later version, we had to add the class name and scope resolution operator:

```
Time::SetTime(h,m,s)
```

The same is true if we want to use the older version of the `==` operator. But in order to do that we need to treat the call more like a traditional function call. So rather than write something like

```
if (*this Time::== tm ...
```

which is invalid syntax, we have to use the expression:

```
if(Time::operator==(tm) ...
```

This means our *MoreTime* version of `==` is coded as

```
bool MoreTime::operator==(const MoreTime& mt) const
{
   return (Time::operator==(mt) && millisecond ==
   ↳mt.millisecond);
}
```

Activity 22.9

Add the new version of the `==` operator to *MoreTime* and check that the new code operates correctly.

Add a new version of the `<` operator to *MoreTime* and test your code.

Methods that are not Inherited

We've seen that a new class can inherit the properties of its base class. But not all properties are inherited. Those not inherited are:

- All constructors
- The destructor
- Friend functions

It should be obvious that a derived class cannot inherit its base class constructor or destructor since these must always have the same name as its class. Therefore the *Clock* class can hardly have a constructor called *Time*!

Constructors in Derived Classes

We already know that every class must have a constructor and a copy constructor. We also know that a zero-argument constructor and copy constructor are created automatically for each class without any input from the programmer.

When we create a derived class its constructor must call the constructor of its base class. Again, the new class's zero-argument constructor and its call to the base class constructor is created automatically as long as the base class also has a zero-argument constructor.

However, if the base class does not have a zero-argument constructor, or we want to create a constructor for the new class which requires one or more arguments, then we have to write our own constructor for the derived class.

So, to know what's required in the way of constructors in *MoreTime* class we need to know what constructors are available in the original *Time* class. Looking at the *ChronoLib.h* file we can see that *Time* has a three-argument constructor:

```
Time(int = 0, int = 0, int = 0);
```

Since each argument is given a default value, this means the constructor can be called without actual parameters and so acts as a zero-argument constructor. This, in turn, means that there is no requirement to write a constructor for the *MoreTime* class.

However, it is likely that we would want to create a constructor for our new class so that objects can be assigned an initial value when they are created. As usual we need to start by updating the definition of *MoreTime*:

```
class MoreTime : public dsk::Time
{
   private:
      int millisecond;
   public:
      MoreTime(int=0, int=0, int=0, int=0);
      void SetMillisecond(int);
      int GetMillisecond() const;
      void SetTime(int, int, int, int);
      MoreTime operator+(const MoreTime&) const;
      bool operator==(const MoreTime&) const;
      bool operator<(const MoreTime&) const;
};
```

and then add the code for the constructor:

```
//*** Initialise MoreTime object ***
MoreTime::MoreTime(int h, int m, int s, int ms)
:Time(h, m, s)
{
   if (ms < 0 || ms > 999)
      millisecond = 0;
   else
      millisecond = ms;
}
```

Notice how we make the call to the base class's constructor. It precedes the opening brace of the new constructor and is of the format

: base class name (parameter list)

If, when we write a constructor for a derived class, the call to the base class's constructor is omitted, then the compiler will automatically add a call to the zero-argument constructor of the base class. If no such constructor exists, the compiler will issue an error message.

The Copy Constructor



Generally, we'd want to code a copy constructor only when the attributes of a class contain pointers.

Like standard constructors, the compiler will automatically generate a copy constructor for a class. This copy constructor will contain a call to the base class's copy constructor.

If we wish to write a copy constructor of our own in a derived class, then that copy constructor should make an explicit call to the base class's copy constructor.

Although a coded copy instructor is unnecessary for the *MoreTime* class, we'll create one here simply to show the syntax involved:

```
//*** Copy constructor ***
MoreTime::MoreTime(const MoreTime& mt)
:Time(mt)
{
   millisecond = mt.millisecond;
}
```

Note that the new copy constructor calls the base class copy constructor using the expression `:Time(mt)`. This is similar to how the *MoreTime* class's standard constructor called the base class standard constructor.

Forcing Constructor Inheritance

Although a derived class will not normally inherit its base class's constructor, we can force our new class to create constructors based on those in its base class. To demonstrate this effect, we'll begin by creating a simple base class with three standard constructors:

```
class Base
{
   public:
      Base()
      {
         cout << "Base class constructor()\n";
      }
      Base(int v)
      {
         cout << "Base class constructor(int)\n";
```

```
        }
        Base(int v, int w)
        {
            cout << "Base class constructor(int,int)\n";
        }
    }
```

Next we'll create a derived class with a single constructor:

```
class Derived :public Base
{
    public:
        Derived(int x, int y, int z)
        {
            cout << "Derived class constructor(int,int,int)\n";
        }
};
```

Finally, we'll create a `main()` function containing a *Derived* object. Since the *Derived* class has only a single constructor, our object will need to be created using three parameters:

```
int main()
{
    Derived obj1(1,2,3);
}
```

Activity 22.10

Create a new project called *TestInheritedConstructors* and implement the *Base*, and *Derive* classes as shown above.

Add a `main()` function with *Derived* object *obj1* using the code given above. What is displayed when the program is executed?

Add the following code to `main()`:

```
Derived obj2(1,2);
Derived obj3(1);
Derived obj4;
```

What happens when you attempt to build this new code?

If we add the line

```
using Base::Base;
```

to the public section of the *Derived* class code, the *Derived* class will inherit the constructors from the *Base* class.

Activity 22.11

Add the above line to the code for the *Derived* class. Does the program run?

Of course, the inherited constructors cannot be used unchanged since a constructor must always have the same name as the class. To solve this problem, the compiler will create new constructors in the *Derived* class in response to the `using` statement we have added.

For example, the new constructor based on `Base(int v,int w)` would be:

```
Derived::Derived(int v, int w)
:Base(v,w)
{}
```

Destructors in Derived Classes

Since destructors can never take parameters, C++ automatically takes care of adding a call to the base class destructor. So, if we are explicitly creating a destructor for the derived class, there is no need to add a call to the base class destructor.

Activity 22.12

In *TestInheritedConstructors*, write destructors for both the *Base* and *Derived* classes. The destructors should output the message "*Base class destructor*" or "*Derived class destructor*" as appropriate.

Observe the order in which the destructors are called.

The Assignment Operator

The requirements of a class's assignment operator are almost identical to that of the copy constructor. However, whereas the copy constructor initialises a newly created object, the assignment operator changes the values in an existing object. Also, unlike the copy constructor, the base class assignment operator is inherited by a derived class. The derived class's assignment operator will use the inherited assignment operator to assign values to the inherited attributes and supply additional code to assign values to any new attributes.

If the new class attributes are basic types (`int`, `float`, etc.) or objects, then there is unlikely to be any need to create a coded assignment operator for the class since the default one created automatically by C++ will be sufficient. But where code is required, generally, we'll want to call the base class's assignment operator to help with the new assignment.

So if we were to write explicit code for the assignment operator in the *MoreTime* class it would be:

```
//*** Copy tm to current object ***
MoreTime& MoreTime::operator=(const MoreTime& tm)
{
   Time::operator=(tm);          //Call base class assignment
   millisecond = tm.millisecond;
   return *this;
}
```

Notice that the call to the base class's assignment operator is within the braces, unlike the call to the base class's copy constructor when we wrote the copy constructor for *MoreTime*.

If we want to stop self-assignment (as in the line `tm = tm;`) then we can add a guard for this:

```
//*** Copy tm to current object ***
MoreTime& MoreTime::operator=(const MoreTime& tm)
{
   if (&tm == this)
      return *this;
   Time::operator=(tm);          //Call base class assignment
   millisecond = tm.millisecond;
```

```
    return *this;
}
```

Activity 22.13

In *TestInheritedConstructors*, add the following assignment operator to the *Base* class:

```
Base& operator=(const Base& bs)
{
      cout << "Base class assignment\n";
      return *this;
}
```

At the end of `main()`, add the line

```
obj4 = obj3;
```

Run the program and observe the message created by the line `obj4 = obj3;` given above.

From the result of Activity 22.13 we can see that the default assignment operator, created automatically for the *Derived* class, has called the *Base* class's assignment operator.

Activity 22.14

In *TestInheritedConstructors*, add the following assignment operator to the *Derived* class:

```
Derived& operator=(const Derived& dr)
{
      cout << "Derived class assignment\n";
      return *this;
}
```

Run the program and observe the message created by the line `obj4 = obj3;`

To have the *Derived* class assignment operator call the *Base* class's assignment operator, we need to add the line

```
Base::operator(dr);
```

Add the above line and check out the change in the messages displayed by the program.

Overloading Friend Functions

We have already used the existing *Time* class friend function `operator<<()` to help output the contents of a *MoreTime* object, but a better solution would be to overload the `<<` operator to take a *MoreTime* reference. And although we could write the new function as

```
//*** Displays the value of time t ***
ostream& operator<<(ostream& ct, const MoreTime& t)
{
   char timestr[12];
   sprintf_s(timestr,"%02d:%02d:%02d:%02d", t.GetHour(),
   ↳t.GetMinute(), t.GetSecond(), t.millisecond);
   ct << timestr;
   return ct;
}
```

this seems to repeat most of what we coded in the original friend function. Another approach would be to use the original friend function to display the *hour*, *minute* and *second* values, then add extra code to display *millisecond*. And this brings us to a similar problem to that encountered when we were rewriting *SetTime()* for the *MoreTime* class: how do we call the original version of the function?

In the case of a method, we saw this was achieved by adding the base class name and the scope resolution operator:

```
Time::SetTime(... //Calls the base class's SetTime()
```

But this time its an earlier version of a friend function we want to call, not a method in the class, so we can't use the same approach. Instead, we have to cast the parameter to the appropriate base class type:

```
//*** Displays the value of time t ***
ostream& operator<<(ostream& ct, const MoreTime& t)
{
   char timestr[12];
   ct << static_cast<const dsk::Time &>(t);
   sprintf_s(timestr,":%03d", t.millisecond);
   ct << timestr;
   return ct;
}
```

Activity 22.15

Reload project *TestMoreTime* and add the code for the **<<** operator as a friend in the *MoreTime* class and check that it can be used to output all four elements of a *MoreTime* object.

The *USDate* Class

We can create a derived class without any new attributes simply in order to override one or more methods. For example, anyone living in the USA probably has a hard time remembering to put the day before the month when they have been testing the *Date* class.

We can easily remedy this problem by creating a *USDate* class derived from our previous *Date* class. The code for the new class is given in FIG-22.6.

FIG-22.6

USDate Class

```
#include <iostream>
#include <ChronoLib.h>
using namespace std;

class USDate : public dsk::Date
{
   public:
      USDate(int = 0, int = 0, int = 0);
      void SetDate(int, int, int);
      friend ostream& operator<<(ostream&, const USDate&);
};

USDate::USDate(int m, int d, int y)
:Date(d,m,y)
{
}
```



FIG-22.6
(continued)

USDate Class

```
void USDate::SetDate(int m, int d, int y)
{
   Date::SetDate(d, m, y);
}

ostream& operator<<(ostream& ct, const USDate& dt)
{
   ct << dt.GetMonth() << '/' << dt.GetDay() << '/' <<
   ↳dt.GetYear();
   return ct;
}

int main()
{
   USDate dt(11, 23, 1963);
   cout << dt << endl;
}
```



By overriding *SetDate()* in the *USDate* class we lose default access to the two versions of *SetDate()* which would have otherwise been inherited from the *Date* class.

Notice that the *USDate* constructor does nothing other than call the *Date* constructor.

Activity 22.16

Create a new project called *TestUSDate* and implement the *USDate* class, testing it to make sure the date is handled correctly and the month is placed before the day in the display.

REMEMBER to add the necessary library information to the project's properties.

We now have two date-related classes: *Date* and *USDate* with the second being derived from the first. Can we copy a *USDate* value into a *Date* object? For example, is it valid to write

```
USDate dt1(11, 23, 1963);
dsk::Date dt2(25,12,2020);
dt2 = dt1;
```

and will we end up with the date 23/11/1963 in object *dt2*?

And what about the other way round. Will

```
USDate dt1(11, 23, 1963);
dsk::Date dt2(25,12,2020);
dt1 = dt2;
```

Will this give us the date 12/25/2020 in *dt1*?

Activity 22.17

Modify *TestUSDate* to check if either of the above assignment statements work.

As we see from the results of Activity 22.17, a *USDate* object's value can be assigned to a *Date* object but not the other way round.

C++ allows a derived class's value to be assigned to an object of its base class (although if the derived class contains additional attributes not present in the base class, these will be ignored). However, a base class value cannot be assigned to a derived class object unless, in the derived class we create a constructor which takes

a base class argument as in the code

```
USDate::USDate(const Date& dt)
   :Date(dt.GetDay(), dt.GetMonth(), dt.GetYear())
{}
```

The *USDate* class would then automatically use this constructor to create a *USDate* object from the *Date* object in the line

```
dt1 = dt2;
```

and hence allow the assignment to take place.

Activity 22.18

Add the above constructor to the *USDate* class along with any other necessary code then test the assignment `dt1 = dt2`.

The *Clock* Class

To state the obvious, most simple clocks show the current time. Clocks with a digital display normally offer a 24 hour option as an alternative to displaying *am* or *pm*.

If we wanted to create a 24-hour *Clock* class, which differs from the *Time* class in that the hour value can never go beyond 23, then one possible attempt at implementing and testing this class is shown in FIG-22.7.

FIG-22.7

Clock Class

```
#include <iostream>
#include <ChronoLib.h>

using namespace std;


class Clock : public dsk::Time
{
   public:
      Clock(int=0, int=0, int=0);
      Clock(const Time&);
      Clock(const Clock&);
};

Clock::Clock(int h, int m, int s)
:Time(h,m,s)
{
   if (h > 23)
      SetTime(0,0,0);
}

Clock::Clock(const Time& tm)
{
   if (tm.GetHour() <= 23)
      SetTime(tm.GetHour(), tm.GetMinute(), tm.GetSecond());
   else
      SetTime(0,0,0);
}

Clock::Clock(const Clock& ck)
   :Time(ck)
{}
```



FIG-22.7
(continued)

Clock Class

```
int main()
{
   Clock ck(23, 59, 59);   //Should be okay
   Clock ck2(24,0,0);      //Should be invalid - time set to 0:0:0

   cout << ck << endl;
   cout << ck2 << endl;
}
```

Activity 22.19

Start a new project called *TestClock* and implement the code given above. Does the program compile and run?

Replacing *SetTime()*

We won't want the clock to show an *hour* value greater than 23 (which the *Time* class allows), so we'll need to replace *SetTime()* – something we also did in the *MoreTime* class. Our new code can be:

```
void Clock::SetTime(int h, int m, int s)
{
   if (h <= 23)
      Time::SetTime(h, m, s);
}
```

Notice that the code just checks that the *h* parameter isn't greater than 23 before getting the inherited version of the method to do all of the work – it's that inherited version that will reject any invalid *m* or *s* parameter values.

Adding our updated *SetTime()* method has the knock-on effect on the existing *Clock* class constructors, since it will now be the new version of *SetTime()* that is called by the constructors – not the inherited version which is now hidden. However, this makes no difference to the code required in the constructors.

Activity 22.20

In *TestClock* add the new *SetTime()* method and use it to set the times for objects *ck* and *ck2*. Check that the results remain the same as before.

In fact, several other inherited methods have the same problem as *SetTime()* in that they allow the time's hour setting to be greater than 23 and if our main purpose was to create a final version of the *Clock* class we would have to create new versions of those methods. However, a change to *SetTime*() will be sufficient for now.

Tick()

Any working clock will update the displayed time every second, so let's add a *Tick()* method which performs that task to the *Clock* class. Our new method will have the following code:

```
void Clock::Tick()
{
   *this = FromSeconds(ToSeconds() + 1);
}
```

And `main()` will be re-coded as:

```
int main()
{
   Clock ck(23, 10, 59);
   Clock ck2(23,59,59);
   ck.Tick();
   ck2.Tick();
   cout << ck << endl;
   cout << ck2 << endl;
}
```

Activity 22.21

Make the necessary changes to *Source.cpp* in *TestClock* to add the *Tick()* method to the *Clock* class and update `main()`. Does the program compile?

But there is a problem with this code. *FromSeconds()* and *ToSeconds()* used in the *Tick()* method were declared as private in the *Time* class and so cannot be accessed in the *Clock* class.

The reason these methods were declared as private was that the designer of the *Time* class did not wish any application programmer who was using *Time* objects to have access to the two methods. They are included in the *Time* class simply as helper functions called when coding other methods of the class (such as `operator+=()`).

Protected Properties

Although the programmer who coded the *Time* class rightly wanted to deny access to *FromSeconds()* and *ToSeconds()* to any application programmer, accessing these two methods could be useful to anyone creating a new class derived from the *Time* class.

So far we have seen that a property in a class can be marked as either `public` or `private`. Public properties can be accessed in both application programs and in derived classes; private properties cannot be accessed in application programs or derived classes.

A third option exists when defining a class property's access specifier. This is `protected`. A `protected` property cannot be accessed in an application program which uses objects of the class, but may be accessed in a derived class. Using this knowledge, a better option for the *Time* class programmer would have been to start the definition of the *Time* class using the code

```
//*** Time class declaration ***
class Time
{
   private:
      int hour;
      int minute;
      int second;
   protected:
      int ToSeconds() const;
      Time FromSeconds(int) const;
   public:
      ...
```

No other changes to the code are required.

Activity 22.22

Load the *MyLibraries* project and change the code in *ChronoLib.h* to make *ToSeconds()* and *FromSeconds()* protected rather than private. Recompile the library.

Reload *TestClock*. Does it compile without errors and produce the expected results?

At this point we might be tempted to make attributes such as *hour*, *minute* and *second* protected rather than public since this would allow us to change the *Clock* class's

```
Clock::Clock(const Time& tm)
{
   SetTime(tm.GetHour(), tm.GetMinute(), tm.GetSecond());
}
```

to

```
Clock::Clock(const Time& tm)
{
   SetTime(tm.hour, tm.minute, tm.second);
}
```

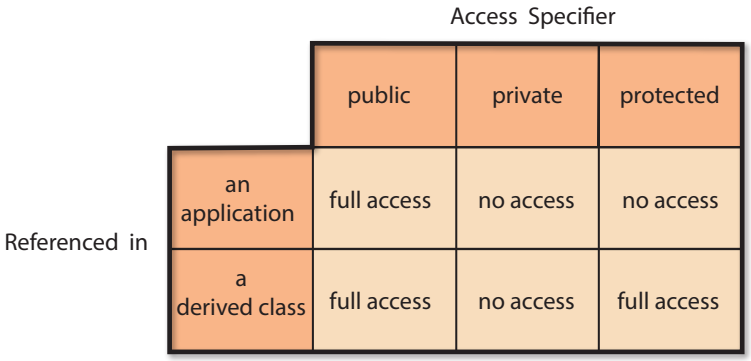
However, it should be pointed out that making an attribute `protected` still leads to a weakening of the encapsulation principles of object-oriented design. We originally made attributes private to ensure any application programmer could not mistakenly store invalid values in those attributes. A programmer creating derived classes is just as likely to make mistakes!

On the other hand, changing a method from private to protected gives more options to the programmer creating a derived class without compromising the application programmer's access.

FIG-22.8 gives a summary of the access allowed by each access specifier.

FIG-22.8

Access Specifier Properties



Changing the Access Specifier of Inherited Properties

using

Let's assume that the creators of the *Clock* class have decided that they would like to allow any application programmer who uses *Clock* objects to have access to the *ToSeconds()* and *FromSeconds()* methods.

With the current implementation of the *Clock* class, that would be impossible since

the application programmer does not have access to these protected features. However, by modifying the definition of the *Clock* class to read

```
class Clock : public dsk::Time
{
   public:
      Clock(int=0, int=0, int=0);
      Clock(const Time&);
      Clock(const Clock&);
      void Tick();
      using Time::ToSeconds;
      using Time::FromSeconds;
};
```

the two inherited methods become public in the *Clock* class. Note that no parentheses are given after the method name in the `using` statements.

We can use this approach to change the access specifier of any protected or public inherited feature. By placing the `using` statement within a class's `private`, `public` or `protected` keywords, the inherited feature takes on that access mode in the class being coded.

The access of inherited private features cannot be modified in this way.

Activity 22.23

In *TestClock*, adjust the definition of the *Clock* class to make *ToSeconds()* and *FromSeconds()* public.

Test the use of the *ToSeconds()* method, by displaying the time stored in object *ck* in seconds both before the line `ck.Tick();` and after.

Let's say we decide that the `operator-()` method inherited from the *Time* class should not be available to application programmers using *Clock* objects. To do this all that is needed is to add the appropriate `using` clause under the term `private` (or `protected`):

```
class Clock : public dsk::Time
{
   private:
      using Time::operator-;
   public:
      ...
```

The above change would allow `operator-()` to be called within any methods of the *Clock* class, but not by any application program using *Clock* objects or any new class derived from the *Clock* class.

Activity 22.24

In *TestClock*, make `operator-()` private and check that it cannot be used by object *ck* in `main()`. Remove the calls to *Tick()* and *ToSeconds()*.

delete

Another way of stopping an application getting access to an inherited method is to delete that method from the derived class. For example, we could delete `operator+()`

from the *Clock* class by including the line

```
Time operator+(const Time&) const = delete;
```

in the *Clock* class definition, as shown below:

```
class Clock : public dsk::Time
{
   private:
      using Time::operator-;
   public:
      Time operator+(const Time&) const = delete;
      ...
```

Activity 22.25

In *TestClock*, stop the *Clock* class from inheriting `operator+()` and check that it cannot be used by object *ck* in `main()`.

The Correct Result for the Wrong Reason

Returning to the times displayed when *TestClock* is run, it is just by luck that object *ck2* shows the correct time. The inherited *FromSeconds()* method is capable of returning a *Time* value greater than 23:59:59. In fact, after excuting the line `ck2.Tick();`, *FromSeconds()* will return a time of 24:00:00. This value will then be used by *Tick()* and assigned to *ck2*'s attributes. In order to do that the *Time* object returned by *FromSeconds()* will need to be converted to a *Clock* value using the *Clock* class's second constructor. This rejects times greater than 23:59:59 and sets the object to the default 00:00:00 (midnight).

Activity 22.26

In *TestClock*, add an updated version of *FromSeconds()* to the *Clock* class which only returns values in the range 0:0:0 to 23:59:59.

The *CountdownClock* Class

In the *Clock* class we created a function called *Tick()* which added one second to the current time. While most devices count time in this way, adding one second for each tick of the clock, there are devices that do exactly the opposite. For example, when awaiting an imminent rocket launch, time counts backwards. Each second that passes reduces the time by one second.

If we want to implement a *CountdownClock* class containing a *Tick()* method which reduces the time by one second, it can be derived from the *Clock* class. The code for the new class is given in FIG-22.9.

FIG-22.9

CountdownClock Class

```
class CountdownClock : public Clock
{
   public:
      CountdownClock(int=0, int=0, int=0);
      CountdownClock(const Time&);
      void Tick();
};

CountdownClock::CountdownClock(int h, int m, int s)
   :Clock(h, m, s)
{}
```



FIG-22.9
(continued)

CountdownClock Class



Negative values are not allowed.

```
CountdownClock::CountdownClock(const Time& tm)
   : Clock(tm)
{}

void CountdownClock::Tick()
{
   if (ToSeconds() == 0)
      return;
   (*this)--;
}
```

Note that the *Tick()* method is implemented using the `--` operator which was originally defined in the *Time* class. In fact, we could have used the line

```
(*this)++;
```

to implement the *Tick()* method in the *Clock* class – but using *FromSeconds()* and *ToSeconds()* gave us a reason to introduce the `protected` access specifier!

Activity 22.27

Modify *TestClock* by adding the code for the *CountdownClock* class.

Change the code in `main()` to

```
int main()
{
      Clock ck(23, 59, 59);
      CountdownClock ck2(23, 59, 59);
      ck.Tick();
      ck2.Tick();
      cout << ck << endl;
      cout << ck2 << endl;
}
```

Check that the *Clock* object adds one second and that the *CountdownClock* object subtracts one second.

Static Attributes in Derived Classes

We already know that a property that has been declared as `static` within a class has only one occurrence irrespective of the number of objects of that class which have been created.

This rule extends to classes derived from a class containing static attributes. For example, the *Time* class contains a static attribute called *SECSINDAY*. No matter how many *Time*, *Clock* or *CountdownClock* objects a program creates, there remains only a single instance of this static attribute and all objects from these two classes access that same instance.

Inheritance Options

When we created the *Clock* class the first line of the code was

```
class Clock : public Time
```

The keyword `public` as used here determines the access modifier for those properties inherited from the *Time* class into our *Clock* class.

Using the term `public` means that the inherited features retain their original access modifier settings as far as access is concerned in the *Clock* class. This means that features labelled as `public` or `protected` can be accessed by the methods of the *Clock* class but those marked as `private` cannot.

If we had written that first line as

```
class Clock : protected Time
```

then the inherited *public* and *protected* properties of the *Time* class would become `protected` features in the *Clock* class. The result is that methods of the *Clock* class can have full access to these properties but application programmers creating *Clock* objects do not have access. Classes derived from the *Clock* class will still have access to these features.

Private properties of the *Time* class remain private and cannot be accessed directly by the *Clock* class.

The final option for the first line is

```
class Clock : private Time
```

Now all public and protected features of the *Time* class will be treated as private. The methods defined within the *Clock* class can access these features but they cannot be accessed by derived classes nor application programs.

Private properties of the *Time* class remain private and cannot be accessed directly by the *Clock* class.

final

If we needed to create a class which can never be the base class for any future class, we can add the keyword `final` immediately after that class's name in the first line of the class definition. For example, we could stop any new class being derived from the *Clock* class by starting the definition with the line

```
class Clock final : public dsk::Time
{
   ...
```

Now if a future program contained a line such as

```
class AtomicClock: public Clock
```

the compiler would raise an error report such as

```
Error C3246  'AtomicClock': cannot inherit from 'Clock' as it has been declared as
↳'final'
```

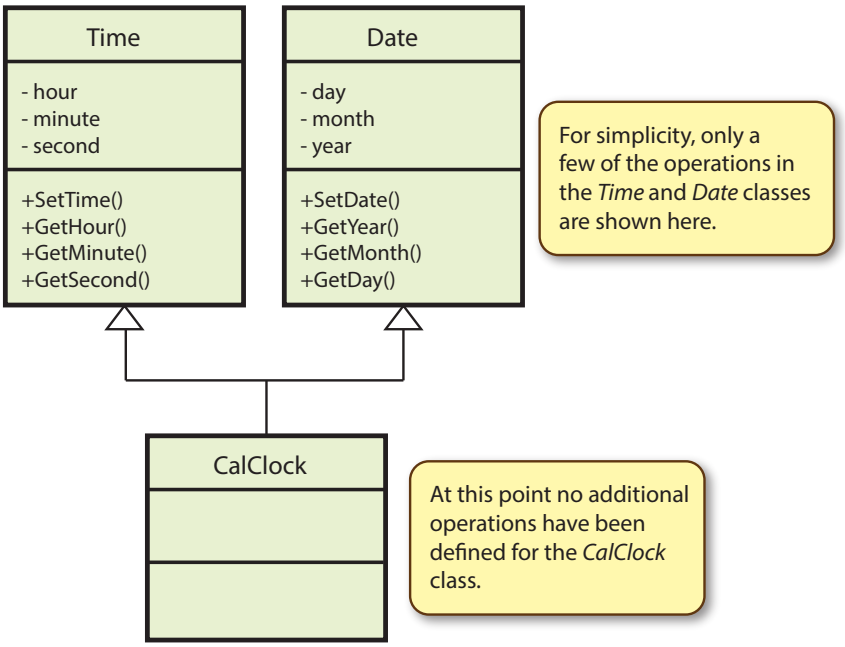
Multiple Inheritance

Imagine we wish to create a clock that shows not only the time, but also the date. For our original clock we derived it from the *Time* class; for our new clock it would be nice if we could derive it from both the *Time* and *Date* class.

C++ allows a class to inherit the features of more than one base class. This is termed multiple inheritance. For simple situations multiple inheritance might seem like a good idea but it can cause unwanted complexity as we will see later. Let's call our new class *CalClock* (short for *Calendar Clock*). FIG-22.10 shows how we represent *CalClock*'s multiple inheritance in a class diagram.

FIG-22.10

Multiple Inheritance



Implementing Multiple Inheritance

The first line of our new class adds a comma-separated list of the classes it inherits from:

```
class CalClock : public dsk::Time, public dsk::Date
```

This is followed by the usual list of new attributes and methods. In this case we'll restrict ourselves to constructors, *SetCalClock()* *GetDate()*, *GetTime()* and *Tick()* methods and an `operator<<()` friend function:

```
class CalClock : public dsk::Time, public dsk::Date
{
   public:
      CalClock()= default;
      CalClock(const Date&, const Time&);
      CalClock(const CalClock&);
      void SetCalClock(const Date&, const Time&);
      dsk::Time GetTime() const;
      dsk::Date GetDate() const;
      void Tick();

      friend ostream& operator<<(ostream&, const CalClock&);
};
```

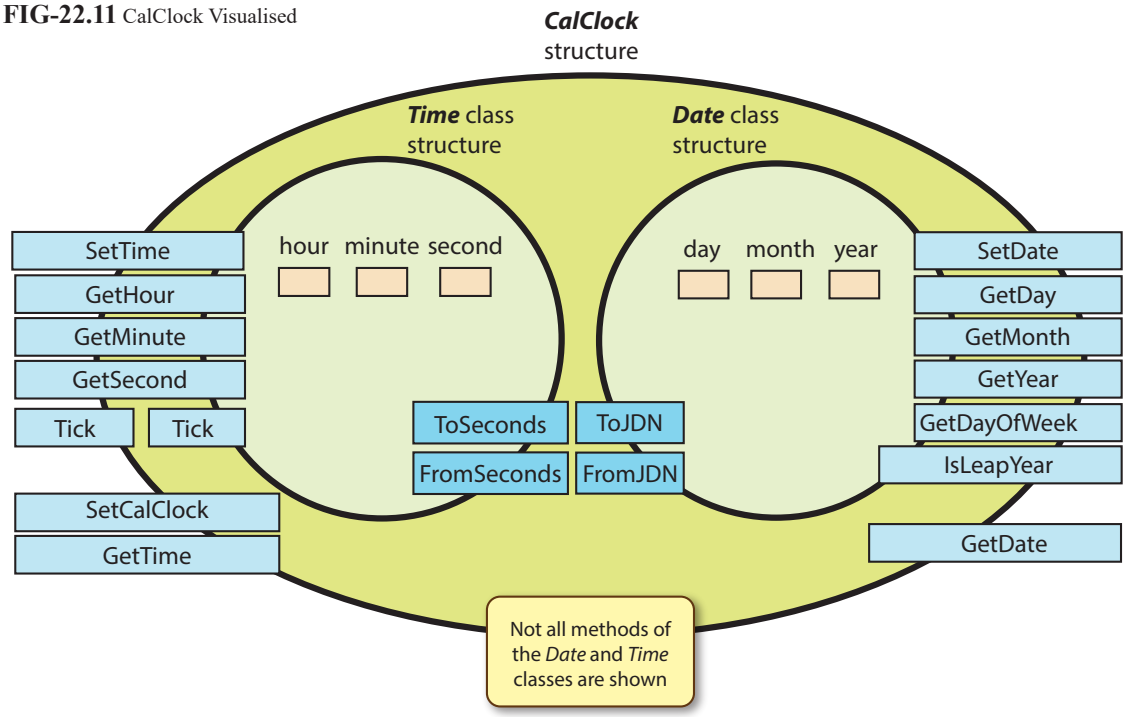
Because the new class has two base classes, the new constructors must call the constructors of both these base classes. If those constructors require parameters, then the calls to them must be explicit:

```
CalClock::CalClock(const Date& dt, const Time& tm)
:Date(dt),Time(tm)
{}
```

Note the calls are comma-separated. The order of the constructor calls is unimportant.

A visual representation of a *CalClock* object is shown in FIG-22.11.

FIG-22.11 CalClock Visualised



Activity 22.28

Re-open project *TestClock* and implement the *CalClock* class and its friend function. Test the new class using the following code in `main()`:

```
int main()
{
   CalClock cck(dsk::Date(1, 1, 2021), dsk::Time(23,59,59));
   cck.Tick();
   cout << cck << endl;
}
```

The Problems with Multiple Inheritance

As a general rule, the use of multiple inheritance is discouraged. In fact, some languages such as Java do not allow multiple inheritance. However, it may, on the rare occasion, offer the best solution to a problem.

The first problem occurs when two or more of the base classes contain methods with the same names and parameter types. For example, the *Time* class and the *Date* class both contain definitions for `operator++()`. Therefore any program which makes use of a *CalClock* object and attempts to execute the ++ operation using code such as

```
CalClock cck(dsk::Date(1, 1, 2021), dsk::Time(11, 59, 59));
cck++;
```

will see a compiler error message such as

```
Error C2593   'operator ++' is ambiguous
```

since the compiler has no way of knowing which inherited version of ++ it should use. To solve this problem is to add the class from which the method was inherited:

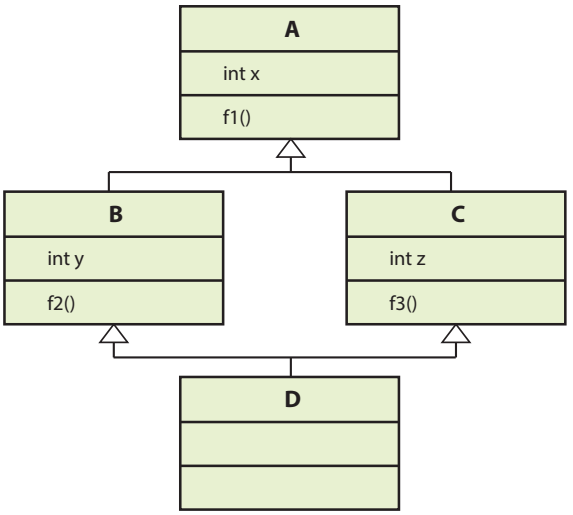
```
cck.Time::operator++();
```

Unfortunately, we have to revert to the function-call format in order to add the class information when dealing with an ambiguous operator.

The next glitch in multiple inheritance is known as **the diamond problem**. The origin of the name becomes apparent if we look at a class diagram for this situation (see FIG-22.12).

FIG-22.12

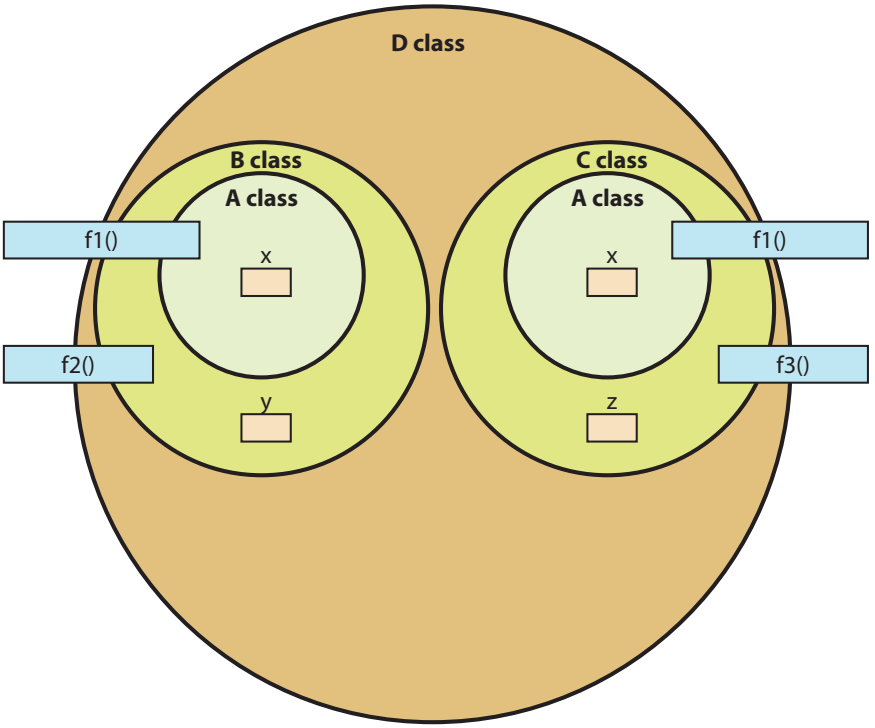
The Diamond Problem



And if we convert this to the visualisation diagram we've used previously we get the setup shown in FIG-22.13.

FIG-22.13

D Class Visualised



As we can now see, because class *A* is a common ancestor of classes *B* and *C* and since class *D* is derived from classes *B* and *C*, class *D* has ended up with two copies

of the attribute *x* and two copies of the method *f1()*.

A program showing a similar situation is given in FIG-22.14.

FIG-22.14

Implementing the Diamond Problem

```
#include <iostream>
using namespace std;

class A
{
   private:
      int x;
   public:
      void f1()
      {
         cout << "Executed f1()\n";
      }
};

class B : public A
{
   private:
      int y;
   public:
      void f2()
      {
         cout << "Executed f2()\n";
      }
};

class C : public A
{
   private:
      int z;
   public:
      void f3()
      {
         cout << "Executed f3()\n";
      }
};

class D : public B, public C
{};


int main()
{
   D var;
   var.f1();
}
```

Activity 22.29

Start a new project called *MultiInherit* and implement the code above.

What error message occurs during compilation?

As we've seen previously, the compiler reports the call to *f1()* as ambiguous. Trying to sort this by changing the line in `main()` to

```
var.A::f1();
```

doesn't help, since there are two copies of *A*'s *f1()* method in the *D* class object.

In fact we have to write

```
var.B::f1();   //OR var.C::f1();
```

to get the program to run. This tells the compiler that it's the version of *f1()* inherited through the embedded *B* class object within *var* that we want to run. Note that it is also valid to write `var.B::A::f1();`

Activity 22.30

Modify *MultiInherit* so that the program compiles and runs.

By adding a zero-argument constructor to each class with code such as

```
class A
{
   private:
      int x;
   public:
      A()
      {
         cout << "A class constructor called\n";
      }
      void f1()
      {
         cout << "Executed f1()\n";
      }
};
```

we can get an insight into how the embedded objects in *var* are created.

Activity 22.31

Modify *MultiInherit* by adding a constructor in the form of the one given above to each class and examine the order in which the constructors are called.

In the next chapter we'll return to the diamond problem and see how it is solved.

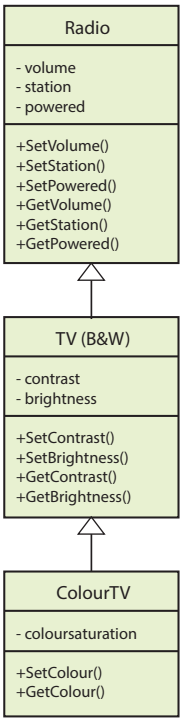
Summary

- A new class can be derived from an existing class.
- The new class is referred to as the derived class, subclass, or child class.
- The existing class is referred to as the base class, superclass, or parent class.
- A derived class inherits all of the attributes and most of the methods of the base class.
- Features in a class can be marked as `private`, `protected` or `public`.
- A class's protected features can be accessed in any future derived classes but cannot be accessed in applications using objects of any class.
- Methods not inherited are the constructors and destructors.
- Friend functions have no special access in derived classes.

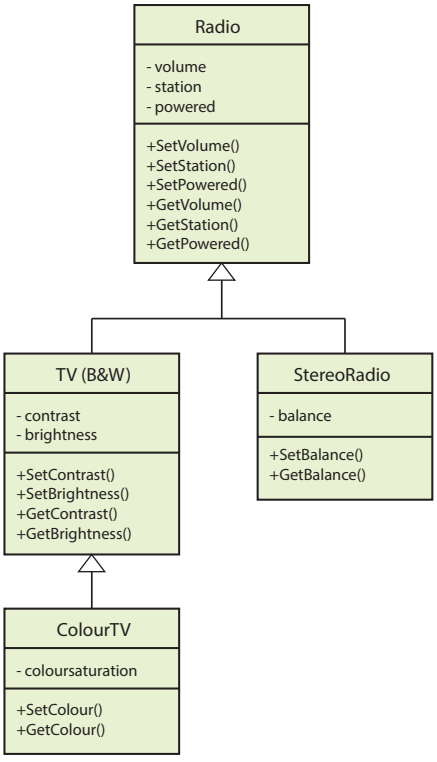
- If a derived class contains a new version of a method, all inherited methods of the same name are hidden.
- Hidden methods can be accessed by prefixing the method name with the base class name and the scope resolution operator (*base_class::method()*).
- A derived class cannot access private features of its base class.
- A derived class's constructor must call its base class's constructor.
- If the base class constructor requires no arguments, its call from the derived class's constructor is automatic.
- A derived class's destructor must call its base class's destructor but this is handled automatically.
- Protected properties can be accessed in the methods of a derived class but not in an application program.
- Use `using` to change the access specifier of an inherited feature (except inherited private features).
- Use `delete` to remove an inherited method from a class.
- Static features are not duplicated in derived class objects. All objects that belong to that family of classes access the same, unique, static feature.
- The derived class can set the access specifier of its base class as `public`, `protected` or `private`.
- If a derived class's base class is public, that base class's protected and public features are treated as protected or public in the derived class.
- If a derived class's base class is protected, that base class's protected and public features are treated as protected in the derived class. This means they cannot be accessed in an application program but can be accessed in a derived class.
- A class marked as `final` cannot be used to derive a new class.
- Multiple inheritance allows a class to have more than one base class.
- Multiple inheritance can give rise to attributes or methods sharing names.
- The diamond problem occurs when two or more classes, sharing a common base class, are then themselves used as the base classes in a multiple inheritance structure.

Solutions

Activity 22.1



Activity 22.2



Activity 22.3

Code for *TestMoreTime*:

```
#include <iostream>
#include<ChronoLib.h>
using namespace std;

class MoreTime : public dsk::Time
{
  private:
    int millisecond;
  public:
    void SetMillisecond(int);
    int GetMillisecond() const;
};

//*** Set the milliseconds value ***
void MoreTime::SetMillisecond(int v)
{
  //*** If invalid value, exit ***
  if (v < 0 || v > 999)
    return;
  millisecond = v;
}

//*** Returns the milliseconds value ***
int MoreTime::GetMillisecond() const
{
  return millisecond;
}

int main()
{
  MoreTime mt;
  mt.SetTime(1, 2, 3);
  mt.SetMillisecond(500);
  cout << mt.GetHour() << ':' << mt.GetMinute() <<
  ':' << mt.GetSecond() << ':' <<
  mt.GetMillisecond() << endl;
}
```

Project>TestMoreTime Properties... settings:

C/C++ > General > Additional Include Directories :
E:\Hands On C++\Programs\MyLibraries\ChronoLib

Linker > General > Additional Library Directories :
E:\Hands On C++\Programs\MyLibraries\Debug

Linker > Input > Additional Dependencies :
ChronoLib.lib

If you have stored your library elsewhere, then the values given above will be different.

The program should compile and run, displaying the value

1:2:3:500

Activity 22.4

Modified code for `main()` in *TestMoreTime*:

```
int main()
{
  MoreTime mt;
  mt.SetTime(1, 2, 3);
  mt.SetMillisecond(500);
  cout << mt << ':' << mt.GetMillisecond() << endl;
}
```

The output is the same as before.

Activity 22.5

The new code does not compile. The attributes *hour*, *minute* and *second* are reported as inaccessible.

Activity 22.6

The new code still does not compile. This time the compiler states that *SetTime()* does not take three arguments.

Activity 22.7

Modified code for *SetTime()* in *TestMoreTime*:

```
//*** Sets the hour, minute, second, millisecond
↳attr ***
void MoreTime::SetTime(int h, int m, int s, int ms)
{
  //*** If any invalid parameter, exit ***
  if (h < 0 || m < 0 || m > 59 || s < 0 || s > 59 ||
  ↳ms < 0 || ms > 999)
    return;
  Time::SetTime(h, m, s);
  millisecond = ms;
}
```

This time the code compiles and runs, giving the same result as before.

Activity 22.8

Modified code for *MoreTime* definition:

```
class MoreTime : public dsk::Time
{
  private:
    int millisecond;
  public:
    void SetMillisecond(int);
    int GetMillisecond() const;
    void SetTime(int, int, int, int);
    MoreTime operator+(const MoreTime&) const;
};
```

New method's code:

```
MoreTime MoreTime::operator+(const MoreTime& mt)
const
{
  MoreTime result;
  int total_millisecs = (GetHour()+mt.GetHour())*
  ↳3600000 + (GetMinute()+mt.GetMinute())*60000 +
  ↳(GetSecond()+ mt.GetSecond()) * 1000 +
  ↳millisecond + mt.millisecond;
  int hours = total_millisecs / 3600000;
  int minutes = (total_millisecs - hours * 3600000)
  ↳/ 60000;
  int seconds = (total_millisecs - (hours * 3600000
  ↳+ minutes
  ↳* 60000)) / 1000;
  int millisecs = total_millisecs % 1000;
  result.SetTime(hours, minutes, seconds,
  ↳millisecs);
  return result;
}
```

Modified code for `main()` in *TestMoreTime*:

```
int main()
{
  MoreTime mt1, mt2, mt3;
  mt1.SetTime(1, 2, 3, 500);
  mt2.SetTime(4, 5, 6, 900);
  mt3 = mt1 + mt2;
  cout << mt3 << ':' << mt3.Gemtillisecond() <<
  ↳endl;
}
```

The sum in *mt3* should be calculated correctly.

Activity 22.9

Modified code for *MoreTime* definition:

```
class MoreTime : public dsk::Time
{
  private:
    int millisecond;
  public:
    void Semtillisecond(int);
    int Gemtillisecond() const;
    void SetTime(int, int, int, int);
    MoreTime operator+(const MoreTime&) const;
    bool operator==(const MoreTime&) const;
    bool operator<(const MoreTime&) const;
};
```

New methods:

```
bool MoreTime::operator==(const MoreTime& mt) const
{
  return (Time::operator==(mt) && millisecond ==
  ↳mt.millisecond);
}

bool MoreTime::operator<(const MoreTime& mt) const
{
  return (Time::operator<(mt) && millisecond <
  ↳mt.millisecond);
}
```

Modified code for `main()` in *TesmtoreTime*:

```
int main()
{
  MoreTime mt1, mt2, mt3;
  mt1.SetTime(1, 2, 3, 500);
  mt2.SetTime(4, 5, 6, 900);
  mt3 = mt1 + mt2;
  cout << mt3 << ':' << mt3.GetMillisecond()<< endl;
  if (mt1 == mt2)
    cout << "Equal\n";
  else
    if (mt1 < mt2)
      cout << "mt1 is less than mt2\n";
    else
      cout << "mt1 is more than mt2\n";
}
```

Activity 22.10

Code for *TestInheritedConstructors*:

```
#include <iostream>
using namespace std;

class Base
{
  public:
    Base()
    {
      cout << "Base class constructor()\n";
    }
    Base(int v)
    {
      cout << "Base class constructor(int)\n";
    }
    Base(int v, int w)
    {
      cout << "Base class constructor(int,int)\n";
    }
  };

class Derived :public Base
{
  public:
    Derived(int x, int y, int z)
    {
      cout << "Derived class constructor(
      ↳int,int,int)\n";
    }
};

int main()
{
  Derived obj1(1,2,3);
}
```

The output produced is

```
Base class constructor()
Derived class constructor(int,int,int)
```

From this we can see that the *Derived* class's constructor has automatically called the *Base* class's zero-argument constructor.

Modified version of `main()`:

```
int main()
{
  Derived obj1(1,2,3);
  Derived obj2(1,2);
  Derived obj3(1);
```

```
    Derived obj4;
}
```

This code does not compile. The following error messages are produced:

```
Error  C2661    'Derived::Derived': no overloaded function
↳takes 2 arguments

Error  C2664    'Derived::Derived(Derived &&)': cannot convert
↳argument 1 from 'int' to 'const Derived &'

Error  C2512    'Derived': no appropriate default constructor
↳available
```

These messages are, in effect, telling us that no constructors exist for the *Derived* class which takes 2, 1 or 0 parameters.

Activity 22.11

Modified code for *Derived class*:

```
class Derived :public Base
{
  public:
    using Base::Base;
    Derived(int x, int y, int z)
    {
      cout << "Derived class constructor(
      ↳int,int,int)\n";
    }
};
```

This produces the following output:

```
Base class constructor()
Derived class constructor(int,int,int)
Base class constructor(int,int)
Base class constructor(int)
Base class constructor()
```

We can see that only *Base* class messages have been displayed for the three new objects.

However, don't forget that the objects *obj2*, *obj3* and *obj4* have been created by calls to the constructors created by the compiler in the *Derived* class in response to our adding the `using` statement.

Since these automatically created constructors call the equivalent *Base* class constructor (which displays messages) while the new *Derived* class constructor contains no output messages of their own so we receive no indication of their execution.

Activity 22.12

Modified code for the *Base* class:

```
class Base
{
  public:
    Base()
    {
      cout << "Base class constructor()\n";
    }
    Base(int v)
    {
      cout << "Base class constructor(int)\n";
    }
    Base(int v, int w)
    {
      cout << "Base class constructor(int,int)\n";
    }
    ~Base()
    {
      cout << "Base class destructor\n";
    }
};
```

Modified code for the *Derived* class:

```
class Derived :public Base
{
  public:
    using Base::Base;
    Derived(int x, int y, int z)
    {
      cout << "Derived class constructor(
      int,int,int)\n";
    }

    ~Derived()
    {
      cout << "Derived class destructor\n";
    }
};
```

When the program is executed, we can see that each object is destroyed by first calling the *Derived* class destructor and then the *Base* class's destructor. This produces, for each object, the following messages:

```
Derived class destructor
Base class destructor
```

Activity 22.13

Modified code for the *Base* class:

```
class Base
{
  public:
    Base()
    {
      cout << "Base class constructor()\n";
    }
    Base(int v)
    {
      cout << "Base class constructor(int)\n";
    }
    Base(int v, int w)
    {
      cout << "Base class constructor(int,int)\n";
    }
    ~Base()
    {
      cout << "Base class destructor\n";
    }
    Base& operator=(const Base& bs)
    {
      cout << "Base class assignment\n";
      return *this;
    }
};
```

Modified code for `main()`:

```
int main()
{
  Derived obj1(1,2,3);
  Derived obj2(1,2);
  Derived obj3(1);
  Derived obj4;
  obj4 = obj3;
}
```

In the output produced, immediately before the first destructor message, the line

```
Base class assignment
```

appears, showing that the default assignment for the *Derived* class has called the coded assignment in the *Base* class.

Activity 22.14

Modified code for the *Derived* class:

```
class Derived :public Base
{
  public:
    using Base::Base;
    Derived(int x, int y, int z)
    {
```

```
      cout << "Derived class constructor(
      ↳int,int,int)\n";
    }
    ~Derived()
    {
      cout << "Derived class destructor\n";
    }
    Derived& operator=(const Derived& dr)
    {
      cout << "Derived class assignment\n";
      return *this;
    }
};
```

This time, the message output by the line `obj4 = obj3;` is

```
Derived class assignment
```

Modified code for the *Derived* class:

```
class Derived :public Base
{
  public:
    using Base::Base;
    Derived(int x, int y, int z)
    {
      cout << "Derived class constructor(
      ↳int,int,int)\n";
    }
    ~Derived()
    {
      cout << "Derived class destructor\n";
    }
    Derived& operator=(const Derived& dr)
    {
      Base::operator=(dr);
      cout << "Derived class assignment\n";
      return *this;
    }
};
```

Now the assignment messages become:

```
Base class assignment
Derived class assignment
```

Activity 22.15

Modified code for *MoreTime* definition:

```
class MoreTime : public dsk::Time
{
  private:
    int millisecond;
  public:
    MoreTime(int=0, int=0, int=0, int=0);
    MoreTime(const MoreTime&);
    void SetMillisecond(int);
    int GetMillisecond() const;
    void SetTime(int, int, int, int);
    MoreTime operator+(const MoreTime&) const;
    bool operator==(const MoreTime&) const;
    bool operator<(const MoreTime&) const;
    friend ostream& operator<<(ostream& ct,
    ↳const MoreTime& t);
};
```

Additional code for new friend function:

```
//*** Displays the value of time t ***
ostream& operator<<(ostream& ct, const MoreTime& t)
{
  char timestr[12];
  ct << static_cast<const dsk::Time &>(t);
  sprintf_s(timestr,":%03d", t.millisecond);
  ct << timestr;
  return ct;
}
```

Modified code for `main()`:

```
int main()
{
  MoreTime mt1(1, 2, 3, 200); //Uses constructor
  MoreTime mt2(mt1);      //Uses Copy constructor
  MoreTime mt3;
  mt3 = mt1;           //Uses assignment operator
  cout << mt1 << endl;
  cout << mt2 << endl;
  cout << mt3 << endl;
}
```

The output is exactly as before.

Activity 22.16

The program code has been supplied in the text. The property settings required are:

C/C++ > General > Additional Include Directories :
E:\Hands On C++\Programs\MyLibraries\ChronoLib

Linker > General > Additional Library Directories :
E:\Hands On C++\Programs\MyLibraries\Debug

Linker > Input > Additional Dependencies :
ChronoLib.lib

The output should show the month first: 11/23/1963

Activity 22.17

The first option

```
int main()
{
  USDate dt1(11, 23, 1963);
  dsk::Date dt2(25,12,2020);
  dt2 = dt1;
  cout << dt2 << endl;
}
```

performs the assignment. However, the second option:

```
int main()
{
  USDate dt1(11, 23, 1963);
  dsk::Date dt2(25,12,2020);
  dt1 = dt2;
  cout << dt2 << endl;
}
```

fails to compile giving the error message

Error C2679 binary '=': no operator found which takes a right-hand operand of type 'dsk::Date' (or there is no acceptable conversion)

Activity 22.18

Modified code for *USDate* class definition:

```
class USDate : public dsk::Date
{
  public:
    USDate(int = 0, int = 0, int = 0);
    USDate(const Date&);
    void SetDate(int, int, int);
    friend ostream& operator<<(ostream&, const
    ↳USDate&);
};
```

Additional code for new constructor:

```
USDate::USDate(const Date& dt)
  :Date(dt.GetDay(), dt.GetMonth(), dt.GetYear())
{}
```

After adding the code for the new constructor the assignment `dt1 = dt2;` now performs correctly with date displayed as

```
12/25/2020
```

Activity 22.19

The program compiles and runs. The output displayed is:

```
23:59:59
00:00:00
```

The value assigned to *ck2* has been rejected and its time set

to the default 0,0,0.

Activity 22.20

As well as adding the code supplied for *SetTime()* we also need to add that method's prototype to the definition of *Clock*:

```
class Clock : public dsk::Time
{
  private:
     using Time::operator-;
  public:
     Clock(int = 0, int = 0, int = 0);
     Clock(const Time&);
     Clock(const Clock&);
     void SetTime(int, int, int);
};
```

Additional code for new method:

```
void Clock::SetTime(int h, int m, int s)
{
  if (h <= 23)
     Time::SetTime(h, m, s);
}
```

The output produced by `main()` remains unchanged.

Activity 22.21

Modified code for *Clock* definition:

```
class Clock : public dsk::Time
{
  public:
     Clock(int=0, int=0, int=0);
     Clock(const Time&);
     Clock(const Clock&);
     void Tick();
};
```

Additional code for new method:

```
void Clock::Tick()
{
   *this = FromSeconds(ToSeconds() + 1);
}
```

The program does not compile because *FromSeconds()* and *ToSeconds()* are declared as private in the *Time* class and are therefore not accessible in the derived *Clock* class.

Activity 22.22

Modified code for *ChronoLib.h* in *MyLibraries*:

```
//*** Time class declaration ***
  class Time
  {
     private:
        int hour;
        int minute;
        int second;
     protected:
        int ToSeconds() const;
        Time FromSeconds(int) const;
     public:
        Time(int = 0, int = 0, int = 0);
        void SetTime(int, int, int);
        int GetHour() const;
        int GetMinute() const;
        int GetSecond() const;
        Time operator+(const Time&) const;
        Time& operator+=(const Time&);
        Time operator-(const Time&) const;
        Time& operator-=(const Time&);
        Time& operator++();
        Time operator++(int);
        Time& operator--();
        Time operator--(int);
        bool operator== (const Time&)const;
        bool operator!= (const Time&)const;
        bool operator<(const Time&)const;
        bool operator<=(const Time&)const;
        bool operator>(const Time&)const;
        bool operator>=(const Time&)const;

        friend std::ostream& operator<<(
        ↳std::ostream&, const Time&);
  };
```

The program now runs as expected.

Activity 22.23

Modified code for *Clock* class definition:

```
class Clock : public dsk::Time
{
  public:
     Clock(int = 0, int = 0, int = 0);
     Clock(const Time&);
     Clock(const Clock&);
     void Tick();
     using Time::ToSeconds;
     using Time::FromSeconds;
};
```

Modified code for `main()`:

```
int main()
{
  Clock ck(23, 10, 59);
  Clock ck2(23, 59, 59);
  cout << ck.ToSeconds() << endl;
  ck.Tick();
  cout << ck.ToSeconds() << endl;
  ck2.Tick();
  cout << ck << endl;
  cout << ck2 << endl;
}
```

The seconds displayed should be 83459 and 83460.

Activity 22.24

Modified code for *Clock* class definition:

```
class Clock : public dsk::Time
{
  private:
     using Time::operator-;
  public:
     Clock(int = 0, int = 0, int = 0);
     Clock(const Time&);
     Clock(const Clock&);
     void SetTime(int, int, int);
     void Tick();
     using Time::ToSeconds;
     using Time::FromSeconds;

};
```

Modified code for `main()`

```
int main()
{
  Clock ck(23, 10, 59);
  Clock ck2(23, 59, 59);
  ck = ck2 - ck;
  cout << ck << endl;
  cout << ck2 << endl;
}
```

The compiler will highlight an error at the line using the subtraction operator.

Activity 22.25

Modified code for *Clock* class definition:

```
class Clock : public dsk::Time
{
  private:
     using Time::operator-;
  public:
     Clock(int = 0, int = 0, int = 0);
     Clock(const Time&);
     Clock(const Clock&);
```

```
    void SetTime(int, int, int);
    void Tick();
    using Time::ToSeconds;
    using Time::FromSeconds;
    Time operator+(const Time&) = delete;
};
```

Modified code for `main()`

```
int main()
{
  Clock ck(23, 10, 59);
  Clock ck2(23, 59, 59);
  ck = ck2 + ck;
  cout << ck << endl;
  cout << ck2 << endl;
}
```

The compiler will highlight an error at the use of the addition operator.

Activity 22.26

The *Clock* class definition is changed to

```
class Clock : public dsk::Time
{
  private:
    using Time::operator-;
  public:
    Clock(int = 0, int = 0, int = 0);
    Clock(const Time&);
    Clock(const Clock&);
    void SetTime(int, int, int);
    void Tick();
    using Time::ToSeconds;
    Time operator+(const Time&) = delete;
    Time FromSeconds(int) const;
};
```

This time it's probably easier just to rewrite the method rather than call the old one to do part of the work:

```
dsk::Time Clock::FromSeconds(int secs) const
{
  if (secs < 0)
    return Time(0, 0, 0);
  int h = (secs / 3600)%24;
  int m = (secs - h * 3600) / 60;
  int s = secs % 60;

  return Time(h, m, s);
}
```

If we restore the call to *Tick()* for each object in `main()`:

```
int main()
{
  Clock ck(23, 10, 59);
  Clock ck2(23, 59, 59);
  ck.Tick();
  ck2.Tick();
  cout << ck << endl;
  cout << ck2 << endl;
}
```

We get the same results as the first time we used *Tick()* – but this time for the correct reasons!

Activity 22.27

The *Clock* object time changes to 00:00:00.

The *CountdownClock* object changes to 23:59:58.

Activity 22.28

Code for *CalClock*:

```
class CalClock : public dsk::Time, public dsk::Date
{
  public:
    CalClock() = default;
    CalClock(const Date&, const Time&);
    CalClock(const CalClock&);
    void SetCalClock(const Date&, const Time&);
    dsk::Time GetTime() const;
    dsk::Date GetDate() const;
    void Tick();

    friend ostream& operator<<(ostream&, const
CalClock&);
};


CalClock::CalClock(const Date& dt, const Time& tm)
  :Date(dt),Time(tm)
{}

CalClock::CalClock(const CalClock&)
  :dsk::Time(GetTime()), dsk::Date(GetDate())
{}

void CalClock::SetCalClock(const Date& dt, const
Time& tm)
{
  SetDate(dt.GetDay(), dt.GetMonth(), dt.GetYear());
  SetTime(tm.GetHour(), tm.GetMinute(),
tm.GetSecond());
}

dsk::Time CalClock::GetTime() const
{
  return dsk::Time(GetHour(), GetMinute(),
GetSecond());
}

dsk::Date CalClock::GetDate() const
{
  return dsk::Date(GetDay(), GetMonth(), GetYear());
}

void CalClock::Tick()
{
  int totalsecs = GetHour()*3600 + GetMinute()*60 +
GetSecond() + 1;
  int h = (totalsecs / 3600);
  int m = (totalsecs - h * 3600) / 60;
  int s = totalsecs % 60;
  h = h % 24;
  SetTime(h, m, s);
  if (GetTime() == Time(0, 0, 0))
  {
    dsk::Date newdate = FromJDN(ToJDN()+1);
    SetDate(newdate.GetDay(), newdate.GetMonth(),
newdate.GetYear());
  }
}

ostream& operator<<(ostream& os, const CalClock& cc)
{
  os << cc.GetDate() << "   " << cc.GetTime();
  return os;
}
```

Code for `main()`:

```
int main()
{
  CalClock cck(dsk::Date(1, 1, 2021), dsk::Time(23,
  59, 59));
  cck.Tick();
  cout << cck << endl;
}
```

The above code produces the following output:

2/1/2021 00:00:00

Activity 22.29

The call to `var.f1()` in `main()` is flagged as ambiguous. This is because there are two copies of *f1()* within the object.

Activity 22.30

By changing the code in `main()` to

```
int main()
{
  D var;
  var.B::f1();
}
```

The ambiguity is removed and the program compiles.

Activity 22.31

Modified code *MultiInherit*:

```
#include <iostream>
using namespace std;

class A
{
  private:
    int x;
  public:
    A()
    {
      cout << "A class constructor called\n";
    }
    void f1()
    {
      cout << "Executed f1()\n";
    }
};

class B : public A
{
  private:
    int y;
  public:
    B()
    {
      cout << "B class constructor called\n";
    }
    void f2()
    {
      cout << "Executed f2()\n";
    }
};

class C :  public A
{
  private:
    int z;
  public:
    C()
    {
      cout << "C class constructor called\n";
    }
    void f3()
    {
      cout << "Executed f3()\n";
    }
};

class D : public B, public C
{
  public:
    D()
    {
      cout << "D class constructor called\n";
    }
};



int main()
{
  D var;
  var.B::f1();
}
```

The output produced is:

```
A class constructor called
B class constructor called
A class constructor called
C class constructor called
D class constructor called
Executed f1()
```

23

Class Relationships - 2

In this Chapter:

- ❑ Pointers to Objects
- ❑ Virtual Methods
- ❑ typeid()
- ❑ Abstract Classes
- ❑ Aggregation and Composition
- ❑ Exceptions and Derived Classes

Pointers to Objects

Pointer Review

Previously we covered the concept of pointers – elements that hold the address of a data item. We also saw that the original pointers offered by C++ did little work for us since we had to create code to both allocate and deallocate any space dynamically reserved. These original pointers are now referred to as **raw pointers**.

More recent versions of C++ introduced the concept of **smart pointers**. Smart pointers automatically handle the deallocation of memory space.

Unique smart pointers ensured that only one pointer can reference an allocated space. The referenced space is deallocated automatically when the pointer no longer exists.

Shared smart pointers allow multiple pointers to reference the same memory space. That space is deallocated only after the last pointer which references it is destroyed.

Object Pointers



dsk is the namespace in which *Time* is defined.

We could create a raw pointer for a *Time* object with the line

```
dsk::Time *ptr1;
```

and have it allocate space for the object with the line

```
ptr1 = new dsk::Time();
```

Note that this involves a call to the *Time* class constructor (using the default initialisation values). Where a zero-argument constructor is being called, we can omit the parentheses:

```
ptr1 = new dsk::Time;
```

If we wanted to assign a specific initial value we could expand the previous line to



This version uses the three parameter version of the *Time* class's constructor.

```
ptr1 = new dsk::Time(12,23,0);
```

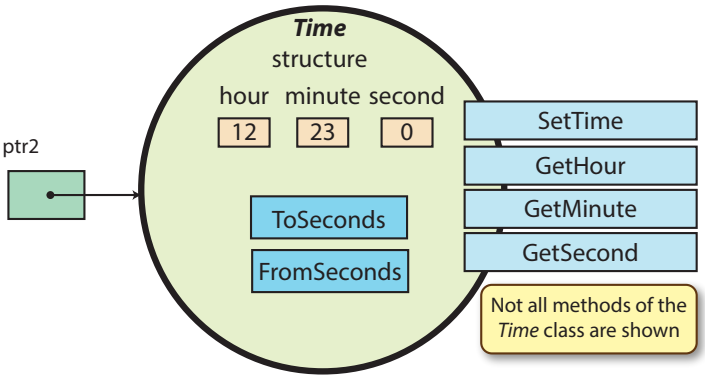
Using a unique smart pointer we could create the pointer and its associated object with the line

```
unique_ptr<dsk::Time> ptr2(make_unique<dsk::Time>(12,23,0));
```

FIG-23.1 represents what is created by the line above.

FIG-23.1

Pointer to a *Time* Object



To access a public method of the object, we need to dereference the pointer. For example,

```
(*ptr1)++;                          //Add one second
cout << ptr2->GetHour() << endl;//Display hour value
```

Activity 23.1

In *TestClock*, remove the existing code in `main()` and add new code to create *ptr1* and *ptr2* as shown above, setting the time in both to 12:23:00.

Add one second to the object referenced by *ptr1* and display the contents of both objects.

Not only can a pointer reference an object of the class for which it was specified (the *Time* class in the case of *ptr1* and *ptr2*), but it can also reference an object of any class derived from the specified class. For example, our *Time* pointers, *ptr1* and *ptr2* in the above example, can be made to reference *Clock* and *CountdownClock* objects:

```
dsk::Time *ptr1 = new Clock(11, 2, 0);
unique_ptr<dsk::Time>ptr2(make_unique <CountdownClock>
(11,2,0));
```

Activity 23.2

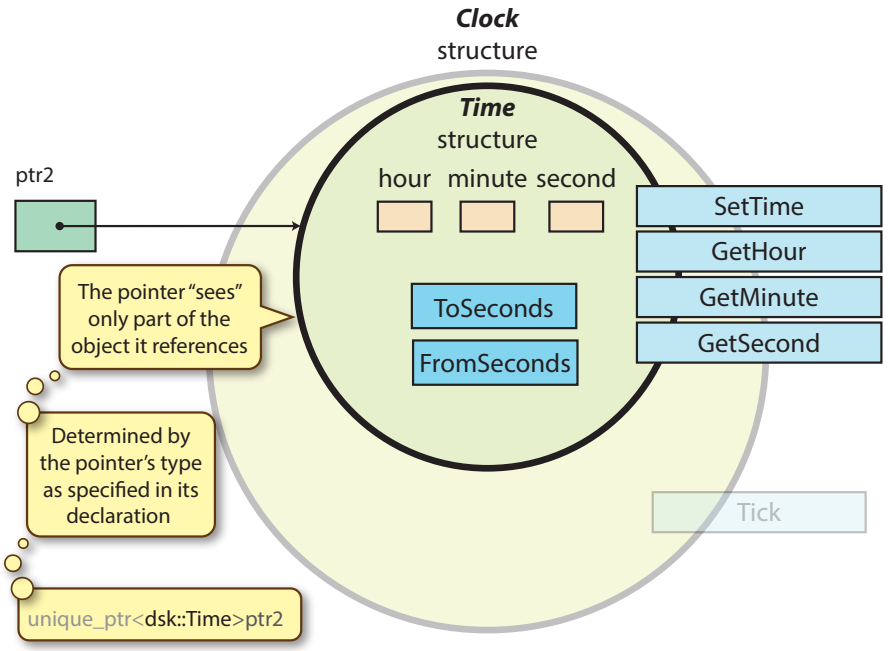
Modify *TestClock* so that *ptr1* and *ptr2* reference *Clock* and *CountdownClock* objects respectively. Both times should be set to 11:02:00.

Remove the `(*ptr1)++;` line and replace it with calls to *Tick()* for both objects. What is the result of this change?

Unfortunately, when a pointer is declared as referencing one class but is then used to reference a derived class, a few problems arise. The first of these is that any methods not mentioned in the original class for which the pointer was defined are not visible when the pointer is dereferenced. This is why attempting to call *Tick()* – which is not mentioned in the *Time* class – causes a compilation error. FIG-23.2 visualises the problem.

FIG-23.2

A Pointer's Limited View



However, we can overcome this problem by casting the pointer to the appropriate type:

```
static_cast<Clock*>(ptr1)->Tick();
```

Unfortunately, unique pointers cannot be cast in this way since such a cast would create a temporary second pointer to the *Clock* object. This would violate the rules of a unique pointer where only one pointer can reference the object at any moment in time. So we cannot use this approach to have *ptr2* execute the *CountdownClock*'s *Tick()* method.

Activity 23.3

In project *TestClock* cast *ptr1* to a *Clock* pointer and remove the *ptr2->Tick(); line*. How does this affect the program?

Let's try a different approach to the problem. Since neither *ptr1* nor *ptr2* actually reference a *Time* object, we could make them pointers to the *Clock* class:

```
Clock* ptr1 = new Clock(11, 2, 0);
unique_ptr<Clock> ptr2(make_unique <CountdownClock>(11,2,0));
```

Now *ptr1* is a *Clock* pointer that actually points to a *Clock* object and *ptr2* is a *Clock* pointer that points to a *CountdownClock* object.

Activity 23.4

In project *TestClock* rewrite `main()` so that both pointers are designed to reference a *Clock* object (with *ptr1* actually referencing a *Clock* object and *ptr2* referencing a *CountdownClock* object).

Set the time in both objects to 11:02:00 and call *Tick()* for each. What time is displayed by each clock?

The problem is that the program isn't smart enough to realise that, since *ptr2* is actually referencing a *CountdownClock* object, it is that class's version of *Tick()* that should be executed. Instead, it uses the *Clock* class version of *Tick()* because the pointer was originally declared as a pointer to that class.

Virtual Methods

virtual

Luckily, we can solve the problem of selecting the appropriate version of a method when using pointers by making a small adjustment to the code for the *Clock* class. In the class declaration, we need to change the prototype for *Tick()* to

```
virtual void Tick();
```

Activity 23.5

In the code for class *Clock*, add the word `virtual` to the *Tick()* method's prototype.

Compile and run your *TestClock* project again. What time is displayed for each clock this time?

Notice that there is no need to add the term `virtual` to the prototype for the new version of *Tick()* declared in the *CountdownClock* class. Once a method is declared as virtual, any new versions of the method defined in derived classes are also assumed to be virtual.

However, this only works if the methods in both classes have exactly the same signature. In this case, that means a method which returns `void`, has no parameters and is not a `const` method. If the new method had the same name but a different signature, then the method in the base class is hidden from the derived class.

Creating a new method with the same heading as a base class virtual function is known as **overriding** the inherited function.

To help the compiler realise that the second version of *Tick()* (in the *CountdownClock* class) is meant to override the one given in the *Clock* class, we may add the term `override` to the prototype in the *CountdownClass* definition:

```
class CountdownClock : public Clock
{
   public:
      CountdownClock(int = 0, int = 0, int = 0);
      CountdownClock(const Time&);
      void Tick() override;
};
```

This addition will make no difference to the execution of our program, but it does allow the compiler to signal an error if the signature of this new method does not match that of the corresponding method in the base class. For example, if the *CountdownClass* had the *Tick()* prototype declared as

```
void Tick(int) override;
```

the compiler would issue an error message such as

> Error C3668 'CountdownClock::Tick': method with override specifier 'override' did not override any base class methods

The same error message would occur if the term `virtual` was missing from the method's prototype in the base class.

Activity 23.6

In the code for class *Clock*, remove the word `virtual` to the *Tick()* method's prototype.

Add the word `override` to the *Tick()* prototype in *CountdownClock* class and check for the compilation error.

Restore the word `virtual` and ensure that the program again compiles and runs.

A Little of How Virtual Methods Work

So what is happening when we add the term `virtual` to a function prototype? Normally, the version of a method to be called is decided at compile time and that version is determined by the class type specified when the pointer is declared. This is known as **early binding**.

But when we add `virtual` to the function prototype, the version of the method is not determined until run time and is based on the class of the actual object referenced by the pointer. This is known as **late binding**. Since our second pointer references a *CountdownClock* object, it is that class's version of *Tick()* that is executed.

Because the version of a virtual method is determined while the program is running there is a slight time penalty involved, but only in the most specialised programs is this likely to be significant.

This use of late binding for any method that is declared as virtual extends to all situations. To highlight this, we'll return to the code for the *Clock* class and add a new method, *DisplayTicking()*, which displays the current time over several seconds, allowing us to watch the time change. The code for the method is



C++ has a function called `clock()` that gets the time units that have passed since execution started. Requires *#include<time.h>*



The actual time between calls to *Tick()* may not be one second, but that is not of consequence to the property being demonstrated.

```
//*** Display ticking clock ***
void Clock::DisplayTicking(int no)
{
   //*** Record the current time ***
   long now = clock();
   //*** Perform no ticks ***
   for (int c = 1; c <= no; c++)
   {
      //*** Display current time ***
      cout << "        \r" << *this << '\r';
      //*** Wait one second ***
      while (clock() - now < 1000);
      //*** Record the current time ***
      now = clock();
      //*** Change time by one second ***
      Tick();
   }
   //*** Display current time ***
   cout << "        \r" << *this << '\r';
}
```

where parameter *no* is the number of seconds over which the time is viewed.

The method makes use of C++'s `clock()` function to check for the passing of one second and then calls the *Tick()* to update the time held in the object. The other point to note is that the `cout` statement ends with the escape character `'\r'` which returns the cursor to the start of the current line. The spaces at the start of the `cout` statement are used to erase any existing characters on the display line allowing all the values being output to appear at the identical position on the screen.

Activity 23.7

In *TestClock*, add the line `#include<time.h>`.

Add a prototype for *DisplayTicking()* in the *Clock* class definition followed by the method's code.

In `main()` change the calls to *Tick()* to calls to *DisplayTicking()* – set the parameter to 10 – and observe the results.

We can see from Activity 23.7 that no matter where *Tick()* is called from, C++ will determine the type of object involved and execute the version of *Tick()* appropriate to that object.

The term **polymorphism** is used to describe the ability of an object to pick the method appropriate to its own class.

final

Should we want to stop future derived classes from overriding a virtual function we can add the term `final` to the end of the function prototype. For example, let say we don't want any class derived from the *CountdownClock* class to create a new version of *Tick()*, then we would write the prototype as

```
void Tick() override final;
```

If the prototype uses the terms `override` and `final`, these may appear in any order.

With the `final` term added, should we attempt something like

```
class AtomicClock : public CountdownClock
{
   ...
   void Tick(); // Not allowed
   ...
```

the compiler would flag an error such as

> Error C3248 'CountdownClock::Tick': function declared as 'final' cannot be overridden by 'AtomicClock::Tick'

The term `final` can only be added to virtual functions.

The Down Side to Referencing a Derived Class

Of course, being able to cast a pointer to the appropriate derived class depends on the programmer knowing which class the referenced object belongs to. This is not always possible at the time the program is being written. The code below allows the user to choose which type of object, *Clock* or *CountdownClock*, the pointer *ckptr* is to reference:

```
Time *ckptr;
char ans;
//*** Select class of object ***
cout << "Do you want a countdown clock? (Y/N): ";
cin >> ans;
if (tolower(ans) == 'y')
   ckptr = new CountdownClock(12,0,0);
else
   ckptr = new Clock(12,0,0);
```

Without knowledge of the type of object that has been created, the programmer no longer knows if the dereferenced pointer should be cast to a *Clock* or *CountdownClock* type.

typeid()

Luckily, C++ supplies us with a way of find out the class of an object using the function `typeid()`. This function takes an argument of an object pointer or class name. Examples are:

```
typeid(*ckptr)
typeid(Clock)
```

An object of class `type_info` (a class defined within C++) is returned by `typeid()`

that contains information about the class referenced by the parameter.

The `type_info` class has three useful operations; these are :

`int operator==(const info_type& t)`
: returns *true* if the two *info_type* objects give details about the same class.

`int operator!=(const info_type& t)`
: returns *true* if the two *info_type* objects do not give details about the same class.

`const char* name()`
: returns the name of the class. The string returned is of the form: "class *namespace*::*class name*". For example, "class dsk::Time". *namespace::* is omitted if the class is defined in the current namespace.

These operations allow us to use statements such as

```
if(typeid(*dt1) == typeid(Clock))
```

or

```
cout << "Object is of class "<<typeid(*dt1).GetHour()<<endl;
```

The `main()` function in FIG-23.3 is used within the *TestClock* project to create a *Clock* or *CountdownClock* object and displays the object's class name.

FIG-23.3

Class Determined at Runtime

```
int main()
{
   Clock* ptr1 = new Clock(11, 2, 0);
   char ans;   // Holds user response to question

   //*** Select clock type ***
   cout << "Do you want a countdown clock ? (Y / N) : ";
   cin >> ans;
   if (tolower(ans) == 'y')
      ptr1 = new CountdownClock(12, 0, 0);
   else
      ptr1 = new Clock(12, 0, 0);
   //*** Display name of class created ***
   cout << "Object's class is " << typeid(*ptr1).name() << endl;
}
```

Activity 23.8

Modify `main()` in project *TestClock* to match the code given above.

Run your project twice, creating different class objects on each occasion.

Unfortunately, `typeid()` has a major limitation when it comes to detecting the class of a reference object.

The problem with `typeid()` is that it can only detect the class of referenced objects accurately if the base class contains at least one virtual function. The *Clock* class's

Tick() function is declared as virtual and hence `typeid()` can detect the correct class both *Clock* and *CountdownClock* objects. However, the *Time* class has no virtual functions and therefore the correct class cannot be detected.

Virtual Classes

In the previous chapter we discovered multiple inheritance's diamond problem where class *A* was the base class for classes *B* and *C,* and class *D* was derived from *B* and *C*. This resulted in class *D* having two copies of class *A*'s features (see Activities 22.29 and 22.30).

To eliminate this duplication we have to make *A* a virtual base class for classes *B* and *C* by adding the term `virtual` in the first line of each classes definition:

```
class B : virtual public A
{
   ...

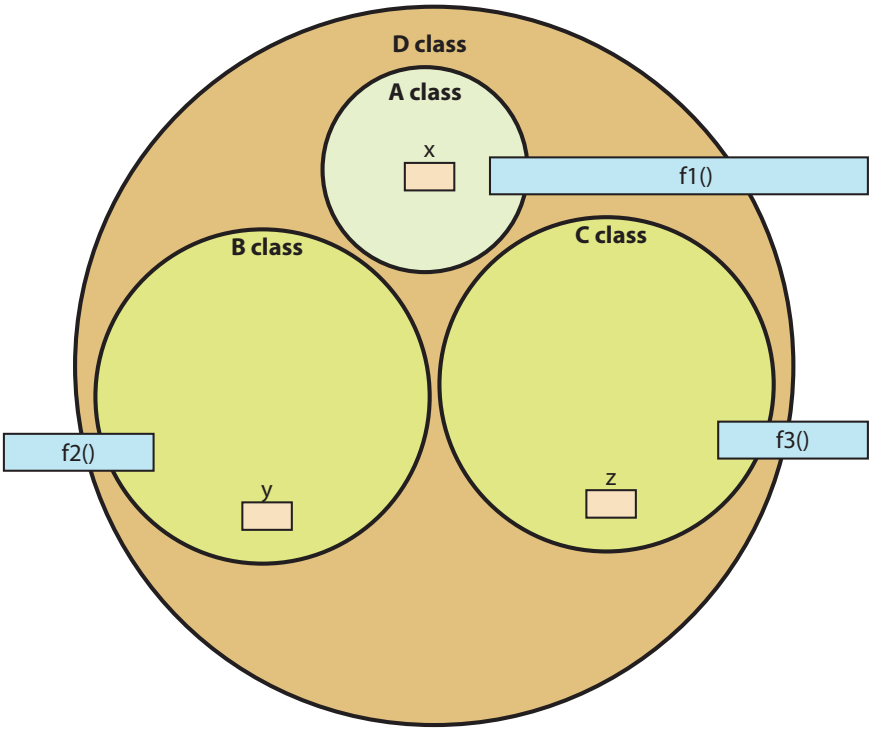
class C : virtual public A
{
   ...
```

No other changes are required.

Now, if an application program creates a *B* or *C* class object they will contain an embedded *A* class object just as before. But when we create a *D* class object it will contain only a single copy of the *A* class features (see FIG-23.4).

FIG-23.4

Virtual Classes Solve the Diamond Problem



Now, since there is only one copy of *f1()*, there's no ambiguity when it is called by the *D* class object:

```
var.f1();   //OK
```

Activity 23.9

Reload your *MultiInherit* project and add the term `virtual` to classes *B* and *C* as shown above.

Adjust the call to *f1()* in `main()` and test the program to see which constructors are executed.

Looking again at the diagram above, we can see that the features of class *A* embedded in class *D* are separate from the features of classes *B* and *C*. This means that it is class *D* that has to call the constructor of class *A*. Of course, since that constructor has zero arguments the call is created automatically by the C++ compiler.

But should the constructor require parameters, then an explicit call to the constructor must be included in the class *D* constructor. For example, let's redefine class *A* as

```
class A
{
   private:
      int x;
   public:
      A()
      {
         cout << "A class constructor called\n";
      }
      A(int v)
      {
         x = v;
         cout << "A class constructor x set to " << x << '\n';
      }
      void f1()
      {
         cout << "Executed f1()\n";
      }
};
```

Then the constructor for *D* would need an explicit call when that version of *A*'s constructor was to be used:

```
class D : public B, public C
{
   public:
      D()
      {
         cout << "D class constructor called\n";
      }

      D(int m)
      : A(m)       //Call A's parameterised constructor
      {
         cout << "D class 2nd constructor\n";
      }
};
```

Activity 23.10

In *MultiInherit* modify class *A* and *D* as shown above.

Modify the creation of *var* in `main()` to use the new constructor in *D* and observe the output when the program is run.

Note that if classes *B* and *C* also had constructors which required parameters, then *D*'s new constructor would be of the form:

```
D(int m, int n, int p)
      : A(m),B(n),C(p)
{}
```

with each of the base class's constructors explicitly called.

Summary

- An object pointer can be used to reference an object of the specified class or any class derived from the specified class.
- An object pointer must be dereferenced to access the public features of the object it points to.
- A pointer designed to reference a specified base class which is then used to point to a derived class object cannot access any public features of that object not named in the base class.
- To access the additional features named in the derived class, a pointer must be cast to that class.

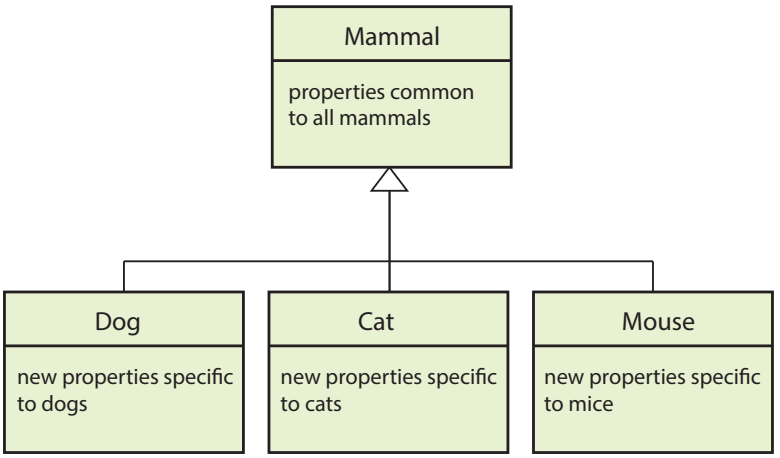
Abstract Classes

Introduction

Dogs, cats and mice are different creatures but we can give them a commonality by saying that they are all types of mammals. If we were to draw this relationship in a class diagram we might end up with the diagram shown in FIG-23.5.

FIG-23.5

Abstract Class Concept



In real life no animal is classified only as a mammal; it's either a dog, cat or mouse (or some other animal). *Mammal* is a class which has been created simply to bring together all the common features of more specific animal types.

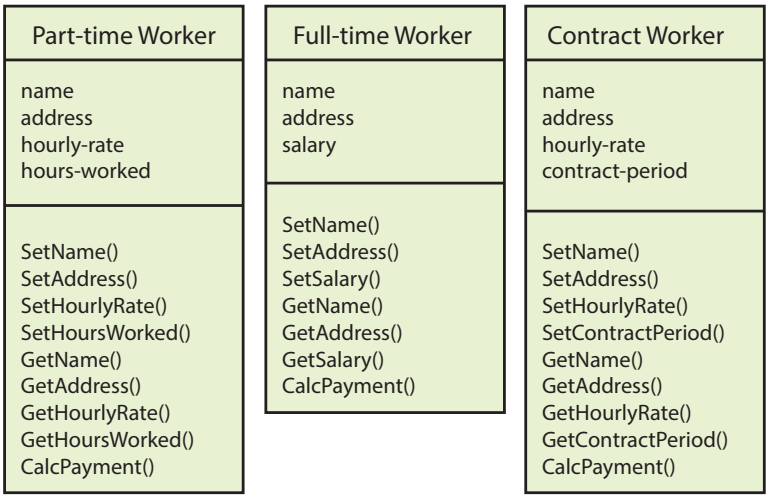
In programming we would refer to the *Mammal* class as an **abstract class** - a class created only to link other classes together by specifying their common features.

Creating an Abstract Class

Let's look at a more program-oriented example of the use of abstract classes. Imagine a computerised wages system which recognises three classes of workers as shown in FIG-23.6.

FIG-23.6

Working Classes

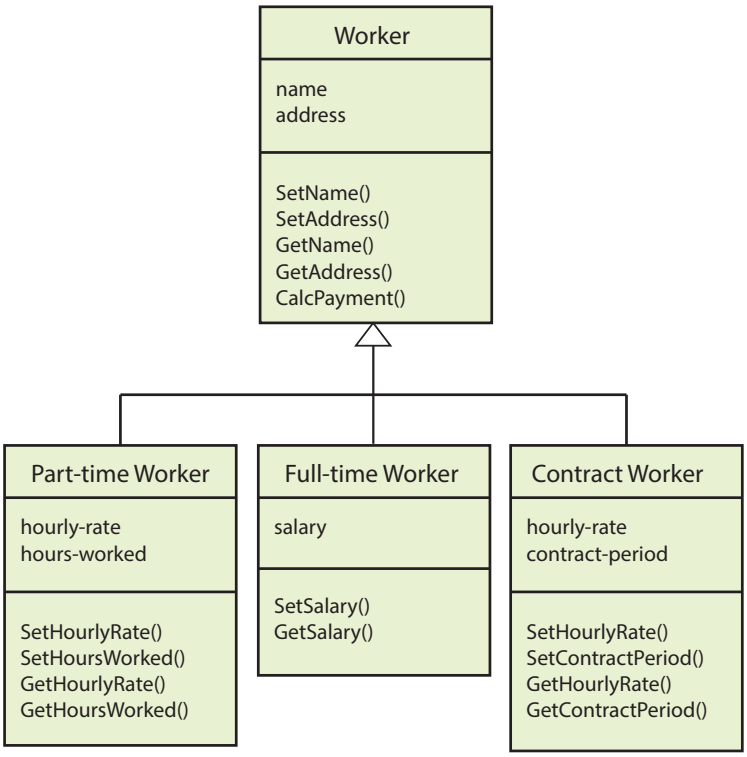


Common sense and a look at the class diagrams easily tells us that there is a relationship between these three separate classes: they are all types of worker and they share many common properties.

To produce a relationship between the three classes we can create an artificial class called *Worker* in which we place the properties that are common to all three of the original classes which then become derived classes (see FIG-23.7).

FIG-23.7

Extracting Properties into an Abstract Class



The new *Worker* class has been created solely to produce a common base class for the three other classes. No actual employee of the company will be classified as a *Worker*, but rather as a *Part-time Worker*, *Full-time Worker* or *Contract Worker*. Because the *Worker* class will never be used on its own to create a *Worker* class object, it is known as an **abstract class**. The classes from which we intend to create objects are known as **concrete classes**.

Coding an Abstract Class

To code an abstract class in C++ we begin in the standard way, declaring the features of the class:

```
class Worker
{
   private:
      string name;
      string address;
   public:
      void SetName(const string&);
      void SetAddress(const string&);
      string GetName() const;
      string GetAddress() const;
      double CalcPayment() const;
   };
```

If the code for any of the operations in the class is going to be identical in all of the descendent classes, then we can also code these here as part of the *Worker* class:

```
//*** Set worker's name ***
void Worker::SetName(const string& n)
{
   name = n;
}
//*** Set worker's address ***
void Worker::SetAddress(const string& ad)
{
   address = ad;
}

//*** Get worker's name ***
string Worker::GetName() const
{
   return name;
}

//*** Get worker's address ***
string Worker::GetAddress() const
{
   return address;
}
```

The last operation that all of the concrete classes have in common is *CalcPayment()*, but in this system the monthly wage for each type of worker is calculated differently:

Part-time workers:	first 160 hours at hourly rate, additional hours at hourly rate + 50%
Full-time workers:	salary / 12
Contract workers:	160 hours at hourly rate (no overtime)

Even though we cannot code *CalcPayment()* in the *Worker* class, nevertheless we can add a prototype for it in the class declaration (as we have already done!). Because we know the method will be overridden in each of the derived classes, we should add the word `virtual` at the start of the prototype:

```
class Worker
{
   private:
      string name;
      string address;
   public:
      void SetName(const string&);
      void SetAddress(const string&);
      string GetName() const;
      string GetAddress() const;
      virtual double CalcPayment()const;
};
```

However, we have not yet created an abstract class. If we were to leave things as they are, then the application programmer would be free to declare an object of type *Worker*:

```
Worker empl;
```

A class is considered abstract (and hence objects of that class cannot be created) when at least one virtual function within the class is a **pure virtual function**.

To declare a pure virtual function we need only add `=0` to the end of that function's prototype and make no attempt to code that method within the class:

```
class Worker
{
   private:
      string name;
      string address;
   public:
      void SetName(const string&);
      void SetAddress(const string&);
      string GetName() const;
      string GetAddress() const;
      virtual double CalcPayment() const = 0;
};
```

With a pure virtual function in the class declaration, any attempt by the application programmer to create a *Worker* object will result in a compilation error.

Activity 23.11

Start a new project called *TestWorkers* and create a *.cpp* file containing the code for the *Worker* class, adding the appropriate `#include` and `namespace` statements.

Add an empty `main()` function and compile your code.

In `main()` declare a *Worker* object, *w*, and attempt to recompile your project.

Remove the declaration of *w* and save your project.

Having coded the *Worker* class we can now add the code for the other three classes. The *PartTimeWorker* class requires the following code:

```
class PartTimeWorker : public Worker
{
   private:
      double hourlyrate;
      double hoursworked;
   public:
      void SetHourlyRate(double);
      void SetHoursWorked(double);
      double GetHourlyRate() const;
      double GetHoursWorked() const;
      double CalcPayment() const;
};

//*** Set PTworker's hourly rate ***
void PartTimeWorker::SetHourlyRate(double r)
{
   if (r > 0 && r < 200)
      hourlyrate = r;
}

//*** Set PTworker's hours worked ***
void PartTimeWorker::SetHoursWorked(double hw)
{
   if (hw >= 0 && hw <= 200)
      hoursworked = hw;
}
//*** Get PTworker's hourly rate ***
```

```
double PartTimeWorker::GetHourlyRate() const
{
   return hourlyrate;
}

//*** Get PTworker's hours worked ***
double PartTimeWorker::GetHoursWorked() const
{
   return hoursworked;
}

//*** Calculate payment due for month ***
double PartTimeWorker::CalcPayment() const
{
   //*** calculate basic pay ***
   float moneydue = hoursworked * hourlyrate;
   //*** Add 50% for any hours over 160 ***
   if (hoursworked > 160)
      moneydue += (hoursworked - 160) * hourlyrate * 0.5;
   return moneydue;
}
```

Notice that the pure virtual function, *CalcPayment()*, has been overridden and coded in this class. This is necessary to create a **concrete class** rather than yet another abstract class.

Activity 23.12

In *TestWorkers* add the code for the *PartTimeWorker* class and check that the program still compiles. Is it possible to create a *PartTimeWorker* object in `main()`?

The code for *FullTimeWorker* is:

```
class FullTimeWorker : public Worker
{
   private:
      double salary;
   public:
      void SetSalary(double);
      double GetSalary() const;
      double CalcPayment() const;
};

//*** Set FTWorker's salary ***
void FullTimeWorker::SetSalary(double s)
{
   if (s > 0)
      salary = s;
}

//*** Get FTworker's salary ***
double FullTimeWorker::GetSalary() const
{
   return salary;
}

//*** Calculate payment due for month ***
double FullTimeWorker::CalcPayment() const
{
   return salary / 12;
}
```

And for *ContractWorker*:

```
class ContractWorker : public Worker
{
   private:
      double hourlyrate;
      int contractperiod;
   public:
      void SetHourlyRate(double);
      void SetContractPeriod(int);
      double GetHourlyRate() const;
      int GetContractPeriod() const;
      double CalcPayment() const;
};

//*** Set Contractworker's hourly rate ***
void ContractWorker::SetHourlyRate(double r)
{
   if (r > 0 && r < 500)
      hourlyrate = r;
}

//*** Set Contract worker's contract period ***
void ContractWorker::SetContractPeriod(int p)
{
   if (p > 0)
      contractperiod = p;
}


//*** Get Contract worker's hourly rate ***
double ContractWorker::GetHourlyRate() const
{
   return hourlyrate;
}


//*** Get Contract worker's contract period ***
int ContractWorker::GetContractPeriod() const
{
   return contractperiod;
}

//*** Calculate payment due for month ***
double ContractWorker::CalcPayment() const
{
   return 160 * hourlyrate;
}
```

Activity 23.13

In *TestWorkers* add the code for the *FullTimeWorker* and *ContractWorker* classes and check that the program still compiles.

Pointers to Abstract Classes

Although we cannot create objects of an abstract class, we are free to create a pointer designed to reference an abstract class object. For example, we may write

```
Worker *wptr;
```

Of course, we can never attempt to create an object of this abstract class with a line such as

```
wptr = new Worker;
```

which would create a compilation error, but we can have the pointer reference an object of a descendant concrete class:

```
wptr = new PartTimeWorker;
```

We can use the pointer directly to access any public methods of the class which were named in the *Worker* class hence the following are valid expressions:

```
wptr->SetName("Estella Havisham");
wptr->SetAddress("Satis House\nRochester\nKent");
cout << "Wage due: "<< wptr->CalcPayment();
```

but the pointer would have to be cast to a *PartTimeWorker* pointer in order to access methods named only in that class:

```
static_cast<PartTimeWorker*>(wptr)->SetHourlyRate(20.80);
static_cast<PartTimeWorker*>(wptr)->SetHoursWorked(200);
```

Activity 23.14

In *TestWorkers* create a `main()` function in which a *Worker* pointer, *wptr*, is declared and have this reference a *PartTimeWorker* object containing the details shown in the coding above.

Display the name address and wage due for the worker.

Modify your code so that the pointer references a *FullTimeWorker* on a salary of 72,000 a year and check that the amount due is correct.

Let's assume that our workers require a new method, *CalcDeductions*(), to calculate and return any deductions to be made from their income. The deductions are as follows:

Part-time workers:	Income tax: 20% of gross
Full-time workers:	Income tax: 20% of gross; Health Insurance: 11% of gross
Contract workers:	No deductions (self-employed)

Activity 23.15

In *TestWorkers* add the code for a new virtual method called *CalcDeductions()* in all four classes and check that each executable version of the method correctly calculates the deductions due.

Use the following test data:

FullTimeWorker	salary:	72,000
PartTimeWorker	hourly rate :	20.80
	hours worked:	200
ContractWorker	hourly rate:	65.85

Calling Uncoded Virtual Methods in an Abstract Class

A final method, *CalcNet()*, calculates the net amount due using the formula:

net wage = gross wage - deductions

Our first instinct might be that we will require four different versions of this routine just as we did with *CalcDeductions()*. However, the net wage calculation is identical for all classes – only how the gross wage and deductions are calculated are different in each case – so we actually only require one version of the new method to be placed in the abstract *Worker* class. The new method is coded as:

```
double Worker::CalcNet()
{
   return (CalcPayment() - CalcDeductions());
}
```

At first glance a *Worker* class method seems to be calling two methods which are not actually coded in that class. But, since *CalcPayment()* and *CalcDeductions()* are virtual methods and coded versions are available in all three concrete classes, the appropriate version of these methods will be called when *CalcNet()* is executed.

Activity 23.16

In *TestWorkers* add the code for the method *CalcNet()* and ensure that it produces the correct result for all three worker types.

Arrays of Abstract Class Pointers

When we first met arrays, we saw that one of their limitations was that a single array could hold only values of the specified type. Hence, if the array was set up to hold `int` values it could not store any other type of value such as a `float` or `char` value.

With an array of pointers we can, at least partially, overcome this limitation. By starting with the line

```
Worker *employees[3];
```

we can store any combination of *PartTimeWorker*, *FullTimeWorker* and *ContractWorker* :

```
employees[0] = new PartTimeWorker;
employees[1] = new FullTimeWorker;
employees[2] = new ContractWorker;
```

A slightly better option would be to use an `array` object to store our workers' details:

```
array<Worker*, 3> employees{ nullptr };
employees[0] = new PartTimeWorker;
employees[1] = new FullTimeWorker;
employees[2] = new ContractWorker;
```

And the best option – which would allow for a change in the number of employees – would be to use a `vector` object:

```
vector<Worker*> employees(3);
```

The program given in FIG-23.8 creates a vector containing three *Worker* pointers and then adds the data for each worker before displaying those details.

FIG-23.8

Using the *Worker* Classes

```
#include <iostream>
#include <vector>
#include <time.h>

using namespace std;
```



FIG-23.8
(continued)

Using the *Worker* Classes



The code for the existing *Worker* classes has been omitted from this listing to save space.

```
//*********************************************
//***             Worker Class              ***
//*********************************************
//code goes here
//*********************************************
//***          PartTimeWorker Class         ***
//*********************************************
//code goes here
//*********************************************
//***          FullTimeWorker Class         ***
//*********************************************
//code goes here
//*********************************************
//***          ContractWorker Class         ***
//*********************************************
//code goes here
//*********************************************
//***             Main program              ***
//*********************************************
int main()
{
   //*** Names and addresses of employees ***
   string names[] = { "Philip Pirrip","Estella Havisham",
   ↳"Herbert Pocket" };
   string addresses[] = { "The Forge\n\t\t\tThe Marshes\n\t\t
   ↳\tKent","Satis House\n\t\t\tRochester\n\t\t\tKent",
   ↳"Barnard's Inn\n\t\t\tLondon" };

   //*** Workers vector ***
   vector<Worker*> employees(3);
   employees[0] = new PartTimeWorker;
   employees[1] = new FullTimeWorker;
   employees[2] = new ContractWorker;
   //*** Add details for all employees ***
   int index = 0;
   srand(time(nullptr));
   for (auto wptr : employees)
   {
      wptr->SetName(names[index]);
      wptr->SetAddress(addresses[index]);
      if (typeid(*wptr) == typeid(PartTimeWorker))
      {
         static_cast<PartTimeWorker*>(wptr)->
         ↳SetHourlyRate(rand()%50);
         static_cast<PartTimeWorker*>(wptr)->SetHoursWorked(rand()
         ↳%100 + 100);
      }
      else if (typeid(*wptr) == typeid(FullTimeWorker))
         static_cast<FullTimeWorker*>(wptr)->SetSalary(rand()
         ↳%30000 + 50000);
      else
         static_cast<ContractWorker*>(wptr)->SetHourlyRate(rand()
         ↳%100 + 30);
      index++;
   }
   //*** Display details of all employees ***
   for (auto wptr : employees)
   {
      cout << "\n\nEmployee's name:\t" << wptr->GetName() << endl;
      cout << "Address:\t\t" << wptr->GetAddress() << endl;
```



FIG-23.8
(continued)

Using the *Worker* Classes

```
        cout << "Wage due :\t\t" << wptr->CalcPayment() << endl;
        cout << "Deductions :\t\t" << wptr->CalcDeductions() << endl;
        cout << "NET wage:\t\t" << wptr->CalcNet() << endl;
    }
}
```

Notice that the tab escape code (`\t`) within various strings has been used in order to align the positioning of the output produced. Also, the various monies required are generated using the random function `rand()`.

Activity 23.17

Modify *TestWorkers* to match the code given in FIG-23.8 (but remember to add the code for the four classes).

Check that the output produced is in alignment.

Add another 7 employees to the list using the following names and addresses:

Paul Dombey
Florence Dombey
Louisa Chick
Lucretia Tox
James Carker
Solomon Gills
Ned Cuttle

The Offices, St. Mary-le-Bow, London
13 Portland Place, London
The Docks, Wapping, London
157 Leadenhall, London
Long's Hotel, Bond Street, London
215 Brook Street, Grosvenor Square, London
3 Bryanston Place, London

Include any necessary tab characters in the address to ensure the output is aligned.

Assign each employee's worker type randomly and display that type on the same output line as their name.

Summary

- Using inheritance, a derived class takes on the features of the base class.
- When defining a derived class, start with the line

  `class` *derived_class_name* `: public` *base_class_name*
- Only new attributes and methods need be coded for the derived class.
- Constructors and destructors are not inherited.
- A derived class's constructor must call the base class's constructor. This is automatic if using the base class's zero-argument constructor.
- A derived class's destructor must call the base class's destructor, but this is automatic.

- When a derived class's constructor calls a base class constructor which requires one or more arguments, then this must be done in the first line of the derived class's constructor using the format

```
derived_class_name::derived_class_name(parameter list) base_class_name
(parameter_list)
{
    derived_class constructor code
}
```

- Inherited methods can be overridden by rewriting the method as part of the new class using the same return type, method name and parameter list as the original method in the bass class.

-

Aggregation and Composition

Introduction

At a design level **aggregation** can be thought of as an object which contains one or more other objects on what might be a temporary basis. For example, a taxi cab might temporarily contain passengers. Another property of an aggregate object is that when it is destroyed, the objects it contains can continue to exist. If the taxi is scrapped, the passengers it had held continue to exist.

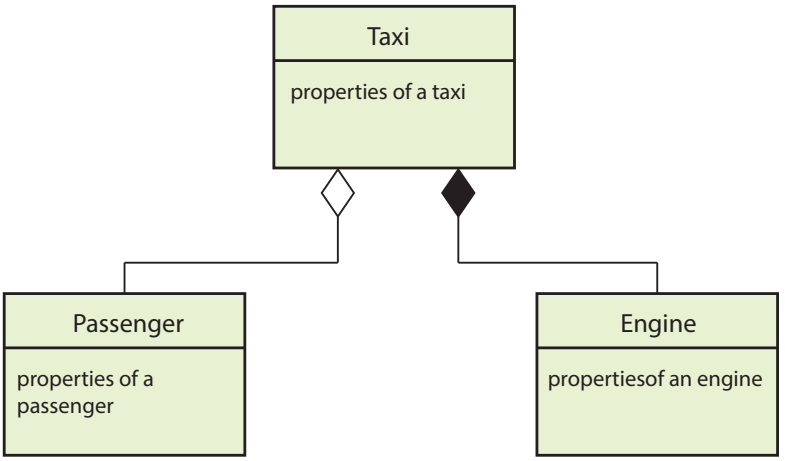
Composition, also reflects a situation where one object contains other objects, but this time on a more permanent basis. For example, the taxi contains an engine. When a composite object is destroyed, the objects it contains are also destroyed.

Graphical Representation

When drawing a class relationship diagram, aggregation is represented by an outline diamond while composition is shown using a filled diamond (see FIG-23.9).

FIG-23.9

Aggregation and Composition



Programming Composition and Aggregation

To see how we program these two relationships, lets start by creating a trivial *Engine* class:

```
//*************************************
//***          Engine Class           ***
//*************************************

class Engine
{
   private:
      int size;
   public:
      Engine(int = 1500);
      ~Engine();
      int GetSize() const;
};
```

```
//*** Create engine of size sz ***
Engine::Engine(int sz)
{
   cout << "Engine created\n";
   if (sz > 800 && sz < 6000)
      size = sz;
   else
      size = 1500;
}


//*** Destroy engine ***
Engine::~Engine()
{
   cout << "Engine destroyed\n";
}


//*** Get engine size ***
int Engine::GetSize() const
{
   return size;
}
```

Notice that messages have been included in the constructor and destructor to help us determine when an object is created and deleted.

Activity 23.18

Create a new project called *AggAndComp*. Create the file *Source.cpp* which should begin with the lines

```
#include<iostream>
using namespace std;
```

followed by the code for the *Engine* class.

Our *Passenger* class will contain only a name, a constructor and a destructor and a *GetName()* method:

```
//*************************************
//***        Passenger Class          ***
//*************************************

class Passenger
{
   private:
      string name;
   public:
      Passenger(string);
      ~Passenger();
      string GetName() const;
};


//*** Create passenger ***
Passenger::Passenger(string s)
{
   cout << "Passenger created\n";
   name = s;
}
```

```
//*** Delete passenger ***
Passenger::~Passenger()
{
   cout << "Passenger destroyed\n";
}


//*** Get name ***
string Passenger::GetName() const
{
   return name;
}
```

When coding a class, a composition attribute is most often declared as a value attribute while an aggregate attribute is usually a pointer or reference, so our minimal *Taxi* class would be coded as:

```
//************************************
//***       Taxi Class Methods       ***
//************************************
class Taxi
{
private:
   Engine eng;             //Details of engine
   int carrying;           //Passengers being carried
   Passenger* carried[4];  //Passengers' details
public:
   Taxi(int);
   ~Taxi();
   int GetEngineSize() const;
   void AddPassenger(Passenger*);
   Passenger* RemovePassenger();
   int GetPassengersInTaxi() const;
};


//*** Create taxi with engine size sz ***
Taxi::Taxi(int sz) :eng(sz)
{
   cout << "Taxi created\n";
   carrying = 0;
}

Taxi::~Taxi()
{
   cout << "Taxi destroyed\n";
}


int Taxi::GetEngineSize()const
{
   return eng.GetSize();
}


void Taxi::AddPassenger(Passenger* p)
{
   if (carrying < 4)
      carried[carrying++] = p;
}
```

```
Passenger* Taxi::RemovePassenger()
{
   if (carrying == 0)
      throw -1;
   else
      return carried[--carrying];
}

int Taxi::GetPassengersInTaxi() const
{
   return carrying;
}
```

Note that *RemovePassenger()* throws an exception if we attempt to remove a passenger when the taxi is empty. To keep things simple, we won't attempt to catch this exception.

The interesting point here is that the *Taxi* class's constructor will automatically call the constructor for the objects contained within the class without the need for any code. In this case, that means the *Engine* class constructor is called to create the *Engine* object within the *Taxi*.

This only works if there is a zero-argument constructor for the object to be created. Luckily, the *Engine* class has such a constructor. Where no zero argument constructor exists, or where we want to call a different constructor, then we must make an explicit call to the constructor within the composition class. We'll see how this is done by setting the taxi's engine size to the value supplied by the *Taxi* constructor:

```
Taxi::Taxi(int sz) :eng(sz)
{
   cout << "Taxi created\n";
   carrying = 0;
}
```

Note that the constructor is called by specifying the object's name (*eng*) within the composition class and supplying the parameters required by that class's constructor. It is also necessary to call the constructor before the opening braces of the composition class's constructor. If there are several objects within the composition class that require explicit constructor calls, then these should be comma separated. For example, if a class *XClass*, contains objects *attr1*, *attr2*, and *attr3* which need to have their constructors called explicitly we might start the *XClass* constructor with the code

```
XClass::XClass() : attr1(14), attr2("abc"), attr3(23,1,'a')
{
```

Activity 23.19

In project *AggAndComp*, add the code for the *Passenger* and *Taxi* class. Use an explicit constructor call to set the engine size to 1500.

All that is required now is a `main()` function to demonstrate the creation and destruction of the various elements used within a *Taxi* class object. The code is:

```
//*************************************
//***          Main program          ***
//*************************************

int main()
{
```

```
    //*** Create people ***
    Passenger people[] = {Passenger("Amy Dorrit"),
    ↳Passenger("Arthur Clennam"),Passenger("Pet Meagles") };
    {
       //*** Create taxi and add people ***
       Taxi tx(2000);
       cout <<"Taxi engine size is "<< tx.GetEngineSize()<<endl;
       for (int c = 0; c < 3; c++)
       {
          tx.AddPassenger(&people[c]);
          cout << "There are " << tx.GetPassengersInTaxi() <<
          ↳" passengers\n";
       }
       //*** Remove people from taxi ***
       int passengercount = tx.GetPassengersInTaxi();
       for (int c = 0; c < passengercount; c++)
       {
          cout << " * ****" << (tx.RemovePassenger())->GetName()
          ↳<< " leaves taxi * ****\n";
          cout << "There are " << tx.GetPassengersInTaxi() <<
          ↳" passengers\n";
       }

       cout << "End of taxi block\n";
    }
    cout << "End of program\n";
}
```

Activity 23.20

In project *AggAndComp*, add `main()` and test your program.

As you can see from the results of Activity 23.18 The composite element within the *Taxi* object (the *Engine* object) is automatically created and destroyed along with the *Taxi* object. The passengers on the other hand exist independently of the *Taxi* and linked to the taxi only temporarily, being created and destroyed quite separately from the *Taxi*.

Summary

- An aggregation refers to a main object which contains other object parts that may exist independently of the main object.
- A composition refers to a main object which contains other object attributes that only exist as part of the main object. That is, they come into existence when the main object is created and they are destroyed along with the main object.
- In C++, there need not be any difference in the coding of an aggregate component and a composite component although composite components must be referenced by pointers in the main object to ensure they continue to exist when the main object is destroyed.
- The constructor of a composite class must call the constructor of any objects it contains. This will be automatic if those objects have zero-argument constructors.

C++ Exception Classes

Introduction

Back in Chapter 18 we covered the basic idea behind exception handling and how we could throw a basic-type value or a class object when an unexpected error occurred.

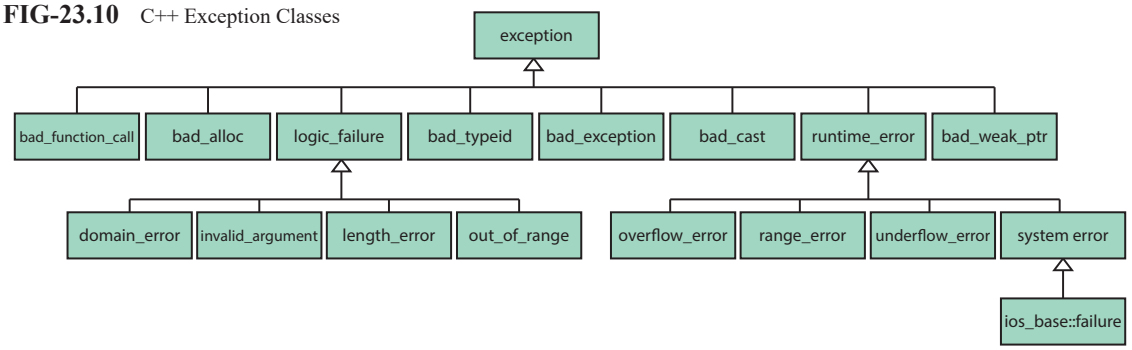
C++ defines its own family of error classes which are thrown by various methods in classes such as `array`, `vector` and `string`. For example, the `at()` method (defined in all three of these classes) throws an `out-of-range` object when an attempt is made to access a value at an invalid position in the object.

In this section we'll look at that family of C++-defined exception classes and create our own to handle *Time* and *Date* exceptions.

C++ Exceptions

C++ comes with a whole raft of classes defined within various libraries. Inevitably these classes have to handle a wide range of possible error conditions. To do this C++ defines a family of exception classes as shown in FIG-23.10.

FIG-23.10 C++ Exception Classes



The `exception` class, which as we can see from the diagram, is the base class for all the others, defines a method called `what()`. This method has the signature

```
virtual const char* what() const
```

and is designed to be overridden in derived classes to return a message giving a more detailed explanation of what has gone wrong.

Note that all of the above exception classes are defined within the `std` namespace.

User-Defined Exceptions

When we are creating exception classes of our own, a common approach is to create the new class by deriving it from the `exception` class and overriding the inherited `what()` method.

For example, in Chapter 19 we created a *Set* class which used a templated standard array to hold the values in a set. Since this array is of a fixed size, there is a possibility that the *AddToSet()* method will be unable to fulfill a request because the array is full. In this situation, we might throw a *Set_full* exception with this new class being derived from the `exception` class.

To code this we need to start by including the exception header file:

```
#include <exception>
```

Then we can define our new class, overriding the inherited version of what():

```
class Set_full : public exception
{
   public:
      const char* what() const
      {
         return "The set\'s array is full";
      }
};
```

Now we can have *AddToSet()* throw a *Set_full* exception if the array is full:

```
//*** Adds v to the set ***
template <class T>
void Set<T>::AddToSet(T v)
{
   if (!IsMember(v))
   {
      if (count == 100)
         throw Set_full();
      store[count] = v;
      count++;
   }
}
```

In `main()` we can place our calls to *AddToSet()* in a `try` block and add a `catch` block for the error:

```
int main()
{
   Set<dsk::Time> A;
   Set<dsk::Time> B;
   //*** Store dates in each set ***
   try
   {
      A.AddToSet(dsk::Time(1, 12, 20));
      A.AddToSet(dsk::Time(23, 11, 19));
      A.AddToSet(dsk::Time(8, 7, 16));
      B.AddToSet(dsk::Time(23, 11, 19));
      B.AddToSet(dsk::Time(8, 7, 16));
   }
   catch (Set_full er)
   {
      cout << er.what() << endl;
   }
      ...
```

Activity 23.21

Modify your *SetContainer* project (created in Chapter 19) so that the *Set_full* exception class is implemented as described above.

To test the exception in operation, change the condition used to throw the exception from `count == 100` to `count == 2` and run the program.

Catching Derived Classes

Earlier in this chapter we saw how an object pointer can be used to reference not only

objects of the class for which it was specified but also objects from classes derived from that class.

A similar rule is also true for exception handling: any try block designed to catch exceptions of a specific class will also automatically catch exceptions of a derived class. For example, if we write a catch block for the exception class as in the code

```
catch (exception er)
{
   cout << er.what() << end;
}
```

that block would also catch at *Set_full* exception since the *Set_full* class is derived from the `exception` class.

Activity 23.22

Modify the catch block in *SetContainer* to catch an `exception` class object rather than a *Set_full* object and check that the program runs exactly as before.

If we wished to have separate catch blocks for both `exception` and *Set_full*, then we must place the catch for the derived class first as in the code

```
catch (Set_full er)
{
   ...
}
catch (exception er)
{
   ...
}
```

If we were to place the catch for `exception` first it would also catch the *Set_full* exceptions making it impossible to execute the other `try` block (remember only one `try` block is executed when an exception is thrown).

Solutions

Activity 23.1

Modified code for `main()` in *TestClock*:

```
int main()
{
  dsk::Time *ptr1 = new dsk::Time(12, 23, 0);
  unique_ptr<dsk::Time> ptr2(make_unique
  ↳<dsk::Time>(12, 23, 0));
  (*ptr1)++;
  cout << *ptr1 << endl;
  cout << *ptr2 << endl;
}
```

The time displayed by the object referenced by *ptr1* is 12:23:01.

The time displayed by the object referenced by *ptr2* is 12:23:00.

Activity 23.2

Modified code for `main()` in *TestClock*:

```
int main()
{
  dsk::Time* ptr1 = new Clock(11, 2, 0);
  unique_ptr<dsk::Time> ptr2(make_unique
  ↳<CountdownClock>(11, 2, 0));
  ptr1->Tick();
  ptr2->Tick();
  cout << *ptr1 << endl;
  cout << *ptr2 << endl;
}
```

The program does not compile. The error displayed is

```
Error   C2039      'Tick': is not a member of 'dsk::Time'
```

Activity 23.3

Modified code for `main()` in *TestClock*:

```
int main()
{
  dsk::Time* ptr1 = new Clock(11, 2, 0);
  unique_ptr<dsk::Time> ptr2 = make_unique
  <CountdownClock>(11, 2, 0);
  static_cast<Clock*>(ptr1)->Tick();
  //ptr2->Tick();
  cout << *ptr1 << endl;
  cout << *ptr2 << endl;
}
```

By casting *ptr1* to type *Clock*, *Tick()* now executes as expected, with its time changing to 11:02:01.

Activity 23.4

Modified code for `main()` in *TestClock*:

```
int main()
{
  Clock* ptr1 = new Clock(11, 2, 0);
  unique_ptr<Clock>ptr2(make_unique
  ↳<CountdownClock>(11, 2, 0));
  ptr1->Tick();
  ptr2->Tick();
  cout << *ptr1 << endl;
  cout << *ptr2 << endl;
}
```

The program now runs but both objects show the same time:

```
11:02:01
11:02:01
```

Activity 23.5

Modified definition for *Clock*:

```
class Clock : public dsk::Time
{
  public:
    using Time::ToSeconds;
    using Time::FromSeconds;
    using Time::operator==;
    Clock(int=0, int=0, int=0);
    Clock(const Time&);
    Clock(const Clock&);
    virtual void Tick();
};
```

The program now runs but both objects show different times:

```
11:02:01
11:01:59
```

Activity 23.6

After adding `override` to the prototype of *Tick()* in the *CountdownClock* class, an error message is displayed when the term `virtual` is removed from the prototype of *Tick()* in *Clock*.

The program compiles and runs when the keyword `virtual` is restored.

Activity 23.7

Modified definition for *Clock*:

```
class Clock : public dsk::Time
{
  public:
    using Time::ToSeconds;
    using Time::FromSeconds;
    using Time::operator==;
    Clock(int=0, int=0, int=0);
    Clock(const Time&);
    Clock(const Clock&);
    virtual void Tick();
    void DisplayTicking(int);
};
```

Modified code for `main()` in *TestClock*:

```
int main()
{
  Clock* ptr1 = new Clock(11, 2, 0);
  unique_ptr<Clock>ptr2(make_unique
  ↳<CountdownClock>(11, 2, 0));
  ptr1->DisplayTicking(10);
  ptr2->DisplayTicking(10);
  cout << *ptr1 << endl;
  cout << *ptr2 << endl;
}
```

The version of *Tick()* called by *DisplayTicking()* is adjusted to match the type of object which is executing *DisplayTicking()*. Therefore we get the seconds being incremented in the execution of the line

`ptr1->DisplayTicking(10);`

while the seconds are decremented when executing

`ptr2->DisplayTicking(10);`

Activity 23.8

The program reports the object as a *Clock* or *CountdownClock* object depending on the response to the question.

Activity 23.9

Modified code for `main()` in *MultiInherit*:

```
int main()
{
  D var;
  var.f1();
}
```

Since there is now only one copy of class *A*'s features

embedded in class *D*, there is no ambiguity when calling method *f1()*.

Activity 23.10

Modified code for `main()` in *MultiInherit*:

```
int main()
{
  D var(8);
  var.f1();
}
```

This creates the following output:

```
A class constructor x set to 8
B class constructor called
C class constructor called
D class 2nd constructor called
Executed f1()
```

Activity 23.11

Code for *TestWorkers*:

```
#include <iostream>
using namespace std;

//*********************************************
//***             Worker Class              ***
//*********************************************
class Worker
{
  private:
    string name;
    string address;
  public:
    void SetName(const string&);
    void SetAddress(const string&);
    string GetName() const;
    string GetAddress() const;
    virtual double CalcPayment() const = 0;
};

//*** Set worker's name ***
void Worker::SetName(const string& n)
{
  name = n;
}
//*** Set worker's address ***
void Worker::SetAddress(const string& ad)
{
  address = ad;
}

//*** Get worker's name ***
string Worker::GetName() const
{
  return name;
}

//*** Get worker's address ***
string Worker::GetAddress() const
{
  return address;
}

//*************************************
//***         Main program          ***
//*************************************
int main()
{
  Worker w;
}
```

The compiler will object to an attempt to create a *Worker* class object since this is an abstract class. An error message such as

```
Error  C2259    'Worker': cannot instantiate abstract class
```

will be produced.

Removing the declaration will eliminate this error.

Activity 23.12

Modified code for *TestWorkers*:

```
#include <iostream>
using namespace std;

//*********************************************
//***             Worker Class              ***
//*********************************************
class Worker
{
  private:
    string name;
    string address;
  public:
    void SetName(const string&);
    void SetAddress(const string&);
    string GetName() const;
    string GetAddress() const;
    virtual double CalcPayment() const = 0;
};

//*** Set worker's name ***
void Worker::SetName(const string& n)
{
  name = n;
}
//*** Set worker's address ***
void Worker::SetAddress(const string& ad)
{
  address = ad;
}

//*** Get worker's name ***
string Worker::GetName() const
{
  return name;
}

//*** Get worker's address ***
string Worker::GetAddress() const
{
  return address;
}

//*************************************
//***     PartTimeWorker Class      ***
//*************************************
class PartTimeWorker : public Worker
{
  private:
    double hourlyrate;
    double hoursworked;
  public:
    void SetHourlyRate(double);
    void SetHoursWorked(double);
    double GetHourlyRate() const;
    double GetHoursWorked() const;
    double CalcPayment() const;
    double CalcDeductions() const;
};

//*** Set PTworker's hourly rate ***
void PartTimeWorker::SetHourlyRate(double r)
{
  if (r > 0 && r < 200)
    hourlyrate = r;
}

//*** Set PTworker's hours worked ***
void PartTimeWorker::SetHoursWorked(double hw)
{
  if (hw >= 0 && hw <= 200)
    hoursworked = hw;
}

//*** Get PTworker's hourly rate ***
double PartTimeWorker::GetHourlyRate() const
{
  return hourlyrate;
}

//*** Get PTworker's hours worked ***
double PartTimeWorker::GetHoursWorked() const
{
  return hoursworked;
}

//*** Calculate payment due for month ***
```

```
double PartTimeWorker::CalcPayment() const
{
  //*** calculate basic pay ***
  float moneydue = hoursworked * hourlyrate;
  //*** Add 50% for any hours over 160 ***
  if (hoursworked > 160)
    moneydue += (hoursworked-160)*hourlyrate*0.5;
  return moneydue;
}
```

A *PartTimeWorker* object can be created with code such as

```
int main()
{
  PartTimeWorker w;
}
```

Activity 23.13

Modified code for *TestWorkers*:

```
#include <iostream>
using namespace std;

//*********************************************
//***              Worker Class             ***
//*********************************************
class Worker
{
  private:
    string name;
    string address;
  public:
    void SetName(const string&);
    void SetAddress(const string&);
    string GetName() const;
    string GetAddress() const;
    virtual double CalcPayment() const = 0;
};

//*** Set worker's name ***
void Worker::SetName(const string& n)
{
  name = n;
}
//*** Set worker's address ***
void Worker::SetAddress(const string& ad)
{
  address = ad;
}

//*** Get worker's name ***
string Worker::GetName() const
{
  return name;
}

//*** Get worker's address ***
string Worker::GetAddress() const
{
  return address;
}

//*************************************
//***     PartTimeWorker Class      ***
//*************************************
class PartTimeWorker : public Worker
{
  private:
    double hourlyrate;
    double hoursworked;
  public:
    void SetHourlyRate(double);
    void SetHoursWorked(double);
    double GetHourlyRate() const;
    double GetHoursWorked() const;
    double CalcPayment() const;
    double CalcDeductions() const;
};

//*** Set PTworker's hourly rate ***
void PartTimeWorker::SetHourlyRate(double r)
{
  if (r > 0 && r < 200)
    hourlyrate = r;
}

//*** Set PTworker's hours worked ***
void PartTimeWorker::SetHoursWorked(double hw)
{
  if (hw >= 0 && hw <= 200)
    hoursworked = hw;
}

//*** Get PTworker's hourly rate ***
double PartTimeWorker::GetHourlyRate() const
{
  return hourlyrate;
}

//*** Get PTworker's hours worked ***
double PartTimeWorker::GetHoursWorked() const
{
  return hoursworked;
}

//*** Calculate payment due for month ***
double PartTimeWorker::CalcPayment() const
{
  //*** calculate basic pay ***
  float moneydue = hoursworked * hourlyrate;
  //*** Add 50% for any hours over 160 ***
  if (hoursworked > 160)
    moneydue += (hoursworked-160)*hourlyrate*0.5;
  return moneydue;
}

//*************************************
//***     FullTimeWorker Class      ***
//*************************************

class FullTimeWorker : public Worker
{
  private:
    double salary;
  public:
    void SetSalary(double);
    double GetSalary() const;
    double CalcPayment() const;
};

//*** Set FTWorker's salary ***
void FullTimeWorker::SetSalary(double s)
{
  if (s > 0)
    salary = s;
}

//*** Get FTworker's salary ***
double FullTimeWorker::GetSalary() const
{
  return salary;
}

//*** Calculate payment due for month ***
double FullTimeWorker::CalcPayment() const
{
  return salary / 12;
}

//*************************************
//***     ContractWorker Class      ***
//*************************************
class ContractWorker : public Worker
{
  private:
    double hourlyrate;
    int contractperiod;
  public:
    void SetHourlyRate(double);
    void SetContractPeriod(int);
    double GetHourlyRate() const;
  int GetContractPeriod() const;
  double CalcPayment() const;
};

//*** Set Contractworker's hourly rate ***
void ContractWorker::SetHourlyRate(double r)
{
  if (r > 0 && r < 500)
    hourlyrate = r;
}

//*** Set Contract worker's contract period ***
void ContractWorker::SetContractPeriod(int p)
{
  if (p > 0)
    contractperiod = p;
}
```

```
//*** Get Contract worker's hourly rate ***
double ContractWorker::GetHourlyRate() const
{
  return hourlyrate;
}


//*** Get Contract worker's contract period ***
int ContractWorker::GetContractPeriod() const
{
  return contractperiod;
}

//*** Calculate payment due for month ***
double ContractWorker::CalcPayment() const
{
  return 160 * hourlyrate;
}


//**************************************
//***         Main program           ***
//**************************************
int main()
{

}
```

The program compiles.

Activity 23.14

Modified code for `main()` in *TestWorker (version 1)*:

```
int main()
{
  Worker* wptr;
  wptr = new PartTimeWorker;
  wptr->SetName("Estella Havisham");
  wptr->SetAddress("Satis House\nRochester\nKent");
  static_cast<PartTimeWorker*>(wptr)->
  SetHourlyRate(20.80);
  static_cast<PartTimeWorker*>(wptr)->
  SetHoursWorked(200);
  cout << "Wage due : " << wptr->CalcPayment();
}
```

The output is

```
Wage due : 4576
```

Modified code for `main()` in *TestWorker (version 2)*:

```
int main()
{
  Worker* wptr;
  wptr = new FullTimeWorker;
  wptr->SetName("Estella Havisham");
  wptr->SetAddress("Satis House\nRochester\nKent");
  static_cast<FullTimeWorker*>(wptr)->
  SetSalary(72000.0);
  cout << "Wage due : " << wptr->CalcPayment();
}
```

The output is

```
Wage due : 6000
```

Activity 23.15

Modified code for *TestWorker*:

```
#include <iostream>
using namespace std;

//*****************************************
//***             Worker Class          ***
//*****************************************
class Worker
{
private:
  string name;
  string address;
public:
  void SetName(const string&);
  void SetAddress(const string&);
  string GetName() const;
  string GetAddress() const;
  virtual double CalcPayment() const = 0;
  virtual double CalcDeductions() const = 0;
};

//*** Set worker's name ***
void Worker::SetName(const string& n)
{
  name = n;
}
//*** Set worker's address ***
void Worker::SetAddress(const string& ad)
{
  address = ad;
}

//*** Get worker's name ***
string Worker::GetName() const
{
  return name;
}

//*** Get worker's address ***
string Worker::GetAddress() const
{
  return address;
}

//**************************************
//***     PartTimeWorker Class       ***
//**************************************
class PartTimeWorker : public Worker
{
private:
  double hourlyrate;
  double hoursworked;
public:
  void SetHourlyRate(double);
  void SetHoursWorked(double);
  double GetHourlyRate() const;
  double GetHoursWorked() const;
  double CalcPayment() const;
  double CalcDeductions() const;
};

//*** Set PTworker's hourly rate ***
void PartTimeWorker::SetHourlyRate(double r)
{
  if (r > 0 && r < 200)
    hourlyrate = r;
}

//*** Set PTworker's hours worked ***
void PartTimeWorker::SetHoursWorked(double hw)
{
  if (hw >= 0 && hw <= 200)
    hoursworked = hw;
}

//*** Get PTworker's hourly rate ***
double PartTimeWorker::GetHourlyRate() const
{
  return hourlyrate;
}

//*** Get PTworker's hours worked ***
double PartTimeWorker::GetHoursWorked() const
{
  return hoursworked;
}

//*** Calculate payment due for month ***
double PartTimeWorker::CalcPayment() const
{
  //*** calculate basic pay ***
  float moneydue = hoursworked * hourlyrate;
  //*** Add 50% for any hours over 160 ***
  if (hoursworked > 160)
    moneydue += (hoursworked - 160) * hourlyrate *
0.5;
  return moneydue;
}

//*** Calculate deductions due ***
double PartTimeWorker::CalcDeductions() const
{
  return CalcPayment() * 0.2;
}
```

```
//************************************
//***     FullTimeWorker Class       ***
//************************************

class FullTimeWorker : public Worker
{
  private:
     double salary;
  public:
     void SetSalary(double);
     double GetSalary() const;
     double CalcPayment() const;
     double CalcDeductions() const;
};

//*** Set FTWorker's salary ***
void FullTimeWorker::SetSalary(double s)
{
  if (s > 0)
     salary = s;
}

//*** Get FTworker's salary ***
double FullTimeWorker::GetSalary() const
{
  return salary;
}

//*** Calculate payment due for month ***
double FullTimeWorker::CalcPayment() const
{
  return salary / 12;
}

//*** Calculate deductions due ***
double FullTimeWorker::CalcDeductions() const
{
  return CalcPayment() * 0.31;
}

//************************************
//***     ContractWorker Class       ***
//************************************
class ContractWorker : public Worker
{
private:
  double hourlyrate;
  int contractperiod;
public:
  void SetHourlyRate(double);
  void SetContractPeriod(int);
  double GetHourlyRate() const;
  int GetContractPeriod() const;
  double CalcPayment() const;
  double CalcDeductions() const;
};

//*** Set Contractworker's hourly rate ***
void ContractWorker::SetHourlyRate(double r)
{
  if (r > 0 && r < 500)
     hourlyrate = r;
}

//*** Set Contract worker's contract period ***
void ContractWorker::SetContractPeriod(int p)
{
  if (p > 0)
     contractperiod = p;
}


//*** Get Contract worker's hourly rate ***
double ContractWorker::GetHourlyRate() const
{
  return hourlyrate;
}


//*** Get Contract worker's contract period ***
int ContractWorker::GetContractPeriod() const
{
  return contractperiod;
}

//*** Calculate payment due for month ***
double ContractWorker::CalcPayment() const
{
  return 160 * hourlyrate;
}

//*** Calculate deductions due ***
double ContractWorker::CalcDeductions() const
{
  return 0.0;
}


//************************************
//***         Main program           ***
//************************************
int main()
{
  Worker* wptr, *wptr2, *wptr3;
  wptr = new FullTimeWorker;
  wptr->SetName("Estella Havisham");
  wptr->SetAddress("Satis House\nRochester\nKent");
  static_cast<FullTimeWorker*>(wptr)->
  SetSalary(72000.0);
  cout << "Employee's name:\n" << wptr->GetName() <<
  endl;
  cout << "Address:\n" << wptr->GetAddress() <<
  endl;
  cout << "Wage due :\n" << wptr->CalcPayment() <<
  endl;
  cout << "Deductions :\n" << wptr->CalcDeductions()
  << endl;
  wptr2 = new PartTimeWorker;
  wptr2->SetName("Philip Pirrip");
  wptr2->SetAddress("The Forge\nThe Marshes\nKent");
  static_cast<PartTimeWorker*>(wptr2)->
  SetHourlyRate(27.35);
  static_cast<PartTimeWorker*>(wptr2)->
  SetHoursWorked(172);
  cout << "\n\nEmployee's name:\n" << wptr2->
  GetName() << endl;
  cout << "Address:\n" << wptr2->GetAddress() <<
  endl;
  cout << "Wage due :\n" << wptr2->CalcPayment() <<
  endl;
  cout << "Deductions :\n" << wptr2->
  CalcDeductions() << endl;
  wptr3 = new ContractWorker;
  wptr3->SetName("Herbert Pocket");
  wptr3->SetAddress("Barnard's Inn\nLondon");
  static_cast<ContractWorker*>(wptr3)->
  SetHourlyRate(65.85);
  cout << "\n\nEmployee's name:\n" << wptr3->
  GetName() << endl;
  cout << "Address:\n" << wptr3->GetAddress() <<
  endl;
  cout << "Wage due :\n" << wptr3->CalcPayment() <<
  endl;
  cout << "Deductions :\n" << wptr3->
  CalcDeductions() << endl;
}
```

Output produced:

```
Employee's name:
Estella Havisham
Address:
Satis House
Rochester
Kent
Wage due :
6000
Deductions :
1860


Employee's name:
Philip Pirrip
Address:
The Forge
The Marshes
Kent
Wage due :
4576
Deductions :
915.2


Employee's name:
Herbert Pocket
Address:
Barnard's Inn
London
Wage due :
10536
```

```
Deductions :
0
```

Activity 23.16

Modified code for the abstract *Worker* class:

```
class Worker
{
  private:
    string name;
    string address;
  public:
    void SetName(const string&);
    void SetAddress(const string&);
    string GetName() const;
    string GetAddress() const;
    virtual double CalcPayment() const = 0;
    virtual double CalcDeductions() const = 0;
    double CalcNet() const;
};

double Worker::CalcNet() const
{
  return (CalcPayment() - CalcDeductions());
}
```

Modified code for `main()`:

```
int main()
{
  Worker* wptr, *wptr2, *wptr3;
  wptr = new FullTimeWorker;
  wptr->SetName("Estella Havisham");
  wptr->SetAddress("Satis House\nRochester\nKent");
  static_cast<FullTimeWorker*>(wptr)->
  SetSalary(72000.0);
  cout << "Employee's name:\n" << wptr->GetName() <<
  endl;
  cout << "Address:\n" << wptr->GetAddress() <<
  endl;
  cout << "Wage due :\n" << wptr->CalcPayment() <<
  endl;
  cout << "Deductions :\n" << wptr->CalcDeductions()
  << endl;
  cout << "NET wage:\n" << wptr->CalcNet() << endl;
  wptr2 = new PartTimeWorker;
  wptr2->SetName("Philip Pirrip");
  wptr2->SetAddress("The Forge\nThe Marshes\nKent");
  static_cast<PartTimeWorker*>(wptr2)->
  SetHourlyRate(20.80);
  static_cast<PartTimeWorker*>(wptr2)->
  SetHoursWorked(200);
  cout << "\n\nEmployee's name:\n" << wptr2->
  GetName() << endl;
  cout << "Address:\n" << wptr2->GetAddress() <<
  endl;
  cout << "Wage due :\n" << wptr2->CalcPayment() <<
  endl;
  cout << "Deductions :\n" << wptr2->
  CalcDeductions() << endl;
  cout << "NET wage:\n" << wptr2->CalcNet() << endl;
  wptr3 = new ContractWorker;
  wptr3->SetName("Herbert Pocket");
  wptr3->SetAddress("Barnard's Inn\nLondon");
  static_cast<ContractWorker*>(wptr3)->
  SetHourlyRate(65.85);
  cout << "\n\nEmployee's name:\n" << wptr3->
  GetName() << endl;
  cout << "Address:\n" << wptr3->GetAddress() <<
  endl;
  cout << "Wage due :\n" << wptr3->CalcPayment() <<
  endl;
  cout << "Deductions :\n" << wptr3->
  CalcDeductions() << endl;
  cout << "NET wage:\n" << wptr3->CalcNet() << endl;
}
```

Activity 23.17

Modified code for `main()`:

```
//*************************************
//***         Main program            ***
//*************************************
int main()
{
  //*** Names and addresses of employees ***
  string names[] = { "Philip Pirrip",
  "Estella Havisham","Herbert Pocket",
  "Paul Dombey","Florence Dombey",
  "Louisa Chick","Lucretia Tox",
  "James Carker","Solomon Gills","Ned Cuttle" };
  string addresses[] = { "The Forge\n\t\t\tThe
  Marshes\n\t\t\tKent","Satis House\n\t\t\t
  Rochester\n\t\t\tKent", "Barnard's Inn\n\t\t\t
  London","The Offices\n\t\t\tSt. Mary-le-Bow\n\t\t
  \tLondon",
  "13 Portland Place\n\t\t\tLondon","The Docks\n\t
  \t\tWapping\n\t\t\tLondon","157 Leadenhall\n\t\t
  \tLondon","Long's Hotel\n\t\t\tBond Street\n\t\t
  \tLondon","215 Brook Street\n\t\t\tGrosvenor
  Square\n\t\t\tLondon","3 Bryanston Place\n\t\t\t
  London" };
  //*** Seed random numbers ***
  srand(time(nullptr));
  //*** Workers vector ***
  vector<Worker*> employees(10);
  //*** Assign worker type randomly ***
  for (int type,c = 0; c < 10; c++)
  {
    type = rand() % 3;
    switch (type)
    {
      case 0:
        employees[c] = new PartTimeWorker;
        break;
      case 1:
        employees[c] = new FullTimeWorker;
        break;
      case 2:
        employees[c] = new ContractWorker;
        break;
    }
  }
  //*** Add details for all employees ***
  int index = 0;
  for (auto wptr : employees)
  {
    wptr->SetName(names[index]);
    wptr->SetAddress(addresses[index]);
    if (typeid(*wptr) == typeid(PartTimeWorker))
    {
      static_cast<PartTimeWorker*>(wptr)->
      SetHourlyRate(rand() % 50);
      static_cast<PartTimeWorker*>(wptr)->
      SetHoursWorked(rand() % 100 + 100);
    }
    else if (typeid(*wptr) == typeid(
    FullTimeWorker))
      static_cast<FullTimeWorker*>(wptr)->
      SetSalary(rand() % 30000 + 50000);
    else
      static_cast<ContractWorker*>(wptr)->
      SetHourlyRate(rand() % 100 + 30);
    index++;
  }
  //*** Display details of all employees ***
  for (auto wptr : employees)
  {
    cout << "\n\nEmployee's name:\t" << wptr->
    GetName() << "   " << string(typeid(
    *wptr).name()).substr(6) << endl;
    cout <<"Address:\t\t"<<wptr->GetAddress()<<endl;
    cout <<"Wage due :\t\t"<< wptr->CalcPayment()
    <<endl;
    cout << "Deductions :\t\t" << wptr->
    CalcDeductions() << endl;
    cout << "NET wage:\t\t" <<wptr->CalcNet()<<endl;
  }
}
```

Activity 23.18

Code for *AggAndComp*:

```
#include <iostream>
using namespace std;

//*************************************
//***          Engine Class           ***
//*************************************

class Engine
{
private:
  int size;
public:
```

```
  Engine(int = 1500);
  ~Engine();
  int GetSize() const;

};


//*** Create engine of size sz ***
Engine::Engine(int sz)
{
  cout << "Engine created\n";
  if (sz > 800 && sz < 6000)
    size = sz;
  else
    size = 1500;
}

//*** Destroy engine ***
Engine::~Engine()
{
  cout << "Engine destroyed\n";
}


//*** Get engine size ***
int Engine::GetSize() const
{
  return size;
}
```

Activity 23.19

Modified code for *AggAndComp*:

```
#include <iostream>
using namespace std;

//*************************************
//***          Engine Class         ***
//*************************************

class Engine
{
private:
  int size;
public:
  Engine(int = 1500);
  ~Engine();
  int GetSize() const;

};


//*** Create engine of size sz ***
Engine::Engine(int sz)
{
  cout << "Engine created\n";
  if (sz > 800 && sz < 6000)
    size = sz;
  else
    size = 1500;
}

//*** Destroy engine ***
Engine::~Engine()
{
  cout << "Engine destroyed\n";
}


//*** Get engine size ***
int Engine::GetSize() const
{
  return size;
}

//*************************************
//***        Passenger Class        ***
//*************************************

class Passenger
{
  private:
    string name;
  public:
    Passenger(string);
    ~Passenger();
    string GetName() const;
};


//*** Create passenger ***
Passenger::Passenger(string s)
{
  cout << "Passenger created\n";
  name = s;
}

//*** Delete passenger ***
Passenger::~Passenger()
{
  cout << "Passenger destroyed\n";
}

//*** Get name ***
string Passenger::GetName() const
{
  return name;
}


//*************************************
//***       Taxi Class Methods       ***
//*************************************
class Taxi
{
private:
  Engine eng;          //Details of engine
  int carrying;        //Passengers being carried
  Passenger* carried[4];//Passengers' details
public:
  Taxi(int);
  ~Taxi();
  int GetEngineSize() const;
  void AddPassenger(Passenger*);
  Passenger* RemovePassenger();
  int GetPassengersInTaxi() const;
};


//*** Create taxi with engine size sz ***
Taxi::Taxi(int sz): eng(sz)
{
  cout << "Taxi created\n";
  carrying = 0;
}

Taxi::~Taxi()
{
  cout << "Taxi destroyed\n";
}

int Taxi::GetEngineSize()const
{
  return eng.GetSize();
}

void Taxi::AddPassenger(Passenger* p)
{
  if (carrying < 4)
    carried[carrying++] = p;
}

Passenger* Taxi::RemovePassenger()
{
  if (carrying == 0)
    throw -1;
  else
    return carried[--carrying];
}

int Taxi::GetPassengersInTaxi() const
{
  return carrying;
}
```

Activity 23.20

Modified code for *AggAndComp*:

```
#include <iostream>
using namespace std;

//*************************************
//***          Engine Class          ***
//*************************************

class Engine
{
private:
```

```
  int size;
public:
  Engine(int = 1500);
  ~Engine();
  int GetSize() const;

};


//*** Create engine of size sz ***
Engine::Engine(int sz)
{
  cout << "Engine created\n";
  if (sz > 800 && sz < 6000)
    size = sz;
  else
    size = 1500;
}

//*** Destroy engine ***
Engine::~Engine()
{
  cout << "Engine destroyed\n";
}


//*** Get engine size ***
int Engine::GetSize() const
{
  return size;
}


//**************************************
//***        Passenger Class         ***
//**************************************

class Passenger
{
  private:
    string name;
  public:
    Passenger(string);
    ~Passenger();
    string GetName() const;
};

//*** Create passenger ***
Passenger::Passenger(string s)
{
  cout << "Passenger created\n";
  name = s;
}

//*** Delete passenger ***
Passenger::~Passenger()
{
  cout << "Passenger destroyed\n";
}

//*** Get name ***
string Passenger::GetName() const
{
  return name;
}


//**************************************
//***       Taxi Class Methods       ***
//**************************************
class Taxi
{
private:
  Engine eng;           //Details of engine
  int carrying;         //Passengers being carried
  Passenger* carried[4];    //Passengers' details
public:
  Taxi(int);
  ~Taxi();
  int GetEngineSize() const;
  void AddPassenger(Passenger*);
  Passenger* RemovePassenger();
  int GetPassengersInTaxi() const;
};


//*** Create taxi with engine size sz ***
Taxi::Taxi(int sz): eng(sz)
{
  cout << "Taxi created\n";
  carrying = 0;
}

Taxi::~Taxi()
{
  cout << "Taxi destroyed\n";
}

int Taxi::GetEngineSize()const
{
  return eng.GetSize();
}

void Taxi::AddPassenger(Passenger* p)
{
  if (carrying < 4)
    carried[carrying++] = p;
}

Passenger* Taxi::RemovePassenger()
{
  if (carrying == 0)
    throw -1;
  else
    return carried[--carrying];
}

int Taxi::GetPassengersInTaxi() const
{
  return carrying;
}


//**************************************
//***          Main program          ***
//**************************************

int main()
{
  //*** Create people ***
  Passenger people[] = {Passenger("Amy Dorrit"),
  ↳Passenger("Arthur Clennam"),Passenger(
  ↳"Pet Meagles") };
  {
    //*** Create taxi and add people ***
    Taxi tx(2000);
    cout << "Taxi engine size is " <<
    ↳tx.GetEngineSize() << endl;
    for (int c = 0; c < 3; c++)
    {
      tx.AddPassenger(&people[c]);
      cout << "There are " <<
      ↳tx.GetPassengersInTaxi() << " passengers\n";
    }
    //*** Remove people from taxi ***
    int passengercount = tx.GetPassengersInTaxi();
    for (int c = 0; c < passengercount; c++)
    {
      cout << " * ****" << (tx.RemovePassenger())
      ↳->GetName() << " leaves taxi * ****\n";
      cout << "There are " <<
      ↳tx.GetPassengersInTaxi() << " passengers\n";
    }
    cout << "End of taxi block\n";
  }
  cout << "End of program\n";
}
```

Activity 23.21

Modified code for *SetContainer*:

```
#include <iostream>
#include <ChronoLib.h>
#include <exception>
using namespace std;


class Set_full : public exception
{
  public:
    const char* what() const
    {
      return "The set\'s array is full";
    }
};
```

```
template <class T>
class Set
{
private:
  T store[2];
  int count;
public:
  Set();
  bool IsEmptySet() const;
  bool IsMember(T) const;
  bool IsSubSet(const Set<T>&) const;
  int Card() const;
  void AddToSet(T);
  void RemoveFromSet(T);
  Set<T> Union(const Set<T>&) const;
  Set<T> Intersection(const Set<T>&) const;
  Set<T> Difference(const Set<T>&) const;
  template <class T>
  friend ostream& operator<<(ostream&, const
Set<T>&);
};

//*** Creates an empty set ***
template <class T>
Set<T>::Set()
{
  count = 0;
}

//*** Return true if set is empty ***
template <class T>
bool Set<T>::IsEmptySet() const
{
  bool result = false;
  if (count == 0)
    result = true;
  return result;
}

//*** Return true if v is a member ***
template <class T>
bool Set<T>::IsMember(T v) const
{
  bool result = false;
  for (int c = 0; c < count; c++)
    if (store[c] == v)
      result = true;
  return result;
}

//*** Returns true if s is a subset ***
template <class T>
bool Set<T>::IsSubSet(const Set<T>& s) const
{
  bool result = true;
  for (int c = 0; c < s.count; c++)
    if (!(IsMember(s.store[c])))
    {
      result = false;
      break;
    }
  return result;
}

//*** Returns the number of members ***
template <class T>
int Set<T>::Card() const
{
  return count;
}

//*** Adds v to the set ***
template <class T>
void Set<T>::AddToSet(T v)
{
  if (!IsMember(v))
  {
    if (count == 2)
      throw Set_full();
    store[count] = v;
    count++;
  }
}

//*** Removes v from the set ***
template <class T>
void Set<T>::RemoveFromSet(T v)
{
  if (IsEmptySet())
    return;
  int c = 0;
  while (store[c] != v && c < count-1)
    c++;
  if (store[c] == v)
  {
    for (int k = c; k < count - 1; k++)
      store[k] = store[k + 1];
    count--;
  }
}

template <class T>
Set<T> Set<T>::Union(const Set<T>& s) const
{
  Set result = *this;
  for (int c = 0; c < s.count; c++)
    if (!result.IsMember(s.store[c]))
      result.AddToSet(s.store[c]);
  return result;
}

//*** Form a set from the members common to two sets
***
template <class T>
Set<T> Set<T>::Intersection(const Set<T>& s) const
{
  Set result;
  for (int c = 0; c < s.count; c++)
    if (IsMember(s.store[c]))
      result.AddToSet(s.store[c]);
  return result;
}

//*** Form a set from members in the current set not
in s ***
template <class T>
Set<T> Set<T>::Difference(const Set<T>& s) const
{
  Set result;
  for (int c = 0; c < count; c++)
    if (!s.IsMember(store[c]))
      result.AddToSet(store[c]);
  return result;
}

//*** Display the contents of a set ***
template <class T>
ostream& operator<<(ostream& os, const Set<T>& s)
{
  for (int c = 0; c < s.count; c++)
    os << s.store[c] << "  ";
  return os;
}


int main()
{
  Set<dsk::Time> A;
  Set<dsk::Time> B;
  //*** Store dates in each set ***
  try
  {
    A.AddToSet(dsk::Time(1, 12, 20));
    A.AddToSet(dsk::Time(23, 11, 19));
    A.AddToSet(dsk::Time(8, 7, 16));
    B.AddToSet(dsk::Time(23, 11, 19));
    B.AddToSet(dsk::Time(8, 7, 16));
  }
  catch (Set_full er)
  {
    cout << er.what() << endl;
  }
  cout << A.Card() << endl;
  cout << A << endl;
  cout << B << endl;
  cout << A.Union(B) << endl;
  cout << A.Intersection(B) << endl;
  cout << A.Difference(B) << endl;
  //*** Additional tests ***
  B.RemoveFromSet(dsk::Time(8, 7, 16));
  cout << "\n\nContents of B : \n";
  cout << B << endl;
  cout << "\nB subset of A ? " << A.IsSubSet(B) <<
  endl;
  cout << "\nA is empty ? " << A.IsEmptySet() <<
  endl;
}
```

Activity 23.22

Modified code for `main()` in *SetContainer*:

```
int main()
{
  Set<dsk::Time> A;
  Set<dsk::Time> B;
  //*** Store dates in each set ***
  try
  {
    A.AddToSet(dsk::Time(1, 12, 20));
    A.AddToSet(dsk::Time(23, 11, 19));
    A.AddToSet(dsk::Time(8, 7, 16));
    B.AddToSet(dsk::Time(23, 11, 19));
    B.AddToSet(dsk::Time(8, 7, 16));
  }
  catch (exception er)
  {
    cout << er.what() << endl;
  }
  cout << A.Card() << endl;
  cout << A << endl;
  cout << B << endl;
  cout << A.Union(B) << endl;
  cout << A.Intersection(B) << endl;
  cout << A.Difference(B) << endl;
  //*** Additional tests ***
  B.RemoveFromSet(dsk::Time(8, 7, 16));
  cout << "\n\nContents of B : \n";
  cout << B << endl;
  cout << "\nB subset of A ? " << A.IsSubSet(B) <<
  ↳endl;
  cout << "\nA is empty ? " << A.IsEmptySet() <<
  ↳endl;
}
```

The code should execute exactly as before.

24

I/O Classes

In this Chapter:

- ❑ Stream Handling Using C++ Classes
- ❑ The Abstract *ios_base* Class
- ❑ The Abstract *ios* Class
- ❑ The *istream* Class
- ❑ The *ostream* Class
- ❑ The *ifstream* Class
- ❑ The *ofstream* Class
- ❑ The *iostream* Class
- ❑ The *fstream* Class

I/O Classes

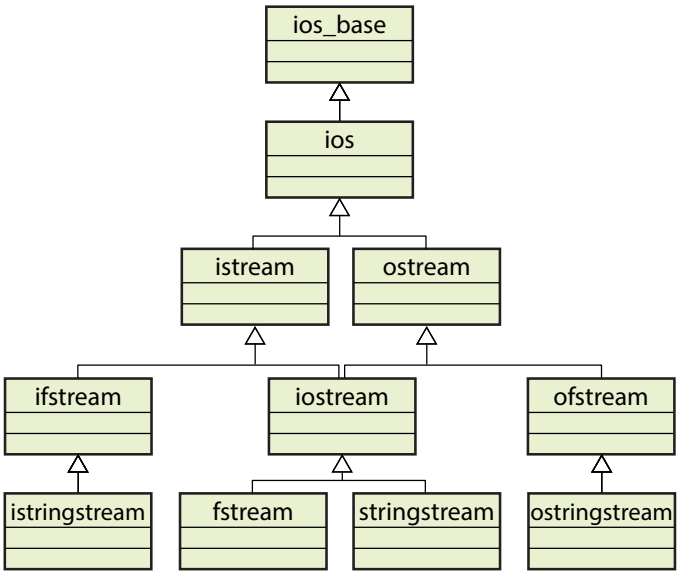
Introduction

We've already seen in Chapter 15 that C++ treats input and output devices and files as a stream of bytes to which data can be sent (output) or from which data can be extracted (input). The file handling techniques shown in that chapter were inherited from C and are retained for backward compatibility with that earlier language. However, C++ also offers a class-based approach to handling both input and output streams.

The core classes involved in I/O are shown in FIG-24.1.

FIG-24.1

C++'s IO Main Classes



The class `ios_base` is an abstract base class for all streams. The class `ios` is another abstract class specifically for streams that use standard, narrow (byte-sized) characters. The classes `istream` (input) and `ostream` (output) are used for handling basic IO normally from the keyboard and screen respectively. The derived classes `ifstream` and `ofstream` are used for reading from and writing to files while `fstream` is used for files which require to be opened in read/write mode (usually for record updating).

Objects from `istream` and `ostream` are used in almost every program we have written so far. This is because, rather than being keywords of the language, `cin` is actually an `istream` object and `cout` is an `ostream` object and are declared in the *iostream* header file.

The classes `istringstream`, `ostringstream` and `stringstream` connect a stream to a standard string.

The *ios_base* Class

This class defines the core properties of all stream classes. These features, inherited by the later classes, provide formatting, error checking and status information. Some of the features are appropriate only to outputting classes (such as `ostream`) while others deal exclusively with input (and used in `istream`).

The `ios_base` class defines a set of flags which affect the layout produced on the screen and how data from an input stream is treated. These formatting flags are stored as individual bits in an attribute called `flags` (see FIG-24.2).

FIG-24.2

IO Flags

Flag	Purpose
skipws	Skips white space charcters in the input stream
unitbuf	Flushes the output buffer after an output operation
uppercase	Uses uppercase X for hexadecimal values and uppercase E for real values
showbase	Displays the numeric base following a displayed number
showpoint	Displays numeric values with a decimal point and trailing zeros
showpos	Displays a + symbol in front of positive numeric values
left	Left justifies output
right	Right justifies output
internal	Right justifies a numeric sign symbol and left justifies the value
dec	Following input values should be treated as decimal
oct	Following input values should be treated as octal
hex	Following input values should be treated as hexadecimal
scientific	Displays real values in scientific notation
fixed	Displays real values in fixed point notation
boolalpha	Input and output bool values are in text form (*true*/*false*)

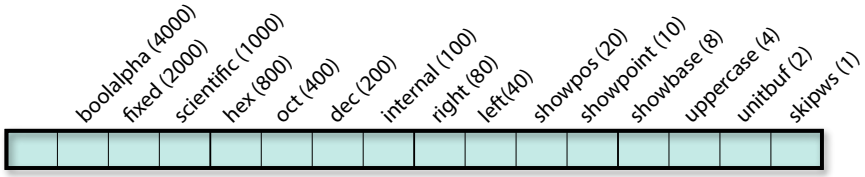


Other compilers may use different numeric values for these flags.

By setting the required flag(s) to 1 the corresponding formatting feature is activated, while resetting the flag to zero, deactivates that option. The flag bit positions and hexadecimal values (as defined in Microsoft's C++) are shown in FIG-24.3.

FIG-24.3

`flags` Mapping in Visual C++



The individual flags are defined as named static constants and can be referenced using the terms given above.

To reference the flags in our code we need to precede the name by the class or object name. Since they are inherited through several classes, we could use terms such as

```
base_ios::left
ios::left
istream::left
```
etc.

or we could make use of the default `istream` and `ostream` objects to write

```
cin.skipws
cout.left
```

Most frequently we are likely to see the `ios::` term used since it requires minimal typing!

Note that some flags are mutually exclusive. Hence, we can only set one flag from the group *left*, *right* and *internal*; one from *dec*, *oct* and *hex*; and one from *scientific* and *fixed*.

long flags()
long flags(long)

This method comes in two versions. The first, without parameters, returns the current state of the flags as a single integer value. For example, if we wanted to display the flag settings for `cout` (which inherits this method) we could use the line

```
cout << cout.flags() << endl;
```

This will display an integer value such as 513. A slightly better option would be to output the value in hexadecimal:

```
cout << hex << cout.flags() << endl;
```

which will display the value 201. From the flag mappings in FIG-24.3, we can see that this value means that the `dec` and `skipws` flags are set.

> **Activity 24.1**
>
> Start a new project called *TestIOS* which displays the numeric value of the default flag settings in decimal and hexadecimal.

Notice that no alignment flag (`left`, `right` and `internal`) nor real formatting flag (`fixed` and `scientific`) are set. These have default values even without a flag being set.

The second version of the `flags()` method returns the current flags setting and accepts a `long int` parameter specifying the new flag settings. For example, if we wanted to set the `hex` and `left` flags to modify how numeric data was displayed we could use the line

```
cout.flags(0x840);
```

Of course, that example depends on us remembering the flag values. An easier alternative would be to write

```
cout.flags(ios::hex+ios::left);
```

Setting flags in this way destroys any previous flag settings. The program in FIG-24.4 uses the flag settings to displays two integer values left justified within a 3 character field.

FIG-24.4

Using IO Formatting Flags

```
#include<iostream>
#include <iomanip>
using namespace std;

int main()
{
   //*** Set to hex, left-justified output ***
   cout.flags(ios::hex|ios::left);
   //*** Output values ***
   cout << setw(3) << 255 << endl;
   cout << setw(3) << 15 << endl;
}
```

Notice the use of the bitwise OR operator (`|`) to combine the required flags. This is simply an alternative to adding the flag values:

```
ios::hex+ios::left
```

Activity 24.2

Modify *TestIOS* to match the code in FIG-24.4. Test the program with and without the line that sets the flags.

long setf(long)

If we want to set flags without disturbing the current settings of other flags, we can use the `setf()` method. This returns the current flag setting as well as accepting new settings. For example,

```
cout.setf(ios::boolalpha)
```

would set the `boolalpha` flags without modifying any other flags.

Activity 24.3

Modify *TestIOS* to use `setf()` to output Boolean values in text format and add the lines

```
cout << true << endl
cout << false << endl;
```

Check the output with the `boolalpha` flag both set and unset.

long setf(long, long)

As we observed earlier, some flags are mutually exclusive. C++ ensures that only one of the flags in any of these groups is set at any one moment in time. To do this, it defines a set of group names as shown in FIG-24.5 and a second version of `setf()` which takes the flag to be set and the group name as parameters. Any previously set flag in that group is automatically reset.

FIG-24.5

Formatting Flag Groupings

Group Name	Flags in Group
adjustfield	left, right, internal
basefield	dec, oct, hex
floatfield	scientific, fixed

For example, the line

```
cout.setf(ios::hex,ios::basefield);
```

will set output to hexadecimal and reset either the `oct` or `dec` flags which may have been set previously.

long unsetf(long)

To reset one or more flags to zero, leaving the others untouched, we can use the `unsetf()` method. This returns the current state of all the flags and resets those specified by the parameter.



`streamsize` is an alias for a signed integer type

streamsize precision()
streamsize precision(streamsize)

The first version of this method returns the current number of characters to which real values are displayed. The second returns the current precision setting and sets a new

number of places. For example, we could have a real number displayed over five character positions using the line

```
cout.precision(5)
```

Exactly how this affects the displayed value depends on which (if any) `floatfield` flag is set. For example, if the default `floatfield` option is selected (and, strangely, this is not the same as either `fixed` or `scientific`), then the precision value represents the maximum number of numeric digits displayed. For example, the code

```
cout.precision(4);
cout << 1.2 << endl;
cout << 1.26789 << endl;
cout << 124.2 << endl;
cout << -124.2 << endl;
cout << 12345.2 << endl;
```

displays the values



The default value for `adjustfield` is `left`.

```
1.2
1.268
124.2
-124.2
1235e+04
```

With the `floatfield` set to `fixed`, or `scientific`, the precision becomes the exact number of numeric digits after the decimal point, meaning that the code

```
cout.setf(ios::fixed, ios::floatfield);
cout.precision(4);
cout << 1.2 << endl;
cout << 1.26789 << endl;
cout << 124.2 << endl;
cout << -124.2 << endl;
cout << 12345.2 << endl;
```

creates the output

```
1.2000
1.2689
124.2000
-124.2000
12345.2000
```

streamsize width()
streamsize width(streamsize)

The current minimum number of character positions assigned to a value sent to an output stream value can be found using the first version of `width()` while the second returns the current width setting and assigns a new width setting.

This represents the minimum width, so, if a value requires more space, its requirements override the width setting. Where the value being output requires less than the specified width, the unused positions will be filled with the current fill character (by default this is the space character).

The width setting is used on a single value only. So, if required for several values, the `width()` method must be called for each. For example, the code

```
cout.setf(ios::fixed, ios::floatfield);
cout.precision(4);
cout.width(10);
cout.setf(ios::right, ios::adjustfield);
cout << 1.2 << endl;
cout.width(10);
```

```
cout << 1.26789 << endl;
cout << 124.2 << endl;
cout << -124.2 << endl;
cout << 12345.2 << endl;
```

creates the output

```
ΔΔΔΔ1.2000
ΔΔΔΔ1.2679
124.2000
-124.2000
12345.2000
```

seekdir – Stream Positioning Options

This is a public, enumerated type used later in derived classes to specify a position within a stream from which characters are to be read or written. The named values are specified in FIG-24.6:

FIG-24.6

Stream Positioning Options

Name	Meaning
beg	(begin) The start of the stream.
cur	(current) The current position in the stream.
end	(end) The end of the stream.

iostate – IO State Flags

All IO streams maintain a set of bits to record any error conditions which may occur. The `iostate` type defines these named flag bits. These four flags are shown in FIG-24.7:

FIG-24.7

IO State Flags

Flag	Description
badbit	Set when an error has occurred in an IO operation.
failbit	Set when an error has occurred because of a logic error in the IO operation.
eofbit	Set when the end-of-file marker is reached during an input operation.
goodbit	Set when an IO operation has been successful.

Derived classes contain methods giving us access to the state of these flags – as we will see shortly.

openmode – Stream Opening Modes

This defines a set of named flags used to define how file streams are to be opened. The named values are shown in FIG-24.8.

FIG-24.8

Stream Opening Modes

Mode	Description
app	(append) Set stream position indicator to end of stream before each I/O.
ate	(at end) Set stream position indicator to end of stream when opening.
binary	(binary) Assume contents are in binary rather than text format.
in	(input) Allow read operations from stream.
out	(output) Allow write operations to the stream.
trunc	(truncate) Discard current contents of stream.

These flags can be used in combination. For example, the term

```
ios::out|ios::app
```

will open a file for output, adding any new data after existing file data.

We'll see these flags used in later classes.

ios Class

The `ios` class inherits the features of `ios_base` and adds the following:

char fill()
char fill(char)

The first version of the method returns the current fill character - used to fill unused character positions (typically when `cout.width()` has defined an over-large space for the value being output).

By default, the space character is used but this can be changed by using the second version of `fill()`. For example using the asterisks as the fill character in the following code

```
cout.setf(ios::fixed, ios::floatfield);
cout.precision(4);
cout.width(10);
cout.fill('*');
cout.setf(ios::right, ios::adjustfield);
cout << 1.2 << endl;
cout.width(10);
cout << 1.26789 << endl;
```

produces the output

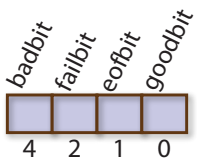
```
****1.2000
****1.2679
```

iostate rdstate()

The value of the *IO state flags* defined in the `ios_base` class can be retrieved using the `rdstate()` method. Each flag has an associated value (see FIG-24.9) and the value returned by `rdstate()` is a total of the value of those flags currently set to 1.

FIG-24.9

IO State Flag Values



The program below reads and displays the error state flags after the first output and after the first input. Since both are likely to have executed without a problem, the output produced in both cases will be zero (the value of `goodbit`).

FIG-24.10

Interrogating IO State Flags

```
#include<iostream>
using namespace std;

int main()
```

FIG-24.10
(continued)

Interrogating IO State Flags

```
{
   //*** Display iostate flags after output ***
   cout << "Performing output" << endl;
   ios::iostate returned;
   returned = cout.rdstate();
   cout << "State after first output " << returned << endl;
   //*** Display iostate flag after input ***
   int num;
   cout << "Enter number ";
   cin >> num;
   returned = cin.rdstate();
   cout << "State after first input " << returned << endl;
}
```

Activity 24.4

Start a new project called *TestRDstate* and implement the code given above.

What output is produced when the following test data is entered:

a) 12
b) X
c) Ctrl+Z (Windows only)

bool bad() , bool eof(), bool fail()

Rather than finding the state of all the IO flags with `rdstate()`, we can check the state of individual flags using the above methods. Each returns *true* if the flag in question is set. However, `fail()` returns *true* if either the `failbit` or `badbit` flags are set.

bool good()

This method returns *true* if none of the other state flags (`badbit`, `failbit` or `eofbit`) are set.

Activity 24.5

Modify *TestRDstate* so that rather than display the value returned by `rdstate()`, the program displays *true* or *false* for each of the flags. Use code such as

```
cout << "Goodbit : " << cin.good() << endl;
```

Test your program with the same data as in the previous Activity:

a) 12
b) X
c) Ctrl+Z (Windows only)

clear(iostate state = goodbit)

This method clears all error state flags before setting those given in the parameter (default is for `goodbit` to be set).

setstate(iostate state)

This method sets the error state flags specified in *state*. For example,

```
cout.setstate(ios::failbit)
```

would set the *failbit* flag. Other error flags are left unchanged.

ostream tie()*
ostream tie(ostream* t)*

When an `ostream` object is tied to an `istream` object, that `ostream` object is always flushed before the `istream` object attempts to extract a value from its stream.

By default `cout` and `cin` are tied. This means that any output requested by the `cout` (this is often a prompt message) will take place before `cin` reads from the standard input device (normally the keyboard). So, when a program contains the lines

```
cout << "Please enter a number ";
cin >> num;
```

we are guaranteed to see the prompt message before we are expected to type in a value.

The method's first format returns a pointer to the `ostream` object tied to the specified `istream` object. For example,

```
ostream * ptr = cin.tie();
```

would return a pointer to `cout`.

The second format changes the tied `ostream` object to *t* and returns the previously tied `ostream` object.

streambuf rdbuf()*
streambuf rdbuf(streambuf* nb)*

Every buffered stream has an associated memory buffer. The first version of `rdbuf()` will return a pointer to that buffer. The second version returns a pointer to the current buffer while setting *nb* as the new buffer area. We'll look at an example of this when describing one of the classes derived from `ios`.

iostate exceptions()
exceptions(iostate excpt)

IO streams are capable of throwing exceptions of type `ios_base::failure` when specified `iostate` flags (such as `badbit` and `failbit`) are set. The set of flags which cause the exception to be thrown is known as the **exception mask**. By default, no flags are specified, meaning that any problem with IO will not throw an exception, instead being content to set the appropriate `iostate` flags.

The first version of `exceptions()` will return the current setting of the exception mask. By default this will return zero since no flags have been specified to cause an exception to be thrown.

To change which flags throw an exception, we can use the second version of the

method. This second version also returns the previously defined set of flags. For example, we could write

```
cin.exceptions(ios::badbit|ios::failbit);
```

to have the `badbit` and `failbit` flags, when set by an operation involving `cin`, cause an exception to be thrown.

The `ios_base::failure` exception is a class containing two public methods:

`what()`	(inherited from the `exception` class) which returns a string giving details of the problem.
`code()`	which returns an error code.

The program given below demonstrates how to enable exceptions for `cin` when either the `badbit` or `failbit` flags are set.

FIG-24.11

IO State Flag Exceptions

```
#include <iostream>
using namespace std;

int main()
{
   //*** Display the value of the flags         ***
   //*** which cause cin to throw an exception ***
   cout << cin.exceptions() << endl;

   //*** Allow failbit and badbit to throw exceptions ***
   cin.exceptions(cin.failbit|cin.badbit);

   //*** Display the value of the flags         ***
   //*** which cause cin to throw an exception ***
   cout << cin.exceptions() << endl;

   //*** Read in a number ***
   int num;
   try
   {
      cout << "Enter number ";
      cin >> num;
   }

   //*** Catch any exception thrown ***
   //*** and display details        ***
   catch (istream::failure er)
   {
      cout << er.what() << endl;
      cout << er.code() << endl;
      cout << "Input failed\n";
   }
}
```

Activity 24.6

Start a new project called *TestIOExceptions* and implement the code given above.

Test the program by entering the value *X* when prompted for a number. What values are displayed?

operator ! and operator bool

These two operators are useful when our code needs to check if a file has been successfully attached to a stream. We'll see how these are used later in this chapter.

istream Class

The `istream` class is the first which we can actually use to create objects. In fact, C++ automatically creates one object of this class. This is `cin` which we have already met in most of our previous programs.

This object is normally designed to read standard characters (not wide characters) from an external source such as a keyboard.

istream& operator >>

We have made extensive use of this **extraction** (or input) operator (`>>`) in most of our programs. It is overloaded within the `istream` class to handle all the core C++ types such as `int`, `float`, `char`, etc.

The method returns a reference to the `istream` object used (normally `cin` when used with a `istream` object rather than an object of a derived class).

int get()
istream& get(char& ch)
istream& get(char st, streamsize n, char term='\n')*

Version 1 of this method extracts a single character from the input stream and returns the integer value of the character extracted. For example,

```
int chcode = cin.get();
```

Version 2 stores the character in *ch* and returns a reference to the `istream` object used. For example,

```
char ch;
cin.get(ch); //Returned istream reference is ignored
```

Version 3 allows us to read up to (*n*-1) characters into *st* – the last character position is automatically filled with a null character (\0) to terminate the string. For example,

```
char name[10];
cin.get(name,8);
```

will read up to 7 characters into *name*.

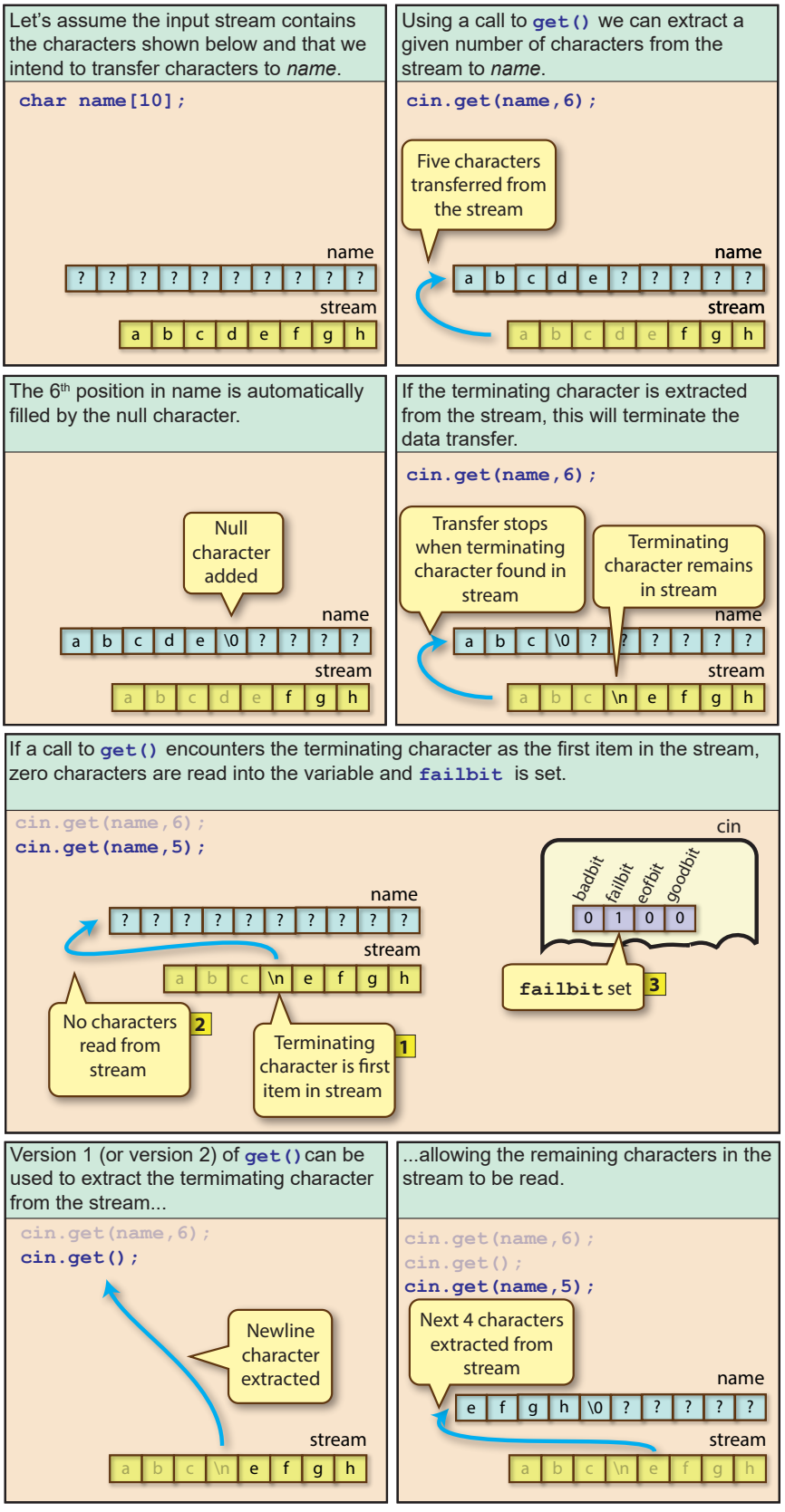
The optional final parameter, *term*, specifies a terminating character. If this character is encountered in the input stream before *n-1* characters are read, input will terminate. This terminating character is not extracted from the input stream.

If no characters are read, `failbit` is set. This could occur if the terminating character is the first character in the stream.

The basic operation of this third version of `get()` is visualised in FIG-24.12.

FIG-24.12

How *get()* Operates



By changing the terminating character we can accept a string typed in over several lines. For example,

```
cin.get(name,8,'*');
```

will accept up to 7 characters but stop when an asterisk is encountered. If we were to enter

```
a
b
c*
```

name would contain the string

```
a\nb\nc\0
```

Note that the **newline** characters (created by pressing the **Enter** key) after each of the first two letters are stored as part of the string.

The program in FIG-24.13 demonstrates how the state flags are affected when the terminating character is encountered in the input stream.

FIG-24.13

Using *get()*

```
#include<iostream>
using namespace std;

void main()
{
   char ch;
   char name[10];

   //*** Place some characters in the keyboard stream ***
   //*** and read the first character                 ***
   ch = cin.get();
   //*** Display value read ***
   cout << "\nValue read (first character): " << ch << endl;
   //*** Display state flag values ***
   cout << "State Flag Settings (after one char read):\n";
   cout << "badbit = " << cin.bad() << endl;
   cout << "failbit = " << cin.fail() << endl;
   cout << "eofbit = " << cin.eof() << endl;
   cout << "goodbit = " << cin.good() << endl;

   //*** Read set of characters ***
   cin.get(name,6,'*');
   //*** Display value read ***
   cout << "\nValue read (first string): " << name << endl;
   //*** Display state flag values ***
   cout << "State Flag Settings:\n";
   cout << "badbit = " << cin.bad() << endl;
   cout << "failbit = " << cin.fail() << endl;
   cout << "eofbit = " << cin.eof() << endl;
   cout << "goodbit = " << cin.good() << endl;

   //*** Read a single character from stream ***
   //ch = cin.get();
   //*** Display value read ***
   //cout << "\nValue read (second character): " << ch << endl;

   //*** Read another set of characters ***
   cin.get(name, 5, '*');
   //*** Display value read ***
   cout << "\nValue read (second string): " << name << endl;
```



FIG-24.13
(continued)

Using *get()*

```
    //*** Display state flag values ***
    cout << "State Flag Settings:\n";
    cout << "badbit = " << cin.bad() << endl;
    cout << "failbit = " << cin.fail() << endl;
    cout << "eofbit = " << cin.eof() << endl;
    cout << "goodbit = " << cin.good() << endl;
}
```

Activity 24.7

Start a new project called *TestIstream01* and implement the code given above.

Using the input *abcd*efghij,* is the second part of the character sequence read successfully?

Remove the comment characters (`//`) from the `ch =` and `cout` statements between the two string reads. How does this affect the program execution when using the same data as before?

getline(char st, streamsize n, char term = '\n')*

This method is almost identical in operation to version 3 of `get()` but differs in that the terminal character is extracted and discarded. A typical statement would be:

```
cin.getline(name,8);
```

If *n-1* characters are read from the input stream without the terminating character (*term*) being encountered, the `failbit` flag is set.

Activity 24.8

Modify *TestIstream01* so that `getline()` is used to read characters into *name* in place of `get()`. How does this affect the output?

At the end of the existing code add another `ch= get();` statement in an attempt to read another character from the stream. What character is in *ch* after the `get()` statement is executed?

istream& read(char st, streamsize n)*

This method represents yet another variation on reading a block of characters from an input stream. Exactly *n* characters are read from the stream and placed in *st*. No null character is appended to *st*.



EOF will normally only be encountered when the stream is linked to a file.

If there are insufficient characters in the stream before the end-of-file is reached, extracted characters will be stored in *st* and `eofbit` and `failbit` will be set.

streamsize gcount()

This method returns the number of characters extracted by the last call to `get()`, `getline()`, or `read()`. This count does not include the terminal character.

The program in FIG-24.14 uses `read()` and `gcount()` to read a sequence of characters from the keyboard and display various details.

FIG-24.14

Input Using *read()*

```
#include<iostream>
using namespace std;

void main()
{
   char name[10];
   //*** Read characters from keyboard ***
   cin.read(name, 8);
   //*** Find out how many characters were read ***
   int read = cin.gcount();
   //*** Add a null character to name ***
   name[read] = '\0';
   //*** Read the next character ***
   char ch;
   cin.get(ch);
   //*** Display the string read and the count ***
   cout << "Data read was " << name << endl;
   cout << read << " characters were read from the stream\n;
   //*** Display the next character read ***
   cout << "Next character read was " << ch << endl;
}
```

Activity 24.9

Start a project called *TestIstream02* and implement the code given above. Test the program by entering the data:

a) *abcdefghi*

b) *abcd*
efgh

streamsize readsome(char st, streamsize n)*

Whereas `read()` will wait for sufficient characters to be entered when reading from the keyboard, `readsome()` will not. Instead it extracts only those characters already in the stream and returns the number of characters it has managed to read from the stream. The program in FIG-24.15 demonstrates the use of this method.

FIG-24.15

Input Using *readsome()*

```
#include<iostream>
using namespace std;

void main()
{
   char name[10];

   //*** Place some characters in the keyboard stream ***
   //*** and read the first character                 ***
   cout << "Enter text ";
   cin.get();

   //*** Attempt to read another 8 characters from the stream ***
   int read = cin.readsome(name, 8);
   //*** Add a null character to name ***
   name[read] = '\0';

   //*** Display the string read and the count ***
   cout << "Data read was " << name << endl;
   cout << read << " characters were read from the stream\n";
}
```

Activity 24.10

Start a new project called *TestIstream03* and implement the code given above. Test the program using:

a) *abc*
b) *abcdefghijkl*

int peek()

This method returns a copy of the first character available in the stream without actually removing it from the stream. For example, let's assume we have typed in the sequence

```
abcd
```

then the code

```
//*** Look at the first character without extracting it ***
ch1 = static_cast<char>(cin.peek());
//*** Now extract the character ***
cin.get(ch2);
//*** Display both values ***
cout << ch1 << "   " << ch2 << endl;
```

would display

```
a  a
```

istream& unget()

This method returns the last character read to the stream. If the method fails for any reason, `failbit` is set.

istream& putback(char ch)

This method places *ch* at the head of the stream, making it the first character to be read by any subsequent extraction. Normally, the value of *ch* should be that of the last character extracted from the stream, however, certain stream types such as the keyboard and file streams allow *ch* to be some other value. If problems occur, this method may set the `badbit` or `failbit` flags. FIG-24.16 demonstrates the effects of `peek()`, `unget()` and `putback()`.

FIG-24.16

How *peek()*, *unget()* and *putback()* Operate

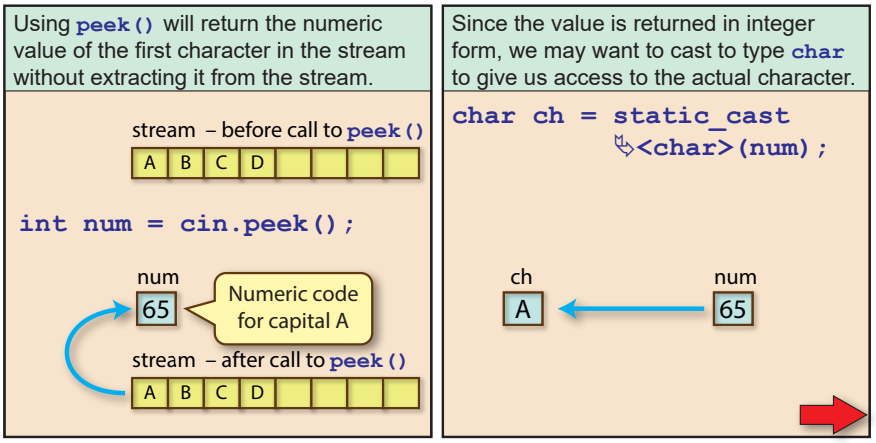
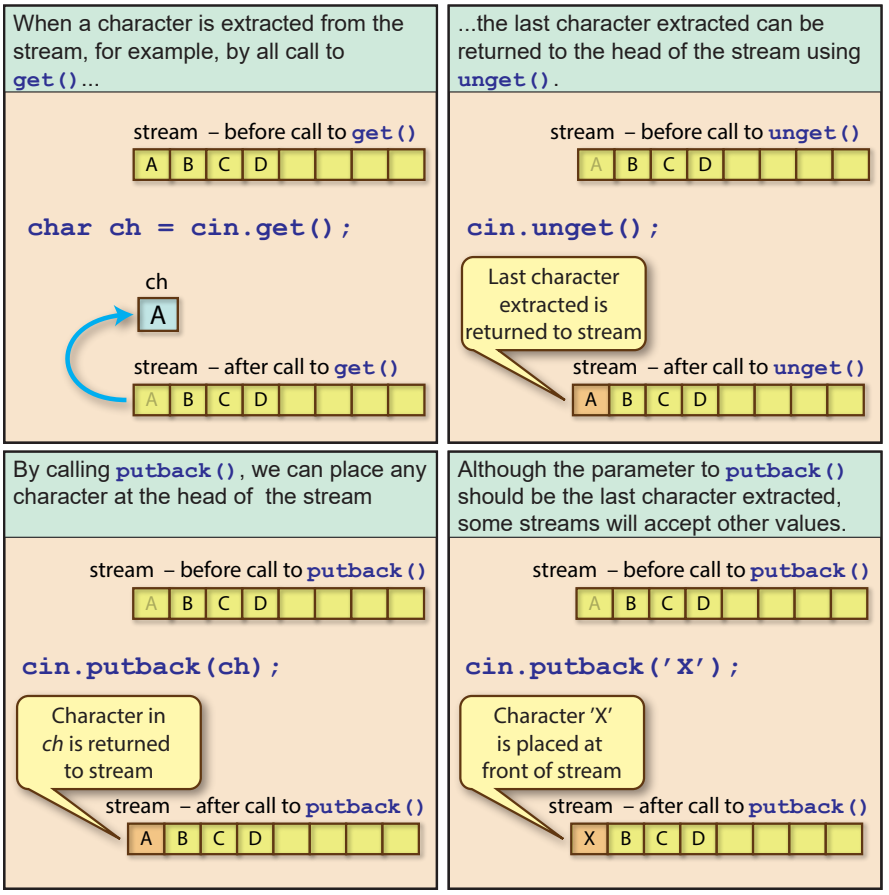


FIG-24.16
(continued)

How *peek()*, *unget()* and *putback()* Operate



In the program below we demonstrate the use of `peek()`, `unget()` and `putback()`.

FIG-24.17

Using *peek()*, *unget()* and *putback()*

```
#include<iostream>
using namespace std;

void main()
{
   char ch;
   char name[10];

   //*** Place some characters in the keyboard stream ***
   //*** and read the first character                  ***
   ch = cin.get();
   //*** Display character read ***
   cout << "Character read from stream : " << ch << endl;
   //*** Look at the first character in the stream ***
   cout << "First character in input stream is " <<
   ↳static_cast<char>(cin.peek()) << endl;
   //*** Return character read to input stream ***
   cin.unget();
   //*** Look at the first character in the stream ***
   cout << "First character in input stream is " <<
   ↳static_cast<char>(cin.peek()) << endl;
   //*** Push a new character at head of stream ***
   cin.putback('X');
```

FIG-24.17
(continued)

Using *peek(), unget() and putback()*

```
   //*** Look at the first character in the stream ***
   cout << "First character in input stream is " <<
   ↳static_cast<char>(cin.peek()) << endl;
   //*** Read the characters in the stream ***
   cin >> name;
   //*** Display characters read ***
   cout << "Characters read from stream : " << name << endl;
}
```

Activity 24.11

Start a new project called *TestIstream04* and implement the code given above.

Test the program using the input *abcd*.

istream& ignore(streamsize n = 1, int term = EOF)

This method extracts and discards *n* characters from the stream, stopping earlier if the character *term* is encountered. The *term* character is extracted. By default *term* is set to EOF (the end-of-file event).

The program below demonstrates the use of this method, by reading the first character in the keyboard stream, ignoring the next four, then reading the next character before displaying both of the characters that have been read.

FIG-24.18

Using *ignore()*

```
#include<iostream>
using namespace std;

void main()
{
   char ch1, ch2;
   //*** Place some characters in the keyboard stream ***
   //*** and read the first character                 ***
   cout << "Enter text ";
   cin.get(ch1);

   //***Ignore the next four characters ***
   cin.ignore(4);

   //*** Read the character at this position ***
   cin.get(ch2);

   //*** Display characters read ***
   cout << "Characters read were " << ch1 << " and " <<ch2 << endl;
}
```

Activity 24.12

Start a new project called *TestIstream05* to match the code given above.

Test the program by entering the data *a1234b*. What values are displayed?

Change the code so that four characters are ignored with an asterisk as the ignore terminator value.

Now run the program using the data *a12*4b*. What values are displayed?



streampos is, in effect, an alias for a signed integer type

istream& seekg(streampos pos)
istream& seekg(streamoof offset, seekdir start)

Version 1 of this method sets the position from which the next character will be extracted from the stream. The value of *pos* represents an offset from the first character in the stream. The first character is at position zero.

Version 2 sets the position from which the next character will be read to *offset*, with this new position being relative to the specified *start* position (start can be **beg**, **cur** or **end**).

Note that although **seekg()** is defined within the **istream** class it doesn't operate on the keyboard stream normally associated with **cin**. In other words, a statement such as

```
cin.seek(3); // Move pointer to fourth character in stream
```

won't work.

streampos tellg()

This method returns the position in the stream from which the next character will be extracted. Unlike **seekg()**, **tellg()** does operate correctly with **cin**.

Although we will wait until a later part of this chapter before demonstrating the use of **seekg()** and **tellg()**, FIG-24.19 (see below) shows how each of those methods operate.

FIG-24.19

How *seekg()* and *tellg()* Operate

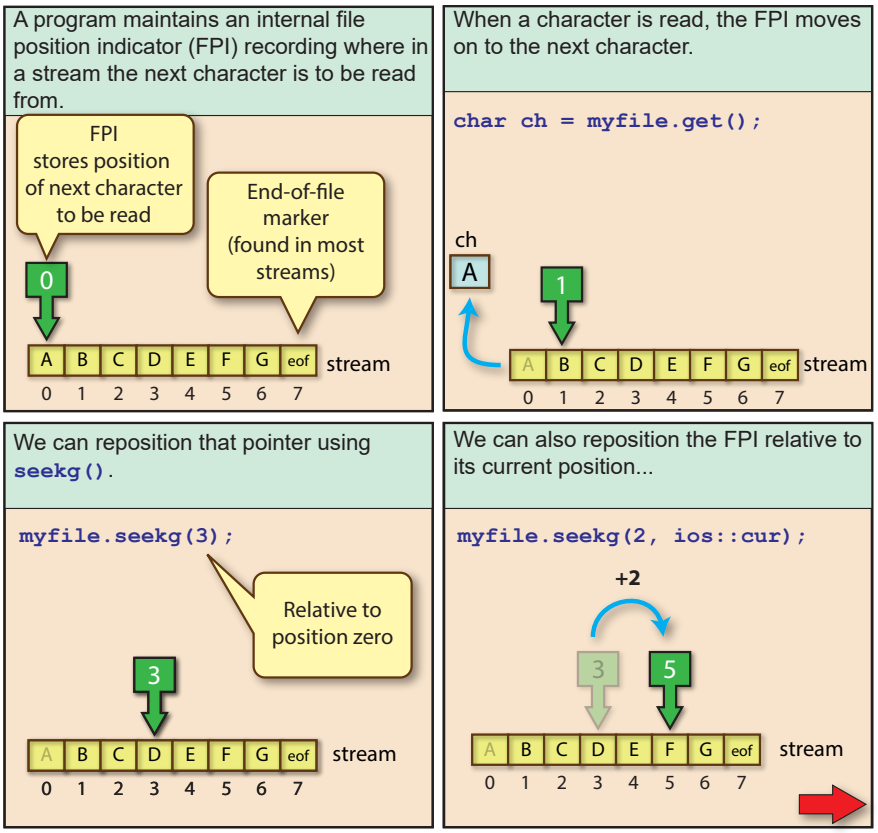
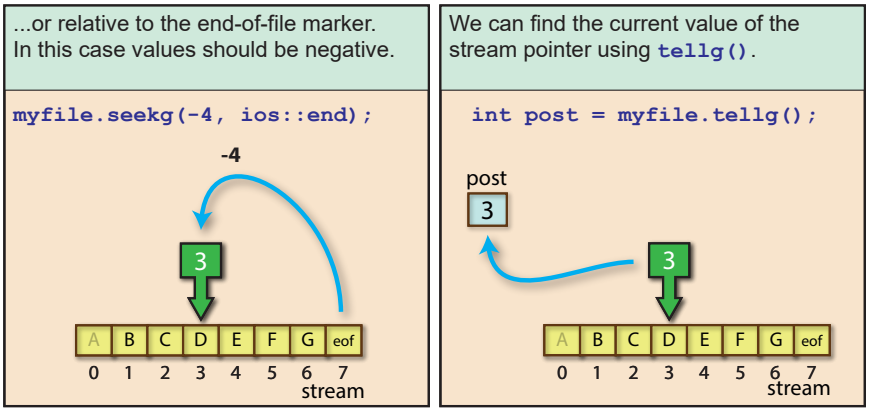


FIG-24.19
(continued)

How *seekg()* and *tellg()* Operate



The *ostream* Class

The `ostream` class, like `istream`, is derived from the `ios` class.

C++ automatically defines three objects of this class for every C++ program. One of these we have already used in all our programs: `cout`. The others are `cerr`, designed to handle error messages and `clog` for logging text.

As a general rule, all three objects will default to the same output device – the screen, but we are free to redirect each object to other devices or files.

By default, `cerr` is tied to the same output stream as `cout` (that is, to the screen), meaning `cout`'s buffer is flushed each time data is written to `cerr`.

ostream& operator<<

Like the extraction operator (<<), the **insertion** (or input) operator has been used in most of our programs. It is overloaded to handle all the core C++ types such as `int`, `float`, `char`, etc. The method returns a reference to the `ostream` object used (normally `cout`).

ostream& put(char ch)

This method adds character *ch* to the stream and returns a reference to the stream being used. For example, the line

```
cout.put('x');
```

would output the character *x* to the screen.

It is possible for this method to set `failbit` or `badbit` if output fails (normally, this would only happen when output is directed to another device or file).

ostream& write(char st, streamsize n)*

This method outputs *n* characters from *st* to the stream. *st* must contain at least *n* characters. A reference to the stream being used is returned.

ostream& flush()

This method forces the contents of the stream buffer to be emptied and sent to the

device or file to which it is linked. A reference to the stream being used is returned.

ostream& seekp(streampos pos)
ostream& seekp(streamoof offset, seekdir start)

We have already seen that a file maintains a file position indicator (FPI) to indicate the position in the file from which the next character should be read. A second file position indicator is also maintained which gives the position of the next character to be written.

The value stored in this write position indicator can be set using the `seekp()` method. A value of zero indicates the start of the file.

The indicator can be positioned relative to the start of the file (`ios::beg`), the current position (`ios::cur`) or the end of the file (`ios::end`).

streampos tellp()

This method returns the position in the stream from which the next character will be written.

As with the `seekg()` and `tellg()` we encountered in the `istream` class, these methods are more appropriately used when handling files – as we will discover shortly.

The *ofstream* Class

This class is designed for use with output files - normally stored on disk or flash memory. In order to use objects of this class, our programs must include the line

```
#include <fstream>
```

Constructor

The class's constructor can be used to specify the name of the file to which the `ofstream` object is to be linked. For example,

```
ofstream myfile("Testdata.txt");
```

The constructor also allows an open mode to be specified (see FIG-24.20 for options).

FIG-24.20

File Access Mode Options

Mode	Description
app	(append) Set stream position indicator to end of stream before each I/O.
ate	(at end) Set stream position indicator to end of stream when opening.
binary	(binary) Assume contents are in binary rather than text format.
in	(input) Allow read operations from stream.
out	(output) Allow write operations to the stream.
trunc	(truncate) Discard current contents of stream.

The open mode defaults to `ios::out` but other options can be added. For example,

```
ofstream myfile("Testdata.txt", ios::out|ios::app);
```

A final option is to call the constructor without parameters:

```
ofstream myfile;
```

open (const char filename, openmode mode = ios::out)*
open (const string filename, openmode mode = ios::out)

This method is used when the `ofstream` object has been created without specifying the file to which it is to be linked or when we want to change that link to a different file.

If the file named fails to open, or is already open, `failbit` is set. If the operation is successful, all error flags are set to zero (that is, `good()` will return true).

The *filename* parameter is a string giving the name of the file to be opened. This may include path information. If the file does not exist, it will be created automatically.

The *mode* parameter is used to set one or more flags which determine how the file is used.

A typical use of this method would be:

```
ofstream myfile;
myfile.open("Testdata.txt");
```

If the file does not exist it will be created. If it does exist, its current contents will be deleted.

To append new data to the existing data already held in the file we could open the file with the statement

```
myfile.open("Testdata.txt", ios::out | ios::app)
```

Opening a file – if successful – associates the stream with a file and creates a buffer area where the data to be written to the file is held before a physical write to the file occurs.

Checking that a File has been Opened

Now we can use the inherited `!` operator to check for a failure:

```
if(!myfile)
   cout << "File failed to open for writing\n";
```

Alternatively, we can use the inherited `bool` operator to check for success:

```
if(myfile)
   cout << "File opened successfully\n";
```

Another approach to checking that a file has been opened correctly is to use the method `is_open()`.

bool is_open()

This method returns *true* if the call to `open()` has been successful, otherwise it returns *false*. Hence rather than write `if(myfile)`, we can use the expression

```
if (myfile.is_open())
   ...
```

close()

When operations on a file are complete, the file must be closed in order to free up resources and ensure any data in the file's memory buffer is physically written to the file. A typical statement would be

```
myfile.close();
```

Attempting to close a file which is not open will set `failbit`.

Writing to the File

Methods `opeartor<<()`, `put()` and `write()` which the class has inherited from `ostream`, allow us to write to the file. So we might write to *myfile* using statements such as

```
myfile << 12 << "  " << 3.14;
myfile.put('X');
myfile.write("abcdef",6);
```

The program in FIG-24.21 demonstrates opening, writing to, and closing a file.

FIG-24.21

Outputting to a Text File

```
#include<iostream>
#include<fstream>
using namespace std;

void main()
{
   ofstream myfile;
   //*** Attempt to open file ***
   myfile.open("datafile.txt");
   //*** If successful, write to file ***
   if (myfile)
   {
      myfile << 12 << "  " << 3.14;
      myfile.put('X');
      myfile.write("abcdef", 6);
      cout << "Data written to file\n";
   }
   //*** Close the file ***
   myfile.close();
   cout << "File closed\n";
}
```

Note that the code sends additional text to the screen to allow the user to see what is happening within the program.

Activity 24.13

Start a new project called *TestTextFile* containing the code given in FIG-24.21.

Run the program and use *Notepad* to examine the contents of the *datafile.txt*.

Modify the program so that new data is appended to the existing file contents. Comment out the `myfile << 12`... and `myfile.put('X');` lines from the program and change the call to `myfile.write()` so that the string "*ghijkl*" is written to the file.

Use *Notepad* to check that the file now contains both the original and new data.

Buffers

When a stream is connected to a physical file, a **buffer area** is maintained within internal memory. So, when our program instructs data to be written to a file with a call to a method such as `put()` or `write()` that data is actually written to the buffer area.

This technique optimises the speed of operation since writing to main memory is many times faster than having to write to a disk. Eventually, the contents of the buffer area will actually be written to the physical disk in a process known as **synchronisation**.

Data is written from the buffer to the file when that buffer is full or the file is closed. Execution of the `flush()` method or *endl* will also cause the buffer to be written to the file.

The *ifstream* Class

This class is designed for use with existing files whose data is to be read. Like `ofstream` objects, `ifstream` objects maintain a buffer area to minimise execution time. When we read from a physical file, a segment of data is copied from that file into the buffer area. So when a program executes a statement such as `get()`, `getline()` or `write()`, the required data is actually fetched from the buffer area. Only when the buffer area is empty is more data read from the actual file.

Constructor

The class's constructor can optionally be used to specify the name of the file to which the `ifstream` object is to be linked. For example,

```
ifstream myfile("Testdata.txt");
```

The constructor also allows an open mode to be specified (default is `ios::in`).

open(const char file, openmode mode = ios::in)*
open(const string filename, openmode mode =ios::in)

This method is used when the `ifstream` object has been created without specifying the file to which it is to be linked or when we want to change that link to a different file.

If the file fails to open (probably because it doesn't exist), or is already open, `failbit` is set. A typical statement would be

```
istream myfile:
myfile.open("("datafile.txt");
```

As with `ofstream` objects, the `open()` method has an optional parameter to specify the opening mode. Generally, the default `ios::in` option is all that is required when reading from a text file.

bool is_open()

This method returns *true* if the call to `open()` has been successful, otherwise it returns *false*. Hence rather than write `if(myfile)`, we can use the expression

```
if (myfile.is_open())
```

close()

When operations on a file are complete, the file should be closed. A typical statement would be

```
myfile.close();
```

Attempting to close a file which is not open will set `failbit`.

Note that the file will be closed automatically when the `ifstream` object to which it is linked is destroyed.

Checking that a File has been Opened

Exactly the same methods are available for this class as we saw for the `ofstream` class. That is:

```
operator!()
operator bool()
is_open()
```

To check that the file has been opened for reading, our program is likely to contain the code

```
if(myfile)
   //Read from file
```

Reading from the File

The inherited methods

```
operator>>()
get()
getline()
read()
```

are normally used to read from the file. For example, we can read a single character from the file using the line

```
char ch = myfile.get();
```

When reading an unknown amount of data from a file, our code will typically employ the following logic:

```
Open file
Attempt a read from file
WHILE end-of-file not detected DO
    Handle the data read
    Attempt a read from file
ENDWHILE
Close the file
```

Character Extraction Positioning

The inherited method `seekg()` allows us to reposition where within the file reading is to start. While `tellg()` informs us of the position in the stream of the next character to be read.

The program in FIG-24.22 complements the previous program. The code reads back the data held in the previously created *datafile.txt* one character at a time and displays that data on the screen.

FIG-24.22

Reading from a Text File

```
#include <iostream>
#include<fstream>
using namespace std;

int main()
{
   //*** Read data from file ***
   char ch;
   ifstream myfile2;
   //*** Attempt to open file ***
   myfile2.open("..\\TestTextFile\\datafile.txt");
   //*** If successful, display contents ***
   if (myfile2)
   {
      cout << "File opened successfully for reading\n";
      //*** Read and display each character in the file ***
      myfile2.get(ch);
      while (!myfile2.eof())
      {
         cout << ch;
         myfile2.get(ch);
      }
      //*** Move cursor to new line ***
      cout << endl;
      //*** Close the file ***
      myfile2.close();
   }
   else
      cout << "File not found\n";
}
```

Note that the call to `myfile2.open()` includes path information to the existing file with a single backslash character represented by two backslash characters (backslash characters are formed using an escape code – see Chapter 4 FIG-4.9).

Activity 24.14

Start a new project called *TestTextFile2* and implement the code given above. Run the program and check that the contents of the file are displayed.

Should we want to skip some of the characters in the file, for example, let's say we only want to display every second character, we could make use of the `tseekg()` method using the code

```
myfile2.get(ch);
while (!myfile2.eof())
{
   cout << ch;
   //*** Skip a character ***
   myfile2.seekg(1, ios::cur);
   //*** Read next character ***
   myfile2.get(ch);
}
```

Activity 24.15

Modify *TestTextFile2* so that it displays every other character.
Run the program and check that the correct characters are displayed.

We can make use of the `tellg()` method to discover the length of the file by skipping to the end of the file using

```
myfile2.seekg(0,ios::end);
```

and then finding the current position in the file:

```
int bytes = myfile2.tellg();
```

Activity 24.16

Modify *TestTextFiles2* so that, after the original output, the number of characters contained in the file is displayed. HINT: The `eofbit` will be set when the code's `while` loop exits.

Separating Data Items

Just as we did back in Chapter 14, we need a way of separating the individual data elements in a field. This easiest option in most cases is simply to insert a newline character (\n) between each value. To achieve this with the sample data already used in *DataFile.txt*, we could change the output lines to

```
myfile << 12 << '\n' << "  \n" << 3.14 << '\n';
myfile.put('X');
myfile.put('\n');
myfile.write("abcdef\n", 7);
myfile.write("ghijkl\n", 7);
```

Reading the data can then be achieved using the lines

```
//*** Read a line from the file ***
myfile2.getline(line, 100);
//*** WHILE not eof DO ***
while (!myfile2.eof())
{
   //*** Display the value ***
   cout << "Data read : " << line << endl;
   //*** Read next value from the file ***
   myfile2.getline(line, 100);
}
```

Activity 24.17

Modify *TestTextFiles* so that a newline character is inserted after each data value as shown above.

Modify *TestTextFiles2* so that, rather than read one character at a time, one line of data (everything up to the next newline character) is read and displayed. How many bytes are reported to be in the file this time?

Using Binary Files

Writing to a Binary File

As explained in Chapter 14, using binary files can create shorter more efficient files at the cost of being mostly unreadable without the use of a program designed specifically for that purpose.

With a binary file, data values are written in exactly the same format as they are stored in memory. In order to read that data back from the file we must remember the order and type of each value we have written. For example, let's say we want to save the name and high score of a video game player, then the data written would consist of:

name (char[ ] type)
score (long int type)

The simplest way to do this is to start by defining a record structure for the two fields:

```
struct HighScoreType
{
   char name[30];
   int score;
};
```

We can set up our test data in an array of the above type:

```
HighScoreType scoredata[] = { {"Esther Summerson",2001},{
↳"John Jarndyce",1905},{"Ada Clare",1897} };
```

We can use `ofstream`'s `write()` method to write each record to the file. For example, we might attempt to write the first record with the line

```
hscorefile.write(&scoredata[0], sizeof(HighScoreType));
```

However, there's a slight problem: `write()` expects its first parameter to be of type `const char*` (in other words, the start address of a C-style string constant) whereas we have specified the start address of a *HighScoreType* record.

We could try a static cast to correct this problem:

```
hscorefile.write(static_cast<const char*>( &scoredata[c]),
↳sizeof(HighScoreType));
```

Unfortunately the compiler objects to this attempted cast. The problem here is that `static_cast` only works on types that the compiler knows how to convert such as `char` to an `int`.

reinterpret_cast

C++ offers another type of cast using the term `reinterpret_cast`. In effect, this is a command to the compiler to re-interpret the bit pattern in the specified variable as a given type without any attempt to convert the format of those bits. As such, we can perform any conversion we wish. So, by changing our previous line to

```
hscorefile.write(reinterpret_cast<const char*>( &scoredata[c]),
↳sizeof(HighScoreType));
```

the compiler will now accept our attempt to write to the file.

FIG-24.23 contains a program which writes our three high score records to a binary file.

FIG-24.23

Writing to a Binary File

```
#include <iostream>
#include <fstream>
using namespace std;

struct HighScoreType
{
   char name[30];
   int score;
};
```



FIG-24.23
(continued)

Writing to a Binary File

```
int main()
{
   HighScoreType scoredata[] = { {"Esther Summerson",2001},{
   ↳"John Jarndyce",1905},{"Ada Clare",1897} };
   ofstream hscorefile;
   hscorefile.open("highscores.dat", ios::out | ios::binary);
   if (hscorefile)
   {
      for (int c = 0; c < 3; c++)
         hscorefile.write(reinterpret_cast<const char*>(
         ↳&scoredata[c]), sizeof(HighScoreType));
      cout << "Data written to file\n";
   }
   else
      cout << "File could not be opened\n";
}
```

Activity 24.18

Start a new project called *TestBinaryFile* and implement the code given above.

When the program has been executed, use *Notepad* to examine the contents of *highscores.dat*. What is unusual about the scores when the file is displayed using *Notepad*?

When examining the contents of the file in Notepad, the *name* field value in each record is exactly 30 characters in length. Also, the *score*'s `int` value displays as a set of strange characters. This is the 2's complement form of the original numbers.

In fact, we can be even more efficient when writing to the file. Assuming we know exactly how many records are to be written at the time of creating the code, we can reduce the code

```
for (int c = 0; c < 3; c++)
   hscorefile.write(reinterpret_cast<const char*>(
   ↳&scoredata[c]), sizeof(HighScoreType));
```

to the single line

```
hscorefile.write(reinterpret_cast<const char*>(scoredata),
3*sizeof(HighScoreType));
```

which, in our case, will write all three records in a single statement.

Activity 24.19

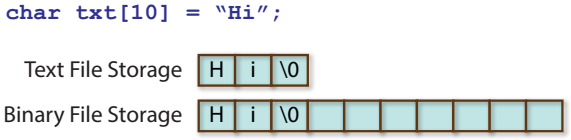
Modify *TestBinaryFile* so that all of the data is written to the file by the execution of a single statement.
Use *Notepad* to check that the file contents are exactly the same as before.

Although the difference between text and binary formats when it comes to storing how a numeric value is immediately obvious, the difference in how C-style strings are stored is more subtle.

In a text file, C-style strings occupy exactly the number of characters required by the data (including a terminating null character). In a binary file, a C-style string occupies the size of the array defined for the string (see FIG-24.24).

FIG-24.24

How Strings are Stored in Text and Binary Files



Although our first thoughts might be that, in the interest of storage requirements, a text file would be preferable, in fact, the binary file's approach means that every record stored is exactly the same number of bytes in length which makes it simpler to jump directly to a specific record within the file.

Reading from a Binary File

To read from a binary file, we'll use an `ifstream` object and its `read()` method – just as we did when reading from a text file. The program in FIG-24.25 reads and then displays the contents of the file *highscores.dat*.

FIG-24.25

Reading from a Binary File

```
#include <iostream>
#include <fstream>
using namespace std;

struct HighScoreType
{
   char name[30];
   int score;
};

int main()
{
   HighScoreType scoredata[3];
   ifstream hscorefile;
   //*** Open binary file for reading ***
   hscorefile.open("..\\TestBinaryFile\\highscores.dat", ios::in |
   ↳ios::binary);
   if (hscorefile)
   {
      //*** Read records into array ***
      int c = 0;
      hscorefile.read(reinterpret_cast<char*>(&scoredata[c]),
      ↳sizeof(HighScoreType));
      while (!hscorefile.eof() && c < 2)
         hscorefile.read(reinterpret_cast<char*>(&scoredata[++c]),
         ↳sizeof(HighScoreType));
      //*** Display the contents of the array ***
      for (c = 0; c < 3; c++)
         cout << scoredata[c].name << "  " << scoredata[c].score <<
         ↳endl;
   }
   else
      cout << "File could not be opened\n";
}
```

Notice that we again use the `reinterpret_cast` statement to convert the target variable where the read data is to be stored to the correct type – this time `char*`.

Activity 24.20

Start a new project called *TestBinaryFile2* and implement the code given above. Check that the data read from the file matches the original data.

If we know exactly how many records are stored in the file, we can use the same approach we employed when writing the records and read all of the data in a single statement:

```
hscorefile.read(reinterpret_cast<char*>(scoredata), 3*
↳sizeof(HighScoreType));
```

Activity 24.21

Modify *TestBinaryFile2* so that the complete content of the file is read in a single statement. Are the three high scores still displayed correctly?

We can jump to a specific record in a binary file using the `seekg()` method. For example, to read the final record in our *highscores.dat* file we could use the following statements:

```
hscorefile.seekg(-static_cast<int>(sizeof(HighScoreType)),
↳ios::end);
scorefile.read(reinterpret_cast<char*>(scoredata),
↳sizeof(HighScoreType));
```

Notice that the `seekg()` call is relative to the end of the file and therefore its first parameter must be a negative number. However, since `sizeof()` returns an `unsigned int` value, we need to convert this to a `signed int` before applying the unary minus at the start of the expression. In the second line, the value read from the file is placed in *scoredata[0]*.

Activity 24.22

Modify *TestBinaryFile2* so that only the last record in the file is read, placing the value in *scoredata[0]*. The value read should be displayed.

The *iostream* Class

This class inherits from both the `istream` class and `ostream` class but its main purpose is as the base class for the `fstream` class.

The *fstream* Class

The `fstream` class allows us to associate a buffered object with a file from which data may be both read and written.

Luckily, there are no new methods for this class since it inherits all the features it needs from its family of base classes.

To open a file for I/O we start by declaring an `fstream` object and using `open()` to open the associated file in read/write mode:

```
fstream hscorefile("..\\TestBinaryFile\\highscores.dat",
↳ios::in|ios::out);
```

When accessing the file we can use `seekg()` to access the correct position prior to a read statement and `seekp()` before a write statement.

The program in FIG-24.26 makes use of the *highscores.dat* file we created previously and adds 100 to all three records. The program uses the following logic:

```
Attempt to open binary file for I/O
IF file opened THEN
```

```
        Read first record into variable
        WHILE NOT end-of-file DO
            Add 100 to score in variable
            Move the file position write indicator to position of record just read
            Write variable back to file
            Move file position read indicator to position of next record
            Read record into variable
        ENDWHILE
        Close file
    ELSE
        File not found
    ENDIF
```

Program code:

FIG-24.26

Updating a File

```
#include <iostream>
#include <fstream>
using namespace std;

struct HighScoreType
{
   char name[30];
   int score;
};


int main()
{
   HighScoreType record;
   //*** Open file for I/O ***
   fstream hscorefile;
   hscorefile.open("..\\TestBinaryFile\\highscores.dat", ios::in |
   ↳ios::out| ios::binary);
   //*** If file opened ***
   if (hscorefile)
   {
      //*** Display message to user ***
      cout << "Updating file - please wait\n";
      //*** Read the first record ***
      hscorefile.read(reinterpret_cast<char*>(&record),
      ↳sizeof(HighScoreType));
      while (!hscorefile.eof())
      {
         //*** Add 100 to score ***
         record.score += 100;
         //*** Move write file position indicator back to start of
         ↳this record ***
         hscorefile.seekp(- static_cast<int>(sizeof(HighScoreType)),
         ↳ios::cur);
         //*** Write record back to file ***
         hscorefile.write(reinterpret_cast<const char*>(&record),
         ↳sizeof(HighScoreType));
         //*** Read next record ***
         hscorefile.seekg(hscorefile.tellp());
         hscorefile.read(reinterpret_cast<char*>(&record),
         ↳sizeof(HighScoreType));
      }
      hscorefile.close();
      cout << "File updated\n";
   }
   else
      cout << "File not found\n";
}
```

Activity 24.23

Start a new project called *TestBinaryFileUpdate* and implement the code given above.

Re-run *TestBinaryFile2* to check that the records are correctly updated.

Saving Objects

Looking back at the definition of *HighScoreType* in the previous coding examples it may seem strange that we have used a C-style string for the *name* field rather than a `string` object.

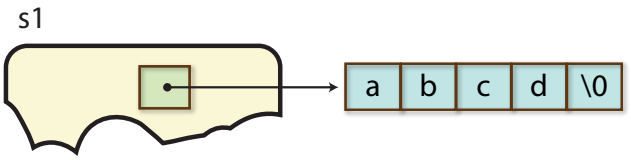
But saving objects which contain pointer attributes is more complex than saving regular types such as `int` and `char`.

The string class uses a pointer to reference the area where the actual sequence of characters held in the string object are stored (see FIG-24.27).

FIG-24.27

How *string* Objects Operate

```
string s1 = "abcd";
```



This means that if we try so save a `string` object to a file, all we end up saving is the address stored in that `string` object's pointer attribute and not the actual string data. The address in that pointer only has meaning during the current execution of the program, so when we attempt to read the string value back from the file, no meaningful data will be retrieved.

To solve this problem we need to convert the contents of the `string` object to a C-style string before saving the contents of our record and then convert the C-style string back to a `string` object when reading from the file. For example, assuming we have changed our *HighScoreType* to be defined as

```
struct HighScoreType
{
   string name;
   int score;
};
```

we can write the contents of the c^{th} record of an array to a binary file using the lines

```
//*** Convert string to C-string ***
strcpy_s(temp_name, scoredata[c].name.c_str());
//*** Write name and score to file ***
hscorefile.write((temp_name), 30);
hscorefile.write(reinterpret_cast<const char*>(
↳&scoredata[c].score), sizeof(int));
```

The modified version of *TestBinaryFile* where *name* is defined as a `string` object is given in FIG-24.28.

FIG-24.28

Saving Objects to a File

```
#include <iostream>
#include <fstream>
using namespace std;

struct HighScoreType
{
   string name;
   int score;
};

int main()
{
   HighScoreType scoredata[] = { {"Esther Summerson",2001},{
   ↳"John Jarndyce",1905},{"Ada Clare",1897}};
   ofstream hscorefile;
   hscorefile.open("highscores.bin", ios::out | ios::binary);
   if (hscorefile)
   {
      char temp_name[30];
      for (int c = 0; c < 3; c++)
      {
         //*** Convert string to C-string ***
         strcpy_s(temp_name, scoredata[c].name.c_str());
         //*** Write name and score to file ***
         hscorefile.write((temp_name), 30);
         hscorefile.write(reinterpret_cast<const char*>(
         ↳&scoredata[c].score), sizeof(int));
      }
      cout << "Data written to file\n";
      hscorefile.close();
   }
   else
      cout << "File could not be opened\n";
}
```

Note that output is to a file named *highscores.bin* rather than *highscores.bat*.

To read the data from the file back into an array of type *HighScoreType* we need to reverse the process, converting the C-style string extracted from the file to a string object value. The modified version of *TestBinaryFile2* required to achieve this is shown in FIG-24.29.

FIG-24.29

Reading Objects from a File

```
#include <iostream>
#include <fstream>
using namespace std;

struct HighScoreType
{
   string name;
   int score;
};

int main()
{
   HighScoreType scoredata[3];
   ifstream hscorefile;

   //*** Open binary file for reading ***
   hscorefile.open("..\\TestBinaryFile\\highscores.bin", ios::in |
   ↳ios::binary);
```



FIG-24.29
(continued)

Reading Objects from a File

```
   if (hscorefile)
   {
      //*** Read records into array ***
      int c = 0;
      char temp_name[30];
      int temp_score;
      hscorefile.read(temp_name, 30);
      hscorefile.read(reinterpret_cast<char*>(&temp_score),
      ↳sizeof(int));
      while (!hscorefile.eof())
      {
         hscorefile.read(temp_name, 30);
         hscorefile.read(reinterpret_cast<char*>(&temp_score),
         sizeof(int));
      }
      //*** Display the contents of the array ***
      for (int d = 0; d < c; d++)
         cout << scoredata[d].name << endl;// "  " <<
         ↳scoredata[d].score << endl;
   }
   else
      cout << "File could not be opened\n";
}
```

Activity 24.24

Modify *TestBinaryFile* and *TestBinaryFile2* to match the code given above and check that the data can be saved and retrieved as before.

The *ostringstream* Class

Generally, our IO will involve a peripheral device or a file, but we can also read and write to a string-based stream.

An `ostringstream` object can be written to in much the same was as we might write to an output device or a file.

This new class is defined in the header file *sstream*, so this must be included at the start of any program which declares an `ostringstream` object (`#include <sstream>`).

The following two lines show how this is done:

```
ostringstream oss;
oss << "Pi to 4 places is " << setprecision(5) << 3.1416 ↳<<
endl;
```

Note that we can use all of the formatting options we used previously with `cout` (such as `setprecision()` and `endl`).

We can go on to add more output to the same `stringstream` object:

```
oss << "And e is 2.7183\n";
```

Since the class is derived from the `ostream` class, it inherits all the methods of the `ios_base`, `ios`, and `ostream` classes.

The Class Constructor

When creating an `ostringstream` object we can create an empty object with a line such as

```
ostringstream oss1;
```

or we can create it containing an initial string:

```
ostringstream("The object contains this text"s);
```

We can meaningfully use `ios::out` (the default), or `ios::app`, `ios::ate`, or `ios::trunc` as an additional parameter in the constructor.

string str() const
void str (const string& s)

The first version of this method will return the contents of the string stream as a standard `string` object. For example:

```
string output = oss.str();
```

The second version sends the contents of *s* to the stream, deleting any previous contents irrespective of the mode in which the stream has been opened.

```
oss.str("New contents\n");
```

The listing in FIG-24.30 shows a complete program which constructs the contents of an `ostringstream` object and displays those contents on the screen before changing the contents of the stream.

FIG-24.30

Using an *ostreamstring* Object

```
#include <iostream>
#include <string>
#include <iomanip>
#include <sstream>

using namespace std;


void main()
{
   stringstream oss;
   //*** Output to ostringstream ***
   oss << "Pi to 4 places is " << setprecision(5) << 3.1416
   ↳<< endl;
   oss << "And e is 2.7183\n";
   //*** Display contents of stream ***
   cout << oss.str() << endl;
   //*** Change contents and display it ***
   oss.str("New contents\n");
   cout << oss.str() << endl;
}
```

Activity 24.25

Start a new project *TestOStringStream* and implement the code given above, checking the output produced.

swap()

The `swap()` method allows us to swap the contents of two `ostreamstring` objects. For example:

```
ostringstream oss1("abc"),oss2("def");
oss1.swap(oss2);
```

The *istringstream* Class

If we want to read from a string stream then the `istringstream` class is available.

Constructor

When creating an `istringstream` object we can create an empty object with a line such as

```
istringstream iss;
```

or containing an initial string:

```
istringstream ois("The object contains this text");
```

By default, the stream is opened using the `ios::in` option.

Using Inherited Methods

Using the inherited `getline()` and `>>` operator, we can read from an `istringstream` in the same manner as we might read from the keyboard. For example, the program in FIG-24.31 reads the name and high score of a game player from an `istringstream` object.

FIG-24.31

Using an *istringstream* Object

```
#include <iostream>
#include <sstream>
using namespace std;

void main()
{
   istringstream iss("Richard Carstone*1700");
   char name[30];
   int score;

   iss.getline(name, 30, '*');
   iss >> score;
   cout << name << "   " << score << endl;
}
```

Activity 24.26

Start a new project *TestIStringStream* and implement the code given above, checking the output produced.

string str() const
void str (const string& s)

The first version of this method will return the contents of the string stream as a standard `string` object. For example:

```
string output = iss.str();
```

The second version sends the contents of *s* to the stream, deleting any previous contents irrespective of the mode in which the stream has been opened.

```
iss.str("New contents\n");
```

swap()

The `swap()` method allows us to swap the contents of two `istreamstring` objects. For example:

```
istringstream iss1("abc"),iss2("def");
iss1.swap(iss2);
```

When we use `cin`, a program will halt and wait for the necessary input. On the other hand, reading from an `istringstream` object should only be attempted when we know there are currently characters remaining within the object. This is easily achieved by either checking the end-of-file condition:

```
while (!iss.eof())
   iss >> value;
```

or more simply

```
while (iss)
   iss >> value;
```

The *stringstream* Class

The `stringstream` offers the ability to both read and write to a string stream. Therefore, if we write to a `stringstream` object with code such as

```
stringstream ss;
ss >> "This data can be read";
char txt[30];
ss << txt;                          //Sets txt to "This"
```

then, as with `cin`, a whitespace character will be taken as the termination of the input.

Constructor

When creating a `stringstream` object we can create an empty object with a line such as

```
stringstream ss;
```

or create it with an initial string:

```
stringstream ss("The object contains this text");
```

By default, the stream is opened using the `ios::in|ios::out` option. `ios::ate` can also be used to place the position indicator at the end of the stream.

string str() const
void str (const string& s)

The first version of this method will return the contents of the string stream as a standard `string` object. For example:

```
string output = ss.str();
```

The second version sends the contents of *s* to the stream, deleting any previous contents irrespective of the mode in which the stream has been opened.

```
ss.str("New contents\n");
```

swap()

The `swap()` method allows us to swap the contents of two `streamstring` objects. For example:

```
istringstream ss1("abc"),ss2("def");
ss1.swap(ss2);
```

Using String Streams

When we created the *Time* class back in Chapter 16 we also coded a friend function to output the stored time in the format *hh:mm:ss*. The code for that function was

```
//*** Displays the value of time t ***
ostream& operator<<(ostream& ct, const Time& t)
{
   char timestr[9];
   sprintf_s(timestr, "%02d:%02d:%02d", t.hour, t.minute,
   ↳t.second);
   ct << timestr;
   return ct;
}
```

We can see from the code that the format was constructed using a C-style string, *timestr* and a call to `sprintf_s()`.

We can achieve the same result using a `stringstream` object as shown in the code below:

```
//*** Displays the value of time t ***
ostream& operator<<(ostream& ct, const Time& t)
{
   stringstream ss;
   ss << setfill('0') << setw(2) << t.hour << ":" << setw(2) <<
   ↳t.minute << ':' << setw(2) << t.second;
   ct << ss.str();
   return ct;
}
```

Activity 24.27

In the *TimeClass* project you created in Activity 16.54 and 16.55, modify the *Time* class's friend function to match the code given above which makes use of a *stringstream* object.

Check that the new function operates correctly by re-running the program.

Summary

ios_base Class

- Defines IO format flags and stream positioning data types.
- type – `seekdir`

beg - measure from the start of the file.
cur - measure from the current position.
end - measure from the end of the file.

- type – `iostate` (the error state flags type)
  badbit - error in IO operation (value 4).
  failbit - logic error in IO operation (value 2).
  eofbit - end-of-file detected (value 1).
  goodbit - operation successful (value 0).

- type – `openmode`
  app - set FPI to end of stream before each op.
  ate - set FPI to end of stream after opening.
  binary - contents of stream is binary format.
  in - open stream for reading.
  out - open stream for writing.
  trunc - current contents of stream deleted.

- `long flags()` returns current flag settings
- `long flags(long f)` sets flags to *f*; returns old flag settings.
- `long setf(long f)` sets specified flags *f*; returns old flag settings.
- `long setf(long f, long g)` sets mutually exclusive flag *f* in flag group *g*; returns old flag settings.
- `long unsetf(long f)` resets specified flags *f* only; returns flags settings after reset.
- `streamsize precision()` returns the number of characters/decimal places to which a numeric value will be output.
- `streamsize precision(streamsize f)` sets the number of characters/decimal places used for stream output to *f*; returns the previous number of characters/decimal places to which a numeric value will be output.
- `streamsize width()` returns the minimum number of character positions used when outputting a value.
- `streamsize width(streamsize f)` sets the minimum number of character positions used when outputting a value to f; returns the previous minimum value.

ios Class

- `char fill()` returns the current character used to fill unused positions in an output field.
- `char fill(char f)` sets the fill character to *f*; returns the previous fill character.
- `iostate rdstate()` returns the flags set by the preceding read/write operation.
- `bool bad()` returns *true* if the *badbit* flag is set.

- `bool fail()` returns *true* if the *failbit* flag is set.
- `bool eof()` returns *true* if the *eofbit* flag is set.
- `bool good()` returns *true* if none of the other flags were set.
- `clear(iostate s)` sets the error state flags specified in *s* (default 0). Other error flags are reset.
- `setstate(iostate s)` sets the error state flags specified in *s* (default 0). Other flags are unchanged.
- `ostream* tie()` returns a pointer to the `ostream` to which the current stream is tied.
- `ostream* tie(ostream s)` sets the `ostream` object to which the current stream is tied to *s*; returns a pointer to the previous `ostream` object to which the current stream was tied.
- `streambuf* rdbuf()` returns a pointer to the buffer used by the current stream.
- `streambuf* rdbuf(streambuf* b)` sets a stream's buffer to the area referenced by *b*; returns a pointer to the previous buffer area.
- `iostate exceptions()` returns the error state flags which cause an exception to be thrown (default none).
- `iostate exceptions(iostate f)` sets the error state flags which will cause an exception to *f*; returns the previous setting.
- `!` and `bool` used to check if a stream has opened successfully.

istream Class

- This is an input stream class.
- `cin` is an object of the `istream` class.
- `>>` operator used to read a value from the input stream.
- `int get()` returns the next character from the input stream as an integer value.
- `istream& get(char& ch)` reads a character from the stream and stores it in *ch*; returns a reference to the stream.
- `istream& get(char* st, streamsize, n, char t)` reads a string from the stream and stores it in *st,* adding a null character to the *st*.
  Input will terminate after *n-1* characters or when the character *t* is encountered.
  If encountered, *t* is not stored in *st* nor extracted from the stream.

- `getline(char* st, streamsize n, char t)`
  reads a string from the stream and stores it in *st,* adding a null character to the *st*.
  Input will terminate after *n-1* characters or when the character t is encountered.
  If encountered, *t* is not stored in *st* but is extracted from the stream.

- `istream& read(char* st, streamsize n)`
  reads a string from the stream and stores it in *st*. Input will terminate after *n* characters. No null character is appended to *st*.

- `streamsize gcount()` returns the number of characters read by a preceding call to `get()`, `getline()` or `read()`.

- `streamsize readsome(char* st, streamsize n)`
  reads up to *n* characters from the stream but less if no more are available.

- `int peek()` returns the numeric code for the first character in the stream without removing it from the stream.

- `istream& unget()` places the last character read back onto the front of its stream; returns a reference to the stream.

- `istream& putback(char ch)`
  places *ch* at the front of the stream; returns a reference to the stream.

- `istream& ignore(streamsize n, int t)`
  ignores the next *n* characters in the stream. Less than *n* characters are ignored if *t* is encountered.

- `istream& seekg(streampos p)`
  The stream position indicator is moved to position *p* in the stream (measured from first position in stream).

- `istream& seekg(streampos p, seekdir s)`
  Moves stream position indicator *p* positions relative to *s*.

- `streampos tellg()` returns the stream position indicator's current value.

ostream Class

- This is an output stream class.
- Every C++ program automatically creates three objects of this class: `cout`, `cerr`, and `clog`.
- `<<` operator used to write a value from the output stream.
- `ostream& put(char ch)` writes the character *ch* to the output stream.
- `ostream& write(char* st, streamsize n)`
  writes *n* characters from the string *st* to the stream.
- `ostream& flush()` empties the streams sending its contents to the

devices linked by the stream.

- `ostream& seekp(streampos p)`
  The stream position indicator is moved to position *p* in the stream (measured from first position in stream).

- `ostream& seekp(streampos p, seekdir s)`
  Moves stream position indicator *p* positions relative to *s*.

- `streampos tellp()` returns the stream position indicator's current value.

ofstream Class

- Used to link a stream to a file for output.
- The file name and open mode can be given in the constructor.

- `open (const char* f, openmode m)`
  `open (const string f, openmode m)`
  links the file named in *f* to the `ofstream` object in mode *m*.

- `bool is_open()` returns *true* if the stream has been successfully linked to a file.

- `close()` closes the file linked to the `ofstream` object and writes any data still in the buffer to the file.

ifstream Class

- Used to link a stream to a file for input.
- The file name and open mode can be given in the constructor.

- `open (const char* f, openmode m)`
  `open (const string f, openmode m)`
  links the file named in *f* to the `ifstream` object in mode *m*.

- `bool is_open()` returns *true* if the stream has been successfully linked to a file.

- `close()` closes the file linked to the `ifstream` object and writes any data still in the buffer to the file.

fstream Class

- Objects of the `fstream` class allow a file to be opened for both reading and writing.

ostringstream Class

- Use an `ostringstream` object to output to a string stream.

istringstream Class

- Use an `istringstream` object to input from a string stream.

stringstream Class

- Use an `stringstream` object to allow both reading and writing in a string stream.

Binary Files

- In a binary file, values are written in the same format as they are held in memory.
- When writing structured data to a file, using binary format means every record contains exactly the same number of bytes.
- Binary files allow easy updating of records within a file.

Solutions

Activity 24.1

Code for *TestIOS*:

```
#include <iostream>
using namespace std;

int main()
{
   cout << cout.flags() << endl;
   cout << hex << cout.flags() << endl;
}
```

The output should be

```
513
201
```

Activity 24.2

Modified code for *TestIOS*:

```
#include<iostream>
#include <iomanip>
using namespace std;

void main()
{
   //*** Set to hex, left-justified output ***
   cout.flags(ios::hex | ios::left);
   //*** Output values ***
   cout << setw(3) << 255 << endl;
   cout << setw(3) << 15 << endl;
}
```

When using the call to `flags()`, the output is

```
ff
f
```

When the call to `flags()` is commented out, the display is

```
255
 15
```

Activity 24.3

Modified code for *TestIOS*:

```
#include<iostream>
#include <iomanip>
using namespace std;

void main()
{
   //*** Set to hex, left-justified output ***
   cout.flags(ios::hex | ios::left);
   //*** Display Booleans in text format ***
   cout.setf(ios::boolalpha);
   //*** Output values ***
   cout << setw(3) << 255 << endl;
   cout << setw(3) << 15 << endl;
   cout << true << endl;
   cout << false << endl;
}
```

With the call to `setf()` active the output is

```
ff
f
true
false
```

Note the hex and left-justified flags are still set.

With the call to `setf()` commented out the output is

```
ff
f
1
0
```

Activity 24.4

Output from test a):

```
Performing output
State after first output 0
Enter number 12
State after first input 0
```

This tells us that the *goodbit* is set and therefore all IO performed without problems.

Output from test b):

```
Performing output
State after first output 0
Enter number X
State after first input 2
```

This time, because a numeric value could not be read from the input stream (`cin`) the *failbit* (value 2) is set.

Output from test c):

```
Performing output
State after first output 0
Enter number ^Z
State after first input 3
```

On a Windows machine, Ctrl+Z represents the end-of-file marker when entered in the keyboard buffer and so, because a numeric value could not be read and the end-of-file was detected both the *failbit* and *eofbit* are set (2 + 1).

Activity 24.5

Modified code for *TestIOState*:

```
#include<iostream>
using namespace std;

int main()
{
   //*** Display iostate flags after output ***
   cout << "Performing output" << endl;
   cout << "goodbit : " << cin.good() << endl;
   cout << "eofbit : " << cin.eof() << endl;
   cout << "failbit : " << (cin.fail()&&(!cin.bad()))
   ↳<< endl;
   cout << "badbit : " << cin.bad() << endl;
   //*** Display iostate flag after input ***
   int num;
   cout << "Enter number ";
   cin >> num;
   cout << "goodbit : " << cin.good() << endl;
   cout << "eofbit : " << cin.eof() << endl;
   cout << "failbit : " << (cin.fail() && (
   ↳!cin.bad())) << endl;
   cout << "badbit : " << cin.bad() << endl;
}
```

Note the check for the `failbit` being set requires us to check for `fail()` AND NOT `bad()` since `fail()` on its own would return *true* if either the `failbit` or `badbit` are set.

Activity 24.6

The output from *TestIOExceptions* is:

```
0
6
Enter number X
ios_base::failbit set: iostream stream error
iostream:1
Input failed
```

The first value, 0, tells us that at the start of the program no flags will cause an exception to be thrown should a probllem arise when using `cin`.

The second value, 6, tells us that both `badbit` (value 4) and `failbit` (value 2), if set during an operation involving `cin`, will cause an exception to be thrown.

ios_base::failbit set: iostream stream error
is the message returned by `cin.what()`.

iostream: 1
is the message returned by `cin.code()`.

Activity 24.7

When the program is first run using the input data *abcd*efghij* the output produced is:

```
Value read (first character): a
State Flag Settings (after one char read):
badbit = 0
failbit = 0
eofbit = 0
goodbit = 1

Value read (first string): bcd
State Flag Settings:
badbit = 0
failbit = 0
eofbit = 0
goodbit = 1

Value read (second string):
State Flag Settings:
badbit = 0
failbit = 1
eofbit = 0
goodbit = 0
```

Note that the second attempt to read a string (more than one character) fails because the first character encountered in the stream is * (the terminating character) so no characters are transferred to *name* and this results in `failbit` being set.

When we remove the comment characters from the `get()` and `cout` lines then the output produced is:

```
Value read (first character): a
State Flag Settings (after one char read):
badbit = 0
failbit = 0
eofbit = 0
goodbit = 1

Value read (first string): bcd
State Flag Settings:
badbit = 0
failbit = 0
eofbit = 0
goodbit = 1

Value read (second character): *

Value read (second string): efgh
State Flag Settings:
badbit = 0
failbit = 0
eofbit = 0
goodbit = 1
```

The additional `get()` statement extracts the * from the stream and allows the second string read to succeed.

Activity 24.8

Modified code for *TestIstream01*:

```
#include<iostream>
using namespace std;

void main()
{
   char ch;
   char name[10];
   //*** Place some characters in the keyboard stream
   ↳***
   //*** and read the first character
   ↳***
   ch = cin.get();
   //*** Display value read ***
   cout << "\nValue read (first character): " << ch <<
   ↳endl;
   //*** Display state flag values ***
   cout << "State Flag Settings (after one char
   ↳read):\n";
   cout << "badbit = " << cin.bad() << endl;
   cout << "failbit = " << cin.fail() << endl;
   cout << "eofbit = " << cin.eof() << endl;
   cout << "goodbit = " << cin.good() << endl;
   cin.getline(name,6,'*');
   //*** Display value read ***
   cout << "\nValue read (first string): " << name <<
   ↳endl;
   //*** Display state flag values ***
   cout << "State Flag Settings:\n";
   cout << "badbit = " << cin.bad() << endl;
   cout << "failbit = " << cin.fail() << endl;
   cout << "eofbit = " << cin.eof() << endl;
   cout << "goodbit = " << cin.good() << endl;
   //*** Read a single character from stream ***
   ch = cin.get();
   //*** Display value read ***
   cout << "\nValue read (second character): " << ch
   ↳<< endl;
   //*** Read another set of characters ***
   cin.getline(name, 5, '*');
   //*** Display value read ***
   cout << "\nValue read (second string): " << name
   ↳<< endl;
   //*** Display state flag values ***
   cout << "State Flag Settings:\n";
   cout << "badbit = " << cin.bad() << endl;
   cout << "failbit = " << cin.fail() << endl;
   cout << "eofbit = " << cin.eof() << endl;
   cout << "goodbit = " << cin.good() << endl;
   ch = cin.get();
   cout << "\nValue read (third character): " << ch
   ↳<< endl;
}
```

The output from the program is:

```
Value read (first character): a
State Flag Settings (after one char read):
badbit = 0
failbit = 0
eofbit = 0
goodbit = 1

Value read (first string): bcd
State Flag Settings:
badbit = 0
failbit = 0
eofbit = 0
goodbit = 1

Value read (second character): e

Value read (second string): fghi
State Flag Settings:
badbit = 0
failbit = 1
eofbit = 0
goodbit = 0

Value read (third character):
```

Notice that because no terminating character (*) was detected when reading the second value for name, the *failbit* was set. With this bit set, any subsequent attempts at reading from the stream will fail.

Activity 24.9

The output from a) is

```
Data read was abcdefgh
8 characters were read from the stream
Next character read was i
```

The output from b) is

```
Data read was abcd
efg
8 characters were read from the stream
Next character read was h
```

Note that the return character between d and e has been included as part of the string created.

Activity 24.10

a)
The `get()` statement extracts the '*a*' from the input stream and `readsome()` extracts what remains in the stream: "*bc**n*".

b)
The `get()` statement extracts the '*a*' from the input stream and `readsome()` extracts the next 8 characters"*bcdefghi*".

Activity 24.11

The program creates the following output when using the test data *abcd*:

```
Character read from stream : a
First character in input stream is b
First character in input stream is a
First character in input stream is X
Characters read from stream : Xabcd
```

Activity 24.12

The output created from the original version of the program using the data *a1234b* is:

```
Characters read were a and b
```

For the second version we need to change the line

```
cin.ignore(4);
```

to

```
cin.ignore(4,'*');
```

When the modified program is run with the test data a12*4b, the output is:

```
Characters read were a and 4
```

Activity 24.13

The file should contain the following data after the *TestTextFile* is first run:

```
12  3.14Xabcdef
```

Modified code for *TestTextFile*:

```
#include<iostream>
#include<fstream>
using namespace std;

void main()
{
  ofstream myfile;
  //*** Attempt to open file ***
  myfile.open("datafile.txt", ios::out | ios::app);
  //*** If successful, write to file ***
  if (myfile)
  {
    //myfile << 12 << "  " << 3.14;
    //myfile.put('X');
    myfile.write("ghijkl", 6);
    cout << "Data written to file\n";
  }
  //*** Close the file ***
  myfile.close();
  cout << "File closed\n";
}
```

The file should contain the following data after the second run of *TestTextFile*:

```
12  3.14Xabcdefghijkl
```

Activity 24.14

The output from the program is:

```
File opened successfully for reading
12  3.14Xabcdefghijkl
```

Activity 24.15

Modified code for *TestTextFile2*:

```
#include <iostream>
#include<fstream>
using namespace std;

int main()
{
  //*** Read data from file ***
  char ch;
  ifstream myfile2;
  //*** Attempt to open file ***
  myfile2.open("..\\TestTextFile\\datafile.txt");
  //*** If successful, display contents ***
  if (myfile2)
  {
    cout << "File opened successfully for
    ↳reading\n";
    //*** Read and display each alternate character
    ↳in the file ***
    myfile2.get(ch);
    while (!myfile2.eof())
    {
      cout << ch;
      //*** Skip a character ***
      myfile2.seekg(1, ios::cur);
      //*** Read character ***
      myfile2.get(ch);
    }
    //*** Move cursor to new line ***
    cout << endl;
    //*** Close the file ***
    myfile2.close();
  }
  else
    cout << "File not found\n";
}
```

The program output is:

```
File opened successfully for reading
1 .4acegik
```

Activity 24.16

Modified code for *TestTextFile2*:

```
#include <iostream>
```

```
#include<fstream>
using namespace std;

int main()
{
  //*** Read data from file ***
  char ch;
  ifstream myfile2;
  //*** Attempt to open file ***
  myfile2.open("..\\TestTextFile\\datafile.txt");
  //*** If successful, display contents ***
  if (myfile2)
  {
    cout << "File opened successfully for
    ↳reading\n";
    //*** Read and display each alternate character
    ↳in the file ***
    myfile2.get(ch);
    while (!myfile2.eof())
    {
      cout << ch;
      //*** Skip a character ***
      myfile2.seekg(1, ios::cur);
      //*** Read character ***
      myfile2.get(ch);
    }
    //*** Move cursor to new line ***
    cout << endl;
    //*** Clear the eofbit ***
    myfile2.clear();
    //*** Display number of bytes in file ***
    myfile2.seekg(0, ios::end);
    cout << "The file contains " << myfile2.tellg()
    ↳<< " bytes\n";
    //*** Close the file ***
    myfile2.close();
  }
  else
    cout << "File not found\n";
}
```

The output should show there are 20 bytes in the file.

Activity 24.17

Modified code for *TestTextFile*:

```
#include<iostream>
#include<fstream>
using namespace std;

void main()
{
  ofstream myfile;
  //*** Attempt to open file ***
  myfile.open("datafile.txt");
  //*** If successful, write to file ***
  if (myfile)
  {
    myfile << 12 << '\n' << "  \n" << 3.14 << '\n';
    myfile.put('X');
    myfile.put('\n');
    myfile.write("abcdef\n", 7);
    myfile.write("ghijkl\n", 7);
    cout << "Data written to file\n";
  }
  //*** Close the file ***
  myfile.close();
  cout << "File closed\n";
}
```

Modified code for *TestTextFile2*:

```
#include <iostream>
#include<fstream>
using namespace std;

int main()
{
  //*** Read data from file ***
  char line[100];
  ifstream myfile2;
  //*** Attempt to open file ***
  myfile2.open("..\\TestTextFile\\datafile.txt");
  //*** If successful, display contents ***
  if (myfile2)
  {
    cout << "File opened successfully for
    ↳reading\n";
    //*** Read a line from the file ***
    myfile2.getline(line, 100);
    //*** WHILE not eof DO ***
    while (!myfile2.eof())
    {
      //*** Display the value ***
      cout << "Data read : " << line << endl;
      //*** Read a line from the file ***
      myfile2.getline(line, 100);
    }
    //*** Clear the eofbit ***
    myfile2.clear();
    //*** Display number of bytes in file ***
    myfile2.seekg(0, ios::end);
    cout << "The file contains " << myfile2.tellg()
    ↳<< " bytes\n";
    //*** Close the file ***
    myfile2.close();
  }
  else
    cout << "File not found\n";
}
```

The file contains 20 bytes.

Activity 24.18

When displayed in Notepad the contents of *highscores.dat* is

Esther Summerson ÌÌÑ John Jarndyce Ìlq
Ada Clare Ìli

The numeric values show up as nonsense characters. The high scores are in 2's complement format and therefore do not show as meaningful characters.

Activity 24.19

Modified code for *TestBinaryFile*:

```
#include <iostream>
#include <fstream>
using namespace std;

struct HighScoreType
{
  char name[30];
  int score;
};

int main()
{
  HighScoreType scoredata[] = { {"Esther Summerson",
  2001},{"John Jarndyce",1905},{"Ada Clare",1897} };
  ofstream hscorefile;
  hscorefile.open("highscores.dat", ios::out |
  ios::binary);
  if (hscorefile)
  {
    hscorefile.write(reinterpret_cast<const
    char*>(scoredata), 3*sizeof(HighScoreType));
    cout << "Data written to file\n";
  }
  else
    cout << "File could not be opened\n";
}
```

Activity 24.20

The names and scores of the three high score records are displayed correctly:

Esther Summerson 2001
John Jarndyce 1905
Ada Clare 1897

Activity 24.21

Modified code for *TestBinaryFile2*:

```
#include <iostream>
#include <fstream>
using namespace std;
```

```
struct HighScoreType
{
  char name[30];
  int score;
};

int main()
{
  HighScoreType scoredata[3];
  ifstream hscorefile;

  //*** Open binary file for reading ***
  hscorefile.open("..\\TestBinaryFile\\
  ↳highscores.dat", ios::in | ios::binary);
  if (hscorefile)
  {
    //*** Read records into array ***
    hscorefile.read(reinterpret_cast<char*>(
    ↳scoredata), 3*sizeof(HighScoreType));
    //*** Display the contents of the array ***
    for (int c = 0; c < 3; c++)
      cout << scoredata[c].name << "  " <<
      ↳scoredata[c].score << endl;
  }
  else
    cout << "File could not be opened\n";
}
```

The display is exactly the same as that produced in the previous Activity.

Activity 24.22

Modified code for *TestBinaryFile2*:

```
#include <iostream>
#include <fstream>
using namespace std;

struct HighScoreType
{
  char name[30];
  int score;
};

int main()
{
  HighScoreType scoredata[3];
  ifstream hscorefile;

  //*** Open binary file for reading ***
  hscorefile.open("..\\TestBinaryFile\\highscores.
  dat", ios::in | ios::binary);
  if (hscorefile)
  {
    //*** Read last record into array ***
    hscorefile.seekg(-static_cast<int>(sizeof(
    HighScoreType)), ios::end);
    hscorefile.read(reinterpret_cast<char*>(
    scoredata), sizeof(HighScoreType));
    //*** Display the record read ***
    cout << scoredata[0].name << "  " <<
    scoredata[0].score << endl;
  }
  else
    cout << "File could not be opened\n";
}
```

Record displayed is:

```
Ada Clare  1897
```

Activity 24.23

The file updates as expected, with the contents of the file now show as:

```
Esther Summerson  2101
John Jarndyce  2005
Ada Clare  1997
```

Activity 24.24

The data appears exactly as before.

Activity 24.25

The output produced is:

```
Pi to 4 places is 3.1416
And e is 2.7183

New contents
```

Activity 24.26

The output produced is:

```
Richard Carstone   1700
```

Activity 24.27

The output remains identical.

25

Program Efficiency

In this Chapter:

- ❑ *decltype* and *declval*
- ❑ *variant* type
- ❑ *any* type
- ❑ *optional* type
- ❑ Move Semantics

Data Flexibility

Introduction

For most variables and objects defining them fixes exactly what type of value they can hold. So an `int` variable must hold an int value and a *Date* object must store a date value.

With variables created as flexible storage types (in fact, they are objects created from classes), we have the option to store any type of value we wish. We've already looked at two such types way back in Chapter 4: `pair` and `tuple`. Those two data types are actually class templates.

All of the following types are defined within the `std` namespace.

decltype and *declval*

decltype

The `decltype()` command (short for **declared type**) returns the type of a variable, object, or of the value produced by the result of an expression. The returned value is not a string, so it cannot be displayed. However, it can be used when declaring another variable or object. For example, if our code contains an `int` variable

```
int c = 12;
```

then we can use that variable as a parameter to `decltype()` and then have the value returned become the data type for a new variable:

```
decltype(c) d = 21;
```

Since *c* is an `int` value, the above statement is equivalent to

```
int d = 21;
```

We can use `decltype()` on a `struct` variable:

```
struct Coords
{
   double x;
   double y;
};
Coords p{ 3.5,2.9 };
decltype(p) p1;          //p1 is of type Coords
p1.x = 0.2;
p1.y = 3.6;
```

or on a field within a `struct` variable:

```
decltype(p.x) f;         //f is of type double
f = 66.1;
```

It can also be used on an object:

```
Date dt;
decltype(dt) dt2;        //dt2 is of class Date
```

An attribute within an object can also be used, but only if that attribute is public. Which means that a statement such as

```
decltype(dt.month) m;
```

would fail since *month* is a private attribute of the *Date* class.

The type can also be a pointer type as in the lines

```
int *ptr {nullptr};
decltype(ptr) ptr2 = new int(31); //ptr2 is a pointer to an int
```

Or a reference as in:

```
char ch = 'B';
char& rch = ch;
decltype(rch) rch2{ ch };
```

In the above *rch*, and *rch2* both reference *ch*.

Another situation in which `decltype()` may be used is with an expression. For example,

```
decltype(c/12.6) quot{3.12};
```

would make *quot* a `double` variable which has an initial value of 3.12.

The expression given as an argument to `decltype()` is not actually calculated, so, for example

```
decltype (c++/12.6) quot{3.12};
```

would not increment the value of variable *c*.

When used with a function argument, `decltype()` returns the type returned by the function itself. Hence, with a function such as

```
double Smaller(double a, double b)
{
   if (a < b)
      return a;
   else
      return b;
}
```

we could declare a variable of `Smaller()`'s return type (`double`) using a line such as

```
decltype(Smaller(3.1, 9.4)) q{ 3.33 };
```

Remembering that operators such as `<<` and `>>` are also really functions, we might use the expression

```
decltype(cout << "Hi\n")
```

to specify an `&ostream` type.

As with an expression, functions specified within the parentheses are not executed so in the above examples *Smaller()* is not executed and *Hi* is not displayed.

We may see `decltype()` in a `using` statement which assigns an alternative name to a type. As a trivial example:

```
using BigType = decltype(Smaller(3.1, 9.4));
BigType r;                                  //r is of type double
```

BigType becomes an alternative name for the type of value returned by *Smaller()*.

We can use the term `decltype(auto)` when specifying a type that must be determined from the value being used to initialise a variable. For example,

```
int a = 12;
decltype(auto) b = a;      //b is of type int
```

At first glance it may seem that `decltype()` does the same job as `auto`, but, in fact, there is a suitable difference. The term `auto` never creates a reference type nor a `const` type.

For example, let's assume we have the following function prototypes:

```
long f1();
long& f2();
const long f3();
```

If we create variables based on the return types using `auto`:

```
auto a = f1();
auto b = f2();
auto c = f3();
```

a, *b* and *c* will all be of type `long` since `auto` will always eliminate reference and `const` settings.

On the other hand, using `decltype()`:

```
decltype(f1()) d = f1();
decltype(f2()) e = f2();
decltype(f3()) f = f3();
```

d, *e* and *f* will match the function return types exactly, making them `long`, `long&` and `const long` respectively.

We are most likely to make use of `decltype()` when writing function templates and class templates. For example,

```
template <class T1, class T2>
auto Subtract(T1 v1, T2 v2) -> decltype(v1-v2)
{
   return v1 - v2;
}
```

Notice that the function uses the alternative way of defining the return type. Replacing `auto` the expression `decltype(v1-v2)` would create a compilation error since *v1* and *v2* are not defined until later in the same line.

If we were to call *Subtract()* with the line

```
cout << Subtract(Date(1,4,2020), Date(17,8,1950);
```

the expression `decltype(v1-v2)` would return type `int` as defined by the method `operator-()` in the *Date* class which would be used to subtract two dates.

Activity 25.1

Start a new project called *TestDeclType* and create a program that uses the function *Subtract()* (as defined above) to first subtract two *Date* objects and then two *Time* objects.

Remember to include *"ChronoLib.h"* and adjust the properties of the project to find the necessary header and library files.

C/C++>General>Additional Include Directories>*...ChronoLib*
Linker>General>Additional Library Directories>*...MyLibraries/Debug*
Linker>Input>Additional Dependencies>*ChronoLib.lib*

declval

The term `declval` is almost always used in conjunction with `decltype`. To understand the purpose of `declval()`, we'll revisit the example above where we used `decltype()` to get the type returned by the function *Smaller()*. We had to supply two specific `double` value parameters for the function call. However, we really just wanted the return type when *Smaller()* is supplied with ANY two `double` values.

This is where `declval()` can be used since it returns an *rvalue* which can be used in such situations. `declval()` is a function template and needs to have the type of value it is to return specified as part of its call. For example, we could rewrite our definition of *BigType* using the line

```
using BigType = decltype(Smaller(declval<double>()),
↳declval<double>()));
```

In effect, this may be interpreted as:

> Bigvalue *is an alternative name for the type returned by* Smaller() *when it is supplied with two* double *values.*

`decltype()` and `declval()` are at their most powerful when involved with function template coding and although generally beyond the complexity of this text we can get a glimpse of that power in a statement such as

```
template<typename T1, typename T2>
using DifferenceType = decltype(declval<T1>() - declval<T2>());
```

which specifies that *DifferenceType* is the type which results from subtracting a value of type *T1* from a value of type *T2*.

variant

When we create an object of type `variant`, (which requires a `#inlude<variant>`) we are, in effect, creating a `union` structure where the stored value can be interpreted as any type we have previously specified. For example, let's say we want a variable (*var*) which can store an `int`, `char`, `double` or `float`. To declare such a variable we would write

```
variant<int, char, double, float> var;
```

specifying the type of values we expect the variable to be assigned at various part of the code that follows the declaration.

The options listed for a `variant` object are given an index value starting at zero. In our example that index would be

```
int    = 0
char   = 1
double = 2
float  = 3
```

Assigning a value is straightforward using lines such as:

```
var = 12;
var = 'X';
```



This code assumes our program has access to the *Time*, *Date* and *Clock* classes.

We are not limited to our `variant` variable only containing primitive types. It might also be used for classes or `struct` types. For example, the following code would make *var2* a *Time* object:

```
variant<dsk::Time, dsk::Date, Clock> var2;
```

```
var2 = dsk::Time(1, 2, 3);
```

Later in the same code block we could write

```
var2 = dsk::Date(1,1,2050);
```

and the program will automatically call an object's destructor when we change the contents from one time to another. For example, in the code above the destructor for the *Time* class would be called when we changed the contents of var2 from a *Time* object to a *Date* object.

Methods

index()

The `index()` method returns the index value for the value currently stored in the variant object. For example, if we have executed the statement

```
var = 2.3F;
```

then `var.index()` would return 3 (since *var* contains a `float` value – the fourth type option listed).

emplace()

As we saw in the container classes, the `emplace()` method creates a value/object directly within our variable rather than having to first construct a temporary copy of that value before transferring it to the required location. So to create a *Clock* object in *var2* we could write

```
var2.emplace<Clock>(12, 0, 9);
```

The parameters to `emplace()` represent those required by the constructor of the *Clock* class.

swap()

Where two `variant` objects have the same sequence of possible types, then the values in current object may be swapped with the value of the other. For example, the following code swaps the values in *v1* and *v2* before displaying their contents.

```
variant <int, char, bool> v1{ 3 }, v2{ '$' };
v1.swap(v2);
```

The program in FIG-25.1 creates two variant objects, displays their index values, swaps their contents and displays the index values for a second time.

FIG-25.1

Using Variants

```
#include <iostream>
#include "ChronoLib.h"
#include <variant>
using namespace std;

int main()
{
   variant <dsk::Date, dsk::Time, int, char, string> var1, var2;
   var1.emplace<dsk::Date>(23, 11, 1963);
   var2 = "Time and Relative Dimension In Space"s;
```



```
    cout << "var1 type index : " << var1.index() << endl;
    cout << "var2 type index : " << var2.index() << endl;
    var1.swap(var2);
    cout << "var1 type index : " << var1.index() << endl;
    cout << "var2 type index : " << var2.index() << endl;
}
```

Activity 25.2

Start a new project called *TestVariant* and implement the code given above.

What output is produced?

Related Functions

Relational Operators

All of the relational operators (`==`, `!=`, `<`, etc.) can be used to compare two `variant` variables as long as those two variables are defined for identical types.

get()

When we want to access the value held in our variable, we must use the `get()` function which also requires the type of value currently stored in the variable. For example,

```
cout << get<float>(var1) << endl;
```

Alternatively, we can use the stored value's type index number:

```
cout << get<3>(var1) << endl;
```

We could output the contents of *var2* (a *Time* object) using the line

```
cout << get<dsk::Time>(var2) << endl;
```

If the type or index value given in the angled brackets is invalid (e.g. `get<5>(var)`), a `bad_variant_acess` exception is thrown.

Note that the value in the angled brackets cannot be a variable, expression or function call. For example, it is invalid to write

```
cout << get<var1.index()>(var1) << endl; //INVALID
```

Activity 25.3

Modify *TestVariant* by adding statements to output the values of *var1* and *var2* after their contents have been swapped.

get_if()

The `get_if()` function returns a pointer to the value stored in a `variant` variable. The type or index of the assumed type held must be given along with the address of the `variant` variable. If this given type/index does not match the type/index of the actual value being held, a null pointer is returned. For example, assuming the declaration

```
variant <string, int, bool> sib;
```

we might attempt to acquire a pointer to the value in variable *sib* with the line

```
auto ptr = get_if<string>(&sib);
```

ptr would then contain the address of the data in *sib* only if that data was a `string` class object. If the contents of *sib* were an `int` or `bool` value, *ptr* would be set to *null*.

Rather than specify the type we are expecting the variable to be holding, we can use the index of that type. Hence, the line

```
auto ptr = get_if<0>(&sib);
```

is equivalent to the earlier code (since `string` is the first type listed as possible types for our variable).

We could display the contents of *sib* via the *ptr* pointer, assuming it held a `string` object, using the code

```
if (ptr != nullptr)
   cout << get<string>(sib) << endl;
else
   cout << "sib does not contain a string object\n";
```

holds_alternative()

The `holds_alternative()` function returns *true* if the variant parameter currently holds the type specified. If the value held is of a different type, *false* is returned.

For example, might use this function as an alternative way of checking that our `variant` object holds a `string` object with the following code:

```
if (holds_alternative<string>(sib))
   cout << get<string>(sib) << endl;
else
   cout << "sib does not contain a string\n";
```

Activity 25.4

Append to *TestVariant* new code which swaps the contents of *var1* and *var2* 50% of the time.

If *var1* now contains a *Date* object display its contents, otherwise display the message *"var1 does not hold a Date"*.

visit()

The `visit()` function is used to perform some operation on a `variant` object. This means that the `visit()` function requires two parameters. The first is the function or lambda (see next chapter) containing the code to be executed and the second is the `variant` object which is the target of that code.

Creating a function for `visit()` requires us to create a class which overloads the parentheses operator, `()`, for each possible type for the variant object. For example, let's say we have defined three variant objects with the line

```
variant <int, double, float> var1{2}, var2{12.3}, var3{-8.0F};
```

and that we want to triple the value held in each of these variables using `visit()` and a suitable function.

To start with we need to define a class which overloads `()` for each possible type:

```
class Triple
{
   public:
      void operator()(int& v)
      {
         v *= 3;
      }
      void operator()(float& v)
      {
         v *= 3;
      }
      void operator()(double& v)
      {
         v *= 3;
      }
};
```

and then, in `main()`, we can subject our variant objects to the appropriate operation using `visit()`:

```
visit(Triple(), var1);
visit(Triple(), var2);
visit(Triple(), var3);
```

Activity 25.5

In *TestVariant,* add a new class called *Increment* that overloads the call operator to add 1 second to a *Time* class object, 1 day to a *Date* object, increment the ASCII code for a `char` variable (keeping it in the range 32 to 126) and performs no operation on a `string` object.

Use `visit()` to apply *Increment()* to *var1* and *var2* then display the contents of the two variant values. This code should come at the end of the existing code in `main()`.

swap()

Swaps the contents of two `variant` objects. For example,

```
swap(var1,var2);
```

Note that this performs exactly the same job as the `swap()` method defined within the variant class.

any

Like `variant`, the `any` class allows us to assign any type of value to a variable (actually an object) of this class.

We can declare an object to be of type `any` with a line such as

```
any v;
```

The class also has a constructor which allow an initial value to be set:

```
any v(21), v2(3.14), v3("Elizabeth");
```

We can also assign a value of any type to an existing `any` object:

```
v = "Hello";;
v2 = 12;
v3 = -4.2;
```

But in order to use `any`, we must add the line

```
#include <any>
```

at the start of our code.

As we might expect from most classes, our `any` object comes with built-in methods and friend functions.

Methods

type()

This method returns a `type_info` object (just like a `typeid()` call). The returned object has a `name()` method which returns a string giving the type of the value held in our `any` object. For example, the lines

```
v = "Hello";
cout << v.type().name() << endl;
```

would display

```
char const *
```

the type of value currently held in *v*.

has_value()

This method returns the value *true* if the `any` object currently contains a value. For example, the lines

```
cout << boolalpha;
any v;
cout << v.has_value() << endl;
v = 3.9;
cout << v.has_value() << endl;
```

will create the following output :

```
false
true
```

emplace()

The function is an efficient way of assigning an object to our `any` variable. For example, to store a *Date* object in *v* we could use the code

```
v.emplace<dsk::Date>(31,12,2020);
```

Of course, the above line assumes that the project in which the line appears has access to the *Date* class which we created in a previous chapter.

Generally, we need to specify the required type in the angled brackets before the parameter list:

```
v2.emplace<dsk::Time>(2,0,0);
```

reset()

The `reset()` method deletes the current contents of the `any` object. For example, the code

```
any v = 3.9;
cout << v.has_value() << endl;
v.reset();
cout << v.has_value() << endl;
```

would display

```
true
false
```

swap()

The `swap()` method swaps the contents of two `any` objects. For example,

```
any v1 = 3.9;
any v2 = "Hello";
v1.swap(v2);
cout << "v1 = " << any_cast<char const*>(v1) << endl;
cout << "v2 = " << any_cast<double>(v2) << endl;
```

will output the following:

```
v1 = Hello
v2 = 3.9
```

Functions

any_cast()

If we are to make use of the value in an `any` object, we must cast it to the appropriate type using `any_cast()`. Some examples are shown below:

```
v = "Hello";
cout << any_cast<char const*>(v) << endl;

v = 12;
int sum = any_cast<int>(v) + 3;
```

This function will throw a `bad_any_cast` exception if the cast type does not match the actual type held in the `any` object.

make_any()

The `make_any` function can be used to create an `any` object containing a specified value. For example, we could store an integer value in `any` object *v* with the line

```
any v = make_any<int>(42);
```

Of course, this seems like a long-winded way of doing things when we could write the much shorter

```
any v = 42;
```

However, it is much more likely we would use this function to create an object which is to be stored in our `any` variable:

```
any v = make_any<dsk::Date>(12,12,2021);
```

swap()

Swaps the contents of two `any` objects.

optional

The `optional` class template, defined within the `std` namespace and requiring a `#include<optional>` command line, allows us to create objects that contain a single value of any type or they may contain no value at all. For example, when we declare an `int` variable as in the line

```
int no1;
```

even if we don't specify a value it will, nonetheless, contain some apparently random value taken from the pattern of 1s and 0s that happen to be already in the space allocated to the variable.

If we were to create an object of the `optional` class in which we wish to store an `int` value of 12, our code would be

```
optional<int> v = 12;
```

However, although the syntax is a little different, we could achieve the same thing by creating a `variant` or `any` object. What makes `optional` different is that objects of this type can be assigned no value at all. So if we write

```
optional<int>v;
```

or

```
optional<int>v{};
```

v won't contain zero or some undefined value. It will contain the value `nullopt`.

This would allow us to write code such as

```
if (v == std::nullopt)
   cout << "v contains no value\n";
```

However, the main use of `optional` variables is as the return type from a function which might not return any meaningful value. For example, let's assume we have written a function which returns the lowest value within an array and have used the code



This code assumes the program in which it occurs contains the line `using namespace std;`

```
int Lowest(const int arr[], const int sz)
{
   int low = arr[0];
   for (int c = 1; c < sz; c++)
      if (arr[c] < low)
         low = arr[c];
   return low;
}
```

The code isn't foolproof since it would fail if we pass it an array with no elements. We could try to allow for this situation by starting the body of the function with the lines

```
if (sz <= 0)
   return;
```

But the compiler would object to this since the function is supposed to return an `int` value. This leaves us with a problem. Any value we come up with as the return value might actually exist in the array when that array does contain one or more values.

So to solve our dilemma we can rewrite *Lowest()* as

```
optional<int> Lowest(const int arr[], const int sz)
{
   if (sz <= 0)
      return nullopt;              //or     return {};
   int low = arr[0];
   for (int c = 1; c < sz; c++)
      if (arr[c] < low)
         low = arr[c];
   return low;
}
```

As we've seen with the `variant` and `any` classes, the `optional` class contains a set of methods and related functions. These are described below.

Methods

value()

Returns the value currently stored in the `optional` object. For example, if we want to display the contents of an optional variable,*v*, then writing

```
cout << v << endl;
```

will fail. Instead, we must write

```
cout << v.value() << endl;
```

* and –>

As an alternative to using `value()`, we can simply dereference our `optional` object:

```
cout << *v << endl;
```

If our optional object is storing a more complex structure such as a *Date* object, we can use the `->` operator to access its methods:

```
cout << v->GetDay() << endl;
```

has_value()

This method returns *true* if the `optional` object contains a value (other than `nullopt`). If it contains `nullopt`, then *false* is returned. For example,

```
if (v.has_value())
   cout << v.value() << endl;
```

value_or()

The `value_or()` method returns the value held in an `optional` object when there is an actual value and returns a specified value when the `optional` object has no value. For example,

```
cout << v.value_or(-99) << endl;
```

This will display the value in *v* but, if there is no value in *v* (*v* contains `nullopt`), the value -99 will be displayed.

The alternative value supplied must be of the same type as specified in the `optional` object's declaration (or convertible to that type).

emplace()

The function is an efficient way of assigning an object to our `optional` variable. For example, to store a *Date* object in *v* we could use the code

```
optional <Date> v;
v.emplace(31,12,2020);
```

reset()

The `reset()` method deletes the current contents of the `optional` object. For example:

```
v.reset();
```

swap()

The `swap()` method swaps the contents of two `optional` objects. For example:

```
optional <int> v1 = 12;
optional <int> v2 = 65;
v1.swap(v2);
```

Note that the two objects must contain the same type of values.

Related Functions and Operators

Relational Operators

All of the relational operators (`==`, `!=`, `<`, `<=`, etc.) are overloaded for use with optional variables. Hence we could write

```
if(v1 == v2)
   ...
```

However, the variables must store the same time of values.

swap()

In addition to the `swap()` method defined within the `optional` class, we also have a related stand-alone function of the same name. This allows us to swap the contents of our two `optional` variables using the expression

```
swap(v1,v2);
```

make_optional()

This function offers us an alternative way of creating an `optional` object. Where the type can be deduced from the argument we can use statements such as

```
auto v = make_optional(3.6);
```

which stores a `double` value in *v*.

Where the type cannot be deduced, then we must specify the required type in angled brackets followed by any arguments required by the type's constructor:

```
auto w2 = make_optional<dsk::Date>(25, 12, 2021);
```

The program in FIG-25.2 returns an optional value from the function *SquareRoot()*. The value `nullopt` is returned by the function if the program attempts to find the

square root of a negative number (the function argument is randomly generated). The result returned is displayed.

FIG-25.2

Using *optional*

```
#include <iostream>
#include <optional>
#include <ctime>
using namespace std;

optional<double> SquareRoot(double v);

int main()
{
   srand(time(0));
   double x = rand() % 100 - 50;
   optional <double> result;
   result = SquareRoot(x);
   if (result.has_value())
      cout << "Square root of " << x << " is " << *result << endl;
   else
      cout << "Value is " << x <<". Cannot find square root of a
      ↳negative number\n";
}

optional<double>SquareRoot(double v)
{
   if (v < 0)
      return nullopt;
   else
      return sqrt(v);
}
```

Activity 25.6

Start a new project called *TestOptional* and implement the code given above.

Run the program a few times so that both output messages appear at some point.

Summary

- `decltype` returns the type of a variable or object. Used to specify the type of new variables or objects.
- `declval` returns an rvalue reference which can be used in conjunction with `decltype.variant`

variant

- A `variant` object can be assigned a value of any previously specified type.
- Each entry in the list of allowable types is assigned an index value starting at zero.

Methods

- `emplace()` creates an object directly within the `variant` object.
- `index()` returns the index value of the type of the current contents of the `variant` object.

- `swap()` — swaps the contents of the current `variant` object with a second `variant` object as long as they both contain the same value type.

Related Functions

- All relational operators can be used with `variant` values.
- `get()` — returns the current value stored in a variant object.
- `get_if()` — returns a pointer to the value held in a variant object if that value is of a specified type, otherwise a null pointer is returned.
- `holds_alternative()` — returns *true* if the variant object holds the type specified.
- `visit()` — executes an operation on a variant object by calling a named function or a lambda.
- `swap()` — swaps the contents of two `variant` objects.

any

- A variable of this type can be assigned a value of any type at any time.
- Requires `#include<any>`

Methods

- `type()` — returns an object of class `type_info` allowing us to call name(), ==, or !=.
- `has_value()` — returns *true* if the object has been assigned a value.
- `emplace()` — creates an object directly within the `any` object.
- `reset()` — removes the current value from the `any` object.
- `swap()` — swaps the contents of the current `any` object with a second `any` object.

Related Functions

- `any_cast()` — returns the value in the any object as a specified type.
- `make_any()` — returns an `any` type object.
- `swap()` — swaps the contents of two `any` objects.

optional

- Objects of this type are of a specified value but may contain no value.
- An empty `optional` object has the value `nullopt`.
- Requires `#include <optional>`

Methods

- **`value()`** returns the value of an **`optional`** object.
- **`*`** use this to access the value in an **`optional`** object (an alternative to **`value()`**).
- **`->`** use this to access the public features of any object held in an **`optional`** object.
- **`has_value()`** returns *true* if the **`optional`** object is not empty.
- **`value_or(v)`** returns the value held in the **`optional`** object. If no value is held, *v* is returned.
- **`emplace()`** creates an object directly within the **`optional`** object.
- **`reset()`** removes current value held.
- **`swap()`** swaps the contents of the current **`optional`** object with a second **`optional`** object.

Related Functions

- All relational operators can be used in conjunction with **`optional`** values.
- **`make_optional()`** returns an **`optional`** type object.
- **`swap()`** swaps the contents of two **`optional`** objects.

Program Efficiency

Execution Time Efficiency

Early on in this book we talked about the efficiency of an algorithm, our aim being to create code whose logic is as efficient as possible. For example, one routine created in Chapter 11 was *SumIntegers()*:

```
//*** Returns the sum of all integers between 1 and value ***
int SumIntegers(int value)
{
   int result = 0;
   for (int c = 1; c <= value; c++)
      result += c;
   return result;
}
```

The above algorithm is unbelievably inefficient in as much as we could arrive at the result required by just writing

```
//*** Returns the sum of all integers between 1 and value ***
int SumIntegers(int value)
{
   return (1+value)/2.0*value;
}
```

Another efficiency that can be achieved in execution time is the use of `constexpr` (see Chapter 12). Using this term is a request to calculate results at compile time where necessary and can lead to faster execution times.

Move Semantics

But it may not matter how efficient our algorithm is if we have not also managed the efficient transfer of data. For example, we made use of reference parameters when creating functions in order to eliminate the need to make exact local copies of the actual parameters being passed to the function. Instead, using a reference parameters means that only a copy of the address of the actual parameter is created locally within the function.

Move semantics allows us to apply the same type of efficiency as reference parameters but this time when copying values to or between variables and objects. To show a simple example of what we are trying to achieve, look at the code in FIG-25.3.

FIG-25.3

Deep Copying

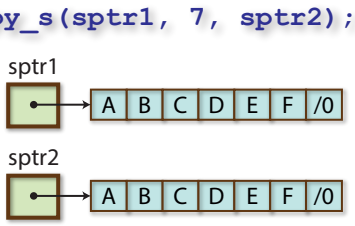
```
#include <iostream>
using namespace std;


int main()
{
   char* sptr1;
   {
      char* sptr2 = new char[7];
      strcpy_s(sptr2, 7, "ABCDEF");
      sptr1 = new char[7];
      strcpy_s(sptr1, 7, sptr2);
   }
   cout << sptr1 << endl;
}
```

The program sets up two C-style strings whose addresses are stored in *sptr1* and *sptr2* with *sptr1* containing an exact copy of the string pointed to by *sptr2* (see FIG-25.4).

FIG-25.4

The Effect of a Deep Copy



This is known as a **deep copy**; each string being entirely separate from the other. And, of course, this makes sense since we want to be able to modify one string without affecting the other. But time and memory space have been needed to make the copy.

If we look more closely at the code in FIG-25.3, we can see that *sptr2* is destroyed immediately after the copy is made since we've arrived at the end of the block in which it is declared.

In this situation we could have saved ourselves the trouble of making the copy referenced by *sptr1* and just have *sptr1* point to the *sptr2*'s space where the required data is already stored – in other words we could replace the lines

```
sptr1 = new char[7];
strcpy_s(sptr1, 7, sptr2);
```

with

```
sptr1 = sptr2;
```

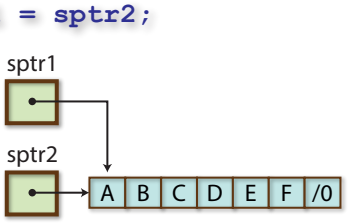
This creates the set up shown in FIG-25.5.

FIG-25.5

Efficient Copying



This method of copying is known as a **shallow copy**.



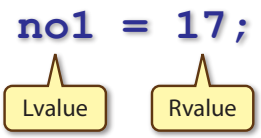
One line later, *sptr2* is destroyed and we are left with only *sptr1* containing the address of the block holding the string. It is this type of efficiency that we are trying to achieve using move semantics.

Lvalues and Rvalues Again

We mentioned **lvalues** and **rvalues** briefly back in Chapter 4 and to simplify things, we can think of an lvalue as referring to any named location where a value can be stored and an rvalue as any unnamed value (see FIG-25.6).

FIG-25.6

Lvalues and Rvalues



The example shows exactly where the two terms are derived from: lvalues are

normally of the left side of the assignment operator and rvalues are normally on the right.

Rvalues can be a constant, the result of an expression, or a value returned by a function as shown in the lines below:

```
double pi = 3.14159; //rvalue: constant
int a = b + c;       //rvalue : the result of the expression
int result = SumIntegers(40);//rvalue: returned from a function
```

Anonymous objects are also rvalues. For example, in the code

```
Date dt1;
dt1 = Date(23,11,1963);
```

the second line creates a temporary *Date* object which is copied into *dt1*.

It should also be noted that these rvalues have a lifetime of only the single line in which they are used. For example, the program assigns unnamed memory space for the result produced by the expression `b + c`, assigns that value to *a* and then frees up that unnamed space.

Although most functions return an rvalue, we can also write a function that returns an lvalue as shown in the short program shown in FIG-25.7. The function *Largest()* returns a reference to the largest value in the global array *list*. Being a global variable, we do not need to pass *list* as a parameter to the function. `main()` calls the function and displays the value found.

FIG-25.7

Functions that Return an Lvalue

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
int& Largest();

//*** Global variables ***
int list[]{ 2,7,1,9,3 };

int main()
{
   cout << Largest() << endl;
}

int& Largest()
{
   int p = 0;
   int highest = list[p];
   for (int c = 1; c < size(list); c++)
      if (list[c] > list[p])
         p = c;
   return list[p];
}
```

Activity 25.7

Start a new project called *TestLRValues* and implement the code given above.

Add code to display the complete contents of *list* at the end of `main()`.

Because *Largest()* returns a reference (which is an lvalue), we can use that returned value of the left side of an assignment as in the line

```
Largest() = 99;
```

which will store the value 99 in the location returned by the function.

Activity 25.8

Modify *TestLRValues* so that the first `cout` statement is replaced by the line

```
Largest() = -1;
```

What values are displayed as the contents of *list*?

We already know that it is possible to find and store the address of an lvalue. For example,

```
int *ptr = &no1;
```

We can also create a reference to an lvalue:

```
int& ref = no1;
```

Even though it must be stored somewhere in memory, it is not normally possible to find the address of an rvalue nor make a reference to it. Hence, the lines

```
int* ptr2 = &17;
int& ref2 = 17;
```

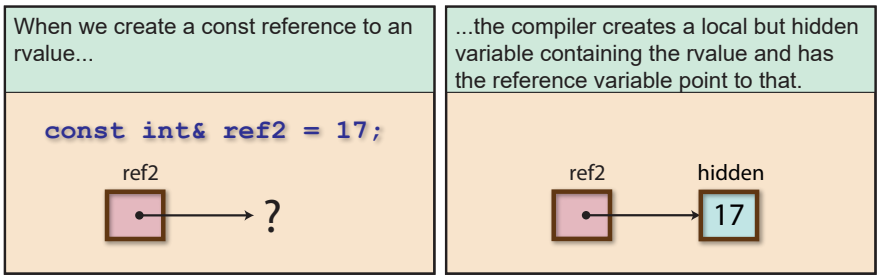
will not compile. And this makes perfect sense since rvalues, such as 17, only exist during the execution of the statement in which they appear and then they vanish. However, C++ will allow us to create a reference to an rvalue if we define that reference as a constant:

```
const int& ref2 = 17;
```

What's really going on here is is shown in FIG-25.8).

FIG-25.8

Referencing an Rvalue



This allows us to link (or bind) a const lvalue to an rvalue and at the same time gives that rvalue an lifetime to match that of its reference variable. But since our lvalue is a constant, we are not allowed to change the value being referenced. So the line

```
ref2 = 21; //Error
```

is invalid.

The same is true for function parameters. So if we had a function whose heading was

```
void f(int& pref)
```

then we would not be allowed to specify an integer constant

```
f(17);      //Not allowed
```

since an lvalue is required as the actual parameter. But if the function heading was written as

```
void f(const int& pref)
```

the call

```
f(17);      //Allowed
```

becomes valid. The situation is highlighted in the program shown in FIG-25.9.

FIG-25.9

Allowing Rvalue Parameters

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
void f1(int&);
void f2(const int&);

int main()
{
   f1(17);
   f2(17);
}

void f1(int& v){cout << v << endl;}

void f2(const int& v){cout << v << endl;}
```

Activity 25.9

Start a new project called *TestLRValues02* and implement the code given above. Does the code compile?

Comment out the call to *f1()* in `main()`. Does the program run?

Rvalue Reference Type

Later versions of C++ have introduced a new way of having an lvalue reference an rvalue. This type is known as **rvalue reference** and allows us to modify the rvalue being referenced.

To create a variable of this type we replace the single ampersand of a standard reference variable with a double ampersand. For example,

```
int&& ref = 17;
```

As before, the compiler deals with this by creating a hidden variable which is referenced by `ref`. But this time, it's not a constant, which means we can follow the previous statement with a line such as

```
ref++;
```

and increment the referenced value.

To differentiate between this new approach and the original one, the `const` reference is termed **lvalue reference**:

```
const int& ref1 = 17;   //lvalue reference
int&& ref2 = 17;        //rvalue reference
```

Activity 25.10

Modify *TestLRValues02* giving *f1()* the signature `void(int&&)`. Re-instate the call to *f1()*.
Does the program compile and run?

Rvalue References and Move Semantics

To understand how we can use an rvalue reference to optimise the copying of data, let's start out with a class in which we create significant amounts of dynamically allocated space. Such a class might be used to store a media element, for example, an image, video or sound recording. Obviously, objects of this class will require significant amounts of space so we will want to reduce the number of times temporary copies of such objects are made.

To demonstrate what we're trying to achieve, we'll start by declaring a simple class in which we intend to store large quantities of data using dynamically allocated space. The coding is given in FIG-25.10.

FIG-25.10

The Media Class

```
#include <iostream>
using namespace std;

class Media
{
   private:
      int size{0};
      int* ptr{nullptr};
   public:
      Media(int sz);
      ~Media();
      Media(const Media& m);
      Media& operator=(const Media& m);
};


Media::Media(int sz)
{
   cout << "Constructor executed\n";
   size = sz;
   ptr = new int[size];
}


Media::~Media()
{
   cout << "Destructor executed\n";
   delete[] ptr;
}


Media::Media(const Media& m)
{
   cout << "Copy constructor executed\n";
   size = m.size;
   ptr = new int[size];
   //*** Copy dynamic space contents (not implemented here) ***
}
```



FIG-25.10
(continued)

The Media Class

```
Media& Media::operator=(const Media& m)
{
   if (this == &m)
      return *this;
   cout << "Assignment operator executed\n";
   delete[] ptr;
   size = m.size;
   ptr = new int[size];
   //*** Copy dynamic space contents (not implemented here) ***
   return *this;
}
```

The attribute *size* determines the size of the space to be created. *ptr* points to the start of that space. The constructor supplies a value for *size*. The other methods are the destructor, copy constructor and the assignment operator.

Notice that no code has been added to copy contents to or from the dynamically allocated block since that operation is not relevant to the point being illustrated here.

Now let's add a minimal `main()` function to the program (see FIG-25.11).

FIG-25.11

Using the Media Class

```
int main()
{
   Media m1(500);
   m1 = Media(100);
}
```

Activity 25.11

Start a new project called *TestMoveSemantics* and implement the *Media* class and `main()` as given above.

When we run the program we see the following output

```
Constructor executed
Constructor executed
Assignment operator executed
Destructor executed
Destructor executed
```

FIG-25.12 highlights what has happened when we execute the lines in `main()` as given above.

FIG-25.12

Creating a Deep Copy

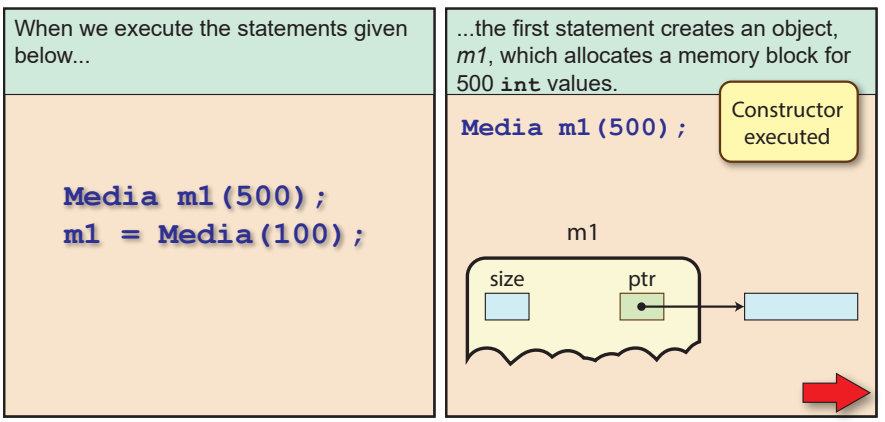
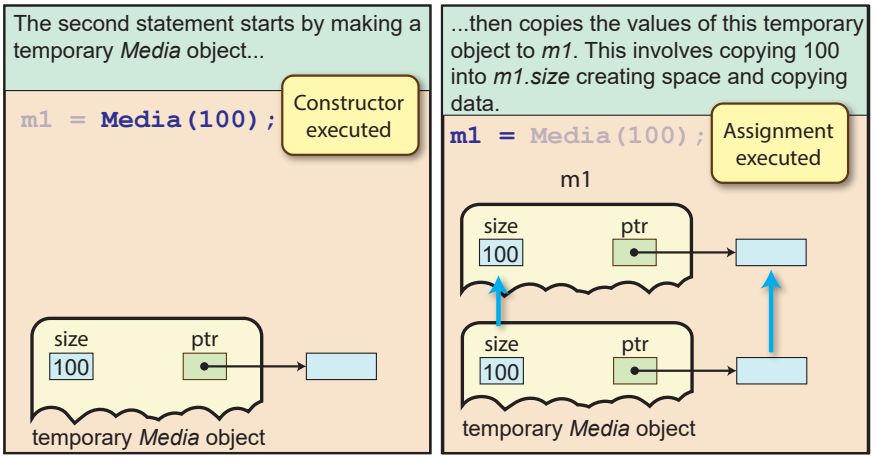


FIG-25.12
(continued)

Creating a Deep Copy



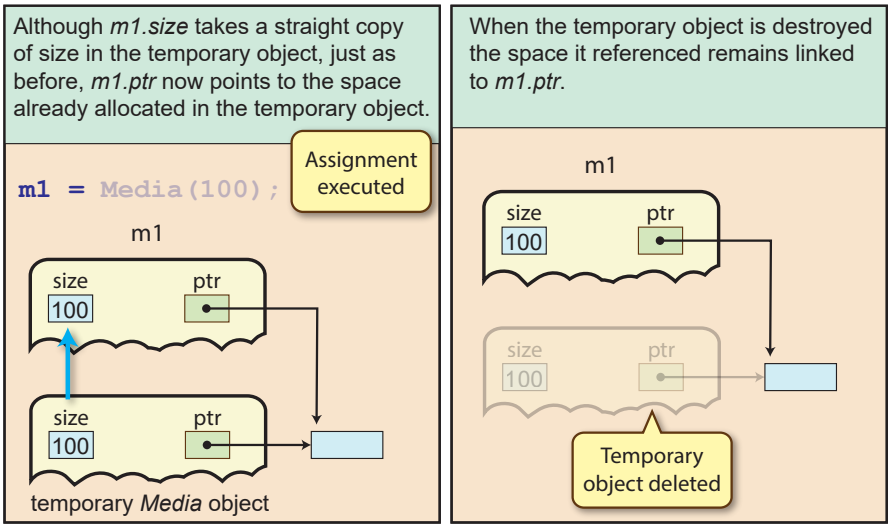
In Frame 4, the space previously pointed to by *m1.ptr* is assumed to have been freed.

Next the temporary, unnamed object is destroyed (the first "Destructor executed" message). And, since we've reached the end of our short program, *m1* is also destroyed (the second "Destructor executed" message).

To optimise our use of time and space we'd like the line `m1 = Media(100);` to execute as shown in FIG-25.13.

FIG-25.13

Using Move Semantics



In Frame 1, the space previously pointed to by *m1.ptr* is assumed to have been freed.

The Move Constructor and the Move Assignment Operator

This is where we find out about two more methods that C++ automatically adds to every class: the **move constructor** and the **move assignment operator**.

Whereas the copy constructor takes a standard reference to an object as its argument (in other words, an lvalue reference), the move constructor takes an rvalue reference as its argument. This means, that in our *Media* class the move constructor would have the heading

```
Media(Media&& m)
```

and the body of the method would be

```
{
   cout << "Move constructor executed\n";
   size = m.size;
   ptr = m.ptr;
   m.ptr = nullptr;
   m.size = 0;
}
```

Notice that the object being created grabs a copy of the parameter's attributes (*m.size* and *m.ptr*) just like the setup in frame 1 of FIG-25.13. But in addition, it has reset the attributes of the parameter, *m*, to zero and `nullptr`.

This last stage, although not entirely necessary, is considered good practice (*m* will be destroyed when we reach the end of the function).

The move assignment operator takes an identical parameter to the move constructor and requires similar logic, but we have to do a little extra work in clearing out the current contents of the object about to be assigned a new value:

```
Media& Media::operator=(Media&& m)
{
   if (this == &m)
      return *this;
   cout << "Move assignment operator executed\n";
   delete[] ptr;
   size = m.size;
   ptr = m.ptr;
   m.ptr = nullptr;
   m.size = 0;
}
```

The `if` statement is a check against self-assignment as in the line

```
m1 = m1;
```

The other extra line,

```
delete[] ptr;
```

frees up the space pointed to by *ptr* before it is assigned a new address.

Activity 25.12

In *TestMoveSemantics* add the two new methods given above to the *Media* class and run the program.

What messages are displayed this time?

This time the new move assignment operator has been used.

The program automatically realises that the expression `Media(100)` creates an unnamed object (that is an rvalue) and therefore calls the version of the assignment operator that takes a reference to that type of object.

move()

In our new move constructor and move assignment operator we've copied the attributes of *m* (*size* and *ptr*) to those in `*this` using the usual assignment operator.

For basic data types such as `int`, `float`, `char` and pointers, this isn't a problem. However, if the attributes include other objects such as `string`, `array` or `vector`

elements, then using the standard assignment operator would result in a deep copy from source to destination and thereby eliminate the efficiency we are trying to achieve with move semantics.

Let's say our *Media* class was defined with an additional field, *filename*, which is a `string` object containing the name of the file from which our media data was loaded. The new definition for the *Media* class would be

```
class Media
{
   private:
      int size;
      int* ptr;
      string filename;
   public:
      Media(int sz);
      ~Media();
      Media(const Media& m);
      Media& operator=(const Media& m);
      Media(Media&& m);
      Media& operator=(Media&& m);
};
```

This would require modifications to the move constructor and the move assignment operator. For example, if we use a standard assignment, the new move constructor code would be

```
Media::Media(Media&& m)
{
   cout << "Move constructor executed\n";
   size = m.size;
   ptr = m.ptr;
   filename = m.filename; //Deep copy
   m.ptr = nullptr;
   m.size = 0;
}
```

Luckily, C++ contains a function temple called `move()` (part of the `std` namespace) which returns an rvalue reference for a given argument. By changing the line

```
filename = m.filename; //Deep copy
```

to

```
filename = move(m.filename);    //Shallow copy
```

The `string` class itself has the necessary move assignment operator method to then handle the rvalue returned by `move()` and move the contents of *m.filename*.

In fact, although not strictly necessary, the `move()` function is normally used to copy all of the attributes, giving us a new coding for our *Media* move constructor:

```
Media::Media(Media&& m)
{
   cout << "Move constructor executed\n";
   size = move(m.size);
   ptr = move(m.ptr);
   filename = move(m.filename);
   m.ptr = nullptr;
   m.size = 0;
}
```

There's no need to reset *m.filename* to an empty string since the string class itself handles this when its move assignment operator is executed.



Don't add a *filename* attribute to the *Media* class.

Activity 25.13

In *TestMoveSemantics* modify the code for the two move methods so that they use the `move()` function.

Is the output the same as before?

Activity 25.14

In *TestMoveSemantics*, if we were to add the line

```
Media m2 = Media(800);
```

what message would we expect this to display on the screen?

Modify the project's code by adding the above statement to `main()`.

Does the program produce the expected message?

The problem here is that the compiler is just too smart! It optimises the code required to create *m2* and eliminates the need for a call to the move constructor.

To force that call, we need to rewrite the line as

```
Media m2 = move(Media(800));
```

which will force a call to the *Media* class's move constructor.

Activity 25.15

In *TestMoveSemantics* use the call to move to assign a value to *m2* as shown above.

How does this affect the output?

Using a Smart Pointer

If we had used a smart pointer to contain the address where our media data was stored as in the lines

```
class Media
{
   private:
      int size{0};
      unique_ptr<int> ptr{nullptr};
      ...
```

then we would have saved ourselves a lot of time. Using a smart pointer, the default move constructor and move assignment will do everything we had to do manually when using a raw pointer.

Our new code for the *Media* class would become

```
class Media
{
   private:
      int size{0};
```

```
        unique_ptr<int> ptr{nullptr};
    public:
        Media(int sz);
        ~Media();
        Media(const Media& m);
        Media& operator=(const Media& m);
        Media(Media&& m) = default;
        Media& operator=(Media&& m) = default;
};
```

Activity 25.16

Update *TestMoveSemantics* to use a smart pointer in the *Media* class and the default move constructor and move assignment.

How does this affect the output?

Summary

- Execution time can be reduced by employing efficient algorithms and the use of the `constexpr` keyword to maximise what can be achieved at compile time.
- Execution time and memory requirements can be reduced by using move semantics where appropriate.
- The term deep copy refers to a copy of an original object which uses separate memory from that original object.
- The term shallow copy refers to the sharing of data space between two or more objects via the use of pointers.
- A term of the form *variable_name&* is termed an lvalue reference.
- A term of the form *variable_name&&* is termed an rvalue reference.
- Rvalue references are used to capture a transient value which can be modified.
- Rvalue parameters can be used to allow the adjusting of pointer contents in such a way as to eliminate the need for deep copying.
- Use `move()` to force a value to be a rvalue reference.

Solutions

Activity 25.1

Code for *TestDeclType*:

```
#include <iostream>
#include "ChronoLib.h"
using namespace std;

template <class T1, class T2>
auto Subtract(T1 v1, T2 v2) -> decltype(v1 - v2)
{
  return v1 - v2;
}

int main()
{
  dsk::Date d1(1, 9, 2023);
  dsk::Date d2(15, 3, 2020);
  dsk::Time t1(12, 36, 19);
  dsk::Time t2(3, 59, 45);
  cout << "Difference between two dates : " <<
  ↳Subtract(d1,d2) << endl;
  cout << "Difference between two times : " <<
  ↳Subtract(t1, t2) << endl;
}
```

Using the values given above, the output is:

```
Difference between two dates : 1265
Difference between two times : 08:36:34
```

Activity 25.2

The output from *TestVariant* is:

```
var1 type index : 0
var2 type index : 4
var1 type index : 4
var2 type index : 0
```

Activity 25.3

Modified code for *TestVariant*:

```
#include <iostream>
#include "ChronoLib.h"
#include <variant>
using namespace std;

int main()
{
  variant <dsk::Date, dsk::Time, int, char, string>
  ↳var1, var2;
  var1.emplace<dsk::Date>(23, 11, 1963);
  var2 = "Time and Relative Dimension In Space"s;
  cout << "var1 type index : " << var1.index() <<
  ↳endl;
  cout << "var2 type index : " << var2.index() <<
  ↳endl;
  var1.swap(var2);
  cout << "var1 type index : " << var1.index() <<
  ↳endl;
  cout << "var2 type index : " << var2.index() <<
  ↳endl;
  cout << " var1 : " << get<4>(var1) << endl;
  cout << " var2 : " << get<0>(var2) << endl;
}
```

The output is:

```
var1 type index : 0
var2 type index : 4
var1 type index : 4
var2 type index : 0
var1 : Time and Relative Dimension In Space
var2 : 23/11/1963
```

Activity 25.4

Modified code for *TestVariant*:

```
#include <iostream>
#include "ChronoLib.h"
#include <variant>
using namespace std;

int main()
{
  variant <dsk::Date, dsk::Time, int, char, string>
  ↳var1, var2;
  var1.emplace<dsk::Date>(23, 11, 1963);
  var2 = "Time and Relative Dimension In Space"s;
  cout << "var1 type index : " << var1.index() <<
  ↳endl;
  cout << "var2 type index : " << var2.index() <<
  ↳endl;
  var1.swap(var2);
  cout << "var1 type index : " << var1.index() <<
  ↳endl;
  cout << "var2 type index : " << var2.index() <<
  ↳endl;
  cout << "var1 : " << get<4>(var1) << endl;
  cout << "var2 : " << get<0>(var2) << endl;
  if (rand() % 2 == 1)
    var1.swap(var2);
  if (holds_alternative<dsk::Date>(var1))
    cout << "var1  now holds: " << get<0>(var1) <<
    ↳endl;
  else
    cout << "var1 does not hold a Date\n";
}
```

Activity 25.5

Modified code for *TestVariant*:

```
#include <iostream>
#include "ChronoLib.h"
#include <variant>
using namespace std;

class Increment
{
public:
  void operator()(dsk::Date& v)
  {
    v+=1;
  }
  void operator()(dsk::Time& v)
  {
    v +=1;
  }
  void operator()(int& v)
  {
    v++;
  }
  void operator()(char& v)
  {
    v = (static_cast<int>(v)+1)%95+32;
  }
  void operator()(string& s) {}
};


int main()
{
  variant <dsk::Date, dsk::Time, int, char, string>
  ↳var1, var2;
  var1.emplace<dsk::Date>(23, 11, 1963);
  var2 = "Time and Relative Dimension In Space"s;
  cout << "var1 type index : " << var1.index() <<
  ↳endl;
  cout << "var2 type index : " << var2.index() <<
  ↳endl;
  var1.swap(var2);
  cout << "var1 type index : " << var1.index() <<
  ↳endl;
  cout << "var2 type index : " << var2.index() <<
  ↳endl;
  cout << "var1 : " << get<4>(var1) << endl;
  cout << "var2 : " << get<0>(var2) << endl;
  if (rand() % 2 == 1)
    var1.swap(var2);
  if (holds_alternative<dsk::Date>(var1))
    cout << "var1  now holds: " << get<0>(var1) <<
  ↳endl;
  else
```

```
    cout << "var1 does not hold a Date\n";
  visit(Increment(), var1);
  visit(Increment(), var2);
  cout << "After Increment(), var1 holds : ";
  if (holds_alternative<dsk::Date>(var1))
    cout << get<0>(var1) << endl;
  else
    cout << get<4>(var1) << endl;
  cout << "After Increment(), var2 holds : ";
  if (holds_alternative<dsk::Date>(var2))
    cout << get<0>(var2) << endl;
  else
    cout << get<4>(var2) << endl;
}
```

The program's last lines of output are:

```
After Increment(), var1 holds : 24/11/1963
After Increment(), var2 holds : Time and Relative Dimension In
↳Space
```

The *Date* object has moved on one day; the **`string`** object is unchanged.

Activity 25.6

No solution required.

Activity 25.7

Modified code for *TestRLValues*:

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
int& Largest();

//*** Global variables ***
int list[]{ 2,7,1,9,3 };

int main()
{
  cout << Largest() << endl;
  cout << "List : ";
  for (auto v : list)
    cout << v<< "  ";
  cout << endl;
}

int& Largest()
{
  int p = 0;
  int highest = list[p];
  for (int c = 1; c < size(list); c++)
    if (list[c] > list[p])
      p = c;
  return list[p];
}
```

Activity 25.8

Modified code for *TestRLValues*:

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
int& Largest();

//*** Global variables ***
int list[]{ 2,7,1,9,3 };

int main()
{
  Largest() = -1;
  cout << "List : ";
  for (auto v : list)
    cout << v<< "  ";
  cout << endl;
}

int& Largest()
{
  int p = 0;
  int highest = list[p];
  for (int c = 1; c < size(list); c++)
    if (list[c] > list[p])
      p = c;
  return list[p];
}
```

The output produced is:

```
List : 2  7  1  -1  3
```

The 9 at the position where the highest value was found has been replaced by -1.

Activity 25.9

*TestRLValues*02 does not compile. The error message

```
'void f1(int &)': cannot convert argument 1 from 'int' to 'int &'
```

is displayed. This highlights the fact that we cannot call *f1()* using an rvalue.

When we comment out this call, the program compiles and runs.

Activity 25.10

Modified code for *TestRLValues*02:

```
#include <iostream>
using namespace std;

//*** Function prototypes ***
void f1(int&&);
void f2(const int&);

int main()
{
  f1(17);
  f2(17);
}

void f1(int&& v) { cout << v << endl; }

void f2(const int& v) { cout << v << endl; }
```

The program now compiles and runs.

Activity 25.11

Code for *TestMoveSemantics*:

```
#include <iostream>
using namespace std;

class Media
{
private:
  int size{ 0 };
  int* ptr{ nullptr };
public:
  Media(int sz);
  ~Media();
  Media(const Media& m);
  Media& operator=(const Media& m);
};


Media::Media(int sz)
{
  cout << "Constructor executed\n";
  size = sz;
  ptr = new int[size];
}


Media::~Media()
{
  cout << "Destructor executed\n";
  delete[] ptr;
}
```

```
Media::Media(const Media& m)
{
   cout << "Copy constructor executed\n";
   size = m.size;
   ptr = new int[size];
   //*** Copy dynamic space contents (not implemented
   here) ***
}

Media& Media::operator=(const Media& m)
{
   if (this == &m)
      return *this;
   cout << "Assignment operator executed\n";
   delete[] ptr;
   size = m.size;
   ptr = new int[size];
   //*** Copy dynamic space contents (not implemented
   here) ***
   return *this;
}


int main()
{
   Media m1(500);
   m1 = Media(100);
}
```

The output produced is:

```
Constructor executed
Constructor executed
Assignment operator executed
Destructor executed
Destructor executed
```

The two *Constructor executed* messages represent the creation of *m1* and the temporary *Media* object.

The *Assignment operator executed* message represents the transfer of data from the temporary object to *m1* using a deep copy approach.

The two *Destructor executed* messages represent the deletion of first the temporary *Media* object and then *m1*.

Activity 25.12

Modified code for *TestMoveSemantics*:

```
#include <iostream>
using namespace std;

class Media
{
private:
   int size{0};
   int* ptr{nullptr};
public:
   Media(int sz);
   ~Media();
   Media(const Media& m);
   Media& operator=(const Media& m);
   Media(Media&& m);
   Media& operator=(Media&& m);
};

Media::Media(int sz)
{
   cout << "Constructor executed\n";
   size = sz;
   ptr = new int[size];
}

Media::~Media()
{
   cout << "Destructor executed\n";
   delete[] ptr;
}

Media::Media(const Media& m)
{
   cout << "Copy constructor executed\n";
   size = m.size;
   ptr = new int[size];
   //*** Copy dynamic space contents (not implemented
   here) ***
}

Media& Media::operator=(const Media& m)
{
   if (this == &m)
      return *this;
   cout << "Assignment operator executed\n";
   delete[] ptr;
   size = m.size;
   ptr = new int[size];
   //*** Copy dynamic space contents (not implemented
   here) ***
   return *this;
}

Media::Media(Media&& m)
{
   cout << "Move constructor executed\n";
   size = m.size;
   ptr = m.ptr;
   m.ptr = nullptr;
   m.size = 0;
}

Media& Media::operator=(Media&& m)
{
   if (this == &m)
      return *this;
   cout << "Move assignment operator executed\n";
   delete[] ptr;
   size = m.size;
   ptr = m.ptr;
   m.ptr = nullptr;
   m.size = 0;
}


int main()
{
   Media m1(500);
   m1 = Media(100);
}
```

The output produced is

```
Constructor executed
Constructor executed
Move assignment operator executed
Destructor executed
Destructor executed
```

As we can see, this time the move assignment operator has been called rather than the standard assignment operator.

Activity 25.13

Modified code for the move constructor and move assignment operator:

```
Media::Media(Media&& m)
{
   cout << "Move constructor executed\n";
   size = move(m.size);
   ptr = move(m.ptr);
   m.ptr = nullptr;
   m.size = 0;
}

Media& Media::operator=(Media&& m)
{
   if (this == &m)
      return *this;
   cout << "Move assignment operator executed\n";
   delete[] ptr;
   size = move(m.size);
   ptr = move(m.ptr);
   m.ptr = nullptr;
   m.size = 0;
}
```

The output is exactly as before.

Activity 25.14

We would expect the sequence of messages to be

Constructor executed	(m1)
Constructor executed	(temporary)
Move assignment operator executed	(temp to m1)
Destructor executed	(temp)
Constructor executed	(new temp)
Move constructor executed	(m2)
Destructor executed	(new temp)
Destructor executed	(m2)
Destructor executed	(m1)

To test our expectations we need to modify the code for **main()** to:

```
int main()
{
  Media m1(500);
  m1 = Media(100);
  Media m2 = Media(800);
}
```

When we execute the program we get the following output:

Constructor executed	(m1)
Constructor executed	(temp)
Move assignment operator executed	(temp to m1)
Destructor executed	(temp)
Constructor executed	(m2)
Destructor executed	(m2)
Destructor executed	(m1)

The actual output is different from our expectations because the compiler has been smart enough to optimise the code at compile time eliminating the need for the *new temp* variable.

Activity 25.15

Modified code for **main()** in *TestMoveSemantics*:

```
int main()
{
  Media m1(500);
  m1 = Media(100);
  Media m2 = move(Media(800));
}
```

This time the output produced meets our original expectations:

Constructor executed
Constructor executed
Move assignment operator executed
Destructor executed
Constructor executed
Move constructor executed
Destructor executed
Destructor executed
Destructor executed

Activity 25.16

Modified code for *TestMoveSemantics*:

```
#include <iostream>
using namespace std;

class Media
{
  private:
    int size;
    unique_ptr<int> ptr;
  public:
    Media(int sz);
    Media(const Media& m);
    Media& operator=(const Media& m);
    Media(Media&& m) = default;
    Media& operator=(Media&& m) = default;
};

Media::Media(int sz)
{
  cout << "Constructor executed\n";
  size = sz;
  ptr = unique_ptr<int>(new int[size]);
}


Media::Media(const Media& m)
{
  cout << "Copy constructor executed\n";
  size = m.size;
  ptr = unique_ptr<int>(new int[size]);
  //*** Copy dynamic space contents (not implemented
  ↳here) ***
}

Media& Media::operator=(const Media& m)
{
  if (this == &m)
    return *this;
  cout << "Assignment operator executed\n";
  size = m.size;
  ptr = unique_ptr<int>(new int[size]);
  //*** Copy dynamic space contents (not implemented
  ↳here) ***
  return *this;
}

int main()
{
  Media m1(500);
  m1 = Media(100);
  Media m2 = move(Media(800));
}
```

Notice that we no longer require a destructor for the Media class since the smart pointer will automatically handle the deallocation of space.

For the same reason, there is no need to call **delete** in the copy assignment operator method.

This time when we run the program the output is

Constructor executed
Constructor executed
Constructor executed

The standard constructor is the only coded method to be executed since the destructor, move constructor and move assignment use the default code which do not contain any message output.

26

In this Chapter:

- ❑ Lambda Syntax
- ❑ Creating a Lambda
- ❑ Calling a Lambda
- ❑ Pointers to Lambdas
- ❑ The function<> Template
- ❑ Passing Parameters to a Lambda
- ❑ Returning a Value from a Lambda
- ❑ Capturing Local Variables in a Lambda
- ❑ Modifying Local Variables in a Lambda
- ❑ Using a Lambda as a Function Parameter
- ❑ Using Lambdas in a Class
- ❑ A Lambda's Class Equivalent

Lambdas

Introduction

Throughout this text, we have written many C-style standard functions. If we were to list the typical characteristics of such functions, we might say that they have a name, take zero or more parameters and often return a value. Where reference parameters are used, the contents of the actual parameter may be modified.

Lambdas (also known as **lambda expressions**) are, in someways, nothing more than short, unnamed functions although they won't normally change the values of data items external to the lambda. Also, while a standard function may be called anywhere throughout a program, lambdas have scope only within the block of code in which they are written. They are perhaps most often used as function parameters.

Lambdas in C++

Lambda Syntax

Lamdas in C++ make use of almost every type of bracket available! The basic format of a lambda is shown in FIG-26.1.

FIG-26.1

Basic Lambda Syntax



where:

capture list	is a comma separated list of in-scope program variables used within the lamda's code. The actual value passed to the lambda is either a copy of the original variable's contents or a reference to that variable.
parameter list	is a list of formal parameters to be used within the lambda. If the parameter list is empty, the parentheses can normally be omitted.
`mutable`	this keyword is used if items in the capture list are to have their values modified in the body of the lambda. Note that this does not affect the original variables unless the captured value is a reference variable.
body	this is the group of executable statements that make up the logic of the lambda.

A lambda definition returns a reference to itself which can be used to allow the lambda to be called in the same way a standard function.

Creating a Lambda

Below we see the code of a lambda:

```
[]() {cout << "This is your lambda calling\n"; };
```

This can be placed anywhere within another function. For example,

```
int main()
{
   [] () {cout << "This is your lambda calling\n"; };
}
```

The above code will happily compile and run – but there will be no output. Creating a lambda is like defining a function; unless we actually call our lambda, it won't get executed.

Executing a Lambda

The simplest way to call our lambda is to add parentheses at the end of the lambda definition:

```
[] () {cout << "This is your lambda calling\n"; }();
```

Alternatively, remembering that a lambda returns its own pointer, we could remove the newly added end parentheses from the lambda and store that returned value. This can then be used to call the code within the lambda:

```
auto lmb = [] () {cout << "This is your lambda calling\n"; };
lmb();   // Call lambda
```

Activity 26.1

Create a new project called *TestLambda01* and test the lambda given above.

In the first version of `main()`, execute the function by adding parentheses to the end of its definition.

In the second version of `main()`, execute the lambda by using the function pointer *lmb*.

Check that both versions produce the same result.

Using a Function Pointer

Although using `auto` to determine the type of the pointer to the lambda is the simplest approach, we can also use a function pointer. The pointer itself could be declared with the statement

```
void(*lmb2)(void);   //lmb2 is the function pointer
```

which tells us that *lmb2* is a pointer to a function that returns no value and takes no parameters. We could then assign the pointer the address of the lambda and execute the lambda using the pointer:

```
lmb2 = [] () {cout << "This is your second lambda calling\n"; };
lmb2();
```

Activity 26.2

In *TestLambda01*, add a second lambda as coded above and check that the output from both lambdas is displayed.

Using *function<>*

The `function<>` template (as defined in the `std` namespace and requiring `#include`

<functional>) is an alternative way to declare a pointer to a lambda:

```
function<void(void)> lmb3 = [] () {cout << "This is your third
↳lambda calling\n"; };
```

Note the similarity to declaring a function pointer. After the keyword `function` and the opening angled bracket, we must give the return type and parameter list types of the lambda. In this case, since the lambda takes no parameters and returns no value, both terms are `void`.

Activity 26.3

Add the third lambda to your project and check for all three outputs.

Using Parameters

Where a lambda requires parameters, formal parameters are added within the first set of parentheses and the actual parameters in the second set of parentheses (see FIG-26.2).

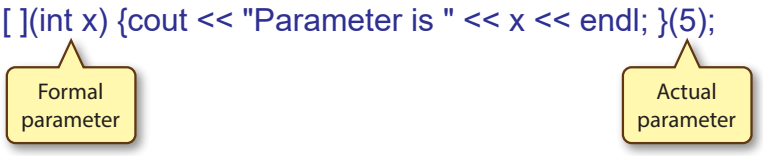


FIG-26.2

Lambda Formal and Actual Parameters

If we are using a pointer to reference the lambda, then the actual parameter will be placed in the parentheses of the call:

```
void(*lmb2)(int) = [](int x){cout<<"Parameter is "<<x<<endl;};
lmb2(5);
```

The actual parameter can be a variable, expression or a constant.

There's no restriction on the number of parameters a lambda can use, nor on the number of code lines within the body of the lambda. For example, we might recreate our *DrawLine()* function which we used to introduce parameters back in Chapter 11 as a lambda:

```
function<void(int, char)> lmb3 = [](int sz, char ch)
{
   for (int c = 1; c <= sz; c++)
      cout << ch;
   cout << endl;
};
lmb3(10,'*');
```



Although no limit is placed on the size of a lambda, most are only one or two lines long.

Activity 26.4

In *TestLambda01*, modify the code in the bodies of the second and third lambdas to match the two examples given above with *x* as a parameter in the first lambda and *sz* and *ch* as parameters in the second. Check that the expected output is produced.

Returning a Value

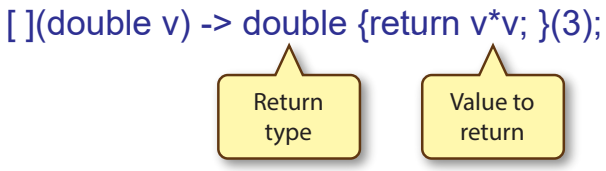
If a lambda is to return a value, the type of that value can be placed between the parentheses and braces. The type should also be preceded by the symbol `->`. The

result is returned by the lambda body by using the keyword `return` – just as we do when returning a value from a standard function.

The lambda in FIG-26.3 returns the square of a given parameter:

FIG-26.3

Returning a Value from a Lambda



Of course, we will probably want to assign the returned value to a variable, used in an expression, or as a parameter to a function:



x is assumed to be some previously declared variable.

DrawLine() is assumed to be the function we created in Chapter 11.

```
double result = [](double v)->double{return v*v;}(x);
double ans = [](double v)->double{return v*v;}(x) + 2;
DrawLine([](double v)->double{return v*v;}(x),'_');
```

Activity 26.5

Start a new project called *TestLambda02* and in `main()` add a lambda to return the square of 28 which should be stored in a variable called *result*.

Finish by displaying the contents of *result*.

In fact, specifying the return type for a lambda is optional since the compiler is smart enough to work out the return type on its own. Therefore, we could rewrite our "squared" lambda call as

```
cout << [](double v) {return v * v; }(3);
```

Activity 26.6

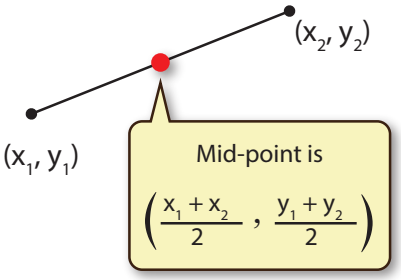
Modify *TestLambda02* so that the lambda is executed within the `cout` statement rather than on a separate line of code.

Returning a *pair* Value

We can return any type of value we need from a lambda. For example, let's assume we want to create a lambda that calculates the mid-point between two given positions in 2D space (see FIG-26.4).

FIG-26.4

Calculating the Mid-Point of a Line



The obvious type of value to return from such a lambda would be a `pair` (see Chapter 9).

The program in FIG-26.5 accepts four values which represent the coordinates of the two end points. These values are passed to a lambda and the mid-point is returned as a `pair` value.

FIG-26.5

Using a Lambda to Return a `pair` Value

```
#include <iostream>
using namespace std;

int main()
{
   double x1, y1, x2, y2;

   //*** Get coordinates of endpoints ***
   cout << " Enter first coordinates :";
   cin >> x1 >> y1;
   cout << " Enter second coordinates :";
   cin >> x2 >> y2;
   //*** Calculate midpoint ***
   pair midpoint =[](double px1, double py1, double px2,
   ↳double py2) {return make_pair((px1 + px2) / 2, (py1 + py2) /
   ↳2); }(x1,y1,x2,y2);
   //*** Display result ***
   cout << "Mid-point : (" << midpoint.first << ',' <<
   ↳midpoint.second << ")\n";
}
```

Activity 26.7

Start a new project called *TestLambda03* and implement the code given above, checking that the returned value is correct.

In 3D space we have an extra dimension to deal with (the z direction). Points in 3D space are defined giving x, y and z values (x,y,z). The midpoint between two points in 3D space uses the same formula as that for 2D space but with a third part to the expression to calculate the z value.

Modify *TestLambda03* so that two 3D coordinates are entered and the lambda returns a `tuple` containing the midpoint coordinates.

Accessing a Local Variable in a Lambda

Although we can pass a copy of a local variable as the actual parameter of a lambda as in the example below

```
double num = 3.2;
cout << [](double v)->double {return v * v; }(num) << endl;
```

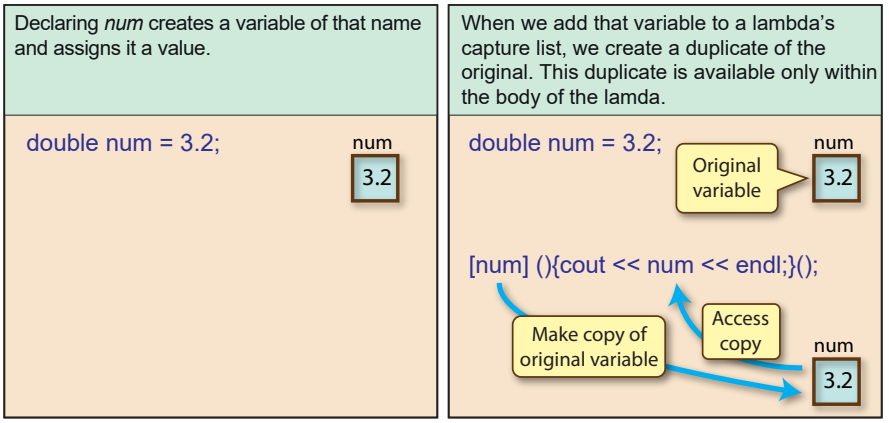
we can access a copy of any local variable by adding its name to the capture list within the square brackets. For example, if we want a lambda to access a copy of the contents of variable *num*, we could write

```
double num = 3.2;
[num]() {cout << num << endl; }();
```

Remember the variable *num* used within the lambda's body is not the same *num* as defined in the previous line, but simply a copy of the contents of that earlier variable (see Fig-26.6).

FIG-26.6

Lambda Captures are Copies



Should we want a lambda to access more than one local variable from the surrounding code, we can create a comma-separated list within the capture brackets:

```
double num = 3.2;
int val = 12;
char ch = 'A';

[num,val,ch](){cout << num <<"   " << val <<"   "<<  ch <<
endl;}();
```

However, if we want to access all local variables that are in-scope, we can replace the capture list with the single character =:

```
double num = 3.2;
int val = 12;
char ch = 'A';
[=](){cout << num <<"   " << val <<"   "<<  ch << endl;}();
```

Although using = for capture gives access to all variables that are in-scope, it only makes copies of those variables that are actually used inside the lambda.

Activity 26.8

Modify *TestLambda03* so that the values *x1*, *y1*, *z1*, *x2*, *y2*, and *z2* are captured rather than passed as parameters. Check that the result is the same as before.

As well as primitive types, objects can also be captured:

```
dsk::Date dt1(1, 1, 2021);
[dt1] {cout << dt1 << endl; }();
```

To make a copy of the contents of an array available within a lambda, all we need to do is add the array name to the capture list. For example, the following code makes the array *list* available within the lambda:

```
int list[] = { 22,6,7,0 };
[list]() {cout << list[0] << endl; }();
```

When a lambda captures a copy of a variable such as *num, dt1* or *list* in the examples above, the actual value captured is that held by the variable at the point in the program where the lambda is defined, not when the lambda is executed. For example, in the following code

```
double num = 12;
auto lmb = [num] {cout << num << endl; };//Define lambda
num = 20;
lmb();                                    //Execute lambda
```

The lambda's empty parentheses have be omitted.

local variable *num* has the value 12 when the lambda is defined but has the value 20 when the lambda is executed. However, execution of the lambda will display the value 12, not 20.

Activity 26.9

Start a new project called *TestLambda04* adding code given above to `main()` and check the value displayed by the lambda in the new code.

Although the lambda now has a copy of local variable *num*, that copy's value cannot be changed within the lambda. Hence, the code

```
auto lmb = [num]
{
   --num;      //Invalid - cannot modify captured variable
   cout << num << endl;
};
```

will not compile since it attempts to change the value of the lambda's copy of *num*.

In order to modify the copy within the lambda we must add the keyword `mutable` immediately before the lambda body's parentheses:



When the term `mutable` is included, the empty parentheses must be included.

```
[num]()mutable
{
   --num;      //Valid
   cout << num << endl;
}();
```

Activity 26.10

Modify *TestLambda04* to match the lambda given above.

Try executing the lambda in the following ways:

a) By adding parentheses to the end of the lambda definition (as shown above).
b) By using a pointer created using the `function<>` template.
c) By using a standard function pointer.

The result of Activity 26.10 c) highlights the difference between standard pointers used to access a lambda and pointers created using `function<>`. Standard function pointers cannot be used for lambdas which contain a non-empty capture list.

Capturing an Expression

The capture list isn't restricted to variables. We can also capture an expression by assigning it to a new variable. For example, we can capture twice the value in variable *val* using the following code

```
int val = 12;
auto lmb = [x = val * 2](){ return x; };
```

Notice that we have to assign the captured expression (`val * 2`) to a new variable *x* but that no type is specified for this variable.

Using an output statement, we can display the contents of the value returned by the lambda:

```
cout << lmb() << endl;
```

Activity 26.11

Start a new project called *TestLambda05* and create a program which creates a lambda based on the three lines given above.

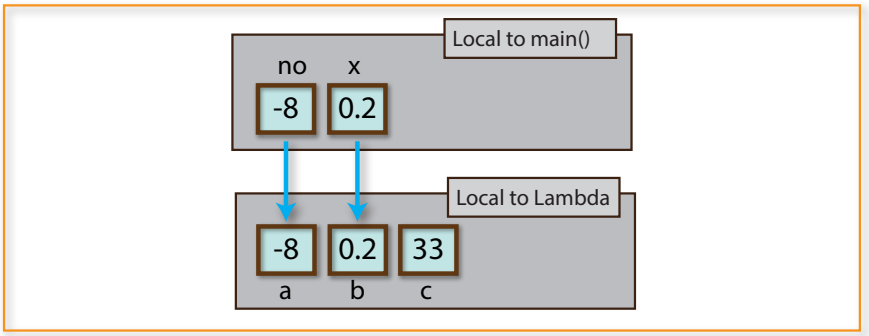
In the capture list, we're not restricted to only assigning new names to expressions as we saw in the example above. We can also assign names to a copied variable or even a constant. For example,

```
int no = -8;
double x = 0.2;
[ a = no, b = x, c = 33](){...}();
```

We can see a visual representation of this in FIG-26.7.

FIG-26.7

Captured By Value



Variables created in the capture list (by giving them a name followed by =) act like static variables, retaining their current value between calls to the lambda. For example, using the following lambda

```
auto lmb = [c = 0]() mutable {return ++c; };
```

c will increment each time the lambda is called. The first call to lambda with a line such as

```
cout << lmb() << endl;
```

will display the value 1. Executing the same line for a second time will display the value 2.

By capturing two expressions, we could create a lambda which returned the next number in the factorial sequence:



Each time the lambda is executed, the new value of *c* is the old value of *c*t* and the new value of *t* is its old value + 1.

```
auto lmb2 = [c = 1, t = 1]()mutable{return c = c*t++; };
cout << lmb2() << endl;
cout << lmb2() << endl;
cout << lmb2() << endl;
cout << lmb2() << endl;
cout << lmb2() << endl;
```

The above code outputs the values

```
1
2
6
24
120
```

Activity 26.12

In *TestLambda05* add the factorial-calculating lambda to `main()` and execute it multiple times as exampled above.

Modifying an External Variable within a Lambda

Although not generally recommended, a lambda is capable of modifying an external variable in the surrounding code by passing the address of that variable in the capture list. For example, the following lambda increments the value in *num* before its contents are displayed.

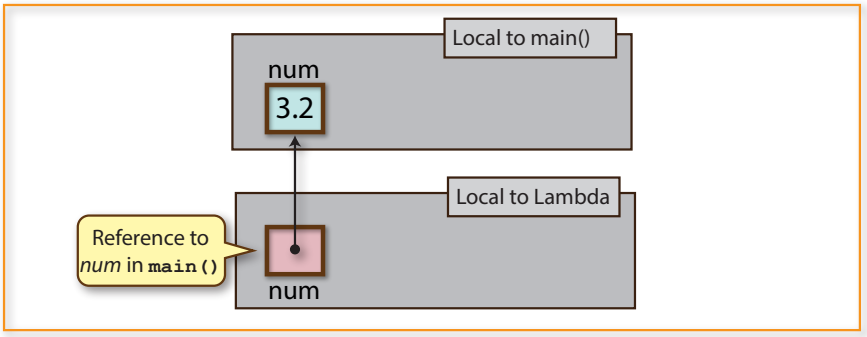
```
double num = 3.2;
[&num](){++num;}();
cout << num << endl;
```

Notice that there is no requirement to include the keyword `mutable` when a variable external to the lambda is having its value modified by the lambda.

The setup created by the above code is shown in FIG-26.8.

FIG-26.8

Captured by Reference



Activity 26.13

Modify *TestLambda05* by adding the three lines given above and check that the value of *num* is modified by the lambda.

If we want to give the local reference variable within the lambda a different name from the one in `main()`, say *refnum*, then the lambda would be written as

```
[&refnum = num](){++refnum;}();
```

If we want all variables in the surrounding code to be mutable, then we replace the capture list with the single character `&`. For example

```
double num = 3.2;
int val = 12;
dsk::Date dt1(1,2,2021);
[&](){++num;--val;dt1++;}();
```

When using a referenced capture it is the captured variable's value at the point of execution that is used, not its value when the lambda is defined. For example, the following code *num* will have the value 20 within the lambda, not 12.

```
int num = 12;
auto lam = [&num]() {cout << num << endl; };
num = 20;
```

```
lam();
```

So, the above code will display the value

```
20
```

Accessing and Returning a *vector* Object

In the next example, the lambda accesses a `vector` object and returns a second `vector` object containing copies of all even values in the first vector.

```
vector <int> vec{ 1,2,3,4,5,6,7,8 };

//*** Define and execute lambda ***
auto ans =
[vec]()
{
   vector<int> result;
   for (auto v : vec)
      if (v % 2 == 0)
         result.push_back(v);
   return result;
}();

//*** Display returned vector ***
for (auto v : ans)
   cout << v << "  ";
cout << endl;
```

Activity 26.14

Start a new project called *TestLambda06* and implement the code given above checking that the vector *result* contains only even values from *ans*.

Using a Lambda as a Function Parameter

Back in Chapter 20 we looked at functions such as `count_if()`, `find_if()` and `for_each()` in the Algorithm Package. Many of these functions required function pointers as parameters. For example, we saw how to obtain a count of how many even numbers were in the vector *testdata* with code such as

```
int sum = count_if(testdata.begin(),testdata.end(),IsEven)
```

where the parameter *IsEven* is the address of the function *IsEven()* which was coded as follows:

```
bool IsEven(int v)
{
   return (v % 2 == 0);
}
```

If this call to `count_if()` is the only time we make use of the *IsEven()* function, it's a waste of effort to write its code as a separate function. Instead, we can use a lambda placing its code directly in the parameter list of the call to `count_if()`:

```
int sum = count_if(testdata.begin(),testdata.end(),[](int v)
↳{return(v % 2 == 0);});
```

This is perfectly valid since a lambda definition returns a pointer to the address of the lambda.

Activity 26.15

Reload *TestAlgorithmLibrary05* a which you created as an Activity in Chapter 20 and replace the *DisplayOdd()* and *Cube()* functions with lambdas.
Also, change the *for_each_n()* parameter so that all 20 elements are processed.

Functions that Return Lambda Pointers

Back in Chapter 20, we ended up creating a class in order to generate random numbers in a given range. But by creating a function that returns a lambda, we can achieve the same result without the need to define a class.

As usual, we need to define the return type of our function and this will be a pointer to a lambda. Our lambda's address is going to be used as the final parameter in the algorithm library's `generate()` function. That final parameter must be the address of a function with the signature `int(void)` – where `int` is the type of value held in the container to be processed.

The function we are about to write must return a pointer to a lambda of that signature, so the first part of our code is

```
function<int(void)>
```

We'll name our function *RandomRange()* and give it parameters which specify the minimum and maximum random values that should be generated:

```
function<int(void)> RandomRange(int min, int max)
```

The lambda inside the function can then capture these two values and use them to generate a random value in the specified range:

```
function<int(void)> RandomRange(int min, int max)
{
   return [min,max]() {return rand() % (max - min + 1) + min;
};
```

A program that uses the value returned by the function in conjunction with `generate()` to store a set of random values between 10 and 21 in a vector *src* is shown in FIG-26.9.

FIG-26.9

Using a Function that Returns a Lambda

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <iomanip>
using namespace std;
//*** Function prototypes ***
function<int(void)> RandomRange(int min, int max);

int main()
{
   vector<int> src(20);
   //*** Generate values for the vector ***
   generate(src.begin(), src.end(), RandomRange(10, 21));
   //*** Display the values stored ***
   for (int v : src)
      cout << setw(2) << v << ' ';
   cout << endl;
}
```



FIG-26.9
(continued)

Using a Function that Returns a Lambda

```
function<int(void)> RandomRange(int min, int max)
{
   return [min,max]() {return rand() % (max - min + 1) + min; };
}
```

Activity 26.16

Start a new project called *TestLambda07* and implement the code given above.

Modify the code so the numbers generated are in the range 1 to 90.

Add a second function, *AddValue()*, that returns a pointer to a lambda with the signature `int(int)`. The function should take an `int` parameter *v*. The lambda within the function should capture *v*, have an `int` parameter *x,* and a body which adds *v* to *x*.

The value returned by this function should be used as the fourth parameter in a call to `transform()` (a Algorithm Library function covered in Chapter 20). The first three parameters being *src.begin()*, *src.end()*, *src.begin()*.

Use this setup to add 5 to every value in *src*.

Using a Lambda in a Class Method

A lambda can be used within the method of a class as shown in the trivial example given in FIG-26.10.

FIG-26.10

Using a Lambda in a Class

```
#include <iostream>
using namespace std;

class Simple
{
   private:
      int v;
   public:
      Simple(int p) { v = p; };
      double Recip(){ return [this]() {return 1.0 / v ; }(); };
};

int main()
{
   Simple x(20);
   cout << x.Recip() << endl;
}
```

Notice that where a lambda needs access to the attributes of a class, these cannot be captured individually, so starting the above lambda with

```
[v](){...
```

is invalid. Instead we have to capture the whole object using the term `this`.

Activity 26.17

Start a new project called *TestLambda08* and implement the code given above checking that the result is as expected (0.05).

Summary

- A lambda is a type of unnamed function.
- A lambda's definition returns the address of the lambda.
- A lambda's lifetime is limited to the block in which it is defined – it cannot be executed outside that block.
- The brackets at the start of a lambda can enclose the names of in-scope variables whose value or reference is to be made available within the lambda's code. These are known as capture values.
- Use `[=]` to allow the capture of any in-scope variables.
- Use `[&]` to allow the capture of a reference to any in-scope variables.
- Captured values are stored in local variables of the same name as the original variable.
- Captured value copies can only be modified if the term `mutable` is included before the body of the lambda.
- Changes to a captured reference will change the contents of the original variable which is being referenced.
- Parameters to a lambda are included within the parentheses in a similar manner to parameters in a standard function.
- Parameters cannot be given default values.
- Optionally, the type of value returned by a call to a lambda can be specified immediately before the block containing the lambda code using the format

  `->` *return type*

- Captured values and references can be assigned a different name within the capture list.
- Captured values are static and retain their value between calls.
- If a captured value is modified within the body of a lambda, the term `mutable` must be added before the body of the lambda.
- The address of a lambda can be stored in a pointer variable as long as it has an empty capture list.
- To assign the address of a lambda containing a capture list to a variable, use `function<>`.
- Lambdas are often used as the actual parameter in a call to functions which take function addresses as parameters.
- A function can return the address of a lambda.
- Lambdas can be embedded within a class.
- A class lambda uses the term `this` in its capture list to gain access to all of the attributes of the class.

Solutions

Activity 26.1

Code for *TestLamda01*:

```
#include <iostream>
using namespace std;

int main()
{
   []() {cout << "This is your lambda calling\n";}();
}
```

Output produced:

```
This is your lambda calling
```

Modified version of *TestLambda01*:

```
#include <iostream>
using namespace std;

int main()
{
   auto lmb = []() {cout << "This is your lambda
   calling\n"; };
   lmb();
}
```

This version produces the same output as before.

Activity 26.2

Modified version of *TestLambda01*:

```
#include <iostream>
using namespace std;

int main()
{
   auto lmb = []() {cout << "This is your lambda
   calling\n"; };
   lmb();

   void(*lmb2)(void);
   lmb2 = []() {cout << "This is your second lambda
   calling\n"; };
   lmb2();
}
```

Output produced:

```
This is your lambda calling
This is your second lambda calling
```

Activity 26.3

Modified version of *TestLambda01*:

```
#include <iostream>
#include <functional>
using namespace std;

int main()
{
   auto lmb = []() {cout << "This is your lambda
   calling\n"; };
   lmb();

   void(*lmb2)(void);
   lmb2 = []() {cout << "This is your second lambda
   calling\n"; };
   lmb2();

   function<void(void)> lmb3 = []() {cout << "This is
   your third lambda calling\n"; };
   lmb3();
}
```

Output produced:

```
This is your lambda calling
This is your second lambda calling
This is your third lambda calling
```

Activity 26.4

Modified version of *TestLambda01*:

```
#include <iostream>
#include <functional>
using namespace std;

int main()
{
   auto lmb = []() {cout << "This is your lambda
   calling\n"; };
   lmb();

   void(*lmb2)(int) = [](int x) {cout << "Parameter
   is " << x << endl; };
   lmb2(5);


   function<void(int, char)> lmb3 = [](int sz, char
   ch)
   {
      for (int c = 1; c <= sz; c++)
         cout << ch;
      cout << endl;
   };
   lmb3(10, '*');
}
```

Output produced:

```
This is your lambda calling
Parameter is 5
**********
```

Activity 26.5

Code for *TestLamda02*:

```
#include <iostream>
using namespace std;

int main()
{
   double result = [](double v)->double {return v*v;
   }(28);
   cout << "The square of 28 is " << result << endl;
}
```

Output produced:

```
The square of 28 is 784
```

Activity 26.6

Modified version of *TestLambda02*:

```
#include <iostream>
using namespace std;

int main()
{
   cout << "The square of 28 is " << [](double v)->
   double {return v * v; }(28) << endl;
}
```

The output produced is unchanged.

Activity 26.7

Modified version of *TestLambda03*:

```
#include <iostream>
using namespace std;

int main()
```

```
{
  double x1, y1, z1, x2, y2, z2;

  //*** Get coordinates of endpoints ***
  cout << " Enter first coordinates :";
  cin >> x1 >> y1 >> z1;
  cout << " Enter second coordinates :";
  cin >> x2 >> y2 >> z2;
  //*** Calculate midpoint ***
  tuple midpoint =[](double px1, double py1, double
  ↳pz1, double px2, double py2, double pz2) {return
  ↳make_tuple((px1 + px2) / 2, (py1 + py2) / 2,
  ↳(pz1 + pz2) / 2); }(x1,y1,z1,x2,y2,z2);
  //*** Display result ***
  cout << "Mid-point : (" << get<0>(midpoint) << ','
  ↳<< get<1>(midpoint) << ',' << get<2>(midpoint)
  ↳<< ")\n";
}
```

Activity 26.8

Modified code for *TestLambda03*:

```
#include <iostream>
using namespace std;

int main()
{
  double x1, y1, z1, x2, y2, z2;

  //*** Get coordinates of endpoints ***
  cout << " Enter first coordinates :";
  cin >> x1 >> y1 >> z1;
  cout << " Enter second coordinates :";
  cin >> x2 >> y2 >> z2;
  //*** Calculate midpoint ***
  tuple midpoint = [=]() {return make_tuple((x1 +
  ↳x2) / 2, (y1 + y2) / 2, (z1 + z2) / 2); }();
  //*** Display result ***
  cout << "Mid-point : (" << get<0>(midpoint) << ','
  ↳<< get<1>(midpoint) << ',' << get<2>(midpoint)
  ↳<< ")\n";
}
```

Activity 26.9

Code for *TestLambda04*:

```
#include <iostream>
using namespace std;

int main()
{
  double num = 12;
  auto lmb = [num] {cout << num << endl; };  //Define
                                              ↳lambda
  num = 20;
  lmb();                          //Execute lambda
}
```

The output produced by the new lambda is:

```
12
```

Activity 26.10

Modified code for *TestLambda04*:

a)

```
#include <iostream>
using namespace std;

int main()
{
  double num = 12;
  [num] ()mutable{--num; cout << num << endl; }();
  num = 20;
}
```

b)

```
#include <iostream>
#include <functional>
using namespace std;

int main()
{
  double num = 12;
  function<void(void)> lmb = [num] ()mutable{--num;
  ↳cout << num << endl; };
  num = 20;
  lmb();
}
```

c)

```
#include <iostream>
using namespace std;

int main()
{
  double num = 12;
  void(*lmb)(void) = [num] ()mutable{--num; cout <<
  ↳num << endl; };
  num = 20;
  lmb();
}
```

Versions a) and b) produce the same output:

```
11
```

Version c) does not compile.

Activity 26.11

Code for *TestLambda05*:

```
#include <iostream>
using namespace std;

int main()
{
  int val = 12;
  auto lmb = [x = val * 2](){ return x; };
  cout << lmb() << endl;
}
```

Activity 26.12

Modified code for *TestLambda05*:

```
#include <iostream>
using namespace std;

int main()
{
  int val = 12;
  auto lmb = [x = val * 2](){ return x; };
  cout << lmb() << endl;

  auto lmb2 = [c = 1, t = 1]()mutable{return c = c *
  ↳t++; };
  cout << lmb2() << endl;
  cout << lmb2() << endl;
  cout << lmb2() << endl;
  cout << lmb2() << endl;
  cout << lmb2() << endl;
}
```

The output produced by the new lambda is:

```
1
2
6
24
120
```

Activity 26.13

Modified code for *TestLambda05*:

```
#include <iostream>
using namespace std;

int main()
{
  int val = 12;
  auto lmb = [x = val * 2](){ return x; };
  cout << lmb() << endl;

  auto lmb2 = [c = 1, t = 1]()mutable{return c = c *
  ↳t++; };
```

```
   cout << lmb2() << endl;
   cout << lmb2() << endl;
   cout << lmb2() << endl;
   cout << lmb2() << endl;
   cout << lmb2() << endl;

   double num = 3.2;
   [&num]() {++num; }();
   cout << num << endl;
}
```

The output produced by the new lambda is:

```
4.2
```

Activity 26.14

Code for *TestLambda06*:

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{

     vector <int> vec{ 1,2,3,4,5,6,7,8 };

     //*** Define and execute lambda ***
     auto ans =
        [vec]()
     {
        vector<int> result;
        for (auto v : vec)
           if (v % 2 == 0)
              result.push_back(v);
        return result;
     }();

     //*** Display returned vector ***
     for (auto v : ans)
        cout << v << "  ";
     cout << endl;
}
```

The output produced is:

```
2 4 6 8
```

Activity 26.15

Modified code for *TestAlgorithmLibrary05*:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iomanip>
using namespace std;

int main()
{
   vector<int> testdata(20);

   //*** Store random values between 0 and 10 ***
   for (int c = 0; c < 20; c++)
      testdata[c] = rand() % 11;
   //*** Display the values stored ***
   for (int v : testdata)
      cout << setw(2) << v << ' ';
   cout << endl;
   //*** Display only odd values ***
   cout << "Odd numbers in testdata : ";
   for_each(testdata.begin(), testdata.end(), []
   (int v) {if (v % 2 == 1) cout << v << "  "; });
   cout << endl;
   //*** Cube all values ***
   for_each_n(testdata.begin(), 20, [](int& v)
   {v *= v * v; });
   //*** Display the values stored ***
   cout << "testdata after cubing : ";
   for (int v : testdata)
      cout << v << "  ";
   cout << endl;
}
```

Note that the functions and their prototypes have been removed.

Activity 26.16

Final code for *TestLambda07*:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <iomanip>
using namespace std;

//*** Function prototypes ***
function<int(void)> RandomRange(int min, int max);
function<int(int)> AddValue(int v);

int main()
{
   vector<int> src(20);

   //*** Generate values for the vector ***
   generate(src.begin(), src.end(), RandomRange(
   1, 90));
   //*** Display the values stored ***
   for (int v : src)
      cout << setw(2) << v << ' ';
   cout << endl;
   //*** Add to every value in src ***
   transform(src.begin(), src.end(), src.begin(),
   AddValue(5));
   //*** Display the values stored ***
   for (int v : src)
      cout << setw(2) << v << ' ';
   cout << endl;
}

function<int(void)> RandomRange(int min, int max)
{
   return [min,max]() {return rand() % (max - min +
   1) + min; };
}

function<int(int)> AddValue(int v)
{
   return [v](int x) {return x + v; };
}
```

Activity 26.17

No solution required.

Index

Symbols

- - 131
! 154
() 100
& 118, 292
&& 154
% 99
+ 98
++ 131
< 147
<< 121
<= 147
!= 147
== 147
> 147
>= 147
>> 116
|| 154
[[deprecated]] 372
[[fallthrow]] 167
.lib files 610
[[maybe_unused]] 373
[[nodiscard]] 373
.obj files 610
#pragma once 595, 598

A

abort() 649
abstract classes 834
accessing record fields 254
access specifier 528
actual parameter 342
address of operator 118, 292
aggregation 845
algorithm package 708
  adjacent_find() 715
  all_of() 708
  any_of() 711
binary_search() 735
copy() 722
copy_backward() 722
copy_if() 722
copy_n() 722
count() 712
count_if() 712
fill() 725
fill_n() 725
find() 713
find_end() 719
find_first_of() 713
find_if() 713
find_if_not() 713
for_each() 720
for_each_n() 721
generate() 727
generate_n() 727
iter_swap() 730
move() 725
move_backward() 725
none_of() 711
remove() 728
remove_copy() 728
remove_copy_if() 728
remove_if() 728
replace() 729
replace_copy() 729
replace_copy_if() 729
replace_if() 729
reverse() 730
reverse_copy() 730
rotate() 731
rotate_copy() 731
search() 717
search_n() 720
shuffle() 732
sort() 734
swap() 730
swap_ranges() 730
transform() 725
unique() 732

unique_copy() 732
algorithms 2
AND 19
any 921
any_cast() 923
emplace() 922
has_value() 922
make_any() 923
reset() 923
swap() 923, 924
type() 922
arithmetic assignment operators 131
array 206
accessing elements 208
cells 206
declaring 207
elements 206
index 207
index expressions 214
index variables 209
initialising 207
multi-dimensional 235
of characters 231
of constants 241
size() 212
subscript 207
wide character 235
array class
at() 672
back() 673
begin() 677
cbegin() 678
cend() 678
data() 675
empty() 672
end() 677
fill() 672
front() 673
iterators 676
max_size() 672
rbegin() 678
relational operators 674
rend() 679
size() 672
swap() 673
array container class 671
array of objects 569
array parameters 361
arrays
of pointers 305
arrays of records 252
arrays within records 251
ASCII 86
assert() 633
assignment operator 797
assignment statement 97
auto 90
and pointers 295
auto return 371

B

base class 785
basic data types 84
binary 44
binary files 495, 890
binary selection 6
bit complement operator 267
bit fields 265
unnamed 266
bitwise AND 270
bitwise assignment operators 272
bitwise OR 271
bitwise XOR 272
black box testing 376
block 150
BMI 122
bool 84, 87, 156
Boolean 84
Boolean expression 7
Boolean operators
mixing 23
Boolean variables 156
boundary testing 378
break 165, 191

within a nested loop 197
byte 44

C

C++ exception classes 850
calculation 25
calling a function 338
call operator 543
case 166
catch block 636
categorising bits 267
char 87
character arrays 231
character coding 58
ASCII 58
Unicode standard 59
UTF-8 59
character functions 485
isalnum() 486
isalpha() 486
isblank() 485
isdigit() 486
isgraph() 486
islower() 487
isprint() 486
ispunct() 487
isscntrl() 485
isspace() 485
isupper() 487
isxdigit() 487
tolower() 489
toupper() 489
child class 785
cin 116
class assignment operator 537
class attributes 525
class constants 547
class diagrams 556
classes 524
class features 525
class scope 526
Clock class 801
comma operator 133
comments 136
comparison 26
compilation errors 632
compiler 69
compiling 75
complex condition 19
composite data structures 250
composition 845
compound condition 153
computer program 4
condition 7
conditional assignment statement 163
console applications 71
const 97
constant 93
constant arrays 241
constant pointers 300
constants 84, 93
constexpr 396
const methods 565
const parameters 358
constructor 534
constructor inheritance 795
constructor initialisation 89
constructors 794
container classes 657
containers 214
continue 192
in nested loop 197
copy constructor 536, 795
cos() 452
CountdownClock class 806
counts 26
cout 121
creating a static library 610
creating static libraries 593
C-style file handling
end-of-file byte 504
fclose() 498
feof() 510
fgets() 502

FILE 497
fopen() 497
fprintf() 498
fputc() 500
fputs() 499
fread() 507
fscanf() 501
fseek() 512
ftell() 514
fwrite() 506
printer output 515
reading a text file 501
reading from a binary file 507
remove() 515
rename() 514
rewind() 513
searching a binary file 510
separating data items 500
updating records 511
writing to a binary file 505
C-style string functions 472
atof() 480
atoi() 481
atol() 482
atoll() 483
gets_s() 473
sprintf() 479
strcat_s() 473
strchr() 475
strcmp() 473
strcpy_s() 473
strcspn() 477
strlen() 473
strncat_s() 475
strncmp() 475
strncpy_s() 474
strpbrk() 477
strrchr() 476
strspn() 476
strstr() 477
strtod() 480
strtof() 481
strtok() 478
strtol() 482
strtold() 481
strtoll() 483
strtoul() 482
strtoull() 482

D

data 24
operations on 25
data hiding 540
data type variations 85
DateType 419
day of the week 138
declaration statement format 90
declaring variables 88
decltype 914
declval 917
decrement operator 131
default parameter values 394
deferred typing 427
delete 313, 805
delimiters 94
denary 44
derived class 785
designing a function 336
desk checking 32
destructors 539, 797
double 86
do...while 183
downloading 67
dry run 32
dynamic link libraries 610

E

editor 68
ELSE 8
encapsulation 526
endl 122
entry condition 18
enumerated types 259
error flags 635

error handling 632
error list 75
errors 31
  logic 31
escape codes 94
exception handler 636
exception handling 636
exceptions 850
exit condition 17
explicit 538
expression 146
extraction operator 116

F

fields 250
file formats 495
file scope 399
final 808, 829
float 86
floating-point 84
fold expressions 436
for 185
  and the comma operator 190
  conditional expression 187
  initial expression 187
  loop expression 187
FOR..ENDFOR 14
formal parameter 341
format string 127
friends 554, 788
fstream class 894
function<> 949
function attributes 372
function complexity 382
function definition 337
function heading 337
function overheads 373
function overloading 401
function parameter pack 432
function pointers
  as parameters 412
function prototype 340
functions 336
  that return lambda pointers 958
function signature 342
function templates 427
  explicit instantiation 430
functors 543

G

getchar() 465
gets_s() 233
global namespace 129
global scope 399
global variables 399
goto 193

H

Hangman 9
header file 594
hexadecimal 93
hex-defined character 95
hiding inherited methods 789

I

identifier 88
if 146
  nested 160
if...else 157
ifstream class 887
  close() 888
  is_open() 887
  open() 887
IF...THEN...ELSE...ENDIF 8
IF...THEN...ENDIF 7
implementation file 596
include guard 600
increment operator 131
indirection operator 297
inheritance 784
initialising struct variables 253
initial value 89
inline functions 395
inline methods 544

inner block 151
inner loop 196
in parameter 341
input 25, 116
insertion operator 121
instance 524
int 85
I/O classes 864
ios_base
  flags 865
  flags() 866
  iostate 869
  openmode 869
  precision() 867
  seekdir 869
  setf(long) 867
  unsetf() 867
  width() 868
ios_base class 864
ios class 870
  bad() 871
  clear() 871
  eof() 871
  exceptions() 872
  fail() 871
  fill() 870
  good() 871
  rdbuf() 872
  rdstate() 870
  setstate() 872
  tie() 872
iostream class 894
istream class 874
  gcount() 877
  get() 874
  getline() 877
  ignore() 881
  operator >> 874
  peek() 879
  putback() 879
  read() 877
  readsome() 878
  seekg() 882
  tellg() 882
  unget() 879
istringstream class 900
  str() 900
  swap() 901
iteration 5, 14, 180
iterator-based functions 739
  advance() 740
  begin() 739
  distance() 739
  end() 740
  next() 740
  prev() 740
iterators 669

J

Julian Day Number 424
jump statements 191

L

label 193
lambdas 948
  and function pointers 949
  as a function parameter 957
  capture list 948
  capturing an expression 954
  creating 948
  executing 949
  in a class method 959
  local variables 952
  modifying external variables 956
  mutable 948
  parameter list 948
  parameters 950
  returning a value 950
libraries 69
lifetime 399
linker 69
linker errors 632
literal 84
literals 93

literal types 93
local variables 340
logic errors 632
long 85
long double 86
long int 85
long long 85
long long int 85
loop counting 182, 185
looping 14, 180
lvalues 109, 931

M

main() 414
  parameters 415
  returning a value 414
  using a batch file 416
maths functions 452
  abs() 457
  acos() 452
  acosh() 454
  asin() 453
  asinh() 454
  atan() 453
  atan2() 453
  atanh() 454
  cbrt() 456
  ceil() 456
  copysign() 457
  cos() 452
  cosh() 453
  exp() 454
  exp2() 454
  floor() 456
  fmax() 457
  fmin() 457
  fmod() 456
  frexp() 455
  hypot() 456
  log() 454
  log2() 455
  log10() 454
  pow() 456
  round() 457
  sin() 452
  sinh() 453
  sqrt() 456
  tan() 452
  tanh() 453
  trunc() 457
memory allocation 84
memory leakage 315
methods 530
Microsoft account 66
mini-spec 337
mips 5
modulo 139
Monte Carlo method 462
MoreTime class 786
move() 938
move and smart pointers 940
move assignment operator 937
move copy constructor 937
move semantics 930
Multi-dimensional array parameters 363
multi-dimensional vectors 697
multiple assignment 134
multiple inheritance 808
multiple throws 644
multi-way selection 9
mutable 109
mutually exclusive 156
mutually exclusive conditions 10

N

named constant 96
namespace 128, 619
naming variables 88
NaN 58
narrowing conversion 105
nested IF statements 12
nested loops 195
nested records 256
new 311

nodiscard 373
NOT 22
null 95, 294
nullptr 294
number systems 44
  binary fractions to decimal 49
  binary to decimal 47
  binary to hexadecimal 50
  decimal fractions to binary 47
  decimal to binary 45
  decimal to hexadecimal 49
  fixed point notation 56
  floating-point storage 57
  floating point values in binary 56
  hexadecimal 49
  hexadecimal to binary 51
  hexadecimal to decimal 50
  negative numbers in binary 54
    one's Complement 54
    two's Complement 55
  octal 52
  scientific notation 56
    exponent 56
    normalised significand 57
    significand 56
  storing numbers 52
    least-significant byte 53
    most-significant byte 53
nybble 44

O

object pointers 568
objects 524
octal 94
ofstream class 884
  close() 886
  is_open() 885
  open () 885
operations 525
operator overloading 404
operator precedence 135
operator priority 99
optional 924
  emplace() 926
  has_value() 925
  make_optional() 926
  relational operators 926
  reset() 926
  swap() 926
  value() 925
  value_or() 925
OR 21
ostream class 883
  flush() 883
  operator<< 883
  put() 883
  seekp() 884
  tellp() 884
  write() 883
ostringstream class 898
  str() 899
outer block 151
outer loop 196
out parameters 349
output 26, 120
output area 72
overflow 269
overloading -- 406
overloading ++ 406
overloading << and >> 553
overloading constructors 535
overloading friends 798
overloading methods 533
overloading operators 542
overloading restrictions 402
overloading the assignment operator 563
override 827

P

pair class
  get() 280
  make_pair() 279
  swap() 280

pair data type 278
parameters 337
parent class 785
parentheses 100
pass-by-pointer 360
pass-by-reference 356
pass-by-value 343
pointer parameters 360
pointers
  and arrays 301
  and auto 295
  and dynamic allocation 311
  arithmetic 310
  dangling 314
  dereferencing 297
  double dereference 301
  memory leakage 315
  to chars 308
  to constants 299
  to objects 824
  to pointers 300
  to strings 308
  to structs 306
  to unions 306
  using 295
  void 316
pointers to functions 410
pointer variables 293
post-conditions 348
precompiler 68
pre-conditions 347
preprocessor 68
printf() 125
programming language 4
project 70
project properties 73
prompt 78, 117
protected properties 803

R

radix 44
random number functions 462
  rand() 462
  srand() 462
range 214
range-based for and records 255
range-based for loop 214, 237
raw pointers 292
real numbers 84
real values 84
record 250
recursive functions 408
reference parameters 354
reference to constant 329
reference to pointers 330
reference variables 327
  range-based for loops 328
reinterpret_cast 891
relational operator 146
remainder operator 99
REPEAT.. UNTIL 17
rethrowing exceptions 648
return 348
returning an array 364
returning a pair 353
returning a pointer 365, 368
returning a reference 365
returning a struct 352
returning a tuple 353
routines 336
runtime errors 632
rvalue reference 934
rvalues 109, 931

S

saving objects 896
scanf_s() 118
scientific notation 119
scope 151, 399
scoped enumerations 261
scope resolution 400
scope resolution operator 129
searching 734
selection 5, 6

sequence 5
setbase() 123
Set container 658
set_terminate() 649
setw() 122
shift left operator 268
shift right operator 269
short 85
short int 85
signed char 87
signed wchar_t 87
sin() 452
single quote separator 94
size() 212
sizeof 134
smart pointers 320
  and functions 370
  get() 322, 324
  make_unique<>() 321
  release() 322
  reset() 321, 324
  shared_ptr 323
  swap() 322, 324
  to memory blocks 324
  unique_ptr 320
  use_count() 324
Snakes and Ladders 6
software 4
solution 70
Solution Explorer 72
solving problems 2
sorting 734
stack 84
standard functions 452
start up 69
static attributes 807
static_cast<> 107
static methods 550
static properties 548
static variables 397, 551
stepwise refinement 28
STL container classes 669
strcmp() 234
strcpy() 234
stream pointer 496
string class 750
  [ ] 758
  and iterator functions 773
  append() 767
  append-assignment operator 754
  assign() 770
  assignment operator 753
  at() 758
  back() 759
  begin() 759
  capacity() 772
  cbegin() 759
  cend() 759
  clear() 769
  compare() 755
  concatenation operator 753
  constants 751
  crbegin() 759
  crend() 759
  c_str() 770
  data() 770
  empty() 753
  end() 759
  erase() 769
  find() 760
  find_first_not_of() 764
  find_first_of() 762
  find_last_not_of() 764
  find_last_of() 764
  front() 758
  getline() 751
  insert() 768
  length() 752
  max_size() 773
  pop_back() 771
  push_back() 771
  rbegin() 759
  relational operators 754
  rend() 759

replace() 765
reserve() 772
resize() 773
rfind() 762
shrink_to_fit() 772
size() 752
stod() 774
stof() 774
stoi() 774
stol() 774
stold() 774
stoll() 774
stoul() 774
stoull() 774
substr() 757
swap() 767
to_string() 774
string conversion functions 480
strings 94
stringstream class 901
str() 901
swap() 902
string_view class 775
struct 250
struct parameters 345
structured binding 285
Student class 559
subclass 785
subroutines 336
superclass 785
support material
Chapter 1
algorithm constructor 37
algorithm tracer 38
reviewer 39
Chapter 2
Integer Number Converter 61
reviewer 61
switch 164
switch initialiser 167
switch limitations 167
syntax diagram 90
syntax errors 632
system functions 464
exit() 464
getenv() 464
system() 465

T

table 252
tan() 452
template classes 663
terminate() 649
test driver 171
test drivers 381
testing
black box 376
compound conditions 155
conditional assignment 164
do...while 185
drivers 381
equivalence class partitioning 377
for 189
functions 376
if 152
if...else 159
switch 170
while 183
testing functions 376
text files 495
this pointer 545
throwing objects 642
Time class 569
time functions 457
asctime() 461
clock() 458
CLOCKS_PER_SEC 458
clock_t 458
ctime() 459
difftime() 461
gmtime() 461
localtime() 461
mktime() 460

time() 459
time_t 458
tm 458
token 78
totals 26
trace table 32
truthtable 21
try block 636
tuple class
get() 281
make_tuple() 281
swap() 282
tie() 282
tuple_cat() 282
tuple data type 278
type aliases 277
type conversion 104
typedef 277
type inference 90

U

unction instantation 428
unhandled exceptions 649
Unicode 87
uniform initialisation 89
union 274
universal handler 644
unscoped enumerations 261
unsigned 86
unsigned char 87
unsigned wchar_t 87
unusual assignments 102
USDate class 799
user-defined functions 336, 394
user-defined type 250
using 277, 330, 804
using a static library 614
using char pointers 562
UTF 87
UTF-16 87

V

value parameters 341
variables 84
variable scope 149
variadic functions 432
variadic templates 435
variant 917
emplace() 918
get() 919
get_if() 919
holds_alternative() 920
index() 918
relational operators 919
swap() 918, 921
visit() 920
vector class
and fold expressions 695
as function parameters 694
assign() 692
assignment operator 689
at() 684
back() 685
begin() 690
capacity() 687
cbegin() 690
cend() 690
clear() 686
crbegin() 691
crend() 691
dangers 695
data() 685
emplace() 692
emplace_back() 693
empty() 685
end() 690
erase() 693
front() 685
insert() 691
max_size() 686
pop_back() 689
push_back() 689
rbegin() 690

rend() 690
reserve() 688
resize() 686
shrink_to_fit() 688
size() 685
swap() 685
vector container class 683
virtual classes 831
virtual methods 826
void pointers 316

W

wchar_t 87, 235
wcin 120
while 180
WHILE.. ENDWHILE 18
white box testing 379
whitespace 78
widening conversion 105
word 266
wstring class 776

Z

Zeller's Congruence 138