

THE EXPERT'S VOICE® IN C++

Beginning C++

Ivor Horton

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.



Contents at a Glance

About the Author	xxiii
About the Technical Reviewer	xxv
Introduction	xxvii
■ Chapter 1: Basic Ideas	1
■ Chapter 2: Introducing Fundamental Types of Data	23
■ Chapter 3: Working with Fundamental Data Types	55
■ Chapter 4: Making Decisions	79
■ Chapter 5: Arrays and Loops.....	105
■ Chapter 6: Pointers and References.....	151
■ Chapter 7: Working with Strings.....	185
■ Chapter 8: Defining Functions.....	213
■ Chapter 9: Lambda Expressions.....	271
■ Chapter 10: Program Files and Preprocessing Directives.....	287
■ Chapter 11: Defining Your Own Data Types	315
■ Chapter 12: Operator Overloading.....	365
■ Chapter 13: Inheritance	399
■ Chapter 14: Polymorphism.....	429

■ **Chapter 15: Runtime Errors and Exceptions 463**

■ **Chapter 16: Class Templates 495**

■ **Chapter 17: File Input and Output 533**

Index.....593

Introduction

Welcome to *Beginning C++*. This is a revised and updated version of my previous book, *Beginning ANSI C++*. The C++ language has been extended and improved considerably since the previous book, so much so that it was no longer possible to squeeze detailed explanations of all of C++ in a single book. This tutorial will teach enough of the essential C++ language and Standard Library features to enable you to write your own C++ applications. With the knowledge from this book you should have no difficulty in extending the depth and scope of your C++ expertise. C++ is much more accessible than many people assume. I have assumed no prior programming knowledge. If you are keen to learn and have an aptitude for thinking logically, getting a grip on C++ will be easier than you might imagine. By developing C++ skills, you'll be learning a language that is already used by millions, and that provides the capability for application development in just about any context.

The C++ language in this book corresponds to the latest ISO standard, commonly referred to as C++ 14. C++ 14 is a minor extension over the previous standard, C++ 11, so there is very little in the book that is C++ 14 specific. All the examples in the book can be compiled and executed using C++ 11-conforming compilers that are available now.

Using the Book

To learn C++ with this book, you'll need a compiler that conforms reasonably well to the C++ 11 standard and a text editor suitable for working with program code. There are several compilers available currently that are reasonably C++ 11 compliant, some of which are free.

The GCC compiler that is produced by the GNU Project has comprehensive support for C++ 11 and it is open source and free to download. Installing GCC and putting it together with a suitable editor can be a little tricky if you are new to this kind of thing. An easy way to install GCC along with a suitable editor is to download Code::Blocks from <http://www.codeblocks.org>. Code::Blocks is a free IDE for Linux, Apple Mac OS X, and Microsoft Windows. It supports program development using several compilers including compilers for GCC, Clang, and open Watcom. This implies you get support for C, C++, and Fortran.

Another possibility is to use Microsoft Visual C++ that runs under Microsoft Windows. It is not fully compliant with C++ 11, but it's getting there. The free version is available as Microsoft Visual Studio 2013 Express and at the time of writing this will compile most of the examples, and should compile them all eventually. You can download it from <http://www.microsoft.com/en-us/download/details.aspx?id=43733>. While the Microsoft Visual C++ compiler is more limited than GCC, in terms of the extent to which C++ 11 is supported, you get a professional editor and support for other languages such as C# and Basic. Of course, you can always install both! There are other compilers that support C++ 11, which you can find with a quick online search.

I've organized the material in this book to be read sequentially, so you should start at the beginning and keep going until you reach the end. However, no one ever learned programming by just reading a book. You'll only learn how to program in C++ by writing code, so make sure you key in all the examples—don't just copy them from the download files—and compile and execute the code that you've keyed in. This might seem tedious at times, but it's surprising how much just typing in C++ statements will help your understanding, especially when you may feel you're struggling with some of the ideas. If an example doesn't work, resist the temptation to go straight back to the book to see why. Try to figure out from your code what is wrong. This is good practice for what you'll have to do when you are developing C++ applications for real.

Making mistakes is a fundamental part of the learning process and the exercises should provide you with ample opportunity for that. It's a good idea to dream up a few exercises of your own. If you are not sure about how to do something, just have a go before looking it up. The more mistakes you make, the greater the insight you'll have into what can, and does, go wrong. Make sure you attempt all the exercises, and remember, don't look at the solutions until you're sure that you can't work it out yourself. Most of these exercises just involve a direct application of what's covered in a chapter—they're just practice, in other words—but some also require a bit of thought or maybe even inspiration.

I wish you every success with C++. Above all, enjoy it!

—Ivor Horton

CHAPTER 1



Basic Ideas

I'll sometimes have to make use of things in examples before I have explained them in detail. This chapter is intended to help when this occurs by giving you an overview of the major elements of C++ and how they hang together. I'll also explain a few concepts relating to the representation of numbers and characters in your computer. In this chapter you'll learn:

- What is meant by Modern C++
- The elements of a C++ program
- How to document your program code
- How your C++ code becomes an executable program
- How object-oriented programming differs from procedural programming
- What binary, hexadecimal, and octal number systems are
- What Unicode is

Modern C++

Modern C++ is programming using of the features of the latest and greatest incarnation of C++. This is the C++ language defined by the C++ 11 standard, which is being modestly extended and improved by the latest standard, C++ 14. This book relates to C++ as defined by C++14.

There's no doubt that C++ is the most widely used and most powerful programming language in the world today. If you were just going to learn one programming language, C++ is the ideal choice. It is effective for developing applications across an enormous range of computing devices and environments: for personal computers, workstations, mainframe computers, tablets, and mobile phones. Just about any kind of program can be written in C++ from device drivers to operating systems, from payroll and administrative programs to games. C++ compilers are available widely too. There are up-to-date compilers that run on PCs, workstations, and mainframes, often with cross-compiling capabilities, where you can develop the code in one environment and compile it to execute in another.

C++ comes with a very extensive Standard Library. This is a huge collection of routines and definitions that provide functionality that is required by many programs. Examples are numerical calculations, string processing, sorting and searching, organizing and managing data, and input and output. The Standard Library is so vast that we will only scratch the surface of what is available in this book. It really needs several books to fully elaborate all the capability it provides. *Beginning STL* is a companion book that is a tutorial on using the Standard Template Library, which is the subset of the C++ Standard Library for managing and processing data in various ways.

Given the scope of the language and the extent of the library, it's not unusual for a beginner to find C++ somewhat daunting. It is too extensive to learn in its entirety from a single book. However, you don't need to learn all of C++ to be able to write substantial programs. You can approach the language step by step, in which case it really isn't difficult. An analogy might be learning to drive a car. You can certainly become a very competent and safe driver without necessarily having the expertise, knowledge, and experience to drive in the Indianapolis 500. With this book you can learn everything you need to program effectively in C++. By the time you reach the end, you'll be confidently writing your own applications. You'll also be well equipped to explore the full extent of C++ and its Standard Library.

C++ Program Concepts

There will be much more detail on everything I discuss in this section later in the book. I'll jump straight in with the complete, fully working, C++ program shown in Figure 1-1, which explains what the various bits of it are. I'll use the example as a base for discussing some more general aspects of C++.

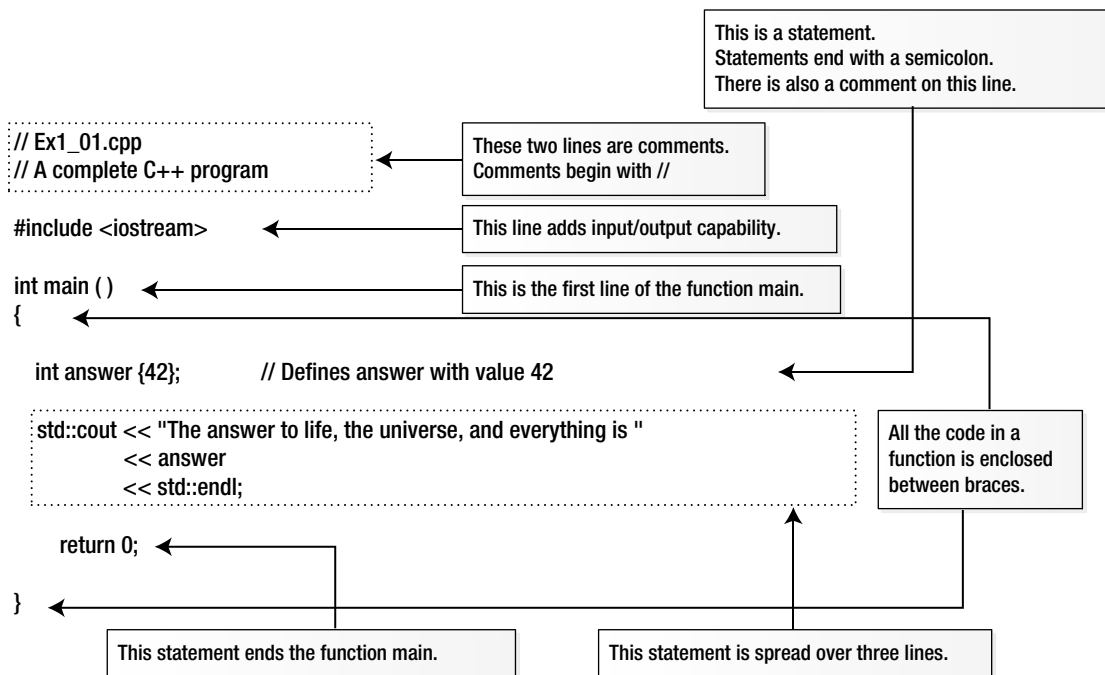


Figure 1-1. A complete C++ program

Comments and Whitespace

The first two lines in Figure 1-1 are *comments*. You *add comments* that document your program code to make it easier for someone else to understand how it works. The compiler ignores everything that follows two successive forward slashes on a line so this kind of comment can follow code on a line. The first line is a comment that indicates the name of the file containing this code. This file is in the code download for the book. I'll identify the file for each working example in the same way. The file extension, `.cpp`, indicates that this is a C++ *source file*. Other extensions such as `.cc` are also used to identify a C++ source file. All the executable code for a program will be in one or more source files.

There's another form of comment that you can use when you need to spread a comment over several lines. For example:

```
/* This comment is
   over two lines. */
```

Everything between `/*` and `*/` will be ignored by the compiler. You can embellish this sort of comment to make it stand out. For example:

```
/******
 * This comment is      *
 * over two lines.      *
 *****/
```

Whitespace is any sequence of spaces, tabs, newlines, form feed characters, and comments. Whitespace is generally ignored by the compiler, except when it is necessary for syntactic reasons to distinguish one element from another.

Preprocessing Directives and Header Files

The third line in Figure 1-1 is a *preprocessing directive*. Preprocessing directives cause the source code to be modified in some way before it is compiled to executable form. This preprocessing directive adds the contents of the standard library header file with the name `iostream` to this source file, `Ex1_01.cpp`. The header file contents are inserted in place of the `#include` directive.

Header files, which are sometimes referred to just as headers, contain definitions to be used in a source file. `iostream` contains definitions that are needed to perform input from the keyboard and text output to the screen using Standard Library routines. In particular, it defines `std::cout` and `std::endl` among many other things. You'll be including the contents of one or more standard library header files into every program and you'll also be creating and using your own header files that contain definitions that you construct later in the book. If the preprocessing directive to include the `iostream` header was omitted from `Ex1_01.cpp`, the source file wouldn't compile because the compiler would not know what `std::cout` or `std::endl` are. The contents of header files are included into a source file before it is compiled.

■ **Tip** Note that there are no spaces between the angle brackets and the standard header file name. With some compilers, spaces are significant between the angle brackets, `<` and `>`; if you insert spaces here, the program may not compile.

Functions

Every C++ program consists of at least one and usually many more *functions*. A function is a named block of code that carries out a well-defined operation such as “read the input data” or “calculate the average value” or “output the results.” You execute or *call* a function in a program using its name. All the executable code in a program appears within functions. There must be one function with the name `main`, and execution always starts automatically with this function. The `main()` function usually calls other functions, which in turn can call other functions, and so on. Functions provide several important advantages:

- A program that is broken down into discrete functions is easier to develop and test.
- You can reuse a function in several different places in a program, which makes the program smaller than if you coded the operation in each place that it is needed.

- You can often reuse a function in many different programs, thus saving time and effort.
- Large programs are typically developed by a team of programmers. Each team member is responsible for programming a set of functions that are a well-defined subset of the whole program. Without a functional structure, this would be impractical.

The program in Figure 1-1 consists of just the function `main()`. The first line of the function is:

```
int main()
```

This is called the *function header*, which identifies the function. Here, `int` is a type name that defines the type of value that the `main()` function returns when it finishes execution - an integer. In general, the parentheses following a name in a function definition enclose the specification for information to be passed to the function when you call it. There's nothing between the parentheses in this instance but there could be. You'll learn how you specify the type of information to be passed to a function when it is executed in Chapter 5. I'll always put parentheses after a function name in the text to distinguish it from other things that are code. The executable code for a function is always enclosed between braces and the opening brace follows the function header.

Statements

A statement is a basic unit in a C++ program. A statement always ends with a semicolon and it's the semicolon that marks the end of a statement, not the end of the line. A statement defines something, such as a computation, or an action that is to be performed. Everything a program does is specified by statements. Statements are executed in sequence until there is a statement that causes the sequence to be altered. You'll learn about statements that can change the execution sequence in Chapter 4. There are three statements in `main()` in Figure 1-1. The first defines a variable, which is a named bit of memory for storing data of some kind. In this case the variable has the name `answer` and can store integer values:

```
int answer {42}; // Defines answer with the value 42
```

The type, `int`, appears first, preceding the name. This specifies the kind of data that can be stored - integers. Note the space between `int` and `answer`. One or more whitespace characters is essential here to separate the type name from the variable name; without the space the compiler would see the name `intanswer`, which it would not understand. An initial value for `answer` appears between the braces following the variable name so it starts out storing 42. There's a space between `answer` and `{42}` but it's not essential. A brace cannot be part of a name so the compiler can distinguish the name from the initial value specification in any event. However, you should use whitespace in a consistent fashion to make your code more readable. There's a somewhat superfluous comment at the end of the first statement explaining what I just described but it does demonstrate that you can add a comments to a statement. The whitespace preceding the `//` is also not mandatory but it is desirable.

You can enclose several statements between a pair of curly braces, `{ }`, in which case they're referred to as a *statement block*. The body of a function is an example of a block, as you saw in Figure 1-1 where the statements in `main()` function appear between curly braces. A statement block is also referred to as a *compound statement* because in most circumstances it can be considered as a single statement, as you'll see when we look at decision-making capabilities in Chapter 4. Wherever you can put a single statement, you can equally well put a block of statements between braces. As a consequence, blocks can be placed inside other blocks—this concept is called *nesting*. Blocks can be nested, one within another, to any depth.

Data Input and Output

Input and output are performed using *streams* in C++. To output something, you write it to an output stream, and to input data you read it from an input stream. A *stream* is an abstract representation of a source of data, or a data sink. When your program executes, each stream is tied to a specific device that is the source of data in the case of an input stream and the destination for data in the case of an output stream. The advantage of having an abstract representation of a source or sink for data is that the programming is then the same regardless of the device the stream represents. You can read a disk file in essentially the same way as you read from the keyboard. The standard output and input streams in C++ are called `cout` and `cin` respectively and by default they correspond to your computer's screen and keyboard. You'll be reading input from `cin` in Chapter 2.

The next statement in `main()` in Figure 1-1 outputs text to the screen:

```
std::cout << "The answer to life, the universe, and everything is "
          << answer
          << std::endl;
```

The statement is spread over three lines, just to show that it's possible. The names `cout` and `endl` are defined in the `iostream` header file. I'll explain about the `std::` prefix a little later in this chapter. `<<` is the insertion operator that transfers data to a stream. In Chapter 2 you'll meet the extraction operator, `>>`, that reads data from a stream. Whatever appears to the right of each `<<` is transferred to `cout`. Writing `endl` to `std::cout` causes a new line to be written to the stream and the output buffer to be flushed. Flushing the output buffer ensures that the output appears immediately. The statement will produce the output:

```
The answer to life, the universe, and everything is 42
```

You can add comments to each line of a statement. For example:

```
std::cout << "The answer to life, the universe, and everything is " // This statement
          << answer                                              // occupies
          << std::endl;                                           // three lines
```

You don't have to align the double slashes but it's common to do so because it looks tidier and makes the code easier to read.

return Statements

The last statement in `main()` is a return statement. A return statement ends a function and returns control to where the function was called. In this case it ends the function and returns control to the operating system. A return statement may or may not return a value. This particular return statement returns 0 to the operating system. Returning 0 to the operating system indicates that the program ended normally. You can return non-zero values such as 1, 2, etc. to indicate different abnormal end conditions. The return statement in `Ex1_01.cpp` is optional, so you could omit it. This is because if execution runs past the last statement in `main()`, it is equivalent to executing `return 0`.

Namespaces

A large project will involve several programmers working concurrently. This potentially creates a problem with names. The same name might be used by different programmers for different things, which could at least cause some confusion and may cause things to go wrong. The Standard Library defines a lot of names, more than you can possibly remember. Accidental use of Standard Library names could also cause problems. *Namespaces* are designed to overcome this difficulty.

A *namespace* is a sort of family name that prefixes all the names declared within the namespace. The names in the standard library are all defined within a namespace that has the name `std`. `cout` and `endl` are names from the standard library so the full names are `std::cout` and `std::endl`. Those two colons together, `::`, have a very fancy title: *the scope resolution operator*. I'll have more to say about it later. Here, it serves to separate the namespace name, `std`, from the names in the Standard Library such as `cout` and `endl`. Almost all names from the Standard Library are prefixed with `std`.

The code for a namespace looks like this:

```
namespace ih_space {

    // All names declared in here need to be prefixed
    // with ih_space when they are reference from outside.
    // For example, a min() function defined in here
    // would be referred to outside this namespace as ih_space::min()

}
```

Everything between the braces is within the `ih_space` namespace.

■ **Caution** The `main()` function must not be defined within a namespace. Things that are not defined in a namespace exist in the *global namespace*, which has no name.

Names and Keywords

`Ex1_01.cpp` contains a definition for a variable with the name `answer` and it uses the names `cout` and `endl` that are defined in the `iostream` Standard Library header. Lots of things need names in a program and there are precise rules for defining names:

- A name can be any sequence of upper or lowercase letters A to Z or a to z, the digits 0 to 9 and the underscore character, `_`.
- A name must begin with either a letter or an underscore.
- Names are case sensitive.

Although it's legal, it's better not to choose names that begin with an underscore; they may clash with names from the C++ Standard Library because it defines names in this way extensively. The C++ standard allows names to be of any length, but typically a particular compiler will impose some sort of limit. However, this is normally sufficiently large that it doesn't represent a serious constraint. Most of the time you won't need to use names of more than 12 to 15 characters.

Here are some valid C++ names:

```
toe_count  shoeSize  Box    democrat  Democrat  number1  x2    y2    pValue  out_of_range
```

Uppercase and lowercase are differentiated so `democrat` is not the same name as `Democrat`. You can see a couple of examples of conventions for writing names that consists of two or more words; you can capitalize the second and subsequent words or just separate them with underscores.

Keywords are reserved words that have a specific meaning in C++ so you must not use them for other purposes. `class`, `double`, `throw`, and `catch` are examples of keywords.

Classes and Objects

A *class* is a block of code that defines a data type. A class has a name that is the name for the type. An item of data of a class type is referred to as an *object*. You use the class type name when you create variables that can store objects of your data type. Being able to define your own data types enables you to specify a solution to a problem in terms of the problem. If you were writing a program processing information about students for example, you could define a *Student* type. Your *Student* type could incorporate all the characteristic of a student - such as age, gender, or school record - that was required by the program.

Templates

You sometimes need several similar classes or functions in a program where the code only differs in the kind of data that is processed. A *template* is a recipe that you create to be used by the compiler to generate code automatically for a class or function customized for particular type or types. The compiler uses a *class template* to generate one or more of a family of classes. It uses a *function template* to generate functions. Each template has a name that you use when you want the compiler to create an instance of it. The Standard Library uses templates extensively.

Program Files

C++ code is stored in two kinds of files. *Source files* contain functions and thus all the executable code in a program. The names of source files usually have the extension `.cpp`, although other extensions such as `.cc` are also used. *Header files* contain *definitions* for things such as classes and templates that are used by the executable code in a `.cpp` file. The names of header files usually have the extension `.h` although other extensions such as `.hpp` are also used. Of course, a real-world program will typically include other kinds of files that contain stuff that has nothing to do with C++, such as resources that define the appearance of a graphical user interface (GUI) for example.

Standard Libraries

If you had to create everything from scratch every time you wrote a program, it would be tedious indeed. The same functionality is required in many programs—reading data from the keyboard for example, or calculating a square root, or sorting data records into a particular sequence. C++ comes with a large amount of prewritten code that provides facilities such as these, so you don't have to write the code yourself. All this standard code is defined in the *Standard Library*. There is a subset of the standard library that is called the *Standard Template Library* (STL). The STL contains a large number of class templates for creating types for organizing and managing data. It also contains many function templates for operations such as sorting and searching collections of data and for numerical processing. You'll learn about a few features of the STL in this book but a complete discussion of it requires a whole book in its own right. Beginning STL is a follow-on to this book that does exactly that.

Code Presentation Style

The way in which you arrange your code can have a significant effect on how easy it is to understand. There are two basic aspects to this. First, you can use tabs and/or spaces to indent program statements in a manner that provides visual cues to their logic, and you can arrange matching braces that define program blocks in a consistent way so that the relationships between the blocks are apparent. Second, you can spread a single statement over two or more lines when that will improve the readability of your program. A particular convention for arranging matching braces and indenting statements is a presentation style.

There are many different presentation styles for code. The following table shows three of many possible options for how a code sample could be arranged:

Style 1	Style 2	Style 3
<pre> namespace mine { bool has_factor(int x, int y) { int f{ hcf(x, y) }; if (f > 1) { return true; } else { return false; } } } </pre>	<pre> namespace mine{ bool has_factor(int x, int y) { int f{ hcf(x, y) }; if (f > 1) { return true; } else { return false; } } } </pre>	<pre> namespace mine{ bool has_factor(int x, int y) { int f{ hcf(x, y) }; if (f > 1){ return true; } else{ return false; } } } </pre>

I will use Style 1 for examples in the book.

Creating an Executable

Creating an executable module from your C++ source code is basically a two-step process. In the first step, your *compiler* processes each .cpp file to produce an *object file* that contains the machine code equivalent of the source file. In the second step, the *linker* combines the object files for a program into a file containing the complete executable program. Within this process, the linker will integrate any Standard Library functions that you use.

Figure 1-2 shows three source files being compiled to produce three corresponding object files. The filename extension that's used to identify object files varies between different machine environments, so it isn't shown here. The source files that make up your program may be compiled independently in separate compiler runs, or most compilers will allow you to compile them in a single run. Either way, the compiler treats each source file as a separate entity and produces one object file for each .cpp file. The link step then combines the object files for a program, along with any library functions that are necessary, into a single executable file.

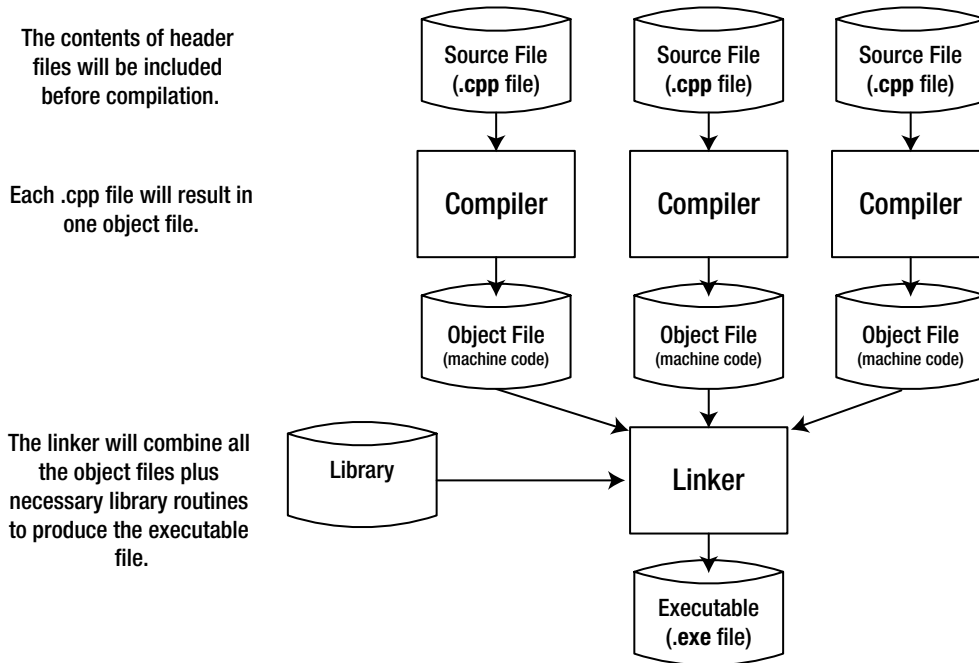


Figure 1-2. *The compile and link process*

In practice, compilation is an iterative process, because you're almost certain to have made typographical and other errors in the code. Once you've eliminated these from each source file, you can progress to the link step, where you may find that yet more errors surface. Even when the link step produces an executable module, your program may still contain logical errors; that is, it doesn't produce the results you expect. To fix these, you must go back and modify the source code and try to compile it once more. You continue this process until your program works as you think it should. As soon as you declare to the world at large that your program works, someone will discover a number of obvious errors that you should have found. It hasn't been proven beyond doubt so far as I know, but it's widely believed that any program larger than a given size will always contain errors. It's best not to dwell on this thought when flying.

Representing Numbers

Numbers are represented in a variety of ways in a C++ program and you need to have an understanding of the possibilities. If you are comfortable with binary, hexadecimal, and floating-point number representation you can safely skip this bit.

Binary Numbers

First, let's consider exactly what a common, everyday decimal number, such as 324 or 911 means. Obviously, what you mean is "three hundred and twenty-four" or "nine hundred and eleven." These are shorthand ways of saying "three hundreds" plus "two tens" plus "four," and "nine hundred" plus "one ten" plus "one." Putting this more precisely, you really mean:

324 is $3 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$, which is $3 \times 10 \times 10 + 2 \times 10 + 4$

911 is $9 \times 10^2 + 1 \times 10^1 + 1 \times 10^0$, which is $9 \times 10 \times 10 + 1 \times 10 + 1$

This is called *decimal notation* because it's built around powers of 10. We also say that we are representing numbers to *base 10* here because each digit position is a power of 10. Representing numbers in this way is very handy for beings with ten fingers and/or ten toes, or indeed ten of any kind of appendage that can be used for counting. Your PC is rather less handy, being built mainly of switches that are either on or off. Your PC is OK for counting in twos, but not spectacular at counting in tens. I'm sure you're aware that this is why your computer represents numbers using base 2, rather than base 10. Representing numbers using base 2 is called the *binary system* of counting. Numbers in base 10 have digits that can be from 0 to 9. In general, for numbers in an arbitrary base, *n*, the digit in each position in a number can be from 0 to *n*-1. Thus binary digits can only be 0 or 1. A binary number such as 1101 breaks down like this:

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0, \text{ which is } 1 \times 2 \times 2 \times 2 + 1 \times 2 \times 2 + 0 \times 2 + 1$$

This is 13 in the decimal system. In Table 1-1, you can see the decimal equivalents of all the numbers you can represent using eight binary digits, more commonly known as bits.

Table 1-1. *Decimal Equivalents of 8-bit Binary Values*

Binary	Decimal	Binary	Decimal
0000 0000	0	1000 0000	128
0000 0001	1	1000 0001	129
0000 0010	2	1000 0010	130
...	...	...	...
0001 0000	16	1001 0000	144
0001 0001	17	1001 0001	145
...	...	...	...
0111 1100	124	1111 1100	252
0111 1101	125	1111 1101	253
0111 1110	126	1111 1110	254
0111 1111	127	1111 1111	255

Using the first seven bits, you can represent positive numbers from 0 to 127, which is a total of 128 different numbers. Using all eight bits, you get 256 or 2⁸ numbers. In general, if you have *n* bits available, you can represent 2^{*n*} integers, with positive values from 0 to 2^{*n*}-1.

Adding binary numbers inside your computer is a piece of cake, because the “carry” from adding corresponding digits can only be 0 or 1. This means that very simple circuitry can handle the process. Figure 1-3 shows how the addition of two 8-bit binary values would work.

Binary	Decimal
0001 1101	29
+ 0010 1011	+ 43
0100 1000	72
	
carries	

Figure 1-3. *Adding binary values*

The addition operation adds corresponding bits in the operands, starting with the rightmost. Figure 1-3 shows that there is a “carry” of 1 to the next bit position for each of the first six bit positions. This is because each digit can only 0 or 1. When you add 1+1 the result cannot be stored in the current bit position and is equivalent to adding 1 in the next bit position to the left.

Hexadecimal Numbers

When you are dealing with larger binary numbers, a small problem arises with writing them. Look at this:

```
1111 0101 1011 1001 1110 0001
```

Binary notation here starts to be more than a little cumbersome for practical use, particularly when you consider that this in decimal is only 16,103,905—a miserable eight decimal digits. You can sit more angels on the head of a pin than that! Clearly you need a more economical way of writing this, but decimal isn’t always appropriate. You might want to specify that the tenth and twenty-fourth bits from the right in a number are 1, for example. To figure out the decimal integer for this is hard work, and there’s a good chance you’ll get it wrong anyway. An easier solution is to use *hexadecimal notation*, in which the numbers are represented using base 16.

Arithmetic to base 16 is a much more convenient option, and it fits rather well with binary. Each hexadecimal digit can have values from 0 to 15 and the digits from 10 to 15 are represented by the letters A to F (or a to f), as shown in Table 1-2. Values from 0 to 15 happen to correspond nicely with the range of values that four binary digits can represent.

Table 1-2. *Hexadecimal Digits and their Values in Decimal and Binary*

Hexadecimal	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A or a	10	1010
B or b	11	1011
C or c	12	1100
D or d	13	1101
E or e	14	1110
F or f	15	1111

Because a hexadecimal digit corresponds to four binary digits, you can represent any binary number in hexadecimal simply by taking groups of four binary digits starting from the right, and writing the equivalent hexadecimal digit for each group. Look at the following binary number:

```
1111 0101 1011 1001 1110 0001
```

Taking each group of four bits and replacing it with the corresponding hexadecimal digit from the table produces:

```
F      5      B      9      E      1
```

You have six hexadecimal digits corresponding to the six groups of four binary digits. Just to prove that it all works out with no cheating, you can convert this number directly from hexadecimal to decimal by again using the analogy with the meaning of a decimal number. The value of this hexadecimal number therefore works out as follows.

F5B9E1 as a decimal value is given by

$$15 \times 16^5 + 5 \times 16^4 + 11 \times 16^3 + 9 \times 16^2 + 14 \times 16^1 + 1 \times 16^0$$

This turns out to be

$$15,728,640 + 327,680 + 45,056 + 2,304 + 224 + 1$$

Thankfully, this adds up to the same number you got when converting the equivalent binary number to a decimal value: 16,103,905. In C++, hexadecimal values are written with 0x or 0X as a prefix, so in code the value would be written as 0xF5B9E1. Obviously, this means that 99 is not at all the same as 0x99.

The other very handy coincidence with hexadecimal numbers is that modern computers store integers in words that are an even number of bytes, typically 2, 4, 8, or 16 bytes. A byte is 8 bits, which is exactly two hexadecimal digits so any binary integer word in memory always corresponds to an exact number of hexadecimal digits.

Negative Binary Numbers

There's another aspect to binary arithmetic that you need to understand: negative numbers. So far, I've assumed that everything is positive—the optimist's view—and so the glass is still half full. But you can't avoid the negative side of life—the pessimist's perspective—that the glass is already half empty. How is a negative number represented in a computer? Well, you have only binary digits at your disposal, so the solution has to be to use at least one of those to indicate whether the number is negative or positive.:

For numbers that can be negative (referred to as *signed numbers*), you must first decide on a fixed length (in other words, the number of binary digits) and then designate the leftmost binary digit as a sign bit. You have to fix the length to avoid any confusion about which bit is the sign bit.

As you know, your computer's memory consists of 8-bit bytes, so binary numbers are going to be stored in some multiple (usually a power of 2) of 8 bits. Thus, you can have numbers with 8 bits, 16 bits, 32 bits, or whatever. As long as you know what the length is in each case, you can find the sign bit—it's just the leftmost bit. If the sign bit is 0, the number is positive, and if it's 1, the number is negative.

This seems to solve the problem. Each number consists of a sign bit that is 0 for positive values and 1 for negative values, plus a given number of other bits that specify the absolute value of the number, the value without the sign in other words. Changing +6 to -6 then just involves flipping the sign bit from 0 to 1. Unfortunately, this representation carries a lot of overhead in terms of the complexity of the circuits that are needed to perform arithmetic. For this reason, most computers take a different approach. You can get an idea of how this approach works by considering how the computer would handle arithmetic with positive and negative values so that operations are as simple as possible.

Ideally, when two integers are added, you don't want the computer to be messing about, checking whether either or both of the numbers are negative. You just want to use simple "add" circuitry regardless of the signs of the operands. The add operation will combine corresponding binary digits to produce the appropriate bit as a result, with a carry to the next digit along where this is necessary. If you add -8 in binary to +12, you would really like to get the answer +4 using the same circuitry that would apply if you were adding +3 and +8.

If you try this with the simplistic solution, which is just to set the sign bit of the positive value to 1 to make it negative, and then perform the arithmetic with conventional carries, it doesn't quite work:

12 in binary is	0000 1100
-8 in binary (you suppose) is	1000 1000
If you now add these together, you get	1001 0100

This seems to be -20, which isn't what you wanted at all. It's definitely not +4, which you know is 0000 0100. "Ah," I hear you say, "you can't treat a sign just like another digit." But that is just what you *do* want to do.

You can see how the computer would *like* to represent -8 by subtracting +12 from +4:

+4 in binary is	0000 0100
+12 in binary is	0000 1100
Subtracting 12 from 4 you get	1111 1000

For each digit after the fourth from the right, you had to "borrow" 1 to do the subtraction, just as you would when performing decimal arithmetic. This result is supposed to be -8, and even though it doesn't look like it, that's exactly what it is. Just try adding it to +12 or +15 in binary, and you'll see that it works! Of course, if you want to produce -8 you can always subtract +8 from 0.

What exactly did you get when you subtracted 12 from 4 or +8 from 0? What you have here is called the *2's complement representation* of a negative binary number. You can produce the negative of any positive binary number by a simple procedure that you can perform in your head. At this point, I need to ask you to have a little faith because I'll avoid getting into explanations of why it works. I'll show you how you can create the 2's complement form of a negative number from a positive value, and you can prove to yourself that it does work. Let's return to the previous example, in which you need the 2's complement representation of -8.

You start with +8 in binary:

0000 1000

You "flip" each binary digit, changing 0s to 1s and vice versa:

1111 0111

This is called the *1's complement* form. If you add 1 to this, you'll get the 2's complement form:

1111 1000

This is exactly the same as the representation of -8 you got by subtracting +12 from +4. Just to make absolutely sure, let's try the original sum of adding -8 to +12:

+12 in binary is	0000 1100
Your version of -8 is	1111 1000
If you add these together, you get	0000 0100

The answer is 4—magic. It works! The "carry" propagates through all the leftmost 1s, setting them back to 0. One fell off the end, but you shouldn't worry about that—it's probably compensating for the one you borrowed from the end in the subtraction you did to get -8. In fact, what's happening is that you're implicitly assuming that the sign bit, 1 or 0, repeats forever to the left. Try a few examples of your own; you'll find it always works, automatically. The great thing about the 2's complement representation of negative numbers is that it makes arithmetic very easy (and fast) for your computer.

Octal Values

Octal integers :are numbers expressed with base 8. Digits in an octal value can only be from 0 to 7. Octal is used rarely these days. It was useful in the days when computer memory was measured in terms of 36-bit words because you could specify a 36-bit binary value by 12 octal digits. Those days are long gone so why am I introducing it? The potential confusion it can cause is the answer. You can still write octal constants in C++. Octal values are written with a leading zero, so while 76 is a decimal value, 076 is an octal value that corresponds to 64 in decimal. So, here’s a golden rule:

■ **Note** *Never* write decimal integers with a leading zero. You’ll get either a value different from what you intended, or an error message from the compiler.

Big-Endian and Little-Endian Systems

Integers are stored in memory as binary values in a contiguous sequence of bytes, commonly groups of 2, 4, 8, or 16 bytes. The question of the sequence in which the bytes appear can be very important—it’s one of those things that doesn’t matter until it matters, and then it *really* matters.

Let’s consider the decimal value 262,657 stored as a 4-byte binary value. I chose this value because in binary it happens to be

0000 0000 0000 0100 0000 0010 0000 0001

so each byte has a pattern of bits that is easily distinguished from the others. If you’re using a PC with an Intel processor, the number will be stored as follows:

Byte address:	00	01	02	03
Data bits:	0000 0001	0000 0010	0000 0100	0000 0000

As you can see, the most significant eight bits of the value—the one that’s all 0s—are stored in the byte with the highest address (last, in other words), and the least significant eight bits are stored in the byte with the lowest address, which is the leftmost byte. This arrangement is described as *little-endian*.

If you’re using a machine based on a Motorola processor, the same data is likely to be arranged in memory like this:

Byte address:	00	01	02	03
Data bits:	0000 0000	0000 0100	0000 0010	0000 0001

Now the bytes are in reverse sequence with the most significant eight bits stored in the leftmost byte, which is the one with the lowest address. This arrangement is described as *big-endian*. Some recent processors such as SPARC and Power-PC processors are *bi-endian*, which means that the byte order for data is switchable between big-endian and little endian.

■ **Note** Regardless of whether the byte order is big-endian or little-endian, the bits within each byte are arranged with the most significant bit on the left and the least significant bit on the right.

This is all very interesting, you may say, but when does it matter? Most of the time, it doesn't. More often than not, you can happily write a program without knowing whether the computer on which the code will execute is big-endian or little-endian. It does matter, however, when you're processing binary data that comes from another machine. You need to know the endian-ness. Binary data is written to a file or transmitted over a network as a sequence of bytes. It's up to you how you interpret it. If the source of the data is a machine with a different endian-ness from the machine on which your code is running, you must reverse the order of the bytes in each binary value. If you don't, you have garbage.

For those who collect curious background information, the terms "big-endian" and "little-endian" are drawn from the book *Gulliver's Travels* by Jonathan Swift. In the story, the emperor of Lilliput commanded all his subjects to always crack their eggs at the smaller end. This was a consequence of the emperor's son having cut his finger following the traditional approach of cracking his egg at the big end. Ordinary, law-abiding Lilliputian subjects who cracked their eggs at the smaller end were described as Little Endians. The Big Endians were a rebellious group of traditionalists in the Lilliputian kingdom who insisted on continuing to crack their eggs at the big end. Many were put to death as a result.

Floating-Point Numbers

We often have to deal with very large numbers—the number of protons in the universe, for example—which need around 79 decimal digits. Clearly there are lots of situations in which you'll need more than the ten decimal digits you get from a 4-byte binary number. Equally, there are lots of very small numbers, for example, the amount of time in minutes it takes the typical car salesperson to accept your generous offer on a 2001 Honda (and it's covered only 480,000 miles . . .). A mechanism for handling both these kinds of numbers is, as you may have guessed, *floating-point* numbers.

A floating-point representation of a number in decimal notation is a decimal value with two parts. One part is called the *mantissa*, which is greater than or equal to 0.9 and less than 1.0 and has a fixed number of digits. The other part is called the *exponent*. The value of the number is the mantissa multiplied by 10 to the power of the exponent. It's easier to demonstrate this than to describe it, so let's look at some examples. The number 365 in normal decimal notation could be written in floating-point form as follows:

0.3650000E03

The E stands for "exponent" and precedes the power of 10 that the 0.3650000 (the mantissa) part is multiplied by to get the required value. That is:

$$0.3650000 \times 10 \times 10 \times 10$$

This is clearly 365.

The mantissa here has seven decimal digits. The number of digits of precision in a floating-point number will depend on how much memory it is allocated. A *single precision* floating-point value occupying 4 bytes will typically provide approximately seven decimal digits accuracy. I say "approximately" because inside your computer these numbers are in binary floating-point form, and a binary fraction with 23 bits doesn't exactly correspond to a decimal fraction with seven decimal digits. A *double precision* floating-point value will typically correspond to around 15 decimal digits accuracy.

Now let's look at a small number:

0.3650000E-04

This is evaluated as $.365 \times 10^{-4}$, which is .0000365.

Suppose you have a large number such as 2,134,311,179. As a single precision floating-point number it looks like this:

0.2134311E10

It's not quite the same. You've lost three low-order digits and you've approximated your original value as 2,134,311,000. This is a small price to pay for being able to handle such a vast range of numbers, typically from 10^{-38} to 10^{+38} either positive or negative. They're called floating-point numbers for the fairly obvious reason that the decimal point "floats" and its position depends on the exponent value.

Aside from the fixed precision limitation in terms of accuracy, there's another aspect you may need to be conscious of. You need to take great care when adding or subtracting numbers of significantly different magnitudes. A simple example will demonstrate the problem. You can first consider adding .365E-3 to .365E+7. You can write this as a decimal sum:

$$0.000365 + 3,650,000.0$$

This produces this result:

$$3,650,000.000365$$

When converted to floating-point with seven digits of precision, this becomes:

$$0.3650000E+7$$

Adding .365E-3 to .365E+7 has had no effect whatsoever so you might as well not have bothered. The problem lies directly with the fact that you carry only six or seven digits precision. The digits of the larger number aren't affected by any of the digits of the smaller number because they're all further to the right. Funnily enough, you must also take care when the numbers are nearly equal. If you compute the difference between such numbers, you may end up with a result that has only one or two digits precision. It's quite easy in such circumstances to end up computing with numbers that are totally garbage. While floating-point numbers enable you to carry out calculations that would be impossible without them, you must always keep their limitations in mind if you want to be sure your results are valid. This means considering the range of values that you are likely to be working with and their relative values.

Representing Characters

Data inside your computer has no intrinsic meaning. Machine code instructions are just numbers, of course numbers are just numbers, and characters are just numbers. Each character is assigned a unique integer value called its *code* or *code point*. The value 42 can be the number of days in six weeks, the answer to life, the universe and everything, or it can be an asterisk character. It depends on how you choose to interpret it. You can write a single character in C++ between single quotes, such as 'a' or '?' or '*' and the compiler will generate the code value for these.

ASCII Codes

Way back in the 1960s, the American Standard Code for Information Interchange (ASCII) was defined for representing characters. This is a 7-bit code so there are 128 different code values. ASCII values 0 to 31 represent various non-printing control characters such as carriage return (code 15) and line feed (code 12). Code values 65 to 90 inclusive are the uppercase letters A to Z and 141 to 172 correspond to lowercase a to z. If you look at the binary values corresponding to the code values for letters, you'll see that the codes for lowercase and uppercase letters only differ in the sixth bit; lowercase letters have the sixth bit as 0, and uppercase letters have the sixth bit as 1. Other codes represent digits 0 to 9, punctuation and other characters. This is fine if you are American or British but if you are French or German you need things like accents and umlauts in text and these are not included in 7-bit ASCII.

To overcome the limitations imposed by a 7-bit code, extended versions of ASCII were defined with 8-bit codes. Values from 0 to 127 represent the same characters as 7-bit ASCII and values from 128 to 255 are variable. One variant of 8-bit ASCII that you have probably met is called Latin-1 which provides characters for most European languages and there are others for languages such as Russian. Of course, if you are Korean, Japanese, or Arabic, an 8-bit coding is totally inadequate. To overcome the limitations of extended ASCII the *Universal Character Set* (UCS) emerged in the 1990s, UCS is defined by the standard ISO 10646 and has codes with up to 32 bits. This provides the potential for over 2 billion unique code values.

UCS and Unicode

UCS defines a mapping between characters and integer code values, called *code points*. It is important to realize that a *code point* is not the same as an *encoding*. A code point is an integer; an encoding specifies a way of representing a given code point as a series of bytes or words. Code values of less than 256 are very popular and can be represented in one byte. It would be very inefficient to use four bytes to store code values that require just one byte, just because there are other codes that require several bytes. Encodings are ways of representing code points that allow them to be stored more efficiently.

Unicode is a standard that defines a set of characters and their code points identical to those in UCS; the code point values are from 0 to 0x10ffff. Unicode also defines several different encodings for these code points and includes additional mechanisms for dealing with such things as right-to-left languages such as Arabic. The range of code points is more than enough to accommodate the character sets for all the languages in the world, as well as many different sets of graphical characters such as mathematical symbols. The codes are divided into 17 code planes, each of which contains 65,536 code values. Code plane 0 contains hexadecimal code values from 0 to 0xffff, code plane 1 contains codes from 0x10000 to 0x1ffff, code plane 2 contains codes from 0x20000 to 0x2ffff, and so on up to code plane 17 that contains codes from 0x100000 to 0x10ffff. Character codes for most national languages are contained within code plane 0, which has code values from 0 to 0xffff. Consequently, strings in the majority of languages can be represented as a sequence of single 16-bit codes.

One aspect of Unicode that can be confusing is that it provides more than one *character encoding method*. The most commonly used encodings are referred to as UTF-8 and UTF-16, either of which can represent all the characters in the Unicode set. The difference between UTF-8 and UTF-16 is in how a given character code point is presented; the numerical code value for any given character is the same in either representation. Here's how these encodings represent characters:

- *UTF-8* represents a character as a variable length sequence of between 1 and 4 bytes. The ASCII character set appears in UTF-8 as single byte codes that have the same codes values as ASCII. Most web pages use UTF-8 to encode text. Code plane 0 is accommodated by one-byte and two-byte codes in UTF-8.
- *UTF-16* represents characters as one or two 16-bit values. UTF-16 includes UTF-8. Because a single 16-bit value accommodates all of code plane 0, UTF-16 covers most situations in programming for a multilingual context.

You have three integer types that store Unicode characters. These are types `wchar_t`, `char16_t`, and `char32_t`. You'll learn more about these in Chapter 2.

C++ Source Characters

You write C++ statements using a *basic source character set*. This is the set of characters that you're allowed to use explicitly in a C++ source file. The character set that you can use to define a name is a subset of this. Of course, the basic source character set in no way constrains the character data that you work with in your code. Your program can create strings consisting of characters outside this set in various ways, as you'll see. The basic source character set consists of the following characters:

- The letters *a* to *z* and *A* to *Z*
- The digits 0 to 9
- The control characters representing horizontal tab, vertical tab, form-feed, and newline
- The characters `_ { } [ ] # ( ) < > % ; . ? * + - / ^ & | ~ ! = , \ " '`

This is easy and straightforward. You have 96 characters that you can use, and it’s likely that these will accommodate your needs most of the time. Most of the time the basic source character set will be adequate, but occasionally you’ll need characters that aren’t in it. You can include Unicode characters in a name. You specify a Unicode character in the form of a hexadecimal representation of its code point, either as `\udddd` or `\Uddddddd`, where `d` is a hexadecimal digit. Note the lowercase `u` in the first case and the uppercase `U` in the second; either is acceptable. However, you must not specify any of the characters in the basic source character set in this way. Also, the characters in a name must not be control characters. Both character and string data can include Unicode characters.

Trigraph Sequences

You’re unlikely to see this in use very often, if ever, but the C++ standard allows you to specify certain characters as *trigraph sequences*. A trigraph sequence is a sequence of three characters that identifies another character. This was necessary way back in the dark ages of computing to accommodate characters that were required by the C language but were missing from 7-bit ASCII. Table 1-3 shows the characters that may be specified as trigraph sequences in C++.

Table 1-3. *Trigraph Sequence Characters*

Character	Trigraph Sequence
#	<u>??=</u>
[	<u>??(</u>
]	<u>??)</u>
\	<u>??/</u>
{	<u>??<</u>
}	<u>??></u>
^	<u>??'</u>
	<u>??!</u>
~	<u>??-</u>

These are still supported for compatibility reasons. The compiler will replace all trigraph sequences with their equivalent characters before any other processing of the source code. There are a few instances where you need to be aware of this, particularly if you are using a feature called *regular expressions*, which are not covered in this book. This is because sequences can occur in regular expressions that correspond to trigraph sequences when it is not intended. Most of the time you can forget about trigraph sequences, though it’s not impossible to specify them by accident.

Escape Sequences

When you want to use *character constants* such as a single character or a character string in a program, certain characters are problematic. Obviously, you can’t enter characters such as newline or tab directly as character constants, as they’ll just do what they’re supposed to do: go to a new line or tab to the next tab position in your source code file. You can enter these problem characters in character constants by means of an *escape sequence*. An escape sequence is an indirect way of specifying a character, and it always begins with a backslash. Table 1-4 shows the escape sequences that represent control characters.

Table 1-4. *Escape Sequences That Represent Control Characters*

Escape Sequence	Control Character
\n	Newline
\t	Horizontal tab
\v	Vertical tab
\b	Backspace
\r	Carriage return
\f	Form feed
\a	Alert/bell

There are some other characters that are a problem to represent directly. Clearly, the backslash character itself is difficult, because it signals the start of an escape sequence. The single and double quote characters that are used as delimiters as in the constant 'A' or the string "text" are also a problem. Table 1-5 shows the escape sequences for these.

Table 1-5. *Escape Sequences That Represent “Problem” Characters*

Escape Sequence	“Problem” Character
\\	Backslash
\'	Single quote
\"	Double quote
\?	Question mark

Because the backslash signals the start of an escape sequence, the only way to enter a backslash as a character constant is by using two successive backslashes (\\).

This program that uses escape sequences outputs a message to the screen. To see it, you’ll need to enter, compile, link, and execute the code:

```
// Ex1_02.cpp
// Using escape sequences
#include <iostream>

int main()
{
    std::cout << "\"Least \'said\' \\n\\t\\tsoonest \'mended\'\"" << std::endl;
}
```

When you manage to compile, link, and run this program, you should see the following output displayed:

```
"Least 'said' \
    soonest 'mended'."
```

The output is determined by what's between the outermost double quotes in the statement:

```
std::cout << "\"Least \'said\' \\n\\t\\tsoonest \'mended\'\\.\"" << std::endl;
```

In principle, *everything* between the outer double quotes in the preceding statement gets sent to `cout`. A string of characters between a pair of double quotes is called a *string literal*. The double quote characters are *delimiters* that identify the beginning and end of the string literal; they aren't part of the string. Each escape sequence in the string literal will be converted to the character it represents by the compiler, so the character will be sent to `cout`, not the escape sequence itself. A backslash in a string literal *always* indicates the start of an escape sequence, so the first character that's sent to `cout` is a double quote character.

Least follows by a space is output next. This is followed by a single quote character, then said, followed by another single quote. Next is a space, followed by the backslash specified by `\\`. Then a newline character corresponding to `\\n` is written to the stream so the cursor moves to the beginning of the next line. You then send two tab characters to `cout` with `\\t\\t`, so the cursor will be moved two tab positions to the right. The word soonest is output next followed by a space, then mended between single quotes. Finally a period is output followed by a double quote.

Procedural and Object-Oriented Programming

Historically, procedural programming is the way almost all programs have been written. To create a procedural programming solution to a problem, you focus on the process that your program must implement to solve the problem. A rough outline of what you do, once the requirements have been defined precisely, is as follows:

- You create a clear, high-level definition of the overall process that your program will implement.
- You segment the overall process into workable units of computation that are, as much as possible, self-contained. These will usually correspond to functions.
- You break down the logic and the work that each unit of computation is to do into a detailed sequence of actions. This is likely to be down to a level corresponding to programming language statements.
- You code the functions in terms of processing basic types of data: numerical data, single characters, and character strings.

Apart from the common requirement of starting out with a clear specification of what the problem is, the object-oriented approach to solving the same problem is quite different:

- From the problem specification, you determine what types of *objects* the problem is concerned with. For example, if your program deals with baseball players, you're likely to identify `BaseballPlayer` as one of the types of data your program will work with. If your program is an accounting package, you may well want to define objects of type `Account` and type `Transaction`. You also identify the set of *operations* that the program will need to carry out on each type of object. This will result in a set of application-specific data types that you will use in writing your program.
- You produce a detailed design for each of the new data types that your problem requires, including the operations that can be carried out with each object type.
- You express the logic of the program in terms of the new data types you've defined and the kinds of operations they allow.

The program code for an object-oriented solution to a problem will be completely unlike that for a procedural solution and almost certainly easier to understand. It will also be a lot easier to maintain. The amount of design time required for an object-oriented solution tends to be greater than for a procedural solution. However, the coding and testing phase of an object-oriented program tends to be shorter and less troublesome, so the overall development time is likely to be roughly the same in either case.

To get an inkling of what an object-oriented approach implies, suppose you're implementing a program that deals with boxes of various kinds. A feasible requirement of such a program would be to package several smaller boxes inside another, larger box. In a procedural program, you would need to store the length, width, and height of each box in a separate group of variables. The dimensions of a new box that could contain several other boxes would need to be calculated explicitly in terms of the dimensions of each of the contained boxes, according to whatever rules you had defined for packaging a set of boxes.

An object-oriented solution might involve first defining a `Box` data type. This would enable you to create variables that can reference objects of type `Box` and, of course, create `Box` objects. You could then define an operation that would add two `Box` objects together and produce a new `Box` object that could contain them. Using this operation, you could write statements like this:

```
bigBox = box1 + box2 + box3;
```

In this context the `+` operation means much more than simple addition. The `+` operator applied to numerical values will work exactly as before, but for `Box` objects it has a special meaning. Each of the variables in this statement is of type `Box`. The statement would create a new `Box` object big enough to contain `box1`, `box2`, and `box3`.

Being able to write statements like this is clearly much easier than having to deal with all the box dimensions separately, and the more complex the operations on boxes you take on, the greater the advantage is going to be. This is a trivial illustration, though, and there's a great deal more to the power of objects than that you can see here. The purpose of this discussion is just to give you an idea of how readily problems solved using an object-oriented approach can be understood. Object-oriented programming is essentially about solving problems in terms of the entities to which the problems relates rather than in terms of the entities that computers are happy with: numbers and characters.

Summary

This chapter's content has been broad-brush to give you a feel for some of the general concepts of C++. You'll encounter everything discussed in this chapter again, and in much more detail, in subsequent chapters. However, some of the basics that this chapter covered are as follows:

- A C++ program consists of one or more functions, one of which is called `main()`. Execution always starts with `main()`.
- The executable part of a function is made up of statements contained between braces.
- A pair of curly braces is used to enclose a statement block.
- A statement is terminated by a semicolon.
- Keywords are reserved words that have specific meanings in C++. No entity in your program can have a name that coincides with a keyword.
- A C++ program will be contained in one or more files. Source files contain the executable code and header files contains definitions used by the executable code.
- The source files that contain the code defining functions typically have the extension `.cpp`.
- Header files that contain definitions that are used by a source file typically have the extension `.h`.

- Preprocessor directives specify operations to be performed on the code in a file. All preprocessor directives execute before the code in a file is compiled.
- The contents of a header file is added to a source file by a `#include` preprocessor directive.
- The Standard Library provides an extensive range of capabilities that supports and extends the C++ language.
- Access to Standard Library functions and definitions is enabled through including Standard Library header files into a source file.
- Input and output is performed using streams and involve the use of the insertion and extraction operators, `<<` and `>>`. `std::cin` is a standard input stream that corresponds to the keyboard. `std::cout` is a standard output stream for writing text to the screen. Both are defined in the `iostream` Standard Library header.
- Object-oriented programming involves defining new data types that are specific to your problem. Once you've defined the data types that you need, a program can be written in terms of the new data types.
- Unicode defines unique integer code values that represents characters for virtually all of the languages in the world as well as many specialized character sets. Code values are referred to as code points. Unicode also defines how these code points may be encoded as byte sequences.

EXERCISES

Exercise 1-1. Create, compile, link, and execute a program that will display the text "Hello World" on your screen.

Exercise 1-2. Create and execute a program that outputs your name on one line and your age on the next line.

Exercise 1-3. The following program produces several compiler errors. Find these errors and correct them so the program can compile cleanly and run.

```
include <iostream>

Int main()
{
    std:cout << "Hello World" << std:endl
}
```

■ **Note** You'll find model answers to all exercises in this book on Apress website at www.apress.com/source-code/.



Introducing Fundamental Types of Data

In this chapter, I'll explain the fundamental data types that are built into C++. You'll need these in every program. All of the object-oriented capability is founded on these fundamental data types, because all the data types that you create are ultimately defined in terms of the basic numerical data your computer works with. By the end of the chapter, you'll be able to write a simple C++ program of the traditional form: input-process-output.

In this chapter, you'll learn about

- Data types in C++
- How you declare and initialize variables
- What literals are and how you define them
- Binary and hexadecimal integers
- How calculations work
- How you can fix the value of a variable
- How to create variables that store characters
- What the `auto` keyword does
- What lvalues and rvalues are

Variables, Data, and Data Types

A variable is a named piece of memory that you define. Each variable only stores data of a particular type. Every variable has a *type* that defines the kind of data it can store. Each fundamental type is identified by a unique type name that is a *keyword*. Keywords are reserved words in C++ that you must not use for anything else.

The compiler makes extensive checks to ensure that you use the right data type in any given context. It will also ensure that when you combine different types in an operation such as adding two values for example, they are either of the same type, or they can be made to be compatible by converting one value to the type of the other. The compiler detects and reports attempts to combine data of different types that are incompatible.

Numerical values fall into two broad categories: integers-whole numbers in other words, and floating-point values, which can be non-integral. There are several fundamental C++ types in each category, each of which can store a specific range of values. I'll start with integer types.

Defining Integer Variables

Here's a statement that defines an integer variable:

```
int apple_count;
```

This defines a variable of type `int` with the name `apple_count`. The variable will contain some arbitrary junk value. You can and should specify an initial value when you define the variable, like this:

```
int apple_count {15};                                // Number of apples
```

The initial value for `apple_count` appears between the braces following the name so it has the value 15. The braces enclosing the initial value is called an *initializer list*. You'll meet situations later in the book where an initializer list will have several values between the braces. You don't have to initialize variables when you define them but it's a good idea to do so. Ensuring variables start out with known values makes it easier to work out what is wrong when the code doesn't work as you expect.

Type `int` is typically 4 bytes, which can store integers from -2,147,483,648 to +2,147,483,647. This covers most situations, which is why `int` is the integer type that is used most frequently.

Here are definitions for three variables of type `int`:

```
int apple_count {15};                                // Number of apples
int orange_count {5};                                // Number of oranges
int total_fruit {apple_count + orange_count};         // Total number of fruit
```

The initial value for `total_fruit` is the sum of the values of two variables defined previously. This demonstrates that the initial value for a variable can be an expression. The statements that define the two variables in the expression for the initial value for `total_fruit` must appear earlier in the source file, otherwise the definition for `total_fruit` won't compile.

The initial value between the braces should be of the same type as the variable you are defining. If it isn't, the compiler will have to convert it to the required type. If the conversion is to a type with a more limited range of values, the conversion has the potential to lose information so the compiler won't convert the value but just flag it as an error. An example would be if you specified the initial value for an integer variable that is not an integer—1.5 for example. You might do this by accident when entering the value 15 for `apple_count`. A conversion to a type with a more limited range of values is called a *narrowing conversion*.

There are two other ways for initializing a variable. *Functional notation* looks like this:

```
int orange_count(5);
int total_fruit(apple_count + orange_count);
```

Alternatively you could write this:

```
int orange_count = 5;
int total_fruit = apple_count + orange_count;
```

While both of these possibilities are valid, I recommend that you adopt the initializer list form. This is the most recent syntax that was introduced in C++ 11 to standardize initialization. It is preferred because it enables you to initialize just about everything in the same way. There's one exception that I'll explain later in this chapter. I'll use initializer lists throughout all the examples in the book except in the instances where it is not appropriate.

You can define and initialize more than one variable of a given type in a single statement. For example:

```
int foot_count {2}, toe_count {10}, head_count {1};
```

While this is legal, most of the time it's better to define each variable in a separate statement. This makes the code more readable and you can explain the purpose of each of them in a comment.

You can write the value of any variable of a fundamental type to the standard output stream. Here's a program that does that:

```
// Ex2_01.cpp
// Writing values of variables to cout
#include <iostream>

int main()
{
    int apple_count {15};           // Number of apples
    int orange_count {5};           // Number of oranges
    int total_fruit {apple_count + orange_count}; // Total number of fruit

    std::cout << "The value of apple_count is " << apple_count << std::endl;
    std::cout << "The value of orange_count is " << orange_count << std::endl;
    std::cout << "The value of total_fruit is " << total_fruit << std::endl;
}
```

If you compile and execute this, you'll see that it outputs the values of the three variables following some text explaining what they are. The binary values are automatically converted to a character representation for output by the insertion operator, <<. This works for values of any of the fundamental types.

Signed Integer Types

Table 2-1 shows the complete set of fundamental types that store signed integers — that is both positive and negative values. The memory allocated for each type, and hence the range of values it can store, may vary between different compilers.

Table 2-1. *Signed Integer types*

Type Name	Typical Size (bytes)	Range of Values
signed char	1	-128 to 127
short	2	-256 to 255
short int		
int	4	-2,147,483,648 to +2,147,483,647
long	4	-2,147,483,648 to +2,147,483,647
long int		
long long	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
long long int		

Type `signed char` is typically 1 byte; the number of bytes occupied by the others depends on the compiler. Where two type names appear in the left column, the abbreviated name that comes first is commonly used so you will usually see `long` used rather than `long int`. Each type will have at least as much memory as the one that precedes it in the list.

Unsigned Integer Types

Of course, there are circumstances where you don't need to store negative numbers. The number of students in a class or the number of parts in an assembly are always positive integers. You can specify integer types that only store non-negative values by prefixing any of the names of the signed integer types with the `unsigned` keyword - types `unsigned char` or `unsigned short` or `unsigned long` for example. Each unsigned type is a different type from the signed type but occupies the same amount of memory.

Type `char` is a different integer type from both `signed char` and `unsigned char`. Type `char` stores a character code and can be a signed or unsigned type depending on your compiler. If the constant `CHAR_MIN` in the `climits` header is 0, then `char` is an unsigned type with your compiler. I'll have more to say about types that store characters later in this chapter.

Here are some examples of variables of some of these types:

```
signed char ch {20};
long temperature {-50L};
long width {500L};
long long height {250LL};
unsigned int toe_count {10U};
unsigned long angel_count {1000000UL};
```

Note how you write constants of type `long` and type `long long`. You must append `L` to the first and `LL` to the second. If there is no suffix, an integer constant is of type `int`. You can use lowercase for the `L` and `LL` suffixes but I recommend that you don't because lowercase `L` is easily confused with the digit `1`. Unsigned integer constants have `u` or `U` appended.

Defining Variables with Fixed Values

Sometimes you'll want to define variables with values that are fixed and must not be changed. You use the `const` keyword in the definition of a variable that must not be changed. For example:

```
const unsigned int toe_count {2U};
```

The `const` keyword tells the compiler that the value of `toe_count` must not be changed. A statement that attempts to modify the value of `toe_count` will be flagged as an error during compilation. You can use the `const` keyword to fix the value of variables of any type.

Integer Literals

Constants of any kind, such as `42`, or `2.71828`, `'Z'`, or `"Mark Twain"`, are referred to as *literals*. These examples are, in sequence, an *integer literal*, a *floating-point literal*, a *character literal*, and a *string literal*. Every literal will be of some type. I'll first explain integer literals, and introduce the other kinds of literals in context later.

Decimal Integer Literals

You can write integer literals in a very straightforward way. Here are some examples of decimal integers:

```
-123L      +123      123      22333      98U  -1234LL      12345ULL
```


You have seen that unsigned integer literals have `u` or `U` appended. Literals of types `long` and `long long` have `L` or `LL` appended respectively, and if they are unsigned, they also have `u` or `U` appended. The `U` and `L` or `LL` can be in either sequence. You could omit the `+` in the second example, as it's implied by default, but if you think putting it in makes things clearer, that's not a problem. The literal `+123` is the same as `123` and is of type `int` because there is no suffix. The fourth example is the number that you would normally write as 22,333, but you must not use commas in an integer literal. Here are some statements using some of these literals:

```
unsigned long age {99UL};           // 99uL would be OK too
unsigned short {10u};               // There is no specific literal type for short
long long distance {1234567LL};
```

You can't write just any old integer value as an initial value for a variable. An initializing value must be within the permitted range for the type of variable as well as match the type. A literal in an expression must be within the range of some type.

Hexadecimal Literals

You can write integer literals as hexadecimal values. You prefix a hexadecimal literal with `0x` or `0X`, so `0x999` is a hexadecimal number of type `int` with three hexadecimal digits. Plain old `999`, on the other hand, is a decimal value of type `int` with decimal digits, so the value will be completely different. Here are some more examples of hexadecimal literals:

Hexadecimal literals:	<code>0x1AF</code>	<code>0x123U</code>	<code>0xAL</code>	<code>0xcad</code>	<code>0xFF</code>
Decimal literals:	<code>431</code>	<code>291U</code>	<code>10L</code>	<code>3245</code>	<code>255</code>

A major use for hexadecimal literals is to define particular patterns of bits. Each hexadecimal digit corresponds to 4 bits so it's easy to express a pattern of bits as a hexadecimal literal. The red, blue, and green components (RGB values) of a pixel color are often expressed as three bytes packed into a 32-bit word. The color white can be specified as `0xFFFFFF`, because the intensity of each of the three components in white have the same maximum value of 255, which is `0xFF`. The color red would be `0xff0000`. Here are some examples:

```
unsigned int color {0x0f0d0eU}; // Unsigned int hexadecimal constant - decimal 986,382
int mask {0xFF00FF00};          // Four bytes specified as FF, 00, FF, 00
unsigned long value {0xdeadLU}; // Unsigned long hexadecimal literal - decimal 57,005
```

Octal Literals

You can also write integer literals as octal values—that is, using base 8. You identify a number as octal by writing it with a leading zero.

Octal literals:	<code>0657</code>	<code>0443U</code>	<code>012L</code>	<code>06255</code>	<code>0377</code>
Decimal literals:	<code>431</code>	<code>291U</code>	<code>10L</code>	<code>3245</code>	<code>255</code>

■ **Caution** Don't write decimal integer values with a leading zero. The compiler will interpret such values as octal (base 8), so a value written as 065 will be the equivalent of 53 in decimal notation.

Binary Literals

You write a binary integer literal as a sequence of binary digits (0 or 1) prefixed by 0b or 0B. A binary literal can have L or LL as a suffix to indicate it is type long or long long, and u or U if it is an unsigned literal. For example:

Binary literals:	0B110101111	0b100100011U	0b1010L	0B11001101	0b11111111
Decimal literals:	431	291U	10L	3245	255

Binary literals were introduced by the C++ 14 standard so at the time of writing there are not many compilers that support them. Here are some examples of their use:

```
int color {0b000011110000110100001110};  
  
int mask {0B11111111000000001111111100000000}; // 4 bytes  
  
unsigned long value {0B1101111010101101UL};
```

I have illustrated in the code fragments how you can write various combinations for the prefixes and suffixes such as 0x or 0X, and UL, LU, or Lu, but it's best to stick to a consistent way of writing integer literals. As far as your compiler is concerned, it doesn't matter which number base you choose when you write an integer value. Ultimately it will be stored as a binary number. The different ways for writing an integer are there just for your convenience. You choose one or other of the possible representations to suit the context.

■ **Note** You can use a single quote as a separator in an integer literal to make the literal easier to read. For example, 0B1111'1010 or 23'568'987UL. Few compilers support this at the time of writing, though, so it may not work for you.

Calculations with Integers

To begin with, let's get some bits of terminology out of the way. An operation such as addition or multiplication is defined by an *operator*—+ for addition; for example, or * for multiplication. The values that an operator acts upon are called *operands*, so in an expression such as 2*3, the operands are 2 and 3. Operators such as multiplication that require two operands are called *binary operators*. Operators that require one operand are called *unary operators*. An example of a unary operator is the minus sign in the expression -width. The minus sign negates the value of width so the result of the expression is a value with the opposite sign to that of its operand. This contrasts with the binary multiplication operator in expressions such as width*height, which acts on two operands, width and height.

The basic arithmetic operations that you can carry out on integers are shown in Table 2-2.

Table 2-2. *Basic Arithmetic Operations*

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (the remainder after division)

The operators in Table 2-2 are all binary operators and work largely in the way you would expect. Multiplication, division, and modulus operations in an expression execute before addition and subtraction. Here's an example of their use:

```
long width {4L};
long length {5L};
long area {0L};
long perimeter {0L};
area = width*length;           // Result is 20
perimeter = 2L*width + 2L*length; // Result is 28
```

The last two lines are *assignment statements* and the = is the *assignment operator*. The arithmetic expression on the right of the assignment operator is evaluated and the result is stored in the variable on the left. There are two variables initialized with 0L. You could omit the 0L in the initializer list here and the effect would be the same because an empty initializer list is assumed to contain zero. The second and third statements that define area and perimeter could be written:

```
long area {};
long perimeter {};
```

It's important to appreciate that an assignment operator is quite different from an = in an algebraic equation. The latter implies equality whereas the former is specifying an action. Consider the assignment statement in the following:

```
int y {5};
y = y + 1;
```

The variable y is initialized with 5 so the expression y+1 produces 6. This result is stored back in y so the effect is to increment y by 1.

You can control the order in which more complicated expressions are executed using parentheses. You could write the statement that calculates a value for perimeter as:

```
perimeter = 2L*(width + length);
```

The sub-expression within the parentheses is evaluated first. The result is multiplied by two, which produces the same result as before. This is more efficient than the original statement because it requires three arithmetic operations instead of four.

Parentheses can be nested, in which case sub-expressions between parentheses are executed in sequence from the innermost pair of parentheses to the outermost. An example of an expression with nested parentheses will show how it works

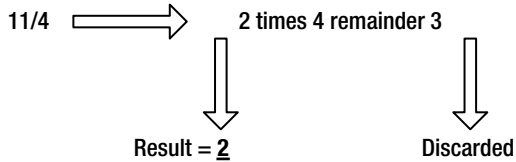
$$2*(a + 3*(b + 4*(c + 5*d)))$$

The expression $c+5*d$ is evaluated first and c is added to the result. That result is multiplied by 4 and b is added. That result is multiplied by 3 and a is added. Finally that result is multiplied by 2 to produce the result of the complete expression.

The division operation is slightly idiosyncratic. Integer operations always produce an integer result, so an expression such as $11/4$ produces 2 rather than 2.75. Integer division returns the number of times that the denominator divides into the numerator. Any remainder is discarded. So far as the C++ standard is concerned, the result of division by zero is undefined, but specific implementations will usually have the behavior defined so check your product documentation.

Figure 2-1 illustrates the effects of the division and modulus operators.

Integer Divide Operator



Modulus Operator

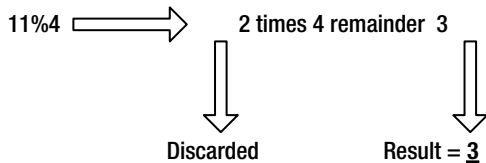


Figure 2-1. *Contrasting the division and modulus operators*

The modulus operator, `%`, complements the division operator in that it produces the remainder after integer division. When either or both operands of the modulus operator are negative, the sign of the remainder is up to the particular compiler you're using, so beware of variations between different systems. Applying the modulus operator inevitably involves a division so the result is undefined when the right operand is zero.

More on Assignment Operations

You can assign a value to more than one variable in a single statement. For example:

```
int a {}, b {}, c {5}, d{4};
a = b = c*c - d*d;
```

The second statement calculates the value of the expression $c*c-d*d$ and stores the result in `b`, so `b` will be set to 9. Next the value of `b` is stored in `a` so `a` will also be set to 9. You can have as many repeated assignments like this as you want.

The operand on the left of an assignment can be a variable or an expression, but if it is an expression, the result must be an *lvalue*. An *lvalue* represents a persistent memory location so a variable is an *lvalue*. Every expression in C++ results in either an *lvalue* or an *rvalue*. An *rvalue* is a result that is not an *lvalue*, so it is transient. The result of the expression $c*c-d*d$ in the statement above is an *rvalue*. The compiler allocates a temporary memory location to store the result of the expression but once the statement has been executed, the result and the memory it occupies is discarded. The difference between *lvalues* and *rvalues* is not important now, but it will become very important when you delve into functions and classes.

Let's see some of the arithmetic operators in action in an example. This program converts distances that you enter from the keyboard and in the process illustrates using the arithmetic operators:

```
// Ex2_02.cpp
// Converting distances
#include <iostream>                                // For output to the screen

int main()
{
    unsigned int yards {}, feet {}, inches {};

    // Convert a distance in yards, feet, and inches to inches
    std::cout << "Enter a distance as yards, feet, and inches "
                << "with the three values separated by spaces:"
                << std::endl;
    std::cin >> yards >> feet >> inches;

    const unsigned int feet_per_yard {3U};
    const unsigned int inches_per_foot {12U};
    unsigned int total_inches {};
    total_inches = inches + inches_per_foot*(yards*feet_per_yard + feet);
    std::cout << "The distances corresponds to " << total_inches << " inches.\n";

    // Convert a distances in inches to yards feet and inches
    std::cout << "Enter a distance in inches: ";
    std::cin >> total_inches;
    feet = total_inches/inches_per_foot;
    inches = total_inches%inches_per_foot;
    yards = feet/feet_per_yard;
    feet = feet%feet_per_yard;
    std::cout << "The distances corresponds to "
                << yards << " yards "
                << feet << " feet "
                << inches << " inches." << std::endl;
}
```

An example of typical output from this example is:

```
Enter a distance as yards, feet, and inches with the three values separated by spaces:
9 2 11
The distances corresponds to 359 inches.
Enter a distance in inches: 359
The distances corresponds to 9 yards 2 feet 11 inches.
```

The first statement in `main()` defines three integer variables and initializes them with zero. They are type `unsigned int` because in this example the distances values cannot be negative. This is an instance where defining three variables in a single statement is reasonable because they are closely related.

The next statement outputs a prompt to `std::cout` for the input. The statement is spread over three lines but it could be written as three separate statements:

```
std::cout << "Enter a distance as yards, feet, and inches ";
std::cout << "with the three values separated by spaces:";
std::cout << std::endl;
```

When you have a sequence of << operators as in the original statement they execute from left to right so the output from the three statements above will be exactly the same as the original.

The next statement reads values from `cin` and stores them in the variables `yards`, `feet`, and `inches`. The type of value that the >> operator expects to read is determined by the type of variable in which the value is to be stored so unsigned integers are expected to be entered. The << operator ignores spaces and the first space following a value terminates the operation. This implies that you cannot read and store spaces using the << operator for a stream, even when you store them in variables that store characters. The input statement in the example could also be written as three separate statements:

```
std::cin >> yards;
std::cin >> feet;
std::cin >> inches;
```

The effect of these statements is the same as the original.

You define two variables, `inches_per_foot` and `feet_per_yard` that you need to convert from yards, feet, and inches to inches and vice versa. The values for these are fixed so you specify the variables as `const`. You could use explicit values for conversion factors in the code but using `const` variables is much better because it is clear what you are doing. The `const` variables are also positive values so you define them as type `unsigned int`. The conversion to inches is done in a single statement:

```
total_inches = inches + inches_per_foot*(yards*feet_per_yard + feet);
```

The expression between parentheses executes first. This converts the `yards` value to feet and adds the `feet` value to produce the total number of feet. Multiplying this result by `inches_per_foot` obtains the total number of inches for the values of `yards` and `feet`. Adding `inches` to that produces the final total number of inches, which you output using this statement:

```
std::cout << "The distances corresponds to " << total_inches << " inches.\n";
```

The first string is transferred to the standard output stream, `cout`, followed by the value of `total_inches`. The string that is transferred to `cout` next has `\n` as the last character, which will cause the next output to start on the next line.

Converting a value from inches to yards, feet and inches requires four statements:

```
feet = total_inches/inches_per_foot;
inches = total_inches%inches_per_foot;
yards = feet/feet_per_yard;
feet = feet%feet_per_yard;
```

You reuse the variables that stored the input for the previous conversion to store the results of this conversion. Dividing the value of `total_inches` by `inches_per_foot` produces the number of whole feet, which you store in `feet`. The `%` operator produces the remainder after division so the next statement calculates the number of residual inches, which is stored in `inches`. The same process is used to calculate the number of yards and the final number of feet.

There's no `return` statement after the final output statement because it isn't necessary. When the execution sequence runs beyond the end of `main()`, it is equivalent to executing `return 0`.

The op= Assignment Operators

In `Ex2_01.cpp`, there was a statement that you could write more economically:

```
feet = feet%feet_per_yard;
```

This statement could be written using an `op=` assignment operator. The `op=` assignment operators are so called because they're composed of an operator and an assignment operator `=`. You could write the previous statement as:

```
feet %= feet_per_yard;
```

This is exactly the same operation as the previous statement.

In general, an `op=` assignment is of the form:

```
lhs op= rhs;
```

`lhs` represents a variable of some kind that is the destination for the result of the operator. `rhs` is any expression. This is equivalent to the statement:

```
lhs = lhs op (rhs);
```

The parentheses are important because you can write statements such as:

```
x *= y + 1;
```

This is equivalent to:

```
x = x*(y + 1);
```

Without the implied parentheses, the value stored in `x` would be the result of `x*y+1`, which is quite different.

You can use a range of operators for `op` in the `op=` form of assignment. Table 2-3 shows the complete set, including some operators you'll meet in Chapter 3.

Table 2-3. *op= Assignment Operators*

Operation	Operator	Operation	Operator
Addition	<code>+=</code>	Bitwise AND	<code>&=</code>
Subtraction	<code>-=</code>	Bitwise OR	<code> =</code>
Multiplication	<code>*=</code>	Bitwise exclusive OR	<code>^=</code>
Division	<code>/=</code>	Shift left	<code><<=</code>
Modulus	<code>%=</code>	Shift right	<code>>>=</code>

Note that there can be no spaces between `op` and the `=`. If you include a space, it will be flagged as an error. You can use `+=` when you want to increment a variable by some amount. For example, the following two statements have the same effect:

```
y = y + 1;
y += 1;
```

The shift operators that appear in the table, `<<` and `>>`, look the same as the insertion and extraction operators that you have been using with streams. The compiler can figure out what `<<` or `>>` means in a statement from the context. You'll understand how it is possible that the same operator can mean different things in different situations later in the book.

using Declarations and Directives

There were a lot of occurrences of `std::cin` and `std::cout` in `Ex2_01.cpp`. You can eliminate the need to qualify a name with the namespace name in a source file with a *using declaration*. Here's an example:

```
using std::cout;
```

This tells the compiler that when you write `cout`, it should be interpreted as `std::cout`. With this declaration before the `main()` function definition, you can write `cout` instead of `std::cout`, which saves typing and makes the code look a little less cluttered.

You could include two *using declarations* at the beginning of `Ex2_01.cpp` and avoid the need to qualify `cin` and `cout`:

```
using std::cin;
using std::cout;
```

Of course, you still have to qualify `endl` with `std`, although you could add a *using declaration* for that too. You can apply *using declarations* to names from any namespace, not just `std`.

A *using directive* imports all the names from a namespace. Here's how you could use any name from the `std` namespace without the need to qualify it:

```
using namespace std; // Make all the names in std available without qualification
```

With this at the beginning of a source file, you don't have to qualify any name that is defined in the `std` namespace. At first sight this seems an attractive idea. The problem is it defeats a major reason for having namespaces. It is unlikely that you know all the names that are defined in `std` and with this *using directive* you have increased the probability of accidentally using a name from `std`.

I'll use a *using directive* for the `std` namespace occasionally in examples in the book where the number of *using declarations* that would otherwise be required is excessive. I recommend that you only make use of *using directives* when there's a very good reason to do so.

The sizeof Operator

You use the `sizeof` operator to obtain the number of bytes occupied by a type, or by a variable, or by the result of an expression. Here are some examples of its use:

```
int height {74};
std::cout << "The height variable occupies " << sizeof height << " bytes." << std::endl;
std::cout << "Type \"long long\" occupies " << sizeof (long long) << " bytes." << std::endl;
std::cout << "The expression height*height/2 occupies "
    << sizeof (height*height/2) << " bytes." << std::endl;
```

These statements show how you can output the size of a variable, the size of a type, and the size of the result of an expression. To use `sizeof` to obtain the memory occupied by a type, the type name must be between parentheses. You also need parentheses around an expression with `sizeof`. You don't need parentheses around a variable name, but there's no harm in putting them in. Thus if you always use parentheses with `sizeof`, you can't go wrong.

You can apply `sizeof` to any fundamental type, class type, or pointer type (you'll learn about pointers in Chapter 5). The result that `sizeof` produces is of type `size_t`, which is an unsigned integer type that is defined in the Standard Library header `cstdint`. Type `size_t` is implementation defined, but don't bother to look it up for your compiler. If you use `size_t`, your code will work with any compiler.

Now you should be able to create your own program to list the sizes of the fundamental integer types with your compiler.

Incrementing and Decrementing Integers

You've seen how you can increment a variable with the `++` operator and I'm sure you've deduced that you can decrement a variable with `--`. There are two other operators that can perform the same tasks. They're called the *increment operator* and the *decrement operators*, `++` and `--` respectively.

These operators are more than just other options. You'll see a lot more of them and you'll find them to be quite an asset once you get further into C++. These are unary operators that you can apply to an integer variable. The following statements that modify count have exactly the same effect:

```
int count {5};
count = count + 1;
count += 1;
++count;
```

Each statement increments count by 1. Using the increment operator is clearly the most concise. The action of this operator is different from other operators that you've seen in that it directly modifies the value of its operand. The effect in an expression is to increment the value of the variable and then to use the incremented value in the expression. For example, suppose count has the value 5, and you execute this statement:

```
total = ++count + 6;
```

The increment and decrement operators execute before any other binary arithmetic operators in an expression. Thus, count will be incremented to 6, and this value will be used in the evaluation of the expression on the right of the assignment. total will therefore be assigned the value 12.

You use the decrement operator in the same way:

```
total = --count + 6;
```

Assuming count is 6 before this statement, the `--` operator will decrement it to 5, and this value will be used to calculate the value to be stored in total, which will be 11.

You've seen how you place a `++` and `--` operator before the variable to which it applies. This is called the *prefix form* of these operators. You can also place them after a variable, which is called the *postfix form*. The effect is a little different.

Postfix Increment and Decrement Operations

The postfix form of `++` increments the variable to which it applies *after* its value is used in context. For example, you can rewrite the earlier example as follows:

```
total = count++ + 6;
```

With an initial value of 5 for count, total is assigned the value 11. count will then be incremented to 6. The preceding statement is equivalent to the following statements:

```
total = count + 6;
++count;
```

In an expression such as `a++ + b`, or even `a+++b`, it's less than obvious what you mean, or indeed what the compiler will do. These two expressions are actually the same, but in the second case you might have meant `a + ++b`, which is different—it evaluates to one more than the other two expressions. It would be clearer to write the preceding statement as follows:

```
total = 6 + count++;
```

Alternatively, you can use parentheses:

```
total = (count++) + 6;
```

The rules that I've discussed in relation to the increment operator also apply to the decrement operator. For example, suppose `count` has the initial value 5, and you write this statement:

```
total = --count + 6;
```

This results in `total` having the value 10 assigned. However, consider this statement:

```
total = 6 + count-- ;
```

In this instance `total` is set to 11.

You must not apply the prefix form of these operators to a given variable more than once in an expression. Suppose `count` has the value 5, and you write this:

```
total = ++count * 3 + ++count * 5;
```

Because the statement modifies the value of `count` more than once, the result is undefined. You should get an error message from the compiler with this statement.

Note also that the effects of statements such as the following are undefined:

```
k = ++k + 1;
```

Here you're incrementing the value of the variable that appears on the left of the assignment operator in the expression on the right, so you're attempting to modify the value of `k` twice. A variable can be modified only once as a result of evaluating a single expression, and the prior value of the variable may only be accessed to determine the value to be stored. Although such expressions are undefined according to the C++ standard, this doesn't mean that your compiler won't compile them. It just means that there is no guarantee of consistency in the results.

The increment and decrement operators are usually applied to integers, particularly in the context of loops, as you'll see in Chapter 5. You'll see later in this chapter that you can apply them to floating-point variables too. In later chapters, you'll explore how they can also be applied to certain other data types, in some cases with rather specialized (but very useful) effects.

Defining Floating-Point Variables

You use floating-point variables whenever you want to work with values that are not integral. There are three floating-point data types, as shown in Table 2-4.

Table 2-4. *Floating-Point Data Types*

Data Type	Description
float	Single precision floating-point values
double	Double precision floating-point values
long double	Double-extended precision floating-point values

The term “precision” refers to the number of significant digits in the mantissa. The types are in order of increasing precision, with float providing the lowest number of digits in the mantissa and long double the highest. Note that the precision only determines the number of digits in the mantissa. The range of numbers that can be represented by a particular type is determined by the range of possible exponents.

The precision and range of values aren’t prescribed by the C++ standard so what you get with each type depends on your compiler. This will depend on what kind of processor is used by your computer and the floating-point representation it uses. Type long double will provide a precision that’s no less than that of type double, and type double will provide a precision that is no less than that of type float.

Typically, float provides 7 digits precision, double provides 15 digits, and long double provides 19 digits precision; double and long double have the same precision with some compilers. Typical ranges of values that you can represent with the floating-point types on an Intel processor are shown in Table 2-5.

Table 2-5. *Floating-Point Type Ranges*

Type	Precision (Decimal Digits)	Range (+or –)
float	7	1.2×10^{-38} to 3.4×10^{38}
double	15	2.2×10^{-308} to 1.8×10^{308}
long double	19	3.3×10^{-4932} to 1.2×10^{4932}

The numbers of digits of precision in Table 2-5 are approximate. Zero can be represented exactly with each type, but values between zero and the lower limit in the positive or negative range can’t be represented, so the lower limits are the smallest possible nonzero values.

Here are some statements that define floating point variables:

```
float inches_to_mm {25.4f};
double pi {3.1415926};           // Ratio of circle circumference to diameter
long double root2 {1.4142135623730950488L}; // Square root of 2
```

As you see, you define floating-point variables just like integer variables. Type double is more than adequate in the majority of circumstances.

Floating-Point Literals

You can see from the code fragment in the previous section that float literals have f (or F) appended and long double literals have L (or l) appended. Floating point literals without a suffix are of type double. A floating-point literal includes either a decimal point, or an exponent, or both; a numeric literal with neither is an integer.

An exponent is optional in a floating-point literal and represents a power of 10 that multiplies the value. An exponent must be prefixed with `e` or `E` and follows the value. Here are some floating-point literals that include an exponent:

`5E3` (5000.0) `100.5E2` (10050.0) `2.5e-3` (0.0025) `-0.1E-3L` (-0.0001L) `.345e1F` (3.45F)

The value between parentheses following each literal with an exponent is the equivalent literal without the exponent. Exponents are particularly useful when you need to express very small or very large values.

Floating-Point Calculations

You write floating-point calculations in the same way as integer calculations. For example:

```
const double pi {3.1414926};    // Circumference of a circle divided by its diameter
double a {0.75};                // Thickness of a pizza
double z {5.5};                 // Radius of a pizza
double volume {};               // Volume of pizza - to be calculated
volume = pi*z*z*a;
```

The modulus operator, `%`, can't be used with floating-point operands, but all the other binary arithmetic operators that you have seen, `+`, `-`, `*`, and `/`, can be. You can also apply the prefix and postfix increment and decrement operators, `++` and `--`, to a floating-point variable with essentially the same effect as for an integer: the variable will be incremented or decremented by 1.0.

Pitfalls

You need to be aware of the limitations of working with floating-point values. It's not difficult for the unwary to produce results that may be inaccurate, or even incorrect. Common sources of errors when using floating-point values are:

- Many decimal values don't convert exactly to binary floating-point values. The small errors that occur can easily be amplified in your calculations to produce large errors.
- Taking the difference between two nearly identical values will lose precision. If you take the difference between two values of type `float` that differ in the sixth significant digit, you'll produce a result that will have only one or two digits of accuracy. The other digits in the mantissa will be garbage.
- Working with values that differ by several orders of magnitude can lead to errors. An elementary example of this is adding two values stored as type `float` with 7 digits of precision where one value is 10^8 times larger than the other. You can add the smaller value to the larger as many times as you like, and the larger value will be unchanged.
- The `cfloat` Standard Library header contains information relating to floating-point operations with your compiler. Among many other things, it defines the following values, where the prefixes `FLT_`, `DBL_`, and `LDBL_` identify constants relating to the types `float`, `double`, and `long double` respectively:
 - `FLT_MANT_DIG`, `DBL_MANT_DIG`, and `LDBL_MANT_DIG` are the number of bits in the mantissa.
 - `FLT_EPSILON`, `DBL_EPSILON`, and `LDBL_EPSILON` are the smallest values that you can add to 1.0 and get a different result.

- FLT_MAX, DBL_MAX, and LDBL_MAX are the maximum floating-point numbers that can be represented.
- FLT_MIN, DBL_MIN, and LDBL_MIN are the minimum non-zero floating-point numbers that can be represented.

It's very easy to output these constants. Here's a complete program that illustrates how:

```
// Ex2_03.cpp
// Writing floating-point properties to cout
#include <iostream> // For output to the screen
#include <cfloat>

int main()
{
    std::cout << "The mantissa for type float has " << FLT_MANT_DIG << " bits." << std::endl;
    std::cout << "The maximum value of type float is " << FLT_MAX << std::endl;
    std::cout << "The minimum non-zero value of type float is " << FLT_MIN << std::endl;
}
```

This example in the code download outputs more constants than shown here.

Invalid Floating-Point Results

The result of division by zero is undefined so far as the C++ standard is concerned, but specific compilers have their own way of dealing with this, so consult your product documentation. Hardware floating-point operations in most computers are implemented according to the IEEE 754 standard (also known as IEC 559). The floating-point standard defines special values having a binary mantissa of all zeroes and an exponent of all ones to represent $+\infty$ or $-\infty$, depending on the sign. When you divide a positive non-zero value by zero, the result will be $+\infty$, and dividing a negative value by zero will result in $-\infty$. Another special floating-point value is called Not a Number, usually abbreviated to NaN. This represents a result that isn't mathematically defined, such as when you divide zero by zero or ∞ by ∞ .

Any operation in which either or both operands are NaN results in NaN. Once an operation results in $\pm\infty$, this will pollute all subsequent operations in which it participates. Table 2-6 summarizes all the possibilities.

value in the table is any non-zero value. You can discover how your compiler presents these values by plugging the following code into `main()`:

```
double a{ 1.5 }, b{}, c{}, result{};
result = a / b;
std::cout << a << "/" << b << " = " << result << std::endl;
std::cout << result << " + " << a << " = " << result + a << std::endl;
result = b / c;
std::cout << b << "/" << c << " = " << result << std::endl;
```

Table 2-6. *Floating-Point Operations with NaN and ±infinity Operands*

Operation	Result	Operation	Result
±value/0	±infinity	0/0	NaN
±infinity ± value	±infinity	±infinity/±infinity	NaN
±infinity*value	±infinity	infinity-infinity	NaN
±infinity/value	±infinity	infinity*0	NaN

You’ll see from the output when you run this how ±infinity and NaN look.

Mathematical Functions

The `cmath` Standard Library header file defines a range of trigonometric and numerical functions that you can use in your programs. All the function names are in the `std` namespace. Table 2-7 presents some of the most useful functions from this header.

Table 2-7. *Numerical Functions in the `cmath` header*

Function	Description
<code>abs(arg)</code>	Returns the absolute value of <code>arg</code> as the same type as <code>arg</code> , where <code>arg</code> can be of any floating-point type. There are versions of <code>abs()</code> in the <code>cstdlib</code> header file for arguments of any integer type that will return the result as an integer type.
<code>fabs(arg)</code>	Returns the absolute value of <code>arg</code> as the same type as the argument. The argument can be <code>int</code> , <code>long</code> , <code>float</code> , <code>double</code> , or <code>long double</code> .
<code>ceil(arg)</code>	Returns a floating-point value of the same type as <code>arg</code> that is the smallest integer greater than or equal to <code>arg</code> , so <code>std::ceil(2.5)</code> produces 3.0 and <code>std::ceil(-2.5)</code> produces -2.0. <code>arg</code> can be of any floating-point type.
<code>floor(arg)</code>	Returns a floating-point value of the same type as <code>arg</code> that is the largest integer less than or equal to <code>arg</code> so <code>std::floor(2.5)</code> results in 2.0 and <code>std::floor(-2.5)</code> results in -3.0. <code>arg</code> can be any floating-point type.
<code>exp(arg)</code>	Returns the value of e^{arg} as the same type as <code>arg</code> . <code>arg</code> can be of any floating-point type.
<code>log(arg)</code>	Returns the natural logarithm (to base e) of <code>arg</code> as the same type as <code>arg</code> . <code>arg</code> can be any floating-point type.
<code>log10(arg)</code>	Returns the logarithm to base 10 of <code>arg</code> as the same type as <code>arg</code> . <code>arg</code> can be any floating-point type.
<code>pow(arg1, arg2)</code>	Returns the value of <code>arg1</code> raised to the power <code>arg2</code> , which is <code>arg1^{arg2}</code> . <code>arg1</code> and <code>arg2</code> can be integer or floating-point types. Thus the result of <code>std::pow(2, 3)</code> will be 8, and the result of <code>std::pow(1.5, 3)</code> will be 3.375.

Table 2-8 shows some of the trigonometric functions provided by the `cmath` header.

Table 2-8. *Trigonometric Functions in the cmath Header*

Function	Description
<code>cos(angle)</code>	Returns the cosine of <code>angle</code> expressed in radians.
<code>sin(angle)</code>	Returns the sine of the <code>angle</code> expressed in radians.
<code>tan(angle)</code>	Returns the tangent of <code>angle</code> expressed in radians.
<code>cosh(angle)</code>	Returns the hyperbolic cosine of <code>angle</code> expressed in radians. The hyperbolic cosine of a variable x is given by the formula $(e^x + e^{-x})/2$.
<code>sinh(angle)</code>	Returns the hyperbolic sine of <code>angle</code> expressed in radians. The hyperbolic sine of a variable x is given by the formula $(e^x - e^{-x})/2$.
<code>tanh(angle)</code>	Returns the hyperbolic tangent of <code>angle</code> expressed in radians. The hyperbolic tangent of a variable x is given by the hyperbolic sine of x divided by the hyperbolic cosine of x .
<code>acos(arg)</code>	Returns the inverse cosine (arccosine) of <code>arg</code> . <code>arg</code> must be between -1 and $+1$. The result is in radians and will be from 0 to π .
<code>asin(arg)</code>	Returns the inverse sine (arcsine) of <code>arg</code> . The argument must be between -1 and $+1$. The result is in radians and will be from $-\pi/2$ to $+\pi/2$.
<code>atan(arg)</code>	Returns the inverse tangent (arctangent) of <code>arg</code> . The result is in radians and will be from $-\pi/2$ to $+\pi/2$.
<code>atan2(arg1, arg2)</code>	This requires two arguments of the same floating-point type. The function returns the inverse tangent of <code>arg1/arg2</code> . The result will be in the range from $-\pi$ to $+\pi$ radians and of the same type as the arguments.

The arguments to these functions can be of any floating-point type and the result will be returned as the same type as the argument(s).

Let's look at some examples of how these are used. Here's how you can calculate the sine of an angle in radians:

```
double angle {1.5};           // In radians
double sine_value {std::sin(angle)};
```

If the angle is in degrees, you can calculate the tangent by using a value for π to convert to radians:

```
float angle_deg {60.0f};      // Angle in degrees
const float pi {3.14159f};
const float pi_degrees {180.0f};
float tangent {std::tan(pi*angle_deg/pi_degrees)};
```

If you know the height of a church steeple is 100 feet and you're standing 50 feet from its base, you can calculate the angle in radians of the top of the steeple like this:

```
double height {100.0}         // Steeple height- feet
double distance {50.0}        // Distance from base
angle = std::atan2(height, distance); // Result in radians
```

You can use this value in `angle` and the value of `distance` to calculate the distance from your toe to the top of the steeple:

```
double toe_to_tip {distance*std::cos(angle)};
```

Of course, fans of Pythagoras of Samos could obtain the result much more easily, like this:

```
double toe_to_tip {std::sqrt(std::pow(distance,2) + std::pow(height, 2))};
```

Let's try a floating-point example. Suppose that you want to construct a circular pond in which you will keep fish. Having looked into the matter, you know that you must allow 2 square feet of pond surface area for every 6 inches of fish length. You need to figure out the diameter of the pond that will keep the fish happy. Here's how you can do it:

```
// Ex2_04.cpp
// Sizing a pond for happy fish
#include <iostream>
#include <cmath> // For square root function
using std::cout;
using std::cin;
using std::sqrt;

int main()
{
    // 2 square feet pond surface for every 6 inches of fish
    const double fish_factor {2.0/0.5}; // Area per unit length of fish
    const double inches_per_foot {12.0};
    const double pi {3.14159265};

    double fish_count {}; // Number of fish
    double fish_length {}; // Average length of fish

    cout << "Enter the number of fish you want to keep: ";
    cin >> fish_count;
    cout << "Enter the average fish length in inches: ";
    cin >> fish_length;
    fish_length /= inches_per_foot; // Convert to feet

    // Calculate the required surface area
    double pond_area {fish_count * fish_length * fish_factor};

    // Calculate the pond diameter from the area
    double pond_diameter {2.0 * sqrt(pond_area/pi)};

    cout << "\nPond diameter required for " << fish_count << " fish is "
         << pond_diameter << " feet.\n";
}
```


With input values of 20 fish with an average length of 9 inches, this example produces the following output:

```
Enter the number of fish you want to keep: 20
Enter the average fish length in inches: 9
Pond diameter required for 20 fish is 8.74039 feet.
```

The three using declarations allow the stream names and the `sqrt()` function name to be used without qualifying them with the namespace name. You first define three `const` variables in `main()` that you'll use in the calculation. Notice the use of a constant expression to specify the initial value for `fish_factor`. You can use any expression for an initial value that produces a result of the appropriate type. You specify `fish_factor`, `inches_per_foot`, and `pi` as `const` because their values are fixed and should not be altered.

Next, you define the `fish_count` and `fish_length` variables in which you'll store the user input. Both have an initial value of zero.

The input for the fish length is in inches so you convert it to feet before you use it in the calculation for the pond. You use the `/=` operator to convert the original value to feet.

You define a variable for the area for the pond and initialize it with an expression that produces the required value:

```
double pond_area {fish_count * fish_length * fish_factor};
```

The product of `fish_count` and `fish_length` gives the total length of all the fish in feet, and multiplying this by `fish_factor` gives the required area for the pond in square feet.

The area of a circle is given by the formula πr^2 , where r is the radius. You can therefore calculate the radius of the circular pond by dividing the area by π and calculating the square root of the result. The diameter is twice the radius so the whole calculation is carried out by this statement:

```
double pond_diameter {2.0 * sqrt(pond_area / pi)};
```

You obtain the square root using the `sqrt()` function from the `cmath` header.

Of course, you could calculate the pond diameter in a single statement like this:

```
double pond_diameter {2.0 * sqrt(fish_count * fish_length * fish_factor/ pi)};
```

This eliminates the need for the `pond_area` variable so the program will be smaller and shorter. It's debatable whether this is better than the original though because it's not so obvious what is going on.

The last statement in `main()` outputs the result. The pond diameter has more decimal places than you need. Let's look into how you can fix that.

Formatting Stream Output

You can change how data is formatted when it is written to an output stream using *stream manipulators*, which are functions declared in the `iomanip` Standard Library header. You apply a stream manipulator to an output stream with the insert operator, `<<`. Stream manipulators require you to supply a parameter value. There are also predefined constants in the `iostream` header that affect how data is presented when you insert them in an output stream. I'll just introduce the most useful manipulators and streams constants.

The `iomanip` header provides these useful parametric manipulators:

<code>std::setprecision(n)</code>	Sets the floating-point precision or the number of decimal places to <code>n</code> digits. If the default floating-point output presentation is in effect, <code>n</code> specifies the number of digits in the output value. If <code>fixed</code> or <code>scientific</code> format has been set, <code>n</code> is the number of digits following the decimal point. The value set by <code>setprecision()</code> remains in effect for subsequent output unless you change it.
<code>std::setw(n)</code>	Sets the output field width to <code>n</code> characters, but only for the next output data item. Subsequent output reverts to the default where the field width is set to the number of output character needed to accommodate the data.
<code>std::setfill(ch)</code>	When the field width has more characters than the output value, excess characters in the field will be the default fill character, which is a space. This sets the fill character to be <code>ch</code> for all subsequent output.

The `iostream` header defines the following stream constants:

<code>std::fixed</code>	Output floating-point data in fixed point notation.
<code>std::scientific</code>	Output all subsequent floating-point data in scientific notation, which always includes an exponent and one digit before the decimal point.
<code>std::defaultfloat</code>	Revert to the default floating-point data presentation.
<code>std::dec</code>	All subsequent integer output is decimal.
<code>std::hex</code>	All subsequent integer output is hexadecimal.
<code>std::oct</code>	All subsequent integer output is octal.
<code>std::showbase</code>	Outputs the base prefix for hexadecimal and octal integer values. Inserting <code>std::noshowbase</code> in a stream will switch this off.
<code>std::left</code>	Output is left-justified in the field.
<code>std::right</code>	Output is right-justified in the field. This is the default.
<code>std::internal</code>	Causes the fill character to be internal to an integer or floating-point output value.

When you insert any of these constants in an output stream, they remain in effect until you change it. Let's see how some of these work in practice. Consider this output statement:

```
cout << "\nPond diameter required for " << fish_count << " fish is "
    << std::setprecision(2)                // Output value is 8.7
    << pond_diameter << " feet.\n";
```

If you replace the output statement at the end of `Ex2_04.cpp` with this, you'll get the floating-point value presented with 2 digits precision, which will correspond to 1 decimal place in this case. Because default handling of floating-point output is in effect, the integer between the parentheses in `setprecision()` specifies the output

precision for floating-point values, which is the total number of digits before and after the decimal point. You can make the parameter specify the number of digits after the decimal point - the number of decimal places in other words, by setting the mode as fixed. For example, try this in `Ex2_04.cpp`:

```
cout << "\nPond diameter required for " << fish_count << " fish is "
    << std::fixed << std::setprecision(2)
    << pond_diameter << " feet.\n";           // Output value is 8.74
```

Setting the mode as fixed or as scientific causes the `setprecision()` parameter to be interpreted as the number of decimal places in the output value. Setting scientific mode causes floating-point output to be in scientific notation, which is with an exponent:

```
cout << "\nPond diameter required for " << fish_count << " fish is "
    << std::scientific << std::setprecision(2)
    << pond_diameter << " feet.\n";           // Output value is 8.74e+000
```

In scientific notation there is always one digit before the decimal point. The value set by `setprecision()` is still the number of digits following the decimal point. There's always a three-digit exponent value, even when the exponent is zero.

The following statement illustrates some of the formatting possible with integer values:

```
int a{16}, b{66};
cout << std::setw(5) << a << std::setw(5) << b << std::endl;
cout << std::left << std::setw(5) << a << std::setw(5) << b << std::endl;
cout << " a = " << std::setbase(16) << std::setw(6) << std::showbase << a
    << " b = " << std::setw(6) << b << std::endl;
cout << std::setw(10) << a << std::setw(10) << b << std::endl;
```

The output from these statements is:

```
   16   66
  6   66
a = 0x10  b = 0x42
x10      0x42
```

It's a good idea to insert `showbase` in the stream when you output integers as hexadecimal or octal so the output won't be misinterpreted as decimal values. I recommend that you try various combinations of these manipulators and stream constants to get a feel for how they all work.

Mixed Expressions and Type Conversion

You can write expressions involving operands of different types. For example, you could have defined the variable to store the number of fish, like this:

```
unsigned int fish_count {};           // Number of fish
```

The number of fish is certainly an integer so this makes sense. The number of inches in a foot is also integral so you would want to define the variable like this:

```
const unsigned int inches_per_foot {12};
```

The calculation would still work OK in spite of the variables now being of differing types. For example:

```
fish_length /=inches_per_foot;           // Convert to feet
double pond_area {fish_count * fish_length * fish_factor};
```

The binary arithmetic operands require both operands to be of the same type. Where this is not the case, the compiler will arrange to convert one of the operand values to the same type as the other. These are called *implicit conversions*. The way this works is that the variable of a type with the more limited range is converted to the type of the other. The `fish_length` variable in the first statement is of type `double`. Type `double` has a greater range than type `unsigned int` so the compiler will insert a conversion for the value of `inches_per_foot` to type `double` to allow the division to be carried out. In the second statement, the value of `fish_length` will be converted to type `double` to make it the same type as `fish_length` before the multiply operation executes.

With each operation with operands of different types, the compiler chooses the operand with the type that has the more limited range of values as the one to be converted to the type of the other. In effect, it ranks the types in the following sequence, from high to low:

1. long double	2. double	3. float
4. unsigned long long	5. long long	
6. unsigned long	7. long	
8. unsigned int	9. int	

The operand to be converted will be the one with the lower rank. Thus, in an operation with operands of type `long long` and type `unsigned int`, the latter will be converted to type `long long`. An operand of type `char`, signed `char`, unsigned `char`, `short`, or unsigned `short` is always converted to at least type `int`.

Implicit conversions can produce unexpected results. Consider these statements:

```
unsigned int x {20u};
int y {30};
std::cout << x - y << std::endl;
```

You might expect the output to be `-10`, but it isn't. The output will be `4294967286`. This is because the value of `y` is converted to `unsigned int` to match the type of `x` so the result of the subtraction is an unsigned integer value.

The compiler will also insert an implicit conversion when the expression on the right of an assignment produces a value that is of a different type from the variable on the left. For example:

```
int y {};
double z {5.0};
y = z;                               // Requires implicit conversion
```

The last statement requires a conversion of the value of the expression on the right of the assignment to allow it to be stored as type `int`. The compiler will insert a conversion to do this but in most cases it will also issue a warning message about possible loss of data.

You need to take care when writing integer operations with operands of different types. Don't rely on implicit type conversion to produce the result you want unless you are certain it will do so. If you are not sure, what you need is an *explicit type conversion*, also call an *explicit cast*.

Explicit Type Conversion

To convert the value of an expression to a given type, you write the following:

```
static_cast<type_to_convert_to>(expression)
```

The `static_cast` keyword reflects the fact that the cast is checked statically; that is, when the code is compiled. Later, when you get to deal with classes, you'll meet *dynamic casts*, where the conversion is checked dynamically; that is, when the program is executing. The effect of the cast is to convert the value that results from evaluating `expression` to the type that you specify between the angle brackets. The expression can be anything from a single variable to a complex expression involving lots of nested parentheses. You could eliminate the warning that arises from the assignment in the previous section by writing it as:

```
y = static_cast<int>(z);    // No compiler warning this time...
```

Here's another example of the use of `static_cast<>()`:

```
double value1 {10.5};
double value2 {15.5};
int whole_number {static_cast<int>(value1) + static_cast<int>(value2)};
```

The initializing value for `whole_number` is the sum of the integral parts of `value1` and `value2`, so they're each explicitly cast to type `int`. `whole_number` will therefore have the initial value 25. The casts do not affect the values stored in `value1` and `value2`, which will remain as 10.5 and 15.5, respectively. The values 10 and 15 produced by the casts are just stored temporarily for use in the calculation and then discarded. Although both casts cause a loss of information, the compiler always assumes that you know what you're doing when you explicitly specify a cast.

Of course, the value of `whole_number` would be different if you wrote:

```
int whole_number {static_cast<int>(value1 + value2)};
```

The result of adding `value1` and `value2` will be 26.0, which results in 26 when converted to type `int`. The compiler will not insert implicit narrowing conversions for values in an initializer list so the statement will not compile without the explicit type conversion.

Generally, the need for explicit casts should be rare, particularly with basic types of data. If you have to include a lot of explicit conversions in your code, it's often a sign that you could choose more suitable types for your variables. Still, there are circumstances when casting is necessary, so let's look at a simple example. This example converts a length in yards as a decimal value to yards, feet, and inches.

```
// Ex2_05.cpp
// Using Explicit Type
#include <iostream>

int main()
{
    const unsigned int feet_per_yard {3};
    const unsigned int inches_per_foot {12};
```

```

double length {};           // Length as decimal yards
unsigned int yards{};       // Whole yards
unsigned int feet {};       // Whole feet
unsigned int inches {};     // Whole inches

std::cout << "Enter a length in yards as a decimal: ";
std::cin >> length;

// Get the length as yards, feet, and inches
yards = static_cast<unsigned int>(length);
feet = static_cast<unsigned int>((length - yards)*feet_per_yard);
inches = static_cast<unsigned int>
    (length*feet_per_yard *inches_per_foot) % inches_per_foot;

std::cout << length << " yards converts to "
    << yards << " yards "
    << feet << " feet "
    << inches << " inches." << std::endl;
}

```

Typical output from this program will be:

```

Enter a length in yards as a decimal: 2.75
2.75 yards converts to 2 yards 2 feet 3 inches.

```

The first two statements in `main()` define conversion constants `feet_per_yard` and `inches_per_foot` as integers. You declare these as `const` to prevent them from being modified accidentally. The variables that will store the results of converting the input to yards, feet, and inches are of type `unsigned int` and initialized with zero.

The statement that computes the whole number of yards from the input value is:

```
yards = static_cast<unsigned int>(length);
```

The cast discards the fractional part of the value in `length` and stores the integral result in `yards`. You could omit the explicit cast here and leave it to the compiler to take care of but it's always better to write an explicit cast in such cases. If you don't, it's not obvious that you realized the need for the conversion and the potential loss of data.

You obtain the number of whole feet with this statement:

```
feet = static_cast<unsigned int>((length - yards)*feet_per_yard);
```

Subtracting yards from `length` produces the fraction of a yard in the `length` as a double value. The compiler will arrange for the value in yards to be converted to type `double` for the subtraction. The value of `feet_per_yard` will then be converted to `double` to allow the multiplication to take place, and finally the explicit cast converts the result from type `double` to type `unsigned int`.

The final part of the calculation obtains the residual number of whole inches:

```
inches = static_cast<unsigned int>
    (length*feet_per_yard *inches_per_foot) % inches_per_foot;
```

The explicit cast applies to the total number of inches in `length`, which results from the product of `length`, `feet_per_yard`, and `inches_per_foot`. Because `length` is type `double`, both `const` values will be converted implicitly to type `double` to allow the product to be calculated. The remainder after dividing the integral number of inches in `length` by the number of inches in a foot is the number of residual inches.

Old-Style Casts

Prior to the introduction of `static_cast` into C++, an explicit cast of the result of an expression was written like this:

```
(type_to_convert_to)expression
```

The result of `expression` is cast to the type between the parentheses. For example, the statement to calculate inches in the previous example could be written like this:

```
inches = (unsigned int)(length*feet_per_yard *inches_per_foot) % inches_per_foot;
```

There are several kinds of casts in C++ that are now differentiated, but the old-style casting syntax covers them all. Because of this, code using the old-style casts is more prone to error. It isn't always clear what you intended, and you may not get the result you expected. You'll still see old-style casting used because it's still part of the language but I strongly recommend that you use only the new casts in your code.

Finding the Limits

You have seen typical examples of the upper and lower limits for various types. The `limits` Standard Library header makes this information available for all the fundamental data types so you can access this for your compiler. Let's look at an example. To display the maximum value you can store in a variable of type `double`, you could write this:

```
std::cout << "Maximum value of type double is " << std::numeric_limits<double>::max();
```

The expression `std::numeric_limits<double>::max()` produces the value you want. By putting different type names between the angled brackets, you can obtain the maximum values for other data types. You can also replace `max()` with `min()` to get the minimum value that can be stored, but the meaning of minimum is different for integer and floating-point types. For an integer type, `min()` results in the true minimum, which will be a negative number for a signed integer type. For a floating-point type, `min()` returns the minimum positive value that can be stored.

You can retrieve many other items of information about various types. The number of binary digits, for example, is returned by this expression:

```
std::numeric_limits<type_name>::digits
```

`type_name` is the type in which you're interested. For floating-point types, you'll get the number of binary digits in the mantissa. For signed integer types, you'll get the number of binary digits in the value; that is, excluding the sign bit. The following program will display the maximums and minimums for some of the numerical data types.

```
// Ex2_06.cpp
// Finding maximum and minimum values for data types
#include <limits>
#include <iostream>

int main()
{
    std::cout << "The range for type short is from "
              << std::numeric_limits<short>::min() << " to "
              << std::numeric_limits<short>::max() << std::endl;
    std::cout << "The range for type int is from "
              << std::numeric_limits<int>::min() << " to "
              << std::numeric_limits<int>::max() << std::endl;
```

```

std::cout << "The range for type long is from "
          << std::numeric_limits<long>::min() << " to "
          << std::numeric_limits<long>::max() << std::endl;
std::cout << "The range for type float is from "
          << std::numeric_limits<float>::min() << " to "
          << std::numeric_limits<float>::max() << std::endl;
std::cout << "The range for type double is from "
          << std::numeric_limits<double>::min() << " to "
          << std::numeric_limits<double>::max() << std::endl;
std::cout << "The range for type long double is from "
          << std::numeric_limits<long double>::min() << " to "
          << std::numeric_limits<long double>::max() << std::endl;
}

```

You can easily extend this to include unsigned integer types and types that store characters.

Working with Character Variables

Variables of type `char` are used primarily to store a code for a single character and occupy 1 byte. The C++ standard doesn't specify the character encoding to be used for the basic character set, so in principle this is down to the particular compiler but it's usually ASCII.

You define variables of type `char` in the same way as variables of the other types that you've seen, for example:

```

char letter;                // Uninitialized - so junk value
char yes {'Y'}, no {'N'};   // Initialized with character literals
char ch {33};               // Integer initializer equivalent to '!'

```

You can initialize a variable of type `char` with a character literal between single quotes or by an integer. An integer initializer must be within the range of type `char` - remember it depends on the compiler whether it is a signed or unsigned type. Of course, you can specify a character as one of the escape sequences you saw in Chapter 1. There are also escape sequences that specify a character by its code expressed as either an octal or a hexadecimal value. The escape sequence for an octal character code is one to three octal digits preceded by a backslash. The escape sequence for a hexadecimal character code is one or more hexadecimal digits preceded by `\x`. You write either form between single quotes when you want to define a character literal. For example, the letter 'A' could be written as hexadecimal `'\x41'` or octal `'\81'` in ASCII. Obviously, you could write codes that won't fit within a single byte, in which case the result is implementation defined.

Variables of type `char` are numeric; after all, they store integer codes that represent characters. They can therefore participate in arithmetic expressions, just like variables of type `int` or `long`. For example:

```

char ch {'A'};
char letter {ch + 5};        // letter is 'F'
++ch;                       // ch is now 'B'
ch += 3;                    // ch is now 'E'

```

When you write a `char` variable to `cout`, it is output as a character, not as an integer. If you want to see it as a numerical value, you can cast it to another integer type. For example:

```

std::cout << "ch is '" << ch
          << "' which is code " << std::hex << std::showbase
          << static_cast<int>(ch) << std::endl;

```


This produces the output:

```
ch is 'E' which is code 0x45
```

When you read from a stream into a variable of type `char`, the first non-whitespace character will be stored. This means that you can't read whitespace characters in this way; they're simply ignored. Further, you can't read a numerical value into a variable of type `char`; if you try, the character code for the first digit will be stored.

Working with Unicode Characters

ASCII is generally adequate for national language character sets that use Latin characters. However, if you want to work with characters for multiple languages simultaneously, or if you want to handle character sets for Asian languages, 256 character codes doesn't go far enough and Unicode is the answer.

Type `wchar_t` is a fundamental type that can store all members of the largest extended character set that's supported by an implementation. The type name derives from *wide characters*, because the character is "wider" than the usual single-byte character. By contrast, type `char` is referred to as "narrow" because of the limited range of character codes that are available.

You define wide-character literals in a similar way to literals of type `char`, but you prefix them with `L`. For example

```
wchar_t wch {L'Z'};
```

This defines `wch` as type `wchar_t` and initializes it to the wide-character representation for `Z`.

Your keyboard may not have keys for representing other national language characters, but you can still create them using hexadecimal notation, for example:

```
wchar_t wch {L'\x0438'};    // Cyrillic И
```

The value between the single quotes is an escape sequence that specifies the hexadecimal representation of the character code. The backslash indicates the start of the escape sequence, and `x` or `X` after the backslash signifies that the code is hexadecimal.

Type `wchar_t` does not handle international character sets very well. It's much better to use type `char16_t` which stores characters encoded as UTF-16, or `char32_t`, which stores UTF-32 encoded characters. Here's an example of defining a variable of type `char16_t`:

```
char16_t letter {u'B'};           // Initialized with UTF-16 code for B
char16_t cyr {u'\x0438'};        // Initialized with UTF-16 code for cyrillic И
```

The lowercase `u` prefix to the literals indicates that are UTF-16. You prefix UTF-32 literals with uppercase `U`. For example:

```
char32_t letter {U'B'};           // Initialized with UTF-32 code for B
char32_t cyr {U'\x044f'};        // Initialized with UTF-32 code for cyrillic я
```

Of course, if your editor has the capability to accept and display the characters, you can define `cyr` like this:

```
char32_t cyr {U'я'};
```

The Standard Library provides standard input and output streams `wcin` and `wcout` for reading and writing characters of type `wchar_t`, but there is no provision with the library from handling `char16_t` and `char32_t` character data. Your compiler may have its own facilities for reading and writing these types.

■ **Caution** You should not mix output operations on `wcout` with output operations on `cout`. The first output operation on either stream sets an orientation for the standard output stream that is either *narrow* or *wide*, depending on whether the operation is to `cout` or `wcout`. The orientation will carry forward to subsequent output operations for either `cout` or `wcout`.

The auto Keyword

You use the `auto` keyword to indicate that the compiler should deduce the type. Here are some examples:

```
auto m = 10;           // m is type int
auto n = 200UL;        // n is type unsigned long
auto pi = 3.14159;     // pi is type double
```

Note the syntax for initialization here—using `=`. The compiler will deduce the types for `m`, `n`, and `pi` from the initial values you supply. Having said that, this is not how the `auto` keyword is intended to be used. Certainly for defining variables of fundamental types you should specify the type explicitly so you know for sure what it is. You'll meet the `auto` keyword again later in the book where it is more appropriately and much more usefully applied.

■ **Caution** You should not use an initializer list with the `auto` keyword because the type will be wrong. This is because an initializer list itself has a type. For example, suppose you write:

```
auto m {10};
```

The type assigned to `m` will not be `int`, but will be `std::initializer_list<int>`, which is the type of this particular initializer list.

You can use functional notation with `auto` for the initial value:

```
auto pi(3.14159);      // pi is type double
```

This is still not the way to use `auto` - just specify the type as `double` and use an initializer list.

Lvalues and Rvalues

Every expression results in either an *lvalue* or an *rvalue* (sometimes written *l-value* and *r-value* and pronounced like that). An lvalue refers to an address in memory in which something can be stored on an ongoing basis. An rvalue is a result that is stored transiently. An lvalue is so called because any expression that results in an lvalue can appear on the left of an assignment operator. If the result of an expression is not an lvalue, it is an rvalue. An expression that consists of a single named variable is always an lvalue.

Consider the following statements:

```
int a {}, b {1}, c {2};
a = b + c;
b = ++a;
c = a++;
```

The first statement defines `a`, `b`, and `c` as type `int` and initializes them to 0, 1, and 2, respectively. In the second statement, the result of evaluating `b+c` is stored temporarily and the value is copied to `a`. When execution of the statement is complete, the memory holding the result of `b+c` is discarded. Thus, the result of evaluating `b+c` is an rvalue.

In the third statement, the expression `++a` is an lvalue because its result is `a` after its value is incremented. The expression `a++` in the fourth statement is an rvalue because it stores the value of `a` temporarily as the result of the expression and then increments `a`.

■ **Note** This is by no means all there is to know about lvalues and rvalues. Most of the time you don't need to worry very much about whether an expression is an lvalue or an rvalue, but sometimes you do. You'll find out when it's important to be able to tell the difference later in the book when you learn about classes.

Summary

In this chapter, I covered the basics of computation in C++. You learned about most of the fundamental types of data that are provided for in the language. The essentials of what I've discussed up to now are as follows:

- Constants of any kind are called literals and literals have a type.
- You can define integer literals as decimal, hexadecimal, binary, or octal values.
- A floating-point literal must contain a decimal point, or an exponent, or both. If there is neither, you have specified an integer.
- The fundamental types that store integers are `short`, `int`, `long` and `long long`. These store signed integers but you can also use the type modifier `unsigned` preceding any of these type names to produce a type that occupies the same number of bytes but stores unsigned integers.
- The floating-point data types are `float`, `double`, and `long double`.
- Variables may be given initial values when they're defined and it's good programming practice to do so. An initializer list is the preferred way for specifying initial values.
- A variable of type `char` can store a single character and occupies 1 byte. Type `char` may be signed or unsigned, depending on your compiler. You can also use variables of the types `signed char` and `unsigned char` to store integers. Types `char`, `signed char`, and `unsigned char` are different types.
- Type `wchar_t` stores a wide character and occupies either 2 or 4 bytes, depending on your compiler. Types `char16_t` and `char32_t` are better for handling Unicode characters.
- You can fix the value of a variable by using the `const` modifier. The compiler will check for any attempts within the program source file to modify a variable defined as `const`.
- You can mix different types of variables and constants in an expression. The compiler will arrange for one operand in a binary operation to be automatically converted to the type of the other operand when they differ.

- The compiler will automatically convert the type of the result of an expression on the right of an assignment to the type of the variable on the left where these are different. This can cause loss of information when the left-side type isn't able to contain the same information as the right-side type — double converted to int, for example, or long converted to short.
- You can explicitly convert a value of one type to another using the `static_cast<>()` operator.
- An lvalue is an object or expression that can appear on the left side of an assignment. Non-const variables are examples of lvalues. An rvalue is a result of an expression that is transient.

EXERCISES

Exercise 2-1. Write a program that will compute the area of a circle. The program should prompt for the radius of the circle to be entered from the keyboard, calculate the area using the formula $\text{area} = \pi * \text{radius} * \text{radius}$, and then display the result.

Exercise 2-2. Using your solution for Exercise 2-1, improve the code so that the user can control the precision of the output by entering the number of digits required. (Hint: Use the `setprecision()` manipulator.)

Exercise 2-3. Create a program that converts inches to feet-and-inches. For example, an input of 77 inches should produce an output of 6 feet and 5 inches. Prompt the user to enter an integer value corresponding to the number of inches, and then make the conversion and output the result. (Hint: Use a `const` to store the inches-to-feet conversion rate; the modulus operator will be very helpful.)

Exercise 2-4. For your birthday you've been given a long tape measure and an instrument that measures angles (the angle between the horizontal and a line to the top of a tree, for instance). If you know the distance, d , you are from a tree, and the height, h , of your eye when peering into your angle-measuring device, you can calculate the height of the tree with the formula $h + d * \tan(\text{angle})$. Create a program to read h in inches, d in feet and inches, and angle in degrees from the keyboard, and output the height of the tree in feet.

There is no need to chop down any trees to verify the accuracy of your program. Just check the solutions on the Apress website!

Exercise 2-5. Here's an exercise for puzzle fans. Write a program that will prompt the user to enter two different positive integers. Identify in the output the value of the larger integer and the value of the smaller integer. (This *can* be done with what you've learned in this chapter!)

Exercise 2-6. Your Body Mass Index (BMI) is your weight w in kilograms divided by the square of your height h in meters ($w / (h * h)$). Write a program to calculate the BMI from a weight entered in pounds and a height entered in feet and inches. A kilogram is 2.2 lbs. and one foot is 0.3048 meters.



Working with Fundamental Data Types

In this chapter, I expand on the types that I discussed in the previous chapter and explain how variables of the basic types interact in more complicated situations. I also introduce some new features of C++ and discuss some of the ways that these are used. In this chapter you'll learn

- How the execution order in an expression is determined
- What the bitwise operators are and how you use them
- How you can define a new type that limits variables to a fixed range of possible values
- How you can define alternative names for existing data types
- What the storage duration of a variable is and what determines it
- What variable scope is and what its effects are

Operator Precedence and Associativity

You already know that there is a priority sequence for executing arithmetic operators in an expression. You'll meet many more operators throughout the book, including a few in this chapter. In general, the sequence in which operators in an expression are executed is determined by the *precedence* of the operators. Operator precedence is just a fancy term for the priority of an operator.

Some operators, such as addition and subtraction, have the same precedence. That raises the question of how an expression such as `a+b-c+d` is evaluated. When several operators from a group with the same precedence appear in an expression, in the absence of parentheses, the execution order is determined by the *associativity* of the group. A group of operators can be *left-associative* which means operators execute from left to right, or they can be *right-associative* which means they execute from right-to-left.

Nearly all operator groups are left-associative so most expressions involving operators of equal precedence are evaluated from left to right. The only right associative operators are the unary operators and assignment operators, which you'll meet later. The precedence and associativity of all the operators in C++ is shown in Table 3-1.

Table 3-1. *The Precedence and Associativity of C++ Operators*

Precedence	Operators	Associativity
1	::	None
2	() [] -> . postfix ++ postfix -- typeid const_cast dynamic_cast static_cast reinterpret_cast	Left
3	logical not ! one's complement ~ unary + unary - prefix ++ prefix -- address-of & indirection * type cast (type) sizeof decltype new new[] delete delete[]	Right
4	.* ->*	Left
5	* / %	Left
6	+ -	Left
7	<< >>	Left
8	== !=	Left
9	&	Left
10	^	Left
11		Left
12	&&	Left
13		Left
14	?: (conditional operator)	Right
15	= *= /= %= += -= &= ^= = <<= >>=	Right
16	throw	Right
17	,	Left

You haven't met most of these operators yet but when you need to know the precedence and associativity of any operator, you'll know where to find it. Each row in Table 3-1 is a group of operators of equal precedence and the rows are in precedence sequence, from highest to lowest. Let's take a simple example to make sure that it's clear how all this works. Consider this expression:

```
x*y/z - b + c - d
```

The `*` and `/` operators are in the same group with precedence that is higher than the group containing `+` and `-` so the expression `x*y/z` is evaluated first, with a result, `r`, say. The operators in the group containing `*` and `/` are left-associative, so the expression is evaluated as though it was `(x*y)/z`. The next step is the evaluation of `r-b+c-d`. The group containing the `+` and `-` operators is also left associative, so this will be evaluated as `((r-b)+c)-d`. Thus the whole expression is evaluated as though it was written as:

$$(((x*y)/z) - b) + c) - d$$

Remember, nested parentheses are evaluated in sequence from the innermost to the outermost. You probably won't be able to remember the precedence and associativity of every operator, at least not until you have spent a lot of time writing C++ code. Whenever you are uncertain, you can always add parentheses to make sure things execute in the sequence you want.

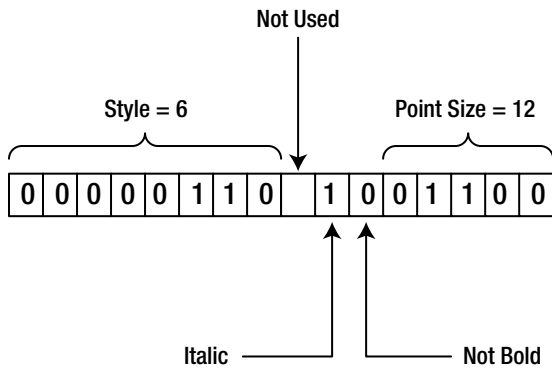
■ **Note** The C++ standard doesn't define the precedence of the operators directly, but it is determined by the syntax rules that are defined within the standard. In most instances it's easier to work out how a given expression will execute from operator precedence than from the syntax rules.

Bitwise Operators

As their name suggests, *bitwise operators* enable you to operate on an integer variable at the bit level. You can apply the bitwise operators to any type of integer, both signed and unsigned, including type `char`. However, they're usually applied to unsigned integer types. A typical application is to set individual bits in an integer variable. Individual bits are often used as *flags*, which is the term used to describe binary state indicators. You can use a single bit to store any value that has two states: on or off, male or female, true or false.

You can also use the bitwise operators to work with several items of information stored in a single variable. For instance, color values are usually recorded as three 8-bit values for the intensities of the red, green, and blue components in the color. These are typically packed into 3 bytes of a 4-byte word. The fourth byte is not wasted either; it usually contains a value for the transparency of the color. Obviously, to work with individual color components, you need to be able to separate out the individual bytes from a word, and the bitwise operators are just the tool for this.

Let's consider another example. Suppose you need to record information about fonts. You might want to store the style and the size of each font and whether it's bold or italic. You could pack all of this information into a 2-byte integer variable, as shown in Figure 3-1.



Using Bits to Store Font Data

Figure 3-1. Packing font data into 2 bytes

Here one bit records whether or not the font is italic—1 signifies italic and 0 signifies normal. Another bit specifies whether or not the font is bold. One byte selects one of up to 256 different styles. Five bits could record the point size up to 32. Thus, in one 16-bit word you have four separate pieces of data. The bitwise operators provide you with the means of accessing and modifying the individual bits and groups of bits from an integer very easily so they provide you with the means of assembling and disassembling the 16-bit word.

The Bitwise Shift Operators

The bitwise **shift operators** shift the contents of an integer variable by a specified number of bits to the left or right. These are used in combination with the other bitwise operators to achieve the kind of operations I described in the previous section. The `>>` operator shifts bits to the right, and the `<<` operator shifts bits to the left. Bits that fall off either end of the variable are lost.

All the bitwise operations work with integers of any type, but I'll use type `short`, which is usually 2 bytes, to keep the illustrations simple. Suppose you define and initialize a variable, `number`, with this statement:

```
unsigned short number {16387U};
```

As you saw in the previous chapter, you write unsigned literals with a letter `U` or `u` appended. You can shift the contents of this variable and store the result back in `number` with this statement:

```
unsigned short result {number << 2}; // Shift left two bit positions
```

The left operand of the left shift operator, `<<`, is the value to be shifted and the right operand specifies the number of bit positions by which the value is to be shifted. Figure 3-2 shows the effect.

Decimal 16,387 in binary is:

0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1

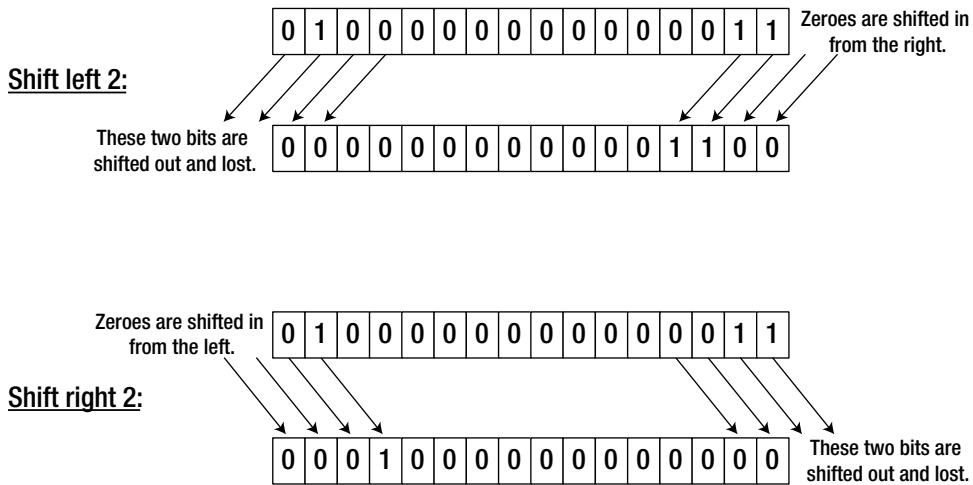


Figure 3-2. Shift operations

As you can see from Figure 3-2, shifting 16,387 two positions to the left produces the value 12. The rather drastic change in the value is the result of losing the high order bit. This statement shifts the value right 2 bit positions:

```
result = number >> 2;           // Shift right two bit positions
```

The result is 4,096 so shifting right two bits effectively divides the value by 4. As long as bits aren't lost, shifting n bits to the left is equivalent to multiplying by 2^n . In other words, it's equivalent to multiplying by $2n$. Similarly, shifting right n bits is equivalent to dividing by $2n$. But beware: As you saw with the left shift of `number`, if significant bits are lost, the result is nothing like what you would expect. However, this is no different from the "real" multiply operation. If you multiplied the 2-byte number by 4 you would get the same result, so shifting left and multiplying are still equivalent. The incorrect result arises because the result of the multiplication is outside the range of a 2-byte integer.

When you want to modify the original value of a variable using a shift operation, you can do so by using a `>>=` or `<<=` operator. For example:

```
number >>= 2;                     // Shift right two positions
```

This is equivalent to:

```
number = number >> 2;             // Shift right two positions
```

There's no confusion between these shift operators and the insertion and extraction operators for input and output. As far as the compiler is concerned, the meaning is clear from the context. If it isn't, the compiler will generate a message in most cases, but you do need to be careful. For example, to output the result of shifting `number` left by two bits, you could write:

```
std::cout << (number << 2);
```

The parentheses are essential here. Without them, the compiler will interpret the shift operator as a stream insertion operator so you won't get the result that you intended.

Shifting Signed Integers

You can apply the bitwise shift operators to signed and unsigned integers. However, the effect of the right shift operator on signed integer types depends on your compiler. In some cases, a right shift will introduce "0" bits at the left to fill vacated bit positions. In other cases, the sign bit is propagated so "1" bits fill the vacated bit positions to the left.

The reason for propagating the sign bit, where this occurs, is to maintain consistency between a right shift and a divide operation. I can illustrate this with a variable of type `signed char`, just to show how it works. Suppose you define value like this:

```
signed char value {-104};
```

Its binary value is 10011000. You can shift it two bits to the right with this operation:

```
value >>= 2;                // Result is 1110 0110
```

The binary result when the sign is propagated is shown in the comment. Two 0s are shifted out at the right end, and because the sign bit is 1, further 1s are inserted on the left. The decimal value of the result is -26, which is the same as if you had divided by 4, as you would expect. With operations on unsigned integer types, of course, the sign bit isn't propagated and 0s are inserted on the left.

As I said, what *actually* happens when you right-shift negative integers is implementation defined. Because for the most part you'll be using these operators for operating at the bit level — where maintaining the integrity of the bit pattern is important — you should always use unsigned integers to ensure that you avoid the high-order bit being propagated.

Logical Operations on Bit Patterns

The four bitwise operators that modify bits in an integer value are shown in Table 3-2.

Table 3-2. Bitwise Operators

Operator	Description
~	The <i>bitwise complement operator</i> is a unary operator that inverts the bits in its operand, so 1 becomes 0 and 0 becomes 1.
&	The <i>bitwise AND operator</i> ANDs corresponding bits in its operands. If the corresponding bits are both 1, then the resulting bit is 1, otherwise, it's 0.
^	The <i>bitwise exclusive OR operator</i> exclusive-ORs corresponding bits in its operands. If the corresponding bits are different, then the result is 1. If the corresponding bits are the same, the result is 0.
	The <i>bitwise OR operator</i> ORs corresponding bits in its operands. If either bit is 1, then the result is 1. If both bits are 0, then the result is 0.

The operators appear in Table 3-1 in order of precedence, so the bitwise complement operator has the highest precedence, and the bitwise OR operator the lowest. The shift operators << and >> are of equal precedence, and they're below the ~ operator but above the & operator.

Using the Bitwise AND

You'll typically use the bitwise AND operator to select particular bits or groups of bits in an integer value. Suppose you are using a 16-bit integer to store the point size, and the style of a font, and whether it is bold and/or italic, as I illustrated earlier in Figure 3-1. Suppose further that you want to define and initialize a variable to specify a 12-point, italic, style 6 font. In fact, the very same one illustrated in Figure 3-1. In binary, the style will be 00000110, the italic bit will be 1, the bold bit will be 0, and the size will be 01100. Remembering that there's an unused bit as well, you need to initialize the value of the font variable to the binary number 0000 0110 0100 1100. Because groups of four bits correspond to a hexadecimal digit, the easiest way to do this is to specify the initial value in hexadecimal notation:

```
unsigned short font {0x064C};           // Style 6, italic, 2 point
```

To work with the size, you need to extract it from the font variable; the bitwise AND operator will enable you to do this. Because bitwise AND only produces 1 bit when both bits are 1, you can define a value that will "select" the bits defining the size when you AND it with font. You need to define a value that contains 1s in the bit positions that you're interested in, and 0s in all the others. This kind of value is called a *mask*, and you can define such a mask with this statement:

```
unsigned short size_mask {0x1F};        // Mask is 0000 0000 0001 1111 to select size
```

The five low-order bits of font represent its size, so you set these bits to 1. The remaining bits are 0, so they will be discarded. (Binary 0000 0000 0001 1111 is hexadecimal 1F.)

You can now extract the point size from font with the statement:

```
unsigned short size {font & size_mask};
```

Where both corresponding bits are 1 in an & operation, the resultant bit is 1. Any other combination of bits results in 0. The values therefore combine like this:

font	0000 0110 0100 1100
size_mask	0000 0000 0001 1111
font & size_mask	0000 0000 0000 1100

I have shown the binary values in groups of four bits just to make it easy to identify the hexadecimal equivalent; it also makes it easier to see how many bits there are in total. The effect of the mask is to separate out the five rightmost bits, which represent the point size.

You can use the same mechanism to select the font style, but you'll also need to use a shift operator to move the style value to the right. You can define a mask to select the left eight bits as follows:

```
unsigned short style_mask {0xFF00};    // Mask is 1111 1111 0000 0000 for style
```

You can obtain the style value with this statement

```
unsigned short style {(font & style_mask) >> 8};
```

The effect of this statement is:

```
font           0000 0110 0100 1100
style_mask     1111 1111 0000 0000
font & style_mask  0000 0110 0000 0000
(font & style_mask) >> 8  0000 0000 0000 0110
```

You should be able to see that you could just as easily isolate the bits indicating italic and bold by defining a mask for each. Of course, you still need a way to test whether the resulting bit is 1 or 0, and you'll see how to do that in the next chapter.

Another use for the bitwise AND operator is to turn bits off. You saw previously that a 0 bit in a mask will produce 0 in the result. To just turn the italic bit off in `font` for example, you bitwise-AND `font` with a mask that has the italic bit as 0 and all other bits as 1. I'll show you the code to do this in the context of the bitwise OR operator, which is next.

■ **Note** I write binary numbers in the text with a space separating groups of 4 bits. This is just to make the numbers easier to read; binary numbers must not include spaces when you write them in code although you can use a single quote to separate groups of digits in any integer literal.

Using the Bitwise OR

You can use the bitwise OR operator for setting one or more bits to 1. Continuing with your manipulations of the `font` variable, it's conceivable that you would want to set the italic and bold bits on. You can define masks to select these bits with these statements:

```
unsigned short italic {0X40U};    // Seventh bit from the right
unsigned short bold {0X20U};      // Sixth bit from the right
```

This statement sets the bold bit to 1:

```
font |= bold;                    // Set bold
```

The bits combine like this:

```
font           0000 0110 0100 1100
bold           0000 0000 0010 0000
font | bold    0000 0110 0110 1100
```

Now `font` specifies that the font is bold as well as italic. Note that this operation will set the bit on regardless of its previous state. If it was on, it remains on.

You can also OR masks together to set multiple bits. The following statement sets both the bold and the italic bit:

```
font |= bold | italic;          // Set bold and italic
```

It's easy to fall into the trap of allowing language to make you select the wrong operator. Because you say "Set italic *and* bold" there's a temptation to use the `&` operator, but this would be wrong. ANDing the two masks would result in a value with all bits 0, so you wouldn't change anything.

As I said, you can use the `&` operator to turn bits off — you just need a mask that contains 0 at the bit position you want to turn off and 1 everywhere else. However, this raises the question of how best to specify such a mask. To specify it explicitly, you need to know how many bytes there are in the variable you want to change (not exactly convenient if you want the program to be in any way portable). However, you can obtain the mask that you want using the bitwise complement operator on the mask that you would use to turn the bit on. You can obtain the mask to turn bold off from the bold mask that turns it on:

```
bold      0000 0000 0010 0000
~bold     1111 1111 1101 1111
```

The effect of the complement operator is to flip each bit, 0 to 1 or 1 to 0. This will produce the result you're looking for, regardless of whether `bold` occupies 2, 4, or 8 bytes.

■ **Note** The bitwise complement operator is sometimes called the NOT operator, because for every bit it operates on, what you get is not what you started with.

Thus all you need to do to turn bold off is to bitwise-AND the complement of the bold mask with `font`. The following statement will do it:

```
font &= ~bold;           // Turn bold off
```

You can set multiple bits to 0 by combining several inverted masks using the `&` operator and bitwise-ANDing the result with the variable you want to modify:

```
font &= ~bold & ~italic; // Turn bold and italic off
```

This sets both the italic and bold bits to 0 in `font`. No parentheses are necessary here because `~` has a higher precedence than `&`. However, if you're ever uncertain about operator precedence, put parentheses in to express what you want. It certainly does no harm, and it really does good when they're necessary.

Using the Bitwise Exclusive OR

The bitwise exclusive OR operator is used much less frequently than the `&` and `|` operators, and there are few common examples of its use. An important application though, arises in the context of graphics programming. One way of creating the illusion of motion on the screen is to draw an object, erase it, and then redraw it in a new position. This process needs to be repeated very rapidly if you are to get smooth animation, and the erasing is a critical part. You don't want to erase and redraw the whole screen, as this is time consuming and the screen may flicker. Ideally you want to erase only the object or objects onscreen that you're moving. You can do this and get reasonably smooth animation by drawing using what is called *exclusive OR mode*.

Exclusive OR mode is based on the idea that once you've drawn an object in a given color, it will disappear if you redraw it in the background color. This is illustrated by the sequence in Figure 3-3.

Drawing Using Exclusive OR Mode

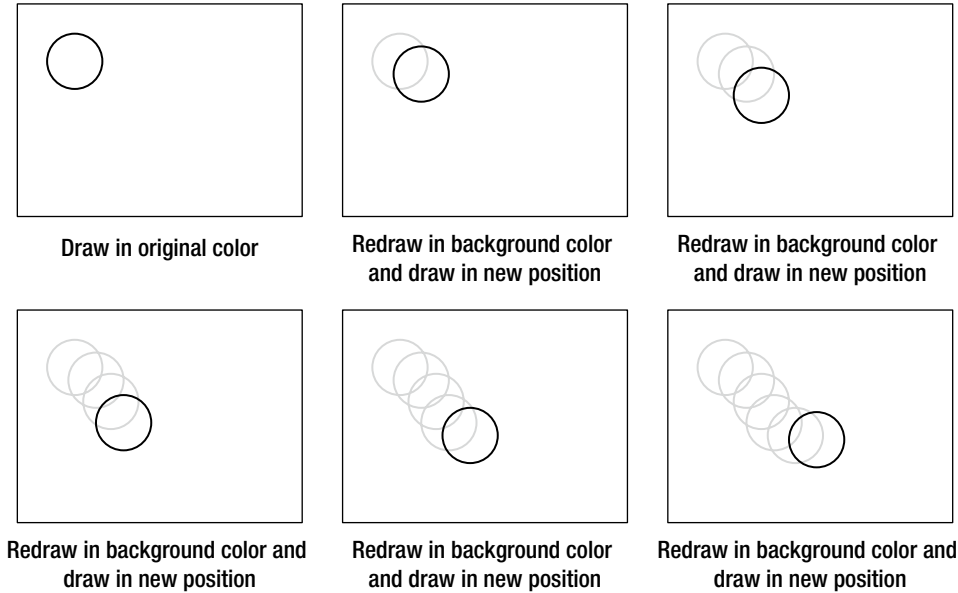


Figure 3-3. Drawing in exclusive OR mode

When you draw in exclusive OR mode, the color automatically alternates between the object color and the background color each time you draw the object. The key to achieving this is the application of the bitwise exclusive OR operator to alternate the colors rapidly and automatically. If you choose your color values suitably, you can flip between two different colors with repeated exclusive-OR operations. That sounds complicated, so let's see how it works by looking at an example.

Suppose you want to alternate between a foreground color (you'll use red), and a white background. As I noted earlier, color is often represented by three 8-bit values, corresponding to the intensities of red, blue, and green that are packed in a single 4-byte integer. By altering the proportions of red, blue, and green, you can get around 16 million different colors in the range from white to black and everything in between. A bright red would be `0xFF0000`, where the red component is set to its maximum and the intensities of the green and blue components are zero. In the same scheme, green would be `0x00FF00` and blue would be `0x0000FF`. White has equal, maximum components of red, blue, and green, so white is `0xFFFFFF`. You can therefore define variables representing red and white with the statements

```
unsigned int red {0xFF0000U};           // Color red
unsigned int white {0xFFFFFFFFU};       // Color white - RGB all maximum
```

You need a mask that you can use to switch the color back and forth between red and white. You'll also need a variable to store the drawing color:

```
unsigned int mask {red ^ white};        // Mask for switching colors
unsigned int draw_color {red};          // Drawing color - starts out red
```

The mask variable is initialized to the bitwise exclusive OR of the colors that you want to alternate, so it will be:

```
red                1111 1111 0000 0000 0000 0000
white              1111 1111 1111 1111 1111 1111
mask (which is red^white) 0000 0000 1111 1111 1111 1111
```

If you exclusive-OR mask with red you get white, and if you exclusive-OR mask with white you get red. This is a very useful result. This means that having drawn an object using the current color in `draw_color`, you can switch it to the other color with this statement:

```
draw_color ^= mask;           // Switch the drawing color
```

The effect of this when `draw_color` contains red is as follows:

```
draw_color         1111 1111 0000 0000 0000 0000
mask               0000 0000 1111 1111 1111 1111
draw_color ^ mask  1111 1111 1111 1111 1111 1111
```

Clearly, you've changed `draw_color` from red to white. Executing the same statement again will flip the color back to red:

```
draw_color         1111 1111 1111 1111 1111 1111
mask               0000 0000 1111 1111 1111 1111
draw_color ^ mask  1111 1111 0000 0000 0000 0000
```

As you can see, `draw_color` is back to red again. This technique works with any two colors, although of course it has nothing to do with colors in particular; you can use it to alternate between any pair of integer values.

It's relatively easy to see why this always works. The `^` operator, like the other binary bitwise operators, is commutative, which just means that the order of the operands doesn't matter. This implies that in the previous code fragments `red^mask` is the same as `red^red^white`, and `white^mask` is the same as `white^white^red`. Exclusive ORing two identical values results in all zeroes. Exclusive ORing a value of all zeroes with any value results in the same value. Thus both `red^red` and `white^white` produce all zeroes, so you can see why exclusive ORing mask with either color process flips the color value.

It's time we looked at some of this stuff in action. This example exercises bitwise operators:

```
// Ex3_01.cpp
// Using the bitwise operators
#include <iostream>
#include <iomanip>
using std::setw;

int main()
{
    unsigned int red {0xFF0000U};           // Color red
    unsigned int white {0xFFFFFU};          // Color white - RGB all maximum

    std::cout << std::hex                   // Hexadecimal output
               << std::setfill('0');        // Fill character 0
```

```

std::cout << "Try out bitwise AND and OR operators:";
std::cout << "\nInitial value red = " << setw(8) << red;
std::cout << "\nComplement ~red = " << setw(8) << ~red;

std::cout << "\nInitial value white = " << setw(8) << white;
std::cout << "\nComplement ~white = " << setw(8) << ~white;

std::cout << "\nBitwise AND red & white = " << setw(8) << (red & white);
std::cout << "\nBitwise OR red | white = " << setw(8) << (red | white);

std::cout << "\n\nNow try successive exclusive OR operations:";
unsigned int mask {red ^ white};
std::cout << "\nmask = red ^ white = " << setw(8) << mask;
std::cout << "\nmask ^ red = " << setw(8) << (mask ^ red);
std::cout << "\nmask ^ white = " << setw(8) << (mask ^ white);

unsigned int flags {0xFF};           // Flags variable
unsigned int bit1mask {0x1};         // Selects bit 1
unsigned int bit6mask {0x20};        // Selects bit 6
unsigned int bit20mask {0x80000};    // Selects bit 20

std::cout << "\n\nUse masks to select or set a particular flag bit:";
std::cout << "\nSelect bit 1 from flags : " << setw(8) << (flags & bit1mask);
std::cout << "\nSelect bit 6 from flags : " << setw(8) << (flags & bit6mask);
std::cout << "\nSwitch off bit 6 in flags : " << setw(8) << (flags &= ~bit6mask);
std::cout << "\nSwitch on bit 20 in flags : " << setw(8) << (flags |= bit20mask)
        << std::endl;
}

```

If you typed the code correctly, the output is:

```

Try out bitwise AND and OR operators:
Initial value:      red = 00ff0000
Complement:        ~red = ff00ffff
Initial value:      white = 00ffffff
Complement:        ~white = ff000000
Bitwise AND: red & white = 00ff0000
Bitwise OR: red | white = 00ffffff

Now try successive exclusive OR operations:
mask:  red ^ white = 0000ffff
      mask ^ red = 00ffffff
      mask ^ white = 00ff0000

Use masks to select or set a particular flag bit:
Select bit 1 from flags : 00000001
Select bit 6 from flags : 00000020
Switch off bit 6 in flags: 000000df
Switch on bit 20 in flags: 000800df

```

There's an `#include` directive for the `iomanip` header because the code uses manipulators to control the formatting of the output. You define variables `red` and `white` as unsigned integers and initialize them with hexadecimal color values.

It will be convenient to display the data as hexadecimal values and inserting `std::hex` in the output stream does this. The hex is modal so all subsequent integer output will be in hexadecimal format. It will be easier to compare output values if they have the same number of digits and leading zeroes. You can arrange for this by setting the fill character as 0 using the `std::setfill()` manipulator and ensuring the field width for each output value is the number of hexadecimal digits, which is 8. The `setfill()` manipulator is modal so it remains in effect until you reset it. The `std::setw()` manipulator is not modal; you have to insert it into the stream before each output value.

You combine `red` and `white` using the bitwise AND and OR operators with these statements:

```
std::cout << "\nBitwise AND red & white = " << setw(8) << (red & white);
std::cout << "\nBitwise OR red | white = " << setw(8) << (red | white);
```

The parentheses around the expressions are necessary here because the precedence of `<<` is higher than `&` and `|`. Without the parentheses, the statements wouldn't compile. If you check the output, you'll see that it's precisely as discussed. The result of ANDing two bits is 1 if both bits are 1; otherwise the result is 0. When you bitwise-OR two bits, the result is 1 unless both bits are 0.

Next, you create a mask to use to flip between the values `red` and `white` by combining the two values with the exclusive OR operator. The output for the value of `mask` shows that the exclusive OR of two bits is 1 when the bits are different and 0 when they're the same. By combining `mask` with either color values using exclusive OR, you obtain the other.

The last group of statements demonstrates using a mask to select a single bit from a group of flag bits. The mask to select a particular bit must have that bit as 1 and all other bits as 0. To select a bit from `flags`, you just bitwise-AND the appropriate mask with the value of `flags`. To switch a bit off, you bitwise-AND `flags` with a mask containing 0 for the bit to be switched off and 1 everywhere else. You can easily produce this by applying the complement operator to a mask with the appropriate bit set, and `bit6mask` is just such a mask. Of course, if the bit to be switched off was already 0, it would remain as 0.

Enumerated Data Types

You'll sometimes need variables that have a limited set of possible values that can be usefully referred to by name — the days of the week, for example, or the months of the year. An *enumeration* provides this capability. When you define an enumeration, you're creating a new type, so it's also referred to as an *enumerated data type*. Let's create an example using one of the ideas I just mentioned — a type for variables that can assume values corresponding to days of the week. You can define this as follows:

```
enum class Day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
```

This defines an enumerated data type called `Day`, and variables of this type can only have values from the set that appears between the braces, `Monday` through `Sunday`. If you try to set a variable of type `Day` to a value that isn't one of these values, the code won't compile. The symbolic names between the braces are called *enumerators*.

Each enumerator will be automatically defined to have a fixed integer value of type `int` by default. The first name in the list, `Monday`, will have the value 0, `Tuesday` will be 1, and so on through to `Sunday` with the value 6. You can define `today` as a variable of the enumeration type `Day` with the statement:

```
Day today {Day::Tuesday};
```

You use type `Day` just like any of the fundamental types. This definition for `today` initializes the variable with the value `Day::Tuesday`. When you reference an enumerator, it must be qualified by the type name.

To output the value of today, you must cast it to a numeric type because the standard output stream will not recognize the type `Day`:

```
std::cout << "today is " << static_cast<int>(today) << std::endl;
```

This statement will output “today is 1”.

By default, the value of each enumerator is one greater than the previous one, and by default the values begin at 0. You can make the implicit values assigned to enumerators start at a different integer value though. This definition of type `Day` has enumerator values to 1 through 7:

```
enum class Day {Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
```

Monday is explicitly specified as 1 and subsequent enumerators without explicit value will be 1 greater than the preceding enumerator. The enumerators don’t need to have unique values. You could define `Monday` and `Mon` as both having the value 1, for example, like this:

```
enum class Day {Monday = 1, Mon = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
```

You can now use either `Mon` or `Monday` as the first day of the week. A variable, `yesterday`, that you’ve defined as type `Day` could then be set with this statement:

```
yesterday = Day::Mon;
```

You can also define the value of an enumerator in terms of a previous enumerator. Throwing everything you’ve seen so far into a single example, you could define the type `Day` as follows:

```
enum class Day { Monday, Mon = Monday,
                Tuesday = Monday + 2,   Tues = Tuesday,
                Wednesday = Tuesday + 2, Wed = Wednesday,
                Thursday = Wednesday + 2, Thurs = Thursday,
                Friday = Thursday + 2,   Fri = Friday,
                Saturday = Friday + 2,   Sat = Saturday,
                Sunday = Saturday + 2,   Sun = Sunday
            };
```

Now variables of type `Day` can have values from `Monday` to `Sunday` and from `Mon` to `Sun`, and the matching pairs of enumerators correspond to the integer values 0, 2, 4, 6, 8, 10, and 12. The implication is that you can assign any integer values you like to the enumerators. Any enumerator other than the first that doesn’t have an explicit value assigned will have a value that is one greater than the previous enumerator in sequence. Values for enumerators must be *compile-time constants*; that is, constant expressions that the compiler can evaluate. Such expressions can only include literals, enumerators that have been defined previously, and variables that you’ve specified as `const`. You can’t use non-`const` variables, even if you’ve initialized them.

You can define variables when you define an enumeration type:

```
enum class Day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}
                yesterday{Day::Monday}, today{Day::Tuesday}, tomorrow{Day::Wednesday};
```

This defines and initializes the variables `yesterday`, `today`, and `tomorrow`, each of which are of type `Day`.

The enumerators can be an integer type that you choose, rather than the default type `int`. You can also assign explicit values to all the enumerators. For example, you could define this enumeration:

```
enum class Punctuation : char {Comma = ',', Exclamation = '!', Question='?'};
```

The type specification for the enumerators goes after the enumeration type name, and separated from it by a colon. You can specify any integral data type for the enumerators. The possible values for variables of type `Punctuation` are defined as char literals, and will correspond to the code values of the symbols. Thus the values of the enumerators are 44, 33, and 63, respectively in decimal, which also demonstrates that the values don't have to be in ascending sequence.

Here's an example that demonstrates some of the things you can do with enumerations:

```
// Ex3_02.cpp
// Operations with enumerations
#include <iostream>
#include <iomanip>
using std::setw;

int main()
{
    enum class Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday }
                    yesterday{ Day::Monday }, today{ Day::Tuesday },
                    tomorrow{ Day::Wednesday };

    Day poets_day{ Day::Friday };

    enum class Punctuation : char { Comma = ',', Exclamation = '!', Question = '?' };
    Punctuation ch{ Punctuation::Comma };

    std::cout << "yesterday's value is " << static_cast<int>(yesterday)
              << static_cast<char>(ch) << " but poets_day's is " << static_cast<int>(poets_day)
              << static_cast<char>(Punctuation::Exclamation) << std::endl;

    today = Day::Thursday;                // Assign a new ...
    ch = Punctuation::Question;           // ... enumerator values
    tomorrow = poets_day;                  // Copy enumerator value

    std::cout << "Is today's value(" << static_cast<int>(today)
              << ") the same as poets_day(" << static_cast<int>(poets_day)
              << ")" << static_cast<char>(ch) << std::endl;

    // ch = tomorrow;                      // Uncomment ...
    // tomorrow = Friday;                  // ... any of these ...
    // today = 6;                          // ... for an error.
}
```

The output is:

```
yesterday's value is 0, but poets_day's is 4!
Is today's value(3) the same as poets_day(4)?
```

I'll leave you to figure out why. Note the commented statements at the end of `main()`. They are all illegal operations. You should try them to see the compiler messages that result.

Old-Style Enumerations

The enumerations I have just described make obsolete the old syntax for enumerations. These are defined without using the `class` keyword. For example, the `Day` enumeration could be defined like this:

```
enum class Day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
```

With this syntax, the enumerators do not have to be qualified by the type name, so you can write:

```
Day today {Tuesday};
```

Although type `Day` is a distinct type, the value of the `today` variable is implicitly convertible to an integer so you can write it to a stream without converting it explicitly. All the enumerators can also be implicitly converted to type `int`. At this point you may be thinking that all this is goodness, but it isn't. The implicit convertibility of these enumeration types means that using them is not type safe. Your code will be less error prone if you stick to `enum class` enumeration types.

Synonyms for Data Types

You've seen how enumerations provide one way to define your own data types. The `typedef` keyword enables you to specify your own data type *name* as an alternative to another type name. Using `typedef`, you can define the type name `BigOnes` as being equivalent to the standard type `long` with the following statement:

```
typedef long BigOnes; // Defines BigOnes as a type alias
```

Of course, this isn't defining a new type. This just defines `BigOnes` as an alternative name for type `long`. You could define a variable `mynum` as type `long` with this statement:

```
BigOnes mynum {};
```

There's no difference between this definition and using the standard type name. You can still use the standard type name as well as the alias but it's hard to come up with a reason for using both.

There's a newer syntax for defining an alias for a type name that uses the `using` keyword. For example, you can define the type alias `BigOnes` like this:

```
using BigOnes = long; // Defines BigOnes as a type alias
```

Because you are just creating a synonym for a type that already exists, this may appear to be a bit superfluous. This isn't the case. A major use for this is to simplify code that involves complex type names. For example, a program might involve a type name such as `std::map<std::shared_ptr<Contact>, std::string>`. This can make the code look very obscure when the type is repeated often. You can avoid cluttering the code with this by defining a type alias:

```
using PhoneBook = std::map<std::shared_ptr<Contact>, std::string>;
```

Using `PhoneBook` in the code instead of the full type specification will make the code much more readable. Another use for a type alias is to provide flexibility in the data types used by a program that may need to be run on a variety of computers. Defining a type alias and using it throughout the code allows the actual type to be modified by just changing the definition of the alias.

The Lifetime of a Variable

All variables have a finite *lifetime*. They come into existence from the point at which you define them and at some point they are destroyed — at the latest, when your program ends. How long a particular variable lasts is determined by its *storage duration*. There are three different kinds of storage duration:

- Variables defined within a block that are not static have *automatic storage duration*. They exist from the point at which they are defined until the end of the block, which is the closing brace. They are referred to as *automatic variables*. Automatic variables are said to have *local scope* or *block scope*. All the variables you have created so far have been automatic variables.
- Variables defined using the `static` keyword have *static storage duration*. They are called *static variables*. Static variables exist from the point at which they are defined and continue in existence until the program ends.
- Variables for which you allocate memory at runtime have *dynamic storage duration*. They exist from the point at which you create them until you release their memory to destroy them. You'll learn how to create variables dynamically in Chapter 5.

Another property that variables have is *scope*. The scope of a variable is the part of a program in which the variable name is valid. Within a variable's scope, you can refer to it, set its value, or use it in an expression. Outside of its scope, you can't refer to its name. Any attempt to do so will result in a compiler error message. Note that a variable may still exist outside of its scope, even though you can't refer to it. You'll see examples of this situation a little later in this discussion.

■ **Note** Remember that the *lifetime* and *scope* of a variable are different things. Lifetime is the period of execution time over which a variable survives. Scope is the region of program code over which the variable name can be used. It's important not to get these two ideas confused.

Positioning Variable Definitions

You have great flexibility in where you define variables. The most important consideration is what scope the variables need to have. Beyond that, you should generally place a definition close to where the variable is first used. This makes your code easier for another programmer to understand. Let's look at variables where this is not the case.

Global Variables

You can define variables outside all of the functions in a program. Variables defined outside of all blocks and classes are called *globals* and have *global scope* (which is also called *global namespace scope*). This means that they're accessible in all the functions in the source file following the point at which they're defined. If you define them at the beginning of a source file, they'll be accessible throughout the file.

Global variables have *static storage duration* by default so they exist from the start of the program until execution of the program ends. If you don't initialize a global variable, it will be initialized with 0 by default. Initialization of global variables takes place before the execution of `main()` begins, so they're always ready to be used within any code that's within the variable's scope.

Figure 3-4 shows the contents of a source file, `Example.cpp` and illustrates the extent of the scope of each variable in the file.

Program file Example.cpp

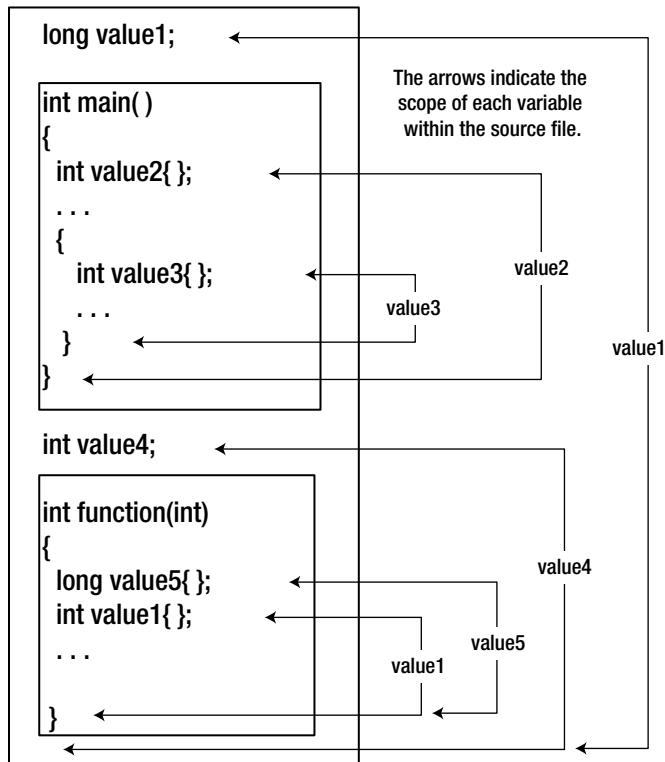


Figure 3-4. Variable scope

The variable `value1` at the beginning of the file is defined at global scope, as is `value4`, which appears after the definition of `main()`. They will be initialized with zero by default. Remember, only global variables have default initial values, not automatic variables. The lifetime of global variables is from the beginning of program execution, to when the program ends. Global variables have a scope that extends from the point at which they're defined to the end of the file. Even though `value4` exists when execution starts, it can't be referred to in `main()` because `main()` isn't within its scope. For `main()` to use `value4`, you would need to move the definition of `value4` to the beginning of the file.

The local variable called `value1` in `function()` will hide the global variable of the same name. If you use the name `value1` in the function, you are accessing the local automatic variable of that name. To access the global `value1`, you must qualify it with the scope resolution operator, `::`. Here's how you could output the values of the local and global variables that have the name `value1`:

```

std::cout << "Global value1 = " << ::value1 << std::endl;
std::cout << "Local value1 = " << value1 << std::endl;

```

Because global variables continue to exist for as long as the program is running, you might be wondering: "Why not make all variables global and avoid this messing about with local variables that disappear?" This sounds attractive at first, but there are serious disadvantages that completely outweigh any advantages. Real programs are composed of a large number of statements, a significant number of functions, and a great many variables. Declaring all at global

scope greatly magnifies the possibility of accidental, erroneous modification of a variable. It makes the job of naming them sensibly quite intractable. Global variables occupy memory for the duration of program execution so the program will require more memory than if you used local variables where the memory is reused. By keeping variables local to a function or a block, you can be sure they have almost complete protection from external effects. They'll only exist and occupy memory from the point at which they're defined to the end of the enclosing block, and the whole development process becomes much easier to manage.

Here's an example that shows aspects of global and automatic variables:

```
// Ex3_03.cpp
// Demonstrating scope, lifetime, and global variables
#include <iostream>
long count1 {999L};           // Global count1
double count2 {3.14};         // Global count2
int count3;                   // Global count3 - default initialization

int main()
{ // Function scope starts here
    int count1 {10};           // Hides global count1
    int count3 {50};           // Hides global count3
    std::cout << "Value of outer count1 = " << count1 << std::endl;
    std::cout << "Value of global count1 = " << ::count1 << std::endl;
    std::cout << "Value of global count2 = " << count2 << std::endl;

    {                          // New block scope starts here...
        int count1 {20};       // This is a new variable that hides the outer count1
        int count2 {30};       // This hides global count2
        std::cout << "\nValue of inner count1 = " << count1 << std::endl;
        std::cout << "Value of global count1 = " << ::count1 << std::endl;
        std::cout << "Value of inner count2 = " << count2 << std::endl;
        std::cout << "Value of global count2 = " << ::count2 << std::endl;

        count1 = ::count1 + 3;  // This sets inner count1 to global count1+3
        ++::count1;             // This changes global count1
        std::cout << "\nValue of inner count1 = " << count1 << std::endl;
        std::cout << "Value of global count1 = " << ::count1 << std::endl;
        count3 += count2;       // Increments outer count3 by inner count2;
    }                           // ...and ends here.

    std::cout << "\nValue of outer count1 = " << count1 << std::endl
              << "Value of outer count3 = " << count3 << std::endl;
    std::cout << "Value of global count3 = " << ::count3 << std::endl;

    std::cout << count2 << std::endl; // This is global count2
} // Function scope ends here
```

The output from this example is:

```
Value of outer count1 = 10
Value of global count1 = 999
Value of global count2 = 3.14

Value of inner count1 = 20
Value of global count1 = 999
Value of inner count2 = 30
Value of global count2 = 3.14

Value of inner count1 = 1002
Value of global count1 = 1000

Value of outer count1 = 10
Value of outer count3 = 80
Value of global count3 = 0
3.14
```

I've duplicated names in this example to illustrate what happens — it's not a good approach to programming. Doing this kind of thing in a real program is confusing and unnecessary, and results in code that is error prone.

There are three variables defined at global scope, `count1`, `count2`, and `count3`. These exist as long as the program continues to execute, but the names will be masked by local variables with the same name. The first two statements in `main()` define two integer variables, `count1` and `count3`, with initial values of 10 and 50, respectively. Both variables exist from this point until the closing brace at the end of `main()`. The scope of these variables also extends to the closing brace at the end of `main()`. Because the local `count1` hides the global `count1`, you must use the scope resolution operator to access the global `count1` in the output statement in the first group of output lines. Global `count2` is accessible just by using its name.

The second opening brace starts a new block. `count1` and `count2`, are defined within this block with values 20 and 30, respectively. `count1` here is different from the `count1` in the outer block, which still exists, but its name is masked by the second `count1` and is not accessible here; global `count1` is also masked but is accessible using the scope resolution operator. The global `count2` is masked by the local variable with that name. Using the name `count1` following the definition in the inner block refers to the `count1` defined in that block.

The first line of the second block of output is the value of the `count1` defined in the inner scope—that is, inside the inner braces. If it was the outer `count1`, the value would be 10. The next line of output corresponds to the global `count1`. The following line of output contains the value of local `count2` because you are using just its name. The last line in this block outputs global `count2` by using the `::` operator.

The statement assigning a new value to `count1` applies to the variable in the inner scope, because the outer `count1` is hidden. The new value is the global `count1` value plus 3. The next statement increments the global `count1` and the following two output statements confirm this. The `count3` that was defined in the outer scope is incremented in the inner block without any problem because it is not hidden by a variable with the same name. This shows that variables defined in an outer scope are still accessible in an inner scope as long as there is no variable with the same name defined in the inner scope.

After the brace ending the inner scope, `count1` and `count2` that are defined in the inner scope cease to exist. Their lifetime has ended. Local `count1` and `count3` still exist in the outer scope, and their values are displayed in the first two lines in the last group of output. This demonstrates that `count3` was indeed incremented in the inner scope. The last line of output corresponds to the global `count3` value.

Static Variables

It's conceivable that you might want to define a variable that you can access locally within a block, and that continues to exist after exiting the block in which it is defined. In other words, you need a variable with block scope, but with static storage duration. The `static` keyword enables you to do just this, and the value of it will become more apparent when you learn about functions in Chapter 8.

A variable that you specify as `static` will continue to exist for the life of a program, even though it's defined within a block and is only available from within that block (or its sub-blocks). It still has block scope, but it has static storage duration. To define a static variable called `count`, you would write

```
static int count;
```

Variables with static storage duration are always initialized to zero by default if you don't provide an initial value so `count` will be initialized with 0. Remember that this is *not* the case with automatic variables. If you don't initialize an automatic variable, it will contain a junk value.

External Variables

You saw in Chapter 1 that programs usually consist of several source files. In a program that consists of more than one source file, you may need to access a global variable in one source file that is defined in another. The `extern` keyword allows you to do this. Suppose you have a program file that contains the following:

```
// File1.cpp
int shared_value {100};           // Global variable

// Other program code ...
```

When code in another source file, `File2.cpp`, needs to access the global `shared_value` variable that is defined in `File1.cpp`, you can arrange for this as follows:

```
// File2.cpp
extern int shared_value;           // Declare variable to be external

int main()
{
    int local_value {shared_value + 10};
    // Plus other code...
}
```

The first statement in `File2.cpp` declares `shared_value` to be external, so this is a *declaration* of the variable, not a *definition*. The reference to `shared_value` in `main()` is to the variable defined in the first file, `File1.cpp`. The linker establishes the connection between the `extern` declaration for `shared_value` in `File2.cpp` and the global variable definition in `File1.cpp`.

You are not defining a variable in an `extern` statement; you are simply stating that it is defined elsewhere so you must not specify an initializing value. If you do specify an initial value, the `extern` keyword will be ignored. For example, suppose you wrote this statement in `File2.cpp`:

```
extern int shared_value {10};      // Wrong! Not an external declaration.
```

The `shared_value` variable here is a new global variable defined in `File2.cpp`, because the `extern` keyword is ignored as a consequence of the initialization.

Summary

This chapter introduced operator precedence and associativity. You don't need to memorize this for all operators but you need to be conscious of it when writing code. Always use parentheses if you are unsure about precedence. The type-safe enumerations type are very useful for representing fixed sets of values, especially those that have names, such as days of the week or suits in a pack of playing cards. The bitwise operators are necessary when you are working with flags - single bits that signify a state. These arise surprisingly relatively - when dealing with file input and output for example. The bitwise operators are also essential when you are working with values packed into a single variable. The essentials of what you've learned in this chapter are:

- By default, a variable defined within a block is automatic, which means that it only exists from the point at which it is defined to the end of the block in which its definition appears, as indicated by the closing brace of the block that encloses its definition.
- You can specify a variable as *static*, in which case it continues to exist for the life of the program. However, it can only be *accessed* within the scope in which it was defined. If you don't initialize a static variable, it will be initialized to 0 by default.
- Variables can be defined outside of all the blocks in a program, in which case they have global namespace scope and static storage duration by default. Variables with global scope are accessible from anywhere within the program file that contains them, following the point at which they're defined, except where a local variable exists with the same name as the global variable. Even then, they can still be reached by using the scope resolution operator (`::`).
- The `typedef` keyword allows you to define aliases for other types. You can also use the `using` keyword to define type aliases.
- The `extern` keyword enables you to identify the name of a global variable that is defined in another source file.

EXERCISES

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Apress website (www.apress.com/source-code/), but that really should be a last resort.

Exercise 3-1. Create a program that prompts for input of an integer and store it as a type `unsigned int`. Invert all the bits in the value and store the result. Output the original value, the value with the bits inverted, and the inverted value plus 1, each in hexadecimal representation on one line, and in decimal representation on the line below. The output values on the two lines should be right aligned in a suitable field width, hexadecimal values should have leading zeroes so 8 hexadecimal digits always appear. Corresponding values on the two output lines should align.

Exercise 3-2. Write a program to calculate how many square boxes can be contained in a single layer on a rectangular shelf, with no overhang. The dimensions of the shelf in feet and the dimension of a side of the box in inches are read from the keyboard. Use variables of type `double` for the length and depth of the shelf and type `int` for the length of the side of a box. Define and initialize an integer constant to convert from feet to inches. Calculate the number of boxes that the shelf can hold in a single layer type `long` and output the result.

Exercise 3-3. Without running it, can you work out what the following code snippet will produce as output?

```
unsigned int k {430U};
unsigned int j {(k >> 4) & ~(~0 << 3)};
std::cout << j << std::endl;
```

Exercise 3-4. Write a program to read four characters from the keyboard and pack them into a single integer variable. Display the value of this variable as hexadecimal. Unpack the 4 bytes of the variable and output them in reverse order, with the low-order byte first.

Exercise 3-5. Write a program that prompts for two integer values to be entered and store them in integer variables, *a* and *b* say. Swap the values of *a* and *b* *without* using a third variable. Output the values of *a* and *b*.

Exercise 3-6. Write a program that defines an enum class of type *Color* where the enumerators are Red, Green, Yellow, Purple, Blue, Black, and White. Define the type for enumerators as an unsigned integer type and arrange for the integer value of each enumerator to be the RGB combination for the color it represents. Create variables of type *Color* initialized with enumerators for yellow, purple, and green. Access the enumerator value and extract and output the RGB components as separate values.



Making Decisions

Decision-making is fundamental to any kind of computer programming. It's one of the things that differentiates a computer from a calculator. It means altering the sequence of execution depending on the result of a comparison. In this chapter, you'll explore how to make choices and decisions. This will allow you to validate program input and write programs that can adapt their actions depending on the input data. Your programs will be able to handle problems where logic is fundamental to the solution. By the end of this chapter, you will have learned:

- How to compare data values
- How to alter the sequence of program execution based on the result of a comparison
- What logical operators and expressions are, and how you apply them
- How to deal with multiple-choice situations

Comparing Data Values

To make decisions, you need a mechanism for comparing things, and there are several kinds of comparisons. For instance, a decision such as, "If the traffic signal is red, stop the car," involves a comparison for equality. You compare the color of the signal with a reference color, red, and if they are equal, you stop the car. On the other hand, a decision such as, "If the speed of the car exceeds the limit, slow down," involves a different relationship. Here you check whether the speed of the car is greater than the current speed limit. Both of these comparisons are similar in that they result in one of two values: they are either *true* or *false*. This is precisely how comparisons work in C++.

You can compare data values using some new operators called **relational operators**. Table 4-1 lists the six operators for comparing two values.

Table 4-1. *Relational Operators*

Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

■ **Caution** The equal to operator, `==`, has two successive equal signs. It's a very common mistake to use one equal sign instead of two to compare for equality. This will not necessarily result in a warning message from the compiler because the expression may be valid but just not what you intended, so you need to take particular care to avoid this error.

Each of these operators compares two values and results in a value of type `bool`; there are only two possible `bool` values, `true` and `false`. `true` and `false` are keywords and are literals of type `bool`. They are sometimes called *Boolean literals* (after George Boole, the father of Boolean algebra).

If you cast `true` to an integer type, the result will be 1; casting `false` to an integer results in 0. You can also convert numerical values to type `bool`. Zero converts to `false`, and any nonzero value converts to `true`. When you have a numerical value where a `bool` value is expected, the compiler will insert an implicit conversion to convert the numerical value to type `bool`. This is very useful in decision-making code.

You create variables of type `bool` just like other fundamental types. Here's an example:

```
bool isValid {true}; // Define, and initialize a logical variable
```

This defines the variable `isValid` as type `bool` with an initial value of `true`.

Applying the Comparison Operators

You can see how comparisons work by looking at a few examples. Suppose you have integer variables `i` and `j`, with values 10 and -5 respectively. Consider the following expressions:

```
i > j          i != j          j > -8          i <= j + 15
```

All of these expressions evaluate to `true`. Note that in the last expression, the addition, `j + 15`, executes first because `+` has a higher precedence than `<=`.

You could store the result of any of these expressions in a variable of type `bool`. For example:

```
isValid = i > j;
```

If `i` is greater than `j`, `true` is stored in `isValid`, otherwise `false` is stored. You can compare values stored in variables of character types, too. Assume that you define the following variables:

```
char first {'A'};
char last {'Z'};
```

You can write comparisons using these variables:

```
first < last          'E' <= first          first != last
```

Here you are comparing code values. The first expression checks whether the value of `first`, which is `'A'`, is less than the value of `last`, which is `'Z'`. This is always `true`. The result of the second expression is `false`, because the code value for `'E'` is greater than the value of `first`. The last expression is `true`, because `'A'` is definitely not equal to `'Z'`.

You can output `bool` values just as easily as any other type —here’s an example that shows how they look by default:

```
// Ex4_01.cpp
// Comparing data values
#include <iostream>

int main()
{
    char first {};           // Stores the first character
    char second {};          // Stores the second character

    std::cout << "Enter a character: ";
    std::cin >> first;

    std::cout << "Enter a second character: ";
    std::cin >> second;

    std::cout << "The value of the expression " << first << '<' << second
                << " is: " << (first < second) << std::endl;
    std::cout << "The value of the expression " << first << "==" << second
                << " is: " << (first == second) << std::endl;
}
```

Here’s an example of output from this program with my compiler:

```
Enter a character: ?
Enter a second character: H
The value of the expression ?<H is: 1
The value of the expression ?==H is: 0
```

The prompting for input and reading of characters from the keyboard is standard stuff that you have seen before. Note that the parentheses around the comparison expressions in the output statement *are* necessary here. If you omit them, the expressions don’t mean what you think they mean and the compiler outputs an error message. The expressions compare the first and second characters that the user entered. From the output you can see that the value `true` is displayed as 1, and the value `false` as 0. These are the default representations for `true` and `false`. You can make `bool` values output as `true` and `false` using the `std::boolalpha` manipulator. Just add this statement before any of the output statements:

```
std::cout << std::boolalpha;
```

If you compile and run the example again, you get `bool` values displayed as `true` or `false`. To return output of `bool` values to the default setting, insert the `std::noboolalpha` manipulator into the stream.

Comparing Floating Point Values

Of course, you can also compare floating-point values. Let's consider some slightly more complicated numerical comparisons. First, define variables with the following statements:

```
int i {-10};
int j {20};
double x {1.5};
double y {-0.25E-10};
```

Now consider the following logical expressions:

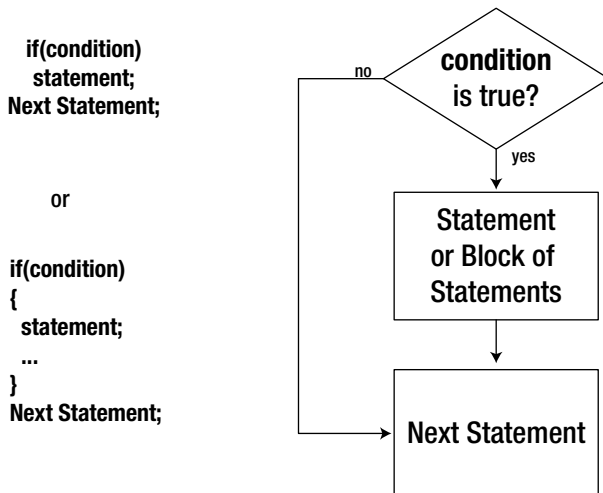
```
-1 < y          j < (10 - i)      2.0*x >= (3 + y)
```

The comparison operators are all of lower precedence than the arithmetic operators so none of the parentheses is strictly necessary but they do help make the expressions clearer. The first comparison evaluates to **true**, because y has a very small negative value (-0.000000000025), which is greater than -1 . The second comparison results in **false**, because the expression $10 - i$ has the value 20 , which is the same as j . The third expression is **true**, because $3 + y$ is slightly less than 3 .

You can use relational operators to compare values of any of the fundamental types. When you learn about classes you'll see how you can arrange for the comparison operators to work with types that you define, too. All you need now is a way to use the result of a comparison to modify the behavior of a program. Let's look into that immediately.

The if Statement

The basic **if** statement enables you to choose to execute a single statement, or a block of statements when a given condition is **true**. Figure 4-1 illustrates how this works.



The statement or block of statements that follows the **if** is only executed if **condition** is **true**.

Figure 4-1. Logic of the simple **if** statement

Here is an example of an if statement that tests the value of a char variable, letter:

```
if(letter == 'A')
    std::cout << "The first capital, alphabetically speaking.\n"; // Only if letter equals 'A'

std::cout << "This statement always executes.\n";
```

If letter has the value 'A', the condition is true and these statements produce the following output:

```
The first capital, alphabetically speaking.
This statement always executes.
```

If the value of letter is not equal to 'A', only the second line appears in the output. You put the condition to be tested between parentheses immediately following the keyword, if. Notice the position of the semicolon (;). It goes after the statement following if and the condition between the parentheses. A semicolon (;) must not appear after the condition in parentheses, because if and the condition are bound with the statement or block that follows. They cannot exist by themselves.

The statement following the if is indented to indicate that it only executes as a result of the condition being true. The indentation is not necessary for the program to compile, but it does help you recognize the relationship between the if condition and the statement that depends on it. Sometimes, you will see simple if statements written on a single line:

```
if(letter == 'A') std::cout << "The first capital, alphabetically speaking\n";
```

You could extend code fragment to change the value of letter if it contains the value 'A':

```
if(letter == 'A')
{
    std::cout << "The first capital, alphabetically speaking.\n";
    letter = 'a';
}

std::cout << "This statement always executes.\n";
```

All the statements in the block will be executed when the if condition is true. Without the braces, only the first statement would be the subject of the if, and the statement assigning the value 'a' to letter would always be executed. Of course, each of the statements in the block are terminated by a semicolon but no semicolon is necessary after the closing brace of the block. You can have as many statements as you like within the block; you can even have nested blocks. Because letter has the value 'A', both statements within the block will be executed so its value will be changed to 'a' after the same message as before is displayed. Neither of these statements execute if the condition is false. Of course, the statement following the block always executes.

Let's try out an if statement for real. This program will range check the value of an integer entered from the keyboard:

```
// Ex4_02.cpp
// Using an if statement
#include <iostream>

int main()
{
    std::cout << "Enter an integer between 50 and 100: ";
```



```

int value {};
std::cin >> value;

if(value < 50)
    std::cout << "The value is invalid - it is less than 50." << std::endl;

if(value > 100)
    std::cout << "The value is invalid - it is greater than 100." << std::endl;

std::cout << "You entered " << value << std::endl;
}

```

The output depends on the value that you enter. For a value between 50 and 100, the output will be something like the following:

```

Enter an integer between 50 and 100: 77
You entered 77

```

Outside the range 50 to 100, a message indicating that the value is invalid will precede the output showing the value. If it is below 50, the output will be:

```

Enter an integer between 50 and 100: 27
The value is invalid - it is less than 50.
You entered 27

```

If the value is greater than 100, the output will be similar.

After prompting for, and reading a value, the first if statement checks whether the value entered is below 50:

```

if(value < 50)
    std::cout << "The value is invalid - it is less than 50." << std::endl;

```

The output statement is executed only when the if condition is true, which is when value is less than 50. The next if statement checks the upper limit in essentially the same way and outputs a message when it is exceeded. Finally the last output statement is always executed and this outputs the value. Of course, checking for the upper limit being exceeded when the value is below the lower limit is superfluous. You could arrange for the program to end immediately if the value entered is below the lower limit, like this:

```

if(value < 50)
{
    std::cout << "The value is invalid - it is less than 50." << std::endl;
    return 0; // Ends the program
}

```

You could do the same with the if statement that checks the upper limit. Then you would only get the last output statement executed when the value entered is within bounds. You can have as many return statements in a function as you need although if there are a lot, it may be sign that you could improve the code by doing things differently.

Nested if Statements

The statement that executes when the condition in an if statement is true can itself be an if statement. This arrangement is called a nested if. The condition of the inner if is only tested if the condition for the outer if is true. An if that is nested inside another can also contain a nested if. You can nest ifs to whatever depth you require. I'll demonstrate the nested if with an example that tests whether a character entered is alphabetic. Although this example is a perfectly reasonable use of a nested if, it has some built-in assumptions that would be best avoided; see if you can spot the problem:

```
// Ex4_03.cpp
// Using a nested if
#include <iostream>

int main()
{
    char letter {};                // Store input here
    std::cout << "Enter a letter: "; // Prompt for the input
    std::cin >> letter;

    if(letter >= 'A')
    {                               // Letter is 'A' or larger
        if(letter <= 'Z')
        {                           // letter is 'Z' or smaller
            std::cout << "You entered an uppercase letter." << std::endl;
            return 0;
        }
    }

    if(letter >= 'a')               // Test for 'a' or larger
    {                               // letter is >= 'a' and <= 'z'
        if(letter <= 'z')
        {
            std::cout << "You entered a lowercase letter." << std::endl;
            return 0;
        }
    }
    std::cout << "You did not enter a letter." << std::endl;
}
```

Here's some typical output:

```
Enter a letter: H
You entered an uppercase letter.
```

After creating the char variable `letter` with initial value zero, the program prompts you to enter a letter. The if statement that follows checks whether the character entered is 'A' or larger. If `letter` is greater than or equal to 'A', the nested if that checks for the input being 'Z' or less executes. If it is 'Z' or less, you conclude that it is an uppercase letter and display a message. You are done at this point so you execute a return statement to end the program.

The next if, using essentially the same mechanism as the first, checks whether the character entered is lowercase, displays a message, and returns. You probably noticed that the test for a lowercase character contains only one pair of braces, whereas the uppercase test has two. The code block between the braces belongs to the inner if here. In fact, both sets of statements work as they should — remember that `if(condition){...}` is effectively a single

statement and does not need to be enclosed within more braces. However, the extra braces do make the code clearer, so it's a good idea to use them. Finally, like the uppercase test, this code contains implicit assumptions about the order of codes for lowercase letters.

The output statement following the last `if` block only executes when the character entered is not a letter, and it displays a message to that effect. You can see that the relationship between the nested `ifs` and the output statement is much easier to follow because of the indentation. Indentation is generally used to provide visual cues to the logic of a program.

These nested `ifs` have two built-in assumptions about the codes that are used to represent alphabetic characters. First, they assume that the letters A to Z are represented by a set of codes where the code for 'A' is the minimum and the code for 'Z' is the maximum. Second, they assume that the codes for the uppercase letters are contiguous, so no nonalphabetic characters lie between the codes for 'A' and 'Z'. It is not a good idea to build these kinds of assumptions into your code, because it limits the portability of your program. You'll see how you can avoid making these assumptions in a moment.

This program illustrates how a nested `if` works, but it is not a good way to test for characters. Using the Standard Library, you can write the program so that it works independently of the character coding.

Code-Neutral Character Handling

The locale Standard Library header provides a wide range of functions for classifying and converting characters. These functions are listed in Table 4-2. In each case, you pass the function a variable or a literal that is the character to be tested. The parameter is specified for each function in the table as type `int`. The compiler will arrange for the character that you pass to the function to be converted to type `int` if necessary.

Table 4-2. *Functions for Classifying Characters*

Function	Operation
<code>isupper(int c)</code>	Tests whether or not <code>c</code> is an uppercase letter, by default 'A' to 'Z'.
<code>islower(int c)</code>	Tests whether or not <code>c</code> is a lowercase letter, by default 'a' to 'z'.
<code>isalpha(int c)</code>	Tests whether or not <code>c</code> is an upper- or lowercase letter.
<code>isdigit(int c)</code>	Tests whether or not <code>c</code> is a digit, 0 to 9.
<code>isxdigit(int c)</code>	Tests whether or not <code>c</code> is a hexadecimal digit, 0 to 9, 'a' to 'f', or 'A' to 'F'.
<code>isalnum(int c)</code>	Tests whether or not <code>c</code> is a letter or a digit (i.e., an alphanumeric character).
<code>isspace(int c)</code>	Tests whether or not <code>c</code> is whitespace, which can be a space, a newline, a carriage return, a form feed, or a horizontal or vertical tab.
<code>isctrl(int c)</code>	Tests whether or not <code>c</code> is a control character.
<code>isprint(int c)</code>	Tests whether or not <code>c</code> is a printable character, which can be an upper- or lowercase letter, a digit, a punctuation character, or a space.
<code>isgraph(int c)</code>	Tests whether or not <code>c</code> is a graphic character, which is any printable character other than a space.
<code>ispunct(int c)</code>	Tests whether or not <code>c</code> is a punctuation character, which is any printable character that's not a letter or a digit. This will be either a space or one of the following: _ { } [] # () < > % : ; . ? * + - / ^ & ~ ! = , \ " ' .

Each of these functions returns a value of type `int`. The value will be non-zero (`true`) if the character is of the type being tested for, and 0 (`false`) if it isn't. You may be wondering why these functions don't return a `bool` value, which would make much more sense. The reason they don't return a `bool` value is that they originate from the C standard library and predate type `bool` in C++.

The `locale` header provides the two functions shown in Table 4-3 for converting between upper- and lowercase characters. The result will be returned as type `int` so you need to explicitly cast it if you want to store it as type `char` for instance.

Table 4-3. *Functions for Converting Characters*

Function	Operation
<code>tolower(int c)</code>	If <code>c</code> is uppercase, the lowercase equivalent is returned; otherwise <code>c</code> is returned.
<code>toupper(int c)</code>	If <code>c</code> is lowercase, the uppercase equivalent is returned; otherwise <code>c</code> is returned.

You could use these functions to implement the previous example without any assumptions about the character coding. The character codes in different environments are always taken care of by the standard library functions. They also make the code simpler:

```
if(isupper(letter))
{
    std::cout << "You entered an uppercase letter." << std::endl;
    return 0;
}

if(islower(letter))
{
    std::cout << "You entered a lowercase letter." << std::endl;
    return 0;
}
```

Note that all these character testing functions, except for `isdigit()` and `isxdigit()`, test the argument in the context of the current locale. A locale determines the national or cultural character set and data representations such as currency and dates that are in effect.

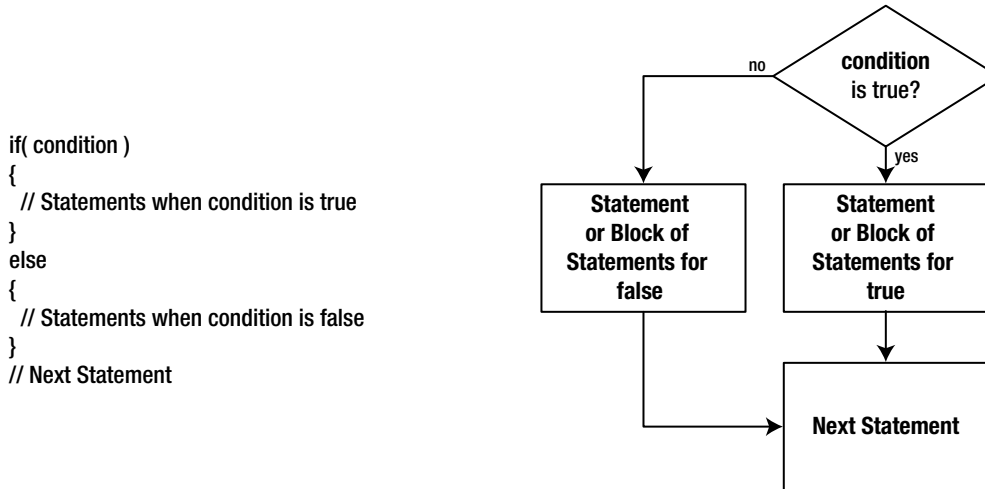
The `locale` header provides for much more extensive capabilities for working with locale-dependent data, including a set of character classification functions in the `std` namespace with names the same as the functions I have described. These functions require a second argument that is a `locale` object that identifies the local to be in effect for the function; they also return a `bool` value. A detailed discussion of locales and the `locale` type is outside the scope of this book.

■ **Note** The `cctype` and the `cwctype` headers that are inherited from C are also part of the C++ Standard Library. The `cctype` header declares the same classification and conversion functions that I have described for the `locale` header. The `cwctype` header declares an equivalent set of functions with slightly different names that work with characters of type `wchar_t`.

The if-else Statement

The if statement that you have been using executes a statement or block if the condition specified is true. Program execution then continues with the next statement in sequence. Of course, you may want to execute one block of statements when the condition is true, and another set when the condition is false. An extension of the if statement called an if-else statement allows this.

The if-else combination provides a choice between two options. Figure 4-2 shows its general logic.



One of the two blocks in an if-else statement is always executed.

Figure 4-2. The if-else statement logic

The flowchart in Figure 4-2 shows the sequence in which statements execute, depending on whether the if condition is true or false. You can always use a block of statements wherever you can put a single statement. This allows any number of statements to be executed for each option in an if-else statement.

You could write an if-else statement that would report whether the character stored in the char variable `letter` was alphanumeric:

```

if(std::isalnum(letter))
{
    std::cout << "It is a letter or a digit." << std::endl;
}
else
{
    std::cout << "It is neither a letter nor a digit." << std::endl;
}
  
```

This uses the `isalnum()` function from the `locale` header you saw earlier. If `letter` contains a letter or a digit, `isalnum()` returns a positive integer. This will be implicitly converted to a bool value, which will be true, so the first message is displayed. If `letter` contains other than a letter or a digit, `isalnum()` returns 0, which converts to false so the output statement after `else` executes. The braces are not mandatory here because they contain single statements but it's clearer if you put them in. The block following the `else` keyword is written without a semicolon appended,

just like the `if` part of the statement. The indentation in the blocks is a visible indicator of the relationship between various statements. You can clearly see which statement is executed to produce a true result and which is executed for false. You should always indent the statements in your programs to show their logical structure.

Here's an example of using `if-else` with a numerical value:

```
// Ex4_04.cpp
// Using the if-else
#include <iostream>

int main()
{
    long number {}; // Stores input
    std::cout << "Enter an integer less than 2 billion: ";
    std::cin >> number;

    if(number % 2L) // Test remainder after division by 2
    { // Here if remainder is 1
        std::cout << "Your number is odd." << std::endl;
    }
    else
    { // Here if remainder is 0
        std::cout << "\nYour number is even." << std::endl;
    }
}
```

Here's an example of output from this program:

```
Enter an integer less than 2 billion: 123456
Your number is even.
```

After reading the input into `number`, the program tests this value in the `if` condition. This is an expression that produces the remainder that results from dividing `number` by 2. The remainder will be 1 if `number` is odd, or 0 if it even, and these values convert to true and false respectively. Thus if the remainder is 1, the `if` condition is true and the statement in the block immediately following the `if` executes. If the remainder is 0, the `if` condition is false, so the statement in the block following the `else` keyword executes.

You could specify the `if` condition as `number % 2L == 0L`, in which case the sequence of blocks would need to be reversed because this expression evaluates to true when `number` is even.

Nested if-else Statements

You have already seen that you can nest `if` statements within `if` statements. You have no doubt anticipated that you can also nest `if-else` statements within `ifs`, `ifs` within `if-else` statements, and `if-else` statements within other `if-else` statements. This provides you with plenty of versatility (and considerable room for confusion), so let's look at a few examples. Taking the first case first, an example of an `if-else` nested within an `if` might look like the following:

```
if(coffee == 'y')
    if(donuts == 'y')
        std::cout << "We have coffee and donuts." << std::endl;
    else
        std::cout << "We have coffee, but not donuts." << std::endl;
```

This would be better written with braces but it's easier to make the point I want to make without. `coffee` and `donuts` are variables of type `char` that can have the value `'y'` or `'n'`. The test for `donuts` only executes if the result of the test for `coffee` is `true`, so the messages reflect the correct situation in each case. The `else` belongs to the `if` that tests for `donuts`. However, it is easy to get this confused.

If you write much the same thing, but with incorrect indentation, you can be trapped into the wrong conclusion about what happens here:

```
if(coffee == 'y')
    if(donuts == 'y')
        std::cout << std::endl
            << "We have coffee and donuts.";
else
    std::cout << "We have no coffee..." << std::endl; // This is indented incorrectly...
// ...Wrong!
```

The indentation now misleadingly suggests that this is an `if` nested within an `if-else`, which is not the case. The first message is correct, but the output as a consequence of the `else` executing is quite wrong. This statement only executes if the test for `coffee` is `true`, because the `else` belongs to the test for `donuts`, not the test for `coffee`. This mistake is easy to see here, but with larger and more complicated `if` structures, you need to keep in mind the following rule about which `if` owns which `else`.

An else always belongs to the nearest preceding if that's not already spoken for by another else.

The potential for confusion here is known as the dangling `else` problem. Braces will always make the situation clearer:

```
if(coffee == 'y')
{
    if(donuts == 'y')
    {
        std::cout << "We have coffee and donuts." << std::endl;
    }
    else
    {
        std::cout << "We have coffee, but not donuts." << std::endl;
    }
}
```

Now it's absolutely clear. The `else` definitely belongs to the `if` that is checking for `donuts`.

Understanding Nested ifs

Now that you know the rules, understanding an `if` nested within an `if-else` should be easy:

```
if(coffee == 'y')
{
    if(donuts == 'y')
        std::cout << "We have coffee and donuts." << std::endl;
}
else if(tea == 'y')
{
    std::cout << "We have no coffee, but we have tea." << std::endl;
}
```

Notice the formatting of the code here. When an `else` block is another `if`, writing `else if` on one line is an accepted convention. The braces enclosing the test for donuts are essential. Without them the `else` would belong to the `if` that's looking out for donuts. In this kind of situation, it is easy to forget to include the braces and thus create an error that may be hard to find. A program with this kind of error compiles without a problem, as the code is correct. It may even produce the right results some of the time. If you removed the braces in this example, you'd get the right results only as long as coffee and donuts were both 'y' so that the check for tea wouldn't execute.

Nesting `if-else` statements in other `if-else` statements can get very messy, even with just one level of nesting. Let's beat the coffee and donuts analysis to death by using it again:

```
if(coffee == 'y')
    if(donuts == 'y')
        std::cout << "We have coffee and donuts."
            << std::endl;
    else
        std::cout << "We have coffee, but not donuts."
            << std::endl;
else if(tea == 'y')
    std::cout << "We have no coffee, but we have tea, and maybe donuts..."
        << std::endl;
else
    std::cout << "No tea or coffee, but maybe donuts..."
        << std::endl;
```

The logic here doesn't look quite so obvious, even with the correct indentation. Braces aren't necessary, as the rule you saw earlier will verify, but it would look much clearer if you included them:

```
if(coffee == 'y')
{
    if(donuts == 'y')
    {
        std::cout << "We have coffee and donuts." << std::endl;
    }
    else
    {
        std::cout << "We have coffee, but not donuts." << std::endl;
    }
}
else
{
    if(tea == 'y')
    {
        std::cout << " We have no coffee, but we have tea, and maybe donuts..."
            << std::endl;
    }
    else
    {
        std::cout << "No tea or coffee, but maybe donuts..." << std::endl;
    }
}
```

There are much better ways of dealing with this kind of logic. If you put enough nested `ifs` together, you can almost guarantee a mistake somewhere. The next section will help to simplify things.

Logical Operators

As you have seen, using `ifs` where you have two or more related conditions can be cumbersome. You have tried your iffy talents on looking for coffee and donuts, but in practice, you may want to check much more complex conditions. For instance, you could be searching a personnel file for someone who is over 21, under 35, female, has a college degree, is unmarried, and who speaks Hindi or Urdu. Defining a test for this could involve the mother of all `ifs`.

The *logical operators* provide a neat and simple solution. Using logical operators, you can combine a series of comparisons into a single expression so that you need just one `if`, almost regardless of the complexity of the set of conditions. What's more, you won't have trouble determining which one to use because there are just the three shown in Table 4-4.

Table 4-4. *Logical Operators*

Operator	Description
<code>&&</code>	Logical AND
<code> </code>	Logical OR
<code>!</code>	Logical negation (NOT)

The first two, `&&` and `||`, are binary operators that combine two operands of type `bool` and produce a result of type `bool`. The third operator, `!`, is unary, so it applies to a single operand of type `bool` and produces a `bool` result. In the following pages I'll explain first how each of these is used, then I'll demonstrate them in an example. It's important not to confuse these with the bitwise operators that operate on the bits within integer operands. These logical operators only apply to operands of type `bool`.

Logical AND

You use the AND operator, `&&`, where you have two conditions that must both be true for a true result. For example, you want to be rich *and* healthy. Earlier, to determine whether a character was an uppercase letter, the value had to be both greater than or equal to 'A' *and* less than or equal to 'Z'. The `&&` operator *only* produces a true result if both operands are true. If either or both operands are false, then the result is false. Here's how you could test a `char` variable, `letter`, for an uppercase letter using the `&&` operator:

```
if(letter >= 'A' && letter <= 'Z')
{
    std::cout << "This is an uppercase letter." << std::endl;
}
```

The output statement executes only if both of the conditions combined by `&&` are true. No parentheses are necessary in the expression because the precedence of the comparison operators is higher than that of `&&`. As usual, you're free to put parentheses in if you want. You could write the statement as:

```
if((letter >= 'A') && (letter <= 'Z'))
{
    std::cout << "This is an uppercase letter." << std::endl;
}
```

Now there's no doubt that the comparisons will be evaluated first.

Logical OR

The OR operator, `||`, applies when you want a true result when either or both of the operands are true. The result is false only when both operands are false.

For example, you might be considered creditworthy enough for a bank loan if your income was at least \$100,000 a year, or if you had \$1,000,000 in cash. This could be tested like this:

```
if(income >= 100000.00 || capital >= 1000000.00)
{
    std::cout << "Of course, how much do you want to borrow?" << std::endl;
}
```

The response emerges when either or both of the conditions are true. (A better response might be, “*Why* do you want to borrow?” It’s strange how banks will only lend you money when you don’t need it.)

Logical Negation

The third logical operator, `!`, applies to single bool operand and inverts its value. So, if the value of a bool variable, `test`, is true, then `!test` is false; if `test` is false, then `!test` results in the value true. For example, suppose `x` has the value 10. The expression `!(x > 5)` evaluates to false, because `x>5` is true.

You could also express a well-known assertion of Charles Dickens using the `!` operator; if `!(income>expenditure)` is true, the result is misery — at least, as soon as the bank starts bouncing your checks.

You can apply all the logical operators to any expressions that evaluate to true or false. Operands can be anything from a single bool variable to a complex combination of comparisons and bool variables.

You can combine conditional expressions and logical operators to any degree to which you feel comfortable. This example implements a questionnaire to decide whether a person is a good loan risk:

```
// Ex4_06.cpp
// Combining logical operators for loan approval
#include <iostream>

int main()
{
    int age {};                // Age of the prospective borrower
    int income {};            // Income of the prospective borrower
    int balance {};           // Current bank balance

    // Get the basic data for assessing the loan
    std::cout << "Please enter your age in years: ";
    std::cin >> age;
    std::cout << "Please enter your annual income in dollars: ";
    std::cin >> income;
    std::cout << "What is your current account balance in dollars: ";
    std::cin >> balance;
```

```

// We only lend to people who over 21 years of age,
// who make over $25,000 per year,
// or have over $100,000 in their account, or both.
if(age >= 21 && (income > 25000 || balance > 100000))
{
    // OK, you are good for the loan - but how much?
    // This will be the lesser of twice income and half balance
    int loan {}; // Stores maximum loan amount
    if(2*income < balance/2)
    {
        loan = 2*income;
    }
    else
    {
        loan = balance/2;
    }
    std::cout << "\nYou can borrow up to $" << loan << std::endl;
}
else
{ // No loan for you...
    std::cout << "\nUnfortunately, you don't qualify for a loan." << std::endl;
}
}

```

Here's some sample output:

```

Please enter your age in years: 25
Please enter your annual income in dollars: 28000
What is your current account balance in dollars: 185000

You can borrow up to $56000

```

The interesting bit is the `if` statement that determines whether or not a loan will be granted. The `if` condition is `age >= 21 && (income > 25000 || balance > 100000)`

This condition requires that the applicant's age be at least 21, and that either their income is than \$25,000, or their account balance is greater than \$100,000. The parentheses around the expression `(income > 25000 || balance > 100000)` are necessary to ensure that the result of ORing the income and balance conditions together is ANDed with the result of the age test. Without the parentheses, the age test would be ANDed with the income test, and the result would be ORed with the balance test. This is because `&&` has a higher precedence than `||`, as you can see from the table back in Chapter 3. Without the parentheses, the condition would have allowed an 8-year-old with a balance over \$100,000 to get a loan. That's not what was intended. Banks never lend to minors or mynahs.

If the `if` condition is true, the block of statements that determine the loan amount executes. The `loan` variable is defined within this block and therefore ceases to exist at the end of the block. The `if` statement within the block determines whether twice the declared income is less than half the account balance. If it is, the loan is twice the income, otherwise it is half the account balance. This ensures the loan corresponds to the least amount according to the rules.

The Conditional Operator

The *conditional operator* is sometimes called the *ternary operator* because it involves three operands — the only operator to do so. It parallels the `if-else` statement, in that instead of selecting one of two statement blocks to execute depending on condition, it selects the value of one of two expressions. Thus the conditional operator enables you to choose between two values. Let's consider an example.

Suppose you have two variables, `a` and `b`, and you want to assign the value of the greater of the two to a third variable, `c`. The following statement will do this:

```
c = a > b ? a : b;           // Set c to the higher of a and b
```

The conditional operator has a logical expression as its first operand, in this case `a > b`. If this expression is true, the second operand — in this case `a` — is selected as the value resulting from the operation. If the first operand is false, the third operand — in this case `b` — is selected as the value. Thus, the result of the conditional expression is `a` if `a` is greater than `b`, and `b` otherwise. This value is stored in `c`. The assignment statement is equivalent to the `if` statement:

```
if(a > b)
{
    c = a;
}
else
{
    c = b;
}
```

Of course, you can use the conditional operator to select the lower of two values. In the previous program, you used an `if-else` to decide the value of the loan; you could use this statement instead:

```
loan = 2*income < balance/2 ? 2*income : balance/2;
```

This produces exactly the same result. You don't need parentheses because the precedence of the conditional operator is lower than that of the other operators in this statement. The condition is `2*income < balance/2`. If this evaluates to true, then the expression `2*income` evaluates and produces the result of the operation. If the condition is false, the expression `balance/2` produces the result of the operation.

Of course, if you think parentheses would make things clearer, you can include them:

```
loan = (2*income < balance/2) ? (2*income) : (balance/2);
```

The general form of the conditional operator, which is often represented by `?:`, is:

```
condition ? expression1 : expression2
```

If `condition` evaluates to true, the result is the value of `expression1`; if it evaluates to false, the result is the value of `expression2`. If `condition` is an expression that results in a numerical value, then it is implicitly converted to type `bool`. Note that only one of `expression1` or `expression2` will be evaluated. This has significant implications for expressions such as the following:

```
a < b ? ++i+1 : i+1;
```

If *a* is less than *b*, *i* is incremented and the result of the operation is the incremented value of *i* plus 1. The variable *i* is not incremented if *a* is not less than *b* - so *a* < *b* is false; in this case, the result of the operation is the current value of *i* plus 1.

You can use the conditional operator to control output depending on the result of an expression or the value of a variable. You can vary a message by selecting one text string or another depending on a condition.

```
// Ex4_07.cpp
// Using the conditional operator to select output.
#include <iostream>

int main()
{
    int mice {};           // Count of all mice
    int brown {};          // Count of brown mice
    int white {};          // Count of white mice

    std::cout << "How many brown mice do you have? ";
    std::cin >> brown;
    std::cout << "How many white mice do you have? ";
    std::cin >> white;

    mice = brown + white;

    std::cout << "You have " << mice
              << (mice == 1 ? " mouse " : " mice ")
              << "in total." << std::endl;
}
```

The output from this program might be:

```
How many brown mice do you have? 2
How many white mice do you have? 3
You have 5 mice in total.
```

The only bit of interest is the output statement that is executed after the numbers of mice have been entered. The expression using the conditional operator evaluates to " mouse " if the value of *mice* is 1, or " mice " otherwise. This allows you to use the same output statement for any number of mice and select singular or plural as appropriate.

There are many other situations in which you can apply this sort of mechanism. For example, selecting between "is" and "are", or "he" and "she", or indeed any situation in which you have a binary choice. You can even combine two conditional operators to choose between three options. Here's an example:

```
cout << (a < b ? "a is less than b." :
        (a == b ? "a is equal to b." : "a is greater than b."));
```

This statement outputs one of three messages, depending on the relative values of *a* and *b*. The second choice for the first conditional operator is the result of another conditional operator.

The switch Statement

You're often faced with a multiple-choice situation in which you need to execute a particular set of statements from a number of choices (that is, more than two), depending on the value of an integer variable or expression. The `switch` statement enables you to select from multiple choices. The choices are identified by a set of fixed integer values and the selection of a particular choice is determined by the value of a given integer expression.

The choices in a `switch` statement are called *cases*. A lottery where you win a prize depending on your number coming up is an example of where it might apply. You buy a numbered ticket, and if you're lucky, you win a prize. For instance, if your ticket number is 147, you win first prize; if it's 387 you can claim second prize; ticket number 29 gets you third prize; any other ticket number wins nothing. The `switch` statement to handle this situation would have four cases: one for each of the winning numbers, plus a "default" case for all the losing numbers. Here's a `switch` statement that selects a message for a given ticket number:

```
switch(ticket_number)
{
case 147:
    std::cout << "You win first prize!";
    break;
case 387:
    std::cout << "You win second prize!";
    break;
case 29:
    std::cout << "You win third prize!";
    break;
default:
    std::cout << "Sorry, you lose.";
    break;
}
```

The `switch` statement is harder to describe than to use. The selection of a particular case is determined by the value of the integer expression between the parentheses that follow the keyword `switch`. In this example, it is simply the variable `ticket_number`, which must be an integer type; what else could it be?

The possible choices in a `switch` statement appear in a block, and each choice is identified by a case value. A *case value* appears in a *case label*, which is of the form:

```
case case_value:
```

It's called a *case label* because it labels the statements or block of statements that it precedes. The statements that follow a particular case label execute if the value of the selection expression is the same as that of the case value. Each case value must be unique but case values don't need to be in any particular order, as the example demonstrates.

Each case value must be an *integer constant expression*, which is an expression that the compiler can evaluate; this implies that it can only involve literals, or `const` variables. Furthermore, any literals must either be of an integer type or be able to be converted to an integer type.

The `default` label in the example identifies the *default case*, which is a catchall that is selected if none of the other cases is selected. You don't have to specify a default case, though. If you don't, and none of the case values is selected, the `switch` does nothing.

The `break` statement that appears after each set of case statements is essential for the logic here. Executing a `break` statement breaks out of the `switch` and causes execution to continue with the statement following the closing brace. If you omit the `break` statement for a case, the statements for the following case will execute. Notice that we *don't* need a `break` after the final case (usually the `default` case) because execution leaves the `switch` at this point anyway. It's good programming style to include it though because it safeguards against accidentally falling through to another case that you might add to a `switch` later. `switch`, `case`, `default`, and `break` are all keywords.

This example demonstrates the switch statement:

```
// Ex4_07.cpp
// Using the switch statement
#include <iostream>

int main()
{
    int choice {};           // Stores selection value

    std::cout << "Your electronic recipe book is at your service.\n"
               << "You can choose from the following delicious dishes:\n"
               << "1 Boiled eggs\n"
               << "2 Fried eggs\n"
               << "3 Scrambled eggs\n"
               << "4 Coddled eggs\n\n"
               << "Enter your selection number: ";
    std::cin >> choice;

    switch(choice)
    {
    case 1:
        std::cout << "\nBoil some eggs." << std::endl;
        break;
    case 2:
        std::cout << "Fry some eggs." << std::endl;
        break;
    case 3:
        std::cout << "Scramble some eggs." << std::endl;
        break;
    case 4:
        std::cout << "Coddle some eggs." << std::endl;
        break;
    default:
        std::cout << "You entered a wrong number - try raw eggs." << std::endl;
    }
}
```

After defining your options in the output statement and reading a selection number into the variable `choice`, the switch statement executes with the selection expression specified simply as `choice` in parentheses, immediately following the keyword `switch`. The possible choices in the switch are between braces and are each identified by a case label. If the value of `choice` corresponds with any of the case values, then the statements following that case label execute. You only have one statement plus a `break` statement for each case in this example, but in general you can have as many statements as you need following a case label, and you don't need to enclose them between braces.

The `break` statement at the end of each group of case statements transfers execution to the statement after the switch. You can demonstrate the essential nature of the `break` statements here by removing them from the example and seeing what happens.

If the value of `choice` doesn't correspond with any of the case values, the statements following the `default` label execute. If you hadn't included a `default` case here and the value of `choice` was different from all the case values, then the switch would have done nothing and the program would continue with the next statement after the switch — Effectively executing `return 0` because the end of `main()` has been reached.

As I said earlier, each of the case values must be a compile-time constant and must be unique. The reason that no two case values can be the same is that if they are, the compiler has no way of knowing which statements should be executed when that particular value comes up. However, different case values don't need to have unique actions. Several case values can share the same action, as the following example shows:

```
// Ex4_08.cpp
// Multiple case actions
#include <iostream>
#include <locale>

int main()
{
    char letter {};
    std::cout << "Enter a letter: ";
    std::cin >> letter;

    if(isalpha(letter))
    {
        switch(tolower(letter))
        {
            case 'a': case 'e': case 'i': case 'o': case 'u':
                std::cout << "You entered a vowel." << std::endl;
                break;
            default:
                std::cout << "You entered a consonant." << std::endl;
                break;
        }
    }
    else
        std::cout << "You did not enter a letter." << std::endl;
}
```

Here is an example of some output:

```
Enter a letter: E
You entered a vowel.
```

The `if` condition first checks that you really do have a letter and not some other character using the `isalpha()` classification function from the Standard Library. The integer returned will be nonzero if the argument is alphabetic and this will be implicitly converted to `true`, which causes the `switch` to be executed. The `switch` condition converts the value to lowercase using a Standard Library character conversion routine, `tolower()`, and uses the result to select a case. Converting to lowercase avoids the need to have case labels for upper and lowercase letters. All of the cases that identify a vowel cause the same statements to be executed. You can see that you just write each of the cases in a series, followed by the statements any of the cases is to select. If the input is not a vowel, it must be a consonant and the default case deals with this.

If `isalpha()` returns 0, which converts to `false`, the `switch` doesn't execute because the `else` clause is selected; this outputs a message indicating that the character entered was not a letter.

It's possible to dispense with the `if` statement by combining the test for a letter with the conversion to lowercase, but it requires some trickery and does make the code more complicated. For example, you could write the `switch` as follows:

```
switch(tolower(letter) * static_cast<bool>(isalpha(letter)))
{
    case 'a': case 'e': case 'i': case 'o': case 'u':
        std::cout << "You entered a vowel." << std::endl;
        break;
    case 0:
        std::cout << "You did not enter a letter." << std::endl;
        break;
    default:
        std::cout << "You entered a consonant." << std::endl;
}
```

Casting the value returned by `isalpha()` to `bool` produces `true` when the letter is alphabetic and `false` otherwise. The multiplication in the `switch` condition requires that both operands be numeric, so the compiler will insert an implicit conversion for the `bool` value. This will result in 1 when the input is alphabetic and 0 otherwise. Multiplying the character returned by `tolower()` by this either leaves it unchanged or results in 0. The latter selects case 0 and the former selects one of the other cases or the default case.

Another possibility is to write the `switch` condition as `tolower(letter)*(isalpha(letter)!=0)`. I'll leave you to figure out why this also works. These versions of the code require effort to figure out what is going on. In general, clearer code is better code.

Unconditional Branching

The `if` statement provides you with the flexibility to choose to execute one set of statements or another, depending on a specified condition. The `switch` statement provides a way to choose from a fixed range of options depending on the value of an integer expression. The `goto` statement, in contrast, is a blunt instrument. It enables you to branch to a specified program statement unconditionally. The statement to be branched to must be identified by a *statement label*, which is an identifier defined according to the same rules as a variable name. This is placed before the statement to be referenced and separated from it by a colon. Here's an example of a labeled statement:

```
MyLabel: x = 1;
```

This statement has the label `MyLabel`, and an unconditional branch to this statement would be written as follows:

```
goto MyLabel;
```

Whenever possible, you should avoid using `goto` statements. They encourage convoluted code that can be extremely difficult to follow. Note that a `goto` that branches into the scope of a variable but bypasses its declaration will cause a compiler error message.

■ **Note** Because the `goto` statement is theoretically unnecessary — you always have an alternative to using `goto` — a significant cadre of programmers says that you should never use it. I don't subscribe to such an extreme view. It is a legal statement, after all, and there are rare occasions when it can reduce code complexity. However, I do recommend that you only use it where you can see an obvious advantage over other options that are available.

Statement Blocks and Variable Scope

A switch statement has its own block between braces that encloses the case statements. An if statement also often has braces enclosing the statements to be executed if the condition is true, and the else part may have such braces too. These statement blocks are no different from any other blocks when it comes to variable scope. Any variable declared within a block ceases to exist at the end of the block, so you cannot reference it outside the block.

For example, consider the following rather arbitrary calculation:

```
if(value > 0)
{
    int savit {value - 1};          // This only exists in this block
    value += 10;
}
else
{
    int savit {value + 1};          // This only exists in this block
    value -= 10;
}
std::cout << savit;                // This will not compile! savit does not exist
```

The output statement at the end causes a compiler error message because the savit variable is undefined at this point. Any variable defined within a block can only be used within that block, so if you want to access data that originates inside a block from outside it, you must define the variable storing that information in an outer block.

Variable definitions within a switch statement block must be reachable in the course of execution, and it must not be possible to bypass them; otherwise the code will not compile. The following code illustrates how illegal declarations can arise in a switch:

```
int test {3};
switch(test)
{
    int i {1};                      // ILLEGAL - cannot be reached

case 1:
{
    int j {2};                      // OK - can be reached and is not bypassed
    std::cout << test + j << std::endl;
    break;
}

    int k {3};                      // ILLEGAL - cannot be reached

case 3:
    std::cout << test << std::endl;
    int m {4};                      // ILLEGAL - can be reached but can be bypassed
    break;

default:
    std::cout << "Default reached." << std::endl;
    break;

    int n {5};                      // ILLEGAL - cannot be reached
}
std::cout << j << std::endl;        // ILLEGAL - j doesn't exist here
```

Only one of the definitions in this switch statement is legal: the one for `j`. For a definition to be legal, it must first be possible for it to be reached and thus executed in the normal course of execution. This is not the case for variables `i`, `k`, and `n`. Secondly, it must not be possible during execution to enter the scope of a variable while bypassing its definition, which is the case for the variable `m`. Variable `j`, however, is only “in scope” from its declaration to the end of the enclosing block, so this declaration cannot be bypassed.

Summary

In this chapter, you have added the capability for decision-making to your programs. You now know how *all* the decision-making statements in C++ work. The essential elements of decision-making that you have learned about in this chapter are:

- You can compare two values using the comparison operators. This will result in a value of type `bool`, which can be `true` or `false`.
- You can convert a `bool` value to an integer type—`true` will convert to 1 and `false` will convert to 0.
- Numerical values can be converted to type `bool`—a zero value converts to `false`, and a nonzero value casts to `true`. When a numerical value appears where a `bool` value is expected - such as in an `if` condition - the compiler will insert an implicit conversion of the numerical value to type `bool`.
- The `if` statement executes a statement or a block of statements depending on the value of a condition expression. If the condition is `true`, the statement or block executes. If the condition is `false` it doesn't.
- The `if-else` statement executes a statement or block of statements when the condition is `true`, and another statement or block when the condition is `false`.
- `if` and `if-else` statements can be nested.
- The `switch` statement provides a way to select one from a fixed set of options, depending on the value of an integer expression.
- The conditional operator selects between two values depending on the value of an expression.
- You can branch unconditionally to a statement with a specified label by using a `goto` statement.

EXERCISES

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck after that, you can download the solutions from the Apress website (www.apress.com/source-code/), but that really should be a last resort.

Exercise 4-1. Write a program that prompts for two integers to be entered and then uses an `if-else` statement to output a message that states whether or not the integers are the same.

Exercise 4-2. Create a program that prompts for input of an integer between 1 and 100. Use a nested `if`, first to verify that the integer is within this range, and then, if it is, to determine whether or not the integer is greater than, less than, or equal to 50. The program should output information about what was found.

Exercise 4-3. Design a program that prompts for input of a letter. Use a library function to determine whether or not the letter is a vowel and whether it is lowercase or not, and output the result. Finally, output the lowercase letter together with its character code as a binary value.

Exercise 4-4. Write a program that determines, using only the conditional operator, if an integer that is entered has a value that is 20 or less, is greater than 20 and not greater than 30, is greater than 30 but not exceeding 100, or is greater than 100.

Exercise 4-5. Create a program that prompts the user to enter an amount of money between \$0 and \$10 (decimal places allowed). Determine how many quarters (25c), dimes (10c), nickels (5c), and pennies (1c) are needed to make up that amount. Output this information to the screen and ensure that the output makes grammatical sense (for example, if you need only one dime then the output should be “1 dime” and not “1 dimes”).



Arrays and Loops

An array enables you to work with several data items of the same type using a single name, the array name. The need for this occurs often — working with a series of temperatures or the ages of a group of people for example. A loop is another fundamental programming facility. It provides a mechanism for repeating one or more statements as many times as your application requires. Loops are essential in the majority of programs. Using a computer to calculate the company payroll, for example, would not be practicable without a loop. There are several kinds of loop, each with their own particular area of application. In this chapter, you'll learn:

- What an array is and how you create an array
- How to use a `for` loop
- How the `while` loop works
- What the merits of the `do-while` loop are
- What the `break` and `continue` statement do in a loop
- What the `continue` statement does in a loop
- How to use nested loops
- How to create and use an array container
- How to create and use a vector container

Arrays

The variables you have created up to now can store only a single data item of the specified type — an integer, a floating-point value, a character, or a `bool` value. An array stores several data items of the same type. You can create an array of integers or an array of characters — in fact an array of any type of data, and there can be as many as the available memory will allow.

Suppose you've written a program to calculate an average temperature. You now want to extend the program to calculate how many samples are above the average and how many are below. You'll need to retain the original sample data to do this, but storing each data item in a separate variable would be tortuous to code and impractical for anything more than a very few items. An array provides you with the means of doing this easily, and many other things besides.

Using an Array

An *array* is a variable that represents a sequence of memory locations; each can store an item of data of the same data type. For example, you could store 366 temperature samples in an array defined as follows:

```
double temperatures[366];           // An array of temperatures
```

This defines an array with the name `temperatures` to store 366 values of type `double`. The data values are called *elements*. The number of elements specified between the brackets is the *size* of the array. The array elements are not initialized in this statement so they contain junk values.

You refer to an array element using an integer called an *index*. The index of a particular array element is its offset from the first element. The first element has an offset of 0 and therefore an index of 0; an index value of 3 refers to the fourth array element — three elements from the first. To reference an element, you put its index between square brackets after the array name, so to set the fourth element of the `temperatures` array to 99.0, you would write:

```
temperatures[3] = 99.0;           // Set the fourth array element to 99
```

Let's look at another array. The Figure 5-1 shows the structure of an array called `height` that has six elements of type `double`.

height [0]	height [1]	height [2]	height [3]	height [4]	height [5]
26	37	47	55	62	75

Figure 5-1. An array with six elements

The array has six elements of type `int`. Each box in Figure 5-1 represents a memory location holding an array element. Each element can be referenced using the expression above it. You can define an array that has six elements of type `int` using this statement:

```
unsigned int height[6];           // Define an array of six heights
```

The compiler will allocate six contiguous storage locations for storing values of type `unsigned int` as a result of this definition. If this type is 4 bytes on your computer, this array will occupy 24 bytes. The definition doesn't specify any initial values for the array, so the elements contain junk values.

■ **Note** The type of the array will determine the amount of memory required for each element. The elements of an array are stored in one contiguous block of memory.

Each element in the `height` array in Figure 5-1 contains a different value. These might be the heights of the members of a family, recorded to the nearest inch. As there are six elements, the index values run from 0 for the first element through to 5 for the last element. You could define the array with these initial values like this:

```
unsigned int height[6] {26, 37, 47, 55, 62, 75}; // Define & initialize array of 6 heights
```

The initializer list contains six values separated by commas. Each array element will be assigned an initial value from the list in sequence, so the elements will have the values shown in Figure 5-1. The initializer list must not have more values than there are elements in the list, otherwise the statement won't compile. There can be less values in the list, in which case the elements for which no initial value has been supplied will be initialized with zero. For example:

```
unsigned int height[6] {26, 37, 47};           // Element values: 26 37 47 0 0 0
```

The first three elements will have the values that appear in the list. The last three will be zero. To initialize all the elements with zero, you can just use an empty initializer list:

```
unsigned int height[6] {};                     // All elements 0
```

Of course, you could put 0 in the initializer list and get the same effect.

Array elements participate in arithmetic expressions like other variables. You could sum the first three elements of `height` like this:

```
unsigned int sum {};  
sum = height[0] + height[1] + height[2];    // The sum of three elements
```

You use references to individual array elements like ordinary integer variables in an expression. As you saw earlier, an array element can be on the left of an assignment to set a new value so you can copy the value of one element to another in an assignment, like this for example:

```
height[3] = height[2];                       // Copy 3rd element value to 4th element
```

However, you can't copy *all* the element values from one array to the elements of another in an assignment. You can only operate on individual elements. To copy the values of one array to another, you must copy the values one at a time. What you need is a *loop*.

Understanding Loops

A loop is a mechanism that enables you to execute a statement or block of statements repeatedly until a particular condition is met. The statements inside a loop are sometimes called *iteration statements*. A single execution of the statement or statement block that is within the loop is an *iteration*.

Two essential elements make up a loop: the statement or block of statements that forms the body of the loop that is to be executed repeatedly, and a *loop condition* of some kind that determines when to stop repeating the loop. A loop condition can take different forms to provide different ways of controlling the loop. For example, a loop condition can:

- Execute a loop a given number of times.
- Execute a loop until a given value exceeds another value.
- Execute the loop until a particular character is entered from the keyboard.
- Execute a loop for each element in a collection of elements.

You choose the loop condition to suit the circumstances. You have the following varieties of loops:

- *The for loop* primarily provides for executing the loop a prescribed number of times but there is considerable flexibility beyond that.
- *The range-based for loop* executes one iteration for each element in a collection of elements.

- The *while loop* continues executing as long as a specified condition is true. The condition is checked at the beginning of an iteration so if the condition starts out as false, no loop iterations are executed.
- The *do-while loop* continues to execute as long as a given condition is true. This differs from the while loop in that the do-while loop checks the condition at the end of an iteration. This implies that at least one loop iteration always executes.

I'll start by explaining how the for loop works.

The for Loop

The for loop executes a statement or block of statements a predetermined number of times, but you can use it in other ways too. You specify how a for loop operates using three expressions separated by semicolons between parentheses following the for keyword. This is shown in Figure 5-2.

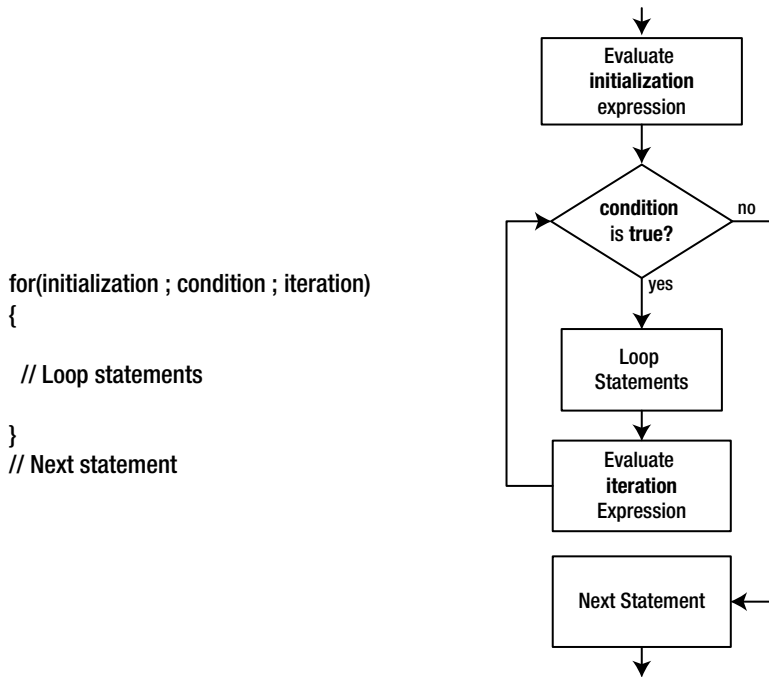


Figure 5-2. The logic of the for loop

You can omit any or all of the expressions controlling a for loop but you must always include the semicolons. I'll explain later in this chapter why and when you might omit one or other of the control expressions. The initialization expression is evaluated only once, at the beginning of the loop. The loop condition is checked next, and if it is true, the loop statement or statement block executes. If the condition is false, the loop ends and execution continues with the statement after the loop. After each execution of the loop statement or block, the iteration expression is evaluated and the condition is checked to decide if the loop should continue.

In the most typical usage of the `for` loop, the first expression initializes a counter, the second expression checks whether the counter has reached a given limit, and the third expression increments the counter. For example, you could copy the elements from one array to another like this:

```
double rainfall[12] {1.1, 2.8, 3.4, 3.7, 2.1, 2.3, 1.8, 0.0, 0.3, 0.9, 0.7, 0.5};
double temp[12] {};
for(size_t i {} ; i<12 ; ++i)      // i varies from 0 to 11
{
    temp[i] = rainfall[i];          // Copy ith element of rainfall to ith element of temp
}
```

The first expression defines `i` as type `size_t` with an initial value of 0. You'll recall that the `sizeof` operator returns a value of `size_t`, which is an unsigned integer type that is used generally for sizes of things as well as counts. `i` will be used to index the arrays so using `size_t` makes sense. Not only is it *legal* to define variables within a `for` loop initialization expression, it is very common. This has some significant implications. A loop defines a scope. The loop statement or block, including any expressions that control the loop fall within the scope of a loop. Any automatic variables declared within the scope of a loop do not exist outside it. Because `i` is defined in the first expression, it is local to the loop so when the loop ends, `i` will no longer exist.

The second expression, the loop condition, is true as long as `i` is less than 12, so the loop continues while `i` is less than 12. When `i` reaches 12, the expression will be false so the loop ends. The third expression increments `i` at the end of each loop iteration so the loop block that copies the `i`th element from `rainfall` to `temp` will execute with values of `i` from 0 to 11.

When you need to be able to access the loop control variable after the loop ends, you just define it before the loop, like this:

```
size_t i {};
for(i = 0 ; i<12 ; ++i)          // i varies from 0 to 11
{
    temp[i] = rainfall[i];        // Copy ith element of rainfall to ith element of temp
}
// i still exists here...
```

Now you can access `i` after the loop - its value will be 12 in this case. `i` is initialized to 0 in its definition so the first loop control expression is superfluous. You can omit any or all of the loop control expressions so the loop can be written as:

```
size_t i {};
for( ; i<12 ; ++i)              // i varies from 0 to 11
{
    temp[i] = rainfall[i];        // Copy ith element of rainfall to ith element of temp
}
```

The loop works just as before. The first control expression is not necessary because `i` is defined and initialized to zero before the loop. I'll discuss omitting other control expressions a little later in this chapter.

Avoiding Magic Numbers

One problem with the preceding code fragment is that it involves the “magic number” 12 for the array sizes. It’s easy to make a mistake when entering the `rainfall` array size of 12 in the definition of the `temp` array and in the `for` loop. It would be better to define a `const` variable for the array size and use that instead of the explicit value:

```
const size_t size {12};
double rainfall[size] {1.1, 2.8, 3.4, 3.7, 2.1, 2.3, 1.8, 0.0, 0.3, 0.9, 0.7, 0.5};
double temp[size] {};
for(size_t i {} ; i<size ; ++i)    // i varies from 0 to size-1
{
    temp[i] = rainfall[i];          // Copy ith element of rainfall to ith element of temp
}
```

This is much less error prone and it is clear that `size` is the number of elements in both arrays. Let’s try out a `for` loop in a complete example:

```
// Ex5_01.cpp
// Using a for loop with an array
#include <iostream>

int main()
{
    const size_t size {6};                // Array size
    unsigned int height[size] {26, 37, 47, 55, 62, 75}; // An array of heights
    unsigned int total {};                 // Sum of heights

    for(size_t i {} ; i<size ; ++i)
    {
        total += height[i];
    }
    int average {total/size};               // Calculate average height
    std::cout << "The average height is " << average << std::endl;

    unsigned int count {};
    for(size_t i {} ; i < size ; ++i)
    {
        if(height[i] < average) ++count;
    }
    std::cout << count << " people are below average height." << std::endl;
}
```

The output is:

```
The average height is 50
3 people are below average height.
```

The definition of the `height` array uses a `const` variable to specify the number of elements. The `size` variable is also used as the limit for the control variable in the two `for` loops. The first `for` loop iterates over each `height` element in turn, adding its value to `total`. The loop ends when the loop variable `i` is equal to `size`, and the statement following the loop is executed, which defines the `average` variable with the initial value as `total` divided by `size`.

After outputting the average height, the second `for` loop iterates over the elements in the array, comparing each value with `average`. The `count` variable is incremented each time an element is less than `average`, so when the loop ends, `count` will contain the number of elements less than `average`. You could replace the `if` statement in the loop with this statement:

```
count += height[i] < average;
```

This works because the `bool` value that results from the comparison will be implicitly converted to an integer. The value `true` converts to 1 and `false` converts to 0 so `count` will be incremented only when the comparison results in `true`.

■ **Caution** Array index values are not checked to verify that they are valid. It's up to you to make sure that you don't reference elements outside the bounds of the array. If you store data using an index value that's outside the valid range for an array, you'll overwrite something in memory or cause a storage protection violation. Either way, your program will almost certainly come to a sticky end.

Defining the Array Size with the Initializer List

You can omit the size of the array when you supply one or more initial values in its definition. The number of elements will be the number of initial values. For example:

```
int values[] {2, 3, 4};
```

This defines an array with three elements of type `int` that will have the initial values 2, 3, and 4. It is equivalent to writing this:

```
int values[3] {2, 3, 4};
```

The advantage of omitting the size is that you can't get the array size wrong; the compiler determines it for you. You can't have an array with no elements so the initializer list must always contain at least one initial value if you omit the array size. An empty initializer list will result in a compilation error if you don't specify the array size.

Determining the Size of an Array

You saw earlier how you could avoid magic numbers for the number of elements in an array by defining a constant initialized with the array size. You also don't want to be specifying a magic number for the array size when you let the compiler decide the number of elements from the initializer list. You need a foolproof way of determining the size when necessary. The `sizeof` operator returns the number of bytes that a variable occupies and this works with an entire array as well as with a single array element. Thus the `sizeof` operator provides a way to determine the number of elements in an array; you just divide the size of the array by the size of the first element. Suppose you've defined this array:

```
int values[] {2, 3, 5, 7, 11, 13, 17, 19};
```

The expression `sizeof(values)` evaluates to the number of bytes occupied by the entire array. The expression `sizeof(values[0])` evaluates to the number of bytes occupied by a single element — since there's always at least one element the first element is a good choice. The expression `sizeof(values)/sizeof(values[0])` divides the number of bytes occupied by the whole array by the number of bytes for one element so this evaluates to the number of elements in the array. Let's try it out.

```
// Ex5_02.cpp
// Obtaining the number of array elements
#include <iostream>

int main()
{
    int values[] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
    std::cout << "There are " << sizeof (values)/sizeof(values[0])
               << " elements in the array." << std::endl;

    int sum {};
    for(size_t i {} ; i < sizeof (values)/sizeof (values[0]) ; ++i)
    {
        sum += values[i];
    }
    std::cout << "The sum of the array elements is " << sum << std::endl;
}
```

This example produces the following output:

```
There are 10 elements in the array.
The sum of the array elements is 129
```

The number of elements in the `values` array is determined by the compiler from the number of initializing values in the definition. The first output statement uses the `sizeof` operator to calculate the number of array elements. This is repeated in the `for` loop that calculates the sum of the array elements. You could avoid having to recalculate the size of the array by initializing a `const` variable like this:

```
int values[] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
const size_t size {sizeof(values) / sizeof(values[0])};
```

Now you can just use `size` whenever you need the size of the array in the code.

None of the control expressions have to be of a particular form. You have seen that you can omit the first control expression. In the `for` loop in the example you could accumulate the sum of the elements within the third loop control expression. The loop would be like this:

```
int sum {};
for(size_t i {} ; i < size ; sum += values[i++])
;
```

The third loop control expression now does two things: it adds the value of the element at index `i` to `sum`, then increments the control variable, `i`. The single semicolon is an empty statement that is the loop body. Note that before `i` was incremented using the prefix `++` operator, whereas now it is incremented using the postfix `++` operator. This is essential here to ensure the element selected by `i` is added to `sum` before `i` is incremented. If you use the prefix form, you get the wrong answer for the sum of the elements; you'll also use an invalid index value that accesses memory beyond the end of the array.

Controlling a for Loop with Floating-Point Values

The for loop examples so far have used an integer variable to control the loop, but you can use anything you like. The following code fragment uses floating-point values to control the loop:

```
const double pi {3.14159265};
for(double radius {2.5} ; radius <= 20.0 ; radius += 2.5)
{
    std::cout << "radius = " << std::setw(12) << radius
        << "   area = " << std::setw(12)
        << pi * radius * radius << std::endl;
}
```

This loop is controlled by the `radius` variable, which is of type `double`. It has an initial value of 2.5 and is incremented at the end of each loop iteration until it exceeds 20.0, whereupon the loop ends. The loop statement calculates the area of a circle for the current value of `radius`, using the standard formula πr^2 , where r is the radius of the circle. The manipulator `setw()` in the loop statement gives each output value the same field width; this ensures that the output values line up vertically. Of course, to use the manipulators in a program, you need to include the `iomanip` header.

You need to be careful when using a floating-point variable to control a for loop. Fractional values may not be representable exactly as a binary floating-point number. This can lead to some unwanted side effects, as this complete example demonstrates.

```
// Ex5_03.cpp
// Floating-point control in a for loop
#include <iostream>
#include <iomanip>

int main()
{
    const double pi { 3.14159265 };    // The famous pi
    const size_t perline {3};          // Outputs per line
    size_t linecount {};               // Count of output lines
    for (double radius {0.2} ; radius <= 3.0 ; radius += 0.2)
    {
        std::cout << std::fixed << std::setprecision(2) << " radius =" << std::setw(5)
            << radius << "   area =" << std::setw(6) << pi * radius * radius;
        if (perline == ++linecount)    // When perline outputs have been written...
        {
            std::cout << std::endl;      // ...start a new line...
            linecount = 0;               // ...and reset the line counter
        }
    }
    std::cout << std::endl;
}
```

On my computer, this produces the following output:

```
radius = 0.20  area =  0.13  radius = 0.40  area =  0.50  radius = 0.60  area =  1.13
radius = 0.80  area =  2.01  radius = 1.00  area =  3.14  radius = 1.20  area =  4.52
radius = 1.40  area =  6.16  radius = 1.60  area =  8.04  radius = 1.80  area = 10.18
radius = 2.00  area = 12.57  radius = 2.20  area = 15.21  radius = 2.40  area = 18.10
radius = 2.60  area = 21.24  radius = 2.80  area = 24.63
```

The loop includes an `if` statement to output three sets of values per line. You would expect to see the area of a circle with radius 3.0 as the last output. After all, the loop should continue as long as `radius` is less than or equal to 3.0. But the last value displayed has the radius at 2.8; what's going wrong?

The loop ends earlier than expected because when 0.2 is added to 2.8, the result is greater than 3.0. This is an astounding piece of arithmetic at face value, but read on! The reason for this is a very small error in the representation of 0.2 as a binary floating-point number. 0.2 cannot be represented exactly in binary floating point. The error is in the last digit of precision, so if your compiler supports 15-digit precision for type `double`, the error is of the order of 10^{-15} . Usually, this is of no consequence, but here you depend on adding 0.2 successively to get *exactly* 3.0—which doesn't happen.

You can see what the difference is by changing the loop to output just one circle area per line and to display the difference between 3.0 and the next value of `radius`:

```
for(double radius {0.2} ; radius <= 3.0 ; radius += .2)
{
    std::cout << std::fixed << std::setprecision(2) << " radius =" << std::setw(5)
               << radius << " area =" << std::setw(6) << pi * radius * radius
               << " delta to 3 = " << std::scientific << ((radius + 0.2) - 3.0) << std::endl;
}
```

On my machine, the last line of output is now this:

```
radius = 2.80  area = 24.63  delta to 3 = 4.44e-016
```

As you can see, `radius + 0.2` is greater than 3.0 by around 4.44×10^{-16} . This causes the loop to terminate before the next iteration.

■ **Note** Any number that is a fraction with an odd denominator cannot be represented exactly as a binary floating-point value.

More Complex for Loop Control Expressions

You can define and initialize more than one variable of a given type in the first `for` loop control expression. You just separate each variable from the next with a comma. Here's a working example that makes use of that:

```
// Ex5_04.cpp
// Multiple initializations in a loop expression
#include <iostream>
#include <iomanip>
```

```

int main()
{
    unsigned int limit {};
    std::cout << "This program calculates n! and the sum of the integers"
              << " up to n for values 1 to limit.\n";
    std::cout << "What upper limit for n would you like? ";
    std::cin >> limit;

    // Output column headings
    std::cout << std::setw(8) << "integer" << std::setw(8) << " sum"
              << std::setw(20) << " factorial" << std::endl;

    for (unsigned long long n {1ULL}, sum {}, factorial {1ULL} ; n <= limit ; ++n)
    {
        sum += n;                      // Accumulate sum to current n
        factorial *= n;                // Calculate n! for current n
        std::cout << std::setw(8) << n << std::setw(8) << sum
                  << std::setw(20) << factorial << std::endl;
    }
}

```

The program calculates the sum of the integers from 1 to n for each integer n from 1 to `count`, where `count` is an upper limit that you enter. It also calculates the factorial of each n . (The factorial of an integer n , written $n!$, is the product of all the integers from 1 to n ; for example, $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$.) Don't enter large values for `count`. Factorials grow very rapidly and easily exceed the capacity of even a variable of type `unsigned long long`. Here's some typical output:

This program calculates $n!$ and the sum of the integers up to n for values 1 to `limit`.

What upper limit for n would you like? 10

integer	sum	factorial
1	1	1
2	3	2
3	6	6
4	10	24
5	15	120
6	21	720
7	28	5040
8	36	40320
9	45	362880
10	55	3628800

First, you read the value for `limit` from the keyboard after displaying a prompt. The value entered for `limit` will not be large so type `unsigned int` is more than adequate. Using `setw()` to specify the field width for the column headings for the output enables the values to be aligned vertically with the headings simply by specifying the same field widths. The `for` loop does all the work. The first control expression defines and initializes three variables of type `unsigned long long`. n is the loop counter, `sum` accumulates the sum of integer from 1 to the current n , and `factorial` will store $n!$. Type `unsigned long long` provides the maximum range of positive integers and so maximizes the range of factorials that can be calculated. Note that there will be no warning if a factorial value cannot be accommodated in the memory allocated; the result will just be incorrect.

The Comma Operator

Although the comma looks as if it's just a humble separator, it is actually a binary operator. It combines two expressions into a single expression, where the value of the operation is the value of its right operand. This means that anywhere you can put an expression, you can also put a series of expressions separated by commas. For example, consider the following statements:

```
int i {1};
int value1 {1};
int value2 {1};
int value3 {1};
std::cout << (value1 += ++i, value2 += ++i, value3 += ++i) << std::endl;
```

The first four statements define four variables with an initial value 1. The last statement outputs the result of three assignment expressions that are separated by the comma operator. The comma operator is left associative and has the lowest precedence of all the operators so the expression evaluates like this:

```
((value1 += ++i), (value2 += ++i)), (value3 += ++i));
```

The effect will be that `value1` will be incremented by 2 to produce 3, `value2` will be incremented by 3 to produce 4, and `value3` will be incremented by 4 to produce 5. The value of the composite expression is the value of the rightmost expression in the series, so the value that is output is 5. Just to illustrate the possibility, you could use the comma operator to incorporate the calculations into the second loop control expression in `Ex5_04.cpp`:

```
for (unsigned long long n {1ULL}, sum {}, factorial {1ULL} ;
      sum += n, factorial *= n, n <= limit ; ++n)
{
    std::cout << std::setw(8) << n << std::setw(8) << sum
              << std::setw(20) << factorial << std::endl;
}
```

The second control expression combines three expressions using the comma operator. The first expression adds the current `n` to `sum`, the second multiplies `factorial` by the current `n` and the third compares `n` to `limit`, as before. The value of the overall expression will be the value of the rightmost, which is the comparison, so the loop is controlled exactly as before. If you replace the loop in `Ex5_04.cpp` by this and run the example again you'll see that it works as before. Note that this is just to illustrate that you can put multiple expressions for the second control expression in a `for` loop and show the comma operator in action. It is not good practice to code like this. You could put the calculations in the third control expression but the output would be incorrect because the third control expression executes at the end of each iteration.

The Ranged-based for Loop

The *range-based for loop* iterates over all the values in a range of values. This raises the immediate question: what is a range? An array is a range of elements and a string is a range of characters. The *containers* provided by the Standard Library for managing are all ranges. I'll introduce two Standard Library containers later in this chapter. The general form of the range-based for loop is:

```
for(range_declaration : range_expression)
    loop statement or block;
```


The `range_declaration` identifies a variable that will be assigned each of the values in the range in turn, a new value being assigned on each iteration. The `range_expression` identifies the range that is the source of the data. This will be clearer with an example. Consider these statements:

```
int values [] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
int total {};
for(int x : values)
    total += x;
```

The variable `x` will be assigned a value from the `values` array on each iteration. It will be assigned values 2, 3, 5, and so on in succession. Thus the loop will accumulate the sum of all the elements in the `values` array in `total`. The variable `x` is local to the loop and does not exist outside it.

Of course, the compiler knows the type of the elements in the `values` array so you could allow the compiler to determine the type for `x` by writing the loop as:

```
for(auto x : values)
    total += x;
```

Using the `auto` keyword causes the compiler to deduce the correct type for `x`. The `auto` keyword is used very often with the range-based `for` loop. This is a very nice way of iterating over all the elements in an array or other kind of range. You don't need to be aware of the number of elements. The loop mechanism takes care of that.

Note that the values from the range are *assigned* to the range variable, `x`. This means that you cannot modify the elements of `values` by modifying the value of `x`. For example, this doesn't change the elements in the `values` array:

```
for(auto x : values)
    x += 2;
```

This just adds 2 to the local variable, `x`, not to the array element. The value stored in `x` is overwritten by the value of the next element from `values` on the next iteration. In the next chapter you'll learn how you *can* change the values in a range using this loop.

The while Loop

The `while` loop uses a logical expression to control execution of the loop body. The general form of the `while` loop is shown in Figure 5-3.

This expression is evaluated at the beginning of each loop iteration. If it is **true** the loop continues, and if it is **false** execution continues with the statement after the loop.

```
while ( condition )
{
    // Loop statements    ...
}
// Next statement
```

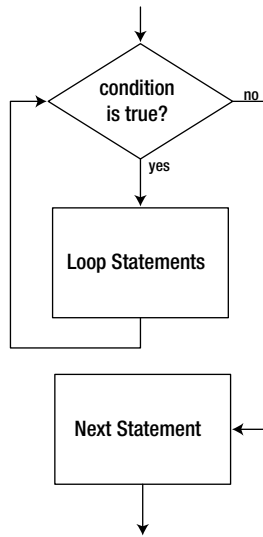


Figure 5-3. How the while loop executes

The flowchart in Figure 5-3 shows the logic of this loop. You can use any expression to control the loop, as long as it evaluates to a value of type `bool`, or can be implicitly converted to type `bool`. If the loop condition expression evaluates to a numerical value for example, the loop continues as long as the value is non-zero. A zero value ends the loop. Of course, `while` is a keyword, so you can't use it to name anything else.

You could implement a version of `Ex5_04.cpp` using a `while` loop to see how it differs:

```
// Ex5_05.cpp
// Using a while loop to calculate the sum of integers from 1 to n and n!
#include <iostream>
#include <iomanip>

int main()
{
    unsigned int limit {};
    std::cout << "This program calculates n! and the sum of the integers"
        << " up to n for values 1 to limit.\n";
    std::cout << "What upper limit for n would you like? ";
    std::cin >> limit;

    // Output column headings
    std::cout << std::setw(8) << "integer" << std::setw(8) << " sum"
        << std::setw(20) << " factorial" << std::endl;
    unsigned int n {};
    unsigned int sum {};
    unsigned long long factorial {1ULL};
```

```

while (++n <= limit)
{
    sum += n;                // Accumulate sum to current n
    factorial *= n;          // Calculate n! for current n
    std::cout << std::setw(8) << n << std::setw(8) << sum
               << std::setw(20) << factorial << std::endl;
}
}

```

The output from this program is the same as Ex5_04.cpp if you entered it correctly. The variables `n`, `sum`, and `factorial` are defined before the loop. Here the types of the variables can be different so `n` and `sum` are defined as `unsigned int`. The maximum value that can be stored in `factorial` limits the calculation so this remains as type `unsigned long long`. Because of the way the calculation is implemented, the counter `n` is initialized to zero. The while loop condition increments `n` and then compares the new value with `limit`. The loop continues as long as the condition is true, so the loop executes with values of `n` from 1 up to `limit`. When `n` reaches `limit+1`, the loop ends. The statements within the loop body are the same as in Ex5_04.cpp.

Allocating an Array at Runtime

The C++14 standard does not permit an array dimension to be specified at runtime; the array dimension must be a constant expression that can be evaluated by the compiler. However, some current C++ compilers do allow array dimensions at runtime because the current C standard, C99, permits this and a C++ compiler will typically compile C code too. The view at present is that this feature may be added to C++ in a future standard specification.

Determining the size of an array is a very useful feature so in case your compiler supports this I'll show how it works with an example. Keep in mind though that this is not strictly in conformance with the C++ language standard. Suppose you want to calculate the average height for a group of people, and you want to accommodate as many people as the user wants to enter heights for. As long as the user can input the number of heights to be processed, you can create an array that is an exact fit for the data that will be entered, like this:

```

size_t count {};
std::cout << "How many heights will you enter? ";
std::cin >> count;
unsigned int height[count];           // Create the array of count elements

```

The height array is created when the code executes and will have `count` elements. Because the array size is not known at compile-time, you cannot specify any initial values for the array.

Here's a working example using this:

```

// Ex5_06.cpp
// Allocating an array at runtime
#include <iostream>

int main()
{
    size_t count {};
    std::cout << "How many heights will you enter? ";
    std::cin >> count;
    unsigned int height[count];        // Create the array of count elements
}

```

```

// Read the heights
size_t entered {};
while(entered < count)
{
    std::cout << "Enter a height: ";
    std::cin >> height[entered];
    if(height[entered] < 0) // Make sure value is positive
    {
        ++entered;
    }
    else
    {
        std::cout << "A height must be positive - try again.\n";
    }
}

// Calculate the sum of the heights
unsigned int total {};
for(size_t i {}; i < count; ++i)
{
    total += height[i];
}
std::cout << "The average height is " << total/count << std::endl;
}

```

Here's some sample output:

```

How many heights will you enter? 6
Enter a height: 47
Enter a height: 55
Enter a height: 0
A height must be positive - try again.
Enter a height: 60
Enter a height: 78
Enter a height: 68
Enter a height: 56
The average height is 60

```

The height array is allocated using the value entered for count. The height values are read into the array in the while loop. Within the loop, the if statement checks whether the value entered is zero. When it is non-zero, the entered variable that counts the number of values entered so far is incremented. When the value is zero, a message is output and the next iteration executes without incrementing entered. Thus the new attempt at entering a value will be read into the current element of height, which will overwrite the zero value that was read on the previous iteration. A straightforward for loop aggregates the total of all the heights and this is used to output the average height. You could have used a range-based for loop here:

```

for(auto h : height)
{
    total += h;
}

```

Alternatively you could accumulate the total of the heights in the while loop and dispense with the for loop altogether. This would shorten the program significantly. The while loop would then look like this:

```
unsigned int total {};
size_t entered {};
while(entered < count)
{
    std::cout << "Enter a height: ";
    std::cin >> height[entered];
    if(height[entered])                // Make sure value is positive
    {
        total += height[entered++];
    }
    else
    {
        std::cout << "A height must be positive - try again.\n";
    }
}
```

Using the postfix increment operator in the expression for the index to the height array when adding the most recent element value to total ensures the current value of entered is used to access the array element before it is incremented for the next loop iteration.

■ **Note** Even if your compiler does not allow array dimensions to be determined at runtime, you can still achieve the same result using a vector, which I discuss later in this chapter.

The do-while Loop

The do-while loop is similar to the while loop in that the loop continues for as long as the specified loop condition remains true. However, the difference is that the loop condition is checked at the *end* of the do-while loop, rather than at the beginning, so the loop statement is always executed at least once.

The logic and general form of the do-while loop are shown in Figure 5-4. Note that the semicolon that comes after the condition between the parentheses is absolutely necessary. If you leave it out, the program won't compile.

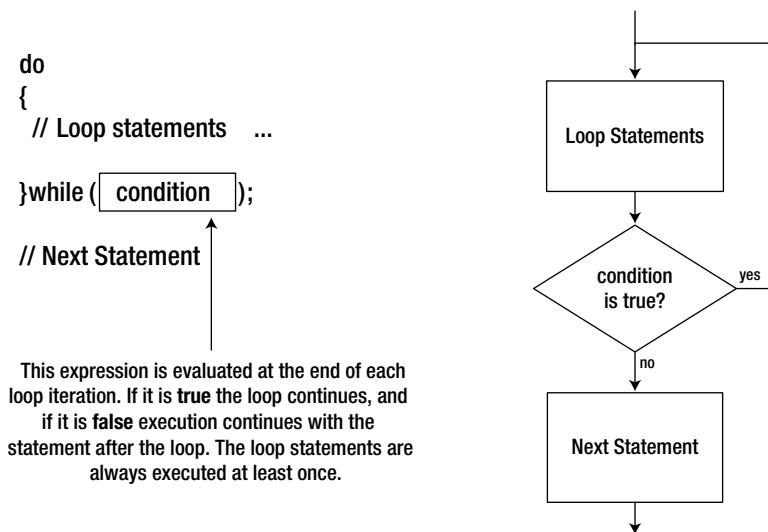


Figure 5-4. How a do-while loop executes

This kind of logic is ideal for situations where you have a block of code that you *always* want to execute once and may want to execute more than once. I can tell that you're not convinced that this is something that you'd ever need to do, so let's have another example.

This program will calculate the average of an arbitrary number of input values—temperatures, for example, without storing them. You have no way of knowing in advance how many values will be entered, but it's safe to assume that you'll always have at least one, because if you didn't, there'd be no point to running the program. That makes it an ideal candidate for a do-while loop. Here's the code:

```
// Ex5_07.cpp
// Using a do-while loop to manage input
#include <iostream>
#include <locale>                                // For tolower() function

int main()
{
    char reply {};                               // Stores response to prompt for input
    int count {};                                // Counts the number of input values
    double temperature {};                       // Stores an input value
    double average {};                           // Stores the total and average
    do
    {
        std::cout << "Enter a temperature reading: "; // Prompt for input
        std::cin >> temperature;                      // Read input value

        average += temperature;                       // Accumulate total of values
        ++count;                                       // Increment count

        std::cout << "Do you want to enter another? (y/n): ";
        std::cin >> reply;                             // Get response
    } while(tolower(reply) == 'y');
    std::cout << "The average temperature is " << average/count << std::endl;
}
```

A sample session with this program produces the following output:

```
Enter a temperature reading: 53
Do you want to enter another? (y/n): Y
Enter a temperature reading: 65.5
Do you want to enter another? (y/n): y
Enter a temperature reading: 74
Do you want to enter another? (y/n): Y
Enter a temperature reading: 69.5
Do you want to enter another? (y/n): n
The average temperature is 65.5
```

This program deals with any number of input values without prior knowledge of how many will be entered. After defining four variables that are required for the input and the calculation, the data values are read in a do-while loop. One input value is read on each loop iteration and at least one value will always be read, which is not unreasonable. The response to the prompt that is stored in `reply` determines whether or not the loop ends. If the reply is `y` or `Y`, the loop continues; otherwise the loop ends. Using the `tolower()` function that is declared in the `locale` header ensure either upper or lowercase is accepted. You could ensure that the response stored in `reply` is only upper or lowercase `y` or `n`; I'll leave that to you as a small exercise.

An alternative to using `tolower()` in the loop condition is to use a more complex expression for the condition. You could express the condition as `reply == 'y' || reply == 'Y'`. This ORs the two `bool` values that result from the comparisons so that either upper or lowercase `y` entered will result in `true`.

Nested Loops

You can place a loop inside another loop. In fact, you can nest loops within loops to whatever depth you require to solve your problem. Furthermore, nested loops can be of any kind: you can nest a `for` loop inside a `while` loop inside a do-while loop inside a range-based `for` loop, if you have the need. They can be mixed in any way that you want.

Nested loops are often applied in the context of arrays but they have many other uses. I'll illustrate how nesting works with an example that provides lots of opportunity for nesting loops. Multiplication tables are the bane of many children's lives at school, but you can easily use a nested loop to generate one:

```
// Ex5_08.cpp
// Generating multiplication tables
#include <iostream>
#include <iomanip>
#include <locale>

int main()
{
    size_t table {};           // Table size
    const size_t table_min {2}; // Minimum table size - at least up to the 2-times
    const size_t table_max {12}; // Maximum table size
    char reply {};             // Response to prompt

    do
    {
        std::cout << "What size table would you like ("
                    << table_min << " to " << table_max << ")? ";
        std::cin >> table;           // Get the table size
        std::cout << std::endl;
```

```

// Make sure table size is within the limits
if(table < table_min || table > table_max)
{
    std::cout << "Invalid table size entered. Program terminated." << std::endl;
    return 1;
}

// Create the top line of the table
std::cout << std::setw(6) << " |";
for(size_t i {1} ; i <= table ; ++i)
{
    std::cout << " " << std::setw(3) << i << " |";
}
std::cout << std::endl;

// Create the separator row
for(size_t i {} ; i <= table ; ++i)
{
    std::cout << "-----";
}
std::cout << std::endl;

for(size_t i {1} ; i <= table ; ++i)
{
    // Iterate over rows
    std::cout << " " << std::setw(3) << i << " |"; // Start the row

    // Output the values in a row
    for(size_t j {1} ; j <= table ; ++j)
    {
        std::cout << " " << std::setw(3) << i*j << " |"; // For each col.
    }
    std::cout << std::endl; // End the row
}

// Check if another table is required
std::cout << "\nDo you want another table (y or n)? ";
std::cin >> reply;
} while(tolower(reply) == 'y');
}

```

Here's an example of the output:

What size table would you like (2 to 12)? 4

	1	2	3	4
1	1	2	3	4
2	2	4	6	8
3	3	6	9	12
4	4	8	12	16

Do you want another table (y or n)? y
 What size table would you like (2 to 12)? 10

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

Do you want another table (y or n)? n

This example includes three standard headers, `iostream`, `iomanip`, and `locale`. Just as a refresher, the first is for stream input/output, the second is for stream manipulators, and the third provides the `tolower()` character conversion function along with many other functions relating to locales.

The input value for the size of the table is stored in `table`. A table will be output presenting the results of all products from 1×1 up to $\text{table} \times \text{table}$. The value entered is validated by comparing it with `table_min` and `table_max`. A table less than `table_min` doesn't make much sense and `table_max` represents a size that is the maximum that is likely to look reasonable when it is output. If `table` is not within range, the program ends with a return code value of 1 to indicate it's not a normal end.

The multiplication table is presented in the form of a rectangular table - what else! The values along the left column and the top row are the operand values in a multiplication. The value at the intersection of a row and column is the product of the row and columns values. The `table` variable is used as the iteration limit in the first for loop that creates the top line of the table. Vertical bars are used to separate columns and the use of the `setw()` manipulator makes all the columns the same width.

The next for loop creates a line of dash characters to separate the top row of multipliers from the body of the table. Each iteration adds six dashes to the row. By starting the count at zero instead of one, you output `table + 1` sets, one for the left column of multipliers, and one for each of the columns of table entries.

The final for loop that contains a nested for loop that outputs the left column of multipliers and the products that are the table entries. The nested loop outputs a complete table row, including the multiplier for the row in the leftmost column. The nested loop executes once for each iteration of the outer loop, so `table` rows are generated.

The code that creates a complete table is within a do-while loop. This provides for as many tables to be produced as required. If `y` or `Y` is entered in response to the prompt after a table has been output, another iteration of the do-while loop executes to allow another table to be created. This example demonstrates three levels of nesting - a for loop inside a for loop that is inside the do-while loop.

Skipping Loop Iterations

Situations arise where you want to skip one loop iteration and press on with the next. The `continue` statement does this:

```
continue;           // Go to the next iteration
```

When this statement executes within a loop, execution transfers immediately to the end of the current iteration. As long as the loop control expression allows it, execution continues with the next iteration. This is best understood

in an example. Let's suppose you want to output a table of characters with their character codes in hexadecimal and decimal format. Of course, you don't want to output characters that don't have a graphical representation — some of these, such as tabs and newline, would mess up the output. So, the program should output just the printable characters. Here's the code:

```
// Ex5_09.cpp
// Using the continue statement to display ASCII character codes
#include <iostream>
#include <iomanip>
#include <cctype>
#include <limits>

int main()
{
    std::cout << std::numeric_limits<unsigned char>::max() << std::endl;
    // Output the column headings
    std::cout << std::setw(11) << "Character " << std::setw(13) << "Hexadecimal "
        << std::setw(9) << "Decimal " << std::endl;
    std::cout << std::uppercase; // Uppercase hex digits

    // Output characters and corresponding codes
    unsigned char ch {};
    do
    {
        if (!std::isprint(ch)) // If it's not printable...
            continue;        // ...skip this iteration
        std::cout << std::setw(6) << ch // Character
            << std::hex << std::setw(12) << static_cast<int>(ch) // Hexadecimal
            << std::dec << std::setw(10) << static_cast<int>(ch) // Decimal
            << std::endl;
    } while (ch++ < std::numeric_limits<unsigned char>::max());
}
```

This outputs all the printable characters with code values from 0 to the maximum `unsigned char` value so it displays a handy list of the codes for the printable ASCII characters. The `do-while` loop is the most interesting bit. The variable, `ch`, varies from zero up to the maximum value for its type, `unsigned char`. You saw the `numeric_limits<>::max()` function back in Chapter 2, which returns the maximum value for the type you place between the angled brackets. Within the loop, you don't want to output details of any character that does not have a printable representation and the `isprint()` function that is declared in the `locale` header only returns true for printable characters. Thus the expression in the `if` statement will be true when `ch` contains the code for a character that is *not* printable. In this case the `continue` statement executes, which skips the rest of the code in the current loop iteration.

The `hex` and `dec` manipulators in the output statements set the output mode for integers to what you require. You have to cast the value of `ch` to `int` in the output statement to display as a numeric value; otherwise it would be output as a character. The judicious use of the `setw()` manipulator for the headings and the output in the loop ensures that everything lines up nicely.

Note that using `unsigned char` as the type for `ch` keeps the code simple. If you used `char` as the type for `ch`, you would need to provide for the possibility that it could be a signed or unsigned type. One complication of signed values is that you cannot cover the range by counting up from 0; adding 1 to the maximum value for signed `char`, 0111 1111 in binary which is 127, produces the minimum value, 1000 0000, which is -128.

You could deal with this by setting the initial value of `ch` to the minimum for the type using `numeric_limits<char>::min()`, but when you cast the negative code values to `int`, of course you get a negative result, so the hexadecimal codes would show the leading digits as `F`.

Note also that a `for` loop isn't suitable here with `ch` as type `unsigned char`. The condition in a `for` loop is checked before the loop block executes so you might be tempted to write the loop as follows:

```
for(unsigned char ch {}; ch <= std::numeric_limits<unsigned char>::max(); ++ch)
{
    // Output character and code...
}
```

This loop never ends. After executing the loop block with `ch` at the maximum value, the next increment of `ch` gives it a value of 0 so the second loop control expression is never false. You could make it work by using type `int` for the control variable in a `for` loop, then casting the value to type `unsigned char` when you want to output it as a character.

I'm sure you noticed when you run the example that the last character code in the output is 126. This is because the `isprint()` function is returning false for code values in excess of this. If you want to see character codes greater than 126 in the output, you could write the `if` statement in the loop as:

```
if(iscntrl(ch))
    continue;
```

This will only execute the `continue` statement for codes that represent control characters, which are code values from `0x00` to `0x1F`. You'll now see some weird and wonderful characters in the last 128 characters; what these are varies by locale.

Breaking Out of a Loop

Sometimes, you need to end a loop prematurely; something might arise within the loop statement that indicates there is no point in continuing. In this case, you can use the `break` statement. Its effect in a loop is much the same as it is in a `switch` statement; executing a `break` statement within a loop ends the loop immediately and execution continues with the statement following the loop. The `break` statement is used most frequently with an *indefinite loop*, so let's look next at what one of those looks like.

Indefinite Loops

An *indefinite loop* can potentially run forever. Omitting the second control expression in a `for` loop results in a loop that potentially executes an unlimited number of iterations. There has to be some way to end the loop within the loop block itself; otherwise the loop repeats indefinitely.

Indefinite loops have many practical uses: programs that monitor some kind of alarm indicator for instance or that collect data from sensors in an industrial plant. An indefinite loop can be useful when you don't know in advance how many loop iterations will be required, such as when you are reading a variable quantity of input data. In these circumstances, you code the exit from the loop within the loop block, not within the loop control expression.

In the most common form of the indefinite `for` loop, all the control expressions are omitted, as shown here:

```
for( ; ; )
{
    // Statements that do something...
    // ... and include some way of ending the loop
}
```

You still need the semicolons (;), even though no loop control expressions exist. The only way this loop can end is if some code within the loop terminates it.

You can have an indefinite while loop, too:

```
while(true)
{
    // Statements that do something...
    // ... and include some way of ending the loop
}
```

The loop condition is always true, so you have an indefinite loop. This is equivalent to the for loop with no control expressions. Of course, you can also have a version of the do-while loop that is indefinite, but it is not normally used because it has no advantages over the other two types of loop.

The obvious way to end an indefinite loop is to use the break statement. You could have used an indefinite loop in Ex5_08.cpp to allow several tries at entering a valid table size, instead of ending the program immediately. This loop would do it:

```
const size_t max_tries {3};
do
{
    for (size_t count {1} ; ; ++count) // Indefinite loop
    {
        std::cout << "What size table would you like ("
                    << table_min << " to " << table_max << ")? ";
        std::cin >> table; // Get the table size

        // Make sure table size is within the limits
        if (table >= table_min && table <= table_max)
        {
            break; // Exit the input loop
        }
        else if (count < max_tries)
        {
            std::cout << "Invalid input - try again.\n";
        }
        else
        {
            std::cout << "Invalid table size entered - yet again! " << "\nSorry, only "
                    << max_tries << " goes - program terminated." << std::endl;
            return 1;
        }
    }
}
```

This indefinite for loop could replace the code at the beginning of the do-while loop in Ex5_08.cpp that handles input of the table size. This allows up to max_tries attempts to enter a valid table size. A valid entry executes the break statement, which terminates this loop and continues with the next statement in the do-while loop.

Here's an example that uses an indefinite while loop to sort the contents of an array in ascending sequence:

```
// Ex5_10.cpp
// Sorting an array in ascending sequence - using an indefinite while loop
#include <iostream>
#include <iomanip>
```

```

int main()
{
    const size_t size {1000};           // Array size
    double x[size] {};                  // Stores data to be sorted
    double temp {};                     // Temporary store for a value
    size_t count {};                    // Number of values in array

    while (true)
    {
        std::cout << "Enter a non-zero value, or 0 to end: ";
        std::cin >> temp;
        if (!temp)
            break;

        x[count++] = temp;
        if (count == size)
        {
            std::cout << "Sorry, I can only store " << size << " values.\n";
            break;
        }
    }
    std::cout << "Starting sort." << std::endl;
    bool swapped{ false };              // true when values are not in order
    while (true)
    {
        for (size_t i {} ; i < count - 1 ; ++i)
        {
            if (x[i] > x[i + 1])
            { // Out of order so swap them
                temp = x[i];
                x[i] = x[i+1];
                x[i + 1] = temp;
                swapped = true;
            }
        }
        if (!swapped)                    // If there were no swaps
            break;                       // ...they are in order...
        swapped = false;                 // ...otherwise, go round again.
    }

    std::cout << "Your data in ascending sequence:\n"
        << std::fixed << std::setprecision(1);
    const size_t perline {10};          // Number output per line
    size_t n {};                         // Number on current line
    for (size_t i {} ; i < count ; ++i)
    {
        std::cout << std::setw(8) << x[i];
        if (++n == perline)              // When perline have been written...
        {

```

```

        std::cout << std::endl;           // Start a new line and...
        n = 0;                           // ...reset count on this line
    }
}
std::cout << std::endl;
}

```

Typical output looks like this:

```

Enter a non-zero value, or 0 to end: 44
Enter a non-zero value, or 0 to end: -7.8
Enter a non-zero value, or 0 to end: 56.3
Enter a non-zero value, or 0 to end: 75.2
Enter a non-zero value, or 0 to end: -3
Enter a non-zero value, or 0 to end: -2
Enter a non-zero value, or 0 to end: 66
Enter a non-zero value, or 0 to end: 6.7
Enter a non-zero value, or 0 to end: 8.2
Enter a non-zero value, or 0 to end: -5
Enter a non-zero value, or 0 to end: 0
Starting sort.
Your data in ascending sequence:
    -7.8    -5.0    -3.0    -2.0     6.7     8.2    44.0    56.3    66.0    75.2

```

The code limits the number of values than can be entered to `size`, which is set to 1000. Only users with amazing keyboard skill and persistence will find out about this. Data entry is managed in the first while loop. This loop runs until either 0 is entered, or the array, `x`, is full because `size` values have been entered, in the latter instance, the user will see a message, indicating the limit. This is rather wasteful with memory but you'll learn how you can avoid this in such circumstances later in this chapter.

Each value is read into the variable `temp`. This allows the value to be tested for zero before it is stored in the array. The `!` operator requires an operand of type `bool`, so the compiler will insert an implicit conversion of the value of `temp` to type `bool`. You'll recall that a numerical value of zero converts to the `bool` value `false`, and non-zero converts to `true`. Thus the `if` expression will be `true` when zero is entered. Each value is stored in the element of the array `x` at index `count`. `count` is incremented after it is used to index the array, so it represents the number of elements in the array when the following `if` statement executes.

The elements are sorted in ascending sequence in the next indefinite while loop. Ordering the values of the array elements is carried out in the nested for loop that iterates over successive pairs of elements, and checking whether they are in ascending sequence. If a pair of elements contain values that are not in ascending sequence, the values are swapped to order them correctly. The `bool` variable, `swapped`, records whether it was necessary to interchange any elements in any complete execution of the nested for loop. If it wasn't, then the elements are in ascending sequence and the `break` statement is executed to exit the while loop. If any pair had to be interchanged, `swapped` will be `true` so another iteration of the while loop will execute, and this causes the for loop to run through pairs of elements again.

This sorting method is called the *bubble sort* because elements gradually "bubble up" to their correct position in the array. It's not the most efficient sorting method, but it has the merit that it is very easy to understand and it's a good demonstration of yet another use for an indefinite loop.

Arrays of Characters

An array of elements of type `char` can have a dual personality. It can simply be an array of characters, in which each element stores one character, *or* it can represent a string. In the latter case, the characters in the string are stored in successive array elements, followed by a special string termination character called the *null character* that you write as `'\0'`; this marks the end of the string.

A character string that is terminated by `'\0'` is a *C-style string*. This contrasts with the `string` type from the Standard Library that I'll explain in detail in Chapter 7. Objects of type `string` don't need a string termination character and are much more flexible and convenient for string manipulation than using arrays of type `char`. For the moment, I'll introduce C-style strings in the context of arrays in general and return to these and to type `string` in detail in Chapter 7.

You can define and initialize an array of elements of type `char` like this:

```
char vowels[5] {'a', 'e', 'i', 'o', 'u'};
```

This isn't a string — it's just an array of five characters. Each array element is initialized with the corresponding character from the initializer list. As with numeric arrays, if you provide fewer initializing values than there are array elements, the elements that don't have explicit initial values will be initialized with the equivalent of zero, which is the null character, `'\0'` in this case. This means that if there are insufficient initial values, the array will effectively contain a string. For example:

```
char vowels[6] {'a', 'e', 'i', 'o', 'u'};
```

The last element will be initialized with `'\0'`. The presence of the null character means that this can be treated as a C-style string. Of course, you can still regard it as an array of characters.

You could leave it to the compiler to set the size of the array to the number of initializing values:

```
char vowels[] {'a', 'e', 'i', 'o', 'u'};    // An array with five elements
```

This also defines an array of five characters initialized with the vowels in the initializer list.

You can also declare an array of type `char` and initialize it with a string literal, for example:

```
char name[10] {"Mae West"};
```

This creates a C-style string. Because you're initializing the array with a string literal, the null character will be stored in the element following the last string character, so the contents of the array will be as shown in Figure 5-5.

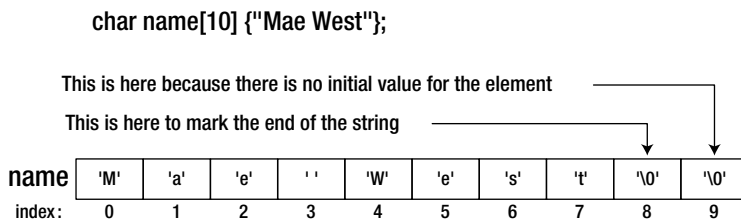


Figure 5-5. An array of elements of type `char` initialized with a string literal

Of course, you can leave the compiler to set the size of the array when you initialize it with a string:

```
char name[] {"Mae West"};
```

This time, the array will have nine elements: eight to store the characters in the string, plus an extra element to store the string termination character. Of course, you could have used this approach when you declared the `vowels` array:

```
char vowels[] {"aeiou"};                // An array with six elements
```

There's a significant difference between this and the previous definition for `vowels` without an explicit array dimension. Here you're initializing the array with a string literal. This has `'\0'` appended to it to mark the end of the string, so the `vowels` array will contain six elements. The array created with the earlier definition will only have five elements and can't be used as a string.

You can output a string stored in an array just by using the array name. The string in the `name` array, for example, could be written to `cout` with this statement:

```
std::cout << name << std::endl;
```

This will display the entire string of characters, up to the `'\0'`. There *must* be a `'\0'` at the end. If there isn't, you'll continue to output characters from successive memory locations until a string termination character turns up or an illegal memory reference occurs.

■ **Caution** You can't output the contents of an array of a numeric type by just using the array name. This only works for `char` arrays.

This example analyzes an array of elements of type `char` to work out how many vowels and consonants are used in it:

```
// Ex5_11.cpp
// Classifying the letters in a string
#include <iostream>
#include <locale>

int main()
{
    const int maxlength {100};           // Array size
    char text[maxlength] {};            // Array to hold input string

    std::cout << "Enter a line of text:" << std::endl;

    // Read a line of characters including spaces
    std::cin.getline(text, maxlength);
    std::cout << "You entered:\n" << text << std::endl;
    size_t vowels {};                    // Count of vowels
    size_t consonants {};                // Count of consonants
    for(int i {} ; text[i] != '\0' ; i++)
    {
        if(isalpha(text[i]))             // If it is a letter...
```



```

{
    switch(tolower(text[i]))
    {
        // ...check lowercase...
        case 'a': case 'e': case 'i': case 'o': case 'u':
            vowels++;
            // ...it is a vowel
            break;

        default:
            consonants++;
            // ...it is a consonant
    }
}
}
std::cout << "Your input contained " << vowels << " vowels and "
    << consonants << " consonants." << std::endl;
}

```

Here's an example of the output:

```

Enter a line of text:
A rich man is nothing but a poor man with money.
You entered:
A rich man is nothing but a poor man with money.
Your input contained 14 vowels and 23 consonants.

```

The text array of type `char` elements has the size defined by a `const` variable, `max_length`. This determines the maximum length string that can be stored, including the terminating null character, so the longest string can contain `max_length-1` characters.

You can't use the extraction operator to read the input, because it won't read a string containing spaces; any whitespace character terminates the input operation with the `>>` operator. The `getline()` function for `cin` that is defined in the `iostream` header reads a sequence characters, including spaces. By default, the input ends when a newline character, `'\n'`, is read, which will be when you press the *Enter* key. The `getline()` function expects two arguments between the parentheses. The first argument specifies where the input is to be stored, which in this case is the text array. The second argument specifies the maximum number of characters that you want to store. This includes the string termination character, `'\0'`, which will be automatically appended to the end of the input.

Although you haven't done so here, you can optionally supply a *third* argument to the `getline()` function. This specifies an alternative to `'\n'` to indicate the end of the input. For example, if you want the end of the input string to be indicated by an asterisk for example, you would use this statement to read the input:

```
std::cin.getline(text, maxLength, '*');
```

This would allow multiple lines of text to be entered because the `'\n'` that results from pressing *Enter* would no longer terminate the input operation. Of course, the total number of characters that you can enter in the read operation is still limited by `maxLength`.

Just to show that you can, the program output the string that was entered using just the array name, `text`. The text string is then analyzed in a straightforward manner in the `for` loop. The second control expression within the loop will be `false` when the character at the current index, `i`, is the null character, so the loop ends when the null character is reached. To work out the number of vowels and consonants, you only need to inspect alphabetic characters, and the `isalpha()` only returns `true` for alphabetic characters. Thus the `switch` statement only executes for letters. Converting the `switch` expression to lowercase avoids having to write cases for uppercase as well as lowercase letters. Any vowel will select the first case and the default case is selected by anything that isn't a vowel, which must be a consonant of course.

Multidimensional Arrays

All the arrays so far have required a single index value to select an element. Such an array is called a *one-dimensional array*, because varying one index can reference all the elements. You can also define arrays that require two or more index values to access an element. These are referred to generically as *multidimensional arrays*. An array that requires two index values to reference an element is called a *two-dimensional array*. An array needing three index values is a *three-dimensional array*, and so on for as many dimensions as you think you can handle.

Suppose, as an avid gardener, that you want to record the weights of the carrots you grow in your small vegetable garden. To store the weight of each carrot, which you planted in three rows of four, you could define a two-dimensional array:

```
double carrots[3][4] {};
```

This defines an array with 3 rows of 4 elements and initializes all elements to zero. To reference a particular element of the `carrots` array, you need two index values. The first index specifies the row, from 0 to 2, and the second index specifies a particular carrot in that row, from 0 to 3. To store the weight of the third carrot in the second row, you could write:

```
carrots[1][2] = 1.5;
```

Figure 5-6 shows the arrangement of this array in memory. The rows are stored contiguously in memory. As you can see, the two-dimensional array is effectively a *one-dimensional array* of three elements, each of which is a one-dimensional array with four elements. You have an array of three arrays that each has four elements of type `double`. Figure 5-6 also indicates that you can use the array name plus a *single* index value between square brackets to refer to an entire row.

```
double carrots [3][4] {}; // Array with 3 rows of 4 elements
```

You can refer to this row as `carrots [0]`

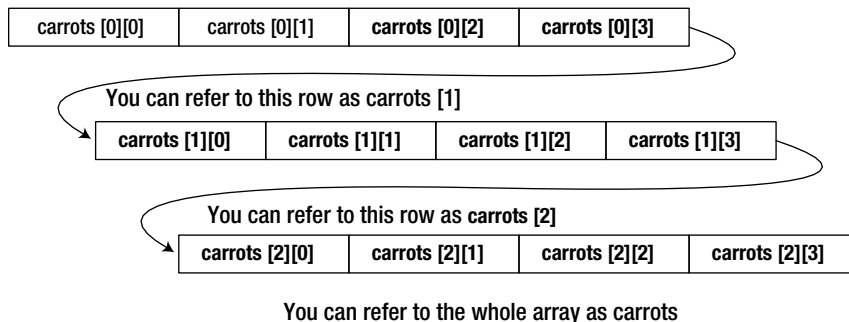


Figure 5-6. Elements in a two-dimensional array

You use two index values to refer to an element. The second index selects an element within the row specified by the first index; the second index varies most rapidly as you progress from one element to the next in memory. You can also envisage a two-dimensional array as a rectangular arrangement of elements the array from left to right, where the first index specifies a row and the second index corresponds to a column. Figure 5-7 illustrates this. With arrays of more than two dimensions, the rightmost index value is always the one that varies most rapidly, and the leftmost index varies least rapidly.

```
double carrots [3][4] { };
```

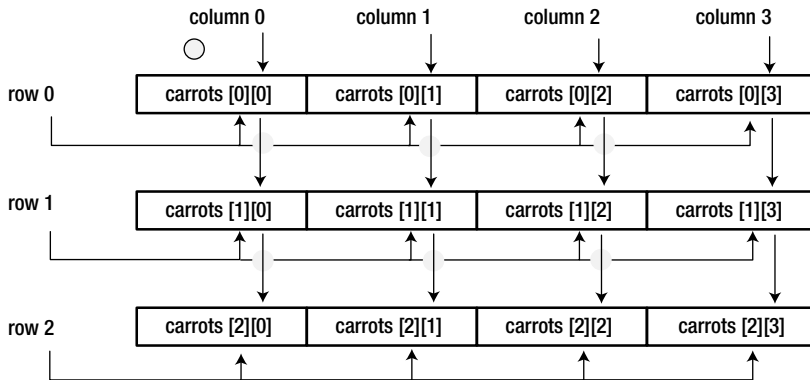


Figure 5-7. Rows and columns in a two-dimensional array

The array name by itself references the entire array. Note that with this array, you can't display the contents of either a row or the whole array using this notation. For example:

```
std::cout << carrots << std::endl;           // Not what you may expect!
```

This statement will output a single hexadecimal value, which happens to be the address in memory of the first element of the array. You'll see why this is the case when I discuss pointers in the next chapter. Arrays of type `char` are a little different, as you saw earlier.

To display the entire array, one row to a line, you must write something like this:

```
for(size_t i { } ; i < 3 ; ++i)                // Iterate over rows
{
    for(size_t j { } ; j < 4 ; ++j)              // Iterate over elements within the row
    {
        std::cout << std::setw(12) << carrots[i][j];
    }
    std::cout << std::endl;                      // A new line for a new row
}
```

This uses magic numbers, 3 and 4, which you can avoid by using the `sizeof` operator:

```
for(size_t i { } ; i < sizeof(carrots)/sizeof(carrots[0]) ; ++i)
{
    for(size_t j { } ; j < sizeof(carrots[0])/sizeof(double) ; ++j)
    {
        std::cout << std::setw(12) << carrots[i][j];
    }
    std::cout << std::endl;
}
```

You could use `sizeof(carrots[0][0])` in place of `sizeof(double)` in the nested loop. Of course, it would be better still not to use magic numbers for the array dimension sizes in the first place, so you *should* define the array as:

```
const size_t nrows {3};           // Number of rows in the array
const size_t ncols {4};          // Number of columns, or number of elements per row
double carrots[nrows, ncols] {};
```

Now you can output elements values like this:

```
for(size_t i {}; i < nrows ; ++i)    // Iterate over rows
{
    for(size_t j {} ; j < ncols ; ++j) // Iterate over elements within the row
    {
        std::cout << std::setw(12) << carrots[i][j];
    }
    std::cout << std::endl;          // A new line for a new row
}
```

Defining an array of three dimensions just adds another set of square brackets. You might want to record three temperatures per day, seven days a week, for 52 weeks of the year. You could declare the following array to store such data as type `long`:

```
long temperatures[52][7][3] {};
```

The array stores three values in each row. There are seven such rows for a whole week's data and 52 sets of these for all the weeks in the year. This array will have a total of 1,092 elements of type `long`. They will all be initialized with zero. To display the middle temperature for day 3 of week 26, you could write this:

```
std::cout << temperatures[25][2][1] << std::endl;
```

Remember that all the index values start at 0, so the weeks run from 0 to 51, the days run from 0 to 6, and the samples in a day run from 0 to 2.

Initializing Multidimensional Arrays

You have seen that an empty initializer list initializes an array with any number of dimensions to zero. It's gets a little more complicated when you want initial values other than zero. The way in which you specify initial values for a multidimensional array derives from the notion that a two-dimensional array is an array of one-dimensional arrays. The initializing values for a one-dimensional array are written between braces and separated by commas. Following on from that, you could declare and initialize the two-dimensional `carrots` array, with this statement:

```
double carrots[3][4] {
    {2.5, 3.2, 3.7, 4.1}, // First row
    {4.1, 3.9, 1.6, 3.5}, // Second row
    {2.8, 2.3, 0.9, 1.1}  // Third row
};
```

I used explicit array dimensions to keep the code fragments short and simple. Each row is a one-dimensional array, so the initializing values for each row are contained within their own set of braces. These three initializer lists are themselves contained within a set of braces, because the two-dimensional array is a one-dimensional array of one-dimensional arrays. You can extend this principle to any number of dimensions — each extra dimension requires another level of nested braces enclosing the initial values.

A question that may immediately spring to mind is, “What happens when you omit some of the initializing values?” The answer is more or less what you might have expected from past experience. Each of the innermost pairs of braces contains the values for the elements in the rows. The first list corresponds to `carrots[0]`, the second to `carrots[1]`, and the third to `carrots[2]`. The values between each pair of braces are assigned to the elements of the corresponding row. If there aren’t enough to initialize all the elements in the row, then the elements without values will be initialized to 0.

Let’s look at an example:

```
double carrots[3][4] {
    {2.5, 3.2      },      // First row
    {4.1          },      // Second row
    {2.8, 2.3, 0.9 }      // Third row
};
```

The first two elements in the first row have initial values, whereas only one element in the second row has an initial value, and three elements in the third row have initial values. The elements without initial values in each row will therefore be initialized with zero, as shown in Figure 5-8.

<code>carrots[1][0]</code>	<code>carrots[0][1]</code>	<code>carrots[0][2]</code>	<code>carrots[0][3]</code>
2.5	3.2	0.0	0.0

<code>carrots[1][0]</code>	<code>carrots[1][1]</code>	<code>carrots[1][2]</code>	<code>carrots[1][3]</code>
4.1	0.0	0.0	0.0

<code>carrots[2][0]</code>	<code>carrots[2][1]</code>	<code>carrots[2][2]</code>	<code>carrots[2][3]</code>
2.8	2.3	0.9	0.0

Figure 5-8. Omitting initial values for a two-dimensional array

If you don’t include sufficient sets of braces to initialize all of the rows in the array, the elements in the rows without braces enclosing initializing values will all be set to 0. If you include several initial values in the initializer list but omit the nested braces enclosing values for the rows, values are assigned sequentially to the elements, as they’re stored in memory — with the rightmost index varying most rapidly. For example, suppose you define the array like this:

```
double carrots[3][4] {1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7};
```

The first four values in the list will initialize elements in row 0. The last three values in the list will initialize the first three elements in row 1. The remaining elements will be initialized with zero.

Setting Dimensions by Default

You can let the compiler determine the size of the first (leftmost) dimension of an array with any number of dimensions from the set of initializing values. Clearly, the compiler can only ever determine one of the dimensions in a multidimensional array, and it has to be the first. If you were to supply 12 initial values for a two-dimensional array, for instance, there’s no way for the compiler to know whether the array should be three rows of four elements, six rows of two elements, or indeed any combination that amounts to 12 elements.

You could define the two-dimensional `carrots` array with this statement:

```
double carrots[][4] {
    {2.5, 3.2           },    // First row
    {4.1                },    // Second row
    {2.8, 2.3, 0.9      }    // Third row
};
```

The array will have three rows, as before, because there are three sets of braces within the outer pair. If there were only two sets, the array would have two rows. The number of inner pairs of braces determines the number of rows.

Here's an example of defining a three-dimensional array:

```
int numbers[][3][4] {
    {
        { 2, 4, 6, 8},
        { 3, 5, 7, 9},
        { 5, 8, 11, 14}
    },
    {
        {12, 14, 16, 18},
        {13, 15, 17, 19},
        {15, 18, 21, 24}
    }
};
```

This array has three dimensions of sizes 2, 3, and 4. The outer braces enclose two further sets of braces, and each of these in turn contains three sets, each of which contains the four initial values for the corresponding row. As this simple example demonstrates, initializing arrays of three dimensions or more gets increasingly complicated, and you need to take great care when placing the braces enclosing the initial values. The braces are nested to as many levels as there are dimensions in the array.

Multidimensional Character Arrays

You can define arrays of two or more dimensions to hold any type of data. A two-dimensional array of type `char` is interesting, because it can be an array of C-style strings. When you initialize a two-dimensional array of `char` elements with string literals, you don't need the braces around the literal for a row—the double quotes delimiting the literal do the job of the braces in this case, for example:

```
char stars[][80] {
    "Robert Redford",
    "Hopalong Cassidy",
    "Lassie",
    "Slim Pickens",
    "Boris Karloff",
    "Oliver Hardy"
};
```

This array will have six rows because there are six string literals as initial values. Each row stores a string containing the name of a movie star, and a terminating null character, `'\0'`, will be appended to each string. Each row will accommodate up to 80 characters according to the row dimension you've specified. We can see this applied in an example:

```
// Ex5_12.cpp
// Working with strings in an array
#include <iostream>

int main()
{
    const size_t max_str {80};      // Maximum string length including \0
    char stars[][max_str] {
        "Fatty Arbuckle", "Clara Bow",
        "Lassie",         "Slim Pickens",
        "Boris Karloff",  "Mae West",
        "Oliver Hardy",   "Greta Garbo"
    };

    size_t choice {};

    std::cout << "Pick a lucky star! Enter a number between 1 and "
               << sizeof(stars)/sizeof(stars[0]) << ": ";
    std::cin  >> choice;

    if(choice >= 1 && choice <= sizeof stars/sizeof stars[0])
    {
        std::cout << "Your lucky star is " << stars[choice - 1] << std::endl;
    }
    else
    {
        std::cout << "Sorry, you haven't got a lucky star." << std::endl;
    }
}
```

Typical output from this program is:

```
Pick a lucky star! Enter a number between 1 and 8: 6
Your lucky star is Mae West
```

Apart from its incredible inherent entertainment value, the main point of interest in the example is the definition of the array, `stars`. It's a two-dimensional array of `char` elements, which can hold multiple strings, each of which can contain up to `max_str` characters, including the terminating null that's automatically added by the compiler. The initializing strings for the array are enclosed between braces and separated by commas. Because the size of the first array dimension is omitted, the compiler creates the array with the number of rows necessary to accommodate all the initializing strings. As you saw earlier, you can only omit the size of the first dimension; you must specify the sizes of any other dimensions that are required.

The upper limit on the integer to be entered is given by the expression `sizeof(stars)/sizeof(stars[0])`. This results the number of rows in the array because it divides the memory occupied by the entire array by the memory occupied by a row. Thus the statement automatically adapts to any changes you may make to the number of literals in the initializing list. You use the same technique in the `if` statement that arranges for the output to be

displayed. The `if` condition checks that the integer that was entered is within range before attempting to display a name. When you need to reference a string for output, you only need to specify the first index value. A single index selects a particular 80-element subarray, and because this contains a string, the operation will output the contents of each element up to the terminating null character. The index is specified as `choice-1` because the choice values start from 1, whereas the index values need to start from 0. This is quite a common idiom when you're programming with arrays.

■ **Note** A disadvantage of using arrays as in this example is the memory that is almost invariably left unused. All of your strings are less than 80 characters, and the surplus elements in each row of the array are wasted. You'll see a better way of dealing with situations like this in the next chapter.

Alternatives to Using an Array

The Standard Library provides alternatives to the arrays that are part of the C++ language. The alternatives are defined in the subset of the Standard Library that is referred to as the Standard Template Library (STL). As its name suggests, the STL is essentially a collection of templates for classes and functions. In particular, the STL defines templates for container classes that offer a variety of ways to organize data. Most of the STL is beyond the scope of this book but I'll introduce you to the container classes that provide an alternative to standard arrays because they are easy to work with, much safer to use and provide more flexibility. The discussion won't be exhaustive; just enough for you to use them like the array types built-in to the language. I'll discuss two types of containers, `std::array<T,N>` and `std::vector<T>`. These are class templates that the compiler uses to create a type based on what you specify for the parameters, `T` and `N`. I'll refer to these container types in the text without the `std` namespace qualification but it will be there in the code. I'll also omit the type parameters when referring to them generically, as `array<>` and `vector<>`.

Using `array<T,N>` Containers

The `array<T,N>` template is defined in the `array` header so you must include this in a source file to use the container type. An `array<T,N>` container is a fixed sequence of `N` elements of type `T`, so it's just like a regular array except that you specify the type and size a little differently. Here's how you create an `array<>` of 100 elements of type `double`:

```
std::array<double, 100> values;
```

This creates an object that has 100 elements of type `double` that are initialized to zero by default. The specification for the parameter `N` must be a constant expression and the number of elements cannot be changed. Any array container you create without specifying explicit initial values will have elements set to the equivalent of zero for the type of element.

Of course, you can initialize the elements in the definition, just like a normal array:

```
std::array<double, 100> values {0.5, 1.0, 1.5, 2.0}; // 5th and subsequent elements are 0.0
```

The four values in the initializer list are used to initialize the first four elements; subsequent elements will be zero.

You can set all the elements to some given value using the `fill()` function for the `array<>` object. For example:

```
values.fill(3.1415926); // Set all element to pi
```


The `fill()` function belongs to the array object. The function is a member of the class type, `array<double, 100>`, of the object all array objects will have a `fill()` member, as well as other members. The period between the name of the variable, `values`, that contains the object, and the member function, `fill()`, is called the direct member selection operator. This operator is used to access members of a class object. Executing this statement causes all elements to be set to the value you pass as the argument to the `fill()` function. Obviously, this must be of a type that can be stored in the container. You'll understand the relationship between the `fill()` function and an `array<>` object better after Chapter 11.

You can access and use elements using an index in the same way as for a standard array, for example:

```
values[4] = values[3] + 2.0*values[1];
```

The fifth element is set to the value of the expression that is the right operand of the assignment.

The `size()` function for an `array<>` object returns the number of elements as type `size_t`, so you could sum the elements in the `values` object like this:

```
double total {};
for(size_t i {} ; i < values.size() ; ++i)
{
    total += values[i];
}
```

The `size()` function provides the first advantage over a standard array because it means that an `array<>` object knows how many elements there are. This is not true for a standard array. As you'll learn in Chapter 8, when you pass a standard array to a function, you must also pass the array size as an additional argument; otherwise the function cannot determine the number of elements. When you use an `array<>` object, the object knows its size so there is no need to supply the size separately.

An `array<>` object is a range, so you can use the range-based for loop to sum the elements more simply:

```
double total {};
for(auto value : values)
{
    total += value;
}
```

Accessing the elements in an `array<>` object using an index between square brackets doesn't check for invalid index values. The `at()` function for an `array<>` object does, and therefore will detect attempts to use an index value outside the legitimate range. The argument to the `at()` function is an index, the same as when you use square brackets, so you could write the for loop that totals the elements like this:

```
double total {};
for(size_t i {} ; i < values.size() ; ++i)
{
    total += values.at(i);
}
```

The expression `values.at(i)` is equivalent to `values[i]`, but with the added security that the value of `i` will be checked. For example, this code will fail:

```
double total {};
for(size_t i {} ; i <= values.size() ; ++i)
{
    total += values.at(i);
}
```

The second loop condition now using the `<=` operator allows `i` to reference beyond the last element. This will result in the program terminating at runtime with a message relating to an exception of type `std::out_of_range` being thrown. Throwing an exception is a mechanism for signaling exceptional error conditions. You'll learn more about exceptions in Chapter 15. If you code this using values `[i]`, the program will silently access the element beyond the end of the array and add whatever it contains to total. The `at()` function provides a further advantage over standard arrays.

You can compare entire `array<>` containers using any of the comparison operators as long as the containers are of the same size and store elements of the same type. For example:

```
std::array<double,4> these {1.0, 2.0, 3.0, 4.0};
std::array<double,4> those {1.0, 2.0, 3.0, 4.0};
std::array<double,4> them {1.0, 3.0, 3.0, 2.0};

if (these == those) std::cout << "these and those are equal." << std::endl;
if (those != them)  std::cout << "those and them are not equal." << std::endl;
if (those < them)   std::cout << "those are less than them." << std::endl;
if (them > those)   std::cout << "them are greater than those." << std::endl;
```

Containers are compared element by element. For a true result for `==`, all pairs of corresponding elements must be equal. For inequality, at least one pair of corresponding elements must be different for a true result. For all the other comparisons, the first pair of elements that differ produces the result. This is essentially the way in which words in a dictionary are ordered where the first pair of corresponding letters that differ in two words determines their order. All the comparisons in the code fragment are true, so all four messages will be output when this executes.

Unlike standard arrays, you can assign one `array<>` container to another, as long as they both store the same number of elements of the same type. For example:

```
them = those;           // Copy all elements of those to them
```

Using the `array<>` container carries very little overhead compared to a standard array so there's every reason to use a container in preference to a standard array in your code. Here's an example that demonstrates `array<>` containers in action:

```
// Ex5_13.cpp
// Using array<T,N> to create Body Mass Index (BMI) table
// BMI = weight/(height*height)
// weight in kilograms, height in meters

#include <iostream>
#include <iomanip>
#include <array>                                // For array<T,N>
using std::cout;
using std::endl;
using std::setw;

int main()
{
    const unsigned int min_wt {100U};           // Minimum weight in table
    const unsigned int max_wt {250U};           // Maximum weight in table
    const unsigned int wt_step {10U};
    const size_t wt_count {1 + (max_wt - min_wt) / wt_step};
```

```

const unsigned int min_ht {48U};           // Minimum height in table
const unsigned int max_ht {84U};           // Maximum height in table
const unsigned int ht_step {2U};
const size_t ht_count { 1 + (max_ht - min_ht) / ht_step };

const double lbs_per_kg {2.2};
const double ins_per_m {39.37};
std::array<unsigned int, wt_count> weight_lbs {};
std::array<unsigned int, ht_count> height_ins {};

// Create weights from 100lbs in steps of 10lbs
for (size_t i{}, w{ min_wt } ; i < wt_count ; w += wt_step, ++i)
{
    weight_lbs[i] = w;
}
// Create heights from 48 inches in steps of 2 inches
size_t i {};
for (unsigned int h{ min_ht } ; h <= max_ht ; h += ht_step)
{
    height_ins.at(i++) = h;
}

// Output table headings
cout << setw(7) << " |";
for (auto w : weight_lbs)
    cout << setw(5) << w << " |";
cout << endl;

// Output line below headings
for (size_t i{1} ; i < wt_count ; ++i)
    cout << "-----";
cout << endl;

double bmi {};                               // Stores BMI
unsigned int feet {};                         // Whole feet for output
unsigned int inches {};                      // Whole inches for output
const unsigned int inches_per_foot {12U};
for (auto h : height_ins)
{
    feet = h / inches_per_foot;
    inches = h % inches_per_foot;
    cout << setw(2) << feet << "'" << setw(2) << inches << "\"" << "|";
    cout << std::fixed << std::setprecision(1);
    for (auto w : weight_lbs)
    {
        bmi = h / ins_per_m;
        bmi = (w / lbs_per_kg) / (bmi*bmi);
        cout << setw(2) << " " << bmi << " |";
    }
    cout << endl;
}

```

```
// Output line below table
for (size_t i {1} ; i < wt_count ; ++i)
    cout << "-----";
cout << "\nBMI from 18.5 to 24.9 is normal" << endl;
}
```

I leave you to run the program to see the output because it takes quite a lot of space. The using directives reduce the line length for the output statements so they fit within the page width. There are two sets of four `const` variables defined that relate to the range of weights and heights for the BMI table. The weights and heights are stored in `array<>` containers with elements of type `unsigned int` because all the weights and heights are integral. The containers are initialized with the appropriate values in `for` loops. The second loop that initializes `height_ins` uses a different approach to setting the values just to demonstrate the `at()` function. This is appropriate in this loop because the loop is not controlled by the index limits for the container so it's possible that a mistake could be made that would use an index outside the legal range for the container. The program would be terminated if this occurred, which would not be the case using square brackets to reference an element.

The next two `for` loops output the table column headings and a line to separate the headings from the rest of the table. The table is created using nested range-based `for` loops. The outer loop iterates over the heights and outputs the height in the leftmost column in feet and inches. The inner loop iterates over the weights and outputs a row of BMI values for the current height.

Using `std::vector<T>` Containers

The `vector<T>` container is a sequence container that is like an `array<T, N>` container except that the size can grow automatically to accommodate any number of elements; hence the requirement for only the type parameter `T` - there's no need for the `N` with a `vector<>`. Additional space is allocated automatically when required, so a `vector<>` can grow as you add more elements. Using the `vector<>` container needs the `vector` header to be included in your source file.

Here's an example of creating a `vector<>` container to store values of type `double`:

```
std::vector<double> values;
```

This has no space for elements allocated so memory will need to be allocated dynamically when you add the first data item. You can add an element using the `push_back()` function for the container object. For example:

```
values.push_back(3.1415926);           // Add an element to the end of the vector
```

The `push_back()` function adds the value you pass as the argument — `3.1415926` in this case — as a new element at the end of the existing elements. Since there are no existing elements here, this will be the first, and this will cause memory to be allocated.

Allocating memory is relatively expensive in time so you don't want to it to occur more often than necessary. You can reduce the likelihood by creating a `vector<>` with a predefined number of elements, like this:

```
std::vector<double> values(20);        // Capacity is 20 double values
```

This container starts out with 20 elements that are initialized with zero by default - just like an `array<>`. If you don't like zero as the default value for elements, you can specify a value that will apply for all elements:

```
std::vector<long> numbers(20, 99L);    // Capacity is 20 long values - all 99
```

The second argument between the parentheses specifies the initial value for all elements so all 20 elements will be 99L. The first argument that specifies the number of elements in the vector does not need to be a constant expression. It could be the result of an expression executed at runtime or read in from the keyboard. Of course, you can add new elements to the end of this or any other vector using the `push_back()` function.

You can use an index between square brackets to set a value for an existing element or just to use its current value in an expression. For example:

```
values[0] = 3.1415926;           // Pi
values[1] = 5.0;                 // Radius of a circle
values[2] = 2.0*values[0]*values[1]; // Circumference of a circle
```

Index values for a `vector<>` start from 0, just like a standard array. You can always reference existing elements using an index between square brackets but you cannot create new elements this way — you must use the `push_back()` function. The index values are not checked when you index a vector like this. You can access memory outside the extent of the array and store values in such locations using an index between square brackets. The `vector<>` object provides the `at()` function, just like an `array<>` container object, so use the `at()` function to refer to elements whenever there is the potential for the index to be outside the legal range.

A further option for creating a `vector<>` is to use an initializer list to specify initial values:

```
std::vector<unsigned int> primes { 2u, 3u, 5u, 7u, 11u, 13u, 17u, 19u};
```

The `primes` vector container will be created with eight elements with the initial values in the initializer list.

The Capacity and Size of a Vector

The *capacity* of a vector is the number elements that it can store without allocating more memory; these elements may or may not exist. The *size* of a vector is the number of elements it actually contains, which is the number of elements that have values stored. Obviously the size cannot exceed the capacity. When the size equals the capacity, adding an element will cause more memory to be allocated. You can obtain the size and capacity of a vector by calling the `size()` or `capacity()` function for the `vector<>` object. These values are returned as integers of an unsigned integral type that is defined by your implementation. For example:

```
std::vector<unsigned int> primes { 2u, 3u, 5u, 7u, 11u, 13u, 17u, 19u};
std::cout << "The size is " << primes.size() << std::endl;
std::cout << "The capacity is " << primes.capacity() << std::endl;
```

The output statements will present the value 8 for the size and the capacity, as determined by the initializer list. However, if you add an element using the `push_back()` function and output the size and capacity again, the size will be 9 and the capacity will be 16. The increment for increasing the capacity when the size is equal to the capacity is increase by some algorithm based on the existing capacity, typically to double the existing capacity.

You might want to store the size or capacity of a vector in a variable. The type for the size or capacity for a `vector<T>` is `vector<T>::size_type`, which implies that `size_type` is defined within the `vector<T>` class that the compiler generates from the class template. Thus for the `primes` vector the size value will be type `vector<unsigned int>::size_type`. You can avoid worrying about such details most of the time by using the `auto` keyword when you define the variable, for example:

```
auto nElements = primes.size();           // Store the number of elements
```

Remember, you must use `=` with `auto` - not an initializer list, otherwise the type will not be determined correctly. A common reason for storing the size is to iterate over the elements in a vector using an index. You can also use a range-based for loop with a vector. There are other ways of iterating over the elements in a container using objects called iterators. Iterators are defined within the STL and a discussion of iterators is outside the scope of this book.

You are now in a position to create a new version of `Ex5_10.cpp` that only use the memory required for the current input data:

```
// Ex5_14.cpp
// Sorting an array in ascending sequence - using a vector<T> container
#include <iostream>
#include <iomanip>
#include <vector>
using std::vector;

int main()
{
    vector<double> x;                // Stores data to be sorted
    double temp {};                 // Temporary store for a value

    while (true)
    {
        std::cout << "Enter a non-zero value, or 0 to end: ";
        std::cin >> temp;
        if (!temp)
            break;

        x.push_back(temp);
    }

    std::cout << "Starting sort." << std::endl;
    bool swapped {false};           // true when values are not in order
    while (true)
    {
        for (vector<double>::size_type i {} ; i < x.size() - 1 ; ++i)
        {
            if (x.at(i) > x.at(i + 1))
            { // Out of order so swap them
                temp = x.at(i);
                x.at(i) = x.at(i + 1);
                x.at(i + 1) = temp;
                swapped = true;
            }
        }
        if (!swapped)                // If there were no swaps
            break;                   // ...they are in order...
        swapped = false;             // ...otherwise, go round again.
    }

    std::cout << "your data in ascending sequence:\n"
        << std::fixed << std::setprecision(1);
    const size_t perline {10};      // Number output per line
    size_t n{};                     // Number on current line
    for (vector<double>::size_type i {} ; i < x.size() ; ++i)
```

```

{
    std::cout << std::setw(8) << x[i];
    if (++n == perline)           // When perline have been written...
    {
        std::cout << std::endl;    // Start a new line and...
        n = 0;                    // ...reset count on this line
    }
}
std::cout << std::endl;
}

```

The output will be exactly the same as `Ex5_10.cpp`. The only difference in the code is that the data is stored in a container of type `vector<double>`. It is no longer necessary to check whether there is space for each value; unless you fill all the available memory in your PC, it will never happen. Memory is allocated incrementally to accommodate whatever input data is entered. Most of the code is the same - the only significant change is in the type of the control variable in the for loops that iterate over the values during sorting and outputting the sorted data. This is now the type defined in the `vector<double>` class. This is usually the same as type `size_t`, the type returned by the `sizeof` operator, but it's best not to assume this is the case. I used the `at()` function for the vector `x` in the first for loop that reorders the elements. This will throw a `std::out_of_range` exception if the index `i` is invalid, just like the array container; this will terminate the program with a message.

The vector is defined with zero capacity. This is a sorting program so there will always be some input. You can set the capacity by calling the `reserve()` function for the vector object. For example:

```
x.reserve(10);           // Set capacity to 10 elements
```

This doesn't create any elements. It just allocates memory for the number of elements specified as the argument to `reserve()`.

Deleting Elements from a Vector container

You can remove all the elements from a vector by calling the `clear()` function for the vector object. For example:

```
std::vector<int> data(100, 99);    // Contains 100 elements initialized to 99
data.clear();                     // Remove all elements
```

The first statement creates a `vector<int>` object with 100 elements so the size is 100 and the capacity is 100. The second statement removes all the elements so the size will be 0; the capacity will still be 100.

You can remove the last element from a vector object by calling its `pop_back()` function. For example:

```
std::vector<int> data(100, 99);    // Contains 100 elements initialized to 99
data.pop_back();                  // Remove the last element
```

The second statement removes the last element so the size of `data` will be 99 and the capacity left as 100.

This is by no means all there is to using `vector<>` containers. You'll learn a little more about working with `array<>` and `vector<>` containers in the next chapter.

Summary

You will see further applications of containers and loops in the next chapter. Almost any program of consequence involves a loop of some kind. Because they are so fundamental to programming, you need to be sure you have a good grasp of the ideas covered in this chapter. The essential points you have learned in this chapter are:

- An array stores a fixed number of values of a given type.
- You access elements in a one-dimensional array using an index value between square brackets. Index values start at 0 so an index is the offset from the first element in a one-dimensional array.
- An array can have more than one dimension. Each dimension requires a separate index value to reference an element. Accessing elements in an array with two or more dimensions requires an index between square brackets for each array dimension.
- A loop is a mechanism for repeating a block of statements.
- There are four kinds of loop that you can use: the while loop, the do-while loop, the for loop, and the range-based for loop.
- The while loop repeats for as long as a specified condition is true.
- The do-while loop always performs at least one iteration, and continues for as long as a specified condition is true.
- The for loop is typically used to repeat a given number of times and has three control expressions. The first is an initialization expression, executed once at the beginning of the loop. The second is a loop condition, executed before each iteration, which must evaluate to true for the loop to continue. The third is executed at the end of each iteration and is usually used to increment a loop counter.
- The range-based for loop iterates over all elements within a range. An array is a range of elements and a string is a range of characters. The array and vector containers define a range so you can use the range-based for loop to iterate over the elements they contain.
- Any kind of loop may be nested within any other kind of loop to any depth.
- Executing a `continue` statement within a loop skips the remainder of the current iteration and goes straight to the next iteration, as long as the loop control condition allows it.
- Executing a `break` statement within a loop causes an immediate exit from the loop.
- A loop defines a scope so that variables declared within a loop are not accessible outside the loop. In particular, variables declared in the initialization expression of a for loop are not accessible outside the loop.
- The `array<T, N>` container stores a sequence of N elements of type T. An `array<>` container provides an excellent alternative to using the arrays that are built in to the C++ language.
- The `vector<T>` container stores a sequence of elements of type T that increases dynamically in size as required when you add elements. You can use a vector container as an alternative to a standard array when the number of elements cannot be determined in advance.

EXERCISES

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck after that, you can download the solutions from the Apress website (www.apress.com/source-code), but that really should be a last resort.

Exercise 5-1. Write a program that outputs the squares of the odd integers from 1 up to a limit that is entered by the user.

Exercise 5-2. Write a program that uses a `while` loop to accumulate the sum of an arbitrary number of integers that are entered by the user. The program should output the total of all the values and the overall average as a floating-point value.

Exercise 5-3. Create a program that uses a `do-while` loop to count the number of non-whitespace characters entered on a line. The count should end when the first `#` character is found.

Exercise 5-4. Create a `vector<>` container with elements containing the integers from 1 to 100 and output the values 6 on a line with the values aligned in columns. Output the elements from the vector that contain values that are not multiple of 7 or 13. 8 on a line aligned in columns.

Exercise 5-5. Write a program that will read and store an arbitrary sequence of records relating to products. Each record includes three items of data - an integer product number, a quantity, and a unit price, such as for product number 1001 the quantity is 25, and the unit price is \$9.95. The program should output each product on a separate line and include the total cost. The last line should output the total cost for all products. Columns should align so output should be something like this:

Product	Quantity	Unit Price	Cost
1001	25	\$9.95	\$248.75
1003	10	\$15.50	\$155.00
			\$403.75

Exercise 5-6. The famous Fibonacci series is a sequence of integers with the first two values as 1 and the subsequent values as the sum of the two preceding values. So it begins 1, 1, 2, 3, 5, 8, 13, and so on. This is not just a mathematical curiosity. It relates to the way shells grow in a spiral and the number of petals on many flowers is a number from this sequence. Create an `array<>` container with 90 elements. Store the first 90 numbers in the Fibonacci series in the array, then output the 5 to a line, aligned in columns.



Pointers and References

The concepts of pointers and references have similarities, which is why I have put them together in a single chapter. Pointers are important because they provide the foundation for allocating memory dynamically. Pointers can also make your programs more effective and efficient in other ways. Both references and pointers are fundamental to object oriented programming.

In this chapter you'll learn

- What pointers are and how they are defined
- How to obtain the address of a variable
- How to create memory for new variables while your program is executing
- How to release memory that you've allocated dynamically
- The difference between raw pointers and smart pointers
- How to create and use smart pointers
- How you can convert from one type of pointer to another
- What a reference is and how it differs from a pointer
- How you can use a reference in a range-based for loop

What Is a Pointer?

Every variable and function in your program is located somewhere in memory so they each have a unique *address* that identifies where they are stored. These addresses depend on where your program is loaded into memory when you run it, so they may vary from one execution to the next. A *pointer* is a variable that can store an address. The address stored can be the address of a variable or the address of a function. I'll discuss pointers that store the address of a function in Chapter 8. Figure 6-1 shows how a pointer gets its name: it "points to" a location in memory where something (a variable or a function) is stored. However, a pointer needs to record more than just a memory address to be useful. A pointer must store *what* is at the address, as well as *where* it is. As you know, an integer has a different representation from a floating-point value, and the number of bytes occupied by an item of data depends on what it is. To use a data item stored at the address contained in a pointer, you need to know the type of the data.

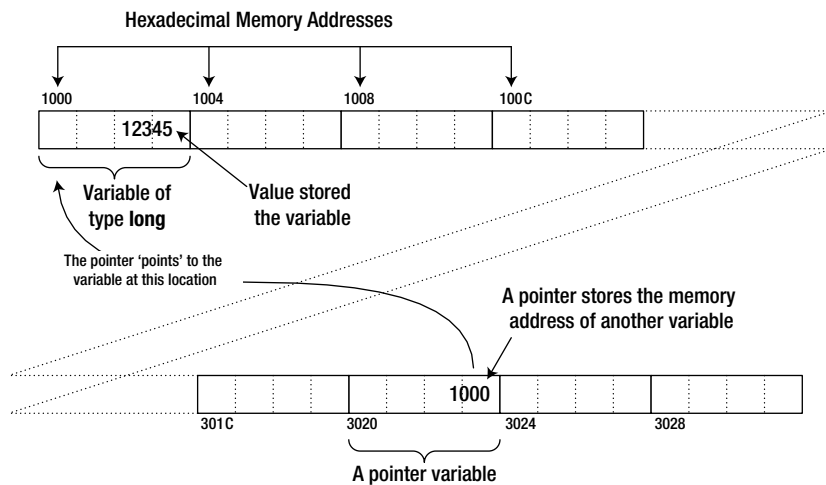


Figure 6-1. What a pointer is

Thus a pointer isn't just a pointer to an address; it's a pointer *to a particular type of data item at that address*. This will become clearer when I get down to specifics, so let's look at how to define a pointer. The definition of a pointer is similar to that of an ordinary variable except that the type name has an asterisk following it to indicate that it's a pointer and not a variable of that type. Here's how you define a pointer called `pnumber` that can store the address of a variable of type `long`:

```
long* pnumber {};           // A pointer to type long
```

The type of `pnumber` is “pointer to long,” which is written as `long*`. This pointer can only store an address of a variable of type `long`. An attempt to store the address of a variable that is other than type `long` will not compile. Because the initializer list is empty, the statement initializes `pnumber` with the pointer equivalent of zero, which is an address that doesn't point to anything. The equivalent of zero for a pointer is written as `nullptr`, and you could specify this explicitly as the initial value:

```
long* pnumber {nullptr};
```

You are not obliged to initialize a pointer when you define it but it's reckless not to. Uninitialized pointers are more dangerous than ordinary variables that aren't initialized so follow this golden rule:

■ **Note** Always initialize a pointer when you define it.

I wrote the pointer type with the asterisk next to the type name, but this isn't the only way to write it. You can position the asterisk adjacent to the variable name, like this:

```
long *pnumber {nullptr};
```

This defines precisely the same variable as before. The compiler accepts either notation. The former is perhaps more common because it expresses the type, “pointer to long,” more clearly.

■ **Note** `nullptr` has a type, `std::nullptr_t`, that is defined in the `cstdint` header. The fact that `nullptr` has a type is important in function overloading, which you'll learn about in Chapter 8.

However, there's potential for confusion if you mix definitions of ordinary variables and pointers in the same statement. Try to guess what this statement does:

```
long* pnumber {}, number {};
```

This defines `pnumber` of type “pointer to long” initialized with `nullptr` and `number` as type `long` initialized with 0L. The notation that juxtaposes the asterisk and the type name makes this less than clear. It's a little clearer if you define the two variables in this form:

```
long *pnumber {}, number {};
```

This is less confusing because the asterisk is now clearly associated with the variable `pnumber`. However, the real solution is to avoid the problem in the first place. It's always better to define pointers and ordinary variables in separate statements:

```
long number {};                // Variable of type long
long* pnumber {};             // Variable of type 'pointer to long'
```

There's no possibility of confusion and there's the added advantage that you can append comments to explain how the variables are used.

It's a common convention to use variable names beginning with `p` for pointers. This makes it easier to see which variables in a program are pointers, which in turn can make the code easier to follow. You can define pointers to any type, including types that you define. Here are definitions for pointer variables of a couple of other types:

```
double* pvalue {};            // Pointer to a double value
char32_t* pch {};             // Pointer to a 32-bit character
```

In practice, most of your use of pointers will be with class types that you define.

The Address-Of Operator

The *address-of* operator, `&`, is a unary operator that obtains the address of a variable. You could define a variable, `number`, and a pointer, `pnumber`, initialized with the address of `number` with these statements:

```
long number {12345L};
long* pnumber {&number};
```

`&number` produces the address of `number` so `pnumber` has this address as its initial value. `pnumber` can store the address of any variable of type `long` so you can write the following assignment:

```
long height {1454L};          // Stores the height of a building
pnumber = &prime;              // Store the address of height in pnumber
```

The result of the statement is that `pnumber` contains the address of `height`. The effect is illustrated in Figure 6-2.

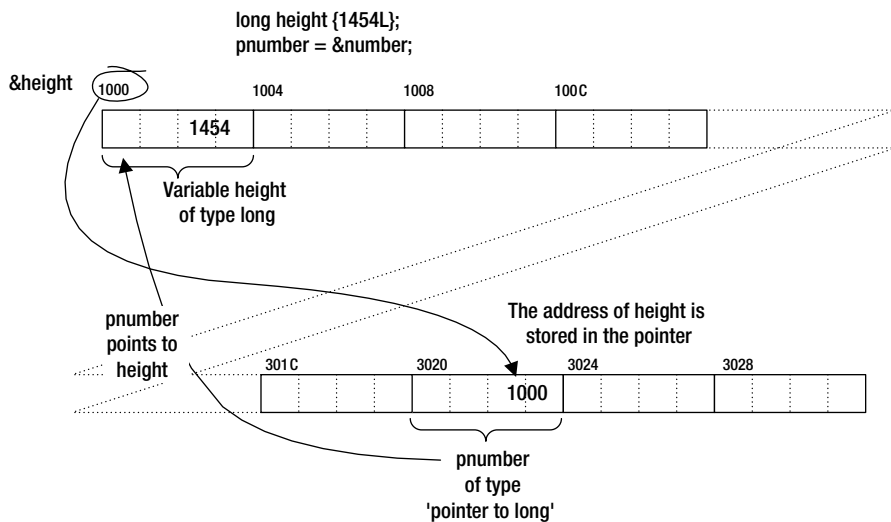


Figure 6-2. Storing an address in a pointer

The `&` operator can be applied to a variable of any type, but you can only store the address in a pointer of the appropriate type. If you want to store the address of a double variable for example, the pointer must have been declared as type `double*`, which is “pointer to double”.

Taking the address of a variable and storing it in a pointer is all very well, but the really interesting thing is how you can use it. Accessing the data in the memory location to which the pointer points is fundamental and you do this using the *indirection operator*.

The Indirection Operator

Applying the *indirection operator*, `*`, to a pointer accesses the contents of the memory location to which it points. The name “indirection operator” stems from the fact that the data is accessed “indirectly.” The operator is sometimes called the *dereference operator*, and the process of accessing the data in the memory location pointed to by a pointer is termed *dereferencing* the pointer. To access the data at the address contained in the pointer, `pnumber`, you use the expression `*pnumber`. Let’s see how dereferencing works in practice with an example. The example is designed to show various ways of using pointers. The way it works will be pointless but not pointerless:

```
// Ex6_01.cpp
// Dereferencing pointers
// Calculates the purchase price for a given quantity of items
#include <iostream>
#include <iomanip>

int main()
{
    int unit_price {295};           // Item unit price in cents
    int count {};                  // Number of items ordered
    int discQ {25};                 // Quantity threshold for discount
    double discount {0.07};         // Discount for quantities over discQ
```

```

int* pcount {&count};           // Pointer to count
std::cout << "Enter the number of items you want: ";
std::cin >> *pcount;
std::cout << "The unit price is " << std::fixed << std::setprecision(2)
    << "$" << unit_price/100.0 << std::endl;

// Calculate gross price
int* punit_price{ &unit_price }; // Pointer to unit_price
int price{ *pcount * *punit_price }; // Gross price via pointers
int* pprice {&price};           // Pointer to gross price

// Calculate net price in US$
double net_price{};
double* pnet_price {nullptr};
pnet_price = &net_price;
if (*pcount > discQ)
{
    std::cout << "You qualify for a discount of "
        << static_cast<int>(discount*100.0) << " percent.\n";
    *pnet_price = price*(1.0 - discount) / 100.0;
}
else
{
    net_price = *pprice / 100.0;
}
std::cout << "The net price for " << *pcount
    << "items is $" << net_price << std::endl;
}

```

Here's some sample output:

```

Enter the number of items you want: 50
The unit price is $2.95
You qualify for a discount of 7 percent.
The net price for 50 items is $137.17

```

I'm sure you realize that this arbitrary interchange between using a pointer and using the original variable is not the right way to code this calculation. However, the example does demonstrate that using a dereferenced pointer is the same as using the variable to which it points. You can use a dereferenced pointer in an expression in the same way as the original variable as the expression for the initial value of `price` shows.

It may seem confusing that you have several different uses for the same symbol, `*`. It's the multiplication operator and the indirection operator, and it's also used in the declaration of a pointer. The compiler is able to distinguish the meaning of `*` by the context. The expression `*pcount * *punit_price` may look slightly confusing, but the compiler has no problem determining that it's the product of two dereferenced pointers. There's no other meaningful interpretation of this expression. If there was, it wouldn't compile. You could add parentheses to make the code easier to read, `(*pcount) * (*punit_price)`.

Why Use Pointers?

A question that usually springs to mind at this point is “Why use pointers at all?” After all, taking the address of a variable you already know about and sticking it in a pointer so that you can dereference it later seems like an overhead you can do without. There are several reasons pointers are important:

1. You can use pointer notation to operate on data stored in an *array*, which may execute faster than if you use array notation.
2. When you define your own functions in Chapter 8, you’ll see that pointers are used extensively to enable a function to access large blocks of data, such as arrays, that are defined outside the function.
3. You’ll see later in this chapter that you can allocate memory for new variables dynamically — that is, during program execution. This allows a program to adjust its use of memory depending on the input. You can create new variables while your program is executing, as and when you need them. When you allocate new memory, the memory is identified by its address so you need a pointer to record it.
4. Pointers are fundamental to enabling *polymorphism* to work. Polymorphism is perhaps the most important capability provided by the object-oriented approach to programming. You’ll learn about polymorphism in Chapter 14.

Pointers to Type char

A variable of type “pointer to char” has the interesting property that it can be initialized with a string literal. For example, you can declare and initialize such a pointer with this statement:

```
char* pproverb {"A miss is as good as a mile."};    // Don't do this!
```

This looks very similar to initializing a char array with a string literal, and indeed it is. The statement creates a null-terminated string literal (actually, an array of elements of type `const char`) from the character string between the quotes and stores the address of the first character in `pproverb`. This is shown in Figure 6-3.

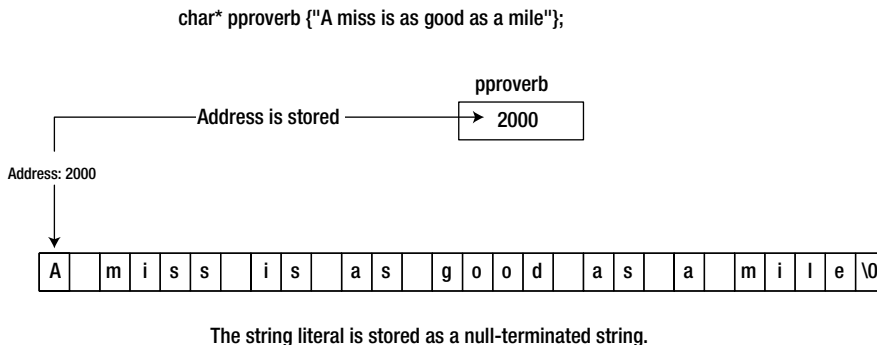


Figure 6-3. Initializing a pointer of type `char*`

Unfortunately, all is not quite as it seems. The type of the string literal is `const`, but the type of the pointer is not. The statement doesn't create a modifiable copy of the string literal; it merely stores the address of the first character. This means that if you attempt to modify the string, there will be trouble. Look at this statement, which tries to change the first character of the string to 'X':

```
*pproverb = 'X';
```

Some compilers won't complain, because they see nothing wrong. The pointer, `pproverb`, wasn't declared as `const`, so the compiler is happy. With other compilers you get a warning that there is a deprecated conversion from type `const char*` to type `char*`. In some environments you'll get an error when you run the program, resulting in a program crash. In other environments the statement does nothing, which presumably is not what was required or expected. The reason for this is that the string literal is still a constant, and you're not allowed to change it.

You might wonder, with good reason, why the compiler allowed you to assign a `const` value to a non-`const` type in the first place, particularly when it causes these problems. The reason is that string literals only became constants with the release of the first C++ standard, and there's a great deal of legacy code that relies on the "incorrect" assignment. Its use is deprecated and the correct approach is to declare the pointer like this:

```
const char* pproverb {"A miss is as good as a mile."}; // Do this instead!
```

This defines `pproverb` to be of type `const char*`. Because it is a `const` pointer type, the pointer can only store an address of something that is `const`. Thus the type is consistent with that of the string literal. There's plenty more to say about using `const` with pointers, so I'll come back to this later in this chapter. For now, let's see how using variables of type `const char*` operates in an example. This is a version of the "lucky stars" example `Ex5_12.cpp` using pointers instead of an array:

```
// Ex6_02.cpp
// Initializing pointers with strings
#include <iostream>

int main()
{
    const char* pstar1 {"Fatty Arbuckle"};
    const char* pstar2 {"Clara Bow"};
    const char* pstar3 {"Lassie"};
    const char* pstar4 {"Slim Pickens"};
    const char* pstar5 {"Boris Karloff"};
    const char* pstar6 {"Mae West"};
    const char* pstar7 {"Oliver Hardy"};
    const char* pstar8 {"Greta Garbo"};
    const char* pstr {"Your lucky star is "};
    size_t choice {};

    std::cout << "Pick a lucky star! Enter a number between 1 and 8: ";
    std::cin >> choice;

    switch (choice)
    {
    case 1:
        std::cout << pstr << pstar1 << std::endl;
        break;
```



```

case 2:
    std::cout << pstr << pstar2 << std::endl;
    break;
case 3:
    std::cout << pstr << pstar3 << std::endl;
    break;
case 4:
    std::cout << pstr << pstar4 << std::endl;
    break;
case 5:
    std::cout << pstr << pstar5 << std::endl;
    break;
case 6:
    std::cout << pstr << pstar6 << std::endl;
    break;
case 7:
    std::cout << pstr << pstar7 << std::endl;
    break;
case 8:
    std::cout << pstr << pstar8 << std::endl;
    break;
default:
    std::cout << "Sorry, you haven't got a lucky star." << std::endl;
}
}

```

Output will be the same as Ex5_12.cpp.

The array of the original example has been replaced by eight pointers, `pstar1` to `pstar8`, each initialized with a string literal. There's an additional pointer, `pstr`, initialized with the phrase to use at the start of a normal output line. Because these pointers contain addresses of string literals, they are specified as `const`.

A switch statement is easier to use than an if statement to select the appropriate output message. Incorrect values entered are taken care of by the default option of the switch.

Outputting a string pointed to by a pointer couldn't be easier. You just use the pointer name. Clearly, the insertion operator `<<` for `cout` treats pointers differently, depending on their type. In Ex6_01.cpp, you had this statement:

```

std::cout << "The net price for " << *pcount
    << " items is $" << net_price << std::endl;

```

If `pcount` wasn't dereferenced here, the address contained in `pcount` would be output. Thus a pointer to a numeric type must be dereferenced to output the value to which it points whereas applying the insertion operator to a pointer to type `char` that is not dereferenced, presumes that the pointer contains the address of a null-terminated string. If you output a dereferenced pointer to type `char`, the single character at the address will be written to `cout`.

Arrays of Pointers

So, what have you gained in Ex6_03.cpp? Well, using pointers has eliminated the waste of memory that occurred with the array in Ex5_12.cpp because each string now occupies just the number of bytes necessary. However, the program is a little long-winded now. If you were thinking "There must be a better way," then you'd be right; you could use an array of pointers:

```

// Ex6_03.cpp
// Using an array of pointers
#include <iostream>

```

```

int main()
{
    const char* pstars[] {
        "Fatty Arbuckle", "Clara Bow",
        "Lassie", "Slim Pickens",
        "Boris Karloff", "Mae West",
        "Oliver Hardy", "Greta Garbo"
    };

    size_t choice {};

    std::cout << "Pick a lucky star! Enter a number between 1 and "
        << sizeof(pstars) / sizeof(pstars[0]) << ": ";
    std::cin >> choice;

    if (choice >= 1 && choice <= sizeof(pstars)/(sizeof pstars[0]))
    {
        std::cout << "Your lucky star is " << pstars[choice - 1] << std::endl;
    }
    else
    {
        std::cout << "Sorry, you haven't got a lucky star." << std::endl;
    }
}

```

Now you're *nearly* getting the best of all possible worlds. You have a one-dimensional array of pointers defined such that the compiler works out the array size from the number of initializing strings. The memory usage that results from this statement is illustrated in Figure 6-4.

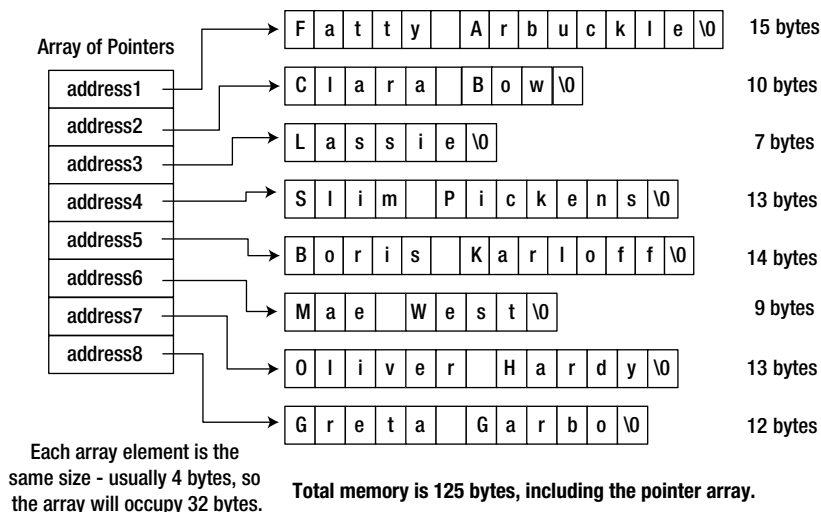


Figure 6-4. An array of pointers

In addition to the memory for each null-terminated string, memory is also occupied by the array elements, which are pointers. The pointer array here requires more memory than a two-dimensional array of char elements containing the same strings. With the char array, each row must have at least the length of the longest string, and eight rows of 15 bytes is 120 bytes. By using a pointer array you'll need 12 more bytes but this is data dependent. If Arnie was an option instead of Fatty, the minimum array dimension would need to accommodate the string "Arnold Schwarzenegger", which requires 21 bytes. In this case, the char array would occupy 168 bytes whereas the array of pointers approach would only need 131 bytes. Generally, the more strings there are, the greater the saving is likely to be. Sometimes with few strings there won't be any saving at all, but generally the pointer array is the more efficient choice.

Saving space isn't the only advantage that you get by using pointers. In many circumstances, you can save time too. For example, think of what happens if you want to swap "Greta Garbo" with "Mae West" in the array. You'd need to do this to sort the strings into alphabetical order for example. With the pointer array, you just reorder the pointers — the strings can stay right where they are. With a char array, a great deal of copying would be necessary. Interchanging the string would require the string "Greta Garbo" to be copied to a temporary location, after which you would copy "Mae West" in its place. Then you would need to copy "Greta Garbo" to its new position. All of this would require significantly more execution time than interchanging two pointers. The code using an array of pointers is very similar to that using a char array. The number of array elements that is used to check that the selection entered is valid is calculated in the same way.

Constant Pointers and Pointers to Constants

In the "lucky stars" program, Ex6_03.cpp, you made sure that the compiler would pick up any attempts to modify the strings pointed to by elements of the pstars array by declaring the array using the const keyword:

```
const char* pstars[] {
    "Fatty Arbuckle", "Clara Bow",
    "Lassie", "Slim Pickens",
    "Boris Karloff", "Mae West",
    "Oliver Hardy", "Greta Garbo"
};
```

Here you are specifying that the objects pointed to by elements of the array are constant. The compiler inhibits any direct attempt to change these, so an assignment statement such as this would be flagged as an error by the compiler, thus preventing a nasty problem at runtime:

```
*pstars[0] = 'X';           // Will not compile...
```

However, you could still legally write the next statement, which would copy the *address* stored in the element on the right of the assignment operator to the element on the left:

```
pstars[5] = pstars[6];      // OK
```

Those lucky individuals due to be awarded Ms. West would now get Mr. Hardy, because both pointers now point to the same name. Of course, this *hasn't* changed the object pointed to by the sixth array element — it has only changed the address stored in it, so the const specification hasn't been contravened.

You really ought to be able to inhibit this kind of change as well, because some people may reckon that good old Ollie may not have quite the same sex appeal as Mae, and of course you can. Look at this statement:

```
const char* const pstars[] {
    "Fatty Arbuckle", "Clara Bow",
    "Lassie", "Slim Pickens",
    "Boris Karloff", "Mae West",
    "Oliver Hardy", "Greta Garbo"
};
```

The extra `const` keyword following the element type specification defines the elements as constant so now the pointers *and* the strings they point to are defined as constant. Nothing about this array can be changed. To summarize, you can distinguish three situations that arise using `const` when applied to pointers and the things to which they point:

- *A pointer to a constant.* You can't modify what's pointed to, but you can set the pointer to point to something else:
 - `const char* pstring {"Some text that cannot be changed"};`
 - Of course, this also applies to pointers to other types, for example:
 - `const int value {20};`
 - `const int* pvalue {&value};`
 - `value` is a constant and can't be changed. `pvalue` is a pointer to a constant, so you can use it to store the address of `value`. You couldn't store the address of `value` in a non-`const` pointer (because that would imply that you can modify a constant through a pointer), but you *could* assign the address of a non-`const` variable to `pvalue`. In the latter case, you would be making it illegal to modify the variable through the pointer. In general, it's always possible to strengthen `const`-ness, but weakening it isn't permitted.
- *A constant pointer.* The address stored in the pointer can't be changed. A constant pointer can only ever point to the address that it's initialized with. However, the *contents* of that address aren't constant and can be changed.
 - Suppose you define an integer variable `data` and a constant pointer `pdata`:
 - `int data {20};`
 - `int* const pdata {&data};`
 - `pdata` is `const`, so it can only ever point to `data`. Any attempt to make it point to another variable will result in an error message from the compiler. The value stored in `data` isn't `const` though, so you can change it. Again, if `data` was declared as `const`, you could not initialize `pdata` with `&data`. `pdata` can only point to a non-`const` variable of type `int`.
- *A constant pointer to a constant.* Here, both the address stored in the pointer and the item pointed to are constant, so neither can be changed.
 - Taking a numerical example, you can define a variable `value` like this:
 - `const int value {20};`
 - `value` is a constant so you can't change it. You can still initialize a pointer with the address of `value`, though:
 - `const int* const pvalue {&value};`
 - `pvalue` is a constant pointer to a constant. You can't change what it points to, and you can't change what is stored at that address.

■ **Note** This isn't confined to pointers of types `char` and `int`. This discussion applies to pointers of any type.

Pointers and Arrays

There is a close connection between pointers and array names. Indeed, there are many situations in which you can use an array name as though it were a pointer. An array name by itself can behave like a pointer when it's used in an output statement. If you try to output an array by just using its name, unless it's a `char` array you'll get is the hexadecimal address of the array. Because an array name can be interpreted as an address, you can use an array name to initialize a pointer:

```
double values[10];
double* pvalue {values};
```

This will store the address of the `values` array in the pointer `pvalue`. Although an array name represents an address, it is not a pointer. You can modify the address stored in a pointer, whereas the address that an array name represents is fixed.

Pointer Arithmetic

You can perform arithmetic operations on a pointer to alter the address it contains. You're limited to addition and subtraction for modifying the address contained in a pointer, but you can also compare pointers to produce a logical result. You can add an integer (or an expression that evaluates to an integer) to a pointer and the result is an address. You can subtract an integer from a pointer and that also results in an address. You can subtract one pointer from another and the result is an integer, not an address. No other arithmetic operations on pointers are legal.

Arithmetic with pointers works in a special way. Suppose you add 1 to a pointer with a statement such as this:

```
++pvalue;
```

This apparently increments the pointer by 1. Exactly *how* you increment the pointer by 1 doesn't matter. You could use an assignment or the `+=` operator to obtain the same effect so the result would be exactly the same with this statement:

```
pvalue += 1;
```

The address stored in the pointer *won't* be incremented by 1 in the normal arithmetic sense. Pointer arithmetic implicitly assumes that the pointer points to an array. Incrementing a pointer by 1 means incrementing it by one *element* of the type to which it points. The compiler knows the number of bytes required to store the data item to which the pointer points. Adding 1 to the pointer increments the address by that number of bytes. In other words, adding 1 to a pointer increments the pointer so that it points to the next element in the array. For example, if `pvalue` is "pointer to double" and type `double` is 8 bytes, then the address in `pvalue` will be incremented by 8. This is illustrated in Figure 6-5.

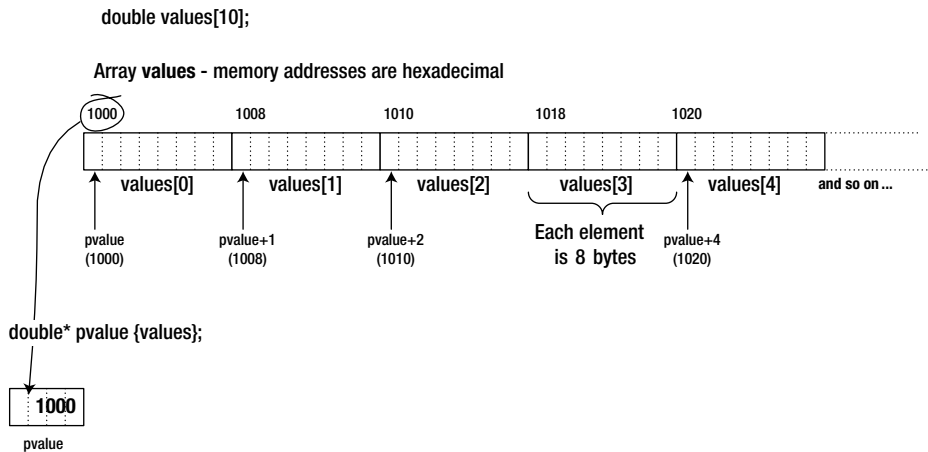


Figure 6-5. Incrementing a pointer

As Figure 6-5 shows, `pvalue` starts out with the address of the first array element. Adding 1 to `pvalue` increments the address it contains by 8, so the result is the address of the next array element. It follows that incrementing the pointer by 2 moves the pointer two elements along. Of course, `pvalue` need not necessarily point to the beginning of the `values` array. You could store the address of the third element of the array in the pointer with this statement:

```
pvalue = &values[2];
```

Now the expression `pvalue + 1` would evaluate to the address of `values[3]`, the fourth element of the `values` array, so you could make the pointer point to this element with this statement:

```
pvalue += 1;
```

In general, the expression `pvalues + n`, in which `n` can be any expression resulting in an integer, will add `n*sizeof(double)` to the address in `pvalue`, because `pvalue` is of type “pointer to double.”

The same logic applies to subtracting an integer from a pointer. If `pvalue` contains the address of `values[2]`, the expression `pvalue - 2` evaluates to the address of the first array element, `values[0]`. In other words, incrementing or decrementing a pointer works in terms of the type of the object pointed to. Incrementing a pointer to long by 1 changes its contents to the next long address, and so increments the address by `sizeof(long)` bytes. Decrementing it by 1 decrements the address by `sizeof(long)`.

Note The address resulting from an arithmetic operation on a pointer can be in a range from the address of the first element of the array to which it points to the address that’s one beyond the last element. Outside these limits, the behavior of the pointer is undefined.

Of course you can dereference a pointer on which you have performed arithmetic. (There wouldn’t be much point to it, otherwise!) For example, consider this statement, assuming `pvalue` is still pointing to `values[2]`:

```
*(pvalue + 1) = *(pvalue + 2);
```

This statement is equivalent to:

```
values[3] = values[4];
```

When you dereference the address resulting from an expression that increments or decrements a pointer, parentheses around the expression are essential because the precedence of the indirection operator is higher than that of the arithmetic operators, + and -. The expression `*pvalue+1` adds 1 to the value stored at the address contained in `pvalue`, so it's equivalent to executing `values[2] + 1`. The result of `*pvalue+1` is a numerical value, not an address and therefore not an lvalue; its use in the previous assignment statement would cause the compiler to generate an error message.

Remember that an expression such as `pvalue+1` doesn't change the address in `pvalue`. It's just an expression that evaluates to a result that is of the same type as `pvalue`. On the other hand the expression `++pvalue` *does* change `pvalue`. Of course, if a pointer contains an invalid address, such an address outside the limits of the array to which it relates, and you store a value using the pointer, you'll attempt to overwrite the memory located at that address. This generally leads to disaster, with your program failing one way or another. It may not be obvious that the cause of the problem is the misuse of a pointer.

The Difference between Pointers

Subtracting one pointer from another is only meaningful when they are of the same type and point to elements in the same array. Suppose you have a one-dimensional array, `numbers`, of type `long` defined as:

```
long numbers[] {10L, 20, 30, 40, 50, 60, 70, 80};
```

Suppose you define and initialize two pointers like this:

```
long *pnum1 {&numbers[6]};           // Points to 7th array element
long *pnum2 {&numbers[1]};           // Points to 2nd array element
```

You can calculate the difference between these two pointers:

```
int difference {pnum1 - pnum2};       // Result is 5
```

`difference` will be set to 5 because the difference between two pointers is measured in terms of elements, not in terms of bytes.

Using Pointer Notation with an Array Name

You can use an array name as though it was a pointer for addressing the array elements. Suppose you define this array:

```
long data[5] {};
```

You can refer to the element `data[3]` using pointer notation as `*(data + 3)`. This notation can be applied generally, so that corresponding to the elements `data[0]`, `data[1]`, `data[2]`, ..., you can write `*data`, `*(data + 1)`, `*(data + 2)`, and so on. The array name by itself refers to the address of the beginning of the array, so an expression such as `data+2` produces the address of the element two elements along from the first.

You can use pointer notation with an array name in the same way as you use an index between square brackets — in expressions or on the left of an assignment. You could set the values of the data array to even integers with this loop:

```
for(size_t i {} ; i < sizeof(data)/sizeof(*data) ; ++i)
{
    *(data + i) = 2 * (i + 1);
}
```

The expression `*(data + i)` refers to successive elements of the array: `*(data + 0)`, which is the same as `*data`, corresponds to `data[0]`, `*(data + 1)` refers to `data[1]`, and so on. The loop will set the values of the array elements to 2, 4, 6, 8, and 10. You could sum the elements of the array like this:

```
long sum {};
for(size_t i {} ; i < sizeof(data)/sizeof(*data) ; ++i)
{
    sum += *(data + i);
}
```

Let's try some of this in a practical context that has a little more meat. This example calculates prime numbers (a prime number is an integer that is divisible only by 1 and itself). Here's the code:

```
// Ex6_04.cpp
// Calculating primes using pointer notation
#include <iostream>
#include <iomanip>

int main()
{
    const size_t max {100};           // Number of primes required
    long primes[max] {2L, 3L, 5L};     // First three primes defined
    size_t count {3};                 // Count of primes found so far
    long trial {5};                    // Candidate prime
    bool isprime {true};               // Indicates when a prime is found

    do
    {
        trial += 2;                     // Next value for checking
        size_t i {};                   // Index to primes array

        // Try dividing the candidate by all the primes we have
        do
        {
            isprime = trial % *(primes + i) > 0; // False for exact division
        } while (++i < count && isprime);

        if (isprime)
        {
            *(primes + count++) = trial; // We got one...
                                           // ...so save it in primes array
        }
    } while (count < max);
```



```
// Output primes 10 to a line
std::cout << "The first " << max << " primes are:" << std::endl;
for (size_t i{} ; i < max ; ++i)
{
    std::cout << std::setw(7) << *(primes + i);
    if ((i+1) % 10 == 0)           // Newline after every 10th prime
        std::cout << std::endl;
}
std::cout << std::endl;
}
```

The output is:

The first 100 primes are:

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107	109	113
127	131	137	139	149	151	157	163	167	173
179	181	191	193	197	199	211	223	227	229
233	239	241	251	257	263	269	271	277	281
283	293	307	311	313	317	331	337	347	349
353	359	367	373	379	383	389	397	401	409
419	421	431	433	439	443	449	457	461	463
467	479	487	491	499	503	509	521	523	541

The constant `max` defines the number of primes to be produced. The `primes` array that stores the results has the first three primes defined to start the process off. The variable `count` records how many primes have been found, so it's initialized to 3.

The `trial` variable holds the next candidate to be tested. It starts out at 5 because it's incremented in the loop that follows. The `bool` variable `isprime` is a flag that indicates when the current value in `trial` is prime.

All the work is done in two loops: the outer `do-while` loop picks the next candidate to be checked and adds the candidate to the `primes` array if it's prime, and the inner loop checks the current candidate to see whether or not it's prime. The outer loop continues until the `primes` array is full.

The algorithm in the loop that checks for a prime is very simple. It's based on the fact that any number that isn't a prime must be divisible by a smaller number that *is* a prime. You find the primes in ascending order, so at any point `primes` contains all the prime numbers lower than the current candidate. If none of the values in `primes` is a divisor of the candidate, then the candidate must be prime.

■ Note You only need to try dividing by primes that are less than or equal to the square root of the number in question, so the example isn't as efficient as it might be.

The inner loop checks whether `trial` is prime:

```
do
{
    isprime = trial % *(primes + i) > 0; // False for exact division
} while (++i < count && isprime);
```

`isprime` is set to the value of the expression `trial % *(primes + i) > 0`. This finds the remainder after dividing `trial` by the number stored at the address `primes + i`. If the remainder is positive, the expression is true. The loop ends if `i` reaches `count` or whenever `isprime` is false. If any of the primes in the `primes` array divides into `trial` exactly, `trial` isn't prime, so this ends the loop. If none of the primes divides into `trial` exactly, `isprime` will always be true and the loop will be ended by `i` reaching `count`.

After the inner loop ends, either because `isprime` was set to false or the set of divisors in the `primes` array has been exhausted, whether or not the value in `trial` was prime is indicated by the value in `isprime`. This is tested in an if statement:

```
if (isprime)
{
    // We got one...
    *(primes + count++) = trial;    // ...so save it in primes array
}
```

If `isprime` contains false, then one of the divisions was exact, so `trial` isn't prime. If `isprime` is true, the assignment statement stores the value from `trial` in `primes[count]` and then increments `count` with the postfix increment operator. When `max primes` have been found, the outer do-while loop ends and the primes are output ten to a line with a field width of ten characters as a result of these statements in a for loop.

Dynamic Memory Allocation

All of the code you've written up to now allocates space for data at compile time. You specify the variables and the arrays that you need in the code, and that's what will be allocated when the program starts, whether you need it or not. Working with a fixed set of variables in a program can be very restrictive, and it's often wasteful.

Dynamic memory allocation is allocating the memory you need to store the data you're working with at runtime, rather than having the amount of memory predefined when the program is compiled. You can change the amount of memory your program has dedicated to it as execution progresses. By definition, dynamically allocated variables can't be defined at compile time so they can't be named in your source program. When you allocate memory dynamically, the space that is made available is identified by its address. The obvious and only place to store this address is in a pointer. With the power of pointers and the dynamic memory management tools in C++, writing this kind of flexibility into your programs is quick and easy. You can add memory to your application when it's needed, then release the memory you have acquired when you are done with it. Thus the amount of memory dedicated to an application can increase and decrease as execution progresses.

Back in Chapter 3, I introduced the three kinds of storage duration that variables can have — automatic, static, and dynamic — and I discussed how variables of the first two varieties are created. Variables for which memory is allocated at runtime always have *dynamic* storage duration.

The Stack and the Heap

You know that an automatic variable is created when its definition is executed. The space for an automatic variable is allocated in a memory area called *the stack*. The stack has a fixed size that is determined by your compiler. There's usually a compiler option that enables you to change the stack size although this is rarely necessary. At the end of the block in which an automatic variable is defined, the memory allocated for the variable on the stack is released, and is thus free to be reused. When you call a function, the arguments you pass to the function will be stored on the stack along with the address of the location to return to when execution of the function ends.

Memory that is not occupied by the operating system or other programs that are currently loaded is called *the heap* or *the free store*. You can request that space be allocated within the free store at runtime for a new variable of any type. You do this using the `new` operator, which returns the address of the space allocated and you store the address in a pointer. The `new` operator is complemented by the `delete` operator, which releases memory that you previously allocated with `new`. Both `new` and `delete` are keywords, so you must not use them for other purposes.

You can allocate space in the free store for variables in one part of a program, and then release the space and return it to the free store in another part of the program when you no longer need it. The memory then becomes available for reuse by other dynamically allocated variables later in the same program or possibly other programs that are executing concurrently. This uses memory very efficiently, and allows programs to handle much larger problems involving considerably more data than might otherwise be possible.

When you allocate space for a variable using `new`, you create the variable in the free store. The variable continues to exist until the memory it occupies is released by the `delete` operator. It continues to exist regardless of whether you still record its address. If you don't use `delete` to release the memory, it will be released automatically when program execution ends.

Using the new and delete Operators

Suppose you need space for a variable of type `double`. You can define a pointer of type `double*` and then request that the memory is allocated at execution time. Here's one way to do this:

```
double* pvalue {};           // Pointer initialized with nullptr
pvalue = new double;         // Request memory for a double variable
```

This is a good moment to recall that *all pointers should be initialized*. Using memory dynamically typically involves having a lot of pointers floating around, and it's important that they not contain spurious values. You should always ensure that a pointer contains `nullptr` if it doesn't contain a legal address.

The `new` operator in the second line of the code returns the address of the memory in the free store allocated to a `double` variable, and this is stored in `pvalue`. You can use this pointer to reference the variable in the free store using the indirection operator as you've seen. For example:

```
*pvalue = 3.14;
```

Of course, under extreme circumstances it may not be possible to allocate the memory. The free store could be completely allocated at the time of the request. The free store can be fragmented by previous usage, which could result in no area of the free store being available that is large enough to accommodate the space you have requested. This isn't likely with the space required to hold a `double` value, but it might just happen when you're dealing with large entities such as arrays or complicated class objects. This is something that you need to consider but for now you'll assume that you always get the memory you request. When it does happen, the `new` operator throws an exception, which by default will end the program. I'll come back to this topic in Chapter 17 when I discuss exceptions.

You can initialize a variable that you create in the free store. Let's reconsider the previous example: the `double` variable allocated by `new`, with its address stored in `pvalue`. You could have initialized its value to `3.14` as it was created by using this statement:

```
pvalue = new double {3.14};           // Allocate a double and initialize it
```

You can also create and initialize the variable in the free store and use its address to initialize the pointer when you create it:

```
double* pvalue {new double {3.14}};    // Pointer initialized with address on the heap
```

This creates the pointer `pvalue`, allocates space for a `double` variable in the free store, initializes the variable in the free store with `3.14` and initializes `pvalue` with the address of the variable.

When you no longer need a dynamically allocated variable, you free the memory that it occupies using the `delete` operator:

```
delete pvalue;                       // Release memory pointed to by pvalue
```

This ensures that the memory can be used subsequently by another variable. If you don't use `delete`, and you store a different address in `pvalue`, it will be impossible to free up the original memory because access to the address will have been lost. The memory will be retained for use by your program until the program ends. Of course, you can't use it because you no longer have the address. Note that the `delete` operator frees the memory but does *not* change the pointer. After the previous statement has executed, `pvalue` still contains the address of the memory that was allocated, but the memory is now free and may be allocated immediately to something else — possibly by another program. The pointer now contains a spurious address so you should always reset a pointer when you release the memory to which it points, like this:

```
delete pvalue;                // Release memory pointed to by pvalue
pvalue = nullptr;            // Reset the pointer
```

Now `pvalue` doesn't point to anything. The pointer cannot be used to access the memory that was released. Using a pointer that contains `nullptr` to store or retrieve data will terminate the program immediately, which is better than the program staggering on in an unpredictable manner with data that is invalid.

Dynamic Allocation of Arrays

Allocating memory for an array at runtime is straightforward. Assuming that you've already declared `pstring`, of type "pointer to `char`," you could allocate an array of type `char` in the free store by writing the following:

```
double* pdata {new double[20]};    // Allocate 100 double values
```

This allocates space for an array of 100 values of type `double` and stores its address in `pdata`.

To remove the array from the free store when you are done with it, you use the `delete` operator, but a little differently:

```
delete [] pdata;                // Release array pointed to by pdata
```

The square brackets are important because they indicate that you're deleting an array. When removing arrays from the free store, you must include the square brackets or the results will be unpredictable. Note that you don't specify any dimensions, simply `[]`.

Of course, you should also reset the pointer, now that it no longer points to memory that you own:

```
pdata = nullptr;                // Reset the pointer
```

Let's see how dynamic memory allocation works in practice. This program calculates the number of primes that you request:

```
// Ex6_05.cpp
// Calculating primes using dynamic memory allocation
#include <iostream>
#include <iomanip>
#include <cmath>                // For square root function

int main()
{
    size_t max {};              // Number of primes required
    size_t count {3};           // Count of primes found
```

```

std::cout << "How many primes would you like? ";
std::cin >> max; // Read number required and...
unsigned long long* primes {new unsigned long long[max]}; // ...allocate memory for them

*primes = 2uLL; // Insert three seed primes...
*(primes + 1) = 3uLL;
*(primes + 2) = 5uLL;
unsigned long long trial {*(primes + 2)}; // Candidate prime
bool isprime {false}; // Indicates when a prime is found

unsigned long long limit {}; // Stores square root of trial
do
{
    trial += 2; // Next value for checking
    limit = static_cast<unsigned long long>(std::sqrt(trial));
    size_t i {}; // Index to primes array
    do
    {
        isprime = trial % *(primes + i) > 0; // False for exact division
    } while (primes[++i] <= limit && isprime);

    if (isprime) // We got one...
        *(primes + count++) = trial; // ...so save it in primes array
} while (count < max);

// Output primes 10 to a line
for (size_t i{} ; i < max ; ++i)
{
    std::cout << std::setw(10) << *(primes + i);
    if ((i + 1) % 10 == 0) // After every 10th prime...
        std::cout << std::endl; // ...start a new line
}
std::cout << std::endl;
delete[] primes; // Free up memory...
primes = nullptr; // ... and reset the pointer
}

```

The output is essentially the same as the previous program so I won't reproduce it here. Overall, the program is similar but not the same as the previous version. After reading the number of primes required from the keyboard and storing it in `max`, you allocate an array of that size in the free store using the `new` operator. The address that's returned by `new` is stored in the pointer, `primes`. This will be the address of the first element of an array of `max` elements of type `unsigned long long`; this type maximizes the upper limit for primes the program can find.

The statements that set up the first three prime values use pointer notation but you could equally well use array notation and write them like this:

```

primes[0] = 2uLL; // Insert three seed primes...
primes[1] = 3uLL;
primes[2] = 5uLL;

```

You can't specify initial values for elements of an array that you allocate dynamically. You have to use explicit assignment statements to set values for the elements. The determination of whether or not a candidate is prime is improved compared to `Ex6_04.cpp`. Dividing the candidate in `trial` by existing primes ceases when primes up to the square root of the candidate have been tried so finding a prime will be faster. The `sqrt()` function from the `cmath` header does this. The `do while` loop condition uses array notation to compare the next prime divisor with `limit` because this demonstrates that you can, and also it's easier to understand than pointer notation here.

When the required number of primes have been output, you remove the array from the free store using the `delete` operator, not forgetting to include the square brackets to indicate that it's an array you're deleting. The next statement resets the pointer. It's not essential here but it's good to get into the habit of always resetting a pointer after freeing the memory to which it points; it could be that you add code to the program at a later date. Of course, if you use a `vector<>` container that you learned about in Chapter 5 to store the primes, you can forget about memory allocation for elements and deleting it when you are done; it's all taken care of by the container.

Member Selection through a Pointer

A pointer can store the address of an object of a class type, such as a `vector<T>` container. Objects usually have member functions that operate on the object - you saw that the `vector<T>` container has an `at()` function for accessing elements and a `push_back()` function for adding an element for example. Suppose you create a `vector<T>` container in the free store with this statement:

```
std::vector<int>* pdata {new std::vector<int>{}};
```

This defines the pointer `pdata` of type `vector<int>*`, which is a "pointer to a vector of `int` elements". The vector is created in the free store. To add an element you call the `push_back()` function for the `vector<int>` object and you have seen how you use a period between the variable representing the vector and the member function name. To access the vector object using the pointer you must use the dereference operator, so the statement to add an element looks like this:

```
(*pdata).push_back(66);           // Add an element containing 66
```

The parentheses around `*pdata` are essential for the statement to compile because the `.` operator is of higher precedence than the `*` operator. This is a clumsy looking expression that occurs very frequently when you are working with objects so C++ provides an operator that combines dereferencing a pointer to an object, then selecting a member of the object. You can write the previous statement like this:

```
pdata->push_back(66);              // Add an element containing 66
```

The `->` operator is formed by a minus sign and a greater than character and is referred to as the *indirect member selection operator*. The arrow is much more expressive of what is happening here. You'll be using this operator extensively later in the book.

Hazards of Dynamic Memory Allocation

There are two kinds of problems that can arise when you allocate memory dynamically. The first is called a *memory leak* and is caused by errors in your code. Unfortunately, memory leaks are quite common. The second is called *memory fragmentation* and is usually due to poor use of dynamic allocation.

Memory Leaks

A memory leak occurs when you allocate memory using `new` and fail to release it. If you lose the address of heap memory you have allocated, by overwriting the address in the pointer you were using to access it for instance, you have a memory leak. This often occurs in a loop, and it's easier to create this kind of problem than you might think. The effect is that your program gradually consumes more and more of the free store, with the program potentially failing at the point when all of the free store has been allocated.

It's relatively easy to see where you've simply forgotten to use `delete` to free memory when use of the memory ceases at a point close to where you allocated it. It becomes more difficult to spot in complex programs, in which memory may be allocated in one part of a program and should be released in a separate part. A good strategy for avoiding memory leaks is to add the `delete` operation at an appropriate place when you use the `new` operator.

When it comes to scope, pointers are just like any other variable. The lifetime of a pointer extends from the point at which you define it in a block to the closing brace of the block. After that it no longer exists so the address it contained is no longer accessible. If a pointer contains the address of a block of memory in the free store goes out of scope, then it's no longer possible to delete the memory.

Fragmentation of the Free Store

Memory fragmentation can arise in programs that allocate and release memory blocks frequently. Each time the `new` operator is used, it allocates a contiguous block of bytes. If you create and destroy many memory blocks of different sizes, it's possible to arrive at a situation in which the allocated memory is interspersed with small blocks of free memory, none of which is large enough to accommodate a new memory allocation request by your program, or by another program that's executing concurrently. The aggregate of the free memory can be quite large, but if all the individual blocks are small (smaller than a current allocation request), the allocation request will fail. The effect of memory fragmentation is illustrated in Figure 6-6.

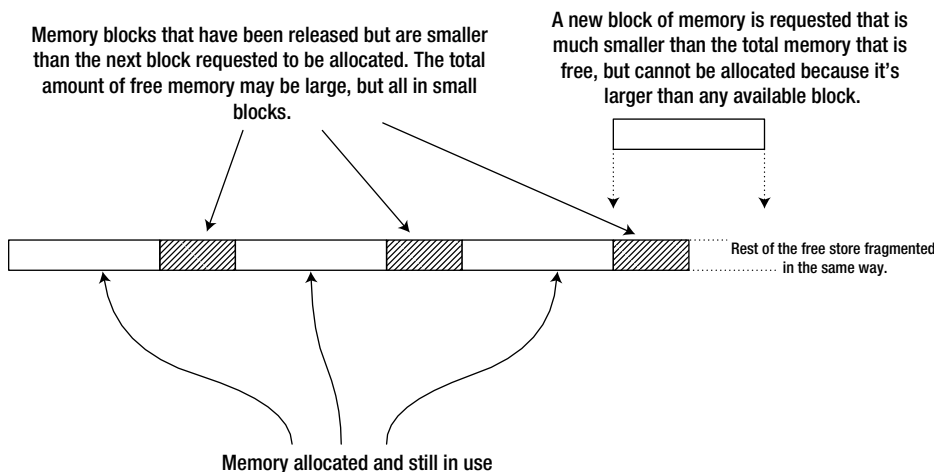


Figure 6-6. Fragmentation of the free store

The problem arises relatively infrequently these days because virtual memory provides a very large memory address space even on quite modest computers. The way to avoid fragmentation of the free store is not to allocate small blocks of memory. Allocate larger blocks and manage the use of the memory yourself.

Raw Pointers and Smart Pointers

All the pointer types I have discussed up to now are part of the C++ language. These are referred to as *raw pointers* because variables of these types contain just an address. A raw pointer can store the address of an automatic variable or a variable allocated in the free store. A *smart pointer* is an object that mimics a raw pointer in that it contains an address and you can use it in the same way in many respects. Smart pointers are only used to store the address of memory allocated in the free store. They are particularly useful for managing class objects that you create dynamically so smart pointers will be of greater relevance from Chapter 11 on. A smart pointer does much more than a raw pointer and for dynamically allocated memory, they are much better than raw pointers. The most notable feature of a smart pointer is that you don't have to worry about using the `delete` operator to free the heap memory. It will be released automatically when it is no longer needed. This means that you avoid the possibility of memory leaks. You can store smart pointers in an `array<T, N>` or a `vector<T>` container, which is very useful when you are working with objects of a class type. Don't assume that smart pointers can do everything a raw pointer can. You cannot increment or decrement a smart pointer, or perform any other arithmetic operations with it.

Smart pointer types are defined by templates that are defined in the `memory` header so you must include this into your source file to use them. There are three types of smart pointers that are defined by the following templates in the `std` namespace:

- A `unique_ptr<T>` object behaves as a pointer to type `T` and is unique, which means there cannot be more than one `unique_ptr<T>` object containing the same address. A `unique_ptr<T>` object owns what it points to exclusively. You cannot assign or copy a `unique_ptr<T>` object. You can move the address stored by one `unique_ptr<T>` object to another using the `std::move()` function that is defined in the `memory` header. After the operation the original object will be invalid.
- A `shared_ptr<T>` object behaves as a pointer to type `T`, and in contrast with `unique_ptr<T>`, there can be any number of `shared_ptr<T>` objects containing the same address. Thus `shared_ptr<T>` objects allow shared ownership of an object in the free store. The number of `shared_ptr<T>` objects that contain a given address is recorded. This is called *reference counting*. The reference count for a `shared_ptr<T>` containing a given heap address is incremented each time a new `shared_ptr<T>` object is created containing that address; the reference count is decremented when a `shared_ptr<T>` object containing the address is destroyed or assigned to point to a different address. When there are no `shared_ptr<T>` objects containing a given address, the reference count will be zero and the heap memory for the object at that address will be released automatically. All `shared_ptr<T>` objects that point to the same address have access to the count of how many there are. You'll understand how this is possible when you learn about classes in Chapter 11.
- A `weak_ptr<T>` is linked to a `shared_ptr<T>` and contains the same address. Creating a `weak_ptr<T>` does not increment the reference count of the linked `shared_ptr<T>` object so it does not prevent the object pointed to from being destroyed. Its memory will be released when the last `shared_ptr<T>` referencing it is destroyed or reassigned to point to a different address, even though associated `weak_ptr<T>` objects may still exist.

The primary reason for having `weak_ptr<T>` objects is that it's possible to inadvertently create *reference cycles* with `shared_ptr<T>` objects. Conceptually, a reference cycle is where a `shared_ptr<T>` object, `pA`, points to another `shared_ptr<T>` object `pB`, and `pB` points to `pA`. With this situation, neither can be destroyed. In practice this occurs in a way that is a lot more complicated. `weak_ptr<T>` objects are designed to avoid the problem of reference cycles. By using `weak_ptr<T>` objects to point to an object that a single `shared_ptr<T>` object points to, you avoid reference cycles. When the single `shared_ptr<T>` object is destroyed, the object pointed to is also destroyed. Any `weak_ptr<T>` objects associated with the `shared_ptr<T>` will then not point to anything.

Using `unique_ptr<T>` Pointers

A `unique_ptr<T>` object stores an address uniquely so the object to which it points is owned exclusively by the `unique_ptr<T>` object. When the `unique_ptr<T>` object is destroyed, the object to which it points is destroyed too. This type of smart pointer is most useful for working with physical facilities that should not be shared, such as communications ports, where these are encapsulated in a class object. A multi-threaded application allows different parts of the program to be executing concurrently. If there is the possibility that one executing thread may try to use a non-sharable resource that is being used by another thread, using a `unique_ptr<T>` object to access the resource will prevent concurrent access. You can create and initialize a `unique_ptr<T>` object like this:

```
std::unique_ptr<double> pdata {new double{999.0}};
```

This creates `pdata` containing the address of a double variable in the free store that is initialized with 999.0. You can also use this syntax to define `pdata`:

```
std::unique_ptr<double> pdata (new double{999.0});
```

You can dereference `pdata` just like an ordinary pointer and you can use the result in the same way:

```
*pdata = 8888.0;
std::cout << *pdata << std::endl;    // Outputs 8888.0
```

The big difference is that you no longer have to worry about deleting the double variable from the free store. You can access the address that a smart pointer contains by calling its `get()` function, for example:

```
std::cout << std::hex << std::showbase << pdata.get() << std::endl; // 0x32a90 on my PC
```

This outputs the value of the address contained in `pdata` as a hexadecimal value. All smart pointers have a `get()` function that will return the address that the pointer contains. The need for accessing the address a smart pointer contains is rare. You should not use raw pointers and smart pointers to point to the same object because this can lead to problems. If you are allocating memory dynamically, stick to one or the other. Unless you have a very good reason not to use them, smart pointers are the better choice because they take care of managing heap memory and eliminate the possibility of memory leaks.

You can create a unique pointer that points to an array:

```
const size_t n {100};                // Array size
std::unique_ptr< double[]> pvalues {new double[n]} ; // Create array of n elements on the heap
```

`pvalues` points to the array of `max` elements of type `double` in the free store. Like a raw pointer, you can use array notation with the smart pointer to access the elements of the array it points to:

```
for (size_t i {} ; i < max ; ++i)
    pvalues[i] = i + 1;
```

This sets the array elements to values from 1 to `max+1`. The compiler will insert an implicit conversion to type `double` for the result of the expression on the right of the assignment. You can output the values of the elements in a similar way:

```
for (size_t i {} ; i < max ; ++i)
{
    std::cout << pvalues[i] << " ";
    if((i + 1) % 10 == 0)
        std::cout << std::endl;
}
```

This just outputs the values 10 on each line. Thus you can use a `unique_ptr<T>` variable that contains the address of an array just like an array name. You cannot do this with the other types of smart pointer. Deleting an array correctly requires the use of `delete[]`, and a `unique_ptr<T>` has this capability. Other types of smart pointer do not by default.

You can reset the pointer contained in any type of smart pointer to `nullptr` by calling its `reset()` function:

```
pvalues.reset(); // Address is nullptr
```

`pvalues` still exists but it no longer points to anything. This is a `unique_ptr<double>` object so because there can be no other unique pointer containing the address of the array, the memory for the array will be released as a result.

Think about the implications of unique pointers being unique. When you store a raw pointer in a `vector<T>` container, the address the pointer contains is copied to the vector element but the pointer variable remains, still containing the original address. You can still use the pointer to access and work with whatever it points to, with the address also stored in the vector. You can't do this with a `unique_ptr<T>` object. It doesn't allow assignment or copying. The only way transfer a `unique_ptr<T>` object somewhere else is to move it using the `std::move()` function. For example:

```
std::unique_ptr<double> pvalue {new double{999.0}};
std::vector<std::unique_ptr<double>>v; // A vector of unique pointers
v.push_back(std::move(pvalue)); // Transfer pvalue to the vector
```

The result of executing this is that `pdata` will cease to point to anything. You can no longer use it. The last element in the vector now has exclusive ownership of the `double` variable in the free store. There can never be two `unique_ptr<T>` objects with the same address. Most of the time, this is not what you want. `shared_ptr<T>` objects don't have this characteristic so you are likely to be using these most of the time. Let's consider those next.

Using `shared_ptr<T>` Pointers

You can define a `shared_ptr<T>` object in a similar way to a `unique_ptr<T>` object:

```
std::shared_ptr<double> pdata {new double{999.0}};
```

You can also dereference it to access what it points to or to change the value stored at the address:

```
*pdata = 8888.0;
std::cout << *pdata << std::endl; // Outputs 8888.0
*pdata = 8889.0;
std::cout << *pdata << std::endl; // Outputs 8889.0
```

Creating a `shared_ptr<T>` object involves a more complicated process than creating a `unique_ptr<T>`, not least because of the need to maintain a reference count. The definition of `pdata` involves one allocation of heap memory for the `double` variable, and another allocation relating to the smart pointer object. Allocating heap memory is expensive on time. You can make the process more efficient by using the `make_shared<T>()` function that is defined in the memory header to create a smart pointer of type `shared_ptr<T>`:

```
auto pdata = std::make_shared<double>(999.0); // Points to a double variable
```

The type of variable to be created in the free store is specified between the angled brackets. This statement allocates memory for the double variable and memory for the smart pointer in a single step, so it's faster. The argument between the parentheses following the function name is used to initialize the double variable it creates. In general, there can be any number of arguments to the `make_shared()` function, the actual number depending of the type of object being created. When you are using `make_shared()` to create objects in the free store, there will often be two or more arguments separated by commas. The `auto` keyword causes the type for `pdata` to be deduced automatically from the object returned by `make_shared<T>()` so it will be `shared_ptr<double>`. Don't forget - you should not use an initializer list when you specify a type as `auto`.

You can initialize a `shared_ptr<T>` with another when you define it:

```
std::shared_ptr<double> pdata2 {pdata};
```

`pdata2` points to the same variable as `pdata`.

You can also assign one `shared_ptr<T>` to another:

```
std::shared_ptr<double> pdata{new double {999.0}};
std::shared_ptr<double> pdata2;           // Pointer contains nullptr
pdata2 = pdata;                          // Copy pointer - both point to the same variable
std::cout << *pd << std::endl;          // Outputs 999.0
```

Of course, copying `pdata` increases the reference count. Both pointers have to be reset or destroyed for the memory occupied by the double variable to be released.

You can't use a `shared_ptr<T>` to store the address of an array created in the free store by default. However, you can store the address of an `array<T>` or `vector<T>` container object that you create in the free store. Here's a working example:

```
// Ex6_06.cpp
// Using smart pointers
#include <iostream>
#include <iomanip>
#include <memory>           // For smart pointers
#include <vector>           // For vector container
#include <locale>           // For toupper()
using std::vector;
using std::shared_ptr;

int main()
{
    vector <shared_ptr<vector<double>>>>records;           // Temperature records by days
    size_t day {1};                                       // Day number
    char answer {};                                       // Response to prompt
    double t {};                                          // A temperature

    while (true)                                         // Collect temperatures by day
    { // Vector to store current day's temperatures created on the heap
        auto pDay = std::make_shared<vector<double>>>();
        records.push_back(pDay);                       // Save pointer in records vector
        std::cout << "Enter the temperatures for day " << day++
                  << " separated by spaces. Enter 1000 to end:\n";
        while (true)
        { // Get temperatures for current day
            std::cin >>t;
            if (t == 1000.0) break;
```

```

    pDay->push_back(t);
}
std::cout << "Enter another day's temperatures (Y or N)? ";
std::cin >> answer;
if (toupper(answer) == 'N') break;
}
double total{};
size_t count{};
day = 1;
std::cout << std::fixed << std::setprecision(2) << std::endl;
for (auto record : records)
{
    std::cout << "\nTemperatures for day " << day++ << ":\n";
    for (auto temp : *record)
    {
        total += temp;
        std::cout << std::setw(6) << temp;
        if (++count % 5 == 0) std::cout << std::endl;
    }
    std::cout << "\nAverage temperature: " << total / count << std::endl;
    total = 0.0;
    count = 0;
}
}

```

Here's how the output looks with arbitrary input values:

```

23 34 29 36 1000
Enter another day's temperatures (Y or N)? y
Enter the temperatures for day 2 separated by spaces. Enter 1000 to end:
34 35 45 43 44 40 37 35 1000
Enter another day's temperatures (Y or N)? y
Enter the temperatures for day 3 separated by spaces. Enter 1000 to end:
44 56 57 45 44 32 28 1000
Enter another day's temperatures (Y or N)? n

Temperatures for day 1:
 23.00 34.00 29.00 36.00
Average temperature: 30.50

Temperatures for day 2:
 34.00 35.00 45.00 43.00 44.00
 40.00 37.00 35.00
Average temperature: 39.13

Temperatures for day 3:
 44.00 56.00 57.00 45.00 44.00
 32.00 28.00
Average temperature: 43.71

```

This program reads an arbitrary number of temperature values recorded during a day, for an arbitrary number of days. The accumulation of temperature records are stored in the `records` vector, which has elements of type `shared_ptr<vector<double>>`. Thus each element is a smart pointer to a vector of type `vector<double>`.

The containers for the temperatures for any number of days are created in the outer `while` loop. The temperature records for a day are stored in a vector container that is created in the free store by this statement:

```
auto pDay = std::make_shared<vector<double>>();
```

The `pDay` pointer type is determined by the pointer type returned by the `make_shared()` function. The function allocates memory for the `vector<double>` object in the free store along with the `shared_ptr<vector<double>>` smart pointer that is initialized with its address and returned. Thus `pDay` is type `shared_ptr<vector<double>>`, which is a smart pointer to a `vector<double>` object. This pointer is added to the `records` container.

The vector pointed to by `pDay` is populated with data that is read in the inner `while` loop. Each value is stored using the `push_back()` function for the current vector pointed to by `pDay`. The function is called using the indirect member selection operator. This loop continues until 1000 is entered, which is an unlikely value for a temperature during the day so there can be no mistaking it for a real value. When all the data for the current day has been entered, the inner `while` loop ends and there's a prompt asking whether another day's temperatures are to be entered. If the answer is affirmative, the outer loop continues and creates another vector in the free store. When the outer loop ends, the `records` vector will contain smart pointers to vectors containing each day's temperatures.

The next loop is a range-based `for` loop that iterates over the elements in the `records` vector. The inner range-based `for` loop iterates over the temperatures values in the vector that the current `records` element points to. This inner loop outputs the data for the day and accumulates to total of the temperatures values. This allows the average temperature for the current day to be calculated when the inner loop ends. In spite of having a fairly complicated data organization with a vector of smart pointers to vectors in the free store, accessing and processing the data is very easy using range-based `for` loops.

The example illustrates how using containers and smart pointers can be a powerful and flexible combination. This program deals with any number of sets of input, with each set containing any number of values. Free store memory is managed by the smart pointers so there no need to worry about using the `delete` operator, or the possibility of memory leaks. The `records` vector could also have been created in the free store too, but I'll leave that as an exercise for you to try.

■ **Note** It is possible to create a `shared_ptr<T>` object that points to an array. This involves supplying a definition for a deleter function that the smart pointer is to use to release the heap memory for the array. The details of how you do this are outside the scope of this book.

Comparing `shared_ptr<T>` Objects

You can compare the address contained in one `shared_ptr<T>` object with another, or with `nullptr` using any of the comparison operators. The most useful are comparisons for equality or inequality, which tell you whether two pointers point to the same object. Given two `shared_ptr<T>` objects, `pA` and `pB`, that point to the same type, `T`, you can compare them like this:

```
if((pA == pB) && (pA != nullptr))
    std::cout << " Both pointers point to the same object.\n";
```

The pointers could both be `nullptr` and be equal so a simple comparison is not sufficient to establish that they both point to the same object. The first expression in the `if` condition compares the two pointers and evaluates to `true` if they contain the same address. The result is ANDed with the second expression that results in `true` if `pA` is not `nullptr`. When both are true the output statement executes confirming that both point to the same object. A `shared_ptr<T>` object can be implicitly converted to type `bool` so you could write the statement as:

```
if(pA && (pA == pB))
    std::cout << " Both pointers point to the same object.\n";
```

weak_ptr<T> Pointers

A `weak_ptr<T>` pointer is always created from a `shared_ptr<T>` pointer. `weak_ptr<T>` pointers are intended for use as class members that store an address of another member of the same class, when objects of the class are created in the free store. Using a `shared_ptr<T>` member to point to another object of the same type in such circumstances has the potential for creating a reference cycle, which would prevent objects of the class type from being deleted from the free store automatically. This is not a common situation but it is possible, as Figure 6-7 shows.

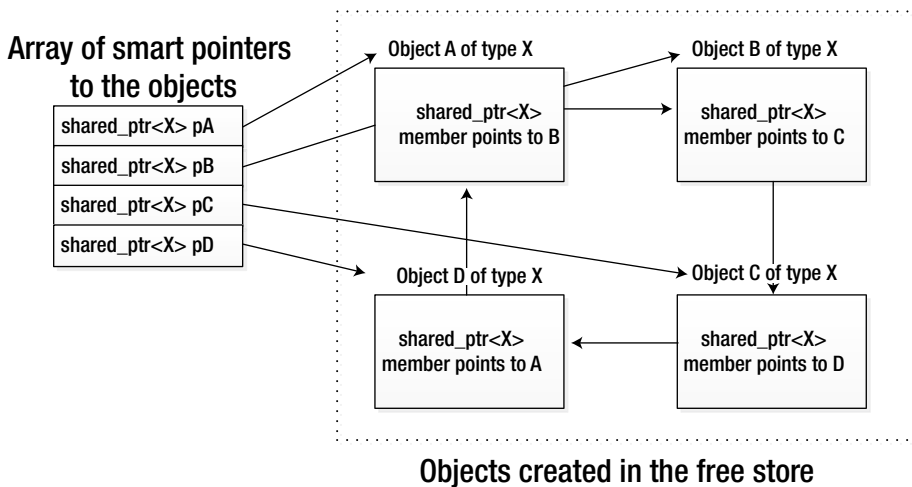


Figure 6-7. How a reference cycle prevents objects from being deleted

Deleting all the smart pointers in the array or resetting them to `nullptr` does not delete the memory for the objects to which they point. There is still a `shared_ptr<X>` object containing the address of every object. There are no external pointers remaining that can access these objects so they cannot be deleted. The problem can be avoided if the objects used `weak_ptr<X>` members to refer to other objects. These would not prevent the objects from being destroyed when the external pointers in the array are destroyed or reset.

You can create a `weak_ptr<T>` object like this:

```
auto pData = std::make_shared<X>(); // Create a pointer to an object of type X
std::weak_ptr<X> pwData {pData};    // Create a weak pointer from shared pointer
std::weak_ptr<X> pwData2 {pData};    // Create a weak pointer from another
```

Thus you can create a `weak_ptr<T>` from a `shared_ptr<T>` or from an existing `weak_ptr<T>`. You can't do very much with a weak pointer - you can't dereference it to access the object it points to for example. You can do two things with a `weak_ptr<T>` object:

- You can test whether the object it points to still exists, which means there's a `shared_ptr<T>` still around that points to it.
- You can create a `shared_ptr<T>` object from a `weak_ptr<T>` object.

Here's how you can test for the existence of the object that a weak pointer references:

```
if(pwData.expired())
    std::cout << "Object no longer exists.\n";
```

The `expired()` function for the `pwData` object returns `true` if the object no longer exists. You can create a shared pointer from a weak pointer like this:

```
std::shared_ptr<X> pNew {wpData.lock()};
```

The `lock()` function locks the object if it exists by returning a new `shared_ptr<X>` object that initializes `pNew`. If the object does not exist, the `lock()` function will return a `shared_ptr<X>` object containing `nullptr`. You can test the result in an `if` statement:

```
if(pNew)
    std::cout << "Shared pointer to object created.\n";
else
    std::cout << "Object no longer exists.\n";
```

Working with `weak_ptr<T>` pointers is fairly advanced stuff so I won't be delving into these any further.

Understanding References

A reference appears similar to a pointer in some respects, which is why I'm introducing it here, but it is completely different. You'll only get a real appreciation of the value of references when I introduce you to defining functions in Chapter 8. References become more important in the context of object-oriented programming. Don't be misled by their simplicity and what might seem to be a trivial concept here. You'll see later in the book that references provide some extraordinarily powerful facilities. There are some things that would be impossible without references.

A reference is a name that you can use as an alias for something. Obviously it must be like a pointer insofar as it refers to something else in memory but it is quite different. There are two kinds of references: *lvalue references* and *rvalue references*. An *lvalue reference* is an alias for another variable; it is called an *lvalue* reference because it refers to a persistent storage location in which you can store data so it can appear on the left of an assignment operator. Because an *lvalue reference* is an alias, the variable for which it is an alias must exist when the reference is defined. Unlike a pointer, a reference cannot be modified to be an alias for something else. An *rvalue reference* can be an alias for a variable, just like an *lvalue reference*, but it differs from an *lvalue reference* in that it can also reference an *rvalue*, which is a value that is transient. The result of evaluating an expression is an *rvalue*, so an *rvalue reference* can be an alias for such a result. You'll see in Chapter 8 how having the two types of reference available enables you to determine whether a value passed to a function is an *rvalue* or an *lvalue*.

Defining lvalue References

Suppose you defined this variable:

```
double data {3.5};
```

You can define an lvalue reference as an alias for data like this variable:

```
double& rdata {data};           // Defines a reference to the variable data
```

The ampersand following the type name indicates that the variable, `rdata`, defined, is a reference to a variable of type `double`. The variable that it represents is specified in the initializer list. Thus `rdata` is of type 'reference to `double`'. You can use the reference as an alternative to the original variable name. For example:

```
rdata += 2.5;
```

This increments `data` by 2.5. None of the dereferencing that you need with a pointer is necessary - you just use the name of the reference as though it is a variable. Note that you can initialize a reference with a literal as long as you specify the reference type as `const`. For example:

```
const int& rTen {10};           // OK
```

You can access the literal 10 using the `rTen` reference. Because `rTen` is `const`, it cannot be used to change the value it references. If you try to define the reference as a non-`const` type, the statement won't compile.

Let's ram home the difference between a reference and a pointer by contrasting the lvalue reference `rdata` in the previous code with the pointer, `pdata`, defined in this statement:

```
long* pdata {&data};           // A pointer containing the address of data
```

This defines a pointer, `pdata`, and initializes it with the address of `data`. This allows you to increment `data` like this:

```
*pdata += 2.5;                 // Increment data through a pointer
```

You must dereference the pointer to access the variable to which it points. With a reference, there is no need for de-referencing; it just doesn't apply. In some ways, a reference is like a pointer that has already been dereferenced, although it can't be changed to reference something else. An lvalue reference is the complete equivalent of the variable for which it is a reference.

Using a Reference Variable in a Range-Based for Loop

You know that you can use a range-based for loop to iterate over all the elements in an array:

```
double temperatures[] {45.5, 50.0, 48.2, 57.0, 63.8};
for(auto t : temperatures)
{
    sum += t;
    ++count;
}
```


The variable `t` is initialized to the value of the current array element on each iteration, starting with the first. `t` does not access the element itself so you cannot use `t` to modify the value of an element. However, you can change the array elements if you use an lvalue reference:

```
const double F_to_C {5.0/9.0};           // Convert Fahrenheit to Centigrade
for(auto& t : temperatures)             // lvalue reference loop variable
    t = (t - 32.0)*FtoC;
```

The loop variable, `t`, is now of type `double&` so it is an alias for each array element. The loop variable is redefined on each iteration and initialized with the current element, so the reference is not being changed. This loop changes the values in the `temperatures` array from Fahrenheit to Centigrade. Using a reference in a range-based for loop is very efficient when you are working with collections of objects. Copying objects can be expensive on time, so avoiding copying by using a reference type makes your code more efficient.

When you use an lvalue reference type for the variable in a range-based for loop and you don't need to modify the values, you can use `const` reference type for the loop variable:

```
for (const auto& t : temperatures)
    std::cout << std::setw(6) << t;
std::cout << std::endl;
```

You still get the benefits of using an lvalue reference type to make the loop as efficient as possible, and at the same time you prevent the array elements from being changed by this loop.

Defining rvalue References

rvalue references don't have any uses with what you have learned so far. I'm explaining them here because the concept is closely related to lvalue references. You'll understand how you use them and what they can do for you starting in Chapter 8. The result of every expression is either an rvalue or an lvalue. A variable is an lvalue when it represents a persistent memory location. The result of expression that is stored in a temporary memory location is an rvalue. An rvalue reference is an alias for a temporary memory location that contains the result of evaluating an expression.

You specify an rvalue reference type using *two* ampersands following the type name. Here's an example:

```
int count {5};
int&& rtemp {count + 3};           // rvalue reference
std::cout << rtemp << std::endl;   // Output value of expression
int& rcount {count};              // lvalue reference
std::cout << rcount << std::endl;   // Output value of count
```

This code will compile and execute, but it is definitely *NOT* the way to use an rvalue reference and you should not code like this. This is just to illustrate what an rvalue reference is. The rvalue reference is initialized to be an alias for the result of the expression `count+3`, which is a temporary value — an rvalue. The output from the next statement will be 8. You cannot do this with an lvalue reference. Is this useful? In this case, no, indeed it is not recommended at all; but in a different context, it is very very useful. You see that it can make your code much more efficient.

Summary

You've explored some very important concepts in this chapter. You will undoubtedly make extensive use of pointers and particularly smart pointers in real-world C++ programs, and you'll see a lot more of them throughout the rest of the book.

The vital points this chapter covered:

- A pointer is a variable that contains an address. A basic pointer is referred to as a raw pointer.
- You obtain the address of a variable using the address-of operator, `&`.
- A smart pointer is an object that can be used like a raw pointer. A smart pointer is only used to store free store memory addresses.
- To refer to the value pointed to by a pointer, you use the indirection operator, `*`. This is also called the dereference operator.
- You access a member of an object through a raw pointer or a smart pointer using the member selection operator, which is a period.
- You access a member of an object through a pointer or smart pointer using the indirect member selection operator, `->`.
- You can add integer values to or subtract integer values from the address stored in a raw pointer. The effect is as though the pointer refers to an array, and the pointer is altered by the number of array elements specified by the integer value. You cannot perform arithmetic with a smart pointer.
- The `new` operator allocates a block of memory in the free store and returns the address of the memory allocated.
- You use the `delete` operator to release a block of memory that you've allocated previously using the `new` operator. You don't need to use the `delete` operator when the address of free store memory is stored in a smart pointer.
- There are three varieties of smart pointers. There can only ever be one `unique_ptr<T>` pointer in existence that points to a given object of type `T`. There can be multiple `shared_ptr<T>` objects containing the address of a given object of type `T` and the object will be destroyed when there are no `shared_ptr<T>` objects containing its address. `weak_ptr<T>` pointers are used with `shared_ptr<T>` pointers to avoid reference cycles.
- An lvalue reference is an alias for a variable that represents a permanent storage location. An rvalue reference is an alias for a temporary memory location that contains the result of evaluating an expression.
- You can use a reference type for the loop variable in a range-based for loop to allow the array element values to be modified.

EXERCISES

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck after that, you can download the solutions from the Apress website (www.apress.com/source-code), but that really should be a last resort.

Exercise 6-1. Write a program that declares and initializes an array with the first 50 even numbers. Output the numbers from the array ten to a line using pointer notation, and then output them in reverse order also using pointer notation.

Exercise 6-2. Write a program that reads an array size from the keyboard and dynamically allocates an array of that size to hold floating-point values. Using pointer notation, initialize all the elements of the array so that the value of the element at index position n is $1.0/(n+1)^2$. Calculate the sum of the elements using pointer notation, multiply the sum by 6, and output the square root of that result.

Exercise 6-3. Repeat the calculation in Exercise 6-2 but using a `vector<>` container allocated in the free store. Test the program with more than 100,000 elements. Do you notice anything interesting about the result?

Exercise 6-4. You know that a two-dimensional array is an “array of arrays.” You also know that you can create an array dynamically using a pointer. If the elements of the array that you create dynamically are also pointers, then each element in the array could store the address of an array. Using this concept, create an array of three pointers to arrays, in which each array can store six values of type `int`. Set the first array of integers to values 1 to 6, the next array to the squares ($N \times N$) of the values stored first array, and the next the cubes ($N \times N \times N$) of the values stored in the first array of integers. Output the contents of the three arrays, and then delete the memory you've allocated.

Exercise 6-5. Write a program that will read an arbitrary number of age values in years for students in each of an arbitrary number of classes. The number of ages is not known in advance but there can be up to 50 students in a class. Store the student age values for each class in an `vector<T>` container that you create in the free store. A `shared_ptr<T>` for each `vector<T>` should be stored in a `vector<T>` that is also created in the free store. After input is complete, list the ages of students in each class, 5 to a line, followed by the average age for the class.



Working with Strings

This chapter is about handling textual data much more effectively and safely than the mechanism provided by a C-style string stored in an array of `char` elements. In this chapter, you'll learn:

- How to create variables of type `string`
- What operations are available with objects of type `string`, and how you use them
- How you can search a string for a specific character or a substring
- How you can modify an existing string
- How you can work with strings containing Unicode characters
- What a raw string literal is

A Better Class of String

You've seen how you can use an array of elements type `char` to store a null-terminated (C-style) string. The `cstring` header provides a range of functions for working with C-style strings including capability for joining strings, searching a string, and comparing strings. All these operations depend on the null character being present to mark the end of a string. If it is missing or gets overwritten, many of these functions will march happily through memory beyond the end of a string until a null character is found at some point, or some catastrophe stops the process. It often results in memory being arbitrarily overwritten. Using C-style strings is inherently unsafe and represents a security risk. Fortunately there's a better alternative.

The `string` header defines the `string` type, which is much easier to use than a null-terminated string. The `string` type is defined by a class (or to be more precise, a class template) so it isn't one of the fundamental types. Type `string` is a *compound type*, which is a type that's a composite of several data items that are ultimately defined in terms of fundamental types of data. A `string` object contains the characters that make up the string it represents, and other data, such as number of characters in the string. Because the `string` type is defined in the `string` header, you must include this header when you're using `string` objects. The `string` type name is defined within the `std` namespace, so you need a `using` declaration to use the type name in its unqualified form. The `string` type name is very often used in its unqualified form because when you are using it, it occurs very frequently in the code. I'll assume a `using` declaration and write it as `string` rather than `std::string` in code as well as in the text. I'll start by explaining how you create `string` objects.

Defining string Objects

An object of type `string` contains a sequence of characters of type `char`, which can be empty. This statement defines a variable of type `string` that contains an empty string:

```
string empty;                // An empty string
```

This statement defines a `string` object that you refer to using the name `empty`. In this case `empty` contains a string that has no characters and so it has zero length.

You can initialize a `string` object with a string literal when you define it:

```
string proverb {"Many a mickle makes a muckle."};
```

`proverb` is a `string` object that contains the string literal shown in the initializer list. The string that's encapsulated by a `string` object doesn't have a string termination character. A `string` object keeps track of the length of the string that it represents, so no termination character is necessary.

■ **Warning** Don't use an initializer list containing `0` or `nullptr` to initialize a `string` object. If you do, it won't contain an empty string or any other kind of string, and it will certainly cause trouble. If you are lucky, you'll get a runtime error - if you're not, the program will crash or otherwise behave incorrectly.

You can obtain the length of the string for a `string` object using its `length()` function, which takes no arguments:

```
std::cout << proverb.length();    // Outputs 29
```

This statement calls the `length()` function for the `proverb` object and outputs the value it returns to `cout`. The record of the string length is maintained by the object itself. When you append one or more characters, the length is increased automatically by the appropriate amount and decreased if you remove characters.

There are some other possibilities for initializing a `string` object. You can use an initial sequence from a string literal for instance:

```
string part_literal {"Least said soonest mended.", 5}; // "Least"
```

The second initializer in the list specifies the length of the sequence from the first initializer to be used to initialize the `part_literal` object.

You can't initialize a `string` object with a single character between single quotes — you must use a string literal between double quotes, even when it's just one character. However, you *can* initialize a string with any number of instances of given character. You can define and initialize a sleepy time `string` object like this:

```
string sleeping {6, 'z'};
```

The `string` object, `sleeping`, will contain "zzzzzz". The string length will be 6. If you want to define a `string` object that's more suited to a light sleeper, you could write this:

```
string light_sleeper {1, 'z'};
```

This initialize `light_sleeper` with the string literal `"z"`. A further option is to use an existing `string` object to provide the initial value. Given that you've defined `proverb` previously, you can define another object based on that:

```
string sentence {proverb};
```

The `sentence` object will be initialized with the string literal that `proverb` contains, so it too will contain "Many a mickle makes a muckle." and have a length of 29.

You can reference characters within a `string` object using an index value starting from 0, just like an array. You can use a pair of index values to identify part of an existing string and use that to initialize a new `string` object, for example:

```
string phrase {proverb, 0, 13}; // Initialize with 13 characters starting at index 0
```

Figure 7-1 illustrates this process.

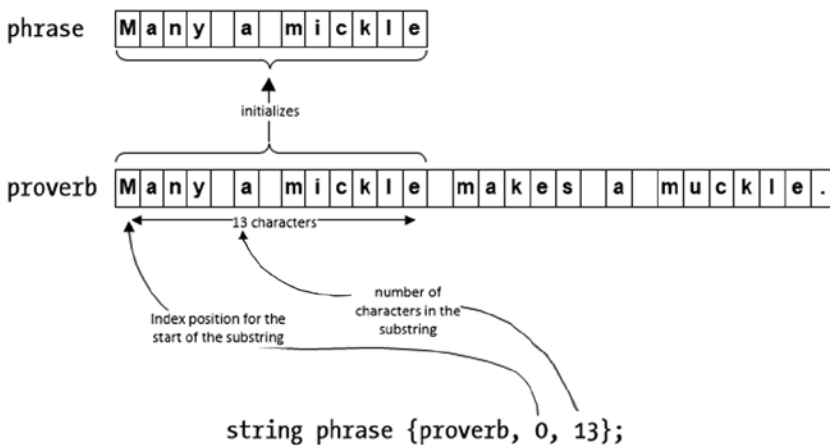


Figure 7-1. Creating a new string from part of an existing string

The first initializer in the initializer list is the source of the initializing string. The second is the index of the character in `proverb` that begins the initializing substring and the third initializer in the list is the number of characters in the substring. Thus `phrase` will contain "Many a mickle". To demonstrate that that is the case you can insert the `phrase` object in the output stream, `cout`:

```
std::cout << phrase << std::endl;
```

Thus you can output string objects just like C-style strings. Extraction from `cin` is also supported for `string` objects:

```
string name;
std::cout << "enter your name: ";
std::cin >> name; // Pressing Enter ends input
```

This reads characters up to the first whitespace character, which ends the input process. Whatever was read is stored in the `string` object, `name`. You cannot enter text with embedded spaces with this process.

Just to summarize: I have described six options for defining and initializing a string object; the comments below identify the initializing string in each case.

1. No initializer list (or an empty list):

```
string empty;                                // The string ""
```

2. An initializer list containing a string literal:

```
string proverb {"Many a mickle makes a muckle."}; // The literal
```

3. An initializer list containing an existing string object:

```
string sentence {proverb};                    // Duplicates proverb
```

4. An initializer list containing two initializers that are a string literal followed by the length of the sequence in the literal to be used to initialize the string object:

```
string part_literal {"Least said soonest mended.", 5}; // "Least"
```

5. An initializer list containing two initializers that are a repeat count followed by the character literal that is to be repeated in the string that initializes the string object:

```
string sleeping {6, 'z'};                     // "zzzzzz"
```

6. An initializer list containing three initializers that are an existing string object, an index specifying the start of the substring in the first initializer, and the length of the substring:

```
string phrase {proverb, 0, 13};                // "many a mickle"
```

Operations with String Objects

A wide range of operations with string objects are supported. Perhaps the simplest is assignment. You can assign a string literal or another string object to a string object, for example:

```
string adjective {"hornswoggling"};           // Defines adjective
string word {"rubbish"};                       // Defines word
word = adjective;                             // Modifies word
adjective = "twotiming";                       // Modifies adjective
```

The third statement assigns the value of `adjective`, which is "hornswoggling", to `word`, so "rubbish" is replaced. The last statement assigns the literal, "twotiming" to `adjective`, so the original value "hornswoggling" is replaced. Thus, after executing these statements, `word` will contain "hornswoggling" and `adjective` will contain "twotiming".

Concatenating Strings

You can join strings using the addition operator; the technical term for this is *concatenation*. You can concatenate the objects defined above:

```
string description {adjective + " " + word + " whippersnapper"};
```

After executing this statement, the `description` object will contain the string "twotiming hornsoggling whippersnapper". You can see that you can concatenate string literals with `string` objects using the `+` operator. This is because the `+` operator has been redefined to have a special meaning with `string` objects. When one operand is a `string` object and the other operand is either another `string` object or a string literal, the result of the `+` operation is a new `string` object containing the two strings joined together.

Note that you *can't* concatenate two string literals using the `+` operator. One of the operands must always be an object of type `string`. The following statement, for example, won't compile:

```
string description {" whippersnapper" + " " + word};    // Wrong!!
```

The problem is that the compiler will try to evaluate the initializer value as `(" whippersnapper"+" ") + word`, and the `+` operator doesn't work with both operands as two string literals. However, you have three ways around this. You can write the first two string literals as a single string literal `{" whippersnapper " + word}` you can omit the `+` between the two literals `{" whippersnapper" " " + word}`, or you can use parentheses `{"whippersnapper" + (" " + word)}`. Two or more string literals in sequence will be concatenated into a single literal by the compiler. The expression between parentheses that joins `" "` with `word` is evaluated first to produce a `string` object, and that can be joined to the first literal using the `+` operator.

That's enough theory for the moment. It's time for a bit of practice. This program reads your first and second names from the keyboard:

```
// Ex7_01.cpp
// Concatenating strings
#include <iostream>
#include <string>
using std::string;

int main()
{
    string first;                // Stores the first name
    string second;              // Stores the second name

    std::cout << "Enter your first name: ";
    std::cin >> first;           // Read first name

    std::cout << "Enter your second name: ";
    std::cin >> second;         // Read second name

    string sentence {"Your full name is "}; // Create basic sentence
    sentence += first + " " + second + "."; // Augment with names

    std::cout << sentence << std::endl;    // Output the sentence

    std::cout << "The string contains "    // Output its length
                << sentence.length() << " characters." << std::endl;
}
```

Here's some sample output:

```
Enter your first name: Phil
Enter your second name: McCavity
Your full name is Phil McCavity.
The string contains 33 characters.
```

After defining two empty string objects, `first` and `second`, the program prompts for input of a first name then a second name. The input operations will read anything up to the first whitespace character. You'll learn how you can read a string that includes whitespace later in this chapter.

After getting the names, you create another string object that is initialized with a string literal. The sentence object is concatenated with the string object that results from the right operand of the `+=` assignment operator:

```
sentence += first + " " + second + ".";           // Augment with names
```

The right operand concatenates `first` with the literal `" "`, then `second` is appended to that result, and finally the literal `"."` is appended to that to produce the final result that is concatenated with the right operand of the `+=` operator. This statement demonstrates that the `+=` operator also works with objects of type `string` in a similar way to the basic types. The statement is equivalent to this statement:

```
sentence = sentence + (first + " " + second + "."); // Augment with names
```

When you use the `+=` operator to append a value to a string object, the right side can be an expression resulting in a null-terminated string, a single character of type `char`, or an expression that results in an object of type `string`. Finally the program uses the stream insertion operator to output the contents of `sentence` and the length of the string it contains.

Accessing Characters in a String

You refer to a particular character in a string by using an index value between square brackets, just as you do with an array. The first character in a string object has the index value 0. You could refer to the third character in `sentence`, for example, as `sentence[2]`. You can use such an expression on the left of the assignment operator, so you can *replace* individual characters as well as access them. The following loop changes all the characters in `sentence` to uppercase:

```
for(size_t i {} ; i < sentence.length() ; ++i)
    sentence[i] = std::toupper(sentence[i]);
```

This loop applies the `toupper()` function to each character in the string in turn and stores the result in the same position in the string. The index value for the first character is 0, and the index value for the last character is one less than the length of the string, so the loop continues as long as `i < sentence.length()` is true.

A string object is a range, so you could also do this with the range-based for loop:

```
for (auto& ch : sentence)
    ch = std::toupper(ch);
```

Specifying `ch` as a reference type allows the character in the string to be modified within the loop. This loop and the previous loop require the `locale` header to be included to compile.

You can exercise this array-style access method in a version of `Ex5_11.cpp` that determined the number of vowels and consonants in a string. The new version will use a string object. It will also demonstrate that you can use the `getline()` function to read a line of text that includes spaces:

```
// Ex7_02.cpp
// Accessing characters in a string
#include <iostream>
#include <string>
#include <locale>
using std::string;
```

```

int main()
{
    string text;                                // Stores the input
    std::cout << "Enter a line of text:\n";
    std::getline(std::cin, text);                // Read a line including spaces

    int vowels {};                               // Count of vowels
    int consonants {};                           // Count of consonants
    for(size_t i {} ; i<text.length() ; ++i)
    {
        if(std::isalpha(text[i]))                // Check for a letter
            switch(std::tolower(text[i]))        // Convert to lowercase
            {
                case 'a': case 'e': case 'i': case 'o': case 'u':
                    ++vowels;
                    break;

                default:
                    ++consonants;
                    break;
            }
    }

    std::cout << "Your input contained " << vowels << " vowels and "
               << consonants << " consonants." << std::endl;
}

```

Here's an example of the output:

```

Enter a line of text:
A nod is as good as a wink to a blind horse.
Your input contained 14 vowels and 18 consonants.

```

The `text` object contains an empty string initially. You read a line from the keyboard into `text` using the `getline()` function. This version of `getline()` is declared in the `string` header; the versions of `getline()` that you have used previously were declared in the `iostream` header. This version reads characters from the stream specified by the first argument, `cin` in this case, until a newline character is read, and the result is stored in the `string` object specified by the second argument, which is `text` in this case. This time you don't need to worry about how many characters are in the input. The `string` object will automatically accommodate however many characters are entered and the length will be recorded in the object.

You can change the delimiter that signals the end of the input by using a version of `getline()` with a third argument that specifies the new delimiter for the end of the input:

```
std::getline(std::cin, text, '#');
```

This reads characters until a '#' character is read. Because newline doesn't signal the end of input in this case, you can enter as many lines of input as you like, and they'll all be combined into a single string. Any newline characters that were entered will be present in the string.

You count the vowels and consonants in much the same way as in `Ex5_11.cpp`, using a `for` loop. You could use the range-based `for` loop instead:

```
for (const auto& ch: text)
{
    if (isalpha(ch))                // Check for a letter
        switch (tolower(ch))        // Convert to lowercase
        {
            case 'a': case 'e': case 'i': case 'o': case 'u':
                ++vowels;
                break;

            default:
                ++consonants;
                break;
        }
}
```

In my mind this is better. The code is simpler and easier to understand than the original. The major advantage of using a `string` object in this example compared to `Ex5_11.cpp` is that you don't need to worry about the length of the string that is entered.

Accessing Substrings

You can extract a substring from a `string` object using its `substr()` function. The function requires two arguments. The first is the index position where the substring starts and the second is the number of characters in the substring. The function returns the substring as a `string` object. For example:

```
string phrase {"The higher the fewer."};
string word1 {phrase.substr(4, 6)};           // "higher"
```

This extracts the six-character substring from `phrase` that starts at index position 4, so `word1` will contain "higher" after the second statement executes. If the length you specify for the substring overruns the end of the `string` object, then the `substr()` function just returns an object contains the characters up to the end of the string. The following statement demonstrates this behavior:

```
string word2 {phrase.substr(4,100)};          // "higher the fewer."
```

Of course, there aren't 100 characters in `phrase`, let alone in a substring. In this case the result will be that `word2` will contain the substring from index position 4 to the end, which is "higher the fewer.". You could obtain the same result by omitting the length argument and just supplying the first argument that specifies the index of the first character of the substring:

```
string word {phrase.substr(4)};               // "higher the fewer."
```

This version of `substr()` also returns the substring from index position 4 to the end. If you omit both arguments to `substr()`, the whole of `phrase` will be selected as the substring.

If you specify a starting index for a substring that is outside the valid range for the `string` object, an *exception* of type `std::out_of_range` will be thrown and your program will terminate abnormally—unless you've implemented some code to handle the exception. You don't know how to do that yet but I'll discuss exceptions and how to handle them in Chapter 15.

Comparing Strings

You saw in the previous example how you can use an index to access individual characters in a `string` object for comparison purposes. When you access a character using an index, the result is of type `char`, so you can use the comparison operators to compare individual characters. You can also compare entire `string` objects using any of the comparison operators. The comparison operators you can use are:

> >= < <= == !=

You can use these to compare two objects of type `string`, or to compare a `string` object with a `string` literal or with a C-style string. The operands are compared character by character until either a pair of corresponding characters are different, or the end of either or both operands is reached. When a pair of characters differ, numerical comparison of the character codes determines which of the strings has the lesser value. If no differing character pairs are found and the strings are of different lengths, the shorter string is “less than” the longer string. Two strings are equal if they contain the same number of characters and all corresponding character codes are equal. Because you’re comparing character codes, the comparisons are obviously going to be case sensitive.

You could compare two `string` objects using this `if` statement:

```
string word1 {"age"};
string word2 {"beauty"};
if(word1 < word2)
    std::cout << word1 << " comes before " << word2 << "." << std::endl;
else
    std::cout << word2 << " comes before " << word1 << "." << std::endl;
```

Executing these statements will result in the output:

```
age comes before beauty.
```

This shows that the old saying must be true. The preceding code looks like a good candidate for using the conditional operator. You can produce a similar result with the following statement:

```
std::cout << word1 << (word1 < word2 ? " comes " : " does not come ")
    << "before " << word2 << "." << std::endl;
```

Let’s compare strings in a working example. This program reads any number of names and sorts them into ascending sequence:

```
// Ex7_03.cpp
// Comparing strings
#include <iostream>                // For stream I/O
#include <iomanip>                 // For stream manipulators
#include <string>                 // For the string type
#include <locale>                 // For character conversion
#include <vector>                 // For the vector container
using std::string;
```

```

int main()
{
    std::vector<string> names;           // Vector of names
    string input_name;                 // Stores a name
    char answer {};                    // Response to a prompt

    do
    {
        std::cout << "Enter a name: ";
        std::cin >> input_name;         // Read a name and...
        names.push_back(input_name);    // ...add it to the vector

        std::cout << "Do you want to enter another name? (y/n): ";
        std::cin >> answer;
    } while(tolower(answer) == 'y');

    // Sort the names in ascending sequence
    string temp;
    bool sorted {false};               // true when names are sorted
    while(!sorted)
    {
        sorted = true;
        for(size_t i {1} ; i < names.size() ; ++i)
        {
            if(names[i-1] > names[i])
            { // Out of order - so swap names
                temp = names[i];
                names[i] = names[i-1];
                names[i-1] = temp;
                sorted = false;
            }
        }
    }

    // Find the length of the longest name
    size_t max_length{};
    for(auto& name : names)
        if(max_length < name.length()) max_length = name.length();

    // Output the sorted names 5 to a line
    std::cout << "In ascending sequence the names you entered are:\n";
    size_t field_width = max_length + 2;
    size_t count {};
    for(auto& name : names)
    {
        std::cout << std::setw(field_width) << name;
        if(!(++count % 5)) std::cout << std::endl;
    }
    std::cout << std::endl;
}

```

Here's some sample output:

```

Enter a name: Zebediah
Do you want to enter another name? (y/n): y
Enter a name: Meshak
Do you want to enter another name? (y/n): y
Enter a name: Eshak
Do you want to enter another name? (y/n): y
Enter a name: Abegnego
Do you want to enter another name? (y/n): y
Enter a name: Moses
Do you want to enter another name? (y/n): y
Enter a name: Job
Do you want to enter another name? (y/n): n
In ascending sequence the names you entered are:
  Abegnego      Eshak      Job      Meshak      Moses
  Zebediah

```

The names are stored in a vector of string elements. Using a vector container means that an unlimited number of names can be accommodated and the container will keep track of how many there are, so there's no need to count them independently. The container also acquires heap memory as necessary to store the string objects, and deletes it when the vector is destroyed.

The sorting process is the bubble sort that you have seen applied to numerical values. Because you need to compare successive elements in the vector and swap them when necessary, the for loop iterates over the index values for vector elements; the range-based for loop is not suitable here.

The sorted names are output in a range-based for loop. You can do this because a vector container represents a range. To align the names vertically using the `setw()` manipulator, you need to know the maximum name length and this is found by the range-based for loop that precedes the output loop.

The compare() Function

The `compare()` function for a string object can compare the object with another string object, or with a string literal, or with a C-style string. Here's an example of an expression that calls `compare()` for a string object, `word`, to compare it with a string literal:

```
word.compare("and")
```

`word` is compared with the argument to `compare()`. The function returns the result of the comparison as a value of type `int`. This will be a positive integer if `word` is greater than "and", zero if `word` is equal to "and", and a negative integer if `word` is less than "and".

In the last example, you could have used the `compare()` function in place of using the comparison operator:

```

for(size_t i {1} ; i < names.size() ; ++i)
{
    if(names[i-1].compare(names[i]) > 0)
    { // Out of order - so swap names
        temp = names[i];
        names[i] = names[i-1];
        names[i-1] = temp;
        sorted = false;
    }
}

```

This is less clear than the original code, but you get the idea of how the `compare()` function can be used. The `>` operator is better in this instance but there are circumstances where `compare()` has the advantage. The function tells you in a single step the relationship between two objects. If `>` results in `false`, you still don't know whether or not the operands are equal whereas with `compare()` you do. The function has another advantage. You can `compare()` a substring of a string object with the argument:

```
string word1 {"A jackhammer"};
string word2 {"jack"};
int result{ word1.compare(2, word2.length(), word2) };
if (!result)
    std::cout << "word1 contains " << word2 << " starting at index 2" << std::endl;
```

The expression that initializes `result` compares the four-character substring of `word1` that starts at index position 2 with `word2`. This is illustrated in Figure 7-2.

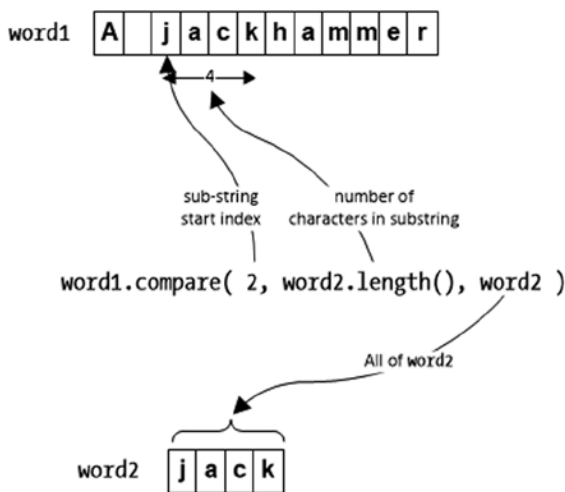


Figure 7-2. Using `compare()` with a substring

The first argument to `compare()` is the index position of the first character in a substring of `word1` that is to be compared with `word2`. The second argument is the number of characters in the substring, which is sensibly specified as the length of the third argument, `word2`. Obviously, if the substring length you specify is not the same as the length of the third argument, the substring and the third argument are unequal by definition.

You could use the `compare` function to search for a substring. For example:

```
string text {"Peter Piper picked a peck of pickled pepper."};
string word {"pick"};
for (size_t i{}; i < text.length() - word.length() + 1; ++i)
    if (!text.compare(i, word.length(), word))
        std::cout << "text contains " << word << " starting at index " << i << std::endl;
```

This loop finds `word` at index positions 12 and 29 in `text`. The upper limit for the loop variable allows the last `word.length()` characters in `text` to be compared with `word`. This is not the most efficient implementation of the search. When `word` is found, it would be more efficient to arrange than the next substring of `text` that is checked is `word.length()` characters further along, but only if there is still `word.length()` characters before the end of `text`. However, there are easier ways to search a string object, as you'll see very soon.

You can compare a substring of one string with a substring of another using the `compare()` function. This involves passing *five* arguments to `compare()`! For example:

```
string text {"Peter Piper picked a peck of pickled pepper."};
string phrase {"Got to pick a pocket or two."};
for (size_t i{}; i < text.length() - 3 ; ++i)
    if (!text.compare(i, 4, phrase, 7, 4))
        std::cout << "text contains " << phrase << " starting at index " << i << std::endl;
```

The two additional arguments are the index position of the substring in phrase and its length. The substring of text is compared with the substring of text.

We're not done yet! The `compare()` function can also compare a substring of a string object with a null-terminated string:

```
string text{ "Peter Piper picked a peck of pickled pepper." };
for (size_t i{}; i < text.length() - 3 ; ++i)
    if (!text.compare(i, 4, "pick"))
        std::cout << "text contains \"pick\" starting at index " << i << std::endl;
```

The output from this will be the same as the previous code; "pick" is found at index positions 12 and 29.

Still another option is to select the first *n* characters from a null-terminated string by specifying the number of characters. The `if` statement in the loop could be:

```
if (!text.compare(i, 4, "picket", 4))
    std::cout << "text contains \"pick\" starting at index " << i << std::endl;
```

The fourth argument to `compare()` specifies the number of characters from "picket" that are to be used in the comparison.

■ **Note** You have seen that the `compare()` function works quite happily with different numbers of arguments of various types. What you have here are several different functions with the same name. These are called *overloaded functions*, and you'll learn how and why you create them in the next chapter.

Comparisons Using `substr()`

Of course, if you're like me you have trouble remembering the sequence of arguments to the more complicated versions of the `compare()` function, you can use the `substr()` function to extract the substring of a string object. You can then use the result with the comparison operators in many cases. For instance, to compare substrings in `word1` and `word2` as shown in Figure 6-9 in the previous chapter, you could write the test as follows:

```
string text {"Peter Piper picked a peck of pickled pepper."};
string phrase {"Got to pick a pocket or two."};
for (size_t i{}; i < text.length() - 3 ; ++i)
    if (text.substr(i, 4) == phrase.substr(7, 4))
        std::cout << "text contains " << phrase.substr(7, 4)
            << " starting at index " << i << std::endl;
```

This seems to me to be more readily understood than the equivalent operation using the `compare()` function.

Searching Strings

Beyond `compare()`, you have many other alternatives for searching a string object. They all involve functions that return an index. I'll start with the simplest sort of search. A string object has a `find()` function that finds the index of a substring within it. You can also use it to find the index of a given character. The substring you are searching for can be another string object or a string literal. Here's an example:

```
string sentence {"Manners maketh man"};
string word {"man"};
std::cout << sentence.find(word) << std::endl;           // Outputs 15
std::cout << sentence.find("Man") << std::endl;          // Outputs 0
std::cout << sentence.find('k') << std::endl;            // Outputs 10
std::cout << sentence.find('x') << std::endl;            // Outputs string::npos
```

In each output statement sentence is searched from the beginning by calling its `find()` function. The function returns the index of the first character of the first occurrence of whatever is being sought. In the last statement, 'x' is not found in the string so the value `string::npos` is returned. This is a constant that is defined in the string header. It represents an illegal character position in a string and is used to signal a failure in a search. Of course, you can use `string::npos` to check for a search failure with a statement such as this:

```
if(sentence.find('x') == string::npos)
    std::cout << "Character not found" << std::endl;
```

Another variation on the `find()` function allows you to search part of a string starting from a specified position. For example, with sentence defined as before, you could write this:

```
std::cout << sentence.find("an", 1) << std::endl;        // Outputs 1
std::cout << sentence.find("an", 3) << std::endl;        // Outputs 16
```

Each statement searches sentence from the index specified by the second argument, to the end of the string. The first statement finds the first occurrence of "an" in the string. The second statement finds the second occurrence because the search starts from index position 3.

You could search for a string object by specifying it as the first argument to `find()`. For example:

```
string sentence {"Manners maketh man"};
string word {"an"};
int count {};                                           // Count of occurrences
size_t position {};                                    // Stores a string index position
for(size_t i {}; i <= sentence.length() - word.length(); i++)
{
    position = sentence.find(word, i);
    if(position == string::npos)
        break;
    count++;
    i = position + 1;
}
std::cout << "\"" << word << "\" occurs in \"" << sentence
    << "\" " << count << " times." << std::endl;    // Two times...
```

A string index is of type `size_t`, so position that stores values returned by `find()` is of that type. The loop index, `i`, defines the starting position for a `find()` operation so this is also of type `size_t`. The last occurrence of `word` in sentence has to start at least `word.length()` positions back from the end of sentence, so the maximum value of `i` in

the loop is `sentence.length()-word.length()`. There's no loop expression for incrementing `i` because this is done in the loop body.

If `find()` returns `string::npos`, then `word` wasn't found, so the loop ends by executing the `break` statement. Otherwise `count` is incremented and `i` is set to one position beyond where `word` was found, ready for the next iteration. You might think you should set `i` to be `i+word.length()`, but this wouldn't allow overlapping occurrences to be found, such as if you were searching for "anna" in the string "annannanna".

You can also search a `string` object for a substring of a C-style string or a string literal. In this case, the first argument to `find()` is the null-terminated string, the second is the index position at which you want to start searching, and the third is the number of characters of the null-terminated string that you want to take as the string you're looking for. For example:

```
std::cout << sentence.find("akat", 1, 2) << std::endl;    // Outputs 9
```

This searches for the first two characters of "akat" (that is, "ak") in `sentence`, starting from position 1. The following searches would both fail and return `string::npos`:

```
std::cout << sentence.find("akat", 1, 3) << std::endl;    // Outputs string::npos
std::cout << sentence.find("akat", 10, 2) << std::endl;    // Outputs string::npos
```

The first search fails because "aka" isn't in `sentence`. The second is looking for "ak", which *is* in `sentence`, but it fails because it doesn't occur after position 10.

Here is a program that searches a `string` object for a given substring and determines how many times the substring occurs:

```
// Ex7_04.cpp
// Searching a string
#include <iostream>
#include <string>
using std::string;

int main()
{
    string text{};                // The string to be searched
    string word{};                // Substring to be found
    std::cout << "Enter the string to be searched and press Enter:\n";
    std::getline(std::cin, text);

    std::cout << "Enter the string to be found and press Enter:\n";
    std::getline(std::cin, word);

    size_t count{};               // Count of substring occurrences
    size_t index{};               // String index
    while ((index = text.find(word, index)) != string::npos)
    {
        ++count;
        index += word.length();
    }
    std::cout << "Your text contained " << count << " occurrences of \""
              << word << "\"." << std::endl;
}
```

Here's some sample output:

```
Enter the string to be searched and press Enter:
Smith, where Jones had had "had had", had had "had". "Had had" had had the examiners' approval.
Enter the string to be found and press Enter:
had
Your text contained 10 occurrences of "had".
```

There are only 10 occurrences of "had". "Had" doesn't count because it starts with an uppercase letter. The program searches text for the string in word, both of which are read from the standard input stream using `getline()`. Input is terminated by a newline, which occurs when you press Enter. The search is conducted in the while loop, which continues as long as the `find()` function for text does not return `string::npos`. A return value of `string::npos` indicates that the search target is not found in text from the specified index to the end of the string, so the search is finished. On each iteration when a value other than `string::npos` is returned, the string in word has been found in text so count is incremented and index is incremented by the length of the string; this assumes that we are not searching for overlapping occurrences. You could code the search as a for loop with no statements in the body of the loop:

```
for (size_t index{};
     (index = text.find(word, index)) != string::npos; index += word.length(), ++count)
    ;
```

I think the while loop is easier to understand. Whichever loop you prefer, the mechanism is essentially the same. There is quite a lot happening in either loop, so to help you follow the action, the process is shown in Figure 7-3.

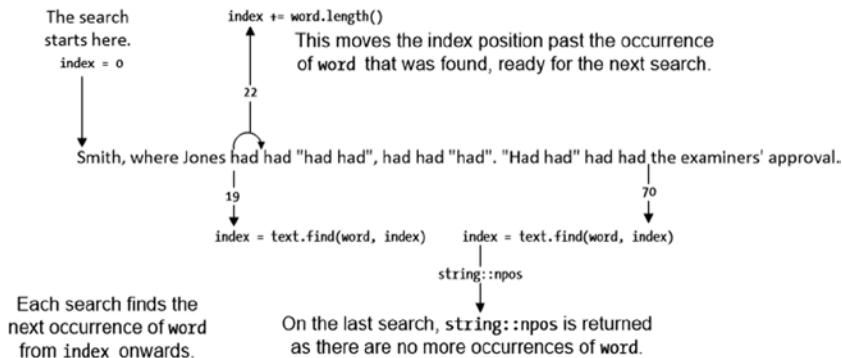


Figure 7-3. Searching a string

Searching for any of a Set of Characters

Suppose you have a string — a paragraph of prose, perhaps — that you want to break up into individual words. You need to find where the separators are, and those could be any of a number of different characters such as spaces, commas, periods, colons, and so on. A function that can find any of a given set of characters in a string would help. This is exactly what the `find_first_of()` function for s string object does:

```
string text {"Smith, where Jones had had \"had had\", had had \"had\"."
            " \"Had had\" had had the examiners' approval."};
string separators {" ,.\\"};
std::cout << text.find_first_of(separators) << std::endl;           // Outputs 5
```

The set of characters sought are defined by a string object that you pass as the argument to the `find_first_of()` function. The first character in `text` that's in `separators` is a comma, so the last statement will output 5. You can also specify the set of separators as a null-terminated string. If you want to find the first vowel in `text`, for example, you could write this:

```
std::cout << text.find_first_of("AaEeIiOoUu") << std::endl;    // Outputs 2
```

The first vowel in `text` is 'i', at index position 2.

You can search backwards from the end of a string object to find the *last* occurrence of a character from a given set by using the `find_last_of()` function. For example, to find the last vowel in `text`, you could write this:

```
std::cout << text.find_last_of("AaEeIiOoUu") << std::endl;    // Outputs 92
```

The last vowel in `text` is the second 'a' in `approval`, at index 92.

You can specify an extra argument to `find_first_of()` and `find_last_of()` that specifies the index where the search process is to begin. If the first argument is a null-terminated string, there's an optional third argument that specifies how many characters from the set are to be included.

A further option is to find a character that's *not* in a given set. The `find_first_not_of()` and `find_last_not_of()` functions do this. To find the position of the first character in `text` that isn't a vowel, you could write:

```
std::cout << text.find_first_not_of("AaEeIiOoUu") << std::endl; // Outputs 0
```

The first character that isn't a vowel is clearly the first, at index 0.

Let's try some of these functions in a working example. This program extracts the words from a string. This combines the use of `find_first_of()` and `find_first_not_of()`. Here's the code:

```
// Ex7_05.cpp
// Searching a string for characters from a set
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using std::string;

int main()
{
    string text;                                // The string to be searched
    std::cout << "Enter some text terminated by *:\n";
    std::getline(std::cin, text, '*');

    const string separators{ " ,;:.\",!?'\n" };    // Word delimiters
    std::vector<string> words;                    // Words found

    size_t start { text.find_first_not_of(separators) }; // First word start index
    size_t end {};                                // Index for end of a word

    while (start != string::npos)                // Find the words
    {
        end = text.find_first_of(separators, start + 1); // Find end of word
        if (end == string::npos)                    // Found a separator?
            end = text.length();                    // No, so set to last + 1
    }
```

```

    words.push_back(text.substr(start, end - start));    // Store the word
    start = text.find_first_not_of(separators, end + 1); // Find 1st character of next word
}

std::cout << "Your string contains the following " << words.size() << " words:\n";
size_t count{};                                     // Number output
for (const auto& word : words)
{
    std::cout << std::setw(15) << word;
    if (!(++count % 5))
        std::cout << std::endl;
}
std::cout << std::endl;
}

```

Here's some sample output:

```

Enter some text terminated by *:
To be, or not to be, that is the question.
Whether tis nobler in the mind to suffer the slings and
arrows of outrageous fortune, or by opposing, end them.*
Your string contains the following 30 words:

```

To	be	or	not	to
be	that	is	the	question
Whether	tis	nobler	in	the
mind	to	suffer	the	slings
and	arrows	of	outrageous	fortune
or	by	opposing	end	them

The string variable, `text`, will contain a string read from the keyboard. The string is read from `cin` by the `getline()` function with an asterisk specified as the termination character, which allows multiple lines to be entered. The `separators` variable defines the set of word delimiters. It's defined as `const` because these should not be modified. The interesting part of this example is the analysis of the string.

You record the index of the first character of the first word in `start`. As long as this is a valid index, which is a value other than `string::npos`, you know that `start` will contain the index of the first character of the first word. The `while` loop finds the end of the current word, extracts the word as a substring and stores it in the `words` vector. It also records the result of searching for the index of the first character of the next word in `start`. The loop continues until a first character is not found, in which case `start` will contain `string::npos` to terminate the loop.

It's possible that the last search in the `while` loop will fail, leaving `end` with the value `string::npos`. This can occur if `text` ends with a letter or anything other than one of the specified separators. To deal with this, you check the value of `end` in the `if` statement, and if the search did fail, you set `end` to the length of `text`. This will be one character beyond the end of the string (because indexes start at 0, not 1), because `end` should correspond to the position *after* the last character in a word.

The program could have used a `vector<std::shared_ptr<string>>` object to store the words. In this case `words` would be stored on the heap. There are few changes necessary beyond the definition of the vector. The statement in the `while` loop that stores words would need to be:

```

words.push_back(std::make_shared<string>(text.substr(start, end - start))); // Store word

```

Apart from that, only the output loop would need a tiny change:

```
for (const auto& word : words)
{
    std::cout << std::setw(15) << *word;
    if (!(++count % 5))
        std::cout << std::endl;
}
```

Dereferencing the word loop variable is the only alteration necessary. Using `shared_ptr<T>` objects makes using heap memory very easy.

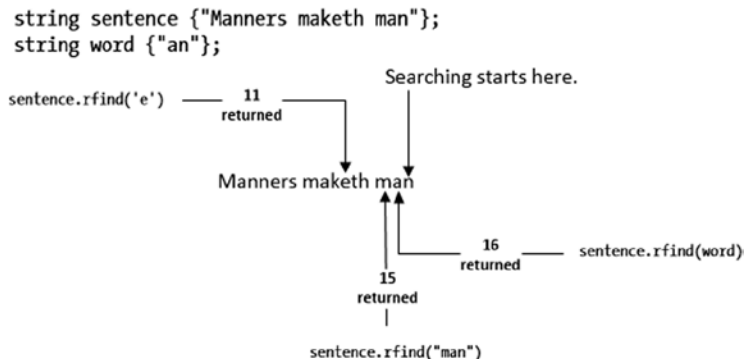
Searching a String Backwards

The `find()` function searches forward through a string, either from the beginning or from a given index. The `rfind()` function, perhaps named from reverse find, searches a string in reverse. `rfind()` comes in the same range of varieties `find()`. You can search a whole string object for a substring that you can define as another string object or as a null-terminated string. You can also search for a character. For example:

```
string sentence {"Manners maketh man"};
string word {"an"};
std::cout << sentence.rfind(word) << std::endl;    // Outputs 16
std::cout << sentence.rfind("man") << std::endl;   // Outputs 15
std::cout << sentence.rfind('e') << std::endl;     // Outputs 11
```

Each search finds the last occurrence of the argument to `rfind()` and returns the index of the first character where it was found. Figure 7-4 illustrates the use of `rfind()`.

Using `rfind()` with one argument



As with `find()`, if the argument is not found then the value `string::npos` is returned.

Figure 7-4. Searching backwards through a string

Searching with `word` as the argument finds the last occurrence of "an" in the string. The `rfind()` function returns the index position of the first character in the substring sought.

If the substring isn't present, `string::npos` will be returned. For example, the following statement will result in this:

```
std::cout << sentence.rfind("miners") << std::endl;    // Outputs string::npos
```

`sentence` doesn't contain the substring "miners" so `string::npos` will be returned and displayed by this statement. The other two searches illustrated in Figure 7-4 are similar to the first. They both search backward from the end of the string looking for the first occurrence of the argument.

Just as with `find()`, you can supply an extra argument to `rfind()` to specify the starting index for the backward search, and you can add a third argument when the first argument is a C-style string. The third argument specifies the number of characters from the C-style string that are to be taken as the substring for which you're searching.

Modifying a String

When you've searched a string and found what you're looking for, you may well want to change the string in some way. You've already seen how you can use an index between square brackets to select a single character in a string object. You can also insert a string into a string object at a given index or replace a substring. Unsurprisingly, to insert a string you use a function called `insert()`, and to replace a substring in a string you use a function called `replace()`. I'll explain inserting a string first.

Inserting a String

Perhaps the simplest sort of insertion involves inserting a string object before a given position in another string object. Here's an example of how you do this:

```
string phrase {"We can insert a string."};
string words {"a string into "};
phrase.insert(14, words);
```

Figure 7-5 illustrates what happens. The `words` string is inserted immediately *before* the character at index 14 in `phrase`. After the operation, `phrase` will contain the string "We can insert a string into a string."

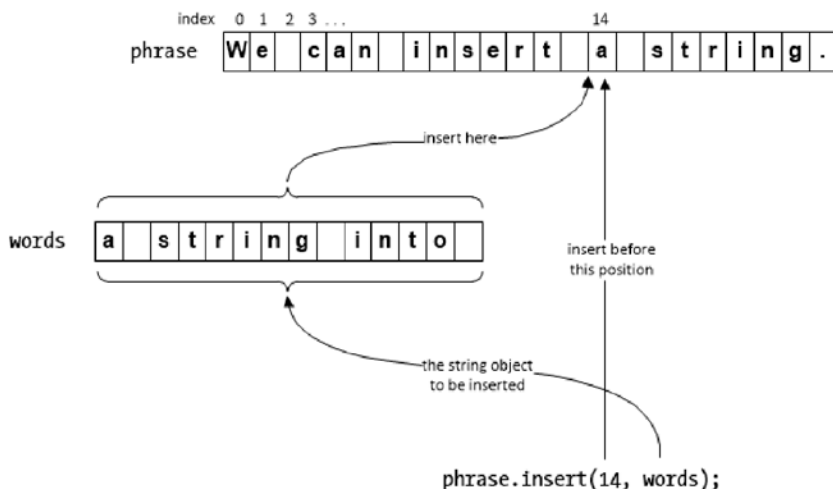


Figure 7-5. Inserting a string into another string

You can also insert a null-terminated string into a `string` object. For example, you could achieve the same result as the previous operation with this statement:

```
phrase.insert(14, "a string into ");
```

Of course, the `'\0'` character is discarded from a null-terminated string before insertion because it's a delimiter and not part of the string proper.

The next level of sophistication is the insertion of a substring of a `string` object into another `string` object. You need to supply two extra arguments to `insert()`: one specifies the index of the first character in the substring to be inserted, and the other specifies the number of characters in the substring. For example:

```
phrase.insert(13, words, 8, 5);
```

This inserts the five-character substring that starts at position 8 in `words`, into `phrase`, preceding index position 13. Given that `phrase` and `words` contain the strings as earlier, this inserts " into" into "We can insert a string.", so that `phrase` becomes "We can insert into a string."

There is a similar facility for inserting a number of characters from a null-terminated string into a `string` object. The following statement produces the same result as the previous one:

```
phrase.insert(13, " into something", 5);
```

This inserts the first five characters of " into something" into `phrase` preceding the character at index 13.

There's a version of `insert()` that inserts a sequence of identical characters:

```
phrase.insert(16, 7, '*');
```

This inserts five asterisks in `phrase` immediately before the character at index 13. `phrase` will then contain the uninformative sentence "We can insert a *****string."

Replacing a Substring

You can replace any substring of a `string` object with a different string — even if the inserted string and the substring to be replaced have different lengths. I'll return to an old favorite and define `text` like this:

```
string text {"Smith, where Jones had had \"had had\", had had \"had\"."};
```

You can replace "Jones" with a less common name with this statement:

```
text.replace(13, 5, "Gruntfuttock");
```

The first argument is the index in `text` of the first character of the substring to be replaced and the second is the length of the substring. Thus this replaces the five characters of `text` that start at index 13 with "Gruntfuttock". If you now output `text`, it would be:

```
Smith, where Gruntfuttock had had "had had" had had "had".
```

A more realistic application of this is to search for the substring to be replaced first, for example:

```
const string separators {" ,;:.\'!\\n"};           // Word delimiters
size_t start {text.find("Jones")};               // Find the substring
size_t end {text.find_first_of(separators, start + 1); // Find the end
text.replace(start, end - start, "Gruntfuttock");
```


This finds the position of the first character of "Jones" in text and uses it to initialize start. The character following the last character of "Jones" is found next by searching for a delimiter from separators using the `find_first_of()` function. These index positions are used in the `replace()` operation.

The replacement string can be a string object or a null-terminated string. In the former case, you can specify a start index and a length to select a substring as the replacement string. For example, the previous replace operation could have been:

```
string name {"Amos Gruntfuttock"};
text.replace(start, end - start, name, 5, 12);
```

These statements have the same effect as the previous use of `replace()`, because the replacement string starts at position 5 of name (which is the 'G') and contains 12 characters.

If the first argument is a null-terminated string, you can specify the number of characters that are the replacement string, for example:

```
text.replace(start, end - start, "Gruntfuttock, Amos", 12);
```

This time, the string to be substituted consists of the first 12 characters of "Gruntfuttock, Amos", so the effect is exactly the same as the previous replace operation.

A further possibility is to specify the replacement string as multiples of a given character. For example, you could replace "Jones" by three asterisks with this statement:

```
text.replace(start, end - start, 3, '*');
```

This assumes that start and end are determined as before. The result is that text will contain:

Smith, where *** had had "had had" had had "had".

Let's try the replace operation in an example. This program replaces a given word in a string with another word:

```
// Ex7_06.cpp
// Replacing words in a string
#include <iostream>
#include <string>
using std::string;

int main()
{
    string text;                                     // The string to be modified
    std::cout << "Enter a string terminated by *\n";
    std::getline(std::cin, text, '*');

    string word;                                     // The word to be replaced
    std::cout << "Enter the word to be replaced: ";
    std::cin >> word;

    string replacement;                             // The word to be substituted
    std::cout << "Enter the string to be substituted for " << word << ": ";
    std::cin >> replacement;
```

```

if (word == replacement)                // Verify there's something to do
{
    std::cout << "The word and its replacement are the same.\n"
    << "Operation aborted." << std::endl;
    return 1;
}

size_t start {text.find(word)};          // Index of 1st occurrence of word
while (start != string::npos)            // Find and replace all occurrences
{
    text.replace(start, word.length(), replacement); // Replace word
    start = text.find(word, start + replacement.length());
}

std::cout << "\nThe string you entered is now:\n"
    << text << std::endl;
}

```

Here's a sample of the output:

```

Enter a string terminated by *:
A rose is a rose is a rose.*
Enter the word to be replaced: rose
Enter the string to be substituted for rose: dandelion

```

```

The string you entered is now:
A dandelion is a dandelion is a dandelion.

```

The string that is to have words replaced is read into `text` by `getline()`. Any number of lines can be entered and terminated by an asterisk. The word to be replaced and its replacement are read using the extraction operator and therefore cannot contain whitespace. The program ends immediately if the word to be replaced and its replacement are the same.

The index position of the first occurrence of `word` is used to initialize `start`. This is used in the `while` loop that finds and replaces successive occurrences of `word`. After each replacement, the index for the next occurrence of `word` in `text` is stored in `start`, ready for the next iteration. When there are no further occurrences of `word` in `text`, `start` will contain `string::npos`, which ends the loop. The modified string in `text` is then output.

Removing Characters from a String

You can remove a substring from a string object using the `replace()` function. You just specify the replacement as an empty string. There's also a specific function for this purpose, `erase()`. You specify the substring to be erased by the index position of the first character and the length. For example, you could erase the first six characters from `text` like this:

```
text.erase(0, 6);                // Remove the first 6 characters
```

You would more typically use `erase()` to remove a specific substring that you had previously searched for so a more usual example might be:

```

string word {"rose"};
size_t index {text.find(word)};
if(index != string::npos)
    text.erase(index, word.length());

```

This searches for `word` in `text`, and after confirming that it exists, removes it using `erase()`. The number of characters in the substring to be removed is obtained by calling the `length()` function for `word`.

The `clear()` function removes all the characters from a `string` object, for example:

```
text.clear();
```

After this statement executes, `text` will be an empty string.

Strings of International Characters

Supporting multiple national character sets is an advanced topic so I'll only introduce the basic facilities that C++ offers, without going into detail of how you apply any them. Thus this section is just a pointer to where you should look when you have to work with more than one national character set. Potentially, you have three options for working with strings that may contain extended character sets:

- You can define `wstring` objects that contain strings of characters of type `wchar_t` - the wide-character type that is built into C++.
- You can define `u16string` objects that store strings of 16-bit Unicode characters, which are of type `char16_t`.
- You can define `u32string` objects that contain strings of 32-bit Unicode characters, which are of type `char32_t`.

The `string` header defines all these types; the last two are more useful than the `string` type that stores `wchar_t` characters.

In theory you can use the `std::string` type you have explored in detail this chapter to store strings of as UTF-8 characters. You define a UTF-8 string by prefixing a regular string literal with `u8`, for example: `u8"This is a UTF-8 string."`. However, the `string` type stores characters as type `char`, and knows nothing about Unicode encodings. The UTF-8 encoding uses from 1 to 4 bytes to encode each character and the functions that operate on string objects will not recognize this. This means for instance that the `length()` function will return the wrong length if the string includes any characters that require two or three bytes to represent them.

Strings of `wchar_t` Characters

The `std::wstring` type that is defined in the `string` header stores strings of characters of type `wchar_t`. You use objects of type `wstring` in essentially the same way as objects of type `string`. You could define a wide string object with this statement:

```
wstring quote;
```

This assumes you have a `using` declaration for `std::wstring` in the source file. You write string literals containing characters of type `wchar_t` between double quotes, but with `L` prefixed to distinguish them from string literals containing `char` characters. Thus you can define and initialize a `wstring` variable like this:

```
wstring saying {L"The tigers of wrath are wiser than the horses of instruction."};
```

The `L` preceding the opening double quote specifies the literal consists of characters of type `wchar_t`. Without it, you would have a `char` string literal and the statement would not compile.

To output wide strings you use the `wcout` stream, for example:

```
std::wcout << saying << std::endl;
```

All the functions I've discussed in the context of string objects apply equally well for `wstring` objects, so I won't wade through them again. Just remember to specify the `L` prefix with string and character literals when you are working with `wstring` objects. The problem with type `wstring` is that the character encoding that applies with type `wchar_t` is implementation defined, so it can vary from one compiler to another. If you need to support multi-national character sets, you are much better off using either types `u16string` or `u32string` that are described in the next section.

Objects that contain Unicode Strings

The string header defines two further types that store strings of Unicode characters. Objects of type `std::u16string` stores strings of characters of type `char16_t` and objects of type `std::u32string` store strings of characters of type `char32_t`. Like `wstring` objects, you must use a literal of the appropriate type to initialize a `u16string` or `u32string` object. For example:

```
u16string question {u"Whither atrophy?"};           // char16_t characters
u32string sentence {U"This sentence contains three errors."}; // char32_t characters
```

These statements demonstrate that you prefix a string literal containing `char16_t` characters with `u` and a literal containing `char32_t` characters with `U`. Objects of the `u16string` and `u32string` types have the same set of functions as the string type.

Raw String Literals

A regular expression is a string that defines a process for searching and transforming text. Essentially a regular expression defines patterns that are to be matched in a string, and patterns that are found can be replaced or reordered. C++ supports regular expressions but I won't be discussing them in this book for reasons of space as much as anything. I'm mentioning them here because they influence how string literals can be defined and you may come across the new type of string literal. If you want to know more about regular expressions, they are supported by the `regex` header.

Regular expression strings usually contain many backslash characters. In the string literals you have seen so far, a backslash has a special meaning: it prefixes an escape sequence, so including a backslash character means having to use `\\`. Having to use the escape sequence for each backslash character can make a regular expression difficult to specify correctly and very hard to read. The *raw string literal* was introduced to solve the problem. A raw string literal can include any character, including backslashes, tabs, and newlines, so no escape sequences are necessary. A raw string literal includes an `R` in the prefix, and any of the types of literal you have seen can also be specified as raw literals. Here's an example:

```
R"(The "\"" escape sequence is a backslash character, \.)"
```

The `R` prefix specifies that this is a raw string literal. If you defined this as a standard string literal it would be:

```
"The \"\\\" escape sequence is a backslash character, \\\."
```

Not exactly as readable as the raw version, is it? All characters between the double quotes in a raw string literal are included in the literal. Escape sequences are not recognized in a raw string literal. So how do you include a double quote in a raw string literal? It's covered. The delimiters that mark the start and end of a raw string literal are flexible. You can use any delimiter of the form `"char_sequence(to mark the beginning of the literal as long as you mark the end with a matching sequence,) char_sequence"`. `char_sequence` is any sequence of up to 16 characters, and you must use the same sequence at both ends. `char_sequence` must not include parentheses, spaces, control characters, or backslash characters. Here's an example:

```
RU"*(The answer is "a - b" not "(c - d)")*"
```

This is a raw string literal that contains `char32_t` characters. You can specify any type of string literal as raw, just by including `R` in the prefix. The delimiter that marks the beginning is `"*(` and the delimiter at the end is `)*"`. As a standard literal this would be:

```
"The answer is \"a - b\" not \"(c - d)\""
```

Summary

In this chapter you learned how you can use the `string` type that's defined in the standard library. The `string` type is much easier and safer to use than C-style strings, so it should be your first choice when you need to process character strings.

The important points from this chapter are:

- The `std::string` type stores a character string without a termination character. The terminating null is unnecessary because a `string` object keeps track of the length of the string.
- You can access and modify individual characters in a `string` object using an index between square brackets. Index values for characters in a `string` object start at 0.
- You can use the `+` operator to concatenate a `string` object with a string literal, a character, or another `string` object.
- Objects of type `string` have functions to search, modify, and extract substrings.
- You can store `string` objects in an array, or better still, in a sequence container such as a vector.
- Objects of type `wstring` contain strings of characters of type `wchar_t`.
- Objects of type `u16string` contain strings of characters of type `char16_t`.

Objects of type `u32string` contain strings of characters of type `char32_t`.

EXERCISES

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck after that, you can download the solutions from the Apress website (www.apress.com/source-code), but that really should be a last resort.

Exercise 7-1. Write a program that reads and stores the first names of any number of students, along with their grades. Calculate and output the average grade, and output the names and grades of all the students in a table with the name and grade for three students on each line.

Exercise 7-2. Write a program that reads text entered over an arbitrary number of lines. Find and record each unique word that appears in the text and record the number of occurrences of each word. Output the words and the number of occurrences of each word, three words and their counts per line. Words and counts should align in columns.

Exercise 7-3. Write a program that reads a text string of arbitrary length from the keyboard and prompt for entry of a word that is to be found in the string. The program should find and replace all occurrences of this word, regardless of case, by as many asterisks as there are characters in the word. It should then output the new string. Only whole words are to be replaced. For example, if the string is "Our house is at your disposal." and the word that is to be found is "our," then the resultant string should be: "*** house is at your disposal." and not "*** house is at y*** disposal."

Exercise 7-4. Write a program that prompts for input of two words and determines whether one is an anagram of the other.

Exercise 7-5. Write a program that reads a text string of arbitrary length from the keyboard followed by a string containing one or more letters. Output a list of all the whole words in the text that begin with any of the letters, upper or lowercase.



Defining Functions

Segmenting a program into manageable chunks of code is fundamental to programming in every language. A *function* is a basic building block in C++ programs. So far every example has had one function, `main()`, and that has typically used functions from the Standard Library. This chapter is all about defining your own functions with names that you choose.

In this chapter you will learn:

- What a function is, and why you should segment your programs into functions
- How to declare and define functions
- How data is passed to a function and how a function can return a value
- What “pass-by-value” means
- How specifying a parameter as a pointer affects the pass-by-value mechanism
- How using `const` as a qualifier for a parameter type affects the operation of a function
- What “pass-by-reference” means, and how you can declare a reference in your program
- How to return a value from a function
- What an inline function is
- The effect of defining a variable as `static` within a function

Segmenting Your Programs

All the programs you have written so far have consisted of just one function, `main()`. A real world C++ application consists of many functions, each of which provides a distinct well-defined capability. Execution starts in `main()`, which must be defined in the global namespace. `main()` calls other functions, each of which may call other functions, and so on. The functions other than `main()` can be defined in a namespace that you create.

When one function calls another that calls another that calls another, you have a situation where several functions are in action concurrently. Each that has called another that has not yet returned, will be waiting for the function that was called to end. Obviously something must keep track of from where in memory each function call was made and where execution should continue when a function returns. This information is recorded and maintained automatically in the *stack*. I introduced the stack when I explained heap memory and the stack is often referred to as the *call stack* in this context. The call stack records all the outstanding function calls and details of the data that was passed to each function. The debugging facilities that come with most C++ development systems usually provide ways for you to view the call stack while your program executes.

Functions in Classes

A class defines a new type and each class definition will usually contain functions that represent the operations that can be carried out with objects of the class type. You have already used functions that belong to a class extensively. In the previous chapter you used functions that belonged to the `string` class, such as the `length()` function that returned the number of characters in the `string` object and the `find()` function for searching a string. The standard input and output stream, `cin` and `cout` are objects, and using the stream insertion and extraction operators calls functions for those objects. Functions that belong to classes are fundamental in object-oriented programming, which you'll learn about from Chapter 11 onwards.

Characteristics of a Function

A function should perform a single, well-defined action and should be relatively short. Most functions do not involve many lines of code, certainly not hundreds of lines. This applies to all functions, including those that are defined within a class. Several of the working examples you have seen earlier could easily be divided into functions. If you look again at `Ex7_05.cpp`, you can see that what the program does falls naturally into three distinct actions: first, the text is read from the input stream, second the words are extracted from the text, and finally the words that were extracted are output. Thus the program could be defined as three functions that perform these actions, plus the `main()` function that calls them.

Defining Functions

A function is a self-contained block of code with a specific purpose. Function definitions in general have the same basic structure as `main()`. A function definition consists of a *function header* followed by a block that contains the code for the function. The function header specifies three things:

- The return type, which is the type of value, if any, that the function returns when it finishes execution. A function can return data of any type, including fundamental types, class types, pointer types, or reference types. It can also return nothing, in which case you specify the return type as `void`.
- The name of the function. Functions are named according to the same rules as variables.
- The number and types of data items that can be passed to the function when it is called. This is called the *parameter list* and it appears between parentheses following the function name.

A general representation of a function looks like this:

```
return_type function_name (parameter_list)
{
    // Code for the function...
}
```

Figure 8-1 shows an example of a function definition.

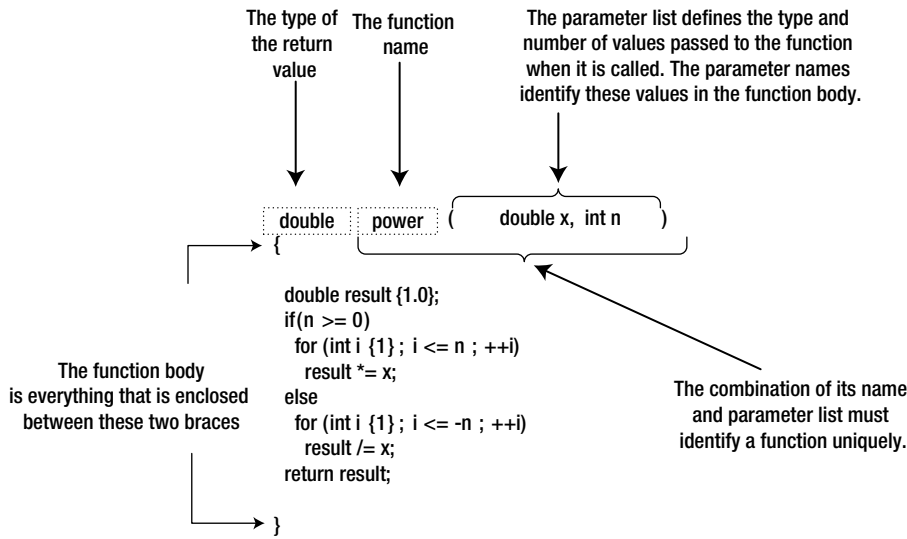


Figure 8-1. An example of a function definition

If nothing is to be passed to a function when it is called, then nothing appears between the parentheses. If there is more than one item in the parameter list, they are separated by commas. The `power()` function in Figure 8-1 has two parameters, `x` and `n`. The parameter names are used in the body of the function to access the corresponding values that were passed to the function. The term *argument* is used for a value that corresponds to a *parameter* in a function call. The code in the function executes with the argument values initializing the corresponding parameters. The sequence of the arguments in a function call must correspond to the sequence of the parameters in the parameter list in the function definition. The data types of the arguments should correspond to those demanded by the parameter list: the compiler won't necessarily warn you if it needs to make implicit type conversions, so you run the risk of losing information.

The combination of the function name and the parameter list is called the *signature* of a function. The compiler uses the signature to decide which function is to be called in any particular instance. Thus functions that have the same name, must have parameter lists that differ in some way to allow them to be distinguished. The return type is not part of the function signature. A function that returns a value can be called without storing or using the value it returns. In this case, the compiler cannot distinguish functions with signatures that only differ in the type of value returned so the return type is not part of the signature.

The `void` keyword is used to specify that a function does not return a value. `void*` is also used to specify a pointer type that is “pointer to an unspecified type”. Thus `void` can mean “nothing at all” in some contexts or “any type” in other contexts. `void*` can be used as a parameter type to allow an argument that is a pointer to any type to be passed as the value for the parameter. You'll see later in the book that there are ways to figure out what the type actually is in this situation.

■ **Note** A function with a return type specified as `void` doesn't return a value so it can't be used in an expression. Attempting to use such a function in this way will cause a compiler error message.

The Function Body

Calling a function executes the statements in the function body with the parameters having the values you pass as arguments. In Figure 8-1, the first line of the function body defines the double variable, `result`, initialized with 1.0. `result` is an automatic variable so only exists within the body of the function. This means that `result` ceases to exist after the function finishes executing.

The calculation is performed in one of two `for` loops, depending on the value of `n`. If `n` is greater than or equal to zero, the first `for` loop executes. If `n` is zero, the body of the loop doesn't execute at all because the loop condition is immediately false. In this case, `result` is left at 1.0. Otherwise, the loop variable `i` assumes successive values from 1 to `n`, and `result` is multiplied by `x` on each iteration. If `n` is negative, the second `for` loop executes, which divides `result` by `x` on each loop iteration.

The variables that you define within the body of a function and all the parameters are local to the function. You can use the same names in other functions for quite different purposes. The scope of each variable you define within a function is from the point at which it is defined until the end of the block that contains it. The only exceptions to this rule are variables that you define as `static` and I'll discuss these later in the chapter.

Let's give the `power()` function a whirl in a complete program.

```
// Ex8_01.cpp
// Calculating powers
#include <iostream>
#include <iomanip>

// Function to calculate x to the power n
double power(double x, int n)
{
    double result {1.0};
    if(n >= 0)
        for(int i {} ; i < n ; ++i)
            result *= x;
    else
        for(int i {} ; i < -n ; ++i)
            result /= x;
    return result;
}

int main()
{
    // Calculate powers of 8 from -3 to +3
    for(int i {-3} ; i <= 3 ; ++i)
        std::cout << std::setw(10) << power(8.0, i);

    std::cout << std::endl;
}
```

This program produces the following output:

0.00195313	0.015625	0.125	1	8	64	512

All the action occurs in the `for` loop in `main()`. The `power()` function is called seven times. The first argument is 8.0 on each occasion, but the second argument has successive values of `i`, from -3 to +3. Thus, seven values are outputs that correspond to 8^{-3} , 8^{-2} , 8^{-1} , 8^0 , 8^1 , 8^2 , and 8^3 .

Return Values

A function with a return type other than `void` *must* return a value of the type specified in the function header. The return value is calculated within the body of the function and is returned by a `return` statement, which ends the function and execution continues from the calling point. There can be several `return` statements in the body of a function with each potentially returning a different value. The fact that a function can return only a single value might appear to be a limitation, but this isn't the case. The single value that is returned can be a *pointer* to anything you like: an array, or a container or even a container with elements that are containers.

How the `return` Statement Works

The `return` statement in Program 8.1 returns the value of `result` to the point where the function was called. `result` is local to the function and ceases to exist when the function finishes executing, so how is it returned? The answer is that a *copy* of the value being returned is made automatically, and this copy is made available to the calling function. The general form of the `return` statement is:

```
return expression;
```

`expression` must evaluate to a value of the type that is specified for the return value in the function header or must be convertible to that type. The expression can be anything, as long it produces a value of the appropriate type. It can include function calls and can even include a call of the function in which it appears, as you'll see later in this chapter.

If the return type is specified as `void`, no expression can appear in a `return` statement. It must be written simply as:

```
return;
```

If the last statement in a function body executes so that the closing brace is reached, this is equivalent to executing a `return` statement with no expression. Of course, in a function with a return type other than `void`, this is an error and the function will not compile. The `main()` function is an exception to this, where reaching the closing brace is equivalent to executing `return 0`.

Function Declarations

`Ex8_01.cpp` works perfectly well as written, but let's try rearranging the code so that the definition of `main()` *precedes* the definition of the `power()` function in the source file. The code in the program file will look like this:

```
// Ex8_02.cpp
// Calculating powers - rearranged
#include <iostream>
#include <iomanip>

int main()
{
    // Calculate powers of 8 from -3 to +3
    for (int i {-3} ; i <= 3 ; ++i)
        std::cout << std::setw(10) << power(8.0, i);

    std::cout << std::endl;
}
```

```
// Function to calculate x to the power n
double power(double x, int n)
{
    double result {1.0};
    if (n >= 0)
        for (int i {} ; i < n ; ++i)
            result *= x;
    else
        for (int i {} ; i < -n ; ++i)
            result /= x;
    return result;
}
```

If you attempt to compile this, you won't succeed. The compiler has a problem because the `power()` function that is called in `main()` is not defined when it is processing `main()`. Of course, you could revert to the original version but in some situations this won't solve the problem. There are two important issues to consider:

1. As you'll see later, a program can consist of several source files. The definition of a function that is called in one source file may be contained in a separate source file.
2. Suppose you have a function `A()` that calls a function `B()`, which in turn calls `A()`. If you put the definition of `A()` first, it won't compile because it calls `B()`; the same problem arises if you define `B()` first because it calls `A()`.

Naturally, there is a solution to these difficulties. You can *declare* a function before you use or define it by means of a *function prototype*.

Function Prototypes

A *function prototype* is a statement that describes a function sufficiently for the compiler to be able to compile calls to it. It defines the function name, its return type, and its parameter list. A function prototype is sometimes referred to as a *function declaration*. A function can only be compiled if the call is preceded by a function declaration in the source file. The definition of a function is also a declaration, which is why you didn't need a function prototype for `power()` in `Ex8_01.cpp`.

You could write the function prototype for the `power()` function as:

```
double power(double x, int n);
```

If you place function prototypes at the beginning of a source file, the compiler is able to compile the code regardless of where the function definitions are. `Ex8_02.cpp` will compile if you insert the prototype for the `power()` function before the definition of `main()`.

The function prototype above is identical to the function header with a semicolon appended. A function prototype is always terminated by a semicolon, but in general, it doesn't have to be *identical* to the function header. You can use different names for the parameters from those used in the function definition (but not different types, of course). For instance:

```
double power(double value, int exponent);
```

This works just as well. The benefit of the names you have chosen here is marginal, but it does illustrate that you can use more explanatory names in the prototype when such names would be too cumbersome to use in the function definition. The compiler only needs to know the *type* each parameter is, so you can omit the parameter names from the prototype, like this:

```
double power(double, int);
```

There is no particular merit in writing function prototypes like this. It is much less informative than the version with parameter names. If both function parameters were of the same type, then a prototype like this would not give any clue as to which parameter was which. I recommend that you always include parameter names in function prototypes.

It's a good idea to get into the habit of always writing a prototype for each function that you use in a source file — with the exception of `main()` of course, which never requires a prototype. Specifying prototypes in the file removes the possibility of compiler errors arising from functions not being sequenced appropriately. It also allows other programmers to get an overview of the functionality of your code.

Most of the examples in the book use functions from the Standard Library, so where are the prototypes for these? The standard headers contain these. A primary use of header files is to collect together the function prototypes for a related group of functions.

Passing Arguments to a Function

It is very important to understand precisely how arguments are passed to a function. This affects how you write functions and ultimately how they operate. There are also a number of pitfalls to be avoided. In general, the function arguments should correspond in type and sequence with the list of parameters in the function definition. You have no latitude so far as the sequence is concerned, but you do have some flexibility in the argument types. If you specify a function argument of a type that doesn't correspond to the parameter type, then the compiler inserts an implicit conversion of the argument to the type of the parameter where possible. The rules for automatic conversions of this kind are the same as those for automatic conversions in an assignment statement. If an automatic conversion is not possible, you'll get an error message from the compiler.

There are two mechanisms by which arguments are passed to functions, *pass-by-value* and *pass-by-reference*. I'll explain the pass-by-value mechanism first.

Pass-by-Value

With the pass-by-value mechanism, the values of variables or constants you specify as arguments are not passed to a function at all. Instead, copies of the arguments are created and these copies are transferred to the function. This is illustrated in Figure 8-2, using the `power()` function again.

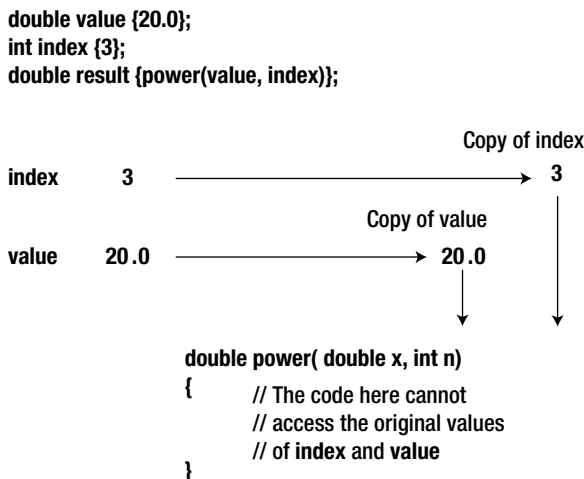


Figure 8-2. The pass-by-value mechanism for arguments to a function

Each time you call the `power()` function, the compiler arranges for copies of the arguments to be stored in a temporary location in the call stack. During execution, all references to the function parameters in the code are mapped to these temporary copies of the arguments. When execution of the function ends, the copies of the arguments are discarded.

I can demonstrate the effects of this with a simple example. This calls a function that attempts to modify one of its arguments and of course, it fails miserably.

```
// Ex8_03.cpp
// Failing to modify the original value of a function argument
#include <iostream>
#include <iomanip>

double change_it(double value_to_be_changed);    // Function prototype

int main()
{
    double it {5.0};
    double result {change_it(it)};

    std::cout << "After function execution, it = " << it
                << "\nResult returned is " << result << std::endl;
}

// Function that attempts to modify an argument and return it
double change_it(double it)
{
    it += 10.0;                                // This modifies the copy
    std::cout << "Within function, it = " << it << std::endl;
    return it;
}
```

This example produces the following output:

```
Within function, it = 15
After function execution, it = 5
Result returned is 15
```

The output shows that adding 10 to `it` in the `change_it()` function has no effect on the variable `it` in `main()`. The `it` variable in `change_it()` is local to the function, and it refers to a copy of whatever argument value is passed when the function is called. Of course, when the value of `it` that is local to `change_it()` is returned, a copy of its current value is made, and it's this copy that's returned to the calling program.

Pass-by-value is the default mechanism by which arguments are passed to a function. It provides a lot of security to the calling function by preventing the function from modifying variables that are owned by the calling function. However, sometimes you do want to modify values in the calling function. Is there a way to do it when you need to? Sure there is: one way is to use a pointer.

Passing a Pointer to a Function

When a function parameter is a pointer type, the pass-by-value mechanism operates just as before. However, a pointer contains the address of another variable; a copy of the pointer contains the same address and therefore points to the same variable.

If you modify the definition of the `change_it()` function to accept an argument of type `double*`, you can pass the address of it as the argument. Of course, you must also change the code in the body of `change_it()` to dereference the pointer parameter. The code is now like this:

```
// Ex8_04.cpp
// Modifying the value of a caller variable
#include <iostream>

double change_it(double* pointer_to_it);           // Function prototype

int main()
{
    double it {5.0};
    double result {change_it(&it);}               // Now we pass the address

    std::cout << "After function execution, it = " << it
               << "\nResult returned is " << result << std::endl;
}

// Function to modify an argument and return it
double change_it(double* pit)
{
    *pit += 10.0;                                // This modifies the original it
    std::cout << "Within function, *pit = " << *pit << std::endl;
    return *pit;
}
```

This version of the program produces the following output:

```
Within function, *pit = 15
After function execution, it = 15
Result returned is 15
```

The way this works is illustrated in Figure 8-3.

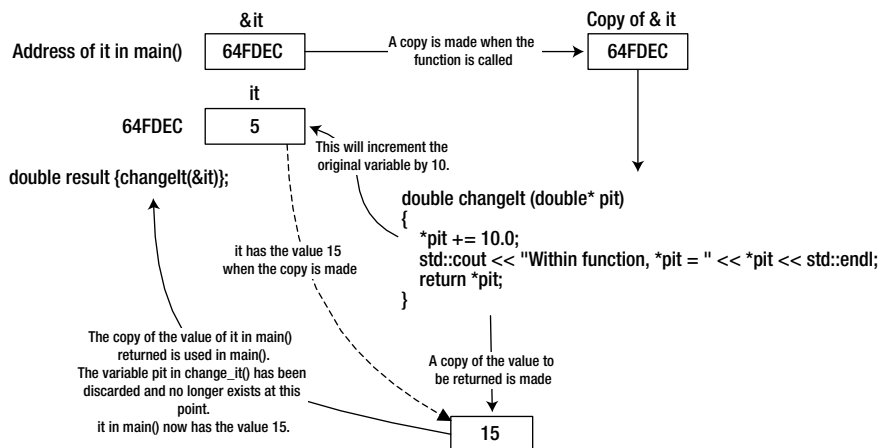


Figure 8-3. *Passing a pointer to a function*

This version of `change_it()` serves only to illustrate how a pointer parameter can allow a variable in the calling function to be modified — it is not a model of how a function should be written. Because you are modifying the value of `it` directly, returning its value is somewhat superfluous.

Passing an Array to a Function

An array name is essentially an address, so you can pass the address of an array to a function just by using its name. The address of the array is copied and passed to the function. This provides several advantages. First, passing the address of an array is a very efficient way of passing an array to a function. Passing all the array elements by value would be very time consuming because every element would be copied. In fact, you can't pass all the elements in an array by value as a single argument because each parameter represents a single item of data. Second, and more significantly, because the function does not deal with the original array variable, but with a copy, the code in the body of the function can treat a parameter that represents an array as a pointer in the fullest sense, including modifying the address that it contains. This means that you can use the power of pointer notation in the body of a function for parameters that are arrays. Before I get to that, let's try the most straight-forward case first — handling an array parameter using array notation. This example includes a function to compute the average of the elements in an array:

```
// Ex8_05.cpp
// Passing an array to a function
#include <iostream>

double average(double array[], size_t count);           // Function prototype

int main()
{
    double values[] {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0};
    std::cout << "Average = " << average(values, (sizeof values)/(sizeof values[0])) << std::endl;
}
```



```
// Function to compute an average
double average(double array[], size_t count)
{
    double sum {};                                // Accumulate total in here
    for (size_t i {} ; i < count ; ++i)
        sum += array[i];                          // Sum array elements
    return sum / count;                            // Return average
}
```

This produces the following very brief output:

```
Average = 5.5
```

The `average()` function works with an array containing any number of double elements. As you can see from the prototype, it accepts two arguments: the array address, and a count of the number of elements. The type of the first parameter is specified as an array of any number of values of type `double`. You can't specify the size of the array between the square brackets. This is because the size of the first dimension of an array is not part of its type. You can pass any one-dimensional array of elements of type `double` as an argument to this function so the second parameter that specifies the number of elements is essential. The function will rely on the correct value for the `count` parameter being supplied by the caller. There's no way to verify that it is correct so the function will quite happily access memory locations outside the array if the value of `count` is greater than the array length. It is up to the caller to ensure that this doesn't happen.

Within the body of `average()`, the computation is expressed in the way you would expect. There's no difference between this and the way you would write the same computation directly in `main()`. The `average()` function is called in `main()` in the output statement. The first argument is the array name, `values`, and the second argument is an expression that evaluates to the number of array elements.

The elements of the array that is passed to `average()` are accessed using normal array notation. I've said that you can also treat an array passed to a function as a pointer and use pointer notation to access the elements. Here's how `average()` would look in that case:

```
double average(double* array, size_t count)
{
    double sum {};                                // Accumulate total in here
    for(size_t i {} ; i < count ; ++i)
        sum += *array++;                          // Sum array elements
    return sum/count;                             // Return average
}
```

const Pointer Parameters

The `average()` function only needs to access values of the array elements, it doesn't need to change them. It would be a good idea to make sure that the code in the function does not inadvertently modify elements of the array. Specifying the parameter type as `const` will do that:

```
double average(const double* array, size_t count)
{
    double sum {};                                // Accumulate total in here
    for(size_t i {} ; i < count ; ++i)
        sum += *array++;                          // Sum array elements
    return sum/count;                             // Return average
}
```

Now the compiler will verify that the elements of the array are not modified in the body of the function. Of course, you must modify the function prototype to reflect the new type for the first parameter; remember that `const` types are quite different from non-`const` types.

Specifying a pointer parameter as `const`, has two consequences: the compiler checks the code in the body of the function to ensure that you don't try to change the value pointed to; and it allows the function to be called with an argument that points to a constant. Passing a non-`const` argument for `const` function parameter will not compile.

■ **Note** There is no purpose in specifying a parameter of a basic type such as `int` or `size_t` as `const`. The pass-by-value mechanism makes a copy of the argument when the function is called, so you can't modify the original value within the function.

Passing a Multidimensional Array to a Function

Passing a multidimensional array to a function is quite straightforward. Suppose you have a two-dimensional array defined as:

```
double beans[2][4] {};
```

The prototype of a hypothetical `yield()` function could look like this:

```
double yield(double beans[][4], size_t count);
```

You could specify the first array dimension explicitly in the type specification for the first parameter, but it is better not to. You have no way to determine the array's first dimension size other than via the second parameter so there's no purpose in specifying it as part of the parameter type. The size of the second array dimension is essential though because `beans` is type `double[][4]`. Any two-dimensional array with a second dimension as 4 can be passed to this function, but an array with a second dimension of 5 for example could not.

If you do specify both array dimensions, how does the compiler know that the first parameter is an array and not a single array element? The answer is simple: you can't specify a parameter as a single array element. Of course you can pass an array element as an argument to a function as long as the parameter type is the same as that of the array element. The array context doesn't apply in this case.

In case you're wondering, you can't circumvent the need for the `count` parameter by using the `sizeof` operator to determine the size of the array. Using `sizeof` on an array parameter name returns the size of the memory location that contains the address of the array. Let's try passing a two-dimensional array to a function in a concrete example:

```
// Ex8_06.cpp
// Passing a two-dimensional array to a function
#include <iostream>

double yield(const double values[][4], size_t n);

int main()
{
    double beans[3][4] {
        { 1.0,  2.0,  3.0,  4.0},
        { 5.0,  6.0,  7.0,  8.0},
        { 9.0, 10.0, 11.0, 12.0}
    };
}
```

```

    std::cout << "Yield = " << yield(beans, sizeof(beans)/sizeof(beans[0]))
              << std::endl;
}

// Function to compute total yield
double yield(const double array[][4], size_t size)
{
    double sum {};
    for(size_t i {} ; i < size ; ++i)           // Loop through rows
    {
        for(size_t j {} ; j < 4 ; ++j)         // Loop through elements in a row
        {
            sum += array[i][j];
        }
    }
    return sum;
}

```

This produces the following output:

```
Yield = 78
```

The first parameter to the `yield()` function is defined as a `const` array of an arbitrary number of rows of four elements of type `double`. When you call the function, the first argument is the `beans` array, and the second argument is the total length of the array in bytes divided by the length of the first row. This evaluates to the number of rows in the array.

Pointer notation doesn't apply particularly well with a multidimensional array. In pointer notation, the statement in the nested `for` loop would be:

```
sum += (*(array+i)+j);
```

I think you'll agree that the computation is clearer in array notation!

Pass-by-Reference

As you know, a *reference* is an alias for another variable. You can specify a function parameter as a reference, in which case the function uses the *pass-by-reference* mechanism with the argument. When the function is called, an argument corresponding to a reference parameter is not copied. The reference parameter is initialized with the argument. Thus it becomes an alias for the argument in the calling program. Wherever the parameter name is used in the body of the function, it accesses the argument value in the calling function directly.

You'll no doubt recall that you specify an lvalue reference type by adding `&` after the type name. To specify a parameter type as "reference to string" for example, you write the type as `string&`. Using a reference parameter improves performance with objects such as type `string`. The pass-by-value mechanism copy the object, which would be time consuming with a long string. With a reference parameter, there is no copying. Calling a function that has a reference parameter is no different from calling a function where the argument is passed by value. A parameter can be an rvalue reference. This has particular significance for functions that belong to a class so I'll defer discussion of rvalue reference parameters until I discuss classes.

References Can Be Risky

A reference parameter enables the function to modify the argument within the calling function. However, the syntax for calling a function that has a reference parameter is no different from calling a function where the argument is passed by value. If you don't have the source code for the function, you have no way to know whether or not a parameter is a reference. This makes it particularly important to use a `const` reference parameter in a function that does not change the argument.

There's a subtle difference in the meaning of `const` between a `const` variable and a `const` parameter. `const` applied to a variable is telling the compiler that the variable is a constant and must not be changed. `const` applied to a reference parameter is about intent; it tells the compiler that the function will not modify the argument so the compiler will make sure that it doesn't. Because the function won't change a `const` reference parameter, the compiler will allow `const` or non-`const` arguments. Only non-`const` arguments can be supplied for a non-`const` reference parameter. Let's investigate the effect of using reference parameters in a new and initially imperfect version of `Ex7_05.cpp` that extracts words from text:

```
// Ex8_07.cpp
// Using a reference parameter
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using std::string;
using std::vector;
void find_words(vector<string>& words, string& str, const string& separators);
void list_words(const vector<string>& words);

int main()
{
    string text;                                     // The string to be searched
    std::cout << "Enter some text terminated by *:\n";
    std::getline(std::cin, text, '*');

    const string separators {" ,;.\"!?'\"\\n"};      // Word delimiters
    vector<string> words;                             // Words found

    find_words(words, text, separators);
    list_words(words);
}

void find_words(vector<string>& words, string& str, const string& separators)
{
    size_t start {str.find_first_not_of(separators)};    // First word start index
    size_t end {};                                       // Index for end of a word

    while (start != string::npos)                       // Find the words
    {
        end = str.find_first_of(separators, start + 1); // Find end of word
        if (end == string::npos)                       // Found a separator?
            end = str.length();                         // No, so set to last + 1
    }
}
```

```

        words.push_back(str.substr(start, end - start));           // Store the word
        start = str.find_first_not_of(separators, end + 1);       // Find 1st character of next word
    }
}

void list_words(const vector<string>& words)
{
    std::cout << "Your string contains the following " << words.size() << " words:\n";
    size_t count {};                                              // Number output
    for (const auto& word : words)
    {
        std::cout << std::setw(15) << word;
        if (!(++count % 5))
            std::cout << std::endl;
    }
    std::cout << std::endl;
}

```

The output is the same as Ex7_05.cpp. Here's a sample:

```

Enter some text terminated by *:
Never judge a man until
you have walked a mile
in his shoes Then who
cares He is a mile
away and you have his
shoes

```

There are now two functions in addition to `main()`, `find_words()` and `list_words()`. The `find_words()` function finds all the words in the string identified by the second argument and stores them in the vector specified by the first argument. The third parameter is a string object containing the word separator characters. The second and third parameters are references and the third parameter is a const reference. If the third parameter was not const, the code would not compile because the third argument in the function call in `main()` is `separators`, which is a const string object. You cannot pass a const object as the argument corresponding to a non-const reference parameter. A const parameter allows a const or non-const argument to be passed to the function. A reference parameter that is not const only accepts arguments that are not const. It would be better to specify the second parameter to `find_words()` as const because the function does not change the argument. The first parameter is a reference, which avoids copying the `vector<string>` object. It cannot be specified as const because the function adds elements to the vector.

The parameter for `list_words` is a const reference because it only accesses the argument, it doesn't change it. Note how the code in both functions is the same as the code that was in `main()` in Ex7_05.cpp. Dividing the program into three functions makes it easier to understand and does not increase the number of lines of code significantly.

Improving the Program

Apart from making the second parameter to `find_words()` a `const` reference, it might improve the program if the calling function did not have to create the vector to hold the words that are extracted. You could define `find_words()` like this:

```
std::shared_ptr<vector<string>> find_words(const string& str, const string& separators)
{
    auto pWords = std::make_shared<vector<string>>();           // Vector of words
    size_t start {str.find_first_not_of(separators)};         // First word start index
    size_t end {};                                             // Index for end of a word

    while (start != string::npos)                               // Find the words
    {
        end = str.find_first_of(separators, start + 1);       // Find end of word
        if (end == string::npos)                               // Found a separator?
            end = str.length();                                 // No, so set to last + 1
        pWords->push_back(str.substr(start, end - start));      // Store the word
        start = str.find_first_not_of(separators, end + 1);    // Find 1st character of next word
    }
    return pWords;
}
```

The function now allocates space for the vector that stores the words on the heap and returns a smart pointer to it when all the words have been found. The only change to the code that finds the words is in the statement that stores each word in the vector. It now uses the `->` operator to call the `push_back()` member of the vector because `pWords` is a pointer. Of course, the `list_words()` function need to be redefined to accept a smart pointer as the argument:

```
void list_words(const std::shared_ptr<vector<string>> pWords)
{
    std::cout << "Your string contains the following " << pWords->size() << " words:\n";
    size_t count {};                                           // Number output
    for (const auto& word : *pWords)
    {
        std::cout << std::setw(15) << word;
        if (!(++count % 5))
            std::cout << std::endl;
    }
    std::cout << std::endl;
}
```

There are minimal changes beyond the parameter specification, which is now a smart pointer to a `const` vector. `size()` is now called using the `->` operator and `pWords` has to be dereferenced in the range-based `for` loop. The code in `main()` to use these functions will be:

```
int main()
{
    string text;                                               // The string to be searched
    std::cout << "Enter some text terminated by *:\n";
    std::getline(std::cin, text, '*');
```

```

const string separators {" ,;.:\\\"'!?'\n"};           // Word delimiters

auto pWords = find_words(text, separators);
list_words(pWords);
}

```

Using the `auto` keywords causes the compiler to figure out the type for `pWords`. If you didn't want to retain the pointer, `pWords`, you could replace the last two statements with this:

```
list_words(find_words(text, separators));
```

This calls `find_words()` in the argument to `list_words()`. `find_words()` finds the words, stores them in a vector that is created on the heap, and returns a smart pointer to the vector. This becomes the argument to the `list_words()` function to output them. The complete code for this version of the program is in the download as `Ex8_07A.cpp`.

Simplifying Code using Type Aliases

You can often make your code easier to follow using *type* aliases that you learned about back in Chapter 3. You could define the following type aliases immediately before the function prototypes in the source file:

```

using Words = vector<string>;           // Type for a vector of words
using PWords = std::shared_ptr<Words>;  // Type for a smart point to a Words object

```

The `PWords` alias is defined using the `Words` alias that precedes it. You can now use `Words` and `PWords` in the code in place of `vector<string>` and `std::shared_ptr<vector<string>>` respectively. The function prototypes can be written as:

```

PWords find_words(const string& str, const string& separators);
void list_words(PWords pWords);

```

That's a lot easier to read, isn't it? A complete version of the program using these type aliases is in the download as `Ex8_07B.cpp`.

References versus Pointers

In most situations, using a reference parameter is preferable to using a pointer. You should specify reference parameters as `const` wherever possible to provide security for the caller arguments and to allow `const` arguments. Of course, when you need to modify an argument corresponding to a reference parameter, you can't specify the parameter as a `const` reference but you should consider whether a pointer might be better. With a pointer, it is always apparent to the caller that the object pointed to can be modified.

An important difference between a pointer and a reference is that a pointer can be `nullptr`, whereas a reference always refers to something — as long as it isn't an alias for a pointer that is `nullptr`, of course. If you want to allow the possibility of a null argument, the only option is a pointer parameter. Of course, because a pointer parameter can be null, you must always test for `nullptr` before using it. Attempting to dereference a null pointer, will cause your program to crash.

Arguments to main()

You can define `main()` so that it accept arguments that are entered on the command line when the program executes. The parameters you can specify for `main()` are standardized: you can either define `main()` with no parameters, or you can define `main()` in the following form:

```
int main(int argc, char* argv[])
{
    // Code for main()...
}
```

The first parameter, `argc`, is a count of the number of string arguments that were found on the command line. It is type `int` for historical reasons, not `size_t` as you might expect because the count cannot be negative. The second parameter, `argv`, is an array of pointers to the command line arguments, including the program name. The array type implies that all command line arguments are received as C-style strings. The program name is always recorded in the first element of `argv`, `argv[0]`. The last element in `argv` (`argv[argc]`) is always `nullptr` so the number of elements in `argv` will be `argc+1`. I'll give you a couple of examples to make this clear. Suppose that to run the program, you enter just the program name on the command line:

Myprog

In this case, `argc` will be 1 and `argv[]` contains two elements: the first is the address of the string "Myprog", and the second will be `nullptr`.

Suppose you enter this:

Myprog 2 3.5 "Rip Van Winkle"

Now `argc` will be 4 and `argv` will have five elements. The first four elements will be pointers to the strings "Myprog.exe", "2", "3.5", and "Rip Van Winkle". The fifth element, `argv[4]`, will be `nullptr`.

What you do with the command line arguments is entirely up to you. The following program shows how you access the command line arguments:

```
// Ex8_08.cpp
// Program that lists its command line arguments
#include <iostream>

int main(int argc, char* argv[])
{
    for (int i {} ; i < argc ; ++i)
        std::cout << argv[i] << std::endl;
}
```

This lists the command line arguments, including the program name. Command line arguments can be anything at all — filenames to a file copy program, for example, or the name of a person to search for in a contact file — anything that is useful to have entered when program execution is initiated.

Default Argument Values

There are many situations in which it would be useful to have default argument values for one or more function parameters. This would allow you to specify an argument value only when you want something different from the default. A simple example is a function that outputs a standard error message. Most of the time, a default message will suffice, but occasionally an alternative is needed. You can do this by specifying a default parameter value in the function prototype. You could define a function to output a message like this:

```
void show_error(const string& message)
{
    std::cout << message << std::endl;
}
```

You specify the default argument value in the function prototype, not in the function definition, like this:

```
void show_error(const string& message = "Program Error");
```

This parameter happens to be a reference. You specify default values for reference and non-reference parameters in exactly the same way. To output the default message, you call the function without an argument:

```
show_error();                                // Outputs "Program Error"
```

To output a particular message, you specify the argument:

```
show_error("Nothing works!");
```

Specifying default parameter values can make functions simpler to use, and you aren't limited to just parameter with a default value.

Multiple Default Parameter Values

All function parameters that have default values must be placed together at the end of the parameter list. When an argument is omitted in a function call, all subsequent arguments in the list must also be omitted. Thus parameters with default values should be sequenced from the least likely to be omitted, to the most likely at the end. These rules are necessary for the compiler to be able to process function calls.

Let's contrive an example of a function with several default parameter values. Suppose that you wrote a function to display one or more data values, several to a line, as follows:

```
void show_data(const int data[], size_t count, const std::string& title,
               size_t width, size_t perline)
{
    std::cout << title << std::endl;                // Display the title

    // Output the data values
    for (size_t i {} ; i < count ; ++i)
    {
        std::cout << std::setw(width) << data[i];    // Display a data item
        if ((i+1) % perline == 0)                    // Newline after perline values
            std::cout << std::endl;
    }
    std::cout << std::endl;
}
```

The data parameter is an array of values to be displayed, and count indicates how many there are. The third parameter of type `const string&` specifies a title that is to head the output. The fourth parameter determines the field width for each item, and the last parameter is the number of data items per line. This function has a lot of parameters. It's clearly a job for default parameter values! Here's an example:

```
// Ex8_09.cpp
// Using multiple default parameter values
#include <iostream>
#include <iomanip>
#include <string>
using std::string;

// The function prototype including defaults for parameters
void show_data(const int data[], size_t count = 1, const string& title = "Data Values",
               size_t width = 10, size_t perLine = 5);

int main()
{
    int samples[] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

    int dataItem {-99};
    show_data(&dataItem);

    dataItem = 13;
    show_data(&dataItem, 1, "Unlucky for some!");

    show_data(samples, sizeof (samples)/sizeof (samples[0]));
    show_data(samples, sizeof (samples)/sizeof (samples[0]), "Samples");
    show_data(samples, sizeof (samples)/sizeof (samples[0]), "Samples", 6);
    show_data(samples, sizeof (samples)/sizeof (samples[0]), "Samples", 8, 4);
}

void show_data(const int data[], size_t count, const std::string& title,
               size_t width, size_t perLine)
{
    std::cout << title << std::endl;           // Display the title

    // Output the data values
    for (size_t i {}; i < count ; ++i)
    {
        std::cout << std::setw(width) << data[i];    // Display a data item
        if ((i+1) % perLine == 0)                    // Newline after perline values
            std::cout << std::endl;
    }
    std::cout << std::endl;
}
```

Here's the output:

```
Data Values
-99
Unlucky for some!
13
Data Values
  1      2      3      4      5
  6      7      8      9     10
11     12
Samples
  1      2      3      4      5
  6      7      8      9     10
11     12
Samples
  1      2      3      4      5
  6      7      8      9     10
11     12
Samples
  1      2      3      4
  5      6      7      8
  9     10     11     12
```

The prototype for `show_data()` specifies default values for all parameters except the first. You have five ways to call this function: you can specify all five arguments, you can omit the last one, or the last two, or the last three, or the last four. You can supply just the first to output a single data item, as long as you are happy with the default values for the remaining parameters.

Remember that you can only omit arguments at the end of the list; you are not allowed to omit the second and the fifth for example:

```
show_data(samples, , "Samples", 15);           // Wrong!
```

Returning Values from a Function

As you know, you can return a value of any type from a function. This is quite straightforward when you're returning a value of one of the basic types, but there are some pitfalls when you are returning a pointer.

Returning a Pointer

When you return a pointer from a function, it must contain either `nullptr`, or an address that is still valid in the calling function. In other words, the variable pointed to must still be in scope after the return to the calling function. This implies the following absolute rule:

Golden Rule *Never* return the address of an automatic local variable from a function.

Suppose you define a function that returns the address of the larger of two argument values. This could be used on the left of an assignment, so that you could change the variable that contains the larger value, perhaps in a statement such as this:

```
*larger(value1, value2) = 100;           // Set the larger variable to 100
```

You can easily be led astray when implementing this. Here's an implementation that doesn't work:

```
int* larger(int a, int b)
{
    if(a > b)
        return &a;           // Wrong!
    else
        return &b;           // Wrong!
}
```

It's easy to see what's wrong with this: *a* and *b* are local to the function. The argument values are copied to the local variables *a* and *b*. When you return *&a* or *&b*, the variables at these addresses no longer exist back in the calling function. You should get a warning from your compiler when you compile this code.

You can specify the parameters as pointers:

```
int* larger(int* a, int* b)
{
    if(*a > *b)
        return a;           // OK
    else
        return b;           // OK
}
```

You could call the function with this statement:

```
*larger(&value1, &value2) = 100;       // Set the larger variable to 100
```

A function to return the address of the larger of two values is not particularly useful, but let's consider something more practical. Suppose we need a program to normalize a set of values of type *double* so that they all lie between 0.0 and 1.0 inclusive. To normalize the values, we can first subtract the minimum sample value from them to make them all non-negative. Two functions will help with that, one to find the minimum and another to adjust the values by any given amount. Here's a definition for the first function:

```
const double* smallest(const double data[], size_t count)
{
    size_t index_min {};
    for (size_t i {1} ; i < count ; ++i)
        if (data[index_min] > data[i])
            index_min = i;

    return &data[index_min];
}
```

You shouldn't have any trouble seeing what's going on here. The index of the minimum value is stored in `index_min`, which is initialized arbitrarily to refer to the first array element. The loop compares the value of the element at `index_min` with each of the others, and when one is less, its index is recorded in `index_min`. The function returns the address of the minimum value in the array. It probably would be more sensible to return the index but I'm demonstrating pointer return values among other things. The first parameter is `const` because the function doesn't change the array. With this parameter `const` you must specify the return type as `const`. The compiler will not allow you to return a non-`const` pointer to an element of a `const` array.

A function to adjust the values of array elements by a given amount looks like this:

```
double* shift_range(double data[], size_t count, double delta)
{
    for (size_t i {} ; i < count ; ++i)
        data[i] += delta;
    return data;
}
```

This function adds the value of the third argument to each array element. The return type could be `void` so it returns nothing, but returning the address of data allows the function to be used as an argument to another function that accepts an array. Of course, the function can still be called without storing or otherwise using the return value.

You could combine using this with the previous function to adjust the values in an array, `samples`, so that all the elements are non-negative:

```
const size_t count {sizeof(samples)/sizeof(samples[0])}; // Element count
shift_range(samples, count, -*smallest(samples, count)); // Subtract min from elements
```

The third argument to `shift_range()` calls `smallest()` which returns a pointer to the minimum element. The expression negates the value, so `shift_range()` will subtract the minimum from each element to achieve what we want. The elements in `data` are now from zero to some positive upper limit. To map these into the range from 0 to 1, we need to divide each element by the maximum element. We first need a function to find the maximum:

```
const double* largest(const double data[], size_t count)
{
    size_t index_max {};
    for (size_t i {1} ; i < count ; ++i)
        if (data[index_max] < data[i])
            index_max = i;
    return &data[index_max];
}
```

This works in essentially the same way as `smallest()`. We could use a function that scales the array elements by dividing by a given value:

```
double* scale_range(double data[], size_t count, double divisor)
{
    if(!divisor) return data; // Do nothing for a zero divisor

    for (size_t i {} ; i < count ; ++i)
        data[i] /= divisor;
    return data;
}
```

Dividing by zero would be a disaster so when the third argument is zero, the function just returns the original array. We can use this function in combination with `largest()` to scale the elements that are now from 0 to some maximum to the range 0 to 1:

```
scale_range(data, count, *largest(data, count));
```

Of course, what the user would probably prefer is a function that will normalize an array of values, thus avoiding the need to get into the gory details:

```
double[] normalize_range(double data[], size_t count)
{
    return scale_range(shift_range(data, count, -(*smallest(data, count))),
                       count, *largest(data, count));
}
```

Remarkably this function only requires one statement. Let's see if it all works in practice:

```
// Ex8_10.cpp
// Returning a pointer
#include <iostream>
#include <iomanip>
#include <string>
using std::string;

void show_data(const double data[], size_t count = 1, const string& title = "Data Values",
               size_t width = 10, size_t perLine = 5);

const double* largest(const double data[], size_t count);
const double* smallest(const double data[], size_t count);
double* shift_range(double data[], size_t count, double delta);
double* scale_range(double data[], size_t count, double divisor);
double* normalize_range(double data[], size_t count);

int main()
{
    double samples[] {
        11.0, 23.0, 13.0, 4.0,
        57.0, 36.0, 317.0, 88.0,
        9.0, 100.0, 121.0, 12.0
    };

    const size_t count{sizeof (samples) / sizeof (samples[0])}; // Number of samples
    show_data(samples, count, "Original Values");                // Output original values
    normalize_range(samples, count);                             // Normalize the values
    show_data(samples, count, "Normalized Values", 12);          // Output normalized values
}

// Finds the largest of an array of double values
const double* largest(const double data[], size_t count)
{
    size_t index_max {};
    for (size_t i {1}; i < count; ++i)
        if (data[index_max] < data[i])
```

```

        index_max = i;
    return &data[index_max];
}

// Finds the smallest of an array of double values
const double* smallest(const double data[], size_t count)
{
    size_t index_min{};
    for (size_t i {1} ; i < count ; ++i)
        if (data[index_min] > data[i])
            index_min = i;

    return &data[index_min];
}

// Modify a range of value by delta
double* shift_range(double data[], size_t count, double delta)
{
    for (size_t i {} ; i < count ; ++i)
        data[i] += delta;
    return data;
}

// Scale an array of values by divisor
double* scale_range(double data[], size_t count, double divisor)
{
    if (!divisor) return data; // Do nothing for a zero divisor

    for (size_t i {} ; i < count ; ++i)
        data[i] /=divisor;
    return data;
}

// Normalize an array of values to the range 0 to 1
double* normalize_range(double data[], size_t count)
{
    return scale_range(shift_range(data, count, -(*smallest(data, count))),
        count, *largest(data, count));
}

// Outputs an array of double values
void show_data(const double data[], size_t count, const string& title, size_t width, size_t perline)
{
    std::cout << title << std::endl; // Display the title

    // Output the data values
    for (size_t i {} ; i < count ; ++i)
    {
        std::cout << std::setw(width) << data[i]; // Display a data item
        if ((i + 1) % perline == 0) // Newline after perline values

```

```

        std::cout << std::endl;
    }
    std::cout << std::endl;
}

```

I got the following output:

Original Values				
11	23	13	4	57
36	317	88	9	100
121	12			
Normalized Values				
0.0223642	0.0607029	0.028754	0	0.169329
0.102236	1	0.268371	0.0159744	0.306709
0.373802	0.0255591			

The output demonstrates that the results are what was required. The last two statements in `main()` could be condensed into one by passing the address returned by `normalize_range()` as the first argument to `show_data()`:

```
show_data(normalize_range(samples, count), count, "Normalized Values", 12);
```

This is more concise, but not necessarily clearer.

Returning a Reference

Returning a pointer from a function is useful, but it can be problematic. Pointers can be null, and attempting to dereference nullptr pointer results in the failure of your program. The solution, as you will surely have guessed from the title of this section, is to return a *reference*. A reference is an alias for another variable so I can state an absolute rule for references:

■ **Golden Rule** *Never* return a reference to an automatic local variable in a function.

By returning an lvalue reference, you allow a function call to the function to be used on the left of an assignment. In fact, returning an lvalue reference from a function is the only way you can enable a function to be used (without dereferencing) on the left of an assignment operation.

Suppose you code a `larger()` function like this:

```

string& larger(string& s1, string& s2)
{
    return s1 > s2 ? s1 : s2;           // Return a reference to the larger string
}

```

The return type is “reference to string” and the parameters are non-const references. Because you want to return a non-const reference to one or other of the arguments, you must not specify the parameters as `const`.

You could use the function to change the larger of the two arguments, like this:

```

string str1 {"abcx"};
string str2 {"adcf"};
larger(str1, str2) = "defg";

```


Because the parameters are not `const`, you can't use string literals as arguments; the compiler won't allow it. A reference parameter permits the value to be changed, and changing a constant is not something the compiler will knowingly go along with. If you make the parameters `const`, you can't use a non-`const` reference as the return type.

You're not going to examine an extended example of using reference return types at this moment, but you can be sure that you'll meet them again before long. As you'll discover, reference return types become essential when you are creating your own data types using classes.

Inline Functions

With functions that are very short, the overhead of the code the compiler generates to deal with passing arguments and returning a result is significant compared to the code involved in doing the actual calculation. The execution times of the two types of code may be similarly related. In extreme cases, the code for calling the function may occupy more memory than the code in the body of the function. In such circumstances, you can suggest to the compiler that it replace a function call with the code from the body of the function, suitably adjusted to deal with local names. This could make the program shorter, faster, or possibly both.

You do this using the `inline` keyword in the function definition. For example:

```
inline int larger(int m, int n)
{
    return m > n ? m : n;
}
```

This definition indicates that the compiler can replace calls with inline code. However, it is only a suggestion, and it's down to the compiler as to whether your suggestion is taken up. When a function is specified as `inline`, the definition must be available in every source file that calls the function. For this reason, the definition of an inline function usually appears in a header file rather than in a source file, and the header is included in each source file that uses the function. Most if not all modern compilers will make short functions `inline`, even when you don't use the `inline` keyword in the definition. If a function you specify as `inline` is used in more than one source file, you should place the definition in a header file that you include in each source file that uses the function. If you don't, you'll get "unresolved external" messages when the code is linked.

Static Variables

In the functions you have seen so far, nothing is retained within the body of the function from one execution to the next. Suppose you want to count how many times a function has been called. How can you do that? One way is to define a variable at file scope and increment it from within the function. A potential problem with this is that *any* function in the file can modify the variable, so you can't be sure that it's only being incremented when it should be.

A better solution is to define a variable in the function body as `static`. A static variable that you define within a function is created the first time its definition is executed. It then continues to exist until the program terminates. This means that you can carry over a value from one call of a function to the next. To specify a variable as static, you prefix the type name in the definition with the `static` keyword. Here's an example:

```
static int count {1};
```

The first time this statement executes, `count` is created and initialized to 1. Subsequent executions of the statement have no further effect. `count` continues to exist until the program terminates. If you don't initialize a static variable, it will be initialized to 0 by default.

Let's consider a very simple example:

```
void nextInteger()
{
    static int count {1};
    std::cout << count++ << std::endl;
}
```

This function increments the static variable `count` after outputting its current value. The first time the function is called, it outputs 1. The second time, it outputs 2. Each time the function is called, it displays an integer that is one larger than the previous value. `count` is created and initialized only once, the first time the function is called. Subsequent calls output the current value of `count` and increment it. `count` survives for as long as the program is executing.

You can specify any type of variable as static, and you can use a static variable for anything that you want to remember from one function call to the next. You might want to hold on to the number of the previous file record that was read for example, or the highest value of previous arguments.

Here is an example that demonstrates using a static variable in generating the Fibonacci sequence. This is a sequence of integers in which each number is the sum of the two that precede it. Here's the code:

```
// Ex8_11.cpp
// Using static variables
#include <iostream>
#include <iomanip>

long next_Fibonacci();

int main()
{
    size_t count {};
    std::cout << "Enter the number of Fibonacci values to be generated: ";
    std::cin >> count;
    std::cout << "The Fibonacci Series:\n";
    for (size_t i {}; i < count ; ++i)
    {
        std::cout << std::setw(10) << next_Fibonacci();
        if (!((i + 1) % 8))                // After every 8th output...
            std::cout << std::endl;        // ...start a new line
    }
    std::cout << std::endl;
}

// Generate the next number in the Fibonacci series
long next_Fibonacci()
{
    static long last;                      // Last number in sequence
    static long last_but_one {1L};        // Last but one in sequence

    long next {last + last_but_one};      // Next is sum of the last two
    last_but_one = last;                  // Update last but one
    last = next;                          // Last is new one
    return last;                          // Return the new value
}
```

This produces the following output:

```
Enter the number of Fibonacci values to be generated: 30
```

```
The Fibonacci Series:
```

1	1	2	3	5	8	13	21
34	55	89	144	233	377	610	987
1597	2584	4181	6765	10946	17711	28657	46368
75025	121393	196418	317811	514229	832040		

The `main()` function calls the `nextFibonacci()` function count times in a loop. No arguments are passed to `nextFibonacci()`; the values returned are generated inside the function. Two static variables defined in the function hold the most recent generated number in the sequence and its predecessor.

Judiciously initializing `last` to 0 by default and `last_but_one` to 1, makes the sequence begin with two 1s. At each call of `nextFibonacci()`, the next number is calculated by summing the previous two numbers. The result is stored in the automatic variable `next`. You can do this because `last` and `last_but_one` are static variables, which retain the values assigned to them in the previous call of `nextFibonacci()`. Before returning `next`, you transfer the previous value in `last` to `last_but_one`, and the new value to `last`.

Although static variables survive as long as the program does, they are only *accessible* within the block in which they are defined, so `last` and `last_but_one` are only accessible from within the body of the `nextFibonacci()` function.

Function Overloading

You'll often find that you need two or more functions that do essentially the same thing, but with parameters of different types. The `largest()` and `smallest()` functions in `Ex8_10.cpp` are likely candidates. You would want these operations to work with arrays of different types such as `int[]`, `double[]`, `float[]` or even `string[]`. Ideally, all such functions would have the same name, `smallest()` or `largest()`. Function overloading makes that possible.

Function overloading allows several functions in a program with the same name as long as they each have a parameter list that is different from the others. You learned earlier in this chapter that the compiler identifies a function by its *signature*, which is a combination of the function name and the parameter list. Overloaded functions have the same name so the signature of each overloaded function must be differentiated by the parameter list alone. That allows the compiler to select the correct function for each function call based on the argument list. Two functions with the same name are different if at least one of the following is true:

- The functions have different numbers of parameters.
- At least one pair of corresponding parameters are of different types.

A program that has two or more functions with the same signature will not compile. Here's an example that uses overloaded versions of the `largest()` function:

```
// Ex8_12.cpp
// Overloading a function
#include <iostream>
#include <string>
#include <vector>
using std::string;
using std::vector;
```

```

// Function prototypes
double largest(const double data[], size_t count);
double largest(const vector<double>& data);
int largest(const vector<int>& data);
string largest(const vector<string>& words);

int main()
{
    double values[] {1.5, 44.6, 13.7, 21.2, 6.7};
    vector<int> numbers {15, 44, 13, 21, 6, 8, 5, 2};
    vector<double> data {3.5, 5, 6, -1.2, 8.7, 6.4};
    vector<string> names {"Charles Dickens", "Emily Bronte", "Jane Austen",
                        "Henry James", "Arthur Miller"};

    std::cout << "The largest of values is "
               << largest(values, sizeof(values)/sizeof(values[0])) << std::endl;
    std::cout << "The largest of numbers is " << largest(numbers) << std::endl;
    std::cout << "The largest of data is " << largest(data) << std::endl;
    std::cout << "The largest of names is " << largest(names) << std::endl;
}

// Finds the largest of an array of double values
double largest(const double data[], size_t count)
{
    size_t index_max {};
    for (size_t i {1} ; i < count ; ++i)
        if (data[index_max] < data[i])
            index_max = i;
    return data[index_max];
}

// Finds the largest of a vector of double values
double largest(const vector<double>& data)
{
    double max {data[0]};
    for (auto value : data)
        if (max < value) max = value;

    return max;
}

// Finds the largest of a vector of int values
int largest(const vector<int>& data)
{
    int max {data[0]};
    for (auto value : data)
        if (max < value) max = value;

    return max;
}

```

```
// Finds the largest of a vector of string objects
string largest(const vector<string>& words)
{
    string max_word {words[0]};
    for (auto& word : words)
        if (max_word < word) max_word = word;

    return max_word;
}
```

This produces the following output:

```
The largest of values is 44.6
The largest of numbers is 44
The largest of data is 8.7
The largest of names is Jane Austen
```

The compiler selects the version of `largest()` to be called in `main()` based on the argument list. Each version of the function has a unique signature because the parameter lists are different. This example illustrates quite nicely how much easier it is to use a `vector<T>` than a standard array. It's important to note that the parameters that accept `vector<T>` arguments are references. If they are not specified as references the vector object will be passed by value and thus copied. This could be expensive for a vector with a lot of elements. Parameters of array types are different. Only the address of an array is passed in this case so they do not need to be reference types.

Overloading and Pointer Parameters

Pointers to different types are different, so the following prototypes declare different overloaded functions:

```
int largest(int* pValues, size_t count);           // Prototype 1
int largest(float* pValues, size_t count);         // Prototype 2
```

Note that a parameter of type `int*` is treated in the same way as a parameter type of `int[]`. Hence the following prototype declares the same function as Prototype 1 above:

```
int largest(int values[], int count);              // Identical signature to prototype 1
```

With either parameter type, the argument is an address and therefore not differentiated. You can implement the function using array notation or pointer notation with either parameter type.

If you pass `nullptr` explicitly as the first argument to the overloaded `largest()` function, the compiler cannot determine which of the two functions to call. If you want to allow `nullptr` as an explicit argument value, you must add a third function overload where the first parameter is of type `std::nullptr_t`. This function will be called whenever the argument is specified as `nullptr`. Of course, if `largest()` is called with an argument that is a pointer variable, `p`, of type `int*`, the first version of the function will be called, even if `p` contains `nullptr`.

Overloading and Reference Parameters

You need to be careful when you are overloading functions with reference parameters. You can't overload a function with a parameter type `data_type` with a function that has a parameter type `data_type&`. The compiler cannot determine which function you want from the argument. These prototypes illustrate the problem:

```
void do_it(string number);                          // These are not distinguishable...
void do_it(string& number);                         // ...from the argument type
```

Suppose you write the following statements:

```
string word {"egg"};
do_it(word);                                // Calls which???
```

The second statement could call either function. The compiler cannot determine which version of `do_it()` should be called. Thus you can't distinguish overloaded functions based on a parameter for one version being of a given type, and the other being a reference to that type.

You should also be wary when you have overloaded a function where one version has a parameter of type `type1&` and another with a parameter *reference to type2*&. The function called depends on the sort of arguments you use, but you may get some surprising results. Let's explore this a little with an example:

```
// Ex8_13.cpp
// Overloading a function with reference parameters
#include <iostream>

double larger(double a, double b);           // Non-reference parameters
long& larger(long& a, long& b);              // lvalue reference parameters

int main()
{
    double a_double {1.5}, b_double {2.5};
    std::cout << "The larger of double values "
                << a_double << " and " << b_double << " is "
                << larger(a_double, b_double) << std::endl;

    int a_int {15}, b_int {25};
    std::cout << "The larger of int values "
                << a_int << " and " << b_int << " is "
                << larger(static_cast<long>(a_int), static_cast<long>(b_int))
                << std::endl;
}

// Returns the larger of two floating point values
double larger(double a, double b)
{
    std::cout << "double larger() called." << std::endl;
    return a > b ? a : b;
}

// Returns the larger of two long references
long& larger(long& a, long& b)
{
    std::cout << "long ref larger() called" << std::endl;
    return a > b ? a : b;
}
```

This produces the following output:

```
double larger() called.
The larger of double values 1.5 and 2.5 is 2.5
double larger() called.
The larger of int values 15 and 25 is 25
```

The third line of output may not be what you were anticipating. You might expect the second output statement in `main()` to call the version of `larger()` with `long&` parameters. This statement has called the version with double parameters — but why? After all, you *did* cast both arguments to `long`.

That is exactly where the problem lies. The arguments are not `a_int` and `b_int`, but temporary locations that contain the same values after conversion to type `long`. The compiler will not use a temporary address, an rvalue, to initialize a reference — it's just too risky. The code in `larger()` has free rein on what it does with the parameters, so either parameter could be modified and/or returned. Allowing the use of a temporary location in this way is not sensible, so the compiler won't do it.

What *can* you do about this? You have a couple of choices. If `a_int` and `b_int` were type `long` the compiler will call the version of `larger()` with parameters of type `long&`. If the variables can't be type `long` you could specify the parameters as `const` references like this:

```
long larger(const long& a, const long& b);
```

Clearly you must change the function prototype too. The function works with either `const` or non-`const` arguments. The compiler knows that the function won't modify the arguments so it will call this version for arguments that are rvalues instead of the version with double parameters. Note that you return type `long` now. If you insist on returning a reference, the return type must be `const` because the compiler cannot convert from a `const` reference to a non-`const` reference. A `const` reference is never an lvalue, so you can't use it on the left of an assignment. Thus, you have nothing to lose by returning a value of type `long` in this instance.

Overloading and const Parameters

A `const` parameter is only distinguished from a non-`const` parameter for references and pointers. For a fundamental type such as `int` for example, `const int` is identical to `int`. Hence, the following prototypes are not distinguishable:

```
long& larger(long a, long b);
long& larger(const long a, const long b);
```

The compiler ignores the `const` attribute of the parameters in the second declaration. This is because the arguments are passed *by value*, meaning that a *copy* of each argument is passed into the function, and thus the original is protected from modification by the function. There is no point to specifying parameters as `const` when the arguments are passed by value.

Overloading with const Pointer Parameters

Overloaded functions are different if one has a parameter of type `type*` and the other has a parameter of `const type*`. The parameters are pointers to different things — so they are different types. For example, these prototypes have different function signatures:

```
long* larger(long* a, long* b);           // Pointer parameters
const long* larger(const long* a, const long* b); // Pointer to const parameter
```

Applying the `const` modifier to a pointer prevents the value at the address from being modified. Without the `const` modifier, the value can be modified through the pointer; the pass-by-value mechanism does not inhibit this in any way. In this example, the first function above is called with these statements:

```
long num1 {1L};
long num2 {2L};
long num3 {*larger(&num1, &num2)}; // Calls larger() that has non-const parameters
```

The latter version of `larger()` with `const` parameters is called by the following code:

```
const long num4 {1L};
const long num5 {2L};
const long num6 {*larger(&num10, &num20)}; // Calls larger() that has const parameters
```

The compiler won't pass a `const` value to a function in which the parameter is a non-`const` pointer. Allowing a `const` value to be passed through a non-`const` pointer would violate the `const`-ness of the variable. Thus, the compiler selects the version of `larger()` for this case with `const` pointer parameters in this case.

In contrast to the previous example, two overloaded functions are *the same* if one of them has a parameter of type “*pointer to type*” and the other has a parameter “*const pointer to type*”. For example:

```
long* larger(long* a, long* b);           // These are...
long* const larger(long* const a, long* const b); // ...identical
```

These two functions are not differentiated and won't compile. The reason is clear when you consider that the first prototype has a parameter of type “*pointer to long*”, and the second has parameter of type “*const pointer to long*”. If you think of “*pointer to long*” as type `T`, then the parameter types are `T` and `const T`—which are not differentiated.

Overloading and const Reference Parameters

Reference parameters are more straightforward when it comes to `const`. Type `T&` and type `const T&` are always differentiated, so type `const int&` is always different from type `int&` for example. This means that you can overload functions in the manner implied by these prototypes:

```
long& larger(long& a, long& b);
long larger(const long& a, const long& b);
```

Each function will have the same function body, which returns the larger of the two arguments, but the functions behave differently. The first prototype declares a function that doesn't accept constants as arguments, but you can use the function on the left of an assignment to modify one or the other of the reference parameters. The second prototype declares a function that accepts constants and non-constants as arguments, but the return type is not an lvalue so you can't use the function on the left of an assignment.

Overloading and Default Argument Values

You know that you can specify default parameter values for a function. However, default parameter values for overloaded functions can sometimes affect the compiler's ability to distinguish one call from another. For example, suppose you have two versions of a `show_error()` function that outputs an error message. Here's a version that has a C-style string parameter:

```
void show_error(const char* message)
{
    std::cout << message << std::endl;
}
```

This version accepts a string argument:

```
void show_error(const string& message)
{
    std::cout << message << std::endl;
}
```


You cannot specify a default argument for both functions because it would create an ambiguity. The statement to output the default message in either case would be:

```
show_error();
```

The compiler has no way of knowing which function is required. Of course, this is a silly example: you have no reason to specify defaults for both functions. A default for just one does everything that you need. However, circumstances can arise where it is not so silly, and overall, you must ensure that all function calls uniquely identify the function that should be called.

A Sausage Machine for Functions

Overloaded functions sometimes contain exactly the same code. The only difference is in the parameter list. It seems an unnecessary overhead to have to write the same code over and over, and indeed it is. In such situations, you can write the code just once, as a *function template*.

A *function template* is a blueprint or a recipe for defining a family of functions; it is not a definition of a function. The compiler uses a function template to generate a function definition when necessary. If it is never necessary, no code results from the template. A function definition that is generated from a template is an *instance* or an instantiation of the template. A function template is a parametric function definition, where a particular function instance is created by one or more parameter values. The parameter values are usually data types, where a function definition can be generated for a parameter value of type `int` for example, and another with a parameter value of type `string`. Parameters are not necessarily types. They can be other things such as a dimension for example. Let's consider a specific example.

The `larger()` function is a good candidate for a template. A template for this function is shown in Figure 8-4.

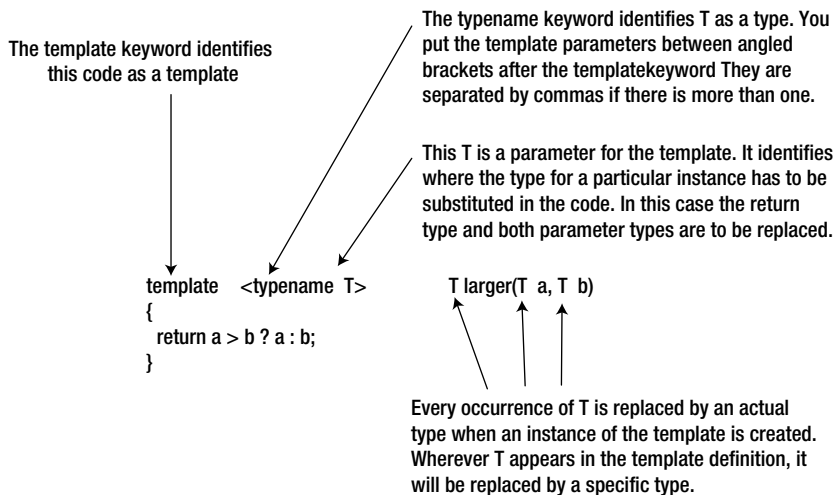


Figure 8-4. A simple function template

The function template starts with the `template` keyword to identify it as such. This is followed by a pair of angled brackets that contains a list of one or more template parameters. In this case, there's one, the parameter `T`. `T` is commonly used as a name for a parameter because most parameters are types but you can use whatever name you like for a parameter; names such as `replace_it` or `my_type` are equally valid.

The `typename` is a keyword that identifies that `T` is a type. You can also use the keyword `class` here, but I prefer `typename` because the type argument can be a fundamental type, not just a class type.

The rest of the definition is similar to a normal function except that the parameter `T` is sprinkled around. The compiler creates an instance of the template by replacing `T` throughout the definition with a specific type.

You can position the template in a source file in the same way as a normal function definition; you can also specify a prototype for a function template. In this case, it would be:

```
template <typename T> T larger(T a, T b);    // Prototype for function template
```

Either the prototype or the definition of the template must appear in the source file before any statement that results in an instance of the template.

Creating Instances of a Function Template

The compiler creates instances of the template from any statement that uses the `larger()` function. Here's an example:

```
std::cout << "Larger of 1.5 and 2.5 is " << larger(1.5, 2.5) << std::endl;
```

You just use the function in the normal way. You don't need to specify a value for the template parameter `T` — the compiler deduces the type that is to replace `T` from the arguments in the `larger()` function call. The arguments to `larger()` are literals of type `double`, so this call causes the compiler to search for an existing definition of `larger()` with `double` parameters. If it doesn't find one, the compiler creates this version of `larger()` from the template by substituting `double` for `T` in the template definition.

The resulting function definition accepts arguments of type `double` and returns a `double` value. With `double` plugged into the template in place of `T`, the template instance will effectively be as follows:

```
double larger(double a, double b)
{
    return a > b ? a : b;
}
```

The compiler generates each template instance once. If a subsequent function call requires the same instance, then it calls the instance that exists. Your program only ever includes a single copy of the definition of each instance, even if the same instance is generated in different source files. Now that you are familiar with the concepts, let's road test a function template:

```
// Ex8_14.cpp
// Using a function template
#include <iostream>

template <typename T> T larger(T a, T b);    // Function template prototype

int main()
{
    std::cout << "Larger of 1.5 and 2.5 is " << larger(1.5, 2.5) << std::endl;
    std::cout << "Larger of 3.5 and 4.5 is " << larger(3.5, 4.5) << std::endl;

    int a_int {35}, b_int {45};
    std::cout << "Larger of " << a_int << " and " << b_int << " is "
        << larger(a_int, b_int) << std::endl;
```

```

long a_long {9L}, b_long {8L};
std::cout << "Larger of " << a_long << " and " << b_long << " is "
    << larger(a_long, b_long) << std::endl;
}

// Template for functions to return the larger of two values
template <typename T>
T larger(T a, T b)
{
    return a > b ? a : b;
}

```

This produces the following output:

```

Larger of 1.5 and 2.5 is 2.5
Larger of 3.5 and 4.5 is 4.5
Larger of 35 and 45 is 45
Larger of 9 and 8 is 9

```

The compiler creates a definition of `larger()` that accepts arguments of type `double` as a result of the first statement in `main()`. The same instance will be called in the next statement. The third statement requires a version of `larger()` that accepts an argument of type `int` so a new template instance is created. The last statement results in yet another template instance being created that has parameters of type `long` and returns a value of type `long`.

Note that if you add the following statement to `main()`, it will not compile:

```

std::cout << "Larger of " << a_long << " and 9.5 is "
    << larger(a_long, 9.5) << std::endl;

```

The arguments to `larger()` are of different types whereas the parameters for `larger()` in the template are of the same type. The compiler cannot create a template instance that has different parameter types. Obviously, one argument could be converted to the type of the other but you have to code this explicitly; the compiler won't do it. You could define the template to allow the parameters for `larger()` to be different types, but this adds a complication that I'll discuss later in this chapter.

Explicit Template Argument

You can specify the argument for a template parameter explicitly when you call the function. This allows you to control which version of the function is used. The compiler no longer deduces the type to replace `T`; it accepts what you specify. There are several situations in which this can be useful:

- Where the function call is ambiguous so the compilation fails.
- When the compiler is unable to deduce the template arguments.
- Where the compiler would generate too many instances of the function template.

You can resolve the problem of using different arguments types with `larger()` with an explicit instantiation of the template:

```

std::cout << "Larger of " << a_long << " and 9.5 is "
    << larger<double>(a_long, 9.5) << std::endl;    // Outputs 9.5

```

You put the explicit type argument for the function template between angled brackets after the function name. This generates an instance with `T` as type `double`. When you use explicit template arguments, the compiler has complete faith that you know what you are doing. It will insert an implicit type conversion for the first argument to type `double`. It will provide implicit conversions, even when this may not be what you want. For example:

```
std::cout << "Larger of " << a_long << " and 9.5 is "
          << larger<long>(a_long, 9.5) << std::endl;    // Outputs 9
```

You are telling the compiler to use a template instance with `T` as type `long`. This necessitates an implicit conversion of the second argument to `long`, so the result is 9, which is maybe not what you really want.

The compiler creates three instances of `larger()` from the template in `Ex8_14.cpp`. Each instance that is generated increases the size of the executable so if you can reduce the number of instances, the executable will be smaller. You could compare values of type `int` using the instance with `T` as type `long`. An explicit template instantiation would do it:

```
std::cout << "Larger of " << a_int << " and " << b_int << " is "
          << larger<long>(a_int, b_int) << std::endl;
```

Now there will be only two template instances. Of course, you have added implicit type conversions for the arguments so the gain might be marginal in this case. With a more complex function template, the gain could be very significant.

Function Template Specialization

Suppose that you extended `Ex8_14.cpp` to call `larger()` with arguments that are pointers:

```
std::cout << "Larger of " << a_long << " and " << b_long << " is "
          << *larger(&a_long, &b_long) << std::endl;    // Outputs 8
```

The compiler instantiates the template with the parameter as type `long*`. This prototype of this version is:

```
long* larger(long*, long*);
```

The return value is an address, and you have to dereference it to output the value. However, the result, 8, is incorrect. This is because the comparison is between addresses passed as arguments, not the values at those addresses. This illustrates how easy it is to create hidden errors using templates. You need to be particularly careful when using pointer types as template arguments.

You can define a *specialization* of the template to accommodate a template argument that is a pointer type. For a specific parameter value, or set of values in the case of a template with multiple parameters, a template specialization defines a behavior that is different from the standard template. The definition of a template specialization must come after a declaration or definition of the original template. If you put a specialization first, then the program won't compile. The specialization must also appear before its first use.

The definition of a specialization starts with the `template` keyword but the parameter is omitted, so the angled brackets following the keyword are empty. You must still define the type of argument for the specialization and you place this between angled brackets immediately following the template function name. The definition for a specialization of `larger()` for type `long*` is:

```
template <> long* larger<long*>(long* a, long* b)
{
    return *a > *b ? a : b;
}
```

The only change to the body of the function is to dereference the arguments `a` and `b` so that you compare values rather than addresses. To use this in `Ex8_14.cpp`, the specialization would need to be placed after the prototype for the template, and before `main()`.

Function Templates and Overloading

You can overload a function template by defining other functions with the same name. Thus you can define “overrides” for specific cases, which will always be used by the compiler in preference to a template instance. As always, each overloaded function must have a unique signature.

Let’s reconsider the previous situation in which you need to overload the `larger()` function to take pointer arguments. Instead of using a template specialization for `larger()`, you could define an overloaded function. The following overloaded function prototype would do it:

```
long* larger(long* a, long* b);           // Function overloading the larger template
```

In place of the specialization definition you’d use this function definition:

```
long* larger(long* a, long* b)
{
    return *a > *b ? a : b;
}
```

It’s also possible to overload an existing template with another template. For example, you could define a template that overloads the `larger()` template in `Ex8_14.cpp` to find the largest value contained in an array:

```
template <typename T>
T larger (const T data[], size_t count)
{
    T result {data[0]};
    for(size_t i {1} ; i < count ; ++i)
        if(data[i] > result) result = data[i];

    return result;
}
```

The parameter list differentiates functions produced from this template from instances of the original template. You could define another template overload for vectors:

```
template <typename T>
T larger (const std::vector<T>& data)
{
    T result {data[0]};
    for(auto& value : data)
        if(value > result) result = value;

    return result;
}
```

You could extend Ex8_14.cpp to demonstrate this. Add the templates above to the end of the source file and these prototypes at the beginning:

```
template <typename T> T larger(const T data[], size_t count);
template <typename T> T larger(const std::vector<T>& data);
```

You'll need `#include` directives for the vector and string headers. The code in `main()` can be changed to:

```
int a_int {35}, b_int {45};
std::cout << "Larger of " << a_int << " and " << b_int << " is "
    << larger(a_int, b_int) << std::endl;

const char text[] {"A nod is as good as a wink to a blind horse."};
std::cout << "Largest character in \"" << text << "\" is '"
    << larger(text, sizeof(text)) << "'" << std::endl;

std::vector<std::string> words {"The", "higher", "the", "fewer"};
std::cout << "The largest word in words is \"" << larger(words)
    << "\"" << std::endl;

std::vector<double> data {-1.4, 7.3, -100.0, 54.1, 16.3};
std::cout << "The largest value in data is " << larger(data) << std::endl;
```

The complete example is in the code download as Ex8_15.cpp. This generates instances of all three overloaded templates. If you compile and execute it, the output will be:

```
Larger of 35 and 45 is 45
Largest character in "A nod is as good as a wink to a blind horse." is 'w'
The largest word in words is "the"
The largest value in data is 54.1
```

Function Templates with Multiple Parameters

You've been using function templates with a single parameter, but there can be several parameters. A classic application for a second template type argument is to provide a way of controlling the return type. You could define yet another template for `larger()` that allows the return type to be specified independently of the function parameter type:

```
template <typename TReturn, typename TArg>
TReturn larger(TArg a, TArg b)
{
    return a > b ? a : b;
}
```

The compiler can't deduce the return type, `TReturn`, so you must always specify it. However, the compiler can deduce the type for the arguments, so you can get away with specifying just the return type. Here's an example:

```
std::cout << "Larger of 1.5 and 2.5 is " << larger<int>(1.5, 2.5) << std::endl;
```

The return type is specified as `int` between the angled brackets following the function name. The type for the function parameters is deduced from the arguments to be `double`. The result of the function call is 2. You can specify both `TReturn` and `TArg`:

```
std::cout << "Larger of 1.5 and 2.5 is " << larger<double, double>(1.5, 2.5) << std::endl;
```

The compiler creates the function that accepts arguments of type `double` and returns a result of type `double`. Clearly, the sequence of parameters in the template definition is important here. If you had the return type as the second parameter, you'd always have to specify both parameters in a call: if you specify only one parameter, it would be interpreted as the argument type, leaving the return type undefined.

You can specify default values for function template parameters. For example you could specify `double` as the default return type in the prototype for the template above like this:

```
template <typename TReturn=double, typename TArg> TReturn larger(TArg a, TArg b);
```

If you don't specify any template parameter values, the return type will be `double`.

```
std::cout << "Larger of 15 and 25 is " << larger(15, 25) << std::endl;
```

The `larger()` template instance resulting from this statement accepts arguments of type `int` and returns the result as type `double`. Of course, if the template definition appears at the beginning of the source file, the default template parameter value would appear in that.

Non-Type Template Parameters

All the template parameters you have seen so far have been types. Function templates can also have *non-type* parameters in this case that require non-type arguments. Arguments corresponding to non-type parameters must be either integral compile-time constants, or references to pointers to objects with external linkage.

You include any non-type template parameters in the parameter list along with any other type parameters when you define the template. You'll see an example in a moment. The type of a non-type template parameter can be one of the following:

- An integral type, such as `int` or `long`
- An enumeration type
- A pointer or reference to an object type
- A pointer or a reference to a function
- A pointer to a class member

You haven't met the last two. I'll introduce pointers to functions later in this chapter and I'll discuss references to functions and pointers to class members when I cover classes. The application of non-type template parameters to these types is beyond the scope of this book. You'll only consider an elementary example with parameters of type `int`, just to see how it works.

Suppose you need a function to perform range checking on a value. You could define a template to handle a variety of types:

```
template <typename T, int upper, int lower>
bool is_in_range(T value)
{
    return (value <= upper) && (value >= lower);
}
```

This template has a type parameter, `T`, and two non-type parameters, `upper` and `lower`, that are both of type `int`. The compiler can't deduce all of the template parameters from the use of the function. The following function call won't compile:

```
double value {100.0};
std::cout << is_in_range(value);                // Won't compile - incorrect usage
```

Compilation fails because `upper` and `lower` are unspecified. To use this template, you must specify the template parameter values. The correct way to use this is:

```
std::cout << is_in_range<double, 0, 500>(value); // OK - checks 0 to 500
```

It would be better to use function parameters for the limits in this case. Function parameters give you the flexibility of being able to pass values that are calculated at runtime, whereas here you must supply the limits at compile time.

Trailing Return Types

The return type of a template function with multiple type parameters may depend on the types used to instantiate the template. Suppose you need a template function to generate the sum of the products of corresponding elements in two vectors of the same size. You might consider defining the template like this:

```
template <typename Treturn, typename T1, typename T2>
Treturn vector_product(const std::vector<T2>& data1, const std::vector<T3>& data2)
{
    if(data1.size() != data2.size()) return 0;    // Guard against unequal vectors

    Treturn sum {};
    for(size_t i {} ; i<count ; ++i) sum += v1[i]*v2[i];

    return sum;
}
```

This will work, but leaves it to the user to specify the return type. It might be better if the return was automatically the type of the result of multiplying two corresponding vector elements but how can you specify that? The `decltype` keyword provides the solution, at least in part. `decltype(expression)` produces the type of the result of evaluating expression. You could use this to rewrite the template as:

```
template<typename T1, typename T2>                // Won't compile!
decltype(data1[0]*data2[0])
    vector_product(const std::vector<T1>& data1, const std::vector<T2>& data2)
{
    if(data1.size() != data2.size()) return 0;    // Guard against unequal vectors

    decltype(data1[0]*data2[0]) sum {};
    for(size_t i {} ; i<count ; ++i) sum += v1[i]*v2[i];

    return sum;
}
```


The return type is now specified to be the type of value produced by the product of the first two vector elements and the type for `sum` is expressed in the same way. This template definition expresses what you want but it won't compile. The compiler processes the template from left to right so when the return type specification is processed, the compiler does not know the types of `data1[0]` and `data2[0]`. To overcome this the *trailing return* type syntax was introduced that permits the return type specification to appear after the parameter list, like this:

```
template <typename T1, typename T2>
auto vector_product(const std::vector<T1>& data1, const std::vector<T2>& data2)
    -> decltype(data1[0]*data2[0])
{
    if(data1.size() != data2.size()) return 0;    // Guard against unequal vectors

    decltype(data1[0]*data2[0]) sum {};
    for(size_t i {} ; i<count ; ++i) sum += v1[i]*v2[i];

    return sum;
}
```

The `auto` keyword before the function name indicates to the compiler that the return type specification is at the end of the function template header. You write the type specification following the indirect member selection operator, `->`, after the parameter list.

This syntax is primarily for use in function templates, and in lambda expressions which you'll meet in the next chapter but you can also use it to define ordinary non-template functions. For example, you could define a `larger()` function like this:

```
auto larger(double a, double b) -> double
{
    return a > b ? a : b;
}
```

If you use the trailing return type syntax in a function definition, you must use the same syntax in the function prototype; the prototype would be:

```
auto larger(double a, double b) -> double;    // Function prototype
```

Pointers to Functions

A “*pointer to a function*” is a variable that can store the address of a function and therefore can point to different functions at different times during execution. You use a pointer to a function to call the function at the address it contains. An address is not sufficient to call a function though. To work properly, a pointer to a function must also store the type of each parameter as well as the return type. Clearly, the information required to define a pointer to a function will restrict the range of functions to which the pointer can point. It can only store the address of a function with a given number of parameters of specific types and with a given return type. This is analogous to a pointer that stores the address of a data item. A pointer to type `int` can only point to a location that contains a value of type `int`.

Defining Pointers to Functions

Here's a definition of a pointer that can store the address of functions that have parameters of type `long*` and `int`, and return a value of type `long`:

```
long (*pfun)(long*, int);
```

This may look a little weird at first because of all the parentheses. The name of the pointer variable is `pfun`. It doesn't point to anything because it is not initialized. Ideally it would be initialized to `nullptr` or with the address of a specific function. The parentheses around the pointer name and the asterisk are essential — without them, this statement would declare a function rather than define a pointer variable because the `*` will bind to the type `long`.

The general form of a pointer to a function definition is:

```
return_type (*pointer_name)(list_of_parameter_types);
```

The pointer can only point to functions with the same `return_type` and `list_of_parameter_types` as those specified in its definition.

Of course, you should always initialize a pointer when you declare it. You can initialize a pointer to a function to `nullptr` or with the name of a function. Suppose you have a function with the following prototype:

```
long max_element(const long* array, size_t count);    // Function prototype
```

You can define and initialize a pointer to this function with this statement:

```
long (*pfun)(long*, size_t) {max_element};
```

The pointer is initialized with the address of `max_element()`. Using `auto` will make this simpler:

```
auto pfun = max_element;
```

This defines `pfun` as a pointer to any function with the same parameter list and return type as `max_element()` and initializes it with the address of `max_element()`. You can store the address of any function with the same parameter list and return type in an assignment. Given that `min_element()` has the same parameter list and return type as `max_element()`, you can make `pfun` point to it like this:

```
pfun = min_element;
```

As with pointers to variables, you should ensure that a pointer to a function contains the address of a function before you use it to call a function. Without initialization, catastrophic failure of your program is guaranteed.

To call `min_element()` using `pfun` you just use the pointer name as though it were a function name. For example:

```
long data[] {23, 34, 22, 56, 87, 12, 57, 76};
std::cout << "value of minimum is " << pfun(data, sizeof (data)/sizeof (data[0]));
```

This outputs the minimum value in the `data` array. To get a feel for these newfangled pointers to functions and how they perform in action, let's try one out in a working program:

```
// Ex8_16.cpp
// Exercising pointers to functions
#include <iostream>
long sum(long a, long b);           // Function prototype
long product(long a, long b);      // Function prototype
```

```

int main()
{
    long(*pDo_it)(long, long) {};           // Pointer to function

    pDo_it = product;
    std::cout << "3*5 = " << pDo_it(3, 5) << std::endl; // Call product thru a pointer

    pDo_it = sum;                           // Reassign pointer to sum()
    std::cout << "3 * (4+5) + 6 = "
                << pDo_it(product(3, pDo_it(4, 5)), 6) << std::endl; // Call thru a pointer twice
}

// Function to multiply two values
long product(long a, long b)
{
    return a*b;
}

// Function to add two values
long sum(long a, long b)
{
    return a + b;
}

```

This example produces the following output:

```

3*5 = 15
3 * (4+5) + 6 = 33

```

This is hardly a useful program but it does show how a pointer to a function is defined, assigned a value, and used to call a function. After the usual preamble, you define and initialize `pDo_it` as a pointer to a function, which can point to either of the functions `sum()` or `product()`.

`pDo_it` is initialized to `nullptr`, so before using it the address of the function `product()` is stored in `pDo_it`. `product()` is then called indirectly through the pointer `pDo_it` in the output statement. The name of the pointer is used just as if it were a function name and is followed by the function arguments between parentheses, exactly as they would appear if the original function name were being used. It would save a lot of complication if the pointer were defined and initialized like this:

```
auto pDo_it = product;
```

Just to show that you can, the pointer is changed to point to `sum()`. It is then used again in a ludicrously convoluted expression to do some simple arithmetic. This shows that you can use a pointer to a function in exactly the same way as the function to which it points. Figure 8-5 illustrates what happens.

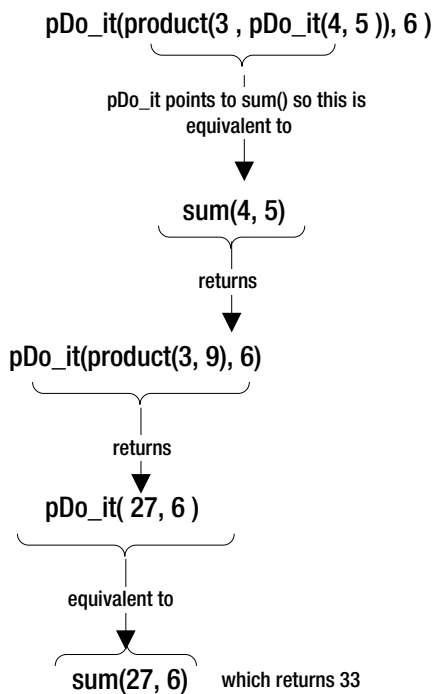


Figure 8-5. Execution of an expression using a function pointer

“*pointer to function*” is a perfectly reasonable type so a function can have a parameter of this type. The function can then use its *pointer to function* parameter to call the function to which the argument points when the function is called. You can specify just a function name as the argument for a parameter that is a “pointer to function” type. A function passed to another function as an argument is referred to as a *callback function*.

Recursion

A function can call itself and a function that contains a call to itself is referred to as a *recursive function*. A recursive function call can be indirect — for example, where a function `fun1()` calls another function `fun2()`, which in turn calls `fun1()`. Recursion may seem to be a recipe for a loop that executes indefinitely, and if you are not careful it certainly can be. A prerequisite for avoiding a loop of unlimited duration is that the function must contain some means of stopping the process.

Recursion can be used in the solution of many different problems. Compilers are sometimes implemented using recursion because language syntax is usually defined in a way that lends itself to recursive analysis. Data that is organized in a tree structure is another example. Figure 8-6 illustrates a tree structure. This shows a tree that contains structures that can be regarded as subtrees. Data that describes a mechanical assembly such as a car is often organized as a tree. A car consists of subassemblies such as the body, the engine, the transmission, and the suspension. Each of these consists of further subassemblies and components until ultimately, the leaves of the tree are reached, which are all components with no further internal structure.

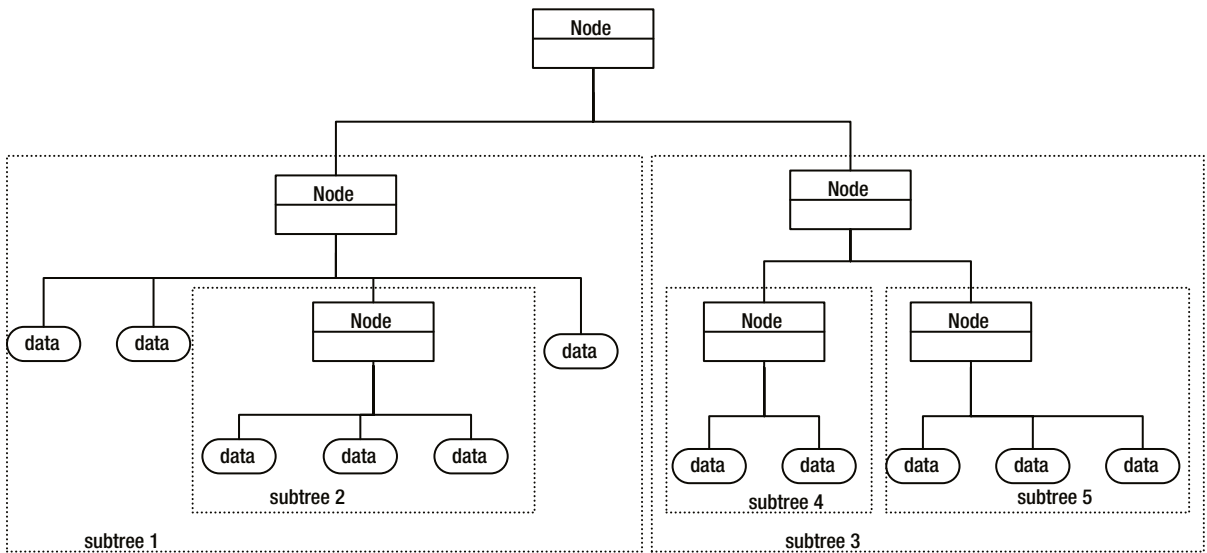


Figure 8-6. An example of a tree structure

Data that is organized as a tree can be traversed very effectively using recursion. Each branch of a tree can be regarded as a subtree, so a function for accessing the items in a tree can simply call itself when a branch node is encountered. When a data item is encountered, the function does what is required with the item and returns to the calling point. Thus, finding the leaf nodes of the tree — the data items — provides the means by which the function stops the recursive calls of itself.

There are many things in physics and mathematics that you can think of as involving recursion. A simple example is the factorial of a positive integer n (written as $n!$), which is the number of different ways in which n things can be arranged. For a given positive integer, n , the factorial of n is the product $1 \times 2 \times 3 \times \dots \times n$. The following recursive function calculates this:

```
long factorial(long n)
{
    if(n == 1L) return 1L;

    return n*factorial(n - 1);
}
```

If this function is called with an argument value of 4, the return statement that calls the function with a value of 3 in the expression executes. This will execute the return to call the function with an argument of 2, which will call `factorial()` with an argument of 1. The `if` expression will be true in this case so 1 will be returned, which will be multiplied by 2 in the next level up, and so on until the first call returns the value $4 \times 3 \times 2 \times 1$. This is very often the example given to show recursion but it is a very inefficient process. It would certainly be much faster to use a loop.

Here's another recursive function in a working example:

```
// Ex8_17.cpp
// Recursive version of function for x to the power n, n positive or negative
#include <iostream>
#include <iomanip>
```

```
double power(double x, int n);

int main()
{
    for (int i {-3} ; i <= 3 ; ++i)          // Calculate powers of 8 from -3 to +3
        std::cout << std::setw(10) << power(8.0, i);

    std::cout << std::endl;
}

// Recursive function to calculate x to the power n
double power(double x, int n)
{
    if (!n) return 1.0;                      // n zero
    if (n > 0) return x*power(x, n - 1);      // n positive

    return 1.0 / power(x, -n);               // n negative
}
```

The output is:

0.00195313	0.015625	0.125	1	8	64	512

The first `if` statement in `power()` returns 1.0 if `n` is 0. For positive `n`, the next `if` statement returns the result of the expression, `x*power(x, n-1)`. This causes a further call of `power()` with the index value reduced by 1. If, in this recursive function execution, `n` is still positive, then `power()` is called again with `n` reduced by 1. Each recursive call is recorded in the call stack, along with the arguments and return location. This repeats until `n` is 0, whereupon 1 is returned and the successive outstanding calls unwind, multiplying by `x` after each return. For a given value of `n` greater than 0, the function calls itself `n` times.

For negative powers of `n`, the reciprocal of `xn` is calculated so this uses the same process. You could shorten the code for the `power()` function by using the conditional operator. The function body comes down to a single line:

```
double power(double x, int n)
{
    return n ? (0 > n ? 1.0/power(x, -n) : x*power(x, n - 1)) : 1.0;
}
```

This doesn't improve the operation particularly, and it's not easier to understand what is happening. With this example too, the recursive call process is very inefficient compared to a loop. Every function call involves a lot of housekeeping. Implementing the `power()` function using a loop would make it execute a lot faster:

```
double power(double x, int n)
{
    if(!n) return 1.0;
    if(n < 0)
    {
        x = 1.0/x;
        n = -n;
    }
}
```

```
double result {x};
for(int i {1} ; i < n ; ++i)
    result *= x;

return result;
}
```

Unless you have a problem that particularly lends itself to using recursion or you have no obvious alternative, it is generally better to use a different approach, such as a loop. You need to make sure that the depth of recursion necessary to solve a problem is not itself a problem. For instance, if a function calls itself a million times, a large amount of stack memory will be needed to store copies of argument values and the return address for each call. However, in spite of the overhead, using recursion can often simplify the coding considerably. Sometimes this gain in simplicity can be well worth the loss in efficiency that you get with recursion.

Applying Recursion

Recursion is often favored in sorting and merging operations. Sorting data can be a recursive process in which the same algorithm is applied to smaller and smaller subsets of the original data. We can develop an example that uses recursion with a well-known sorting algorithm called *Quicksort*. The example will sort a sequence of words. I have chosen this because it demonstrates a lot of different coding techniques and it's sufficiently complicated to tax a few brain cells more than the examples you've seen up to now. The example involves more than 100 lines of code, so I'll show and discuss each of the functions in the book separately and leave you to assemble them into a complete working program. The complete program is available in the code download as `Ex8_18.cpp`.

The Quicksort Algorithm

Applying the Quicksort algorithm to a sequence of words involves choosing an arbitrary word in the sequence, and arranging the other words so that all those “less than” the chosen word precede it and all those “greater than” the chosen word follow it. Of course, the words on either side of the chosen word in the sequence will not necessarily be in sequence themselves. Figure 8-7 illustrates this process.

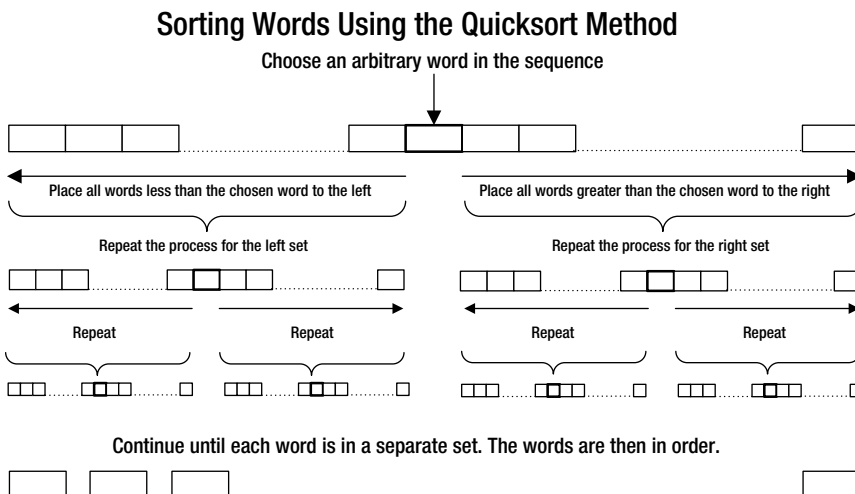


Figure 8-7. How the Quicksort algorithm works

The same process is repeated for smaller and smaller sets of words until each word is in a separate set. When that is the case, the process ends and the words are in ascending sequence. Of course, you'll rearrange addresses in the code, not move words around. The address of each word can be stored as a smart pointer to a `string` object and the pointers can be stored in a vector container.

The type of a vector of smart pointers to `string` objects is going to look a bit messy so won't help the readability of the code. Two type aliases will help to make the code easier to read:

```
using PWord = std::shared_ptr<string>;
using PWords = std::vector<PWord>;
```

`PWord` is a type alias for a smart pointer to a `string` object containing a word. `PWords` is a vector of such smart pointers so the full type of `PWords` in all its glory is `std::vector<std::shared_ptr<std::string>>`.

The `main()` Function

The definition of `main()` will be simple because all the work will be done by other functions. There will be several `#include` directives and prototypes for the other functions in the application preceding the definition of `main()`:

```
#include <iostream>
#include <iomanip>
#include <memory>
#include <string>
#include <vector>
using std::string;
using PWord = std::shared_ptr<string>;
using PWords = std::vector<PWord>;

void swap(PWords& pwords, size_t first, size_t second);
void sort(PWords& pwords, size_t start, size_t end);
void extract_words(PWords& pwords, const string& text, const string& separators);
void show_words(const PWords& pwords);
size_t max_word_length(const PWords& pwords);
```

I think by now you know why all these Standard Library headers are needed. `memory` is for smart pointer template definitions and `vector` contains the templates for vector containers. The `using` declaration for `std::string` will make the code less cluttered, as will the two type aliases.

There are five function prototypes:

- `swap()` is a helper function that interchanges the elements at indexes `first` and `second` in the words vector.
- `sort()` will use the Quicksort algorithm to sort a contiguous sequence elements in words from index `start` to index `end` inclusive. Indexes specifying a range are needed because the Quicksort algorithm involves sorting subsets of a sequence, as you saw earlier.
- `extract()` extracts words from `text` and stores smart pointers to the words in the words vector.
- `show_words()` outputs the words in words.
- `max_word_length()` determines the length of the longest word in words and is just to help make the output pretty.

The last two functions have `const` reference parameters for the words vector because they don't need to change it. The others have non-`const` reference parameters because they do. Here's the code for `main()`:

```
int main()
{
    PWords pwords;
    string text;                                // The string to be sorted
    const string separators{",.!?\\\"\\n"};      // Word delimiters

    // Read the string to be searched from the keyboard
    std::cout << "Enter a string terminated by *:" << std::endl;
    getline(std::cin, text, '*');

    extract_words(pwords, text, separators);
    if (pwords.size() == 0)
    {
        std::cout << "No words in text." << std::endl;
        return 0;
    }

    sort(pwords, 0, pwords.size() - 1);        // Sort the words
    show_words(pwords);                        // Output the words
}
```

The vector of smart pointers is defined using the type alias, `PWords`. The vector will be passed by reference to each function to avoid copying the vector and to allow it to be updated when necessary. Forgetting the `&` in the type parameter can lead to a mystifying error. If the parameter to a function that changes words is not a reference, then words is passed by value and the changes will be applied to the copy of words that is created when the function is called. The copy is discarded when the function returns and the original vector will be unchanged.

The process in `main()` is straightforward. After reading some text into the string object `text`, the text is passed to the `extract_words()` function that stores pointers to the words in `words`. After a check to verify that words is not empty, `sort()` is called to sort the contents of words, and `show_words()` is called to output the words.

The `extract_words()` Function

You have seen a function similar to this. Here's the code:

```
void extract_words(PWords& pwords, const string& text, const string& separators)
{
    size_t start {text.find_first_not_of(separators)};    // Start 1st word
    size_t end {};                                         // Index for the end of a word

    while (start != string::npos)
    {
        end = text.find_first_of(separators, start + 1); // Find end separator
        if (end == string::npos)                          // End of text?
            end = text.length();                          // Yes, so set to last+1
        pwords.push_back(std::make_shared<string>(text.substr(start, end - start)));
        start = text.find_first_not_of(separators, end + 1); // Find next word
    }
}
```

The last two parameters are `const` because the function won't change the arguments corresponding to these. The `separators` object could conceivably be defined as a static variable within the function, but passing it as an argument makes the function more flexible. The process is essentially the same as you have seen previously. Each substring that represents a word is passed to the `make_shared()` function that is defined in the memory header. The substring is used by `make_shared()` to create a `string` object in the free store along with a smart pointer to it. The smart pointer that `make_shared()` returns is passed to the `push_back()` function for the `words` vector to append it as a new element in the sequence.

The `swap()` Function

There'll be a need to swap pairs of addresses in the vector in several places, so it's a good idea to define a helper function to do this:

```
void swap(PWords& pwords, size_t first, size_t second)
{
    PWord temp{pwords[first]};
    pwords[first] = pwords[second];
    pwords[second] = temp;
}
```

This just swaps the addresses in words at indexes `first` and `second`.

The `sort()` function

You can use `swap()` in the implementation of the Quicksort method because it involves rearranging the elements in the vector. The code for the sorting algorithm looks like this:

```
void sort(PWords& pwords, size_t start, size_t end)
{
    // start index must be less than end index for 2 or more elements
    if (!(start < end))
        return;

    // Choose middle address to partition set
    swap(pwords, start, (start + end) / 2); // Swap middle address with start

    // Check words against chosen word
    size_t current {start};
    for (size_t i {start + 1} ; i <= end ; i++)
    {
        if (*(pwords[i]) < *(pwords[start])) // Is word less than chosen word?
            swap(pwords, ++current, i); // Yes, so swap to the left
    }

    swap(pwords, start, current); // Swap the chosen word with last in

    if (current > start) sort(pwords, start, current - 1); // Sort left subset if exists
    if (end > current + 1) sort(pwords, current + 1, end); // Sort right subset if exists
}
```

The parameters are the vector of addresses and the index positions of the first and last addresses in the subset to be sorted. The first time the function is called, `start` will be 0 and `end` will be the index of the last element. In subsequent recursive calls, a subsequence of the vector elements are to be sorted so `start` and/or `end` will be interior index positions in many cases.

The steps in the `sort()` function code are:

- The check for `start` not being less than `end` stops the recursive function calls. If there's one element in a set, the function returns. In each execution of `sort()` the current sequence is partitioned into two smaller sequences in the last two statements that call `sort()` recursively — so eventually you must end up with a sequence that has only one element.
- After the initial check, an address in the middle of the sequence is chosen arbitrarily as the pivot element for the sort. This is swapped with the address at index `start`, just to get it out of the way—you could also put it at the end of the sequence.
- The `for` loop compares the chosen word with the words pointed to by elements following `start`. If a word is *less* than the chosen word, its address is swapped into a position following `start`: the first into `start+1`, the second into `start+2`, and so on. The effect of this process is to position all the words less than the chosen word before all the words that are greater than or equal to it. When the loop ends, `current` contains the index of the address of the last word found to be less than the chosen word. The address of the chosen word at `start` is swapped with the address at `current` so the addresses of words less than the chosen word are now to the left of `current` and the addresses of words that are greater or equal are to the right.
- The last step sorts the subsets on either side of `current` by calling `sort()` for each subset. The indexes of words less than the chosen word run from `start` to `current-1` and the indexes of those greater run from `current+1` to `end`.

With recursion the code for the sort is relatively easy to follow, and it is shorter and less convoluted than if you implemented it as a loop. A loop is still faster, though.

The `max_word_length()` Function

This is a helper function that is used by the `show_words()` function:

```
size_t max_word_length(const PWords& pwords)
{
    size_t max {};
    for (auto& pword : pwords)
        if (max < pword->length()) max = pword->length();
    return max;
}
```

This steps through the words that the vector elements point to and finds and returns the length of the longest word. You could put the code in the body of this function directly in the `show_words()` function. However, code is easier to follow if you break it into small well-defined chunks. The operation that this function performs is self-contained and makes a sensible unit for a separate function.

The show_words() Function

This function outputs the words pointed to by the vector elements. It's quite long because it lists all words beginning with the same letter on the same line, with up to 10 words per line. Here's the code:

```
void show_words(const PWords& pwords)
{
    const size_t field_width {max_word_length(pwords) + 2};
    const size_t words_per_line {8}; // Word_per_line
    std::cout << std::left << std::setw(field_width) << *pwords[0]; // Output the first word

    size_t words_in_line {}; // Words in current line
    for (size_t i {1} ; i < pwords.size() ; ++i)
    { // Output - words newline when initial letter changes or after 10 per line
        if ((*pwords[i])[0] != (*pwords[i - 1])[0] || ++words_in_line == words_per_line)
        {
            words_in_line = 0;
            std::cout << std::endl;
        }
        std::cout << std::setw(field_width) << *pwords[i]; // Output a word
    }
    std::cout << std::endl;
}
```

The `field_width` variable is initialized to 2 more than the number of characters in the longest word. The variable is used for the field width for each word, so they will be aligned neatly in columns. There's also `words_per_line`, which is the maximum number of words on a line. The first word in output before the `for` loop. This is because the loop compare the initial character in the current word with that of the previous word to decide whether or not it should be on a new line. Outputting the first word separately ensures we have a previous word at the start. The `std::left` manipulator that is defined in the `iostream` header causes data to be left aligned in the output field. There's a complementary `std::right` manipulator. The rest of the words are output within the `for` loop. This output a newline character when 8 words have been written on a line or when a word with an initial letter that is different from the preceding word is encountered.

If you assemble the functions into a complete program, you'll have a good-sized example of a program split into several functions. Here's an example of the output:

Enter a string terminated by *:

```
It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of
foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of
light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we
had everything before us, we had nothing before us, we were all going direct to Heaven, we were all
going direct the other way—in short, the period was so far like the present period, that some of its
noisiest authorities insisted on its being received, for good or for evil, in the superlative degree
of comparison only.*
```

Darkness							
Heaven							
It							
Light							
age	age	all	all	authorities			
before	before	being	belief	best			
comparison							
degree	despair	direct	direct				
epoch	epoch	everything	evil				
far	foolishness	for	for				
going	going	good					
had	had	hope					
in	incredulity	insisted	it	it	it	it	it
it	it	it	it	its	its		
like							
noisiest	nothing						
of	of	of	of	of	of	of	of
of	of	of	of	on	only	or	other
period	period	present					
received							
season	season	short	so	some	spring	superlative	
that	the	the	the	the	the	the	the
the	the	the	the	the	the	the	times
times	to						
us	us						
was	was	was	was	was	was	was	was
was	was	was	way-in	we	we	we	we
were	were	winter	wisdom	worst			

Of course, words beginning with an uppercase letter precede all words beginning with lowercase letters.

Summary

This marathon chapter has introduced you to writing and using functions but this isn't everything relating to functions. You'll see more about functions in the context of user-defined types, starting in Chapter 11. The important bits that you should take away from this chapter are:

- Functions are self-contained compact units of code with a well-defined purpose. A well-written program consists of a large number of small functions, not a small number of large functions.
- A function definition consists of the function header that specifies the function name, the parameters, and the return type, followed by the function body containing the executable code for the function.
- A function prototype enables the compiler to process calls to a function even though the function definition has not been processed.
- The pass-by-value mechanism for arguments to a function passes copies of the original argument values, so the original argument values are not accessible from within the function.

- Passing a pointer to a function allows the function to change the value that is pointed to, even though the pointer itself is passed by value.
- Declaring a pointer parameter as `const` prevents modification of the original value.
- You can pass the address of an array to a function as a pointer.
- Specifying a function parameter as a reference avoids the copying that is implicit in the pass-by-value mechanism. A reference parameter that is not modified within a function should be specified as `const`.
- Specifying default values for function parameters allows arguments to be optionally omitted.
- Returning a reference from a function allows the function to be used on the left of an assignment operator. Specifying the return type as a `const` reference prevents this.
- The signature of a function is defined by the function name together with the number and types of its parameters.
- Overloaded functions are functions with the same name but with different signatures and therefore different parameter lists. Overloaded functions cannot be differentiated by the return type.
- A function template is a parameterized recipe used by the compiler to generate overloaded functions.
- The parameters in a function template can be type variables or non-type variables. The compiler creates an instance of a function template for each function call that corresponds to a unique set of template parameter arguments.
- A function template can be overloaded with other functions or function templates.
- The trailing return type syntax is used in the definition of a function template that has two or more parameters and where the return type in an instance of the template depends on the type arguments.
- A pointer to a function stores the address of a function, plus information about the number and types of parameters and the return type for a function. A pointer to a function can store the address of any function with the specified return type, and number and types of parameters.
- You can use a pointer to a function to call the function at the address it contains. You can also pass a pointer to a function as a function argument.
- A recursive function is a function that calls itself. Implementing an algorithm recursively can sometimes result in very elegant and concise code, but usually at the expense of execution time when compared to other methods of implementing the same algorithm.

EXERCISES

These exercises enable you to try out some of what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Apress website (www.apress.com/source-code), but that really should be a last resort.

Exercise 8-1. Write a function, `validate_input()`, that accepts two integer arguments that represent the upper and lower limits for an integer that is to be entered. It should accept a third argument that is a string describing the input, the string being used in the prompt for input to be entered. The function should prompt for input of the value within the range specified by the first two arguments and include the string identifying the type of value to be entered. The function should check the input and continue to prompt for input until the value entered by the user is valid. Use the `validate_input()` function in a program that obtains a user's date of birth and outputs it in the form of this example:

November 21, 2012

The program should be implemented so that separate functions, `month()`, `year()`, and `day()` manage the input of the corresponding numerical values. Don't forget leap years - February 29, 2013 is not allowed!

Exercise 8-2. Write a function that reads a string or array of characters as input and reverses it. Justify your choice of parameter type? Provide a `main()` function to test your function that prompts for a string of characters, reverses them, and outputs the reversed string.

Exercise 8-3. Write a program that accepts from two to four command line arguments. If it is called with less than two, or more than four arguments, output a message telling the user what they should do, and then exit. If the number of arguments is correct, output them, each on a separate line.

Exercise 8-4. Create a function, `plus()`, that adds two values and returns their sum. Provide overloaded versions to work with `int`, `double`, and `string` types, and test that they work with the following calls:

```
int n {plus(3, 4)};
double d {plus(3.2, 4.2)};
string s {plus("he", "llo")};
string s1 {"aaa"};
string s2 {"bbb"};
string s3 {plus(s1, s2)};
```

Can you explain why the following doesn't work?

```
int d {plus(3, 4.2)};
```

Exercise 8-5. Write a function that returns a reference to the smaller of two arguments of type `long`. Write another function that returns a reference to the larger of two arguments of type `long`. Use these functions to generate as many numbers of the Fibonacci sequence as the user requests. You will recall from this chapter that each number in the sequence is the sum of the two preceding it. (Hint: You can start with two numbers `n1` and `n2` that start out as 1. If you store the sum of the two in the smaller of the two and then output the larger, you should get what you want. The hard bit may be figuring out why this works!)

Exercise 8-6. Define the `plus()` function from Exercise 8-4 as a template, and test that it works for numeric types. Does your template work for the statement `plus("he", "llo")`? Can you explain this behavior? Suggest a solution to the problem.

Exercise 8-7. A recursive function called Ackerman's function is popular with some lecturers in computer science and mathematics. The function can be defined like this:

If `m` and `n` are integers, where `n >= 0` and `m >= 0`,

then $\text{ack}(m, n) = n + 1$, if `m == 0`;

$\text{ack}(m, n) = \text{ack}(m - 1, 1)$, if `m > 0` and `n == 0`;

$\text{ack}(m, n) = \text{ack}(m - 1, \text{ack}(m, n - 1))$, if `m > 0` and `n > 0`.

Define a function to compute Ackerman's function recursively. Test the function for values of `n` between 0 and 5, and `m` between 0 and 3. One particular property of this function is that the depth of recursion increases dramatically for small increases in `m` and `n`. For instance, calculating Ackerman's function recursively for quite modest values such as `n > 9` and `m > 3` is extremely difficult if not impossible on most computers.

Exercise 8-8. Define a function template to Quicksort a vector of numerical values of any type. Demonstrate its typical use for types `float` and `long long`.



Lambda Expressions

This chapter is dedicated to lambda expressions that provide a capability similar to that of a function. In this chapter you will learn:

- What a lambda expression is and what you use it for
- How you define a lambda expression
- How you pass a lambda expression as an argument to a function
- What options you have for specifying a function parameter that accepts a lambda expression as an argument
- What a capture clause is and how you use it
- How you can define recursive lambda expressions

Introducing Lambda Expressions

A *lambda expression* has a lot in common with the functions you learned about in the previous chapter, which is why the topic appears in this chapter. A lambda expression provides a way to define a function with no name—an anonymous function. More precisely, a lambda expression defines an object that encapsulates a function—usually referred to as a *function object*. At first sight you may find it difficult to imagine the applicability of this but it is immensely useful - and powerful. The primary use for lambda expressions is to pass a function as an argument to another function. Lambda expressions can be applied extensively in the Standard Template Library, although I won't be delving very far into that in this book.

A lambda expression is different from a regular function in that it can access variables that exist in the enclosing scope where it is defined. Because you can pass a lambda expression as an argument to another function, it provides an alternative to using function pointers. A lambda expression defines an anonymous object that represents a function. There's no need for an explicit definition of the type of the object. There's no generic "lambda expression type." Since I have said that you typically use a lambda expression to pass a function as an argument to another function, this immediately raises the question of how you define a function parameter where the argument is to be an arbitrary lambda expression. There is more than one possibility. A simple answer is to define a function template where the type parameter is the type of a lambda expression. When a lambda expression is passed as an argument to the function, the compiler deduces the parameter type to be used in the template instance.

Defining a Lambda Expression

Let's consider an example of a lambda expression. Suppose you want to calculate the cubes (x^3) of numerical values of type `double`. You can easily define a lambda expression to do this:

```
[] (double value) { return value*value*value; }
```

The opening square brackets are called the *lambda introducer*. They mark the beginning of the lambda expression. There's more to the lambda introducer than there is here—the brackets are not always empty—but I'll explain this in more depth a little later in this chapter. The lambda introducer is followed by the *lambda parameter list* between parentheses. This is just like a regular function parameter list. In this case, there's just a single parameter, `value`, but there could be more. There are restrictions on the parameter list for a lambda expression compared to normal functions. You cannot specify default parameter values for the parameters for one thing.

The body of the lambda expression between braces follows the parameter list, again just like a normal function. The body for this lambda contains just one statement, a `return` statement that also calculates the value that is returned. In general the body of a lambda can contain any number of statements. Note that there's no return type specification here. The return type defaults to that of the value returned. If nothing is returned, the return type is `void`. You can specify the return type. You use the trailing return type syntax that you met in the previous chapter to do this. You could supply it for the lambda above like this:

```
[] (double value) -> double { return value*value*value; }
```

The return type is specified following the `->` operator that comes after the parameter list and is type `double` here. You can execute a lambda expression when you define it. Here's a somewhat useless demonstration of this:

```
double cube {};  
cube = [] (double value) -> double { return value*value*value; }(3.5);    // 42.875
```

You could even define and execute a lambda in an initializer list:

```
cube {[] (double value) -> double { return value*value*value; }(3.5)};    // 42.875
```

It's obviously easier to just write an arithmetic expression here but there could be circumstances where this is useful - where the initial value is to be the maximum or minimum of a set of values in a sequence for example.

Naming a Lambda Expression

Although a lambda expression is an anonymous object, you can still store its address in a variable. You don't know what its type is, but the compiler does:

```
auto cube = [] (double value) -> double { return value*value*value; };
```

The `auto` keyword tells the compiler to figure out the type that the variable `cube` should have from whatever appears on the right of the assignment, so it will have the type necessary to store the address of the lambda expression. You can always do this if there is nothing between the square brackets - the lambda introducer. You'll see later that sometimes things between the square brackets will prevent you from using `auto` in this way. You can use `cube` just like the function pointers you learned about in the previous chapter. For example:

```
double x{2.5};  
std::cout << x << " cubed is " << cube(x) << std::endl;
```

The output statement will present the cube of 2.5. To convince you that this all really works, I'll put together a complete example:

```
// Ex9_01.cpp
// Using lambda expressions
#include <iostream>
#include <vector>
#include <string>
using std::string;

int main()
{
    auto cube = [](double value) -> double { return value*value*value; };
    double x {2.5};
    std::cout << x << " cubed is " << cube(x) << std::endl;

    auto average = [](const std::vector<double>& v) -> double
    {
        double sum{};
        for (const auto& x : v)
            sum += x;
        return sum / v.size();
    };
    std::vector<double> data {1.5, 2.5, 3.5, 4.5, 5.5};
    std::cout << "Average of values in data is " << average(data) << std::endl;

    string original {"ma is as selfless as I am"};
    string copy{original};
    reverse(copy);
    std::cout << "\"" << original << "\" reversed is \"" << copy << "\"" << std::endl;
}
```

The output is:

```
2.5 cubed is 15.625
Average of values in data is 3.5
"ma is as selfless as I am" reversed is "ma I sa sselfles sa si am"
```

There are three lambdas defined. The first contains a single statement and returns the cube of the argument as type double. The auto keyword for the type of cube causes the compiler to specify its type to store the address of the lambda. The address of the second lambda stored is average. Its definition is more complex in that its body consists of several statements to compute the average of the elements in the vector that it passes as the argument. The parameter is a const reference type, which allows elements values to be accessed but not modified; it also avoids copying the vector argument. The third lambda has a non-const parameter to allow the string argument to be modified. It reverses the sequence of characters in the string. There's no return value but you could return a reference to the argument, which would allow a call of the lambda to be used as a function argument. This would allow you to omit the statement that calls reverse() to reverse the characters in copy, and put the call directly in the output statement, like this:

```
std::cout << "\"" << original << "\" reversed is \"" << reverse(copy) << "\"" << std::endl;
```

The reference that `reverse()` returns is now passed to the insertion operator for `cout`. `copy` still exists in this scope with its characters reversed.

The example demonstrates that when you store the address of a lambda in a variable, you can use the variable to call the lambda in the same way that you would call a regular function.

Passing a Lambda Expression to a Function

As I've said, a lambda expression defines an anonymous object that represents a function. There's no need for an explicit definition of the type of the object. In general, you don't know the type of a lambda expression. There's no generic "lambda expression type." Since I have also said that you typically use a lambda expression to pass a function as an argument to another function, this immediately raises the question of how you define the type of a function parameter where the argument is to be a lambda expression. There is more than one possibility. A simple answer is to define a function template where the type parameter is the type of a lambda expression. When a lambda expression is passed as an argument in a function call, the compiler can deduce the parameter type for the template instance.

Function Templates that Accept Lambda Expression Arguments

The compiler always knows the type of a lambda expression so it can instantiate a function template with a parameter that will accept a given lambda expression as an argument. It's easy to see how this works with an example.

Suppose that you have a number of `double` values stored in a vector container that you want to be able to transform in arbitrary ways; sometimes you want to replace the values by their squares, or their square roots, or some more complex transformation that depends on whether or not the values lie within a particular range. You can define a template that allows the transformation of the vector elements to be specified by a lambda expression. Here's how the template looks:

```
template <typename F>
std::vector<double>& change(std::vector<double>& vec, F fun)
{
    for(auto& x : vec)
        x = fun(x);

    return vec;
}
```

The `fun` parameter will accept any suitable lambda expression that has a parameter of type `double` and returns a value of type `double`. You may wonder how the compiler deals with this template, bearing in mind that there's no information as to what `fun` does. The answer is that the compiler doesn't deal with it. The compiler doesn't process a template in any way until it needs instantiating. A template can contain any number of coding errors that won't be detected until the compiler meets a statement that uses the template. As soon as you write a statement that makes use of the template, an instance is created that is then compiled, which reveals any coding errors. In the case of the template above, all the information about the lambda is available to the compiler when you use it. Here's a program that uses this template:

```
// Ex9_02.cpp
// Passing lambda expressions as function arguments
#include <iostream>
#include <vector>

// Put the change<F>() template definition here...
```

```

int main()
{
    auto cube = [](double value) -> double { return value*value*value; };
    auto average = [](const std::vector<double>& v) -> double
    {
        double sum{};
        for (auto x : v)
            sum += x;
        return sum / v.size();
    };

    std::vector<double> data {1.5, 2.5, 3.5, 4.5, 5.5};
    std::cout << "Average of values in data is " << average(data) << std::endl;

    change(data, [](double x){ return (x + 1.0)*(x + 2.0); }); // Direct lambda argument
    std::cout << "Average of changed values in data is " << average(data) << std::endl;
    std::cout << "Average of cubes of values in data is " << average(change(data, cube))
        << std::endl;
}

```

The output is:

```

Average of values in data is 3.5
Average of changed values in data is 26.75
Average of cubes of values in data is 36257.2

```

This uses the `cube` and `average` lambdas from the previous example. The code in `main()` demonstrates that you can pass a variable containing the address of a lambda as the second argument to `change<F>()`, or you can pass an anonymous lambda explicitly as the second argument. The compiler determines the type `F` for `fun` to create an instance of the function template from the lambda that is passed as the second argument.

The last output statement passes the vector returned by `change<F>()` after the element values have been transformed by the `cube` lambda as the argument to the `average()` lambda.

A Function Parameter Type for Lambda Arguments

In the previous example, the `fun` parameter implies that the argument must represent a function that accepts a single parameter of type `double`. It also returns a result that can be stored as type `double`. You could replace the `change()` template definition in the previous example, `Ex9_02.cpp`, with a function definition, where the type for the `fun` parameter is a pointer to a function:

```

std::vector<double>& change(std::vector<double>& vec, double(*fun)(double))
{
    for (auto& x : vec)
        x = fun(x);

    return vec;
}

```

This specifies the `fun` parameter as a pointer to a function with a parameter of type `double` that returns a `double` value. The `change()` function will accept any function pointer that conforms to this type, including lambda expressions.

Only lambda expressions that have nothing between the lambda introducer will be accepted as arguments though. Stuff that does appear within the lambda introducer not only changes the type, it changes what the lambda can do very significantly. Before I get to that, let's look at another possibility for specifying a function parameter type to accept a lambda expression as the argument. This option will work when you do include stuff within the lambda introducer.

Using the `std::function` Template Type

The functional header in the Standard Library defines a template type, `std::function<>`, that is a wrapper for any kind of pointer to a function with a given set of return and parameter types; of course, this includes lambda expressions. The type argument for the `std::function` template is of the form `Return_Type(Param_Types)`. `Return_Type` is the type of value returned by the lambda expression (or function pointed to). `Param_Types` is a list of the parameter types for the lambda expression separated by commas. You can use the `std::function` type template to specify the type of any lambda expression, regardless of whether or not anything appears in the lambda introducer.

I can demonstrate how you use `std::function` with a revised version of the example from the previous section:

```
std::vector<double>& change(std::vector<double>& vec, std::function<double(double)> fun)
{
    for (auto& x : vec)
        x = fun(x);

    return vec;
}
```

The second parameter to the `change()` function uses the `std::function` template to specify the type. The lambda expressions that are acceptable arguments in a `change()` function call have a parameter of type `double` and return a `double` value so the type argument for the `std::function` template is `double(double)`. This version of `change()` will accept any lambda expression or pointer to function as an argument that has the specified combination of return and parameter types. You can plug it into `Ex9_02.cpp` to see it working.

A lambda expression is frequently used as a way of passing a comparison mechanism to a function. A new version of the `sort()` function from `Ex8_18.cpp` provides a more substantial demonstration of this:

```
void mysort(PWords& data, size_t start, size_t end,
            std::function<bool(const PWord, const PWord)> compare)
{
    // start index must be less than end index for 2 or more elements
    if (!(start < end))
        return;

    // Choose middle address to partition set
    std::swap(data[start], data[(start + end) / 2]);           // Swap middle address with start

    // Check words against chosen word
    size_t current{start};
    for (size_t i{start + 1}; i <= end; i++)
    {
        if (compare(data[i], data[start]))                    // Is word less than chosen word?
            std::swap(data[++current], data[i]);              // Yes, so swap to the left
    }
}
```

```

std::swap(data[start], data[current]); // Swap the chosen word with last in

if (current > start) mysort(data, start, current - 1, compare); // Sort left subset if exists
if (end > current + 1) mysort(data, current + 1, end, compare); // Sort right subset if exists
}

```

The first three parameters are the same as the `sort()` function in `Ex8_18.cpp`. The additional parameter allows a lambda expression to be passed that compares elements in the vector, which are of type `PWord`. A lambda argument must return a value of type `bool`.

The code in the body of `mysort()` is very similar to the original `sort()` function except for two differences; it uses the function pointed to by the fourth argument to compare vector elements and it uses a `swap()` function that is an instance of a template in the utility Standard Library header. Instances of this template can interchange two data items of any type. Using a lambda expression to compare elements will provide a great deal of flexibility in how the function can be used. You can now sort in ascending or descending sequence. Here's a complete example to demonstrate that:

```

// Ex9_03.cpp
// Sorting words in ascending or descending sequence
#include <iostream>
#include <iomanip> // For stream manipulators
#include <memory> // For smart pointers
#include <string> // for type string
#include <vector> // For vector<T> container
#include <functional> // For function<> type
#include <utility> // For swap() function template
using std::string;
using PWord = std::shared_ptr<string>;
using PWords = std::vector<PWord>;

// Function prototypes
void mysort(PWords& data, size_t start, size_t end,
            std::function<bool(const PWord, const PWord)> compare);
void extract_words(std::vector<std::shared_ptr<string>>& pwords, const string& text,
                  const string& separators);
void show_words(const std::vector<std::shared_ptr<string>>& pwords);
size_t max_word_length(const std::vector<std::shared_ptr<string>>& pwords);

int main()
{
    PWords pwords;
    string text; // The string to be sorted
    const string separators {" ,.!?\\"n"}; // Word delimiters

    // Read the string to be searched from the keyboard
    std::cout << "Enter a string terminated by *:" << std::endl;
    getline(std::cin, text, '*');
}

```

```

extract_words(pwords, text, separators);
if (pwords.size() == 0)
{
    std::cout << "No words in text." << std::endl;
    return 0;
}
size_t start {}, end {pwords.size() - 1};
std::cout << "\nWords in ascending sequence:\n";

// Sort the words
mysort(pwords, start, end, [](const PWord p1, const PWord p2) { return *p1 < *p2; });

show_words(pwords); // Output the words
std::cout << "\nWords in descending sequence:\n";
mysort(pwords, start, end, [](const PWord p1, const PWord p2) {return *p1 > *p2; });
// Sort the words
show_words(pwords); // Output the words
}

// Put definition of mysort() here...
// Put definitions from Ex8_18.cpp for extract_words(), show_words() and max_word_length() here...

```

The complete code is in the download for Ex9_03.cpp. Here's an example of the output:

```

Enter a string terminated by *:
To be, or not to be, that is the question.*

```

```

Words in ascending sequence:

```

```

To
be      be
is
not
or
question
that    the    to

```

```

Words in descending sequence:

```

```

to      the    that
question
or
not
is
be      be
To

```

The `mysort()` function uses the lambda that is passed as the fourth argument to compare elements in the vector. To sort in ascending sequence, you pass a lambda that defines a “less-than” comparison. For descending sequence you pass a lambda that defines a “greater-than” comparison. A further refinement would be to specify a default value for the compare parameter to `mysort()`. You just specify the default parameter value in the prototype:

```
void mysort(PWords& data, size_t start, size_t end,
           std::function<bool(const PWord, const PWord)> compare =
               [](const PWord p1, const PWord p2) {return *p1 < *p2; });
```

The default specification provides for sorting in ascending sequence so you omit the fourth argument in a `mysort()` function call to get this. When you want words sorted in descending sequence, you specify the fourth argument with the appropriate lambda expression.

■ **Note** Writing your own sort function is educational but not necessary. The algorithm Standard Library header defines an excellent function template `std::sort<>()` that will sort just about any sequence of elements. It provides for passing a lambda expression to specify the comparison, too.

The Capture Clause

As I’ve said, the lambda introducer, `[]`, is not necessarily empty. It can contain a *capture* clause that specifies how variables in the enclosing scope can be accessed from within the body of the lambda. The body of a lambda expression with nothing between the square brackets can only work with the arguments and with variables that are defined locally within the lambda. A lambda with no capture clause is called a *stateless lambda expression* because it cannot access anything in its enclosing scope.

A *default capture* clause applies to all variables in the scope enclosing the definition of the lambda. If you put `=` between the square brackets, the body of the lambda can access all automatic variables in the enclosing scope by value — that is, the values of the variables are made available within the lambda expression, but the values stored in the original variables cannot be changed. If you put `&` between the square brackets, all variables in the enclosing scope are accessible by reference, so their values can be changed by the code in the body of the lambda. To be accessible, variables must be defined preceding the definition of the lambda expression. You cannot use `auto` to specify the type of a variable to store the address of a lambda that accesses the variable containing its address. This implies you are trying to initialize the variable with an expression that uses the variable. You cannot use `auto` with any lambda that refers to the variable being defined—self-reference is not allowed with `auto`.

Here’s an example that uses the `change()` function template from the previous section:

```
std::vector<double> data {1.5, 2.5, 3.5, 4.5, 5.5};
double factor {10.0};
change(data, [=](double x){ return factor*x; });
std::cout << "The values in data are now:\n"
for(const auto& x : data)
    std::cout << " " << x;
std::cout << std::endl;
```

The `=` capture clause allows all the variables that are in scope where the definition of the lambda appears to be accessed by value from within the body of the lambda expression that is the second argument to the template function, `change()`. The effect is rather different from passing arguments by value. First, the value `factor` is available within the lambda, but you cannot update the temporary memory location that contains the value of `factor` because it is `const`. The following statement will not compile, for example:

```
change(data, [=](double x){ factor += 2.0;
                        return factor*x; });
```

If you want to modify the copy of a variable in the enclosing scope from within the lambda, you must add the `mutable` keyword to the lambda definition following the parentheses enclosing the parameter list, like this:

```
change(data, [=](double x) mutable { factor += 2.0;
                                    return factor*x; });
```

Adding the `mutable` keyword enables you to modify the copy of any variable within the enclosing scope, which doesn't change the original variable of course. After executing this statement, the value of `factor` will still be 10.0.

There's another significant difference from accessing arguments passed by value to a function. The lambda remembers the local value of `factor` from one call to the next, so the copy is effectively static. The `change()` function applies the lambda that is passed as the second argument to successive elements in `data`. For the first element in `data`, the local `factor` copy will be $10.0 + 2.0 = 12.0$, for the second element, it will be $12.0 + 2.0 = 14.0$, and so on. After `change()` finishes executing, `factor` in the outer scope will still be 10.0.

■ **Warning** Capturing all the variables in the enclosing scope by value can add a lot of overhead because they will each have a copy created, whether or not you refer to them.

Using `&` as the capture clause allows access to `factor` and any other variables in the enclosing scope by reference, so the original value of `factor` will be modified by the lambda:

```
change(data, [&](double x) { factor += 2.0;
                          return factor*x; });
```

The `mutable` keyword is not necessary in this case. All variables within the outer scope are available by reference, so the lambda can use and alter their values. The result of executing this will be that the value of `factor` will be 20.0. The lambda expression executes once for each element in `data`, and the elements will be multiplied by successive values of `factor` from 12.0 to 20.0. Although the `&` capture clause is legal, capturing all variables in the outer scope by reference is not usually a good idea because of the potential for accidentally modifying one of them. It's much better to capture only the variables you need. I'll explain how you do this next.

Capturing Specific Variables

You can identify specific variables in the enclosing scope that you want to access by reference by listing them in the capture clause, with each name prefixed with `&`. You could rewrite the previous statement as:

```
change(data, [&factor](double x) { factor += 2.0;
                                return factor*x; });
```

Here, `factor` is the only variable in the enclosing scope that can be accessed from within the body of the lambda and the `&factor` specification makes it available by reference. Without the `&`, the `factor` variable in the outer scope would be available by value and not updatable. When you put several variables in the capture clause, you separate them with commas. You can include `=` in the capture clause along with specific variable names that are to be captured. The capture clause `[=, &factor]` would allow access to `factor` by reference and any other variables in the enclosing scope by value. Alternatively, you could write the capture clause as `[&, factor]` which would capture `factor` by value and all other variables by reference. You would also need to specify the `mutable` keyword to modify the copy of `factor`.

Using Lambda Expressions in a Template

You can define and use a lambda expression inside a function template definition. The next example will show this as well as passing a lambda to a function template instance. Here's the code:

```
// Ex9_04.cpp
// Using lambda expressions in function templates
#include <iostream>
#include <iomanip>
#include <vector>
#include <functional>
#include <cmath>                                // For pow()
using std::vector;

// Template function to set a vector to values determined by a lambda expression
template <typename T> void setValues(vector<T>& vec, std::function<void(T)> fun)
{
    for (size_t i {} ; i < vec.size() ; ++i)
        fun(vec[i]);
}

// Template function to list the values in a vector
template<class T> void listVector(const vector<T>& vec)
{
    int count {} ;                                // Counts number of outputs
    const int valuesPerLine {5};
    auto print = [&count, valuesPerLine](T value) {
        std::cout << std::setw(10) << value << " ";
        if (++count % valuesPerLine == 0) std::cout << std::endl;
    };
    for (size_t i {} ; i < vec.size() ; ++i)
        print(vec[i]);
}

int main()
{
    // Populate vector with values 1+1=2, 2+2=4, 4+3=7, 7+4=11, ...
    vector<int> integerData(50);
    int current {1};
    int increment {1};
    setValues<int>(integerData, [increment, &current](int& v) mutable{ v = current + increment++;
                                                                    current = v; });

    std::cout << "Integer vector contains : " << std::endl;
    listVector(integerData);
}
```

```
// Populate vector with x to nth power, x = 2.5
vector<double> values(10);
size_t power {};
double x {2.5};
setValues<double>(values, [power, x](double& v) mutable{ v = std::pow(x, power++); });
std::cout << "\nDouble vector contains:" << std::endl;
listVector(values);
}
```

This example produces the following output:

Integer vector contains :

2	4	7	11	16
22	29	37	46	56
67	79	92	106	121
137	154	172	191	211
232	254	277	301	326
352	379	407	436	466
497	529	562	596	631
667	704	742	781	821
862	904	947	991	1036
1082	1129	1177	1226	1276

Double vector contains:

1	2.5	6.25	15.625	39.0625
97.6563	244.141	610.352	1525.88	3814.7

The first function template, `setValues<T>()`, sets the elements of the vector passed as the first argument using the lambda expression that is passed as the second argument. The lambda has a parameter of type `T` and has no return value. The code in the body of the template is just a `for` loop that iterates over the elements in the vector, applying the lambda to each in turn.

The second function template, `listVector<T>()`, lists the elements in a vector of elements of type `T`. Each element is output in a `for` loop by the lambda that initializes the `print` variable. You can see that the lambda parameter is of type `T`, which is the template type parameter, so the lambda will be defined for each specific instance of the function template. The `count` variable in the enclosing scope is captured by reference so the lambda can increment it for each value that is written to `cout`. `values_per_line` is captured by value so the lambda only has access to a copy.

The `main()` function applies these template functions first to a vector of 50 integer elements. The lambda that is passed to `setValues<T>()` captures `increment` by value and `current` by reference. The `mutable` keyword allows the lambda to modify the copy of `increment` and carry its value forward from one call of the lambda to the next. The original value of `increment` in the enclosing scope will be unchanged. Note the explicit type argument for the function template instance; this is essential to allow a call of `setValues<T>()` to compile because the compiler cannot otherwise deduce the type of the lambda parameter. Outputting the values of the vector elements just requires calling `listVector()` with `vec` as the argument. There's no need to specify the template type argument explicitly, although you can if you want to.

The second block of code in `main()` applies the function templates to a vector of double elements. The lambda that is passed to `setValues<T>()` is designed to set the element values to x^0 , x , x^2 , x^3 , and so on, with x having the value 2.5. The lambda captures `power` and `x` from the enclosing scope by value. The `mutable` keyword enables the copy of `power` to be updated on each call, and its value retained from one call to the next. Obviously, the copy of `x` could also be updated if this was necessary. The reference parameter for the lambda allows the element that is passed to be updated in the lambda. I hope this little examples shows you a little of the potential of lambda expressions.

Recursion in Lambda Expressions

When you store the address of a lambda expression in a variable, you make it possible for the lambda to be called from within another lambda expression; this can include the same lambda expression if you define the variable in the right way. Of course, the variable that points to the lambda is defined in the enclosing scope. A lambda that calls another lambda must capture the variable in the enclosing scope that contains its address. Recursion is evidently self-referential, so using `auto` for the type of the variable that stores the address of the lambda is not allowed. You must use `std::function` in this case.

I'll demonstrate this in action with an example that finds the highest common factor (HCF) for a pair of integer values. The HCF is the largest number that divides into both integers with zero remainder. The HCF is also referred to as the greatest common divisor (GCD). The program uses Euclid's method for finding the highest common factor for two integer values. The process is to first divide the larger number by the smaller number. If the remainder is zero, the HCF is the smaller number. If the remainder is non-zero, the process continues by dividing the previous smaller number by the remainder, and a zero remainder indicates the remainder is the HCF. If the remainder is non-zero, the process is repeated until the remainder is zero. Here's the code to implement Euclid's method:

```
// Ex9_05.cpp
// Recursive calls in a lambda expression
#include <iostream>
#include <functional>           // For function<>

int main()
{
    // A lambda expression that returns the HCF of its arguments
    std::function<long long(long long, long long)> hcf =
        [&hcf](long long i, long long j) mutable ->long long {
            if (i < j) return hcf(j, i);
            long long remainder{i%j};
            if (!remainder) return j;
            return hcf(j, remainder);    });

    // A lambda expression that outputs the HCF of the arguments
    auto showHCF = [&hcf](long long a, long long b) {
        std::cout << "The highest common factor of " << a << " and " << b
            << " is " << hcf(a, b) << std::endl;
    };

    long long a {}, b {};
    while (true)
    {
        std::cout << "\nEnter two integers to find their HCF, or 0 to end: ";
        std::cin >> a;
        if (!a) break;
        std::cin >> b;

        showHCF(a, b);
    }
}
```

This produces the output:

```
Enter two integers to find their HCF, or 0 to end: 8961 9001891
The highest common factor of 8961 and 9001891 is 103

Enter two integers to find their HCF, or 0 to end: 7932729108943 483489887381237
The highest common factor of 7932729108943 and 483489887381237 is 941

Enter two integers to find their HCF, or 0 to end: 0
```

The variable `hcf` stores the address of the lambda expression that determines the HCF of its arguments. The type of `hcf` is specified using the `std::function` template type. The lambda calls itself recursively using `hcf` in order to work, and because this is defined outside the scope of the lambda, you cannot use a regular function pointer type for `hcf`. You cannot use `auto` either because self-reference is prohibited with `auto`. However, `std::function` can always hack it. The lambda expression has two parameters of type `long long` and returns a value of type `long long`, so `hcf` is of type:

```
function<long long(long long, long long)>
```

The code for the lambda expression assumes `m` is the larger of the two arguments, so if it isn't, it calls `hcf()` with the arguments reversed. If `m` is the larger number, the remainder after dividing `m` by `n` is calculated. If the remainder is zero, then `n` is the highest common factor and is returned. If the remainder is not zero, `hcf()` is called with `n` and the remainder.

The next statement in `main()` initializes `showHCF` with a lambda to find the HCF and output it. `auto` is fine here because the capture clause only captures `hcf` and it does not reference itself. The rest of `main()` calls this lambda through the `showHCF` function pointer in a loop.

Summary

This chapter introduced the basics of how you define and use lambda expressions. Lambda expressions are a powerful tool when applied in general. They come into their own in the context of the Standard Template Library where many of the template functions have a parameter for which you can supply a lambda expression as the argument. The most important points covered in this chapter are:

- A lambda expression defines an anonymous function. Lambda expressions are typically used to pass a function as an argument to another function.
- A lambda expression always begins with a lambda introducer that consists of a pair of square brackets that can be empty.
- The lambda introducer can contain a capture clause that specifies which variables in the enclosing scope can be accessed from the body of the lambda expression. Variables can be captured by value or by reference.
- There are two default capture clauses: `&` specifies that all variables in the enclosing scope are captured by reference, `=` specifies that all variables in the enclosing scope are to be captured by value.
- A capture clause can specify specific variables to be captured by value or by reference.

- Variables captured by value will have a local copy created. The copy is not modifiable by default. Adding the `mutable` keyword following the parameter list allows local copies of variables captured by value to be modified.
- When the `mutable` keyword is specified, copies of variables from the enclosing scope are essentially static so their values are carried forward from one call of the lambda to the next.
- You can specify the return type for a lambda expression using the trailing return type syntax. If you don't specify a return type, the compiler deduces the return type from the first return statement in the body of the lambda.
- You can use the `std::function<>` template type that is defined in the functional header to specify the type of a function parameter that will accept a lambda expression as an argument.

EXERCISES

Exercise 9-1. Define a lambda expression to find the largest even number in a vector of non-zero elements of type `int`. Demonstrate its use in a suitable test program.

Exercise 9-2. Define a lambda expression that will multiply the value of a double variable that is passed by reference by a scale factor that is defined in the enclosing scope. Demonstrate that the lambda works by applying it to the elements of a vector.

Exercise 9-3. Define and test a lambda expression that returns the count of the number of elements in a `vector<string>` container that begin with a given letter.

Exercise 9-4. Define and demonstrate a recursive lambda expression that accepts an unsigned integer as an argument and returns an integer that corresponds to the argument with its decimal digits reversed.



Program Files and Preprocessing Directives

This chapter is more about managing code than writing code. I'll discuss how multiple program files and header files interact, and how you manage and control their contents. The material in this chapter has implications for how you define your data types, which you'll learn about starting in the next chapter.

In this chapter you will learn:

- How header files and source files interrelate
- What a translation unit is
- What linkage is and why it is important
- More detail on how you use namespaces
- What preprocessing is, and how to use the preprocessing directives to manage code
- The basic ideas in debugging, and the debugging help you can get from preprocessing and the Standard Library
- How you use the `static_assert` keyword

Understanding Translation Units

You know that header files primarily contain definitions that are used by source files that contain the executable code. The contents of a header file are made available in a source file by using an `#include` preprocessing directive. So far you have only used header files that provide the information necessary for using Standard Library capabilities. The program examples have been short and simple; consequently, they have not warranted the use of separate header files containing your own definitions. In the next chapter, when you learn how to define your own data types, the need for header files will become apparent. A typical practical C++ program involves many header files that are included into many source files.

Each source file along with the contents of the header files that you include into it is called a *translation unit*. The term "translation unit" is a somewhat abstract term because this isn't necessarily a file in general, although it will be with the majority of C++ implementations. The compiler processes each translation unit in a program independently to generate an object file. The object file contains machine code and information about references to entities such as functions that were not defined in the translation unit - external entities in other words. The set of object files for a complete program are processed by the *linker*, which establishes all necessary connections between the object files to produce the executable program module. If an object file contains references to an external entity that is not found in any of the other object files, no executable module will result and there will be one or more error messages from the linker.

The “One Definition” Rule

Each variable, function, class type, enumeration type, or template in a translation unit must only be *defined* once. You can have more than one *declaration* for a variable or function for example, but there must be only one *definition* that determines what it is and causes it to be created. If there's more than one definition, the code will not compile.

Inline functions are an exception. The definition of an inline function *must* appear in every translation unit that calls the function, but all definitions of a given inline function in all translation units must be identical. For this reason, you should always define inline functions in a header file that you include in a source file when one is required.

You have seen that you can define variables in different blocks to have the same name but this does not violate the one definition rule; the variables may have the same name but they are distinct.

Of course, you will usually use a given data type in more than one translation unit so several translation units in a program can each include a definition for the type; this is legal only as long as the definitions are identical. In practice you achieve this by placing the definition for a type in a header file and use an `#include` directive to add the header file to any source file that requires the type definition. However, duplicate definitions for a given type are illegal within a single translation unit, so you need to be careful how you define the contents of header files. You must make sure that duplicate type definitions within a translation unit cannot occur. You'll see how you do this later in this chapter.

Program Files and Linkage

Entities in one translation unit often need to be accessed from code in another translation unit. Functions are obvious examples of where this is the case, but you can have others — variables defined at global scope that are shared across several translation units, for instance. Because the compiler processes one translation unit at a time, such references can't be resolved by the compiler. Only the linker can do this when all the object files from the translation units in the program are available.

The way that names in a translation unit are handled in the compile/link process is determined by a property that a name can have called *linkage*. Linkage expresses where in the program code the entity that is represented by a name can be. Every name that you use in a program either has linkage, or doesn't. A name has linkage when you can use it to access something in your program that is *outside* the scope in which the name is declared. If this isn't the case, it has no linkage. If a name has linkage, then it can have *internal linkage* or *external linkage*. Therefore, every name in a translation unit has internal linkage, external linkage, or no linkage.

Determining Linkage for a Name

The linkage that applies to a name is not affected by whether its declaration appears in a header file or a source file. The linkage for each name in a translation unit is determined *after* the contents of any header files have been inserted into the `.cpp` file that is the basis for the translation unit. The linkage possibilities have the following meanings:

Internal linkage: The entity that the name represents can be accessed from anywhere within the same translation unit. For example, the names of variables defined at global scope that are specified as `const` have internal linkage by default.

External linkage: A name with external linkage can be accessed from another translation unit in addition to the one in which it is defined. In other words, the entity that the name represents can be shared and accessed throughout the entire program. All the functions that you have written so far have external linkage and so do non-`const` variables that are defined at global scope.

No linkage: When a name has no linkage, the entity it refers to can only be accessed from within the scope that applies to the name. All names that are defined within a block — local names, in other words — have no linkage.

Now, the interesting question is this: From within a function, how do you access a variable that is defined in another translation unit? This comes down to how you declare a variable to be *external*.

External Names

In a program made up of several files, the linker establishes (or *resolves*) the connection between a function call in one source file and the function definition in another. When the compiler compiles a *call* to the function, it only needs the information contained in a function prototype to create the call. The compiler doesn't mind whether the function's *definition* occurs in the same file or elsewhere. This is because function names have external linkage by default. If a function is not defined within the translation unit in which it is called, the compiler flags the call as external and leaves it for the linker to sort out.

Variables are different. The compiler needs to know if the definition for a variable name is external to the current translation unit. If you want to access a variable that is defined *outside* the current translation unit, then you must declare the variable name using the `extern` keyword, as you saw in Chapter 3:

```
extern double pi;
```

This statement is a declaration that `pi` is a name that is defined outside of the current block. The type must correspond exactly to the type that appears in the definition. You can't specify an initial value in an `extern` declaration because it's a declaration of the name, not a definition of a variable. Declaring a variable as `extern` implies that it is defined in another translation unit. This causes the compiler to mark the variable as having external linkage. It is the linker that makes the connection between the name and the variable to which it refers.

const Variables with External Linkage

Suppose that a source file defines the following at global scope:

```
const double pi {3.14159265};
```

A `const` variable has internal linkage by default, which makes it unavailable in other translation units. You can override this by using the `extern` keyword in the definition:

```
extern const double pi {3.14159265};    // Has external linkage
```

The `extern` keyword tells the compiler that the name should have external linkage, even though it is `const`. When you want to access `pi` in another source file, you must declare it as `const` and `external`:

```
extern const double pi;                // Variable is defined in another file
```

Within any block in which this declaration appears, the name `pi` refers to the constant defined in another file. The declaration can appear in any translation unit that needs access to `pi`. You can place the declaration either at global scope in a translation unit so that it's available throughout the code in the source file, or within a block in which case it is only available within that local scope.

Global variables can be useful for constant values that you want to share because they are accessible in any translation unit. By sharing constant values across all of the program files that need access to them, you can ensure that the same values are being used for the constants throughout your program. However, although up to now I have shown constants defined in source files, the best place for them is in a header file.

Preprocessing Your Source Code

Preprocessing is a process executed by the compiler before a source file is compiled into machine instructions. Preprocessing prepares and modifies the source code for the compile phase according to instructions that you specify by *preprocessing directives*. All preprocessing directives begin with the symbol `#`, so they are easy to distinguish from C++ language statements. Table 10-1 shows the complete set.

Table 10-1. *Preprocessing Directives*

Directive	Description
<code>#include</code>	Supports header file inclusion
<code>#if</code>	Enables conditional compilation
<code>#else</code>	else for <code>#if</code>
<code>#elif</code>	Equivalent to <code>#else #if</code>
<code>#endif</code>	Marks the end of an <code>#if</code> directive
<code>#define</code>	Defines an identifier
<code>#undef</code>	Deletes an identifier
<code>#if defined</code> (or <code>#ifdef</code>)	Does something if an identifier defined
<code>#if !defined</code> (or <code>#ifndef</code>)	Does something if an identifier is not defined
<code>#line</code>	Redefines the current line number and/or filename
<code>#error</code>	Outputs a compile-time error message and stop the compilation. This is typically part of a conditional preprocessing directive sequence.
<code>#pragma</code>	Offers machine-specific features while retaining overall C++ compatibility

Note All preprocessing directives begin with `#`.

The preprocessing phase analyzes, executes, and then removes all preprocessing directives from a source file. This generates the translation unit that consists purely of C++ statements that is then compiled. The linker must then process the object file that results along with any other object files that are part of the program to produce the executable module.

Several of these directives are primarily applicable in C and are not so relevant with current C++. The language capabilities of C++ provide much more effective and safer ways of achieving the same result as some of the preprocessing directives. I'll only discuss the preprocessing directives that are important in C++. You are already familiar with the `#include` directive. There are other directives that can provide considerable flexibility in the way in which you specify your programs. Keep in mind that preprocessing operations occur before your program is compiled. Preprocessing modifies the set of statements that constitute your program and the preprocessing directives no longer exist in the source file that is compiled and thus they are not involved in the execution of your program at all.

You may wonder why you would want to use of the `#line` directive to change the line number. The need for this is rare, but one example is a program that maps some other language into C or C++. An original language statement may generate several C++ statements and by using the `#line` directive you can ensure that C++ compiler error messages identify the line number in the original code, rather than the C++ that results. This makes it easier to identify the statement in the original code that is the source of the error.

Defining Preprocessing Identifiers

The general form of the `#define` preprocessing directive is:

```
#define identifier sequence_of_characters
```

This defines `identifier` as an alias for `sequence_of_characters`. `identifier` must conform to the usual definition of an identifier in C++ — any sequence of letters and digits, the first of which is a letter, and where the underline character counts as a letter. `sequence_of_characters` can be any sequence of characters, including an empty sequence.

One use for `#define` is to define an identifier that is to be replaced in the source code by a substitute string during preprocessing. Here's how you could define `PI` to be an alias for a sequence of characters that represents a numerical value:

```
#define PI 3.14159265
```

`PI` *looks* like a variable but this has nothing to do with variables. `PI` is a symbol or **token**, which is exchanged for the specified sequence of characters by the preprocessor before the code is compiled. `3.14159265` is not a numerical value in the sense that no validation is taking place; it is merely a string of characters. The string `PI` will be replaced during preprocessing by its definition, the sequence of characters `3.14159265`, wherever the preprocessing operation deems that the substitution makes sense. If you wrote `3, !4!5` as the replacement character sequence, the substitution would still occur. The `#define` directive is often used to define symbolic constants in C but don't do this in C++. It is much better to define a constant variable, like this:

```
const long double pi {3.14159265L};
```

`pi` is a constant value of a particular type. The compiler ensures that the value for `pi` is consistent with its type. You could place this definition in a header file for inclusion in any source file where the value is required, or define it with external linkage:

```
extern const long double pi {3.14159265L};
```

Now you may access `pi` from any translation unit just by adding an extern declaration for it wherever it is required.

Here's another example:

```
#define BLACK WHITE
```

Any occurrence of `BLACK` in the file will be replaced by `WHITE`. The identifier will only be replaced when it is a token. It will not be replaced if it forms part of an identifier or appears in a string literal or a comment. There's no restriction on the sequence of characters that is to replace the identifier. It can even be absent in which case the identifier exists but with no predefined substitution string - the substitution string is empty. If you don't specify a substitution string for an identifier, then occurrences of the identifier in the code will be replaced by an empty string — in other words, the identifier will be removed. For example:

```
#define VALUE
```

The effect is that all occurrences of `VALUE` that follow the directive will be removed. The directive also defines `VALUE` as an identifier and its existence can be tested by other directives, as you'll see.

The major use for the `#define` directive with C++ is in the management of header files, as you'll see later in this chapter.

■ **Caution** Using a `#define` directive to define an identifier that you use to specify a value in C++ code has three major disadvantages: there's no type checking support, it doesn't respect scope, and the identifier name cannot be bound within a namespace.

Undefining an Identifier

You may want to have the identifier resulting from a `#define` directive only to exist in *part* of a program file. You can nullify a definition for an identifier using the `#undef` directive. You can negate a previously defined `VALUE` identifier with this directive:

```
#undef VALUE
```

`VALUE` is no longer defined following this directive so no substitutions for `VALUE` can occur. The following code fragment illustrates this:

```
#define PI 3.142
// All occurrences of PI in code from this point will be replaced by 3.142
// ...
#undef PI
// PI is no longer defined from here on so no substitutions occur.
// Any references to PI will be left in the code.
```

Between the `#define` and `#undef` directives, preprocessing replaces appropriate occurrences of `PI` in the code with `3.142`. Elsewhere, occurrences of `PI` are left as they are. The combination of `#define` and `#undef` directives has another use, which I'll explain when I deal with decision-making preprocessing directives later in this chapter.

Including Header Files

A header file is an external file contents are included in a source file using the `#include` preprocessing directive. Header files contain primarily type definitions, template definitions, function prototypes, and constants. You are already completely familiar with statements such as this:

```
#include <iostream>
```

The contents of the `iostream` standard library header replaces the `#include` directive. This will be the definitions required to support input and output with the standard streams. Any Standard Library header name can appear between the angled brackets. If you `#include` a header that you don't need, the primary effect is to extend the compilation time and the executable may occupy more memory than necessary. It may also be confusing for anyone who reads the program.

You include your own header files into a source file with a slightly different syntax where you enclose the header file name between double quotes. Here's an example:

```
#include "myheader.h"
```

The contents of the file named `myheader.h` are introduced into the program in place of the `#include` directive. The contents of any file can be included into your program in this way. You simply specify the file name of the file between quotes as in the example. With the majority of compilers, you the file name can use upper- and lowercase characters. In theory, you can assign any name and extension you like to your header files — you don't have to use the extension `.h`. However, it is a convention adhered to by most C++ programmers, and I'd recommend that you follow it too.

The process used to find a header file depends on whether you specify the file name between double quotes or between angled brackets. The precise operation is implementation-dependent and should be described in your compiler documentation. Usually, the compiler only searches the default directories that contain the Standard Library headers for the file when the name is between angled brackets. This implies that your header files will not be found if you put the name between angled brackets. If the header name is between *double quotes*, the compiler searches the current directory (typically the directory containing the source file that is being compiled) followed by the directories containing the standard headers. If the header file is in some other directory, you may need to put the complete path for the header file or the path relative to the directory containing the source file between the double quotes.

■ **Note** A file introduced into a source file by an `#include` directive can contain `#include` directives. The `#include` directives in an included header are preprocessed in the same way. This continues until there are no `#include` directives in the source file.

Preventing Duplication of Header File Contents

You have already seen that you don't have to specify a value when you define an identifier:

```
#define MYHEADER_H
```

This creates `MYHEADER_H` so it exists from here on and represents an empty character sequence. You can use the `#if defined` directive to test whether a given identifier has been defined and include code or not in the file depending on the result:

```
#if defined MYHEADER_H
    // The code here will be placed in the source file...
    // ...if MYHEADER_H has been defined. Otherwise it will be omitted.
#endif
```

All the lines following `#if defined` up to the `#endif` directive will be kept in the file if the identifier, `MYHEADER_H`, has been defined previously and omitted if it has not. The `#endif` directive marks the end of the text that is controlled by the `#if defined` directive. You can use the abbreviated form, `#ifdef`, if you prefer:

```
#ifdef MYHEADER_H
    // The code down to #endif will be placed in the source file...
    // ...if MYHEADER_H has been defined...
    // ...otherwise it will be omitted.
#endif
```

You can use the `#if !defined`, or its equivalent, `#ifndef`, to test for an identifier not having been defined:

```
#if !defined MYHEADER_H
```

```
// The code down to #endif will be placed in the source file...
// ...if MYHEADER_H has NOT been defined...
// ...otherwise, the code will be omitted.
#endif
```

Here, the lines following `#if !defined` down to the `#endif` are included in the file to be compiled provided the identifier has not been defined previously. This pattern is the basis for the mechanism that is used to ensure that the contents of a header file are not duplicated in a source file:

```
// Header file myheader.h
#ifndef MYHEADER_H
// If MYHEADER_H has NOT been defined...
// ...everything down to #endif will be included in the source file...
// ...including the next directive that defines MYHEADER_H
#define MYHEADER_H
// The entire code for myheader.h is placed here.
// This code will be placed in the source file...
// ...only if MYHEADER_H has NOT been defined previously.
#endif
```

If a header file, `myheader.h`, that has contents like this is included into a source file more than once, the first `#include` directive will include the code because `MYHEADER_H` has not been defined. In the process it will define `MYHEADER_H`. Any subsequent `#include` directives for `myheader.h` in the source file or in other header files that are included into the source file will not include the code because `MYHEADER_H` will have been defined previously.

This is an important mechanism that you should put in all your header files. All the code in every header should be between an `#ifndef` - `#endif` pair of directives in the pattern above. As I noted earlier, a header file that you include into a source file can contain `#include` directives; this feature is used extensively in large programs and in the Standard Library headers. With a complex program involving many header files, there's a good chance that a header file may potentially be `#included` more than once in a source file and in some situations it is unavoidable. By using the `#ifndef` - `#endif` pattern above in your header files, you eliminate the potential for violations of the “one definition” rule.

■ **Tip** Some compilers use the `#pragma` directive to achieve the same effect as the pattern I have described. For example, `#pragma once` at the beginning of a header file may be all that is necessary to prevent duplication of the contents.

Namespaces

I introduce *namespaces* back in Chapter 1 but there's a bit more to it than I explained then. With large programs, choosing unique names for all the entities that have external linkage can become difficult. When an application is developed by several programmers working in parallel, using namespaces to prevent name clashes becomes essential. Name clashes are perhaps most likely in the context of user-defined types, or classes, which you will meet in the next few chapters.

A namespace is a block that attaches an extra name—the namespace name—to every entity name that is declared or defined within it. The full name of each entity is the namespace name followed by the scope resolution operator, `::`, followed by the basic entity name. Different namespaces can contain entities with the same name, but the entities are differentiated because they are qualified by different namespace names.

■ **Note** A declaration introduces a name into a scope. A definition introduces the name and defines what it is so a definition is also a declaration.

You typically use a separate namespace within a single program for each collection of code that encompasses a common purpose. Each namespace would represent some logical grouping of functions, together with any related global variables and declarations. A namespace would also be used to contain a unit of release, such as a library.

You are already aware that Standard Library names are declared within the `std` namespace. You also know that you can reference any name from a namespace without qualifying it with the namespace name by using a blanket using directive:

```
using namespace std;
```

However, this defeats the purpose of using namespaces in the first place and increases the likelihood of errors due to the accidental use of a name in the `std` namespace. It is much better to use qualified names or add using declarations for the names from another namespace that you are referencing.

The Global Namespace

All the programs that you've written so far have used names that you defined in the *global namespace*. The global namespace applies by default if a namespace hasn't been defined. All names within the global namespace are just as you declare them, without a namespace name being attached. In a program with multiple source files, all the names with linkages are within the global namespace.

With small programs, you can define your names within the global namespace without running into any problems. With larger applications, the potential for name clashes increases, so you should use namespaces to partition your code into logical groupings. That way, each code segment is self-contained from a naming perspective, and name clashes are prevented.

Defining a Namespace

You can define a namespace with these statements:

```
namespace myRegion
{
    // Code you want to have in the namespace, including
    // function definitions and declarations, global variables,
    // templates, etc.
}
```

Note that no semicolon is required after the closing brace in a namespace definition. The namespace name here is `myRegion`. This uniquely identifies the namespace, and this name will be attached to all the entities defined within it. The braces enclose the scope for the namespace `myRegion`, and every name declared within the namespace scope has the name `myRegion` attached to it.

■ **Caution** You must not include the `main()` function within a namespace. The runtime environment expects `main()` to be defined in the global namespace.

You can extend a namespace scope by adding a second namespace block with the same name. For example, a program file might contain the following:

```
namespace calc
{
    // This defines namespace calc
    // The initial code in the namespace goes here
}
namespace sort
{
    // Code in a new namespace, sort
}
namespace calc
{
    /* This extends the calc namespace
       Code in here can refer to names in the previous
       calc namespace block without qualification */
}
```

There are two blocks defined as namespace `calc`, separated by a namespace `sort`. The second `calc` block is treated as a continuation of the first, so functions declared within each of the `calc` blocks belong to the same namespace. The second block is called an *extension namespace* definition because it extends the original namespace definition. You can have several extension namespace definitions in a translation unit.

Of course, you wouldn't choose to organize a source file so that it contains multiple namespace blocks in this way but it can occur anyway. If you include several header files into a source file then you may effectively end up with the sort of situation I just described. An example of this is when you include several Standard Library headers (each of which contributes to the namespace `std`), interspersed with your own header files:

```
#include <iostream>           // In namespace std
#include "mystuff.h"          // In namespace calc
#include <string>              // In namespace std - extension namespace
#include "morestuff.h"        // In namespace calc - extension namespace
```

Note that references to names from inside the same namespace do not need to be qualified. For example, names that are defined in the namespace `calc` can be referenced from within `calc` without the qualifying them with the namespace name.

Let's look at an example that illustrates the mechanics of declaring and using a namespace. The program will consist of two source `.cpp` files. The first containing definitions of some `const` variables:

```
// Ex10_01_data.cpp
// Using a namespace
#include <string>
namespace data
{
    extern const double pi {3.14159265};
    extern const std::string days[]
        {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"};
}
```

I organized the program as two source files to show how you access definitions in one translation unit from another but it would be better to put definitions in a header file as I'll explain shortly. `pi` and `days[]` are defined within the `data` namespace. The `days[]` array is of type `string`, which is defined in the Standard Library, so the type name is qualified with `std`.

The second source file contains `main()`:

```
// Ex10_01_code.cpp
// Using a namespace
#include <iostream>
#include <string>

namespace data
{
    extern const double pi;           // Variable is defined in another file
    extern const std::string days[];  // Array is defined in another file
}

int main()
{
    std::cout << "pi has the value " << data::pi << std::endl;
    std::cout << "The second day of the week is " << data::days[1] << std::endl;
}
```

The two source files are separate translation units that are compiled independently to produce object files. These are linked to produce the executable module. This example produces the following output:

```
pi has the value 3.14159
The second day of the week is Monday
```

You must declare `pi` and `days[]` as external in the file containing `main()` because they are defined in a separate translation unit. The declarations for the external variables are within the `data` namespace because the variables are defined within this namespace in the first `.cpp` file. This demonstrates the point that I discussed earlier — a namespace can be defined piecemeal. A single file can contain several namespace blocks with the same namespace name, and their contents will be in the same namespace. Type `string` is defined within the standard library namespace so you have to supply the qualified name `std::string` in the declaration.

As I said, this is not the best way to organize the code for this program. The definitions for `pi` and `days` should be in a header file, `data.h`, for example. The contents of this header file would be:

```
// Ex10_01.h
// Definitions for globals in namespace data
#include <string>
#ifndef EX10_01_H
#define EX10_01_H
namespace data
{
    extern const double pi {3.14159265};
    extern const std::string days[] {
        "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"
    };
}
#endif
```

To make the definitions available in the file containing `main()` or any other file that needs access to these constants, you would add an `#include` directive at the beginning:

```
#include "Ex10_01.h"
```

There is now only one translation unit. The preprocessing directives in the header ensure that its contents cannot appear more than once in a translation unit.

Applying using Declarations

Just to formalize what I have been doing in previous examples I'll remind you of the using declaration for a single name from a namespace:

```
using namespace_name::identifier;
```

`using` is a keyword, `namespace_name` is the name of the namespace, and `identifier` is the name that you want to use unqualified. This declaration introduces a single name from the namespace, which could represent anything that has a name. For instance, a set of overloaded functions defined within a namespace can be introduced with a single using declaration.

Although I've placed using declarations and directives at global scope in the examples, you can also place them within a namespace, or within a function, or even within a statement block. In each case, the declaration or directive applies until the end of the block that contains it.

■ **Note** When you use an unqualified name, the compiler first tries to find the definition in the current scope, prior to the point at which it is used. If the definition is not found, the compiler looks in the immediately enclosing scope. This continues until the global scope is reached. If a definition for the name is not found at global scope (which could be an extern declaration), the compiler concludes that the variable is not defined.

Functions and Namespaces

For a function to exist within a namespace, it is sufficient for the function prototype to appear in the namespace. You can define the function elsewhere using the qualified name for the function; in other words, the function definition doesn't have to be enclosed in a namespace block. Let's explore an example. Suppose you write two functions, `max()` and `min()`, that return the maximum and minimum of a vector of values. You can put the declarations for the functions in a namespace as follows:

```
// compare.h
// For Ex10_02.cpp
#include <vector>
#ifndef COMPARE_H
#define COMPARE_H

namespace compare
{
    using std::vector;
    double max(const vector<double>& data);
    double min(const vector<double>& data);
}
#endif
```

This code would be in a header file, `compare.h`, which can be included by any source file that used the functions. The definitions for the functions can now appear in a `.cpp` file. You can write the definitions without enclosing them in a namespace block, as long as the name of each function is qualified with the namespace name. The contents of the file would be:

```
// compare.cpp
// For Ex10_02.cpp
#include <vector>
#include "compare.h"

// Function to find the maximum
double compare::max(const std::vector<double>& data)
{
    double result {data[0]};
    for(const auto value : data)
        if(result < value) result = value;
    return result;
}

// Function to find the minimum
double compare::min(const std::vector<double>& data)
{
    double result {data[0]};
    for(const auto value : data)
        if(result > value) result = value;
    return result;
}
```

You need the `compare.h` header file to be included so that the namespace is identified. This tells the compiler to deduce that the functions are declared within the namespace. There's an `#include` directive for the vector header that is also included into `compare.h`. The contents of the vector header will only appear once in this file because all the Standard Library headers have preprocessing directives to prevent duplication. It's a good idea in general to have `#include` directives for every header that a file uses, even when one header may include another header that you use. This makes the file independent of potential changes to the header files.

Of course you could place the code for the function definitions within the `compare` namespace directly. In this case, the contents of `compare.cpp` would be:

```
#include <vector>

namespace compare
{
    double max(const std::vector<double>& data)
    {
        // Code for max() as above...
    }

    double min(const std::vector<double>& data)
    {
        // Code for min() as above...
    }
}
```

If you write the function definitions in this way, then you don't need to `#include compare.h` into this file. This is because the definitions are within the namespace. Using the functions is the same, however you have defined them. To confirm how easy it is, let's try it out with the functions that you've just defined. Create the `compare.h` header file with the contents I discussed earlier. Create the first version of `compare.cpp` where the definitions are not defined in a namespace block. All you need now is a `.cpp` file containing the definition of `main()` to try the functions out:

```
// Ex10_02.cpp
// Using functions in a namespace
#include <iostream>
#include <vector>
#include "compare.h"

using compare::max;           // Using declaration for max
using compare::min;          // Using declaration for min

int main()
{
    std::vector<double> data {1.5, 4.6, 3.1, 1.1, 3.8, 2.1};
    std::cout << "Minimum value is " << min(data) << std::endl;
    std::cout << "Maximum double is " << max(data) << std::endl;
}
```

All the files for examples with more than one file will be in a separate folder in the code download so the files for this example will be in the *Ex10_02* folder. If you compile the two `.cpp` files and link them, executing the program produces the following output:

```
Minimum double is 1.1
Maximum double is 4.6
```

Caution The example assumes that `compare.h` is in the same directory as the source file. If `compare.h` is in a different folder from the source files, then the `#include` directive must contain the full path to `compare.h` or the path relative to the folder containing the source file.

There is a `using` declaration for each function in `compare.h` so you can use the names without having to add the namespace name. You could equally well have used a `using` directive for the `compare` namespace in this case:

```
using namespace compare;
```

The namespace only contains the functions `max()` and `min()` so this would have been just as good and one less line of code. Without the `using` declarations for the function names (or a `using` directive for the `compare` namespace), you would have to qualify the functions like this:

```
std::cout << "Minimum value is " << compare::min(data) << std::endl;
std::cout << "Maximum double is " << compare::max(data) << std::endl;
```

Unnamed Namespaces

You don't have to assign a name to a namespace, but this doesn't mean it doesn't have a name. You can declare an unnamed namespace with the following code:

```
namespace
{
    // Code in the namespace, functions, etc.
}
```

This creates a namespace that has an internal name that is generated by the compiler. Only one “unnamed” namespace exists in a file, so additional namespace declarations without a name will be extensions of the first. However, unnamed namespaces within distinct translation units are distinct unnamed namespaces.

Note that an unnamed namespace is not within the global namespace. This fact, combined with the fact that an unnamed namespace is unique to a translation unit, has significant consequences. It means that functions, variables, and anything else declared within an unnamed namespace are local to the translation unit in which they are defined. They can't be accessed from another translation unit. Placement of function definitions within an unnamed namespace has the same effect as declaring the functions as `static` in the global namespace. Declaring functions and variables as `static` at global scope was a common way of ensuring they weren't accessible outside their translation unit. An unnamed namespace is a much better way of restricting accessibility where necessary, and using `static` for this is deprecated.

Namespace Aliases

In a large program with multiple development groups, long namespace names may be necessary to ensure that you don't have accidental name clashes. Such long names may be unduly cumbersome to use; having to attach names such as `System_Group5_Process3_Subsection2` to every function call would be more than a nuisance. To get over this, you can define an alias for a namespace name on a local basis. The general form of the statement you'd use to define an alias for a namespace name is as follows:

```
namespace alias_name = original_namespace_name;
```

You can then use `alias_name` in place of `original_namespace_name` to access names within the namespace. For example, to define an alias for the namespace name in the previous paragraph, you could write this:

```
namespace SG5P3S2 = SystemGroup5_Process3_Subsection2;
```

Now you can call a function within the original namespace with a statement such as this:

```
int maxValue {SG5P3S2::max(data)};
```

Nested Namespaces

You can define one namespace inside another. The mechanics of this are easiest to understand if I take a specific context. For instance, suppose you have the following nested namespaces:

```
// outin.h
namespace outer
{
    double max(const std::vector<double>& data)
    {
        // body code..
    }

    double min(const std::vector<double>& data)
    {
        // body code..
    }

    namespace inner
    {
        void normalize(std::vector<double>& data)
        {
            // ...
            double minValue {min(data, size)};    // Calls max() in outer namespace
            // ...
        }
    }
}
```

From within the inner namespace, the `normalize()` function can call the function `min()` in the namespace `outer` without qualifying the name. This is because the declaration of `normalize()` in the inner namespace is also within the outer namespace.

To call `min()` from the global namespace, you qualify the function name in the usual way:

```
double result{outer::min(data)};
```

Of course, you could use a `using` declaration for the function name or specify a `using` directive for the namespace. To call `normalize()` from the global namespace, you must qualify the function name with both namespace names:

```
outer::inner::normalize(data);
```

The same applies if you include the function prototype within the namespace and supply the definition separately. You could write just the prototype of `normalize()` within the inner namespace and place the definition of `normalize()` in the file `outin.cpp`:

```
// outin.cpp
#include "outin.h"
void outer::inner::normalize(std::vector<double>& data)
```

```

{
    // ...
    double minValue{min(data)};           // Calls min() in outer
    // ...
}

```

Of course, to compile this successfully, the compiler needs to know about the namespaces. Therefore `out.in.h`, which I `#include` here prior to the function definition, needs to contain the namespace declarations.

Logical Preprocessing Directives

The logical `#if` works in essentially the same way as an `if` statement in C++. Among other things this allows conditional inclusion of code and/or further preprocessing directives in a file, depending on whether or not preprocessing identifiers have been defined, or based on identifiers having specific values. This is particularly useful when you want to maintain one set of code for an application that may be compiled and linked to run in different hardware or operating system environments. You can define preprocessing identifiers that specify the environment for which the code is to be compiled and select code and or `#include` directives accordingly.

The Logical `#if` Directive

You have seen in the context of managing the contents of a header file that a logical `#if` directive can test whether or not a symbol has been previously defined. You can also use the directive test whether or not a constant expression is true. Of course, you can use the technique that protects the contents of a header file from multiple inclusions to selectively include code in a source file. Suppose you put the following code in your program file:

```

// code that sets up the array data[...]...

#ifdef CALCAVERAGE
double average {};
size_t count {sizeof data/sizeof data[0]};
for(size_t i {} ; i < count ; ++i)
    average += data[i];
average /= count;
std::cout << "Average of data array is " << average << std::endl;
#endif

// rest of the program...

```

If the identifier `CALCAVERAGE` has been defined by a previous preprocessing directive, the code between the `#if` and `#endif` directives is compiled as part of the program. If `CALCAVERAGE` has not been defined, the code won't be included.

Testing for Specific Identifier Values

The general form of the `#if` directive is:

```
#if constant_expression
```


The `constant_expression` must be an integral constant expression that does not contain casts. All arithmetic operations are executed with the values treated as type `long` or `unsigned long`. If the value of `constant_expression` is nonzero, then lines following the `#if` down to the `#endif` will be included in the code to be compiled. The most common application of this uses simple comparisons to check for a particular identifier value. For example, you might have the following sequence of statements:

```
#if ADDR == 64
    // Code taking advantage of 64-bit addressing...
#endif
```

The statements between the `#if` directive and `#endif` are only included in the program here if the identifier `ADDR` has been defined as 64 in a previous `#define` directive.

Multiple Choice Code Selection

The `#else` directive works in the same way as the C++ `else` statement, in that it identifies a sequence of lines to be included in the file if the `#if` condition fails. This provides a choice of two blocks, one of which will be incorporated into the final source. Here's an example:

```
#if ADDR == 64
    std::cout << "64-bit addressing version." << std::endl;
    // Code taking advantage of 64-bit addressing...
#else
    std::cout << "Standard 32-bit addressing version." << std::endl;
    // code for standard processors...
#endif
```

One or other the sequences of statements will be included in the file, depending on whether or not `ADDR` has been defined as 64.

There is a special form of `#if` for multiple choice selections. This is the `#elif` directive, which has the following general form:

```
#elif constant_expression
```

Here is an example of how you might use this:

```
#if LANGUAGE == ENGLISH
    #define Greeting "Good Morning."
#elif LANGUAGE == GERMAN
    #define Greeting "Guten Tag."
#elif LANGUAGE == FRENCH
    #define Greeting "Bonjour."
#else
    #define Greeting "Hi."
#endif
std::cout << Greeting << std::endl;
```

With this sequence of directives, the output statement will display one of a number of different greetings, depending on the value assigned to `LANGUAGE` in a previous `#define` directive.

Another possible use is to include different code depending on an identifier that represents a version number:

```
#if VERSION == 3
    // Code for version 3 here...
#elif VERSION == 2
    // Code for version 2 here...
#else
    // Code for original version 1 here...
#endif
```

This allows you to maintain a single source file that compiles to produce different versions of the program depending on how `VERSION` has been set in a `#define` directive.

Standard Preprocessing Macros

There are several standard predefined preprocessing macros and the most useful are listed in Table 10-2.

Table 10-2. *Predefined Preprocessing Macros*

Macro	Description
<code>__LINE__</code>	The line number of the current source line as a decimal integer literal.
<code>__FILE__</code>	The name of the source file as a character string literal.
<code>__DATE__</code>	The date when the source file was processed as a character string literal in the form <code>Mmm dd yyyy</code> . Here, <code>Mmm</code> is the month in characters, (Jan, Feb, etc.); <code>dd</code> is the day in the form of a pair of characters 1 to 31, where single digit days are preceded by a blank; and <code>yyyy</code> is the year as four digits (such as 2014).
<code>__TIME__</code>	The time at which the source file was compiled, as a character string literal in the form <code>hh:mm:ss</code> , which is a string containing the pairs of digits for hours, minutes, and seconds separated by colons.

Note that each of the macro names in Table 10-2 start and end with two underscore characters. The `__LINE__` and `__FILE__` macros cause reference information relating to the source file to be displayed. You can modify the current line number using the `#line` directive and subsequent line numbers will increment from that. For example, to start line numbering at 1000 you would add this directive:

```
#line 1000
```

You can use the `#line` directive to change the string returned by the `__FILE__` macro. It usually produces the fully qualified file name, but you can change it to whatever you like. Here's an example:

```
#line 1000 "The program file"
```

This directive changes the line number of the next line to 1000, and alters the string returned by the `__FILE__` macro to "The program file". This doesn't alter the file name, just the string returned by the macro. Of course, if you just wanted to alter the apparent file name and leave the line numbers unaltered, you could use the `__LINE__` macro in the `#line` directive:

```
#line __LINE__ "The program file"
```

You can use the date and time macros to record when your program was last compiled with a statement such as this:

```
std::cout << "Program last compiled at " << __TIME__ << " on " << __DATE__ << std::endl;
```

When this statement is compiled, the values displayed by the statement are fixed until you compile it again. Thus the program outputs the time and date of its last compilation.

Debugging Methods

Most of your programs will contain errors, or *bugs*, when you first complete them. There are many ways in which bugs can arise. Most simple typos will be caught by the compiler so you'll find these immediately. Logical errors or failing to consider all possible variations in input data will take longer to find. Debugging is the process of eliminating these errors. Debugging a program represents a substantial proportion of the total time required to develop it. The larger and more complex the program, the more bugs it's likely to contain, and the more time and effort you'll need to make it run properly. Very large programs — operating systems, for example, or complex applications such as word processing systems, or even the C++ program development system that you may be using at this moment — can be so complex that the system will never be completely bug free. You will already have some experience with this through the regular patches and updates to the operating system and some of the applications on your computer. Most bugs in this context are relatively minor and don't limit the usability of the product greatly. The most serious bugs in commercial products tend to be security issues.

Your approach to writing a program can significantly affect how difficult it will be to test and debug. A well-structured program that consists of compact functions, each with a well-defined purpose, is much easier to test than one without these attributes. Finding bugs will also be easier with a program that has well-chosen variable and function names, and comments that document the operation and purpose of its component functions. Good use of indentation and statement layout can also make testing and fault-finding simpler.

It is beyond the scope of this book to deal with debugging comprehensively. The book concentrates on the standard C++ language and library, independent of any particular C++ development system and it's more than likely you'll be debugging your programs using tools that are specific to the development system you have. Nevertheless, I'll explain some basic ideas that are general and common to most debugging systems. I'll also introduce the rather elementary debugging aids within the Standard Library.

Integrated Debuggers

Many C++ compilers come with a program development environment that has extensive debugging tools built in. These potentially powerful facilities can dramatically reduce the time needed to get a program working and if you have such a development environment, familiarizing yourself with how you use it for debugging will pay substantial dividends. Common tools include the following:

Tracing Program Flow: This allows you to execute a program by *stepping through* the source code one statement at a time. A program has to be compiled in "debug mode" to make this possible. It depends on the presence of additional machine instructions that allow you to pause execution after each statement has been executed; it continues with the next statement when you press a designated key. Other provisions of the debug environment usually allow you to display information about the variables at each pause.

Setting Breakpoints: Stepping through a large program one statement at a time can be very tedious. It may even be impossible to step through the program in a reasonable period of time. Stepping through a loop that executes 10,000 times is an unrealistic proposition. Breakpoints identify specific statements in program at which execution pauses to allow you to check the program state. Execution continues to the next breakpoint when you press a specified key.

Setting Watches: A watch identifies a specify variable whose value you wish to track as execution progresses. The values of variables identified by watches you have set are displayed at each pause point. If you step through your program statement by statement, you can see the exact point at which values are changed, and sometimes when they unexpectedly don't change.

Inspecting Program Elements: You can usually examine a variety of program components when execution is paused. For example, at breakpoints you can examine details of a function, such as its return type and its arguments, or information relating to a pointer, such as its location, the address it contains, and the data at that address. It is sometimes possible to access the values of expressions and to modify variables. Modifying variables can often allow problem areas to be bypassed, allowing subsequent code to be executed with correct data.

Preprocessing Directives in Debugging

Although many C++ development systems provide powerful debug facilities, adding your own tracing code can still be useful. You can use conditional preprocessing directives to include blocks of code to assist during testing, and omit the code when testing is complete. You can control the formatting of data that will be displayed for debugging purposes, and you can arrange for the output to vary according to conditions or relationships within the program.

I'll illustrate how you can use preprocessing directive to help with debugging through a somewhat contrived program that calls functions at random through an array of function pointers. This example also gives you a chance to review a few of the techniques that you should be familiar with by now. Just for this exercise you'll declare three functions that you'll use in the example within a namespace, `fun`. First, you'll put the namespace declaration in a header file:

```
// functions.h
#ifndef FUNCTIONS_H
#define FUNCTIONS_H
namespace fun
{
    // Function prototypes
    int sum(int, int);           // Sum arguments
    int product(int, int);       // Product of arguments
    int difference(int, int);     // Difference between arguments
}
#endif
```

Enclosing the contents of the header file between an `#if/#endif` directive combination prevents the contents from being `#included` into a translation unit more than once. The prototypes are defined within the namespace, `fun`, so the function names are qualified with `fun` and the function definitions must appear in the same namespace.

You can put the functions definitions in the file `functions.cpp`:

```
// functions.cpp

#ifndef TESTFUNCTION           // Uncomment to get trace output

#ifdef TESTFUNCTION
#include <iostream>             // Only required for trace output...
#endif

#include "functions.h"

// Definition of the function sum
int fun::sum(int x, int y)
{
    #ifdef TESTFUNCTION
    std::cout << "Function sum called." << std::endl;
    #endif
}
```

```

    return x+y;
}

// Definition of the function product
int fun::product(int x, int y)
{
    #ifdef TESTFUNCTION
        std::cout << "Function product called." << std::endl;
    #endif

    return x*y;
}

// Definition of the function difference
int fun::difference(int x, int y)
{
    #ifdef TESTFUNCTION
        std::cout << "Function difference called." << std::endl;
    #endif

    return x-y;
}

```

You only need the `iostream` header because you use stream output statements to provide trace information in each function. The `iostream` header will only be included, and the output statements compiled, if the identifier `TESTFUNCTION` is defined in the file. `TESTFUNCTION` isn't defined at present because the directive is commented out.

The `main()` function is in a separate `.cpp` file:

```

// Ex10_03.cpp
// Debugging using preprocessing directives
#include <iostream>
#include <cstdlib>                // For random number generator
#include <ctime>                  // For time function

#include "functions.h"
using std::cout;
using std::endl;

#define TESTINDEX

// Function to generate a random integer 0 to count-1
size_t random(size_t count)
{
    return static_cast<size_t>(
        count*static_cast<unsigned long>(std::rand())/(RAND_MAX+1UL));
}

int main()
{
    int a {10}, b {5};           // Starting values
    int result {};               // Storage for results
}

```

```

// Declaration for an array of function pointers
int (*pfun[])(int, int) {fun::sum, fun::product, fun::difference};

size_t fcount {sizeof pfun/sizeof pfun[0]};
size_t select {}; // Index for function selection
srand(static_cast<unsigned>(time(0))); // Seed random generator

// Select function from the pointer array at random
for(size_t i {} ; i < 10 ; ++i)
{
    select = random(fcount); // Generate random index 0 to fcount-1

#ifdef TESTINDEX
    std::cout << "Random number = " << select << std::endl;
    if((select >= fcount) || (select < 0))
    {
        std::cout << "Invalid array index = " << select << std::endl;
        return 1;
    }
#endif

    result = pfun[select](a, b); // Call random function
    cout << "result = " << result << endl;
}
result = pfun[1](pfun[0](a, b), pfun[2](a, b));
std::cout << "The product of the sum and the difference = " << result
    << std::endl;
}

```

Here's an example of the output:

```

Random number = 2 result = 5
Random number = 2 result = 5
Random number = 1 result = 50
Random number = 0 result = 15
Random number = 1 result = 50
Random number = 1 result = 50
Random number = 0 result = 15
Random number = 1 result = 50
Random number = 2 result = 5
Random number = 1 result = 50
The product of the sum and the difference = 75

```

In general, you should get something different. If you want to get the trace output for the functions in the namespace `fun`, you must uncomment the `#define` directive at the beginning of `functions.cpp`.

The `#include` directive for `functions.h` adds the prototypes for `sum()`, `product()`, and `difference()`. The functions are defined within the namespace `fun`. These functions are called in `main()` using a random index to select from the array of pointers to them. The index to the array of function pointers is produced by the `random()`. The Standard Library function `rand()` from `stdlib` that is called in `random()` generates a sequence of pseudo-random numbers of type `int` in the range 0 to `RAND_MAX`, where `RAND_MAX` is a symbol defined as an integer in the `cstdlib` header. You must initialize the sequence that `rand()` produces before the first `rand()` call by passing an unsigned

integer seed value to `srand()`. Each different seed value will typically result in a different integer sequence from successive `rand()` calls. The `time()` function that is declared in the `ctime` header returns the number of seconds since January 1, 1970 as an integer, so using this as the argument to `srand()` ensures that you get a different random sequence each time the program executes.

The range of values returned by `rand()` needs to be scaled to the range of index values you need. However, you cannot rely on `RAND_MAX` being a value that you can increment as type `int`. An expression such as `(count*rand())/(RAND_MAX+1)` to scale the values will not produce the correct result if `RAND_MAX` is the maximum in the range for type `int`. Adding 1 to the maximum in a signed integer range results in the minimum - the largest negative integer in the range. Even using the expression `(count*rand())/(RAND_MAX+1L)` may fail with implementations where type `long` has the same range as type `int`. Using unsigned `long` ensures that the result of adding 1 to `RAND_MAX` will always produce the correct result.

Defining the identifier `TESTINDEX` in `Ex10_03.cpp` switches on diagnostic output in `main()`. With `TESTINDEX` defined, the code to output diagnostic information in `main()` will be included in the source that is compiled. If you remove the `#define` directive, the trace code will not be included. The trace code checks to make sure you use a valid index for the array, `pfun`. Because you don't expect to generate invalid index values, you shouldn't get this output!

■ **Tip** It's easy to generate invalid index values and verify the diagnostic code works. To do this, the `random()` function must generate a number other than 0, 1, or 2. If you add 1 to the value produced in the return statement, you should get an illegal index value roughly 25 percent of the time.

If you define the `TESTFUNCTION` identifier in `functions.cpp`, you'll get trace output from each function. This is a convenient way of controlling whether or not the trace statements are compiled into the program. You can see how this works by looking at one of the functions that may be called, `product()`:

```
int fun::product(int x, int y)
{
    #ifdef TESTFUNCTION
        std::cout << "Function product called." << std::endl;
    #endif

    return x*y;
}
```

The output statement simply displays a message, each time the function is called, but the output statement will only be compiled if `TESTFUNCTION` has been defined. A `#define` directive for a preprocessing symbol such as `TESTFUNCTION` is local to the source file in which it appears, so each source file that requires `TESTFUNCTION` to be defined needs to have its own `#define` directive. One way to manage this is to put all your directives that control trace and other debug output into a separate header file. You can then include this into all your `.cpp` files. In this way, you can alter the kind of debug output you get by making adjustments to this one header file.

Of course, diagnostic code is only included while you are testing the program. Once you think the program works, you quite sensibly leave it out. Therefore, you need to be clear that this sort of code is no substitute for error detection and recovery code that deals with unfortunate situations arising in your fully tested program (as they most certainly will).

■ **Note** The `rand()` function in the `stdlib` header does not generate random numbers that have satisfactory properties for general use. I recommend that you investigate the functions provided by the `random` Standard Library header when you need random numbers in an application. The details of the extensive random number generation capabilities provided by the `random` header are outside the scope of this book.

Using the assert() Macro

The `assert()` preprocessor macro is defined in the library header `cassert`. This enables you to test logical expressions in your program. Including a line of the form `assert(expression)` results in code that causes the program to be terminated with a diagnostic message if `expression` evaluates to false. I can demonstrate this with a simple example:

```
// Ex10_04.cpp
// Demonstrating assertions
#include <iostream>
#include <cassert>
int main()
{
    int y {5};

    for(int x {} ; x < 20 ; ++x)
    {
        std::cout << "x = " << x << " y = " << y << std::endl;
        assert(x<y);
    }
}
```

You should see an assertion message in the output when the value of `x` reaches 5. The program is terminated by the `assert()` macro by calling `abort()` when `x<y` evaluates to false. The `abort()` function is from the Standard Library, and its effect is to terminate the program immediately. As you can see from the output, this happens when `x` reaches the value 5. The macro displays the output on the standard error stream, `cerr`, which is always the command line. The message contains the condition that failed, and also the file name and line number in which the failure occurred. This is particularly useful with multi-file programs, where the source of the error is pinpointed exactly.

Assertions are often used for critical conditions in a program where, if certain conditions are not met, disaster will surely ensue. You would want to be sure that the program wouldn't continue if such errors arise. You can use any logical expression as the argument to the `assert()` macro, so you have a lot of flexibility.

`Ex10_03` generates index values using a random number generator so it contains exactly this kind of situation. With this technique, you always have the possibility of a bug resulting in an invalid index and if the index is outside the limits of the `pfun` array, the result is pretty much guaranteed to be catastrophic. You could use the `assert()` statement to verify the validity of the index value instead of the `#ifdef` block, you can simply write this statement:

```
assert((select >= 0) && (select < fcount));
```

Using `assert()` is simple and effective and when things go wrong, it provides sufficient information to pin down where the program has terminated.

Switching Off assert() Macros

You can switch off the preprocessor assertion mechanism when you recompile the program by defining `NDEBUG` at the beginning of the program file:

```
#define NDEBUG
```

This causes all assertions in the translation unit to be ignored. If you add this `#define` at the beginning of `Ex10_04.cpp`, you'll get output for all values of `x` from 0 to 19, and no diagnostic message. Note that this directive is only effective if it's placed before the `#include` statement for `cassert`.

■ **Caution** `assert()` is for detecting programming errors, not for handling errors at runtime. Evaluation of the logical expression shouldn't cause side effects or be based on something beyond the programmer's control (such as whether or not opening a file succeeds). Your program should include code to handle all error conditions that might be expected to occur occasionally.

Static Assertions

Static assertions are part of the C++ language, and are nothing to do with the preprocessor and the `assert()` macro, which is why I'm introducing them here! Static assertions are for checking conditions at compile time. A static assertion is a statement of the form:

```
static_assert(constant_expression, error_message);
```

`static_assert` is a keyword. `constant_expression` must produce a result at compile time that can be converted to type `bool`. `error_message` is a string literal that is output as an error message by the compiler when `constant_expression` is false. When `constant_expression` is true, the statement does nothing.

A common use for static assertions is in template definitions to verify the characteristics of a type parameter. A static assertion typically uses a template that is defined in the `type_traits` Standard Library header for testing for a type or class of types. Suppose that you define a function template for computing the average of a vector of elements of type `T`. Clearly, this is an arithmetic operation so you want to be sure the template cannot be used with vectors of non-numeric types. A static assertion can do that:

```
// average.h
#ifndef AVERAGE_H
#define AVERAGE_H

#include <type_traits>
#include <vector>

template<class T>
T average(const std::vector<T>& values)
{
    static_assert(std::is_arithmetic<T>::value,
                  "Type parameter for average() must be arithmetic.");
    T sum {};
    for(auto& value : values)
        sum += value;
    return sum/values.size();
}
#endif
```

The function template sums the elements in the vector that is the argument and divides by the number of elements. The static assertion uses the `is_arithmetic<T>` template from the `type_traits` header. The `value` member of the `is_arithmetic<T>` template will be true if `T` is an arithmetic type and false otherwise. It will be false when the compiler processes the `average<T>()` template used with a non-arithmetic type; in this case compilation will

fail and the error message will be displayed. An arithmetic type is any floating-point type or any integral type. The following will demonstrate this:

```
// Ex10_05.cpp
// Using a static assertion
#include <vector>
#include <iostream>
#include <string>
#include "average.h"

int main()
{
    std::vector<double> data {1.5, 2.5, 3.5, 4.5};
    std::cout << "The average of data values is " << average(data) << std::endl;

    // Uncomment the next two lines for a compiler error...
    // std::vector<std::string> words {"this", "that", "them", "those"};
    // std::cout << "The average of words values is " << average(words) << std::endl;
}
```

The `type_traits` header contains a large number of type testing templates including `is_integral<T>`, `is_signed<T>`, `is_unsigned<T>`, `is_floating_point<T>`, and `is_enum<T>`. Each of these has a value member that will be true if `T` conforms to the type and false otherwise. There are many other useful templates in the `type_traits` header and it is well worth exploring the contents further, especially once you have learned about classes.

Summary

This chapter has discussed capabilities that operate between, within, and across program files. C++ programs typically consist of many files, and the larger the program, the more files you have to contend with. It's vital that you really understand namespaces, preprocessing, and debugging techniques if you are to develop real-world C++ programs.

The important points from this chapter include:

- Each entity in a program must have only one definition.
- A name can have internal linkage, meaning that the name is accessible throughout a translation unit; external linkage, meaning that the name is accessible from any translation unit; or it can have no linkage, meaning that the name is only accessible in the block in which it is defined.
- You use header files to contain definitions and declarations required by your source files. A header file can contain template and type definitions, enumerations, constants, function declarations, inline function definitions, and named namespaces. By convention, header files use file names with the extension `.h`.
- Your source files will contain function definitions and global variables. A C++ source file usually has the file name extension `.cpp`.
- You insert the contents of a header file into a `.cpp` files by using an `#include` directive.
- A `.cpp` file is the basis for a translation unit that is processed by the compiler to generate an object file.
- A namespace defines a scope; all names declared within this scope have the namespace name attached to them. All declarations of names that are not in an explicit namespace scope are in the global namespace.

- A single namespace can be made up of several separate namespace declarations with the same name.
- Identical names that are declared within different namespaces are distinct.
- To refer to an identifier that is declared within a namespace from outside the namespace, you need to specify the namespace name and the identifier, separated by the scope resolution operator, `::`.
- Names declared within a namespace can be used without qualification from inside the namespace.
- The preprocessing phase executes directives to transform the source code in a translation unit prior to compilation. When all directives have been processed, the translation unit will only contain C++ code, with no preprocessing directives remaining.
- You can use conditional preprocessing directives to ensure that the contents of a header file are never duplicated within a translation unit.
- You can use conditional preprocessing directives to control whether trace or other diagnostic debug code is included in your program.
- The `assert()` macro enables you to test logical conditions during execution and issue a message and abort the program if the logical condition is false.
- You can use `static_assert` to check type arguments for template parameters in a template instance to ensure that a type argument is consistent with the template definition.

EXERCISES

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck after that, you can download the solutions from the Apress website (www.apress.com/source-code), but that really should be a last resort.

Exercise 10-1. Write a program that calls two functions, `print_this(const string& s)` and `print_that(const string& s)`, each of which calls a third function, `print(const string& s)`, to print the string that is passed to it. Define each function and `main()` in separate source files, and create three header files to contain the prototypes for `print_this()`, `print_that()`, and `print()`. Make sure that the header files are guarded against being included more than once.

Exercise 10-2. Modify the program from Exercise 10-1 so that `print()` uses a global integer variable to count the number of times it has been called. Output the value of this variable in `main()` after calls to `print_this()` and `print_that()`.

Exercise 10-3. In the `print.h` header file from Exercise 10-2, delete the existing prototype for `print()`, and instead create two namespaces, `print1` and `print2`, each of which contains a `print(const string& s)` function. Implement both functions in the `print.cpp` file so that they print the namespace name and the string. Change `print_this()` so that it calls `print()` defined in the `print1` namespace, and change `print_that()` to call the version in the `print2` namespace. Run the program, and verify that the correct functions are called.

Exercise 10-4. Modify the `main()` function from the previous exercise so that `print_this()` is only called if a `DO_THIS` preprocessing identifier is defined. When this is not the case, `print_that()` should be called.



Defining Your Own Data Types

In this chapter, I'll introduce one of the most fundamental tools in the C++ programmer's toolbox: classes. I'll also present some ideas that are implicit in object-oriented programming and show how these are applied.

In this chapter you'll learn:

- What the basic principles in object-oriented programming are
- How you define a new data type as a class, and how you can create and use objects of a class type
- What class constructors are, and how you define them
- What the default constructor is, and how you can supply your own version
- What the default copy constructor is
- What a friend function is
- What privileges a friend class has
- What the pointer `this` is, and how and when you use it
- What `const` functions in a class are and how they are used
- What a class destructor is and when you should define it

Classes and Object-Oriented Programming

You define a new data type by defining a *class*, but before I get into the language, syntax, and programming techniques of classes, I'll explain how your existing knowledge relates to the concept of object-oriented programming. Almost everything you have seen up to now has been *procedural programming*, which involves programming a solution in terms of fundamental data types. The essence of *object-oriented programming* (commonly abbreviated to *OOP*) is that you write programs in terms of *objects* in the domain of the problem you are trying to solve, so part of the program development process involves designing a set of types to suit the problem context. If you're writing a program to keep track of your bank account, you'll probably need to have data types such as `Account` and `Transaction`. For a program to analyze baseball scores, you may have types such as `Player` and `Team`. The variables of the fundamental types don't allow you to model real-world objects (or even imaginary objects) very well. It's not possible to model a baseball player realistically in terms of just an `int` or `double`, value or any other fundamental data type. You need several values of a variety of types for any meaningful representation of a baseball player.

Classes provide a solution. A class type can be a composite of variables of other types—of fundamental types or of other class types. A class can also have functions as an integral part of its definition. You could define a class type called `Box` that contains variables that store a length, a width, and a height to represent boxes. You could then define variables of type `Box`, just as you define variables of fundamental types. Each `Box` object would contain its own length, width and height dimensions and you could create and manipulate as many `Box` objects as you need in a program.

This goes quite a long way toward making programming in terms of real-world objects possible. Obviously, you can apply this idea of a class to represent a baseball player, or a bank account, or anything else. You can use classes to model whatever kinds of objects you want and write your programs around them. So, that's object-oriented programming all wrapped up then?

Well, not quite. A class as I've defined it up to now is a big step forward, but there's more to it than that. As well as the notion of user-defined types, object-oriented programming incorporates some additional important ideas (famously *encapsulation* and *data hiding*, *inheritance*, and *polymorphism*). I'll give you a rough, intuitive idea of what these additional OOP concepts mean right now. This will provide a reference frame for the detailed programming you'll be getting into in this and the next three chapters.

Encapsulation

In general, the definition of an object of a given type requires a combination of a specific number of different properties—the properties that make the object what it is. An object contains a precise set of data values that describe the object in sufficient detail for your needs. For a box, it could be just the three dimensions: length, width, and height. For an aircraft carrier, it is likely to be much more. An object can also contain a set of functions that operate on it—functions that use or change the properties for example or provide further characteristics of an object such as the volume of a box. The functions in a class define the set of operations that can be applied to an object of the class type: what you can do with it—or to it. Every object of a given type incorporates the same combination of things: the set of data values as *data members* of the class that characterize an object, and the set of operations as *function members* of the class. This packaging of data values and functions within an object is referred to as *encapsulation*. Figure 11-1 illustrates this with the example of an object that represents a loan account with a bank.

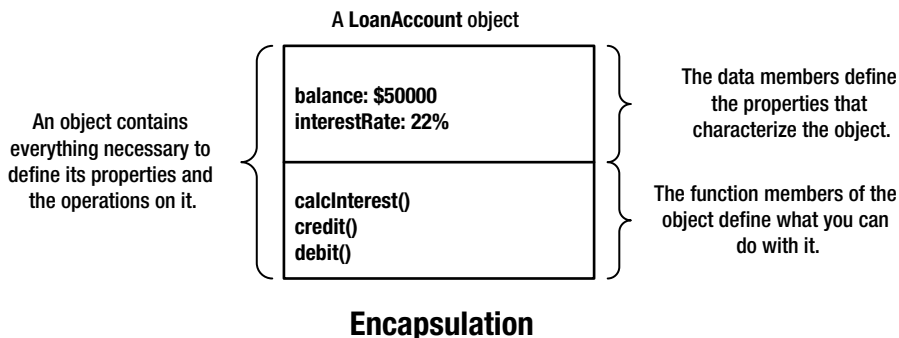


Figure 11-1. An example of encapsulation

Every LoanAccount object has its properties defined by the same set of data members; in this case, one holds the outstanding balance and the other holds the interest rate. Each object also contains a set of function members that define operations on the object; the one shown in Figure 12-1 calculates interest and adds it to the balance. The properties and operations are all encapsulated in every object of the type LoanAccount. Of course, this choice of what makes up a LoanAccount object is arbitrary. You might define it quite differently for your purposes, but however you define the LoanAccount type, all the properties and operations that you specify are encapsulated within every object of the type.

Note that I said earlier that the data values defining an object needed to be “sufficient for your needs,” not “sufficient to define the object in general.” A person could be defined very simply—perhaps just by the name, address, and phone number if you were writing an address-book application. A person as a company employee or as a medical patient is likely to be defined by many more properties and many more operations would be required. You just decide what you need in the contexts in which you intend to use the object.

Data Hiding

Of course, the bank wouldn't want the balance for a loan account (or the interest rate for that matter) changed arbitrarily from outside an object, as you were able to do with your structure objects in the Chapter 11. To permit this would be a recipe for chaos. Ideally, the data members of a `LoanAccount` object are protected from direct outside interference, and are only modifiable in a controlled way. The ability to make the data values for an object generally inaccessible is called *data hiding*. Figure 11-2 shows data hiding applied to a `LoanAccount` object.

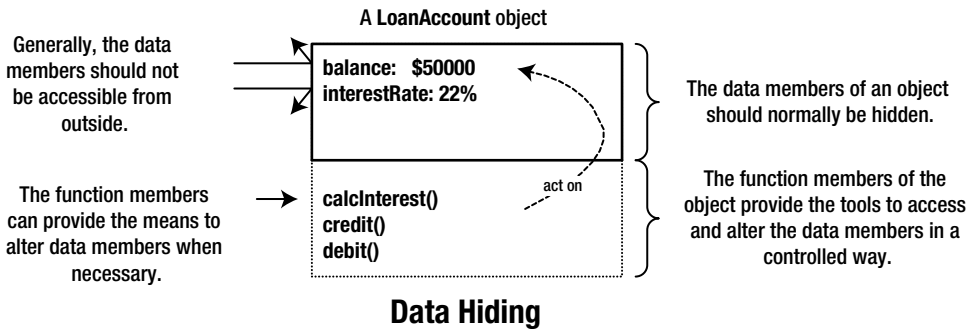


Figure 11-2. An example of data hiding

With a `LoanAccount` object, the function members of the object can provide a mechanism that ensures any changes to the data members follow a particular policy, and that the values set are appropriate. Interest shouldn't be negative, for instance, and generally, the balance should reflect the fact that money is owed to the bank, and not the reverse.

Data hiding is important because it is necessary if you are to maintain the integrity of an object. If an object is supposed to represent a duck, it should not have four legs; the way to enforce this is to make the leg count inaccessible—to “hide” the data. Of course, an object may have data values that can legitimately vary, but even then you often want to control the range; after all, a duck doesn't usually weigh 300 pounds. Hiding the data belonging to an object prevents it from being accessed directly, but you can provide access through functions that are members of the object, either to alter a data value in a controlled way, or simply to obtain its value. Such functions can check that the change they're being asked to make is legal and within prescribed limits where necessary.

Hiding the data within an object is not mandatory, but it's generally a good idea for at least a couple of reasons. First, as I said, maintaining the integrity of an object requires control of how changes are made. Second, direct access to the values that define an object undermines the whole idea of object-oriented programming. Object-oriented programming is supposed to be programming in terms of *objects*, not in terms of the bits that make up an object.

You can think of the data members as representing the *state* of the object, and the function members' functions that manipulate them as representing the object's *interface* to the outside world. Using the class then involves programming using the functions declared as the interface. A program using the class interface is only dependent on the function names, parameter types, and return types specified for the interface. The internal mechanics of these functions don't affect the program that is creating and using objects of the class. That means it's important to get the class interface right at the design stage—you can subsequently change the implementation to your heart's content without necessitating any changes to programs that use the class.

Inheritance

Inheritance is the ability to define one type in terms of another. For example, suppose you have defined a `BankAccount` type that contains members that deal with the broad issues of bank accounts. Inheritance allows you to create the `LoanAccount` type as a specialized kind of `BankAccount`. You could define a `LoanAccount` as being like a `BankAccount`, but with a few extra properties and functions of its own. The `LoanAccount` type *inherits* all the members of `BankAccount`, which is referred to as its *base class*. In this case, you'd say that `LoanAccount` is *derived* from `BankAccount`.

Each `LoanAccount` object contains all the members that a `BankAccount` object does, but it has the option of defining new members of its own, or of *redefining* the functions it inherits so that they are more meaningful in its context. This last ability is very powerful, as you'll see.

Extending the current example, you might also want to create a new `CheckingAccount` type by adding different characteristics to `BankAccount`. This situation is illustrated in Figure 11-3.

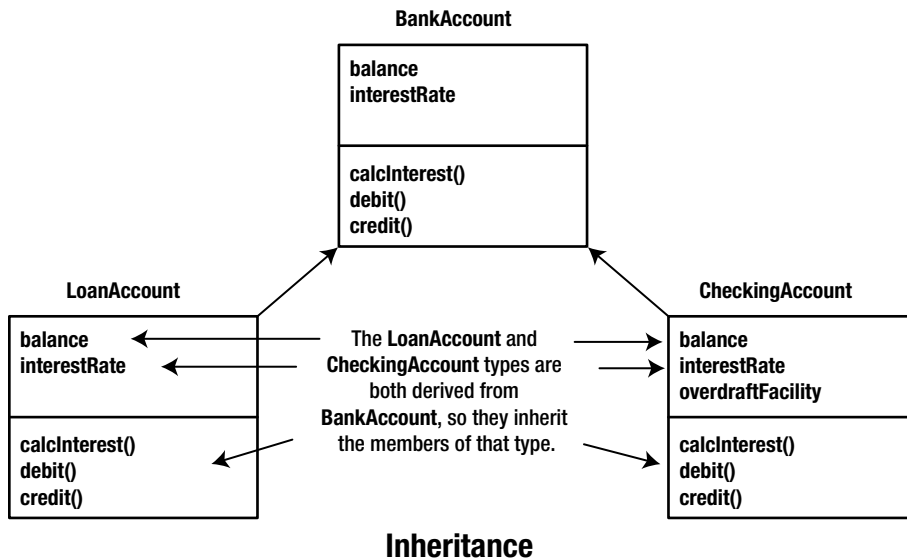


Figure 11-3. An example of inheritance

Both of the `LoanAccount` and `CheckingAccount` types are defined so that they are derived from the type `BankAccount`. They inherit the data members and function members of `BankAccount`, but they are free to define new characteristics that are specific to their own type.

In this example, `CheckingAccount` has added a data member called `overdraftFacility` that is unique to itself, and both the derived classes can redefine any of the function member that they inherit from the base class. It's likely they would redefine `calcInterest()` for example because calculating and dealing with the interest for a checking account involves something rather different than doing it for a loan account.

Polymorphism

Polymorphism means the ability to assume different forms at different times. Polymorphism in C++ always involves calling a function member of an object, using either a pointer or a reference. Such function calls can have different effects at different times—sort of Jekyll and Hyde function calls. The mechanism only works for objects of types that are derived from a common base type, such as the `BankAccount` type. Polymorphism means that objects belonging to a “family” of inheritance-related classes can be passed around and operated on using base class pointers and references.

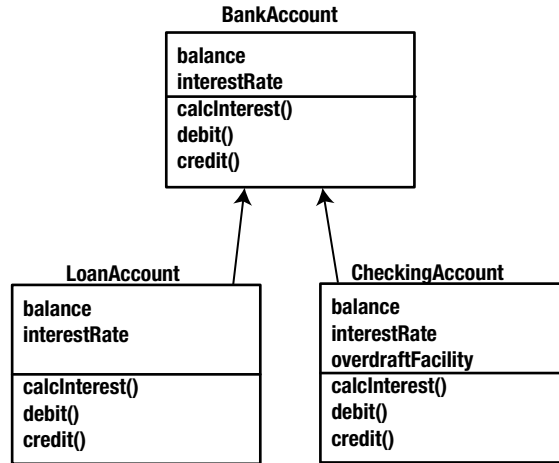
The `LoanAccount` and `CheckingAccount` objects can both be passed around using a pointer or reference to `BankAccount`. The pointer or reference can be used to call the inherited function members of whatever object it refers to. The idea and implications of this will be easier to appreciate if I take a specific case.

Suppose you have the `LoanAccount` and `CheckingAccount` types defined as before, based on the `BankAccount` type. Suppose further that you have defined objects of these types, `debt` and `cash` respectively, as illustrated in Figure 11-4. Because both types are based on the `BankAccount` type, a variable of type *pointer to BankAccount*, such as `pAcc` in Figure 11-4, can store the address of either of these objects.

```
BankAccount* pAcc;    // Pointer to base class
LoanAccount debt;
CheckingAccount cash;
```

```
pAcc = &cash;         // Points to check a/c
pAcc->calcInterest(); // Adds interest
```

```
pAcc = &debt;         // Points to loan a/c
pAcc->calcInterest(); // Debits interest
```



Polymorphism

Figure 11-4. An example of polymorphism

The beauty of polymorphism is that the function called by `pAcc->calcInterest()` varies depending on what `pAcc` points to. If it points to a `LoanAccount` object, then the `calcInterest()` function for that object is called and interest is debited from the account. If it points to a `CheckingAccount` object, the result is different because the `calcInterest()` function for that object is called and interest is credited to the account. The particular function that is called through the pointer is decided at runtime, not when the program is compiled, but when it executes. Thus, the same function call can do different things depending on what kind of object the pointer points to. Figure 11-4 shows just two different types, but in general, you can get polymorphic behavior with as many different types derived from a common base class as your application requires. You need quite a bit of C++ language know-how to accomplish what I've described, and that's exactly what you'll be exploring in the rest of this chapter and throughout the next three chapters.

Terminology

Here's a summary of the terminology that I'll be using when I'm discussing classes. It includes some terms that you've come across already:

- A *class* is a user-defined data type.
- The variables and functions defined within a class are *members* of the class. The variables are *data members* and the functions are *function members*. The function members of a class are sometimes referred to as *methods*.
- Variables of a class type store *objects*. Objects are sometimes called *instances* of the class.
- Defining an instance of a class is referred to as *instantiation*.
- *Object-oriented programming* is a programming style based on the idea of defining your own data types as classes. It involves the ideas of *encapsulation* of data, class *inheritance*, and *polymorphism*, which I've just discussed.

When you get into the detail of object-oriented programming, it may seem a little complicated in places. Getting back to the basics can often help make things clearer; so use this list to always keep in mind what objects are really about. Object-oriented programming is about writing programs in terms of the objects that are specific to the domain of your problem. All the facilities around classes are there to make this as comprehensive and flexible as possible.

Defining a Class

A class is a user-defined type. The definition of a type uses the `class` keyword. The basic organization of a class definition looks like this:

```
class ClassName
{
    // Code that defines the members of the class...
};
```

The name of this class type is `ClassName`. It's a common convention to use the uppercase name for user-defined classes to distinguish class types from variable names. I'll adopt this convention in the examples. The members of the class are all specified between the braces. The definitions for function members can be inside or outside the class definition. If the definition of a function member is outside the class, the member name in the definition must be qualified by the class name. Note that the semicolon after the closing brace for the class definition must be present.

All the members of a class are `private` by default, which means they cannot be accessed from outside the class. This is obviously not acceptable for the function members that form the interface. You use the `public` keyword followed by a colon to make all subsequent members accessible from outside the class. Members specified after the `private` keyword are not accessible from outside the class. `public` and `private` are *access specifiers* for the class members. There's another access specifier, `protected`, that you'll meet later. Here's how an outline class looks with access specifiers:

```
class ClassName
{
    private:
    // Code that specifies members that are not accessible from outside the class...

    public:
    // Code that specifies members that are accessible from outside the class...
};
```

`public` and `private` precede a sequence of members that are or are not accessible outside the class. The specification of `public` or `private` applies to all members that follow until there is a different specification. You could omit the first `private` specification here and get the default status of `private`, but it's better to make it explicit. Members in a `private` section of a class can only be accessed from functions that are members of the same class. Data members or function members that need to be accessed by a function that is not a member of the class must be specified as `public`. A function member can reference any other member of the same class, regardless of the access specification, by just using its name. To make all this generality clearer, let's start with an example of defining a class to represent a box:

```
class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};
```

```
public:
    // Function to calculate the volume of a box
    double volume()
    {
        return length*width*height;
    }
};
```

`length`, `width`, and `height` are data members of the `Box` class and are all of type `double`. They are also `private` because they are preceded by the `private` access specification and therefore cannot be accessed from outside the class. Only the public `volume()` function member can refer to these private members. Each of the data members is initialized to 1 because a zero dimension for a box would not make sense. You don't have to initialize data members in this way—there are other ways of setting their values as you'll see in the next section. If their values are not set by some mechanism though, they will contain junk values.

In general, you can repeat any of the access specifiers in a class definition as many times as you want. This enables you to place data members and function members in separate groups within the class definition, each with their own access specifier. It can be easier to see the internal structure of a class definition if you arrange to group the data members and the function members separately, according to their access specifiers.

Every `Box` object will have its own set of data members. This is obvious really—if they didn't have their own data members, all objects would be identical. You could create a variable of type `Box` like this:

```
Box myBox;                                // A Box object with all dimensions 1
```

The `myBox` variable refers to a `Box` object with the default data member values. You could call the `volume()` member for the object to calculate the volume:

```
std::cout << "Volume of myBox is" << myBox.volume() << std::endl; // Volume is 1.0
```

Of course the volume will be 1 because the initial values for the three dimensions are 1. The fact that the data members of the `Box` class are `private` means that we have no way to set these members. You could specify the data members as `public`, in which case you can set them explicitly from outside the class, like this:

```
myBox.length = 1.5;
myBox.width = 2.0;
myBox.height = 4.0;
std::cout << "Volume of myBox is" << myBox.volume() << std::endl; // Volume is 12.0
```

I said earlier that it's not good practice in general to make data members `public`. To set the values of `private` data members when an object is created, you must add a `public` function member of a special kind to the class, called a *constructor*. Objects of a class type can only be created using a constructor.

■ **Note** C++ also includes the ability to define a *structure* that is similar to a class and defines a type. The structure originated in C. You define a structure in essentially the same way as a class but using the `struct` keyword instead of the `class` keyword. In contrast to members of a class, the members of a structure are `public` by default. Structures are still used frequently in C++ programs to define types that represent simple aggregates of several variables of different types—the margin sizes and dimensions of a printed page for example. I won't discuss structures as a separate topic because aside from the default access specification and the use of the `struct` keyword, you define a structure in exactly the same way as a class.

Constructors

A class *constructor* is a special kind of function in a class that differs in significant respects from an ordinary function member. A constructor is called whenever a new instance of the class is defined. It provides the opportunity to initialize the new object as it is created and to ensure that data members contain valid values. A class constructor always has the same name as the class. `Box()`, for example, is a constructor for the `Box` class. A constructor does not return a value and therefore has no return type. It is an error to specify a return type for a constructor.

Just a moment! I hear you cry. We created a `Box` object in the previous section—and calculated its volume. How did that happen when there was no constructor defined? Well, there's no such thing as a class with no constructors. If you don't define a constructor for a class, the compiler will supply a *default constructor*. The `Box` class really looks like this:

```
class Box
{
private:
    double length {1};
    double width {1};
    double height {1};

public:
    // The default constructor that is supplied by the compiler...
    Box()
    {
        // Empty body so it does nothing...
    }

    // Function to calculate the volume of a box
    double volume()
    {
        return length*width*height;
    }
};
```

The default constructor has no parameters and its sole purpose is to allow an object to be created. It does nothing else so the data members will have their default values. If there are no initial values specified for data members, they will contain junk values. Note that when you do define a constructor, the default constructor is not supplied. There are circumstances in which you need a constructor with no parameters in addition to a constructor that you define that has parameters. In this case *you* must ensure that there is a definition for the no-arg constructor in the class.

■ **Note** You'll see later that there is even more to the `Box` class that is provided by default than I've shown here.

Let's extend the Box class from the previous example to incorporate a constructor and then check that it works:

```
// Ex11_01.cpp
// Defining a class constructor
#include <iostream>

// Class to represent a box
class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructor
    Box(double lengthValue, double widthValue, double heightValue)
    {
        std::cout << "Box constructor called." << std::endl;
        length = lengthValue;
        width = widthValue;
        height = heightValue;
    }

    // Function to calculate the volume of a box
    double volume()
    {
        return length*width*height;
    }
};

int main()
{
    Box firstBox {80.0, 50.0, 40.0};           // Create a box
    double firstBoxVolume {firstBox.volume()}; // Calculate the box volume
    std::cout << "Volume of Box object is" << firstBoxVolume << std::endl;
}
```

This produces the following output:

```
Box constructor called.
Volume of Box object is 160000
```

The constructor for the Box class has three parameters of type double, corresponding to the initial values for the length, width, and height members of an object. No return type is allowed and the name of the constructor must be the same as the class name, Box. The first statement in the constructor body outputs a message to show when it's called. You wouldn't do this in production programs but it's helpful when you're testing a program and to understand what's happening and when. I'll use it regularly to trace what is happening in the examples. The rest of the code in the body of the constructor assigns the arguments to the corresponding data members. You could include checks that look for valid, nonnegative arguments that are the dimensions of a box. In the context of a real application, you'd probably want to do this, but here you only need to learn how a constructor works so I'll keep it simple for now.

The `firstBox` object is created with this statement:

```
Box firstBox {80.0, 50.0, 40.0};
```

The initial values for the data members, `length`, `width`, and `height`, appear in the initializer list and are passed as arguments to the constructor. Because there are three values in the list, the compiler looks for a `Box` constructor with three parameters. When the constructor is called, it displays the message that appears as the first line of output, so you know that the constructor that you have added to the class is called.

I said earlier that if you define a constructor, the compiler won't supply a default constructor. This means that this statement won't compile:

```
Box box1;                                // Causes a compiler error message
```

This object would have the default dimensions. If you want to allow `Box` objects to be defined like this, you must add a definition for a constructor without arguments:

Defining Constructors Outside the Class

I said earlier that the definition of a function member can be placed outside the class definition. This is also true for class constructors. I can define the `Box` class in a header file like this:

```
// Box.h
#ifndef BOX_H
#define BOX_H

class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructors
    Box(double lengthValue, double widthValue, double heightValue);
    Box();                                // No-arg constructor

    double volume();                      // Function to calculate the volume of a box
};

#endif
```

The definitions for the `volume()` member and the constructor must go in a `.cpp` file. The name of each function member and constructor in the source must be qualified with the class name so the compiler knows to which class they belong:

```
// Box.cpp
#include <iostream>
#include "Box.h"
```

```
// Constructor definition
Box::Box(double lengthValue, double widthValue, double heightValue)
{
    std::cout << "Box constructor called." << std::endl;
    length = lengthValue;
    width = widthValue;
    height = heightValue;
}

Box::Box() {} // No-arg constructor

// Function to calculate the volume of a box
double Box::volume()
{
    return length*width*height;
}
```

If `Box.h` was not included into `Box.cpp`, the compiler would not know that `Box` is a class so the code would not compile. Separating the definitions of classes from the definitions of their function members makes the code easier to manage. Large class with lots of function members and constructors would be very cumbersome if all the function definitions appeared within the class. Any source file that creates objects of type `Box` just needs to include the header file `Box.h`. A programmer using this class doesn't need access to the source code definitions of the function members, only to the class definition in the header file. As long as the *class* definition remains fixed, you're free to change the implementations of the function members without affecting the operation of programs that use the class.

Defining a function member outside a class is not quite the same as placing the definition inside the class. Function definitions *within* a class definition are implicitly `inline`. (This doesn't necessarily mean that they will be *implemented* as inline functions—the compiler still decides that, as I discussed in Chapter 8).

The previous example would look like this with the `Box` class split into `.h` and `.cpp` files:

```
// Ex11_01A.cpp
// Defining a class constructor
#include <iostream>
#include "Box.h"

int main()
{
    Box firstBox {80.0, 50.0, 40.0}; // Create a box
    double firstBoxVolume{firstBox.volume()}; // Calculate the box volume
    std::cout << "Volume of Box object is" << firstBoxVolume << std::endl;
}
```

This is the same version of `main()` as in the previous example. The only difference is the `#include` directive for the `Box.h` header file that contains the definition of the `Box` class.

Default Constructor Parameter Values

When I discussed “ordinary” functions, you saw that you can specify *default values* for the parameters in the function prototype. You can do this for class function members, including constructors. Default parameter values for constructors and function members always go inside the class, not in an external constructor or function definition. I can change the class definition in the previous example to the following:

```
class Box
{
private:
    double length;
    double width;
    double height;

public:
    // Constructors
    Box(double lv = 1.0, double wv = 1.0, double hv = 1.0);
    Box(); // No-arg constructor

    double volume(); // Function to calculate the volume of a box
};
```

If you make this change to the last example, what happens? You get an error message from the compiler of course! The message basically says that you have multiple default constructors defined. The reason for the confusion is the constructor with three parameters allows all three arguments to be omitted, which is indistinguishable from a call to the no-arg constructor. The obvious solution is to get rid of the constructor that accepts no parameters in this instance. If you do so, everything compiles and executes OK. However, don’t assume that this is always the best way to implement the default constructor.

Using a Constructor Initialization List

So far, you’ve set values for data members the body of a constructor using explicit assignment. You can use an alternative and more efficient technique that uses *a constructor initialization list*. I’ll illustrate this with an alternative version of the Box class constructor:

```
// Constructor definition using an initializer list
Box::Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv}
{
    std::cout << "Box constructor called." << std::endl;
}
```

The values of the data members are specified as initializing values in the initialization list that is part of the constructor header. `length` is initialized with `lv`, for example. The initialization list is separated from the parameter list by a colon (:), and each initializer is separated from the next by a comma (,). This is more than just a different notation. When you initialize a data member using an assignment statement in the body of the constructor, the data member is first created (using a constructor call if it is an instance of a class) after which the assignment is carried out as a separate operation. When you use an initialization list, the initial value is used to initialize the data member *as it is created*. This can be a much more efficient process, particularly if the data member is a class instance. If you substitute this version of the constructor in the previous example, you’ll see that it works just as well. This technique for initializing parameters in a constructor is important for another reason. As you’ll see, it is the *only* way of setting values for certain types of data members.

Use of the explicit Keyword

A problem with class constructors that have a *single* parameter is that the compiler can use such a constructor as an implicit conversion from the type of the parameter to the class type. This can produce undesirable results in some circumstances. Let's consider a particular situation. Suppose that you define a class that defines boxes that are cubes with all the sides have the same length:

```
// Cube.h
#ifndef CUBE_H
#define CUBE_H
class Cube
{
public:
    double side;

    Cube(double side);           // Constructor
    double volume();            // Calculate volume of a cube
    bool compareVolume(Cube aCube); // Compare volume of a cube with another
};
#endif
```

You can define the constructor in `Cube.cpp` as:

```
Cube::Cube(double len) : side {len} { std::cout << "Cube constructor called." << std::endl; }
```

The definition of function that calculates the volume will be:

```
double Cube::volume() { return side*side*side; }
```

The `compareVolume()` member can be defined as:

```
bool Cube::compareVolume(Cube aCube) { return volume() > aCube.volume(); }
```

One `Cube` object is greater than another if its volume is the greater of the two.

The constructor requires only one argument of type `double`. Clearly, the compiler could use the constructor to convert a `double` value to a `Cube` object, but under what circumstances is that likely to happen?

The class defines a `volume()` function and a function to compare the current object with another `Cube` object passed as an argument, which returns true if the current object has the greater volume. You might use the `Cube` class in the following way:

```
// Ex11_02.cpp
// Problems of implicit object conversions
#include <iostream>
#include "Cube.h"

int main()
{
    Cube box1 {7.0};
    Cube box2 {3.0};
    if(box1.compareVolume(box2))
        std::cout << "box1 is larger than box2." << std::endl;
```



```

else
    std::cout << "box1 is less than or equal to box2." << std::endl;

std::cout << "volume of box1 is" << box1.volume() << std::endl;
if(box1.compareVolume(50.0))
    std::cout << "Volume of box1 is greater than 50" << std::endl;
else
    std::cout << "Volume of box1 is less than or equal to 50" << std::endl;
}

```

Here's the output:

```

Cube constructor called.
Cube constructor called.
box1 is larger than box2.
volume of box1 is 343
Cube constructor called.
Volume of box1 is less than or equal to 50

```

The output shows that the volume of box1 is definitely not less than 50 but the last line of output indicates the opposite. The code presumes that `compareVolume()` compares the volume of the current object with 50.0. In reality the function compares two `Cube` objects. The compiler knows that the argument to the `compareVolume()` function should be a `Cube` object, but it compiles this quite happily because a constructor is available that converts the argument 50.0 to a `Cube` object. The code the compiler produces is equivalent to:

```

if(box1.compareVolume(Cube {50.0}))
    std::cout << "Volume of box1 is greater than 50" << std::endl;
else
    std::cout << "Volume of box1 is less than or equal to 50" << std::endl;

```

The function is not comparing the volume of the box1 object with 50.0, but with 125000.0, the volume of a `Cube` object with a side of length 50.0! The result is very different from what was expected.

Happily, you can prevent this nightmare from happening by declaring the constructor as `explicit`:

```

class Cube
{
public:
    double side;

    explicit Cube(double side);           // Constructor
    double volume();                     // Calculate volume of a cube
    bool compareVolume(Cube aCube);      // Compare volume of a cube with another
};

```

With this definition for `Cube`, `Ex11_02.cpp` will not compile. The compiler never uses a constructor declared as `explicit` for an implicit conversion; it can only be used explicitly in the program. By using the `explicit` keyword with constructors that have a single parameter you prevent implicit conversions from the parameter type to the class type. The `compareVolume()` member only accepts a `Cube` object as an argument so calling it with an argument of type `double` does not compile.

Delegating Constructors

A class can have several constructors that provide different ways of creating an object. The code for one constructor can call another of the same class in the initialization list. This can avoid repeating the same code in several constructors. Here's a simple illustration of this using the `Box` class:

```
class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructors
    Box(double lv, double wv, double hv);
    Box(double side);                // Constructor for a cube
    Box() {}                        // No-arg constructor

    double volume();                // Function to calculate the volume of a box
};
```

Notice that I have restored the initial values for the data members and removed the default values for the constructor parameters. This is because the compiler would not be able to distinguish between a call of the constructor with a single parameter and a call of the constructor with three parameters with the last two arguments omitted. This removes the capability for creating an object with no arguments and the compiler will not supply the default so I have added the definition of the no-arg constructor to the class.

The implementation of the first constructor can be:

```
Box::Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv}
{
    std::cout << "Box constructor 1 called." << std::endl;
}
```

The second constructor creates a `Box` object with all sides equal and we can implement it like this:

```
Box::Box(double side) : Box {side, side, side}
{
    std::cout << "Box constructor 2 called." << std::endl;
}
```

This constructor just calls the previous constructor in the initialization list. The `side` argument is used as all three values in the initializer list for the previous constructor. This is called a *delegating constructor* because it delegates the construction work to the other constructor. Delegating constructors help to shorten and simplify constructor code and can make the class definition easier to understand. Here's an example that exercises this:

```
// Ex11_03.cpp
// Using a delegating constructor
#include <iostream>
#include "Box.h"
```

```
int main()
{
    Box box1 {2.0, 3.0, 4.0};           // An arbitrary box
    Box box2 {5.0};                     // A box that is a cube
    std::cout << "box1 volume = " << box1.volume() << std::endl;
    std::cout << "box2 volume = " << box2.volume() << std::endl;
}
```

The complete code is in the download. The output is:

```
Box constructor 1 called.
Box constructor 1 called.
Box constructor 2 called.
box1 volume = 24
box2 volume = 125
```

You can see from the output that creating the first object just calls constructor 1. Creating the second object calls constructor 1 followed by constructor 2. This also shows that execution of the initialization list for a constructor occurs before the code in the body of the constructor. The volumes are as you would expect.

You should only call a constructor for the same class in the initialization list for a constructor. Calling a constructor of the same class in the body of a delegating constructor is not the same. Further you must not initialize data members in the initialization list of a delegating constructor. The code will not compile if you do. You can set values for data members in the body of a delegating constructor but in this case you should consider whether the constructor should really be implemented as a delegating constructor.

The Copy Constructor

Suppose you add the following statement to `main()` in `Ex11_03.cpp`:

```
Box box3 {box2};
std::cout << "box3 volume = " << box3.volume() << std::endl;    // Volume = 125
```

The output shows that `box3` does indeed have the dimensions of `box2` but there's no constructor defined with a parameter of type `Box` so how was `box3` created? The answer is that the compiler supplied a default *copy constructor*, which is a constructor that creates an object by copying an existing object. The default copy constructor copies the values of the data members of the object that is the argument to the new object. This is fine in the case of `Box` objects but it can cause problems when one or more data members are pointers. Just copying a pointer does not duplicate what it points to, which means that when an object is created by the copy constructor, it is interlinked with the original object. Both objects will contain a member pointing to the same thing. A simple example is if an object contains a pointer to a string. A duplicate object will have a member pointing to the same string so if the string is changed for one object, it will be changed for the other. This is not usually what you want. In this case *you* must define a copy constructor.

Implementing the Copy Constructor

The copy constructor must accept an argument of the same class type and create a duplicate in an appropriate manner. This poses an immediate problem that you must overcome; you can see it clearly if you try to define the copy constructor for the `Box` class like this:

```
Box::Box(Box box) : length {box.length}, width {box.width}, height {box.height} // Wrong!!
{}
```

Each data member of the new object is initialized with the value of the object that is the argument. No code is needed in the body of the copy constructor in this instance. This looks OK but consider what happens when the constructor is called. The argument is passed *by value*, but because the argument is a `Box` object the compiler arranges to call the copy constructor for the `Box` class to make a copy of the argument. Of course, the argument to this call of the copy constructor is passed by value, so another call to the copy constructor is required, and so on. In short, you've created a situation where an unlimited number of recursive calls to the copy constructor will occur. Your compiler won't allow this code to compile. To avoid the problem the parameter for the copy constructor must be a *reference*.

Reference Parameters

A copy constructor should be defined with a `const` reference parameter, so for the `Box` class it looks like this:

```
Box::Box(const Box& box) : length {box.length}, width {box.width}, height {box.height}
{}
```

Now the argument is no longer passed by value, so recursive calls of the copy constructor are avoided. The compiler initializes the parameter `box` with the object that is passed to it. The parameter should be `const` because a copy constructor is only in the business of creating duplicates; it should not modify the original. A `const` reference parameter allows `const` and non-`const` objects to be copied; if the parameter was not `const`, the constructor would not accept a `const` object as the argument. You can conclude from this that the parameter type for a copy constructor is *always* a `const` reference to an object of the same class type. In other words, the form of the copy constructor is the same for any class:

```
Type::Type(const Type& object)
{
    // Code to duplicate of object...
}
```

Of course the copy constructor may also have an initialization list. I'll return to the question of defining a copy constructor in the next chapter.

Accessing Private Class Members

Inhibiting all external access to the values of private data members of a class is rather extreme. It's a good idea to protect them from unauthorized modification, but if you don't know what the dimensions of a particular `Box` object are, you have no way to find out. Surely it doesn't need to be that secret?

It doesn't, and you don't need to expose the data members by using the `public` keyword. You can provide access to the values of private data members by adding function members to return their values. To provide access to the dimensions of a `Box` object from outside the class, you just need to add three functions to the class definition:

```
class Box
{
private:
    double length;
    double width;
    double height;

public:
    // Constructors
    Box(double lv = 1.0, double ww = 1.0, double hv = 1.0);

    double volume(); // Function to calculate the volume of a box

    // Functions to provide access to the values of data members
    double getLength() {return length;}
    double getWidth() {return width;}
    double getHeight() {return height;}
};
```

The values of the data members are fully accessible, but they can't be changed from outside the class so the integrity of the class is preserved without the secrecy. Functions of this kind usually have their definitions within the class because they are short, and this makes them `inline` by default. Consequently the overhead involved in accessing the value of a data member is minimal. Functions that retrieve the values of data members are often referred to as *accessor* functions.

Using these accessor functions is simple:

```
Box myBox {3.0, 4.0, 5.0};
std::cout << "myBox dimensions are" << myBox->getLength() << " by "
    << myBox->getWidth() << " by " << myBox->getHeight() << std::endl;
```

You can use this approach for any class. You just write an accessor function for each data member that you want to make available to the outside world.

There will be situations in which you *do* want to allow data members to be changed from outside the class. If you supply a function member to do this rather than exposing the data member directly, you have the opportunity to perform integrity checks on the value. For example, you could add a function to allow the height of a `Box` object to be changed:

```
class Box
{
private:
    double length;
    double width;
    double height;
```

```

public:
    // Constructors
    Box(double lv = 1.0, double wv = 1.0, double hv = 1.0);

    double volume();                                // Function to calculate the volume of a box

    // Functions to provide access to the values of data members
    double getLength() {return length;}
    double getWidth() {return width;}
    double getHeight() {return height;}

    // Functions to set data member values
    void setLength(double lv) { if(lv > 0) length = lv;}
    void setWidth(double wv) { if(wv > 0) width = wv;}
    void setHeight(double hv) { if(hv > 0) height = hv; }
};

```

The if statement in each set function ensures that you only accept new values that are positive. If a new value is supplied for a data member that is zero or negative, it will be ignored. Member functions that allow data members to be modified are often referred to as *mutators*.

Friends

Under normal circumstances, you'll hide the data members of your classes by declaring them as `private`. You may well have `private` function members of the class too. In spite of this, it is sometimes useful to treat selected functions that are not members of the class as “honorary members” and allow them to access non-public members of a class object. Such functions are called *friends* of the class. A friend can access any of the members of a class object, regardless of their access specification. The need for friend functions does not arise often, but you'll meet one circumstance where it can be necessary in the next chapter when you learn about operator overloading.

You need to consider two situations that involve friends: an individual function can be specified as a friend of a class, or a whole class can be specified as a friend of another class. In the latter case, all the function members of the friend class have the same access privileges as a normal member of the class. I'll consider individual functions as friends first.

The Friend Functions of a Class

To make a function a friend of a class, you must declare it as such within the class definition using the `friend` keyword. It's the class that determines its friends; there's no way to make a function a friend of a class from outside the class definition. A friend function can be a global function or it can be a member of another class. By definition a function can't be a friend of the class of which it is a member so access specifiers don't apply to the friends of a class.

The need for friend functions in practice is limited. They are useful in situations where a function needs access to the internals of two different kinds of objects; making the function a friend of both classes makes that possible. I will demonstrate how they work in simpler contexts that don't necessarily reflect a situation where they are required.

Suppose that you want to implement a friend function in the `Box` class to compute the surface area of a `Box` object. To make the function a friend, you must declare it as such within the `Box` class definition. Here's a version that does that:

```
class Box
{
private:
    double length;
    double width;
    double height;

public:
    // Constructors
    Box(double lv = 1.0, double wv = 1.0, double hv = 1.0);

    double volume();                // Function to calculate the volume of a box

    friend double surfaceArea(const Box& aBox);    // Friend function for the surface area
};
```

`Box.cpp` will contain the following code:

```
// Box.cpp
#include <iostream>
#include "Box.h"

// Constructor definition
Box::Box(double lv, double wv, double hv) : length(lv), width(wv), height(hv)
{
    std::cout << "Box constructor called." << std::endl;
}

// Function to calculate the volume of a box
double Box::volume()
{
    return length*width*height;
}
```

Here the code to try out the friend:

```
// Ex11_04.cpp
// Using a friend function of a class
#include <iostream>
#include <memory>
#include "Box.h"

int main()
{
    Box box1 {2.2, 1.1, 0.5};                // An arbitrary box
    Box box2;                                // A default box
    auto pBox3 = std::make_shared<Box>(15.0, 20.0, 8.0);    // Box on the heap
}
```

```

std::cout << "Volume of box1 = " << box1.volume() << std::endl;
std::cout << "Surface area of box1 = " << surfaceArea(box1) << std::endl;

std::cout << "Volume of box2 = " << box2.volume() << std::endl;
std::cout << "Surface area of box2 = " << surfaceArea(box2) << std::endl;

std::cout << "Volume of box3 = " << box3->volume() << std::endl;
std::cout << "Surface area of box3 = " << surfaceArea(*pBox3) << std::endl;
}

// friend function to calculate the surface area of a Box object
double surfaceArea(const Box& aBox)
{
    return 2.0*(aBox.length*aBox.width + aBox.length*aBox.height + aBox.height*aBox.width);
}

```

Here's the output:

```

Box constructor called.
Box constructor called.
Box constructor called.
Volume of box1 = 1.21
Surface area of box1 = 8.14
Volume of box2 = 1
Surface area of box2 = 6
Volume of box3 = 2400
Surface area of box3 = 1160

```

You declare the `boxSurface()` function as a friend of the `Box` class by writing the function prototype within the `Box` class definition preceded by the `friend` keyword. The function doesn't alter the `Box` object that is passed as the argument so it's sensible to use a `const` reference parameter specification. It's also a good idea to be consistent when placing the friend declaration within the definition of the class. You can see that I've chosen to position this declaration at the end of all the public members of the class. The rationale for this is that the function is part of the class interface because it has full access to all class members.

`boxSurface()` is a global function and its definition follows that of `main()`. You could put it in `Box.cpp` because it is related to the `Box` class, but placing it in the main file helps indicate that it's a global function.

Notice that you access the data members of the object within the definition of `boxSurface()` by using the `Box` object that is passed to the function as a parameter. A friend function is *not* a class member so the data members can't be referenced by their names alone. They each have to be qualified by an object name in exactly the same way as they would be in an ordinary function that accesses public members of a class. A friend function is the same as an ordinary function, except that it can access all the members of a class without restriction.

The `main()` function creates a `Box` object by specifying the dimensions, an object with no dimensions specified so the defaults will apply, and a `Box` object created on the heap. This shows that you can create a smart pointer to a `Box` object on the heap in the way that you have seen with `std::string` objects. From the output you can see that everything works as expected with all three objects.

Although this example demonstrates how you write a friend function, it is not very realistic. You could have used accessor function members to return the values of the data members. Then `surfaceArea()` wouldn't need to be a friend function. Perhaps the best option would have been to make `surfaceArea()` a public function member of the class so that the capability for computing the surface area of a box becomes part of the class interface.

Friend functions are part of the interface to a class, but it is better programming practice to define the interface to a class entirely in terms of function members if you can. As I explained at the beginning of this discussion, the only circumstances in which they are really necessary is when you need to access the non-public members of two different classes; even then, you may be able to do what you want without involving friend functions.

Friend Classes

You can declare a whole class to be a friend of another class. All the function members of a friend class have unrestricted access to all the members of the class of which it has been declared a friend.

For example, suppose you have defined a `Carton` class and want to allow the function members of the `Carton` class to have access to the members of the `Box` class. Including a statement in the `Box` class definition that declares `Carton` to be a friend will enable this:

```
class Box {
    // Public members of the class...

    friend class Carton;

    // Private members of the class...
};
```

Friendship is not a reciprocal arrangement. Functions in the `Carton` class can access all the members of the `Box` class, but functions in the `Box` class have no access to the private members of the `Carton` class. Friendship amongst classes is not transitive either; just because class A is a friend of class B, and class B is a friend of class C, it doesn't follow that class A is a friend of class C.

A typical use for a friend class is where the functioning of one class is highly intertwined with that of another. A linked list basically involves two class types: a `List` class that maintains a list of objects (usually called nodes), and a `Node` class that defines what a node is. The `List` class needs to stitch the `Node` objects together by setting a pointer in each `Node` object so that it points to the next `Node` object. Making the `List` class a friend of the class that defines a node would enable members of the `List` class to access the members of the `Node` class directly.

The this Pointer

The `volume()` function in the `Box` class was implemented in terms of the unqualified class member names. *Every* object of type `Box` contains these members so there must be a way for the function to refer to the members of the particular object for which it has been called. In other words, when the code in `volume()` accesses the `length` member, there has to be a way for `length` to refer to the member of the object for which the function is called, and not some other object.

When a class function member executes, it automatically contains a hidden pointer with the name `this`, which contains the address of the object for which the function was called. For example, suppose you write this statement:

```
std::cout << box1.volume() << std::endl;
```

The `this` pointer in the `volume()` function contains the address of `box1`. When you call the function for a different `Box` object, `this` will contain the address of that object. This means that when the data member `length` is accessed in the `volume()` function during execution, it is actually referring to `this->length`, which is the fully specified reference to the object member that is being used. The compiler takes care of adding the `this` pointer name to the member names in the function. In other words, the compiler implements the function as:

```
double Box::volume()
{
    return this->length * this->width * this->height;
}
```

You could write the function explicitly using the pointer `this` if you wanted to, but it isn't necessary. However, there are situations where you *do* need to use `this` explicitly. For example, when you need to return the address of the current object.

■ **Note** You'll learn about static function members of a class later in this chapter that do not contain the `this` pointer.

Returning `this` from a Function

If the return type for a function member is a pointer to the class type, you can return `this`. You can then use the pointer returned by one function member to call another. Let's consider an example of where this would be useful.

Suppose you add mutator functions to the `Box` class to set the length, width, and height of a box, and you define these functions so they return `this`:

```
class Box
{
private:
    double length;
    double width;
    double height;

public:
    // Constructors
    Box(double lv = 1.0, double wv = 1.0, double hv = 1.0);

    double volume();                                // Function to calculate the volume of a box

    // Mutator functions
    Box* setLength(double lv);
    Box* setWidth(double wv);
    Box* setHeight(double hv);
};
```

You can implement these in `Box.cpp` as follows:

```
Box* Box::setLength(double lvalue)
{
    if(lv > 0) length = lv;
    return this;
}

Box* Box::setWidth(double wv)
{
    if(wv > 0) width = wv;
    return this;
}

Box* Box::setHeight(double hv)
{
    if(hv > 0) height = hv;
    return this;
}
```

Now you can modify all the dimensions of a `Box` object in a single statement:

```
Box aBox {10.0,15.0,25.0};           // Create a box
aBox.setLength(20.0)->setWidth(40.0)->setHeight(10.0); // Set all dimensions of aBox
```

Because the mutator functions return the `this` pointer, you can use the value returned by one function to call the next. Thus the pointer returned by `setLength()` is used to call `setWidth()`, which returns a pointer you can use to call `setHeight()`. Isn't that nice?

const Objects and const Member Functions

Let's look again at the `volume()` function member of the `Box` class in `Ex11_03`. Suppose you change the code in `main()` so that `box1` is `const`:

```
const Box box1 {2.0, 3.0, 4.0};           // A box that is a constant
Box box2 {5.0};                           // A box that is a cube
std::cout << "box1 volume = " << box1.volume() << std::endl; // Won't compile!
std::cout << "box2 volume = " << box2.volume() << std::endl;
```

You can specify any variable as `const`, including variables that are of class types. Now the example will no longer compile. The compiler will not allow you to call the `volume()` function member for a `const` object because there's the risk that it could change the object. `volume()` doesn't alter the object for which it is called so you need a way to tell the compiler this. First, you specify the function as `const` in the class definition:

```
class Box
{
    // Rest of the class as before...
    double volume() const;           // Function to calculate the volume of a box
};
```

You must also change the function definition in `Box.cpp` the same way:

```
double Box::volume() const
{
    return length*width*height;
}
```

With these changes the modified version of `Ex11_03` will work. You can only call `const` function members for `const` objects so you should specify all function members that do not change the object for which they are called as `const`. Specifying a function member as `const` makes the `this` pointer `const` for the function. Thus you can call a `const` function member for `const` or non-`const` objects. Non-`const` function members can only be called for non-`const` objects because the `this` pointer in a non-`const` function member is not `const`.

Declaring a function member as `const` affects the function signature. This means that you can overload a non-`const` function member with a `const` version. However, you should be careful about overloading a function member on the basis of `const`-ness, as it can be confusing to someone using the class.

Ordinarily the data members of a `const` object cannot be modified. Sometimes you want to allow particular class members to be modifiable even for a `const` object. You can do this by specifying such members as `mutable`. For example:

```
class Box
{
private:
    double length;
    double width;
    double height;
    mutable std::string name;           // Name of a box

    // Rest of the class definition...
};
```

The `mutable` keyword indicates that the `name` member can be changed, even when the object is `const`. `const` or non-`const` functions can always make changes to data members specified as `mutable`.

Casting Away `const`

Very rarely, circumstances can arise where a function is dealing with a `const` object, either passed as an argument or the object pointed to by `this`, and it is necessary to make it non-`const`. This could be because you want to pass it as an argument to another function—perhaps written by someone else—that has a non-`const` parameter. The `const_cast<>()` operator enables you to do this. The general form of using the `const_cast<>()` operator is

```
const_cast<Type>(expression)
```

Here, the type of `expression` must be either `const Type` or the same as `Type`. You *should not* use this operator to undermine the `const`-ness of an object. The only situations in which you should use it are those where you are sure the `const` nature of the object won't be violated as a result.

Arrays of Class Objects

You can create an array of objects of a class type in exactly the same way as you create an array of elements of any other type. Each array element has to be created by a constructor and for each element that does not have an initial value specified, the compiler arranges for the no-arg constructor to be called. You can see this happening with an example. The `Box` class definition in `Box.h` is:

```
// Box.h
#ifndef BOX_H
#define BOX_H
#include <iostream>

class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructors
    Box(double lv, double wv, double hv);

    Box(double side) : Box{side, side, side}           // Constructor for a cube
    { std::cout << "Box constructor 2 called." << std::endl; }

    Box()                                               // No-arg constructor
    { std::cout << "No-arg Box constructor called." << std::endl; }

    Box(const Box& box)                                // Copy constructor
        : length {box.length}, width {box.width}, height {box.height}
    { std::cout << "Box copy constructor called." << std::endl; }

    double volume() const;                             // Function to calculate the volume of a box
};
#endif
```

The contents of `Box.cpp` is:

```
#include <iostream>
#include "Box.h"

// Constructor definition
Box::Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv}
{ std::cout << "Box constructor 1 called." << std::endl; }

// Function to calculate the volume of a box
double Box::volume() const
{ return length*width*height; }
```

The Ex11_05.cpp defining main() will contain:

```
// Ex11_05.cpp
// Creating an array of objects
#include <iostream>
#include "Box.h"

int main()
{
    const Box box1 {2.0, 3.0, 4.0};           // An arbitrary box
    Box box2 {5.0};                          // A box that is a cube
    std::cout << "box1 volume = " << box1.volume() << std::endl;
    std::cout << "box2 volume = " << box2.volume() << std::endl;
    Box box3 {box2};
    std::cout << "box3 volume = " << box3.volume() << std::endl; // Volume = 125
    Box boxes[6] {box1, box2, box3, Box {2.0}};
}
```

The output is:

```
Box constructor 1 called.
Box constructor 1 called.
Box constructor 2 called.
box1 volume = 24
box2 volume = 125
Box copy constructor called.
box3 volume = 125
Box copy constructor called.
Box copy constructor called.
Box copy constructor called.
Box constructor 1 called.
Box constructor 2 called.
No-arg Box constructor called.
No-arg Box constructor called.
```

The interesting bit is the last seven lines, which result from the creation of the array of Box objects. The initial values for the first three array elements are existing objects so the compiler calls the copy constructor to duplicate box1, box2, and box3. The fourth element is initialized with an object that is created in the initializer list for the array by the constructor 2, which calls constructor 1 in its initialization list. The last two array elements have no initial values specified so the compiler calls the no-arg constructor to create these.

The Size of a Class Object

You obtain the size of a class object by using the `sizeof` operator in exactly the way you have previously with fundamental data types. You can apply the operator to a particular object, or to the class type. The size of a class object is generally the sum of the sizes of the data members of the class, although on some machines, it may turn out to be greater than this occasionally. This isn't something that should bother you, but it's nice to know why.

On some computers, for performance reasons, two-byte variables must be placed at an address that is a multiple of two, four byte variables must be placed at an address that is a multiple of four, and so on. This is called *boundary alignment*. A consequence of this is that sometimes, the compiler must leave gaps between the memory for

one value and the next. If, on such a machine, you have three variables that occupy two bytes, followed by a variable that requires four bytes, a gap of two bytes may be left in order to place the fourth variable on the correct boundary. In this case, the total space required by all four is greater than the sum of the individual sizes.

Static Members of a Class

You can declare members of a class as `static`. *Static data members* of a class are used to provide class-wide storage of data that is independent of any particular object of the class type, but is accessible by any of them. They record properties of the class as a whole, rather than of individual objects. You can use static data members to store constants that are specific to a class, or you could store information about the objects of a class in general, such as how many there are in existence.

A *static function member* is independent of any individual class object, but can be invoked by any class object if necessary. It can also be invoked from outside the class if it is a public member. A common use of static function members is to operate on static data members, regardless of whether any objects of the class have been defined.

Because the context is a class, there is a little more to this topic than the effect of the `static` keyword outside a class, so I'll go into it in a little more detail.

Static Data Members

Static data members of a class are associated with the class as a whole, not with any particular object of the class. When you declare a data member of a class as `static`, the static data member is defined only once, and will exist even if no class objects have been created. Each static data member is accessible in any object of the class and is shared among however many objects there are. An object gets its own independent copies of the ordinary data members but only one instance of each static data member exists, regardless of how many class objects have been defined.

One use for a static data member is to count how many objects of a class exist. You could add a static data member to the `Box` class by adding the following statement to your class definition:

```
static size_t objectCount;           // Count of objects in existence
```

Figure 11-5 shows how this member exists outside of any objects but is available to all of them. Now you have a problem. How do you initialize the static data member? A static data member is not part of an object so in general you can't initialize a static data member in the class definition—the class is simply a blueprint for an object, and the initialization of non-static data members occurs for each object when it is created. The one exception is if the static member is `const` and is an integral or enumeration type, in which you can specify the initial value in the class. You don't want to initialize it in a constructor, because you want to increment it each time a constructor is called; and anyway, it exists even if no objects exist (and therefore no constructors have been called).

```

class Box
{
private:
    static size_t objectCount;
    double length;
    double breadth;
    double height;
    ...
}

```

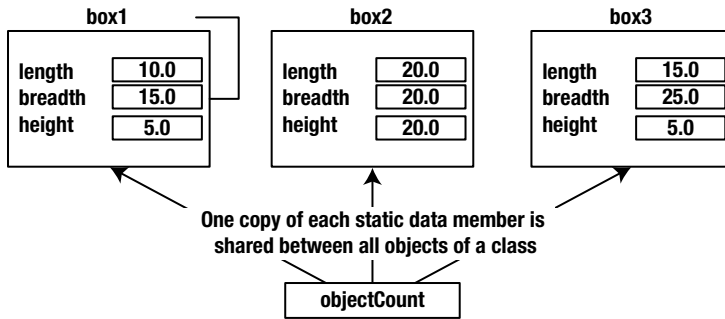


Figure 11-5. Static class members are shared between objects

The answer is to initialize each static member outside the class with a statement such as this:

```
size_t Box::objectCount {};
```

// Initialize static member of Box class to 0

This *defines* `objectCount`. The line in the class definition declares that it is a static member of the class. Even though the static data member is specified as `private`, you can still initialize it in this fashion. Indeed, this is the *only* way you can initialize it. Of course, because it's `private`, you can't access `objectCount` from outside the class. Because this statement defines the class static member, it must occur only once in a program. The logical place to put it is the `Box.cpp` file. Note that the `static` keyword is not included in the definition—indeed, you must not include it here. You do need to qualify the member name with the class name so that the compiler understands that you are referring to a static member of the class. Otherwise, you'd simply create a global variable that has nothing to do with the class.

Let's add the static data member and the object counting capability to `Ex11_05`. You need two extra statements in the class definition: one to declare the new static data member, and another to define a function that will retrieve its value. The constructors also need to increment `objectCount`:

```

class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};
    static size_t objectCount;
    // Count of objects in existence

public:
    // Constructors
    Box(double lv, double wv, double hv);
}

```



```

Box(double side) : Box {side, side, side}           // Constructor for a cube
{
    std::cout << "Box constructor 2 called." << std::endl;
}

Box()                                                 // No-arg constructor
{
    ++objectCount;
    std::cout << "No-arg Box constructor called." << std::endl;
}

Box(const Box& box) :                                // Copy constructor
    length {box.length}, width {box.width}, height {box.height}
{
    ++objectCount;
    std::cout << "Box copy constructor called." << std::endl;
}

double volume() const;                               // Function to calculate the volume of a box
size_t getObjectCount() const { return objectCount; }
};

```

The `getObjectCount()` function has been declared as `const` because it doesn't modify any of the data members of the class and you might want to call it for `const` or non-`const` objects. You can add the statement to initialize the static member `objectCount` in the `Box.cpp` file:

```

#include <iostream>
#include "Box.h"

size_t Box::objectCount {};                          // Initialize static member of Box class to 0

// Constructor definition
Box::Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv}
{
    ++objectCount;
    std::cout << "Box constructor 1 called." << std::endl;
}

// Function to calculate the volume of a box
double Box::volume() const
{
    return length*width*height;
}

```

This constructor definition now updates the count when an object is created. You can modify the version of `main()` from `Ex11_05` to output the object count:

```

// Ex11_06.cpp
// Using a static data member
#include <iostream>
#include "Box.h"

```

```

int main()
{
    const Box box1 {2.0, 3.0, 4.0};           // An arbitrary box
    Box box2 {5.0};                          // A box that is a cube
    std::cout << "box1 volume = " << box1.volume() << std::endl;
    std::cout << "box2 volume = " << box2.volume() << std::endl;
    Box box3 {box2};
    std::cout << "box3 volume = " << box3.volume() << std::endl;    // Volume = 125
    Box boxes[6] {box1, box2, box3, Box {2.0}};
    std::cout << "There are now " << box1.getObjectCount() << " objects." << std::endl;
}

```

This program will produce the following output:

```

Box constructor 1 called.
Box constructor 1 called.
Box constructor 2 called.
box1 volume = 24
box2 volume = 125
Box copy constructor called.
box3 volume = 125
Box copy constructor called.
Box copy constructor called.
Box copy constructor called.
Box constructor 1 called.
Box constructor 2 called.
No-arg Box constructor called.
No-arg Box constructor called.
There are now 9 objects.

```

This code shows that, indeed, only one copy of the static member `objectCount` exists, and all the constructors are updating it. The `getObjectCount()` function is called for the `box1` object but you could use any object including any of the array elements to get the same result. Of course, you're only counting the number of objects that get created. The count that is output corresponds to the number of objects created here. In general though you have no way to know when objects are destroyed, so the count won't necessarily reflect the number of objects that are around at any point. You'll find out later in this chapter how to account for objects that get destroyed.

Note that the size of a `Box` object will be unchanged by the addition of `objectCount` to the class definition. This is because static data members are not part of any object—they belong to the class. Because static data members are not part of a class object, a const function member can modify non-const static data members without violating the const nature of the function.

Accessing Static Data Members

Suppose that in a reckless moment, you declared `objectCount` as a public class member. You no longer need the `getObjectCount()` function to access it. To output the number of objects in `main()`, just write this:

```
std::cout << "Object count is " << firstBox.objectCount << std::endl;
```

There's more: I claimed that a static data member exists even if no objects have been created. This means that you should be able to get the count *before* you create the first `Box` object, but how do you refer to the data member? The answer is that you use the class name, `Box`, as a qualifier:

```
std::cout << "Object count is " << Box::objectCount << std::endl;
```

You can always use the class name to access a public static member of a class. It doesn't matter whether any objects exist or not. Try it out by modifying the last example; you'll see that it works as described.

A Static Data Member of the Class Type

A static data member is not part of a class object so it can be of the same type as the class. The `Box` class can contain a static data member of type `Box`, for example. This might seem a little strange at first, but it can be useful. I'll use the `Box` class to illustrate just how. Suppose you need a standard "reference" box for some purpose; you might want to relate `Box` objects in various ways to a standard box for example. Of course, you could define a standard `Box` object outside the class, but if you are going to use it within function members of the class, it creates an external dependency that it would be better to lose.

```
class Box
{
private:
    const static Box refBox;                // Standard reference box

    // Rest of the class as before...
};
```

`refBox` is `const` because it is a standard `Box` object that should not be changed. However, you must still define and initialize it outside the class. You could put a statement in `Box.cpp` to define `refBox`:

```
const Box Box::refBox {10.0, 10.0, 10.0};
```

This calls the `Box` class constructor to create `refBox`. Because static data members of a class are created before any objects are created, at least one `Box` object will always exist. Any of the static or non-static function members can access `refBox`. It isn't accessible from outside the class because it is a `private` member. A class constant is one situation where you might want to make the data member `public` if it has a useful role outside the class. As long as it is declared as `const`, it can't be modified.

Static Function Members

A static function member is independent of any class object. A public static function member can be called even if no class objects have been created. Declaring a static function in a class is easy: you simply use the `static` keyword as you did with `objectCount`. You could have declared the `getObjectCount()` function as static in the previous example. You call a static function member using the class name as a qualifier. Here's how you could call the static `getObjectCount()` function:

```
std::cout << "Object count is" << Box::getObjectCount() << std::endl;
```

Of course, if you have created class objects, you can call a static function member through an object of the class in the same way as you call any other function member. For instance:

```
std::cout << "Object count is" << box1.getObjectCount() << std::endl;
```

A static function member has no access to the object for which it is called. In order for a static function member to access an object of the class, it would need to be passed as an argument to the function. Referencing members of a class object from within a static function must then be done using qualified names (as you would with an ordinary global function accessing a public data member).

Of course, a static function member is a full member of the class in terms of access privileges. If an object of the same class is passed as an argument to a static function member, it can access `private` as well as `public` members of the object. It wouldn't make sense to do so, but just to illustrate the point, you could include a definition of a static function in the `Box` class as shown here:

```
static double edgeLength(Box aBox)
{
    return 4.0*(aBox.length + aBox.width + aBox.height);
}
```

Even though you are passing the `Box` object as an argument, the `private` data members can be accessed. Of course, it would make more sense to do this with an ordinary function member.

■ **Caution** Static function members can't be `const`. Because a static function member isn't associated with any class object, it has no `this` pointer, so `const`-ness doesn't apply.

Destructors

At the end of a block in which a class object is created, the object is destroyed, just like a variable of a fundamental type. When an object is destroyed, a special member of the class called a *destructor* is executed to deal with any clean-up that may be necessary. A class can have only one destructor. The compiler provides a default version of the destructor that does nothing if you don't define one. The definition of the default constructor looks like this:

```
~ClassName() {}
```

The name of the destructor for a class is always the class name prefixed with a tilde, `~`. The destructor cannot have parameters or a return type. The default destructor in the `Box` class is:

```
~Box() {}
```

Of course, if the definition is placed outside the class, the name of the destructor would be prefixed with the class name:

```
Box::~~Box() {}
```

The destructor for a class is always called automatically when an object is destroyed. The circumstances where you need to call a destructor explicitly are so rare you can ignore the possibility. Calling a destructor when it is not necessary can cause problems. You only need to define a class destructor when something needs to be done when an object is destroyed. A class that deals with physical resources such as a file then needs to be closed is one example

and of course if memory is allocated by a constructor using `new`, the destructor is the place to release the memory. I'll go into an example where you must define a destructor in Chapter 12. The `Box` class in `Ex11_06` would benefit from a destructor implementation that decremented `objectCount`:

```
class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};
    static size_t objectCount;                                // Count of objects in existence

public:
    // Constructors
    Box(double lv, double wv, double hv);

    Box(double side) : Box {side, side, side}                // Constructor for a cube
    {
        std::cout << "Box constructor 2 called." << std::endl;
    }

    Box()                                                      // No-arg constructor
    {
        ++objectCount;
        std::cout << "No-arg Box constructor called." << std::endl;
    }

    Box(const Box& box)                                         // Copy constructor
    : length {box.length}, width {box.width}, height {box.height}
    {
        ++objectCount;
        std::cout << "Box copy constructor called." << std::endl;
    }

    double volume() const;                                     // Function to calculate the volume of a box

    static size_t getObjectCount() { return objectCount; }

    ~Box()                                                      // Destructor
    {
        std::cout << "Box destructor called." << std::endl;
        --objectCount;
    }
};
```

The destructor has been added to decrement `objectCount`, and `getObjectCount()` is now a static function member. The destructor outputs a message when it is called so you can see when this occurs. If `Box.cpp` is the same as in `Ex11_05`, the following code will check the destructor operation out:

```
// Ex11_07.cpp
// Implementing a destructor
#include <iostream>
#include <memory>
#include "Box.h"

int main()
{
    std::cout << "There are now" << Box::getObjectCount() << "objects." << std::endl;
    const Box box1 {2.0, 3.0, 4.0};           // An arbitrary box
    Box box2 {5.0};                          // A box that is a cube
    std::cout << "There are now" << Box::getObjectCount() << "objects." << std::endl;
    for (double d {} ; d < 3.0 ; ++d)
    {
        Box box {d, d + 1.0, d + 2.0};
        std::cout << "Box volume is" << box.volume() << std::endl;
    }
    std::cout << "There are now" << Box::getObjectCount() << "objects." << std::endl;

    auto pBox = std::make_shared<Box>(1.5, 2.5, 3.5);
    std::cout << "Box volume is" << pBox->volume() << std::endl;
    std::cout << "There are now" << pBox->getObjectCount() << "objects." << std::endl;
}
```

The output from this example is:

```
There are now 0 objects.
Box constructor 1 called.
Box constructor 1 called.
Box constructor 2 called.
There are now 2 objects.
Box constructor 1 called.
Box volume is 0
Box destructor called.
Box constructor 1 called.
Box volume is 6
Box destructor called.
Box constructor 1 called.
Box volume is 24
Box destructor called.
There are now 2 objects.
Box constructor 1 called.
Box volume is 13.125
There are now 3 objects.
Box destructor called.
Box destructor called.
Box destructor called.
```

This example shows when constructors and the destructor are called and how many objects exist at various points during execution. The first line of output shows there are no `Box` objects at the outset. `objectCount` clearly exists without any objects because we retrieve its value using the static `getObjectCount()` member. `box1` and `box2` are created in the way you saw in the previous example and the output shows that there are indeed two objects in existence. The `for` loop created a new object on each iteration and the output shows that the new object is destroyed at the end of the current iteration, after its volume has been output. After the loop ends, there are just the original two objects in existence. The last object is created on the heap by calling the `make_shared<Box>()` function the template for which is defined in the memory header. This calls the `Box` constructor that has three parameters to create the object on the heap. Just to show that you can, `getObjectCount()` is called using the smart pointer, `pBox`. You can see the output from the three destructor calls that occur when `main()` ends and that destroy the remaining three `Box` objects.

You now know that the compiler will add a default constructor, a default copy constructor, and a destructor to each class when you don't define these. There are other members that the compiler can add to a class and you'll learn about those in Chapter 12.

Pointers and References to Class Objects

You can define and use pointers and references to class objects in the same way as for fundamental types of data. Pointers and references to class objects are essential in object-oriented programming and they each provide particular advantages. You can use a pointer to a class object in three ways:

1. As a means of invoking operations on an object—that is, calling functions using the `->` operator.
2. As an argument to a function.
3. As a data member of a class.

The first of these enables you to call a function polymorphically, where the function called depends on the type of object pointed to. You'll learn about this in detail in Chapter 14. When you use a pointer as an argument to a function, you avoid the copying that is implicit in the pass-by-value mechanism. This can vastly improve the efficiency of a program, especially for large objects because copying large objects can be time consuming.

A class member that is a pointer can store the address of a data item on the heap. A pointer can also store the address of another object of the same type as the class for which it is a member which allows a series of objects to be linked. It can even allow objects of different types to be linked. Pointers as class members enables objects to be organized into structures such as linked lists, graphs, or trees. Returning a pointer to an object from a function member also has advantages.

References to objects have great importance as parameter types for functions. Passing an object by reference also avoids the copying that is inherent in the pass-by-value mechanism. A reference parameter is essential to implementing a copy constructor, as you'll see.

Using Pointers As Class Members

I'll define a class with a data member that is a pointer and use instances of the class to create a *linked list* of objects.

■ **Note** You don't need to create your own classes for linked lists. Very flexible versions are already defined in the `list` standard library header. Defining your own class for a linked list is very educational though.

I'll define a class that represents a collection of any number of Box objects—the contents of the header file for the Box class definition will be:

```
// Box.h
#ifndef BOX_H
#define BOX_H
#include <iostream>
#include <iomanip>

class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructors
    Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv} {};

    Box() {} // No-arg constructor

    Box(const Box& box) : // Copy constructor
        length {box.length}, width {box.width}, height {box.height} {}

    double volume() const // Volume of a box
    {
        return length*width*height;
    }

    int compare(const Box& box)
    {
        if (volume() < box.volume()) return -1;
        if (volume() == box.volume()) return 0;
        return 1;
    }

    void listBox()
    {
        std::cout << " Box(" << std::setw(2) << length << ", "
            << std::setw(2) << width << ", "
            << std::setw(2) << height << ")";
    }
};
#endif
```

I have omitted the accessor function members because they are not required here, but I have added a `listBox()` member to output a Box object. In this case, a Box object represents a unit of a product to be delivered, and a collection of Box objects represents a truckload of boxes, so I'll call the class `Truckload`; the collection of Box objects will be a linked list. A linked list can be as long or as short as you need and you can add objects anywhere in the list. The class will allow a `Truckload` object to be created from a single Box object or from a vector of Box objects. It will provide for adding and deleting a Box object, and for retrieving all the Box objects in the `Truckload`.

A Box object has no built-in facility for linking it with another Box object. Changing the definition of the Box class to incorporate this capability would be inconsistent with the idea of a box—boxes aren't like that. One way to collect Box objects into a list is to define another type of object, which I'll call Package. A Package object will have two members: a pointer to a Box object, and a pointer to another Package object. Thus it will be possible to create a chain of Package objects.

Figure 11-6 shows how each Package object points to a Box object and also forms a link in a chain of Package objects that are connected by pointers—a linked list. The list of Package objects can be of unlimited length. As long as you can access the first Package object, you can access the next Package through the pNext pointer it contains, which allows you to reach the next through the pNext pointer that contains, and so on through all objects in the list. Each Package object can provide access to the Box object through its pBox member. This arrangement is superior to the Package class having a member that is of type Box, which would require a new Box object to be created for each Package object. The Package class is just a means of tying Box objects together in a linked list and each Box object should exist independently from the Package objects.

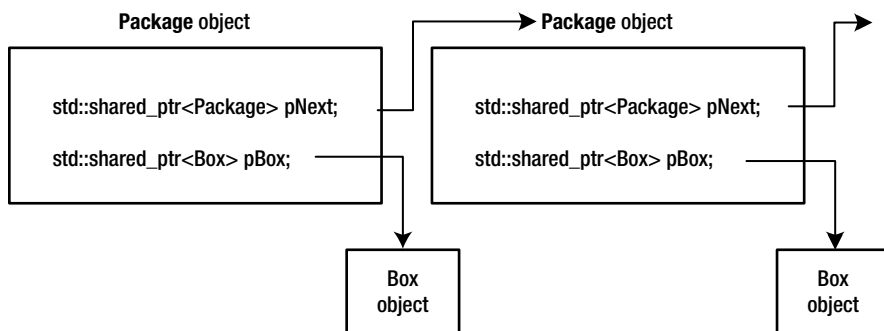


Figure 11-6. Linked Package objects

A Truckload object will create and manage a list of Package objects. A Truckload object represents an instance of a truckload of boxes. There can be any number of boxes in a truckload and each box will be referenced from within a package. A Package object provides the mechanism for the Truckload object to access the pointer to the Box object it contains. The relationship between these objects is illustrated in Figure 11-7.

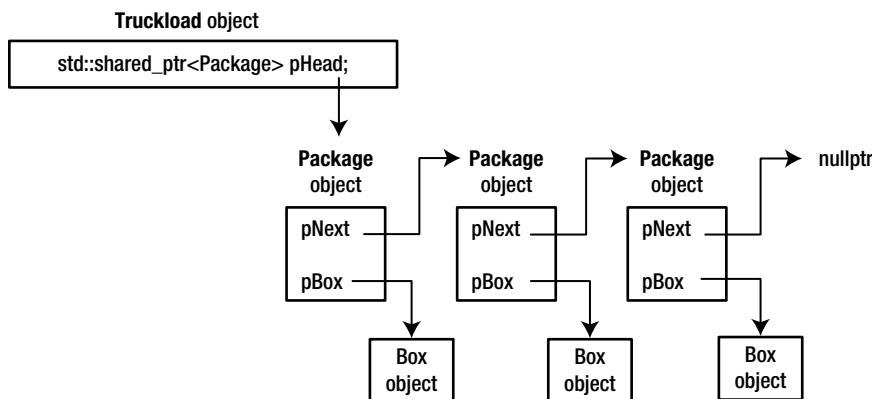


Figure 11-7. A Truckload object managing a linked list of three Package objects

Figure 11-7 shows a `Truckload` object that manages a list of `Package` objects; each `Package` object contains a `Box` object and a pointer to the next `Package` object. The `Truckload` object only needs to keep track of the first `Package` object in the list; the `pHead` member contains its address. By following the `pNext` pointer links you can find any of the objects in the list. In this elementary implementation the list can only be traversed from the start. A more sophisticated implementation could provide each `Package` object with a pointer to the previous object in the list which would allow the list to be traversed backwards as well as forwards. Let's put the ideas into code.

Defining the Package Class

Based on the preceding discussion, the `Package` class can be defined in `Package.h` like this:

```
// Package.h
#ifndef PACKAGE_H
#define PACKAGE_H
#include <memory>
#include "Box.h"
template <typename T> using ptr = std::shared_ptr<T>;

class Package
{
private:
    ptr<Box> pBox;                // Pointer to the Box object
    ptr<Package> pNext;          // Pointer to the next Package

public:
    Package(ptr<Box> pb) : pBox {pb}, pNext {} {} // Constructor

    ptr<Box>& getBox() { return pBox; }             // Retrieve the Box pointer
    ptr<Package>& getNext() { return pNext; }        // Get next Package address

    void setNext(ptr<Package>& pPackage)           // Point to next object
    {
        pNext = pPackage;
    }
};
#endif
```

This uses a template for a using statement that defines an alias for a templated type. Thus you can use `ptr<T>` as the type for a smart pointer to an object of type `T` so `ptr<Box>` is an alias for `std::shared_ptr<Box>`. This template makes the code a little less cluttered. The `ptr<Box>` member of the `Package` class will store the address of a `Box` object and the `ptr<Package>` member will point to the next `Package` object in the list. The `pNext` member for the last `Package` object in a list will contain `nullptr`.

The constructor allows a `Package` object to be created that contains the address of the `Box` argument. The `pNext` member will be `nullptr` by default but it can be set to point to a `Package` object by calling the `setNext()` member. The `setNext()` function updates `pNext` to the next `Package` in the list. To add a new `Package` object to end of the list, you pass its address to the `setNext()` function for the last `Package` object in a list.

Defining the Truckload Class

A Truckload object will encapsulate a list of Package objects. The class must provide everything necessary to create and extend and delete from the list and also the means by which Box objects can be retrieved. A pointer to the first Package object in the list as a data member will allow you can get to any Package object in the list by stepping through the chain of pNext pointers, using the getNext() function from the Package class. The getNext() function will be called repeatedly to step through the list one Package object at a time, so the Truckload object will need to track the object that was retrieved most recently. It's also useful to store the address of the last Package object, as this makes it easy to add a new object to the end of the list. Figure 11-8 shows this.

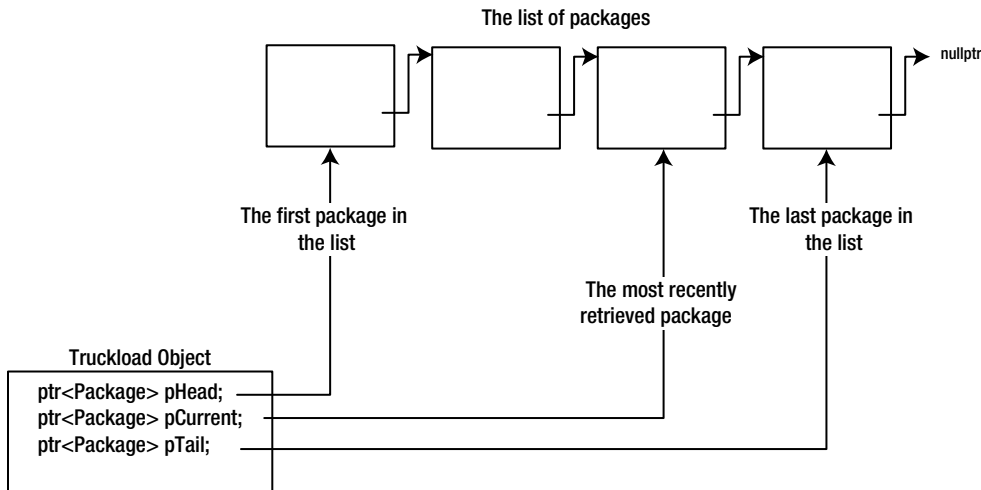


Figure 11-8. Information needed in a Truckload object to manage the list

Consider how retrieving Box objects from a Truckload object could work. This inevitably involves stepping through the list so the starting point is the first object in the list. You could define a getFirstBox() function member in the Truckload class to retrieve the pointer to the first Box object and record the address of the Package object that contained it in pCurrent. You can then implement a getNextBox() function member that will retrieve the pointer to the Box object from the *next* Package object in the list and then update pCurrent to reflect that. Another essential capability is the ability to add a Box to the list and delete a Box from the list, so you'll need function members to do that; addBox() and deleteBox() would be suitable names for these. A function member to list all the Box objects in the list will also be handy.

Here's a definition for the Truckload class based on these ideas:

```
#include "Package.h"

class Truckload
{
private:
    ptr<Package> pHead;           // First in the list
    ptr<Package> pTail;           // Last in the list
    ptr<Package> pCurrent;        // Last retrieved from the list
```

```

public:
    Truckload() {} // No-arg constructor empty truckload

    Truckload(ptr<Box> pBox) // Constructor - one Box
    { pHead = pTail = std::make_shared<Package>(pBox); }

    Truckload(const std::vector< ptr<Box> >& boxes); // Constructor - vector of Boxes

    ptr<Box> getFirstBox(); // Get the first Box
    ptr<Box> getNextBox(); // Get the next Box
    void addBox(ptr<Box> pBox); // Add a new Box
    bool deleteBox(ptr<Box> pBox); // Delete a Box
    void listBoxes(); // Output the Boxes
};
#endif

```

The data members are private because they don't need to be accessible outside the class. There are three constructors. The no-arg constructor defines an object containing an empty list. You can also create an object from a single pointer to a Box object or from a vector of pointers. The `getFirstBox()` and `getNextBox()` members provide the mechanism for retrieving Box objects. Each of these needs to modify the `pCurrent` pointer, so they cannot be `const`. The `addBox()` and `deleteBox()` functions also change the list so they cannot be `const` either.

The constructor that accepts a vector of pointers to Box objects and the other function members of the class require external definitions, which I'll put in a `Truckload.cpp` file so they will not be `inline`. You could define them as `inline` and include the definitions in `Truckload.h`.

Implementing the Truckload Class

The `Truckload.cpp` file will need the following `#include` directives:

```

#include <memory> // For smart pointers
#include <vector> // For vector<T>
#include "Box.h"
#include "Package.h"
#include "Truckload.h"

```

I'll start with the constructor definition. This creates a list of one or more Package objects from a vector of smart pointers to Box objects:

```

Truckload::Truckload(const std::vector< ptr<Box> >& boxes)
{
    for (auto pBox : boxes)
    {
        addBox(pBox);
    }
}

```

The parameter is a `const` reference to avoid copying of the argument. The vector elements are of type `std::shared_ptr<Box>`. The loop iterates through the vector elements passing each one to the `addBox()` member of the `Truckload` class, which will create and add a `Package` object on each call.

The `addBox()` member definition will be:

```
void Truckload::addBox(ptr<Box> pBox)
{
    auto pPackage = std::make_shared<Package>(pBox); // Create a Package

    if (pHead)                                     // Check list is not empty
        pTail->setNext(pPackage);                  // Add the new object to the tail
    else                                           // List is empty
        pHead = pPackage;                          // so new object is the head

    pTail = pPackage;                             // Store its address as tail
}
```

The function creates a new `Package` object from the `pBox` pointer in the free store and stores its address in the local smart pointer, `pPackage`. If `pHead` is non-null, then the list is not empty, in which case the new object is added to the end of the list by storing its address in the `pNext` member of the last member that is pointed to by `pTail`. If `pHead` is `nullptr`, then the list is empty so the address of the new object is the first member of the list. In either case the new `Package` object is at the end of the list, so `pTail` is updated to reflect this.

The `getFirstBox()` function definition is a piece of cake—just two statements:

```
ptr<Box> Truckload::getFirstBox()
{
    pCurrent = pHead->getNext();
    return pHead->getBox();
}
```

The address of the first `Package` object in the list is in `pHead`. Calling the `getBox()` function for this `Package` object obtains the address of the `Box` object, which is returned from `getFirstBox()`. Before that occurs, the address of the next `Package` object, which may be `nullptr` of course, is stored in `pCurrent`.

The `getNextBox()` function can access the `Package` object that follows that pointed to by `pCurrent` by calling its `getNext()` function and then calling `getBox()`. It's possible that `pCurrent` may be `nullptr` so `getNextBox()` must verify that is not the case before calling `getNext()`. The code for `getNextBox()` is:

```
ptr<Box> Truckload::getNextBox()
{
    if (!pCurrent)                                // If there's no current...
        return getFirstBox();                     // ...return the 1st

    auto pPackage = pCurrent->getNext();           // Save the next package
    if (pPackage)                                  // If there is one...
    {
        pCurrent = pPackage;                       // Update current to the next
        return pPackage->getBox();
    }

    pCurrent = nullptr;                            // If we get to here...
    return nullptr;                                // ...there was no next
}
```

When `pCurrent` contains the address of a `Package` object, you call its `getNext()` member to obtain the address of the next `Package` object and save it in `pPackage`. If `pCurrent` is `nullptr`, then the first in the list is obtained and returned by calling `getFirstBox()`. If the pointer to the next `Package` object, `pPackage`, is not `nullptr`, it is stored in `pCurrent` and the `Box` pointer it contains is returned. If `pPackage` is `nullptr`, the end of the list has been reached so `pCurrent` is reset to `nullptr` and `nullptr` is returned. Because `nullptr` is returned to signal there is no next pointer to a `Box`, we cannot return a reference.

The member to list the contents of the `Truckload` object can be implemented like this:

```
void Truckload::listBoxes()
{
    pCurrent = pHead;
    size_t count {};
    while (pCurrent)
    {
        pCurrent->getBox()->listBox();
        pCurrent = pCurrent->getNext();
        if(! (++count % 5)) std::cout << std::endl;
    }
    if (count % 5) std::cout << std::endl;
}
```

This steps through the `Package` objects in the linked list and outputs the `Box` object that each contains by calling `getBox()` for the `Package` object to access the pointer to the `Box` and using that to call `listBox()` for the `Box` object. `Box` objects are output five on a line. The last statement outputs a newline when the last line contains output for less than five `Box` objects.

We now need some code to try out the `Truckload` class:

```
// Ex11_08.cpp
// Using a linked list
#include <iostream>
#include <memory>
#include <vector>
#include <cstdlib>           // For random number generator
#include <ctime>             // For time function
#include "Box.h"
#include "Truckload.h"

// Function to generate a random integer 1 to count
inline size_t random(size_t count)
{
    return 1 + static_cast<size_t> (count*static_cast<double>(std::rand())/(RAND_MAX + 1.0));
}

int main()
{
    const size_t dimLimit {99};           // Upper limit on Box dimensions
    std::srand((unsigned)std::time(0));    // Initialize the random number generator

    Truckload load1;                      // Create an empty list
```

```

// Add 12 random Box objects to the list
const size_t boxCount {12};
for (size_t i {} ; i < boxCount ; ++i)
    load1.addBox(std::make_shared<Box>(random(dimLimit), random(dimLimit), random(dimLimit)));

std::cout << "The first list:\n";
load1.listBoxes();

// Find the largest Box in the list
ptr<Box> pBox {load1.getFirstBox()};
ptr<Box> pNextBox {};
while (pNextBox = load1.getNextBox()) // Assign & then test pointer to next Box
    if (pBox->compare(*pNextBox) < 0)
        pBox = pNextBox;

std::cout << "\nThe largest box in the first list is:";
pBox->listBox();
std::cout << std::endl;
load1.deleteBox(pBox);
std::cout << "\nAfter deleting the largest box, the list contains:\n";
load1.listBoxes();

const size_t nBoxes {20}; // Number of vector elements
std::vector< ptr<Box> > boxes; // Array of Box objects

for (size_t i {} ; i < nBoxes ; ++i)
    boxes.push_back(std::make_shared<Box>(
        random(dimLimit), random(dimLimit), random(dimLimit)));

Truckload load2(boxes);
std::cout << "\nThe second list:\n";
load2.listBoxes();

pBox = load2.getFirstBox();
while (pNextBox = load2.getNextBox())
    if (pBox->compare(*pNextBox) > 0)
        pBox = pNextBox;

std::cout << "\nThe smallest box in the second list is";
pBox->listBox();
std::cout << std::endl;
}

```

Here's some sample output from this program:

The first list:

```
Box(69,78,42) Box(42,85,57) Box(91,16,41) Box(20,91,78) Box(89,66,17)
Box(19,72,90) Box(82,68,98) Box(88,11,79) Box(21,93,75) Box(49,65,93)
Box(92,90,39) Box(99,21, 3)
```

The largest box in the first list is: Box(82,68,98)

After deleting the largest box, the list contains:

```
Box(69,78,42) Box(42,85,57) Box(91,16,41) Box(20,91,78) Box(89,66,17)
Box(19,72,90) Box(88,11,79) Box(21,93,75) Box(49,65,93) Box(92,90,39)
Box(99,21, 3)
```

The second list:

```
Box( 6,66,81) Box(98, 2, 7) Box(67,67,72) Box(68,69,64) Box(50,89,69)
Box( 8,87,92) Box(57,99,64) Box(74,31, 2) Box(56,37,52) Box( 9,50,35)
Box(46,74, 9) Box(13,18,78) Box(20,27,88) Box(17,74,37) Box(21,21, 5)
Box(70,85,64) Box(57,32,13) Box(38,62,15) Box(79,86,59) Box(88, 6,91)
```

The smallest box in the second list is Box(21,21, 5)

The `main()` function first creates an empty `Truckload` object, then adds `Box` objects in the `for` loop. It then finds the largest `Box` object in the list and deletes it. The output demonstrates that all these operations are working correctly. Just to show it works, `main()` creates a `Truckload` object from a vector of pointers to `Box` objects. It then finds the smallest `Box` object and outputs it. Clearly, the capability to list the contents of a `Truckload` object is also working well. Note that the template for the `ptr<T>` type alias can be used in `main()` because it is defined in `Package.h` and therefore available in this source file. Using smart pointers through has saved a considerable amount of work and made the code much safer. If the classes used raw pointers as members, it would be necessary to manage deleting objects from the free store so at the very least it would be necessary to define destructors. With smart pointers, as soon as there are no pointers to a given object, the object is deleted from the free store automatically.

Nested Classes

It's sometimes desirable to limit the accessibility of a class. The `Package` class was designed to be used specifically within the `TruckLoad` class. It would make sense to ensure that `Package` objects can only be created by function members of the `TruckLoad` class. What you need is a mechanism where `Package` objects are private to `Truckload` class members and not available to the rest of the world. You can do this by using a *nested class*.

A *nested class* is a class that has its definition inside another class definition. The name of the nested class is within the scope of the enclosing class and is subject to the member access specification in the enclosing class. We could put the definition of the `Package` class inside the definition of the `TruckLoad` class, like this:

```
#include <memory>
#include <vector>
template <typename T> using ptr = std::shared_ptr<T>;
```



```

class Truckload
{
private:
    // Package is private to the Truckload class
    class Package
    {
    public:
        ptr<Box> pBox;                // Pointer to the Box object
        ptr<Package> pNext;           // Pointer to the next Package

        Package(ptr<Box> pb) : pBox {pb}, pNext {} {} // Constructor
    };

    ptr<Package> pHead;                // First in the list
    ptr<Package> pTail;                // Last in the list
    ptr<Package> pCurrent;            // Last retrieved from the list

public:
    Truckload() {}                    // No-arg constructor empty truckload

    Truckload(ptr<Box> pBox)           // Constructor - one Box
    {
        pHead = pTail = std::make_shared<Package>(pBox);
    }

    Truckload(const std::vector< ptr<Box> >& boxes); // Constructor - vector of Boxes

    ptr<Box> getFirstBox();             // Get the first Box
    ptr<Box> getNextBox();              // Get the next Box
    void addBox(ptr<Box> pBox);         // Add a new Box
    bool deleteBox(ptr<Box> pBox);      // Delete a Box
    void listBoxes();                  // Output the Boxes
};

```

The Package type is now local to the scope of the TruckLoad class definition. Because the definition of the Package class is in the private section of the TruckLoad class, Package objects cannot be created from outside the TruckLoad class.

Because the Package class is entirely private to the TruckLoad class, you can make the Package members public. Hence, they're directly accessible to function members of a TruckLoad object. The getBox() and getNext() members of the original Package class are no longer needed. All of the Package members are directly accessible from Truckload objects, but inaccessible outside the class.

The definitions of the member functions of the TruckLoad class need to be changed to access the data members of the Package class directly. The addBox() function can add a new object to the end of the list by accessing the pNext member of the last object directly:

```

void Truckload::addBox(ptr<Box> pBox)
{
    auto pPackage = std::make_shared<Package>(pBox);    // Create a Package

    if (pHead)                                           // Check list is not empty
        pTail->pNext = pPackage;                        // Add the new object to the tail
}

```

```

else                                     // List is empty
    pHead = pPackage;                   // so new object is the head

pTail = pPackage;                       // Store its address as tail
}

```

The function to retrieve the first Box object can access the pointer to the Box object directly, so the definition now becomes:

```

ptr<Box> Truckload::getFirstBox()
{
    pCurrent = pHead->pNext;
    return pHead->pBox;
}

```

The function to obtain the address of the next Box object in the list can now be defined as:

```

ptr<Box> Truckload::getNextBox()
{
    if (!pCurrent)                       // If there's no current...
        return getFirstBox();           // ...return the 1st

    auto pPackage = pCurrent->pNext;      // Save the next package
    if (pPackage)                        // If there is one...
    {
        pCurrent = pPackage;             // Update current to the next
        return pPackage->pBox;
    }
    pCurrent = nullptr;                  // If we get to here...
    return nullptr;                      // ...there was no next
}

```

Finally the listBoxes() member of Truckload will be:

```

void Truckload::listBoxes()
{
    pCurrent = pHead;
    size_t count {};
    while (pCurrent)
    {
        pCurrent->pBox->listBox();
        pCurrent = pCurrent->pNext;
        if(! (++count % 5)) std::cout << std::endl;
    }
    if (count % 5) std::cout << std::endl;
}

```

The Truckload class definition with Package as a nested class will work with the Ex11_08.cpp source file. A complete example is in the code download as Ex11_09.

■ **Note** Nesting the `Package` class inside the `TruckLoad` class simply defines the `Package` type in the context of the `TruckLoad` class. Objects of type `TruckLoad` aren't affected in any way—they'll have exactly the same members as before.

Function members of a nested class can directly reference static members of the enclosing class, as well as any other types or enumerators defined in the enclosing class. Other members of the enclosing class can only be accessed from the nested class in the normal ways: via a class object, or a pointer, or a reference to a class object.

Nested Classes with Public Access

Of course, you could put the `Package` class definition in the `public` section of the `TruckLoad` class. This would mean that the `Package` class definition was part of the public interface so it *would* be possible to create `Package` objects externally. Because the `Package` class name is within the scope of the `TruckLoad` class, you can't use it by itself. You must qualify the `Package` class name with the name of the class in which it is nested. Here's an example:

```
TruckLoad::Package aPackage(aBox);           // Define a Package object
```

Of course, making the `Package` type public in the example would defeat the rationale for making it a nested class in the first place! Of course, there can be other circumstances where a public nested class makes sense.

Summary

In this chapter you have learned the basic ideas involved with defining and using class types. However, although you have covered a lot of ground, this is just the start. There's a great deal more to implementing the operations applicable to class objects and there are subtleties in this too. In subsequent chapters, you'll be building on what you have learned here, and you'll see more about how you can extend the capabilities of your classes. In addition, you'll explore more sophisticated ways to use classes in practice. The key points to keep in mind from this chapter are as follows:

- A *class* provides a way to define your own data types. Classes can represent whatever types of *objects* your particular problem requires.
- A class can contain *data members* and *function members*. The function members of a class always have free access to the data members of the same class.
- Objects of a class are created and initialized using function members called *constructors*. A constructor is called automatically when an object declaration is encountered. Constructors can be overloaded to provide different ways of initializing an object.
- Members of a class can be specified as `public`, in which case they are freely accessible from any function in a program. Alternatively, they can be specified as `private`, in which case they may only be accessed by function members or friend functions of the class.
- Data members of a class can be `static`. Only one instance of each static data member of a class exists, no matter how many objects of the class are created.
- Although static data members of a class are accessible in a function member of an object, they aren't part of the object and don't contribute to its size.
- `static` function members can be called even if no objects of the class have been created.
- Every non-`static` function member contains the pointer `this`, which points to the current object for which the function is called.

- A static function member of a class doesn't contain the pointer `this`.
- `const` function members can't modify the data members of a class object unless the data members have been declared as `mutable`.
- Data members that have been specified as `mutable` can always be modified, even when the object is `const`.
- Using references to class objects as arguments to function calls can avoid substantial overheads in passing complex objects to a function.
- A copy constructor is a constructor for an object that is initialized with an existing object of the same class. The compiler generates a default copy constructor for a class if you don't define one.
- A destructor is a function member that called for a class object when it is destroyed. If you don't define a class destructor the compiler supplies a default destructor. #A class has only one destructor and the destructor has no parameters and does not return a value.
- A nested class is a class that is defined inside another class definition.

EXERCISES

The following exercises enable you to try out what you've learned in this chapter.

If you get stuck, look back over the chapter for help. If you're still stuck after that, you can download the solutions from the Apress website (www.apress.com/source-code), but that really should be a last resort.

Exercise 11-1. Create a class called `Integer` that has a single, `private` data member of type `int`. Provide a class constructor that outputs a message when an object is created. Define function members to *get* and *set* the data member, and to output its value. Write a test program to create and manipulate at least three `Integer` objects, and verify that you can't assign a value directly to the data member. Exercise all the class function members by getting, setting, and outputting the value of the data member of each object.

Exercise 11-2. Modify the constructor for the `Integer` class in the previous exercise so that the data member is initialized to zero in the constructor initialization list and implement a copy constructor. Add a function member that compares the current object with an `Integer` object passed as an argument. The function should return `-1` if the current object is less than the argument, `0` if they = objects are equal, and `+1` if the current object is greater than the argument. Try two versions of the `Integer` class, one where the `compare()` function argument is passed by value and the other where it is passed by reference. What do you see output from the constructors when the function is called? Make sure that you understand why this is so.

You can't have both functions present in the class as overloaded functions. Why not?

Exercise 11-3. Implement function members `add()`, `subtract()`, and `multiply()` for the `Integer` class that will add, subtract, and multiply the current object by the value represented by the argument of type `Integer`. Demonstrate the operation of these functions in your class with a version of `main()` that creates `Integer` objects encapsulating values 4, 5, 6, 7, and 8, and then uses these to calculate the value of $4 \times 5^3 + 6 \times 5^2 + 7 \times 5 + 8$. Implement the functions so that the calculation and the output of the result can be performed in a single statement.

Exercise 11-4. Change your solution for Exercise 11-2 so that it implements the `compare()` function as a friend of the `Integer` class.

Exercise 11-5. Modify the `Package` class in `Ex11_08` so that it contains a smart pointer to the previous object in the list. Modify the `Package` and `Truckload` classes to make use of this, including providing the ability to iterate through `Box` objects in the list in reverse order and to list the objects in a `Truckload` object in reverse sequence. Devise a `main()` program to demonstrate the new capabilities.



Operator Overloading

In this chapter, you'll learn how to add support for operators such as add and subtract to your classes so that they can be applied to objects. This will make the types that you define behave more like fundamental data types and offer a more natural way to express some of the operations between objects. You've already seen how classes can have function members that operate on the data members of an object. Operator overloading enables you to write function members that enable the basic operators to be applied to class objects.

In this chapter you will learn:

- What operator overloading is
- Which operators you can implement for your own data types
- What function members the compiler can supply by default
- How to implement function members that overload operators
- How to implement operator functions as ordinary functions
- How to implement the assignment operator and when you *must* implement it
- When you must implement the assignment operator
- How to implement comparison operators for a class
- How to define type conversions as operator functions
- What a function object is and how you create and use it

Implementing Operators for a Class

The Box class in the previous chapter could be applied in an application that is primarily concerned with the volume of a box. For such an application, you obviously need the ability to compare box volumes so that you can determine the relative sizes of the boxes. In Ex11_08, there was this code:

```
if (pBox->compare(*pNextBox) < 0)
    pBox = pNextBox;
```

Wouldn't it be nice if you could write the following instead?

```
if(*pBox < *pNextBox)
    pBox = pNextBox;
```

Using the less-than operator is much clearer and easier to understand than the original. You might also like to add the volumes of two Box objects with an expression such as `Box1 + Box2`, or multiply Box as `10*box1` to obtain a new Box object that has the capacity to hold ten box1 boxes. I'll explain how you can do all this and more by implementing functions that overload the basic operators for objects of a class type.

Operator Overloading

Operator overloading enables you to apply standard operators such as `+`, `-`, `*`, `<`, and so on, to objects of your own class types. To do this, you write a function that redefines each operator that you want to use with your class. In general, the name of a function that overloads a given operator is composed of the operator keyword followed by the operator that you are overloading. In the case of operators that use alphabetic characters, such as `new` and `delete`, there must be at least one space between operator and the operator itself. For operators that are not alphabetic, the space is optional.

Operators That Can Be Overloaded

You can't invent new operators or change operator precedence. You can't change the number of operands. An overloaded version of an operator will have the same precedence and associativity as the original. Although you can't overload all the operators, the restrictions aren't particularly oppressive. Table 12-1 lists the operators that you *can't* overload.

Table 12-1. Operators that Cannot be Overloaded

Operator	Symbol
The scope resolution operator	::
The conditional operator	?:
The direct member access operator	.
The dereference pointer to class member operator	.*
The sizeof operator	sizeof

Anything else is fair game, which gives you quite a bit of scope. You can only overload the `->` operator, the assignment operator `=`, and the array subscript operator `[]`, for a class by function members. A function that overloads any of the other operators for a class doesn't have to be a member; it can be an ordinary function. I'll show examples of both.

Obviously, it's a good idea to make your version of a standard operator reasonably consistent with its normal usage, or at least intuitive in its meaning and operation. It wouldn't be sensible to produce an overloaded `+` operator for a class that performed the equivalent of a multiply. The best way to understand how operator overloading works is to step through an example, so I'll start by explaining how you implement the less-than operator, `<`, for the Box class.

Implementing an Overloaded Operator

A binary operator that is implemented as a class member has one parameter, which I'll explain in a moment. Here's the function member to overload the < operator in the Box class definition:

```
class Box
{
private:
    // Members as before...

public:
    bool operator<(const Box& aBox) const;           // Overloaded 'less-than' operator

    // The rest of the Box class as before...
};
```

Because you're implementing a comparison, the return type is bool. The operator<() function will be called as a result of comparing two Box objects using <. The function will be called as a member of the object that is the left operand and the argument will be the right operand so this will point to the left operand. Because the function doesn't change either operand, the parameter and the function are specified as const. To see how this works, consider the following statement:

```
if(box1 < box2)
    std::cout << "box1 is less than box2" << std::endl;
```

The if expression will result in the operator function being called. The expression is equivalent to the function call box1.operator<(box2). If you were so inclined, you could write it like this in the if statement:

```
if(box1.operator<(box2))
    cout << "box1 is less than box2" << endl;
```

Knowing how the operands in the expression box1 < box2 map to the function call makes implementing the overloaded operator very easy. The definition is shown in Figure 12-1.

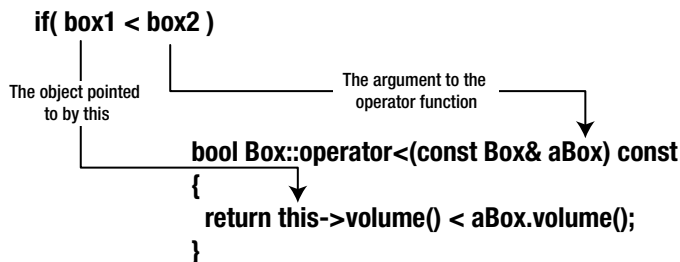


Figure 12-1. Overloading the less-than operator

The reference function parameter avoids unnecessary copying of the argument. The return expression calls the volume() member to calculate the volume of the object pointed to by this and compares that with the volume of aBox using the basic < operator. Thus, true is returned if the object pointed to by this has a smaller volume than the object passed as the argument—and false otherwise.

■ **Note** I used the `this` pointer in Figure 12-1 just to show the association with the first operand. It isn't necessary to use `this` explicitly here.

Let's see if this works in an example. Here's how `Box.h` looks:

```
// Box.h
#ifndef BOX_H
#define BOX_H
#include <iostream>

class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructors
    Box(double lv, double wv, double hv) : length{lv}, width{wv}, height{hv} {}

    Box() {} // No-arg constructor

    Box(const Box& box) : // Copy constructor
        length {box.length}, width {box.width}, height {box.height} {}

    double volume() const // Function to calculate the volume
    { return length*width*height; }

    // Accessors
    double getLength() const { return length; }
    double getWidth() const { return width; }
    double getHeight() const { return height; }

    bool operator<(const Box& aBox) const // Less-than operator
    { return volume() < aBox.volume(); }
};
#endif
```

All function members are defined inside the class so they are all `inline` and `Box.cpp` is not needed. It's important to ensure operator functions are `inline` to maximize efficiency. A program to exercise this is:

```
// Ex12_01.cpp
// Implementing a less-than operator
#include <iostream>
#include <vector>
#include "Box.h"
```

```
int main()
{
    std::vector<Box> boxes {Box {2.0, 2.0, 3.0}, Box {1.0, 3.0, 2.0},
                          Box {1.0, 2.0, 1.0}, Box {2.0, 3.0, 3.0}};
    Box smallBox {boxes[0]};
    for (auto& box : boxes)
    {
        if (box < smallBox) smallBox = box;
    }

    std::cout << "The smallest box has dimensions : "
        << smallBox.getLength() << "x" << smallBox.getWidth() << "x"
        << smallBox.getHeight() << std::endl;
}
```

This produces the following output:

```
The smallest box has dimensions : 1x2x1
```

The `main()` function first creates a vector initialized with four `Box` objects. You arbitrarily assume that the first array element is the smallest, and use it to initialize `smallBox`, which will involve the copy constructor of course. The ranged-based for loop compares each element of `boxes` with `smallBox` and a smaller element is stored in `smallBox` in an assignment statement. When the loop ends, `smallBox` contains the `Box` object with the smallest volume. If you want to track calls of the `operator<()` function, add an output statement to it.

Notice that the assignment operator works with `Box` objects. This is because the compiler supplies a default version of `operator=()` in the class that copies the values of members of the right operand to the members of the left operand. This is not always satisfactory and you'll see later in this chapter how you can define your own version of the assignment operator.

Global Operator Functions

The `volume()` function is a public member of the `Box` class, so you could implement `operator<()` as an ordinary function, not a member of the class. In this case the definition would be:

```
inline bool operator<(const Box& box1, const Box& box2)
{
    return Box1.Volume()<Box2.Volume();
}
```

The `operator<()` function is specified as `inline` because you want it to be compiled as such if possible. With the operator defined in this way, the previous example would work in exactly the same way. Of course, you must not declare this version of the operator function as `const`; `const` only applies to functions that are members of a class. Because this is specified as `inline`, you would put the definition in `Box.h`. This ensures that it's available to any source file that uses the `Box` class.

Even if an operator function needed access to private members of the class, it's still possible to implement it as an ordinary function by declaring it as a friend of the class. Generally though, if a function must access private members of a class, it is better practice to define it as a class member.

Implementing Full Support for an Operator

Implementing an operator such as `<` for a class creates an expectation. You can write expressions like `box1<box2` but what about `box1<25.0`, or `10.0<box2`? The current `operator<()` won't handle either of these. When you implement overloaded operators for a class, you need to consider the likely range of circumstances in which the operator might be used.

You can easily support these possibilities for comparing `Box` objects by adding overloads for `operator<()`. I'll first add a function for `<` where the first operand is a `Box` object and the second operand is of type `double`. I'll define it as an `inline` function with the definition outside the class in this instance, just to show how it's done. You need to add the following member specification to the public section of `Box` class definition:

```
bool operator<(double aValue) const;           // Compare Box volume < double value
```

The `Box` object that is the left operand will be accessed in the function via the implicit pointer `this`, and the right operand is `aValue`. Implementing this is as easy as the first operator function—there's just one statement in the function body:

```
// Compare the volume of a Box object with a constant
inline bool Box::operator<(double aValue) const
{
    return volume() < aValue;
}
```

This definition can follow the class definition in `Box.h`. An `inline` function should not be defined in a `.cpp` file because the definition of an `inline` function must appear in every source file that uses it. Putting it together with the class definition ensures this will always be so. If you put the definition of an `inline` member in a separate source file, it will be in a separate translation unit and you will get linker errors. For consistency I'll define the existing `operator<()` function in the same way in `Box.h`.

Dealing with an expression such as `10.0<box2` isn't harder—it's just different. A *member* operator function always provides the `this` pointer as the left operand. In this case the left operand is type `double` so you can't implement the operator as a function member. That leaves you with two choices: to implement it as an ordinary global operator function, or to implement it as a friend function. Because you don't need to access private members of the class, you can implement it as an ordinary function:

```
// Function comparing a constant with volume of a Box object
inline bool operator<(double aValue, const Box& aBox)
{
    return aValue < aBox.volume();
}
```

This is an `inline` function so you can put it in `Box.h`. You now have three overloaded versions of the `<` operator for `Box` objects to support all three less-than comparison possibilities. Let's see that in action. I'll assume you have modified `Box.h` as described.

Here's a program that uses the new comparison operator functions for `Box` objects:

```
// Ex12_02.cpp
// Using the overloaded 'less-than' operators for Box objects
#include <iostream>
#include <vector>
#include "Box.h"
```

```
// Display box dimensions
void show(const Box& box)
{
    std::cout << "Box" << box.getLength() << "x"
               << box.getWidth() << "x" << box.getHeight() << std::endl;
}

int main()
{
    std::vector<Box> boxes {Box {2.0, 2.0, 3.0}, Box {1.0, 3.0, 2.0},
                          Box {1.0, 2.0, 1.0}, Box {2.0, 3.0, 3.0}};
    double minVol {6.0};
    std::cout << "Objects with volumes less than" << minVol << "are:\n";
    for (auto& box : boxes)
        if (box < minVol) show(box);

    std::cout << "Objects with volumes greater than" << minVol << "are:\n";
    for (auto& box : boxes)
        if (minVol < box) show(box);
}
```

You should get this output:

```
Objects with volumes less than 6 are:
Box 1x2x1
Objects with volumes greater than 6 are:
Box 2x2x3
Box 2x3x3
```

The `show()` function that is defined preceding `main()` outputs the details of the `Box` object that is passed as an argument. This is just a helper function for use in `main()`. The output shows the overloaded operators are working. Again, if you want to see when they are called, put an output statement in each definition. Of course, you don't need separate functions to compare `Box` objects with integers. When this occurs the compiler will insert an implicit cast to type `double` before calling one of the existing functions.

Implementing All Comparison Operators in a Class

We have implemented `<` for the `Box` class but there's still `==`, `<=`, `>`, `>=`, and `!=`. It's a lot, but it's going to be easier than you think. The test for equality for `Box` objects can be defined in the class as:

```
bool operator==(const Box& aBox) const
{ return volume() == aBox.volume(); }
```

Of course, I could plow on and define all the others in the class but I can get some help from the Standard Library. The utility header defines templates for operator functions `<=`, `>`, `>=`, and `!=` for comparing two objects of type `T`. The templates define these operators in terms of the less-than and equality operators, so they must already be defined in a class for the templates to work. Once you have defined the less-than and equality operators, the templates will be used by the compiler to generate the others when required. One advantage of this is that the functions *won't* be generated in a program that doesn't use them, so memory is not occupied by functions that are not used.

The templates for comparison functions are defined in the `rel_ops` namespace that is named from '**relational operators**'. This namespace is nested within the `std` namespace so the function template names are qualified by `std::rel_ops`. This isn't a problem though. If you add the definition for `operator==( )` to the version of the `Box` class from Ex12_02, you can use it to try out some of the templates in the `rel_ops` namespace with the following program;

```
// Ex12_03.cpp
// Using the templates for overloaded comparison operators for Box objects
#include <iostream>
#include <string>
#include <vector>
#include <utility>
#include "Box.h"

using namespace std::rel_ops;

void show(const Box& box1, const std::string relationship, const Box& box2)
{
    std::cout << "Box" << box1.getLength() << "x" << box1.getWidth() << "x" << box1.getHeight()
        << relationship
        << "Box" << box2.getLength() << "x" << box2.getWidth() << "x" << box2.getHeight()
        << std::endl;
}

int main()
{
    std::vector<Box> boxes {Box {2.0, 2.0, 3.0}, Box {1.0, 3.0, 2.0},
                          Box {1.0, 2.0, 1.0}, Box {2.0, 3.0, 3.0}};

    Box theBox {3.0, 1.0, 3.0};

    for (auto& box : boxes)
        if (theBox > box) show(theBox, " is greater than ", box);

    std::cout << std::endl;
    for (auto& box : boxes)
        if (theBox != box) show(theBox, " is not equal to ", box);

    std::cout << std::endl;
    for (size_t i {}; i < boxes.size() - 1; ++i)
        for (size_t j {i+1}; j < boxes.size(); ++j)
        {
            if (boxes[i] <= boxes[j])
                show(boxes[i], " less than or equal to ", boxes[j]);
        }
}
```

The output from this program is:

```
Box 3x1x3 is greater than Box 1x3x2
Box 3x1x3 is greater than Box 1x2x1

Box 3x1x3 is not equal to Box 2x2x3
Box 3x1x3 is not equal to Box 1x3x2
Box 3x1x3 is not equal to Box 1x2x1
Box 3x1x3 is not equal to Box 2x3x3

Box 2x2x3 less than or equal to Box 2x3x3
Box 1x3x2 less than or equal to Box 2x3x3
Box 1x2x1 less than or equal to Box 2x3x3
```

There's a different version of the `show()` helper function; it now outputs a statement about two `Box` objects. You can see that `main()` makes use of the `>`, `!=`, and `<=` operators with `Box` objects. All these are created from the templates that are defined in the utility header. The `using` statement before `main()` is necessary because without it the compiler would not be able to match the operator function names it deduces, such as `operator>()`, with the names of the templates. The `using` statement is in effect from the point at which it appears to the end of the source file. You could put the `using` statement in the body of `main()`, in which case its effect would be restricted to `main()`. The output shows that the three operators are working. Of course, if you need to compare `Box` objects in various ways with other types, you must still implement those.

You could put the `#include` directive for the utility header and the `using` statement for the `std::rel_ops` namespace name in `Box.h`. The disadvantage would be that the utility header would be included into every source file that included `Box.h` and compiled, even when the templates were not used. Also, the names in the `std::rel_ops` namespace would be available without qualification throughout the source file, which could be undesirable in some situations. Of course, if you define one or more of the operator functions in a class for which there are templates in the `std::rel_ops` namespace, the compiler will always call the existing function rather than create a template instance.

Operator Function Idioms

All the binary operators that can be overloaded always have operator functions of the form that you've seen in the previous section. When an operator, `Op`, is overloaded and the left operand is an object of the class for which `Op` is being overloaded, the function member defining the overload is of the form:

```
Return_Type operator Op(Type right_operand);
```

`Return_Type` depends on what the operator does. For comparison and logical operators, it is typically `bool` (although you could use `int`). Operators such as `+` and `*` need to return an object in some form—you'll see how later in this chapter.

You implement a binary operator as a non-function member using this form:

```
Return_Type operator Op(Class_Type left_operand, Type right_operand);
```

`Class_Type` is the class for which you are overloading the operator. `Type` can be any type, including `Class_Type`.

If the left operand for a binary operator is of class `Type`, and `Type` is not the class for which the operator function is being defined, then the function must be implemented as a global operator function of this form:

```
Return_Type operator Op(Type left_operand, Class_Type right_operand);
```

Unary operators defined as function members don't usually require a parameter. The increment and decrement operators are exceptions, as you'll see. The general form of a unary operator function for the operation `Op` as a member of the `Class_Type` class is:

```
Class_Type& operator Op();
```

Unary operators defined as global functions have a single parameter that is the operand. The prototype for a global operator function for a unary operator `Op` is:

```
Class_Type& operator Op(Class_Type& obj);
```

You have no flexibility in the number of parameters for operator functions—either as class members or as global functions. You *must* use the number of parameters specified for the particular operator. I won't go through examples of overloading every operator, as most of them are similar to the ones you've seen. However, I will explain the details of operators that have particular idiosyncrasies when you overload them.

Default Class Members

You know that the compiler can sometimes supply a default constructor and copy constructor. It's interesting to note what the compiler may provide with a very simple class, beyond what you have seen so far. Here's a class with just a single data member:

```
class Data
{
public:
    int value;
};
```

You actually may get the following, assuming your compiler conforms to the current language standard:

```
class Data
{
public:
    int value;

    Data();                               // No-arg constructor
    Data(const Data& aData);               // Copy constructor
    Data(Data&& aData);                     // Move constructor

    ~Data();                               // Destructor

    Data& operator=(const Data& aData);     // Assignment operator
    Data& operator=(const Data&& aData);    // Move assignment operator
};
```

As you see, the compiler supplies up to six function members when required. The *default copy constructor* does member by member copy from the argument object to the members of the new object. This is called a *shallow copy* because the process does not take account of the possibility that the data being copied may refer to other data. If a data member is a pointer, a shallow copy creates an interdependence between objects because two objects will

contain pointers to the same thing. This can result in a lot of problems. The *default assignment operator* provides the same member-by-member copying process from the right operand to the left operand and therefore can potentially cause the same problems.

Don't confuse the copy constructor with the assignment operator function; they are definitely *not* the same. The copy constructor is called when a class object is created from an existing object of the same type, or when an object is passed to a function by value. The assignment operator function is called when the left and right operands of an assignment operator are objects of the same class type.

The *move constructor* and *move assignment operator* have r-value reference parameters so these will be called when the argument is a temporary object. The idea of move operations is that since the argument is temporary, the function doesn't necessarily need to copy data members; it can steal the data from the object that is the argument. If members of the argument object are pointers for example, the pointers can be transferred without copying what they point to because the argument object will be destroyed and so doesn't need them. I'll show you concrete examples of later in this chapter.

The compiler only supplies the default function members if you use them, and of course, if you implement any of these, the default will not be supplied. There are other circumstances where you won't get the defaults. You won't get a default copy constructor and copy assignment operator if you implement a move constructor and a move assignment operator. You won't get a default move constructor and move assignment operator if you define a destructor, a copy constructor, and a copy assignment operator. You can specify that you *do* want a default function member to be supplied by using the `default` keyword, and you can express that you don't want a default member by using the `delete` keyword—for example:

```
class Data
{
public:
    int value;

    Data(int n) : value{n}{}
    Data() = default;                // Supply default no-arg constructor

    Data(const Data& aData)=delete;   // No default copy constructor
    Data& operator=(const Data& aData)=delete; // No assignment operator
};
```

In this case the default no-arg constructor will be supplied by the compiler but the copy constructor and copy assignment operator will not. This is often useful. There are situations where you don't want to allow the duplication of existing objects for example.

Defining the Destructor

You saw in the previous chapter that every class has a destructor and there can only be one. The role of the destructor is to do any necessary clean up when an object is destroyed. You should always define a destructor if a class allocates memory on the heap using the `new` operator. Look at this class definition:

```
#ifndef MESSAGE_H
#define MESSAGE_H
#include <iostream>
#include <string>
```



```

class Message
{
private:
    std::string* ptext;           // Pointer to object text string

public:

    // Function to display a message
    void show() const
    {
        std::cout << "Message is:" << *ptext << std::endl;
    }

    // Constructor
    Message(const char* text = "No message")
    {
        ptext = new std::string {text};    // Allocate space for text
    }

    // Destructor
    ~Message()
    {
        delete ptext;
    }
};
#endif

```

Of course, this is not a sensible way to define the `Message` class but I'm using a raw pointer to a `string` object here to illustrate a point. A smart pointer would remove the need to implement a destructor. The `ptext` member points to an object that is created on the heap using `new` in the constructor so a destructor that releases the memory when an object is destroyed is essential. Without it, you have a memory leak. The more objects a program creates and destroys, the more memory will be allocated and not released. Even with the destructor defined, the class still has a problem. The following code shows what can happen:

```

// Ex12_04.cpp
// Warning this example will crash!
// Defining a destructor
#include "Message.h"

// Output a copy of a Message object
void print(Message message)
{
    message.show();
}

int main()
{
    Message beware {"Careful"};
    print(beware);
    std::cout << "After print() call, output the beware directly:\n";
    beware.show();
}

```

If you compile and run this, the program crashes when trying to output `beware`. The crash is caused by passing `beware` by value to `print()`. Passing an object by value results in the copy constructor being called to provide a copy to the function. In this case it's the default copy constructor because there is no copy constructor defined. The default copy constructor copies the address from `pnext` in the original object to `pnext` in the new object. When the `print()` function returns, the copy of the argument is destroyed so the `Message` class destructor is called. This deletes the memory pointed to by the `pnext` member of the copy. Unfortunately the `pnext` member of the original object points to the same memory that has been released—hence the crash. If you need to write a destructor for a class, you will usually need to implement a copy constructor and the assignment operator for the class too.

When to Define a Copy Constructor

If a class has data members that are pointers—and this includes smart pointers, you should implement the copy constructor. If you don't, the default copy constructor will copy an object by copying the values of the data members, which means just the addresses for pointers will be copied—not what they point to. The result will be two or more objects with members pointing to the same object. A change to an object that is pointed to by a data member of one object will affect all the duplicate objects. With members that are smart pointers, the interdependence between objects will be a problem. With members that are raw pointers it is likely to be disastrous, and `Ex12_04` illustrates how catastrophic this can be. Interdependence between objects is not what you want most of the time. You must implement the copy constructor so that it duplicates the object that a data member points to. If you add the following definition to the `Message` class in `Ex12_04`, you'll see that the example then works OK:

```
Message(const Message& message)
{
    pnext = new std::string(*message.pnext); // Duplicate the object in the heap
}
```

The copy constructor duplicates the `string` object to which the `pnext` member of the argument to the copy constructor points, so the new object is independent of the original. When the original object is destroyed, its memory is released without resulting in any problems for the duplicate.

Implementing the Assignment Operator

As you've seen, the default assignment operator copies the members of the object on the right of an assignment to the members of the object of the same type on the left, which can cause the same problems as the default copy constructor. You call the assignment operator when you write the following:

```
Message message;
    Message beware {"Careful"};
message = beware; // Call the assignment operator
```

The default assignment operator for the `Message` class will cause exactly the same problem as the default copy constructor. Any class that has problems with the default copy constructor will also have problems with the default assignment operator and vice versa. If you need to implement one, you also need to implement the other.

You saw earlier that the assignment operator returns a reference, so in the `Message` class it would look like this:

```
Message& operator=(const Message& message); // Assignment operator
```

The parameter is a `const` reference and the return type is a non-`const` reference. The code for the assignment operator will transfer data from the members of the right operand to the members of the left operand, so you may wonder why it has to return a reference—or indeed, why it needs to return anything. Consider how the assignment operator is applied in practice. With normal usage you can write this:

```
message1 = message2 = message3;
```

These are three objects of the same type so this statement makes `message1` and `message2` copies of `message3`. Because the assignment operator is right associative, this is equivalent to:

```
message1 = (message2 = message3);
```

The result of executing the rightmost assignment is evidently the right operand for the leftmost assignment so you definitely need to return something. In terms of `operator=()`, this statement is equivalent to:

```
message1.operator=(message2.operator=(message3));
```

It's clear from this that whatever you return from `operator=()` can end up as the argument to another `operator=()` call. The parameter for `operator=()` is a reference to an object so the operator function must return the left operand, which is the object that is pointed to by `this`. Further, to avoid unnecessary copying of the object that is returned, the return type must be a reference.

The process for duplicating the right operand is the same as the one you used for the copy constructor, and because you now know what the return type should be, you can have a first stab at defining the assignment operator function for the `Message` class. First consider the following, and then I'll explain what's wrong with it:

```
Message& operator=(const Message& message)
{
    ptext = new std::string(*message.ptext); // Duplicate the object in the heap
    return *this;                          // Return the left operand
}
```

The `this` pointer contains the address of the left argument, so returning `*this` returns the object. Apart from that, this code is the same as the copy constructor. The function looks OK, and it appears to work most of the time, but there is a problem with it. Suppose someone writes:

```
message1 = message1;
```

The likelihood of someone writing this explicitly is very low, but it could occur indirectly. The result of this statement is that you replace the current address in `ptext` with the address of a copy of the original `string` object. The original `string` object has been cast adrift and its memory cannot be released. In other words, you have a memory leak. The solution is to check for identical left and right operands:

```
Message& operator=(const Message& message)
{
    if(this != &message)
        ptext = new std::string(*message.ptext); // Duplicate the object in the heap

    return *this;                                // Return the left operand
}
```

Now if this contains the address of the argument object, the function does nothing and just returns the same object. If you put this in the `Message` class definition, the following code will show it working:

```
// Ex12_05.cpp
// Defining a destructor and the copy constructor
#include "Message.h"

// Output a copy of a Message object
void print(Message message)
{
    message.show();
}
int main()
{
    Message beware {"Careful"};
    Message warning;

    warning = beware;                // Call assignment operator
    std::cout << "After assignment beware is:\n";
    beware.show();
    std::cout << "After assignment warning is:\n";
    warning.show();
}
```

The output will demonstrate that everything works as it should. You're not limited to overloading the copy assignment operator just to copy an object. You can have several overloaded versions of the assignment operator for a class. Additional versions can have a parameter type that is different from the class type, so they are effectively conversions. In any event, the return type should be a reference to the left operand. Of course, you can also overload the `op=` operators too.

■ **Golden Rule** If a class has members that are pointers, always implement a copy constructor and an assignment operator; if it has members that are raw pointers, always define a destructor too.

Implementing Move Operations

Implementing move operations is an advanced subject so I'll just introduce it briefly. The value of a move constructor and a move assignment operator in a class is the improved efficiency that results from stealing resources from a temporary object rather than copying them when copying is expensive on time. This typically applies when there are members that are pointers—yet again! If the members are raw pointers, particular care is necessary because the destructor for the argument object that is being cannibalized can cause problems. The `Message` class will show this. Here's a definition of a move constructor for the `Message` class:

```
Message(Message&& message)
{
    ptext = message.ptext;           // Steal the string object - no need to copy it
    message.ptext = nullptr;        // A most important operation!
}
```

The parameter is an rvalue reference so this constructor will only be called when the argument is a temporary object. This means that the message object will be destroyed when the execution of the constructor ends. It is therefore possible to steal the message that the `message.ptext` member points to just by copying the address. The second statement in the body of the constructor is of the utmost importance. If `message.ptext` is not set to `nullptr`, the destructor for the message object will delete the string object from the heap so the `ptext` member of the new object will contain an invalid address, which will surely cause the program to crash.

The circumstances under which the move constructor for the `Message` class is called are rather unlikely. This statement would do it:

```
Message message { Message {"Tell it to them."} };
```

A `Message` object is created in the initializer list for `message` so this will cause the move constructor to be called. This is not a sensible statement though because just putting the literal in the initializer list will produce the same result. The move assignment operator is even less likely to be required for the `Message` class. However, there are classes where both can be used frequently, and the `std::string` class is an excellent example. Here's how the move constructor and the move assignment operator for the `string` class gets called:

```
string word1{"move"};
string word2{"assignment"};
string combined {word1 + " " + word2 };    // Calls move constructor
combined = combined + "operator";         // Calls move assignment operator
```

The value in the initializer list in the third statement will be the temporary `string` object that results from the expression between the braces, which happens to use the overloaded `operator+()` member of the `string` class. The right operand of the assignment in the fourth statement is also a temporary `string` object, so this will call the move assignment operator for the class.

Implementing the move assignment operator for a class is similar to implementing the copy assignment operator but with two differences: there's no need to check for identical left and right operands because that cannot arise, and pointer members are just copied without duplicating what they point to. It's still essential that any raw pointer members of the argument object are reset to `nullptr` after copying and the function should still return the left operand.

Overloading the Arithmetic Operators

I'll explain how you overload the arithmetic operators by looking at how you might overload the addition operator for the `Box` class. This is an interesting example, because addition is a binary operation that involves creating and returning a new object. The new object will be the sum (whatever you define that to mean) of the two `Box` objects that are its operands.

What might the sum of two `Box` objects mean? There are several possibilities we could consider, but because the primary purpose of a box is to hold something, its volumetric capacity is of primary interest so we might reasonably presume that the sum of two boxes was a new box that could hold both. Using this assumption, I'll define the sum of two `Box` objects to be a `Box` object that's large enough to contain the two original boxes stacked on top of each other. This is consistent with the notion that the class might be used for packaging, because adding several `Box` objects together results in a `Box` object that can contain all of them.

You can implement the addition operator in a simple way, as follows. The `length` member of the new object will be the larger of the `length` members of the objects being summed and a `width` member will be determined in a similar way. If the `height` member is the sum of the `height` members of the operands, the resultant `Box` object can contain the two `Box` objects. By modifying the constructor, I'll arrange that the `length` member of an object is always greater than or equal to the `width` member.

Figure 12-2 illustrates the Box object that will be produced by adding two Box objects. Because the result of this addition is a new Box object, the function implementing addition must return a Box object. If the function that overloads the + operator is to be a function member, then the declaration of the function in the Box class definition can be:

```
Box operator+(const Box& aBox) const;    // Adding two Box objects
```

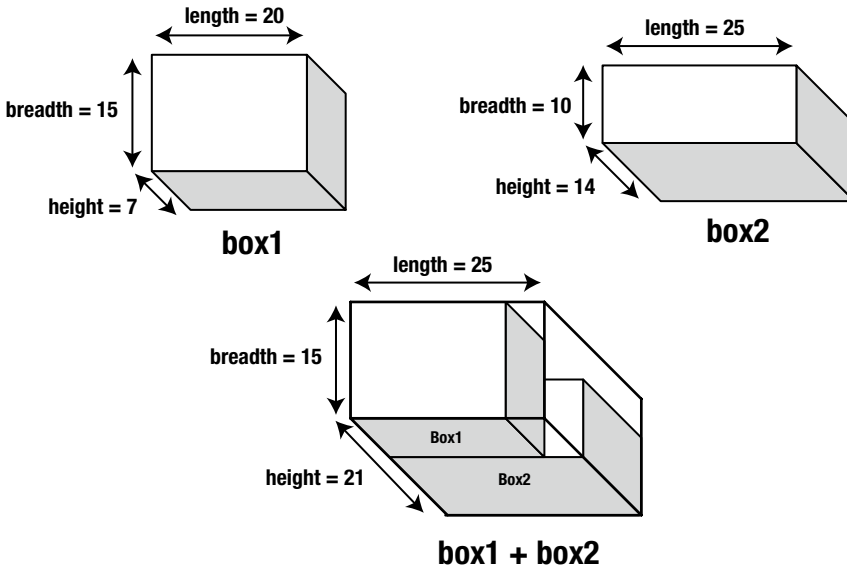


Figure 12-2. The object that results from adding two Box objects

The *parameter* is `const` because the function won't modify the argument, which is the right operand. It's a `const` reference to avoid unnecessary copying of the right operand. The *function* is specified as `const` because it doesn't alter the left operand. The definition of the function member in `Box.h` will be:

```
// Operator function to add two Box objects
inline Box Box::operator+(const Box& aBox) const
{
    // New object has larger length and width, and sum of heights
    return Box{ length > aBox.length ? length : aBox.length,
               width > aBox.width ? width : aBox.width,
               height + aBox.height };
}
```

Notice that you don't create a Box object in the free store to return to the caller. This would be a very poor way of implementing the function because it's difficult to ensure that the memory is released when the object is destroyed. Returning a pointer would also affect how other operators, such as `operator=()`, are written. Here, a local Box object is created and a *copy* of that is returned to the calling program. Because these are automatic variables, the memory management is taken care of automatically.

We can see how the addition operator works in an example. I'll modify the Box class from Ex12_03:

```
// Box.h
#ifndef BOX_H
#define BOX_H
#include <iostream>
#include <iomanip>
#include <algorithm>                // For max() and min() functions

class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructors
    Box(double lv, double wv, double hv) :
        length {std::max(lv,wv)}, width {std::min(lv,wv)}, height {hv} {}
    Box()=default;

    Box(const Box& box) :                // Copy constructor
        length {box.length}, width {box.width}, height {box.height} {}

    double volume() const                // Function to calculate the volume
    {
        return length*width*height;
    }

    // Accessors
    double getLength() const { return length; }
    double getWidth() const { return width; }
    double getHeight() const { return height; }

    bool operator<(const Box& aBox) const;                // Less-than operator
    bool operator<(double aValue) const;                // Compare Box volume < double value
    Box operator+(const Box& aBox) const;                // Function to add two Box objects
    void listBox();                // Output the Box
};
```

The new constructor definition uses the `max()` and `min()` functions that are defined by templates in the `algorithm` header. They return the maximum and minimum respectively of the two arguments. These functions work with any argument types that support `operator<()`. The Box class uses the default keyword to get the default constructor supplied by the compiler when necessary. The inline definition of the `operator+()` function you saw earlier also goes in `Box.h`. I added the `listBox()` member from Ex11_08 to output a Box object. This is defined immediate following the class definition as:

```
inline void Box::listBox()
{
    std::cout << " Box(" << std::setw(2) << length << ", "
        << std::setw(2) << width << ", "
        << std::setw(2) << height << ")";
}
```

Here's the code to try it out:

```
// Ex12_06.cpp
// Using the addition operator for Box objects
#include <iostream>
#include <vector>
#include <cstdlib>           // For random number generator
#include <ctime>             // For time function
#include "Box.h"

using namespace std::rel_ops;

// Function to generate integral random box dimensions from 1 to max_size
inline double random(double max_size)
{
    return 1 + static_cast<int>(max_size* static_cast<double>(std::rand())/(RAND_MAX + 1.0));
}

int main()
{
    const double dimLimit {99.0};           // Upper limit on Box dimensions
    std::srand((unsigned)std::time(0));     // Initialize the random number generator

    const size_t boxCount {20};             // Number of Box object to be created
    std::vector<Box> boxes;                 // Vector of Box objects

    // Create 20 Box objects
    for (size_t i {}; i < boxCount; ++i)
        boxes.push_back(Box {random(dimLimit), random(dimLimit), random(dimLimit)});

    size_t first {};                        // Index of first Box object of pair
    size_t second {1};                     // Index of second Box object of pair
    double minVolume {(boxes[first] + boxes[second]).volume()};

    for (size_t i {}; i < boxCount - 1; ++i)
        for (size_t j {i + 1}; j < boxCount; j++)
            if (boxes[i] + boxes[j] < minVolume)
            {
                first = i;
                second = j;
                minVolume = (boxes[i] + boxes[j]).volume();
            }

    std::cout << "The two boxes that sum to the smallest volume are:";
    boxes[first].listBox();
    boxes[second].listBox();
    std::cout << "\nThe volume of the first box is" << boxes[first].volume();
    std::cout << "\nThe volume of the second box is" << boxes[second].volume();
    std::cout << "\nThe box that the sum of these boxes is";
    (boxes[first] + boxes[second]).listBox();
    std::cout << "\nThe volume of the sum is" << minVolume << std::endl;
}
```


I got the following output:

```
The two boxes that sum to the smallest volume are: Box( 4,87, 5) Box(28,22,10)
The volume of the first box is 1740
The volume of the second box is 6160
The box that the sum of these boxes is Box(28,87,15)
The volume of the sum is 36540
```

You should get a different result each time you run the program. Just to emphasize what I have said previously—the `rand()` function is OK when you don't care about the quality of the random number sequence but when you need something better, use the pseudo-random number generation facilities provided by the `random` Standard Library header.

The `main()` function generates a vector of twenty `Box` objects that have arbitrary integral dimensions from 1.0 to 99.0. The nested for loops then test all possible pairs of `Box` objects to find the pair that combines to the minimum volume. The `if` statement in the inner loop uses the `operator+()` member to produce a `Box` object that is the sum of the current pair of objects. The `operator<()` member is then used to compare this resultant `Box` object with the value of `minVolume`. The output shows that everything works at it should. I suggest you instrument the operator functions and the `Box` constructors just to see when and how often they are called.

Of course, you can use the overloaded addition operator in more complex expressions to sum `Box` objects. For example, you could write this:

```
Box box4 {box1 + box2 + box3};
```

This calls the `operator+()` member twice to create a `Box` object that is the sum of the three, and this is passed to the copy constructor for the `Box` class to create `box4`. The result is a `Box` object `box4` that can contain the other three `Box` objects stacked on top of each other.

You could implement the addition operation for the class as a non-function member, because the dimensions of a `Box` object are accessible through public function members. Here's the prototype of such a function:

```
Box operator+(const Box& aBox, const Box& bBox);
```

If the values of the data members were inaccessible, you can still write it as a normal function that you declared as a friend function within the `Box` class. Of these choices, the friend function is always the least desirable. Operator functions are fundamental to class capability, so I prefer to implement them as class members, which makes the operations integral to the type.

Improving Output Operations

Now we know how to overload operators we could make the output statements for `Box` objects better by overloading the `<<` operator for output streams. The standard output stream, `cout`, is of type `std::ostream`, as are other output streams that you'll meet later in the book. The dimensions of a `Box` object are available through accessor functions so we can define `operator<<()` as an ordinary function, like this:

```
std::ostream& operator<<(std::ostream& stream, const Box& box)
{
    stream << " Box(" << std::setw(2) << box.getLength() << ", "
        << std::setw(2) << box.getWidth() << ", "
        << std::setw(2) << box.getHeight() << ")";

    return stream;
}
```

The first parameter identifies the left operand as an ostream object and the second specifies the right operand as a Box object. The return type is a reference so the stream object can be used in further output operations using the << operator. Here's a variation on the previous example that uses this:

```
// Ex12_06.cpp
// Using the addition operator for Box objects
#include <iostream>
#include <vector>
#include <cstdlib> // For random number generator
#include <ctime> // For time function
#include "Box.h"

// Stream output for Box objects
std::ostream& operator<<(std::ostream& stream, const Box& box)
{
    stream << " Box(" << std::setw(2) << box.getLength() << ","
        << std::setw(2) << box.getWidth() << ","
        << std::setw(2) << box.getHeight() << ")";
    return stream;
}

// Function to generate integral random box dimensions from 1 to max_size
inline double random(double max_size)
{
    return 1 + static_cast<int>(max_size* static_cast<double>(std::rand())/(RAND_MAX + 1.0));
}

int main()
{
    const double dimLimit {99.0}; // Upper limit on Box dimensions
    std::srand((unsigned) std::time(0)); // Initialize the random number generator

    const size_t boxCount {20}; // Number of Box object to be created
    std::vector<Box> boxes; // Vector of Box objects

    // Create 20 Box objects
    for (size_t i {}; i < boxCount; ++i)
        boxes.push_back(Box {random(dimLimit), random(dimLimit), random(dimLimit)});

    size_t first {}; // Index of first Box object of pair
    size_t second {1}; // Index of second Box object of pair
    double minVolume {(boxes[first] + boxes[second]).volume()};

    for (size_t i {}; i < boxCount - 1; ++i)
        for (size_t j {i + 1}; j < boxCount; j++)
            if (boxes[i] + boxes[j] < minVolume)
            {
                first = i;
                second = j;
                minVolume = (boxes[i] + boxes[j]).volume();
            }
}
```

```

std::cout << "The two boxes that sum to the smallest volume are:"
    << boxes[first] << boxes[second];
std::cout << "\nThe volume of the first box is" << boxes[first].volume();
std::cout << "\nThe volume of the second box is" << boxes[second].volume();
std::cout << "\nThe box that the sum of these boxes is" << boxes[first] + boxes[second];
std::cout << "\nThe volume of the sum is" << minVolume << std::endl;
}

```

Now the output statements are more natural. You can output a `Box` object to `cout` just like a variable of a fundamental type. It should be apparent now that using the `<<` operator for output to the standard stream is achieved by overloading the operator for the `ostream` class. You also should be able to figure out how it comes about that outputting a pointer of type `char*` writes a C-style string whereas outputting any other pointer, including a smart pointer, writes the address it contains. The fact that type `char*` is treated as a special case implies that there must be a specific implementation of `operator<<()` to do this. On the other hand, the fact that outputting any other type of pointer, including pointers to any class that you define, outputs the address the pointer contains means that this must be the result of a function template for `operator<<()`.

■ **Note** Overloading `<<` to allow outputting objects of your own class types to a stream has to be an ordinary function. In the example, the function can access data members of the `Box` class through accessor function members. If this was not the case, this would be an example of where a `friend` function declaration in the `Box` class would be necessary.

Implementing One Operator in Terms of Another

One thing always leads to another. If you implement the addition operator for a class, you inevitably create the expectation that the `+=` operator will work too. If you are going to implement both, it's worth noting that you can implement `+` in terms of `+=` very economically.

First, I'll define `+=` for the `Box` class. Because assignment is involved, the operator function needs to return a reference:

```

// Overloaded += operator
inline Box& Box::operator+=(const Box& right)
{
    length = length > right.length ? length : right.length;
    width = width > right.width ? width : right.width;
    height += right.height;
    return *this;
}

```

This is very straightforward. You simply modify the left operand, which is `*this`, by adding the right operand according to the definition of addition for `Box` objects. You can now implement `operator+()` using `operator+=()`, so the definition of `operator+()` simplifies to:

```

// Function to add two Box objects
inline Box Box::operator+(const Box& aBox) const
{
    return Box(*this) += aBox;
}

```

The expression `Box(*this)` calls the copy constructor to create a copy of the left operand to use in the addition. The `operator+=()` function is then called to add the right operand object, `right`, to the new `Box` object. This object is then returned.

Overloading the Subscript Operator

The subscript operator `[]` provides very interesting possibilities for certain kinds of classes. Clearly, this operator is aimed primarily at selecting one of a number of objects that you can interpret as an array; but where the objects could be contained in any one of a number of different containers. You can overload the subscript operator to access the elements of a sparse array (where many of the elements are empty), or an associative array, or even a linked list. The data might even be stored in a file, and you could use the subscript operator to hide the complications of file input and output operations.

The `Truckload` class from `Ex11_09` in Chapter 11 is an example of a class that could support the subscript operator. A `Truckload` object contains an ordered set of objects, so the subscript operator could provide a means of accessing these objects through an index value. An index of 0 would return the first object in the list, an index of 1 would return the second; and so on. The inner workings of the subscript operator would take care of iterating through the list to find the object required.

The `operator[]()` function for the `Truckload` class needs to accept an index value as an argument that is a position in the list and to return the pointer to the `Box` object at that position. The declaration for the function member in the `Truckload` class is:

```
class Truckload
{
private:
    // Members as before...

public:
    ptr<Box> operator[](size_t index) const;           // Overloaded subscript operator
    // Rest of the class as before...
};
```

You could implement the function like this:

```
inline ptr<Box> Truckload::operator[](size_t index) const
{
    ptr<Package> p {pHead};                          // Pointer to first Package
    size_t count {};                                  // Package count
    do {
        if (index == count++)                          // Up to index yet?
            return p->pBox;                             // If so return the pointer to Box
    } while (p = p->pNext);
    return nullptr;
}
```

The do-while loop traverses the list, incrementing the count on each iteration. When the value of count is the same as index, the loop has reached the `Package` object at position index, so the smart pointer to the `Box` object in that `Package` object is returned. If the entire list is traversed without count reaching the value of index, then index must be out of range, so `nullptr` is returned. Let's see how this pans out in practice by trying another example.

This example will use the `Box` class from `Ex12_06`, but with `operator<<()` implemented to allow `Box` objects to be written to an output stream. Add the following friend declaration to the `Box` class definition:

```
friend std::ostream& operator<<(std::ostream& stream, const Box& box);
```

The friend function can be inline so you can put it in `Box.h` following the class definition:

```
inline std::ostream& operator<<(std::ostream& stream, const Box& box)
{
    stream << " Box(" << std::setw(2) << box.length << ", "
        << std::setw(2) << box.width << ", "
        << std::setw(2) << box.height << ")";
    return stream;
}
```

Because it's an inline friend function, `operator<<()` comes along with the `Box` class automatically when the class is included into a source file. The implementation as a friend allows access to the private members of the `Box` class directly so it does not require accessor functions for the data members to be present in the class.

We can remove the `listBoxes()` member of `Truckload` and add an overload for the `<<` operator for outputting `Truckload` objects to a stream as a friend, analogous to the `Box` class. The definition for it as an inline function is:

```
inline std::ostream& operator<<(std::ostream& stream, Truckload& load)
{
    load.pCurrent = load.pHead;
    size_t count {};
    while (load.pCurrent)
    {
        std::cout << *(load.pCurrent->pBox);
        load.pCurrent = load.pCurrent->pNext;
        if (!(++count % 5)) std::cout << std::endl;
    }
    if (count % 5) std::cout << std::endl;
    return stream;
}
```

The code is similar to that for `listBoxes()` except that now the members are identified as belonging to the `load` parameter and members of `Package` objects are accessed directly. The function makes use of the `operator<<()` function that is a friend of the `Box` class. Outputting a `Truckload` object will now be very simple—you just use `<<` to write it to `cout`. With the subscript operator and the stream output operator added to the `Truckload` class, `Truckload.h` will contain:

```
// Truckload.h
#ifndef TRUCKLOAD_H
#define TRUCKLOAD_H

#include <memory>
#include <vector>
#include "Box.h"
template <typename T> using ptr = std::shared_ptr<T>;
```

```

class Truckload
{
private:
    class Package
    {
    public:
        ptr<Box> pBox;                // Pointer to the Box object
        ptr<Package> pNext;           // Pointer to the next Package

        Package(ptr<Box> pb) : pBox {pb}, pNext {} {} // Constructor
    };

    ptr<Package> pHead;                // First in the list
    ptr<Package> pTail;                // Last in the list
    ptr<Package> pCurrent;             // Last retrieved from the list

public:
    Truckload() {}                    // No-arg constructor empty truckload

    Truckload(ptr<Box> pBox)           // Constructor - one Box
    {
        pHead = pTail = std::make_shared<Package>(pBox);
    }

    Truckload(const std::vector< ptr<Box> >& boxes); // Constructor - vector of Boxes

    ptr<Box> getFirstBox();             // Get the first Box
    ptr<Box> getNextBox();              // Get the next Box
    void addBox(ptr<Box> pBox);         // Add a new Box
    bool deleteBox(ptr<Box> pBox);      // Delete a Box
    ptr<Box> operator[](size_t index) const; // Overloaded subscript operator

    friend std::ostream& operator<<(std::ostream& stream, Truckload& load);
};

// Subscript operator
inline ptr<Box> Truckload::operator[](size_t index) const
{
    ptr<Package> p {pHead};            // Pointer to first Package
    size_t count {};                  // Package count
    do {
        if (index == count++)         // Up to index yet?
            return p->pBox;            // If so return the pointer to Box
    } while (p = p->pNext);
    return nullptr;
}

```

```
// >> operator for output to a stream
inline std::ostream& operator<<(std::ostream& stream, Truckload& load)
{
    load.pCurrent = load.pHead;
    size_t count {};
    while (load.pCurrent)
    {
        std::cout << *(load.pCurrent->pBox);
        load.pCurrent = load.pCurrent->pNext;
        if (!(++count % 5)) std::cout << std::endl;
    }
    if (count % 5) std::cout << std::endl;
    return stream;
}
#endif
```

listBoxes() is no longer a member of the Truckload class so you must remove its definition from Truckload.cpp. The code to exercise The Truckload class with its subscript operator is:

```
// Ex12_07.cpp
// Using the subscript operator
#include <iostream>
#include <memory>
#include <cstdlib> // For random number generator
#include <ctime> // For time function
#include "Truckload.h"

// Function to generate integral random box dimensions from 1 to max_size
inline double random(double max_size)
{
    return 1 + static_cast<int>(max_size* static_cast<double>(std::rand()) / (RAND_MAX + 1.0));
}

int main()
{
    const double dimLimit {99.0}; // Upper limit on Box dimensions
    std::srand((unsigned) std::time(0)); // Initialize the random number generator
    Truckload load;
    const size_t boxCount {20}; // Number of Box object to be created

    // Create boxCount Box objects
    for (size_t i {}; i < boxCount; ++i)
        load.addBox(std::make_shared<Box>(random(dimLimit), random(dimLimit), random(dimLimit)));

    std::cout << "The boxes are:\n";
    std::cout << load;
```

```

// Find the largest Box in the list
double maxVolume {};
size_t maxIndex {};
size_t i {};
while (load[i] )
{
    if (load[i]->volume() > maxVolume)
    {
        maxIndex = i;
        maxVolume = load[i]->volume();
    }
    ++i;
}

std::cout << "\nThe largest box is:";
std::cout << *load[maxIndex] << std::endl;

load.deleteBox(load[maxIndex]);
std::cout << "\nAfter deleting the largest box, the list contains:\n";
std::cout << load;
}

```

When I ran this example, it produced the following output:

```

The largest box in the list is
90 by 79 by 77The boxes are:
Box(26,68,23) Box(89,60,94) Box(46,82,27) Box(22, 2,29) Box(98,23,90)
Box(25,81,55) Box(52,64,28) Box(98,33,40) Box(83,14,80) Box(91,78,94)
Box(28,54,50) Box(57,79,18) Box(91,89,99) Box(26,39,57) Box(26,42,35)
Box(15,29,74) Box(10,17,21) Box(91,86,68) Box(94, 5,30) Box(87,10,94)

```

The largest box is: Box(91,89,99)

```

After deleting the largest box, the list contains:
Box(26,68,23) Box(89,60,94) Box(46,82,27) Box(22, 2,29) Box(98,23,90)
Box(25,81,55) Box(52,64,28) Box(98,33,40) Box(83,14,80) Box(91,78,94)
Box(28,54,50) Box(57,79,18) Box(26,39,57) Box(26,42,35) Box(15,29,74)
Box(10,17,21) Box(91,86,68) Box(94, 5,30) Box(87,10,94)

```

The `main()` function now uses the subscript operator to access pointers to `Box` objects from the `Truckload` object. You can see from the output that the subscript operator works and the result of finding and deleting the largest `Box` object is correct. The subscript operator masks an inefficient process in this case though. It's easy to forget that each use of the subscript operator involves traversing at least part of the list from the beginning. More than one access to an entry at a given index would be best avoided, especially if the `Truckload` object contains a large number of pointers to `Box` objects. Output of `Truckload` and `Box` objects to the standard output stream now works the same as for fundamental types.

lvalues and the Overloaded Subscript Operator

You'll encounter circumstances under which you might want to overload the subscript operator and use the object it returns as an lvalue—that is, on the left of an assignment. With your present implementation of `operator[]()` in the `Truckload` class, a program compiles but won't work correctly if you write this:

```
load[0] = load[1];
```

This will compile and execute but it won't affect the items in the list. What you want is that the first pointer in the list is replaced by the second, but this doesn't happen. One problem is the return value from `operator[]()`. The function returns a *temporary copy* of a smart pointer object that points to the same `Box` object as the original pointer in the list, but is a *different pointer*. The assignment operates, but is changing just a copy of the first pointer in the list, which won't be around for very long. Of course, each time you use `load[0]` on the left of an assignment, you get a *different* copy of the first pointer in the list.

To allow the subscript operator to be used on the left of an assignment you must define the operator so that it returns a reference that *can* be used as an lvalue. Obviously, you must not return a reference to a local object in this situation. Doing this for the `Truckload` class has significant ramifications.

First, you cannot return `nullptr` from `operator[]()` in the `Truckload` class because you cannot return a reference to `nullptr`. You need to devise another way to deal with an invalid index. One possibility is to return a `ptr<Box>` object that doesn't point to anything, but this cannot be a local object. Second, the `getBox()` member of the `Package` class also must return a reference, which means the function member cannot be `const`.

You could define a `ptr<Box>` object as a static member of the `Truckload` class by adding the following declaration to the private section of the class:

```
static ptr<Box> nullBox;                                // Pointer to nullptr
```

As you saw in Chapter 11, you initialize static class members outside the class. The following statement in `Truckload.cpp` will do it:

```
ptr<Box> Truckload::nullBox {};                        // Initialize static class member
```

Now we can change the definition of the subscript operator to:

```
inline ptr<Box>& Truckload::operator[](size_t index) const
{
    ptr<Package> p {pHead};                            // Pointer to first Package
    size_t count {};                                    // Package count
    do {
        if (index == count++)                          // Up to index yet?
            return p->pBox;                             // If so return the pointer to Box
    } while (p = p->pNext);
    return nullBox;
}
```

This allows stepping through the elements in the list. It now returns a reference to the pointer and the function member is no longer `const`. Here's an extension of `Ex12_07` to try out the subscript operator on the left of an assignment. I have simply extended `main()` from `Ex12_07` to show that iterating through the elements in a `Truckload` list still works:

```
// Ex12_08.cpp
// Using the subscript operator on the left of an assignment
#include <iostream>
#include <memory>
#include <cstdlib>           // For random number generator
#include <ctime>             // For time function
#include "Truckload.h"

// Function to generate integral random box dimensions from 1 to max_size
inline double random(double max_size)
{
    return 1 + static_cast<int>(max_size* static_cast<double>(std::rand()) / (RAND_MAX + 1.0));
}

int main()
{
    // All the code from main() in Ex12_07 here...

    load[0] = load[1];           // Copy 2nd element to 1st
    std::cout << "\nAfter copying the 2nd element to the 1st, the list contains:\n";
    std::cout << load;

    load[1] = std::make_shared<Box>(*load[2] + *load[3]);
    std::cout << "\nAfter making the 2nd element a pointer to the 3rd plus 4th,"
                " the list contains:\n";

    std::cout << load;
}
```

The first part of the output is similar to the previous example, after which the output is:

After copying the 2nd element to the 1st, the list contains:

```
Box(65,31, 6) Box(65,31, 6) Box(75, 4, 4) Box(40,18,48) Box(32,67,21)
Box(78,48,72) Box(22,71,41) Box(36,37,91) Box(19, 9,71) Box(98,78,30)
Box(85,54,53) Box(98,13,66) Box(50,57,39) Box(56,80,88) Box(17,60,23)
Box(85,42,41) Box(51,31,61) Box(41, 9, 8) Box(75,79,43)
```

After making the 2nd element a pointer to the sum of 3rd and 4th, the list contains:

```
Box(65,31, 6) Box(75,18,52) Box(75, 4, 4) Box(40,18,48) Box(32,67,21)
Box(78,48,72) Box(22,71,41) Box(36,37,91) Box(19, 9,71) Box(98,78,30)
Box(85,54,53) Box(98,13,66) Box(50,57,39) Box(56,80,88) Box(17,60,23)
Box(85,42,41) Box(51,31,61) Box(41, 9, 8) Box(75,79,43)
```

The first block of output shows that the first two elements point to the same Box object so the assignment worked as expected. The second block of output results from assigning a new value to the second element in the Truckload object; the new value is a pointer to the Box object produced by summing the third and fourth Box objects. The output shows that the second element points to a new object that is the sum of the next two. Just to make it clear what is happening, the statement that does this is equivalent to:

```
load.operator[](1).operator=(*(load.operator[](2)).operator+(*(load.operator[](3))));
```

That's much much clearer, isn't it?

Overloading Type Conversions

You can define an operator function as a class member to convert from the class type to another type. The type you're converting to can be a fundamental type or a class type. Operator functions that are conversions for objects of an arbitrary class, `Object`, are of this form:

```
class Object
{
public:
    operator Type();                // Conversion from Object to Type
// Rest of Object class definition...
};
```

`Type` is the destination type for the conversion. Note that no return type is specified because the target type is always implicit in the function name, so here the function must return a `Type` object.

As an example, you might want to define a conversion from type `Box` to type `double`. For application reasons, you could decide that the result of this conversion would be the volume of the `Box` object. You could define this as follows:

```
class Box
{
public:
    operator double() const { return volume(); }

// Rest of Box class definition...
};
```

The operator function would be called if you wrote this:

```
Box box {1.0, 2.0, 3.0};
double boxVolume {};
boxVolume = box;                // Calls conversion operator
```

This causes an implicit conversion to be inserted by the compiler. You could call the operator function explicitly with this statement:

```
double total { 10.0 + static_cast<double>(box) };
```

You can prevent implicit calls of a conversion operator function by specifying it as `explicit` in the class. In the `Box` class you could write:

```
explicit operator double() const { return volume(); }
```

Now the compiler will not use this member for implicit conversions to type `double`.

Potential Ambiguities with Conversions

When you implement conversion operators for a class, it is possible to create ambiguities that will cause compiler errors. You have seen that a constructors can also effectively implement a conversion—a conversion from type `Type1` to type `Type2` can be implemented by including a constructor in class `Type2` with this declaration:

```
Type2(const Type1& theObject);           // Constructor converting Type1 to Type2
```

This can conflict with this conversion operator in the `Type2` class:

```
operator Type1();                       // Conversion from type Type1 to Type2
```

The compiler will not be able to decide which of the constructor or the conversion operator function to use when an implicit conversion is required. To remove the ambiguity, declare either or both members as `explicit`.

Overloading the Increment and Decrement Operators

The `++` and `--` operators present a new problem for the functions that implement them for a class because they behave differently depending on whether or not they prefix the operand. You need two functions for each operator: one to be called in the prefix case and the other for the postfix case. The postfix form of the operator function for either operator is distinguished from the prefix form by the presence of a parameter of type `int`. This parameter only serves to distinguish the two cases and is not otherwise used. The declarations for the functions to overload `++` for an arbitrary class, `Object`, will be

```
class Object
{
public:
    Object& operator++();           // Overloaded prefix increment operator

    const Object operator++(int);   // Overloaded postfix increment operator

    // Rest of Object class definition...
};
```

The return type for the prefix form normally needs to be a reference to the current object, `*this`, after the increment operation has been applied to it. Here's how an implementation of the prefix form for the `Box` class might look:

```
Box& Box::operator++()
{
    ++length;
    ++width;
    ++height;
    return *this;
}
```

This just increments each of the dimensions by 1 then returns the current object.

For the postfix form of the operator, you must create a copy of the original object before you modify it; then return the *copy* of the original after the increment operation has been performed on the object. Here's how that might be implemented for the `Box` class:

```
const Box Box::operator++(int)
{
    Box box {*this};           // Create a copy of the current object
    ++length;                  // Increment the current object...
    ++width;
    ++height;
    return box;                // Return the unincremented copy
}
```

The return value for the postfix operator is `const` to prevent expressions such as `theObject++++` from compiling. Such expressions are inelegant, confusing, and inconsistent with the normal behavior of the operator. However, if you don't declare the return type as `const`, such usage is possible.

■ **Note** For any class implementation that overloads the increment and decrement operators, the return type for the prefix form will *always* be a reference to the current object, and the return type for the postfix form will *always* be a copy of the original object before it has been incremented.

Function Objects

A *function object* is an object of a class that overloads the function call operator, which is `()`. A function object is also called a *functor*. The operator function in a class looks like a misprint—it is `operator()()`. A function object can be passed as an argument to a function so it provides yet another way to pass functions around. The Standard Template Library uses function objects quite extensively, particularly in the functional header. I'll show you how function objects work with an example.

Suppose I define a `Volume` class like this:

```
class Volume
{
public:
    double operator()(double x, double y, double z) {return x*y*z;}
};
```

I can use a `Volume` object to calculate a volume:

```
Volume volume;           // Create a functor
double room { volume(16, 12, 8.5) }; // Room volume in cubic feet
```

The value in the initializer list for `room` is the result of calling `operator()()` for the `volume` object so the expression is equivalent to `volume.operator()(16, 12, 8.5)`. Of course you can define more than one version of the `operator()()` function in a class:

```
class Volume
{
public:
    double operator()(double x, double y, double z) {return x*y*z;}

    double operator()(const Box& box)
    { return getLength()*getWidth()*getHeight(); }
};
```

Now a `Volume` object can return the volume of a `Box` object:

```
Box box{1.0, 2.0, 3.0};
std::cout << "The volume of the box is" << volume(box) << std::endl;
```

To enable a `Volume` object to be passed as an argument to a function, you just specify the parameter as type `Volume&`.

Summary

In this chapter, you learned how to add functions to make objects of your own data types work with the basic operators. What you need to implement in a particular class is up to you. You need to decide the nature and scope of the facilities each class should provide. Always keep in mind that you are defining a data type—a coherent entity—and that the class needs to reflect its nature and characteristics. You should also make sure that your implementation of an overloaded operator doesn't conflict with what the operator does in its standard form.

The important points from in this chapter include:

- You can overload any operator within a class to provide class-specific behavior—except for the scope resolution operator (`::`), the conditional operator (`?:`), the member access operator (`.`), the dereference pointer to class member operator (`.*`), and the `sizeof` operator.
- Operator functions can be defined as members of a class or as global operator functions.
- For a unary operator defined as a class function member, the operand is the class object.
- For a unary operator defined as a global operator function, the operand is the function parameter.
- For a binary operator function declared as a member of a class, the left operand is the class object and the right operand is the function parameter.
- For a binary operator defined by a global operator function, the first parameter specifies the left operand, and the second parameter specifies the right operand.
- To overload the increment or the decrement operator, you need two functions that provide the prefix and postfix form of the operator. The function to implement a postfix operator has an extra parameter of type `int` that serves only to distinguish the function from the prefix version.
- Functions that implement the overloading of the `+=` operator can be used in the implementation of the `+` function. This is true for all `op=` operators.

EXERCISES

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck after that, you can download the solutions from the Apress website (www.apress.com/source-code), but that really should be a last resort.

Exercise 12-1. Define an operator function in the `Box` class from `Ex12_08` that allows a `Box` object to be multiplied by an integer, `n`, to produce a new object that has a height that is `n` times to original object. Demonstrate that your operator function works as it should.

Exercise 12-2. Define an operator function that will allow a `Box` object to be premultiplied by an integer `n` to produce the same result as the operator in Exercise 12-1. Demonstrate that this operator works.

Exercise 12-3. Define division for `Box` objects so that `box1/box2` results in an integer that is the number of times `box2` can be contained in `box1`. All instances of `box2` in `box1` must have the same orientation (i.e. all `box2` lengths, widths and heights parallel) but `box2` can be in any orientation relative to `box1` (e.g. the `box2` width does not have to be parallel to the `box1` width).

Exercise 12-4. Define the remainder operator for `Box` objects so that `box1 % box2` results in the volume left unoccupied when the maximum number of `box2` objects are placed in `box1`.



Inheritance

In this chapter, you're going to look into a topic that lies at the heart of object-oriented programming: *inheritance*. Inheritance is the means by which you can create new classes by reusing and expanding on existing class definitions. Inheritance is also fundamental to making *polymorphism* possible, and polymorphism is a basic feature of object-oriented programming. I'll discuss polymorphism in the next chapter, so you what you'll learn there is an integral part of what inheritance is all about. There are subtleties in inheritance that I'll tease out using code that shows what is happening.

In this chapter you'll learn:

- How inheritance fits into the idea of object-oriented programming
- What base classes and derived classes are, and how they're related
- How to define a new class in terms of an existing class
- The use of the `protected` keyword as an access specification for class members
- How constructors behave in a derived class and what happens when they're called
- What happens with destructors in a class hierarchy
- The use of `using` declarations within a class definition
- What multiple inheritance is
- How to convert between types in a class hierarchy

Classes and Object-Oriented Programming

I'll begin by reviewing what you've learned so far about classes and explain how that leads to the ideas I'll introduce in this chapter. In Chapter 11, I explained the concept of a class and that a class is a type that you define to suit your own application requirements. In Chapter 12 you learned how you can overload the basic operators so that they work with objects of your class types. The first step in applying object-oriented programming to solve a problem is to identify the types of entities to which the problem relates and to determine the characteristics for each type and the operations that will be needed to solve the problem. Then you can define the classes and their operations, which will provide what you need to program the solution to the problem in terms of instances of the classes.

Any type of entity can be represented by a class — from the completely abstract such as the mathematical concept of a complex number to something as decidedly physical as a tree or a truck. A class definition characterizes a *set* of entities, which share a common set of properties. So as well as being a data type, a class can also be a definition of a set of real-world objects, or at least an approximation that is sufficient for solving a given problem.

In many real-world problems, the *types* of the entities involved are related. For example, a dog is a special kind of animal. A dog has all the properties of an animal plus a few more that characterize a dog. Consequently, classes that define the `Animal` and `Dog` types should be related in some way. A dog is a specialized kind of animal so you can say a *Dog is an Animal* so you would expect the class definitions to reflect this. A different sort relationship is illustrated by an automobile and an engine. You can't say that an *Automobile is an Engine* or vice versa. What you can say is that an *Automobile has an Engine*. In this chapter you'll see how the “*is a*” and “*has a*” relationships are expressed by classes.

Hierarchies

In previous chapters, I defined the `Box` class to represent a rectilinear box. The defining properties of a `Box` object were just the three orthogonal dimensions. You can apply this basic definition to the many different kinds of rectangular boxes that you find in the real world: cardboard cartons, wooden crates, candy boxes, cereal boxes, and so on. All these have three orthogonal dimensions, and in this way they're just like generic `Box` objects. However, each of them has other properties such as the things they're designed to hold, or the material from which they're made. You could describe them as specialized kinds of `Box` objects.

For example, a `Carton` class could have the same properties as a `Box` object — namely the three dimensions — plus the additional property of its composite material. You could then specialize even further by using the `Carton` definition to describe a `FoodCarton` class, which is a special kind of `Carton` that is designed to hold food. A `FoodCarton` object will have all the properties of a `Carton` object and an additional member to model the contents. Of course, a `Carton` object has the properties of a `Box` object so a `FoodCarton` object will have those too. The connections between classes that express these relationships are shown in Figure 13-1.

The `Carton` class is an extension of the `Box` class. You might say that the `Carton` class is *derived* from the `Box` class. In a similar way, the `FoodCarton` class has been derived from the `Carton` class. It's common to indicate this relationship diagrammatically by using an arrow pointing toward the more general class in the hierarchy. This notation is called *UML* (**U**niversal **M**odelling **L**anguage) and I've used UML in Figure 13-1.

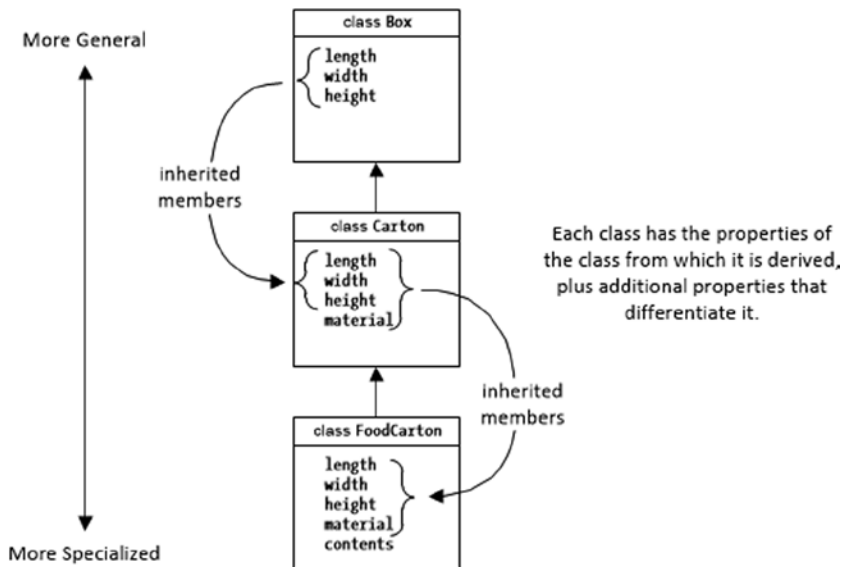


Figure 13-1. Classes in a hierarchy

In specifying one class in terms of another, you're developing a hierarchy of interrelated classes. One class is derived from another by adding extra properties — in other words, by *specialization* — making the new class a specialized version of the more general class. In Figure 13-1, each class in the hierarchy has *all* the properties of the Box class, which illustrates precisely the mechanism of class inheritance. You could define the Box, Carton, and FoodCarton classes quite independently of each other, but by defining them as related classes, you gain a tremendous amount. Let's look at how this works in practice.

Inheritance in Classes

To begin with, I'll introduce the terminology that is used for related classes. Given a class A, suppose you create a new class B that is a specialized version of A. Class A is the *base* class, and class B is the *derived* class. You can think of A as being the “parent” and B as being the “child.” A base class is sometimes referred to as a *superclass* of a class that is derived from it and the derived class is a *subclass* of its base. A derived class automatically contains all the data members of its base class, and (with some restrictions that I'll discuss) all the function members. A derived class *inherits* the data members and function members of its base class.

If class B is a derived class defined *directly* in terms of class A, then class A is a *direct base class* of B. Class B is *derived from* A. In the preceding example, the Carton class is a direct base class of FoodCarton. Because Carton is defined in terms of the Box class, the Box class is an *indirect base class* of the FoodCarton class. An object of the FoodCarton class will have inherited members from Carton, including the members that the Carton class inherits from the Box class. Figure 13-2 illustrates the way in which a derived class inherits members from a base class.

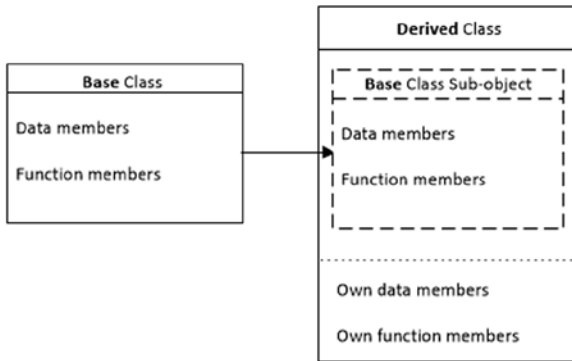


Figure 13-2. Derived class members inherited from a base class

As you can see, the derived class has a complete set of data and function members from the base class, plus its own data and function members. Thus each derived class object contains a complete base class sub-object, plus its own members.

Inheritance vs. Aggregation

Class inheritance isn't just a means of getting members of one class to appear in another. There's a very important idea that underpins the whole concept: derived class objects should be sensible specializations of base class objects. To decide whether this is the case in a specific instance you can apply the “*is a*” test: any derived class object is *a* base class object. In other words, a derived class should define a subset of the objects that are represented by the base class. I explained earlier that a Dog class might be derived from an Animal class because a dog *is an* animal; more precisely, a Dog object is a reasonable representation of a particular kind of Animal object. On the other hand, a Table class shouldn't be derived from the Dog class. Although Table and Dog objects share a common attribute in that they both usually have four legs, a Table object can't really be considered to be a Dog in any way or vice versa.

The “*is a*” test is an excellent first check, but it’s not infallible. For example, suppose you define a `Bird` class that among other things reflects the fact that most birds can fly. Now, an ostrich *is a* bird, but it’s nonsense to derive a class `Ostrich` from the `Bird` class, because ostriches can’t fly! The problem arises because of a poor definition for `Bird` objects. You really need a base class that doesn’t have the ability to fly as a property. You can then derive two subclasses, one for birds that can fly and the other for birds that can’t. If your classes pass the “*is a*” test, you should double-check by asking: Is there anything I can say about (or demand of) the base class that’s inapplicable to the derived class? If there is, then the derivation probably isn’t safe. Deriving `Dog` from `Animal` is sensible, but deriving `Ostrich` from `Bird` as I described it, isn’t.

If classes fail the “*is a*” test, then you almost certainly shouldn’t use class derivation. In this case, you could check the *has a* test. A class object passes the “*has a*” test if it contains an instance of another class. You can accommodate this by including an object of the second class as a data member of the first. The `Automobile` and `Engine` classes that I mentioned earlier are an example; an `Automobile` object would have an `Engine` object as a data member; it may well have other major subassemblies as data members of types such as `Transmission` and `Differential`. This type of relationship is called *aggregation*.

Of course, what is appropriate to include in the definition of a class depends on the application. Sometimes, class derivation is used simply to assemble a set of capabilities, so that the derived class is an envelope for packaging a given set of functions. Even then, the derived class generally represents a set of functions that are related in some way. Let’s see what the code to derive one class from another looks like.

Deriving Classes

Here’s a simplified version of the `Box` class from Chapter 12:

```
// Box.h - defines Box class
#ifndef BOX_H
#define BOX_H
#include <iostream>           // For standard streams
#include <iomanip>            // For stream manipulators

class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructors
    Box(double lv, double ww, double hv) : length {lv}, width {ww}, height {hv} {}
    Box()=default;              // No-arg constructor

    double volume() const      // Function to calculate the volume
    { return length*width*height; }

    // Accessors
    double getLength() const { return length; }
    double getWidth() const { return width; }
    double getHeight() const { return height; }

    friend std::ostream& operator<<(std::ostream& stream, const Box& box);
};
```

```
// Stream output for Box objects
inline std::ostream& operator<<(std::ostream& stream, const Box& box)
{
    stream << " Box(" << std::setw(2) << box.length << ", "
        << std::setw(2) << box.width << ", "
        << std::setw(2) << box.height << ")";
    return stream;
}
#endif
```

I can define a Carton class based on the Box class. A Carton object will be similar to a Box object but with an extra data member that indicates the material from which it's made. I'll define Carton as a derived class, using the Box class as the base class:

```
// Carton.h - defines the Carton class with the Box class as base
#ifndef CARTON_H
#define CARTON_H
#include <string> // For the string class
#include "Box.h" // For Box class definition
using std::string;

class Carton : public Box
{
private:
    string material;

public:
    Carton(const string desc = "Cardboard") : material{desc} {} // Constructor
};
#endif
```

The #include directive for the Box class definition is necessary because it is the base class for Carton. The first line of the Carton class definition indicates that Carton is derived from Box. The base class name follows a colon that separates it from the derived class name, Carton in this case. The public keyword is a base class access specifier that determines how the members of Box can be accessed from within the Carton class. I'll discuss this further in a moment.

In all other respects, the Carton class definition looks like any other. It contains a new member, material, which is initialized, by the constructor. The constructor defines a default value for the string describing the material of a Carton object, so that this is also the no-arg constructor for the Carton class. Carton objects contain all the data members of the base class, Box, plus the additional data member, material. Because they inherit all the characteristics of a Box object, Carton objects are also Box objects. There's a glaring inadequacy in the Carton class in that it doesn't have a constructor defined that permits the values of inherited members to be set, but I'll return to that later. Let's see how these class definitions work in an example:

Here's the code for your first example using a derived class:

```
// Ex13_01.cpp
// Defining and using a derived class
#include <iostream>
#include "Box.h" // For the Box class
#include "Carton.h" // For the Carton class
```

```

int main()
{
    // Create a Box object and two Carton objects
    Box box {40.0, 30.0, 20.0};
    Carton carton;
    Carton candyCarton {"Thin cardboard"};
    // Check them out - sizes first of all
    std::cout << "box occupies " << sizeof box << " bytes" << std::endl;
    std::cout << "carton occupies " << sizeof carton << " bytes" << std::endl;
    std::cout << "candyCarton occupies " << sizeof candyCarton << " bytes" << std::endl;

    // Now volumes...
    std::cout << "box volume is " << box.volume() << std::endl;
    std::cout << "carton volume is " << carton.volume() << std::endl;
    std::cout << "candyCarton volume is " << candyCarton.volume() << std::endl;

    std::cout << "candyCarton length is " << candyCarton.getLength() << std::endl;

    // Uncomment any of the following for an error...
    // box.length = 10.0;
    // candyCarton.length = 10.0;
}

```

I get the following output:

```

box occupies 24 bytes
carton occupies 56 bytes
candyCarton occupies 56 bytes
box volume is 24000
carton volume is 1
candyCarton volume is 1
candyCarton length is 1

```

The `main()` function creates a `Box` object and two `Carton` objects and outputs the number of bytes occupied by each object. The output shows what you would expect — that a `Carton` object is larger than a `Box` object. A `Box` object has three data members of type `double`; each of these occupies 8 bytes on my machine, so that's 24 bytes in all. Both of the `Carton` objects are the same size: 56 bytes. The additional memory occupied by each `Carton` object is down to the data member `material`, so it's the size of a `string` object that contains the description of the material. The output of the volumes for the `Carton` objects shows that the `volume()` function is indeed inherited in the `Carton` class and that the dimensions have the default values of 1.0. The next statement shows that the accessor functions are inherited too, and can be called for a derived class object.

Uncommenting either of the last two statements results in an error message from the compiler. The data members that are inherited by the `Carton` class were `private` in the base class and they are still `private` in the derived class, `Carton`, so they cannot be accessed from outside the class. There's more though. Try adding this function to the `Carton` class definition as a public member:

```
double carton_volume() { return length*width*height; }
```

This won't compile. The reason is that although the data members of `Box` are inherited, they are inherited as private members of the `Box` class. The private access specifier determines that members are totally private to the class. Not only can they not be accessed from outside the `Box` class, they also cannot be accessed from inside a class that inherits them.

Access to inherited members of a derived class object is not only determined by their access specification in the base class but by *both* the access specifier in the base class and the access specifier of the base class in the derived class. I'll go into that a bit more next.

protected Members of a Class

The private members of a base class being only accessible to member functions of the base class is, to say the least, inconvenient. Most of the time you want the members of a base class to be *accessible* from within the derived class, but nonetheless *protected* from outside interference. In addition to the public and private access specifiers for class members, you can declare members as protected. Within the class the protected keyword has exactly the same effect as the private keyword. protected members cannot be accessed from outside the class except from functions that have been specified as friend functions. Things change in a derived class though. Members of a base class that are declared as protected are freely accessible in function members of a derived class, whereas the private members of the base class are not.

I can modify the `Box` class to have protected data members:

```
class Box
{
protected:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Rest of the class as before...
};
```

Now the data members of `Box` are still effectively private in that they can't be accessed by ordinary global functions, but they're now accessible within member functions of a derived class. If you now try compiling `Carton` with the `carton_volume()` member uncommented and the `Box` class members specified as protected, you'll find that it compiles without a problem.

The Access Level of Inherited Class Members

In the `Carton` class definition, I specified the `Box` base class as public. In general there are three possibilities for the base class access specifier: public, protected, or private. If you omit the base class access specifier, the default is private. So if you omit the specifier altogether — for example, by writing `class Carton: Box` at the top of the `Carton` class definition in `Ex13_01` — then the private access specifier for `Box` is assumed. You also know that the access specifiers for class members come in three flavors. Again, the choice is the same: public, protected, or private. The base class access specifier affects the access status of the inherited members in a derived class. There are nine possible combinations. I'll cover all possible combinations in the following paragraphs, although the usefulness of some of these will only become apparent in the next chapter when you learn about polymorphism.

First let's consider how private members of a base class are inherited in a derived class. Regardless of the base class access specifier (public, protected, or private), a private base class member *always* remains private to the base class. As you have seen, inherited private members are private members of the derived class, so they're inaccessible outside the derived class. They're also inaccessible to member functions of the derived class because they're private to the base class.

Now, let's look into how public and protected base class members are inherited. In all the remaining cases, inherited members can be accessed by member functions of the derived class. The inheritance of public and protected base class members works like this:

1. When the base class specifier is `public`, the access status of the inherited members is unchanged. Thus, inherited public members are public, and inherited protected members are protected in a derived class.
2. When the base class specifier is `protected`, both public and protected members of a base class are inherited as protected members.
3. When the base class specifier is `private`, inherited public and protected members become private to the derived class, so they're accessible by member functions of the derived class but cannot be accessed if they're inherited in another derived class.

This is summarized in Figure 13-3. Being able to change the access level of inherited members in a derived class gives you a degree of flexibility, but remember that you can only make the access level more stringent; you can't relax the access level that is specified in the base class.

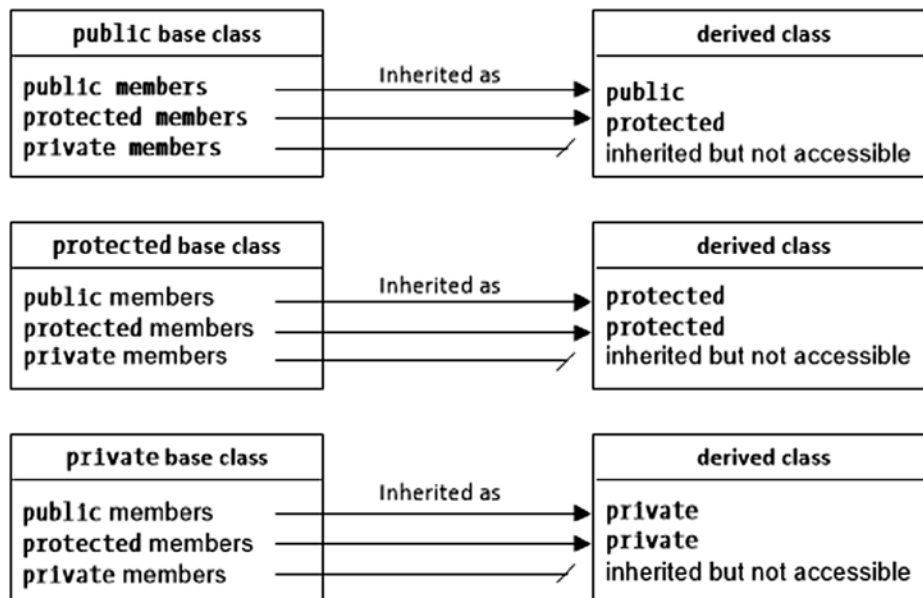


Figure 13-3. The effect of the base class specifier on the accessibility of inherited members

Choosing Access Specifiers in Class Hierarchies

You have two aspects to consider when defining a hierarchy of classes: the access specifiers for the members of each class, and the base class access specifier in each derived class. The public members of a class define the external interface to the class and this shouldn't normally include data members. Class members that aren't part of the class interface should not be directly accessible from outside the class, which means that they should be `private` or `protected`. Which access specification you choose for a particular member depends on whether or not you want to allow access in a derived class. If you do, use `protected`; otherwise, use `private`.

Figure 13-4 shows how the accessibility of inherited members is only affected by the access specifiers of the members in the base class. Within a derived class, public and protected base class members are always accessible, and private base class members are never accessible. From outside the derived class, only public base class members may be accessed — and this is only the case when the base class is declared as public.

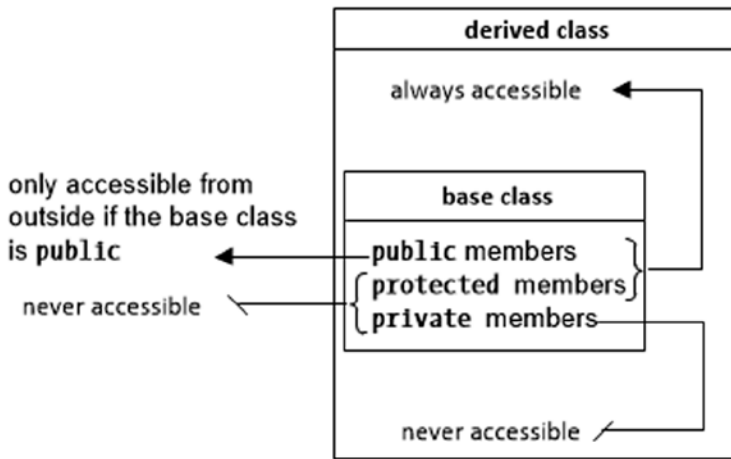


Figure 13-4. The effect of access specifiers on base class members

If the base class access specifier is `public`, then the access status of inherited members remains unchanged. By using the `protected` and `private` base class access specifiers, you are able to do two things:

1. You can prevent access to public base class members from outside the derived class — either specifier will do this. If the base class has public function members, then this is a serious step because the class interface for the base class is being removed from public view in the derived class.
2. You can affect how the inherited members of the derived class are inherited in another class that uses the derived class as its base.

Figure 13-5 shows how the public and protected members of a base class can be passed on as protected members of another derived class. Members of a privately inherited base class won't be accessible in any further derived class. In the majority of instances, the public base class access specifier is most appropriate with the base class data members declared as either `private` or `protected`. In this case the internals of the base class sub-object is internal to the derived class object and is therefore not part of the public interface for the derived class object. In practice, because the derived class object *is a* base class object, you'll want the base class interface to be inherited in the derived class, and this implies that the base class must be specified as `public`.

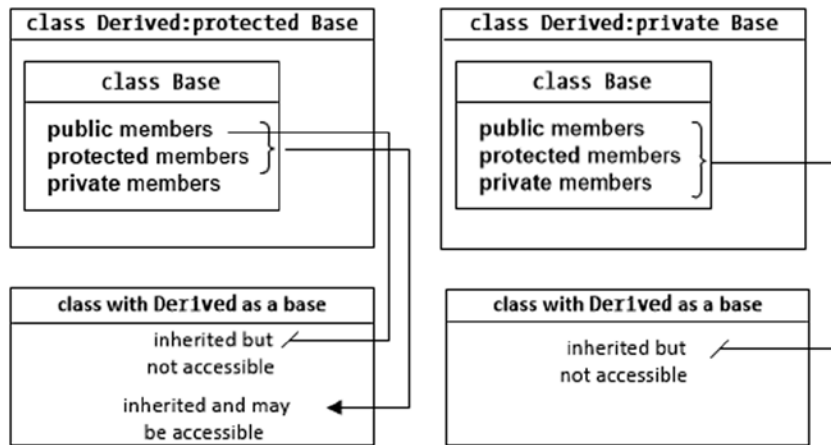


Figure 13-5. *Affecting the access specification of inherited members*

Constructors are not normally inherited for very good reasons but you'll see later in this chapter how you can cause constructors to be inherited in a derived class.

Changing the Access Specification of Inherited Members

You might want to exempt a particular base class member from the effects of a protected or private base class access specification. This is easy to understand with an example. Suppose you derive the `Carton` class from the `Box` class in `Ex13_01` but with `Box` as a private base class. All members inherited from `Box` will now be private in `Carton` but you'd like the `volume()` function to remain public in the derived class, as it is in the base class. You can restore the public status for a particular inherited member that was public in the base class with a `using` declaration.

This is essentially the same as the `using` declaration for namespaces. You can force the `volume()` function to be public in the derived class by defining the `Carton` class like this:

```
class Carton : private Box
{
private:
    string material;

public:
    using Box::volume;           // Inherit as public
    Carton(const string desc = "Cardboard") : material {desc} {} // Constructor
};
```

The class definition defines a scope, and the `using` declaration within the class definition introduces a name into that class scope. The member access specification applies to the `using` declaration so the `volume` name is introduced into the public section of the `Carton` class so it overrides the private base class access specification for the `volume()` member of the base class. The function will be inherited as public in the `Carton` class, not as private. `Ex13_01A` in the code download shows this working.

There are several points to note here. First, when you apply a `using` declaration to the name of a member of a base class, you must qualify the name with the base class name, because this specifies the context for the member name. Second, note that you don't supply a parameter list or a return type for a function member — just the qualified name. Third, the `using` declaration works with inherited data members in a derived class.

You can use a `using` declaration to override an original `public` or `protected` base class access specifier in a base class. Hence, you can allow a base class member more or less accessibility in the derived class in this way. For example, if the `volume()` function was `protected` in the `Box` base class, you could make it `public` in the derived `Carton` class with the same `using` declaration in a `public` section of `Carton`. However, you can't apply a `using` declaration to relax the specification of a `private` member of a base class because `private` members cannot be accessed in a derived class.

Constructor Operation in a Derived Class

If you put output statements in the constructors for the `Carton` class and the `Box` class and rerun the example, you'll see what happens when a `Carton` object is created. You'll need to define the default `Box` and `Carton` class constructors to include the output statements. Creating each `Carton` object always results in the default no-arg `Box` constructor being called first, followed by the `Carton` class constructor.

Derived class objects are always created in the same way, even when there are several levels of derivation. The most base class constructor is called first, followed by the constructor for the class derived from that, followed by the constructor for the class derived from that, and so on until the constructor for the most derived class is called. This makes sense if you think about it. A derived class object has a complete base class object inside it, and this needs to be created before the rest of the derived class object. If that base class is derived from another class, the same applies.

Although in `Ex13_01` the default base class constructor was called automatically, this doesn't have to be the case. You can call a particular base class constructor in the initialization list for the derived class constructor. This will enable you to initialize the base class data members with a constructor other than the default. It will also allow you to choose a particular base class constructor, depending on the data supplied to the derived class constructor. Let's see it working in another example:

Here's a new version of the `Box` class:

```
class Box
{
protected:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructors
    Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv}
    { std::cout << "Box(double, double, double) called.\n"; }

    Box(double side) : Box {side, side, side} { std::cout << "Box(double) called.\n"; }

    Box() { std::cout << "Box() called.\n"; }    // No-arg constructor

    double volume() const                      // Function to calculate the volume
    {
        return length*width*height;
    }
}
```

```
// Accessors
double getLength() const { return length; }
double getWidth() const { return width; }
double getHeight() const { return height; }

friend std::ostream& operator<<(std::ostream& stream, const Box& box);
};
```

There are now three Box constructors and they all output a message when they are called. `operator<<()` is defined as in Ex13_01.

The Carton class looks like this:

```
class Carton : public Box
{
private:
    string material {"Cardboard"};

public:
    Carton(double lv, double wv, double hv, const string desc) : Box {lv, wv, hv}, material {desc}
    { std::cout << "Carton(double,double,double,string) called.\n"; }

    Carton(const string desc) : material {desc}
    { std::cout << "Carton(string) called.\n"; }

    Carton(double side, const string desc) : Box {side}, material {desc}
    { std::cout << "Carton(double,string) called.\n"; }

    Carton() { std::cout << "Carton() called.\n"; }
};
```

This also has three constructors, including a no-arg constructor. You must define this here because if you define any constructor, the compiler will not supply a default no-arg constructor.

Here's the code to exercise this class:

```
// Ex13_02.cpp
// Calling base class constructors in a derived class constructor
#include <iostream>
#include "Box.h"           // For the Box class
#include "Carton.h"        // For the Carton class

int main()
{
    // Create four Carton objects
    Carton carton1;
    Carton carton2 {"Thin cardboard"};
    Carton carton3 {4.0, 5.0, 6.0, "Plastic"};
    Carton carton4 {2.0, "paper"};
```

```

std::cout << "carton1 volume is " << carton1.volume() << std::endl;
std::cout << "carton2 volume is " << carton2.volume() << std::endl;
std::cout << "carton3 volume is " << carton3.volume() << std::endl;
std::cout << "carton4 volume is " << carton4.volume() << std::endl;
}

```

The output is:

```

Box(double, double, double) called.
Box() called.
Carton() called.
Box() called.
Carton(string) called.
Box(double, double, double) called.
Carton(double,double,double,string) called.
Box(double, double, double) called.
Box(double) called.
Carton(double,string) called.
carton1 volume is 1
carton2 volume is 1
carton3 volume is 120
carton4 volume is 8

```

The output shows which constructors are called for each of the four Carton objects that are created in `main()`. Creating the first Carton object, `carton1`, results in the no-arg constructor for the Box class being called first, followed by the no-arg constructor for the Carton class. Creating `carton2` calls the no-arg Box constructor followed by the Carton constructor with a string parameter. Creating `carton3` calls the no-arg Box constructor followed by the Carton constructor. Creating the `carton3` object calls the Box constructor with three parameters followed by the Carton constructor with four parameters. Creating `carton4` causes two Box constructors to be called because the Box constructor with a single parameter of type `double` that is called by the Carton constructor calls the Box constructor with three parameters in its initialization list. This is all consistent with constructors being called in sequence from the most base to the most derived.

■ **Note** The notation for calling the base class constructor is exactly the same as that used for initializing data members in a constructor. This is perfectly consistent with what you're doing here, because essentially you're initializing the Box sub-object of the Carton object using the arguments passed to the Carton constructor.

Although inherited data members that are not private to the base class can be *accessed* from a derived class, they can't be *initialized* in the initialization list for a derived class constructor. For example, try replacing the first Carton class constructor in Ex13_02 with the following:

```

// Constructor that won't compile!
Carton::Carton(double lv, double ww, double hv, const string desc):
    length {lv}, width {ww}, height {hv}, material{desc}
{ std::cout << "Carton(double,double,double,string) called.\n"; }

```

You might expect this to work, because `length`, `width`, and `height` are protected base class members that are inherited publicly, so the `Carton` class constructor should be able to access them. However, the compiler complains that `length`, `width`, and `height` are *not* members of the `Carton` class. This will be the case even if you make the data members of the `Box` class public. So what's really happening here?

The answer is that a derived class constructor *can* refer to protected base class members in the body of the function, but not in the initialization list because at that stage they don't exist. The initialization list is processed before the base class constructor is called and before the base part of the object has been created. If you want to initialize the inherited data members explicitly, you must do it in the *body* of the derived class constructor. The following constructor definition would work:

```
// Constructor that will compile!
Carton::Carton(double lv, double ww, double hv, const string desc) : material{desc}
{
    length = lv;
    width = ww;
    height = hv;
    std::cout << "Carton(double,double,double,string) called.\n";
}
```

By the time the body of the `Carton` constructor begins executing, the base part of the object has been created. In this case, the base part of the `Carton` object is created by an implicit call of the no-arg `Box` class constructor. You can subsequently refer to the names of the non-private base class members without a problem.

The Copy Constructor in a Derived Class

You already know that the copy constructor is called when an object is created and initialized with another object of the same class type. The compiler will supply a default copy constructor that creates the new object by copying the original object member by member if you haven't defined your own version. Now let's examine the copy constructor in a derived class. To do this, I'll add to the class definitions in `Ex13_02`. First, I'll add a copy constructor to the base class, `Box`, by inserting the following code in the public section of the class definition:

```
// Copy constructor
Box(const Box& box) : length{box.length}, width{box.width}, height{box.height}
{ std::cout << "Box copy constructor" << std::endl; }
```

■ **Note** You saw in Chapter 12 that the parameter for the copy constructor *must* be a reference.

This initializes the data members by copying the original values and generates some output to track when the copy constructor is called.

Here's a first attempt at a copy constructor for the `Carton` class:

```
// Copy constructor
Carton(const Carton& carton) : material {carton.material}
{ std::cout << "Carton copy constructor" << std::endl; }
```

Let's see if this works (it won't!):

```
// Ex13_03
// Using a derived class copy constructor
#include <iostream>
#include "Box.h"                // For the Box class
#include "Carton.h"             // For the Carton class

int main() {
    // Declare and initialize a Carton object
    Carton carton(20.0, 30.0, 40.0, "Glassine board");

    Carton cartonCopy(carton);    // Use copy constructor

    std::cout << "Volume of carton is " << carton.volume() << std::endl
        << "Volume of cartonCopy is " << cartonCopy.volume() << std::endl;
}
```

This produces the following output:

```
Box() called.
Carton(double,double,double,string) called.
Box() called.
Carton copy constructor
Volume of carton is 1
Volume of cartonCopy is 1
```

All is not as it should be. Clearly the volume of `cartonCopy` isn't the same as `carton`, but the output also shows the reason for this. To copy the `carton` object you call the copy constructor for the `Carton` class. The `Carton` copy constructor should make a copy of the `Box` sub-object of `carton`, and to do this it should call the `Box` copy constructor. However, the output clearly shows that the *default* `Box` constructor is being called instead.

The `Carton` copy constructor won't call the `Box` copy constructor if you don't tell it to. The compiler knows that it has to create a `Box` sub-object for the object `carton` but if you don't specify how, the compiler won't second-guess your intentions - it will just create a default base object.

■ **Caution** When you define a constructor for a derived class, you are responsible for ensuring that the members of the derived class object are properly initialized. This includes all the directly inherited data members, as well as the data members that are specific to the derived class.

The obvious fix for this is to call the `Box` copy constructor in the initialization list of the `Carton` copy constructor. Simply change the copy constructor definition to this:

```
Carton(const Carton& carton) : Box {carton}, material {carton.material}
{ std::cout << "Carton copy constructor" << std::endl; }
```

The Box copy constructor is called with the carton object as an argument. The carton object is of type Carton, but it is also a perfectly good Box object. The parameter for the Box class copy constructor is a reference to a Box object so the compiler will pass carton as type Box&, which will result in only the base part of carton being passed to the Box copy constructor. This effect is called *object slicing*, and is something to beware of in general, because it can occur when you don't want a derived class object to have its derived member sliced off. If you compile and run the example again, the output will be:

```
Box(double, double, double) called.
Carton(double,double,double,string) called.
Box copy constructor
Carton copy constructor
Volume of carton is 24000
Volume of cartonCopy is 24000
```

The output shows that the constructors are called in the correct order. In particular, the Box copy constructor is called to create the Box sub-object of carton before the Carton copy constructor. By way of a check, you can see that the volumes of the candyCarton and copyCarton objects are now identical.

The Default Constructor in a Derived Class

You know that the compiler will not supply a default no-arg constructor if you define one or more constructors for a class. You also know that you can tell the compiler to insert a default constructor in any event using the default keyword. You could replace the definition of the no-arg constructor in the Carton class definition in Ex13_02 with this statement:

```
Carton()=default;
```

Now the compiler will supply a definition, even though you have defined other constructors. The definition that the compiler supplies for a derived class calls the base class constructor, so it looks like this:

```
Carton() : Box() {};
```

This implies that if the compiler supplies the no-arg constructor in a derived class, the no-arg constructor *must* be defined in the base class. If it isn't, the code will not compile. You can easily demonstrate this by removing the no-arg constructor from the Box class in Ex13_02. With the compiler-supplied default constructor specified for the Carton class, the code will no longer compile. It's easy to forget to define the default constructor in a base class when the code does not call it explicitly. Remember though, *every* derived class constructor calls a base class constructor. If a derived class constructor does not explicitly call a base constructor in its initialization list, the no-arg constructor will be called, so most of the time you need the no-arg constructor to be defined in a base class, either explicitly, or by making the compiler supply it.

Inheriting Constructors

Base class constructors are not normally inherited in a derived class. This is because a derived class typically has additional data members that need to be initialized and a base class constructor would have no knowledge of these. However, you can cause constructors to be inherited from a direct base class by putting a using declaration in the derived class. Here's how a version of the Carton class from Ex13_02 could be made to inherit the Box class constructors:

```
class Carton : public Box
{
    using Box::Box;                      // Inherit Box class constructors

private:
    string material {"Cardboard"};

public:
    Carton(double lv, double wv, double hv, const string desc) : Box {lv, wv, hv}, material {desc}
    { std::cout << "Carton(double,double,double,string) called.\n"; }
};
```

If the Box class definition is the same as in Ex13_02, the Carton class will inherit three constructors: Box(double, double, double), Box(double), and the no-arg constructor Box(). The constructors in the derived class will look like this:

```
Carton(double lv, double wv, double hv) : Box {lv, wv, hv} {}
Carton(double side) : Box {side} {}
Carton() : Box {} {}
```

Each inherited constructor has the same parameter list as the base constructor and calls the base constructor in its initialization list. The body of each constructor is empty. You can add further constructors to a derived class that inherits from its direct base, as the Carton class example illustrates. You could try this out by modifying Ex13_02 to create the following objects in main():

```
Carton cart;                      // Calls inherited no-arg constructor
Carton cartcopy {cart};           // Calls inherited copy constructor
Carton carton {1.0, 2.0, 3.0};     // Calls inherited constructor
Carton candyCarton (50.0, 30.0, 20.0, "Thin cardboard"); // Calls Carton class constructor
```

The output statements in the Box constructors will show that they are indeed called to create the first three objects.

Inherited constructors are most useful when you are deriving a class without adding additional data members. I prefer not to use constructor inheritance for the simple reason that it makes it necessary to look at the base class definition in order to determine what constructors are available in the derived class. It's also not a huge effort to define the constructors in the derived class to call base class constructors as in the previous code fragment.

Destructors Under Inheritance

Destroying a derived class object involves both the derived class destructor *and* the base class destructor. You can demonstrate this by adding destructors with output statements in the Box and Carton class definitions. You can amend the class definitions in the correct version of Ex13_03. Add the destructor definition to the Box class:

```
// Destructor
~Box() { std::cout << "Box destructor" << std::endl; }
```

And for the Carton class:

```
// Destructor
~Carton()
{
    std::cout << "Carton destructor. Material = " << material << std::endl;
}
```

Of course, if the classes allocated heap memory and stored the address in a raw pointer, defining the class destructor would be essential to avoid memory leaks. The Carton destructor outputs the material so you can tell which Carton object is being destroyed by assigning a different material to each. Let's see how these classes behave in practice:

```
// Ex13_04.cpp
// Destructors in a class hierarchy
#include <iostream>
#include "Box.h"                // For the Box class
#include "Carton.h"             // For the Carton class

int main()
{
    Carton carton;
    Carton candyCarton {50.0, 30.0, 20.0, "Thin cardboard"};

    std::cout << "carton volume is " << carton.volume() << std::endl;
    std::cout << "candyCarton volume is " << candyCarton.volume() << std::endl ;
}
```

Here's the output:

```
Box() called.
Carton() called.
Box(double, double, double) called.
Carton(double,double,double,string) called.
carton volume is 1
candyCarton volume is 30000
Carton destructor. Material = Thin cardboard
Box destructor
Carton destructor. Material = Cardboard
Box destructor
```

The point of this exercise is to see how the destructors behave. The output from the destructor calls indicates two aspects of how objects are destroyed. First, you can see the order in which destructors are called for a particular object, and second, you can see the order in which the objects are destroyed. The destructor calls recorded by the output correspond to the following actions:

Destructor Output	Object Destroyed
Carton destructor Material = Thin cardboard	candyCarton object
Box destructor	Box subobject of candyCarton
Carton destructor Material = Cardboard	carton object
Box destructor	Box subobject of carton

This shows that the objects that make up a derived class object are destroyed in the *reverse* order from which they were created. The carton object was created first and destroyed last; the candyCarton object was created last and destroyed first. This order is chosen to ensure that you never end up with an object in an illegal state. An object can only be used after it has been defined — this means that any given object can only contain pointers (or references) that point (or refer) to objects that have already been created. By destroying a given object *before* any objects that it might point (or refer) to, you ensure that the execution of a destructor can't result in any invalid pointers or references.

The Order in Which Destructors Are Called

The order of destructor calls for a derived class object is the reverse of the constructor call sequence for the object. The derived class destructor is called first, and then the base class destructor, just as in the example. The case of a three-level class hierarchy is illustrated in Figure 13-6.

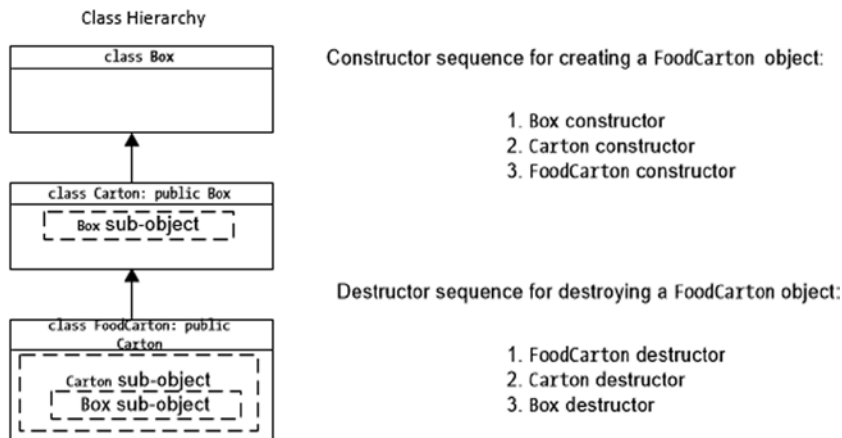


Figure 13-6. The order of destructor calls for derived class objects

For an object with several levels of derivation class, this order of destructor calls runs through the hierarchy of classes, starting with the most derived class destructor and ending with the destructor for the most base class.

Duplicate Data Member Names

It's possible that a base class and a derived class each have a data member with the same name. If you're really unlucky, you might even have names duplicated in the base class and in an indirect base. Of course, this is confusing, and you should never deliberately set out to create such an arrangement in your own classes. However, circumstances may dictate that this is how things turn out. For example, if you're deriving your class from a base class designed by another programmer, you would almost certainly know nothing about the `private` data members of his class; you would only know about the base class interface. What happens if data members in the base and derived classes have the same names?

Duplication of names is no bar to inheritance and you can differentiate between identically named base and derived class members. Suppose you have a class `Base`, defined as follows:

```
class Base
{
public:
    Base(int number = 10) : value {number} {}           // Constructor

protected:
    int value;
};
```

This just contains a single data member, `value`, and a constructor. You can derive a class `Derived` from `Base` as follows:

```
class Derived: public Base
{
public:
    Derived(int number = 20) : value {number} {}        // Constructor
    int total() const;                                  // Total value of data members

protected:
    int value;
};
```

The derived class has a data member called `value`, and it will also inherit the `value` member of the base class. You can see that it's already starting to look confusing! I'll show how you can distinguish the two members with the name `value` in the derived class by writing a definition for the `total()` function. Within the derived class function member, `value` by itself refers to the member declared within that scope; that is, the derived class member. The base class member is declared within a different scope, and to access it from a derived class function member, you must qualify the member name with the base class name. Thus, you can write the `total()` function as:

```
int Derived::total() const
{
    return value + Base::value;
}
```

The expression `Base::value` refers to the base class member, and `value` by itself refers to the member declared in the `Derived` class.

Duplicate Function Member Names

What happens when base class and derived class function members share the same name? There are two situations that can arise in relation to this. The first is when the functions have the same name but different parameter lists. Although the function signatures are different, this is *not* a case of function overloading. This is because overloaded functions must be defined within the same scope, and each class, base or derived, defines a separate scope. In fact, scope is the key to the situation. A derived class function member will hide an inherited function member with the same name. Thus, when base and derived function members have the same name, you must introduce the qualified name of the base class member function into the scope of the derived class with a `using` declaration if you want to access it. Either function can then be called for a derived class object, as illustrated in Figure 13-7.

```
class Base
{
public:
    void doThat(int arg);
    ...
};

class Derived: public Base
{
public:
    void doThat(double arg);
    using Base::doThat;
    ...
};
```

*By default the derived class function **doThat()** would hide the inherited function with the same name. The **using** declaration introduces the base class function name, **doThat**, into the derived class's scope, so both versions of the function are available within the derived class. The compiler can distinguish them in the derived class because they have different signatures.*

```
Derived object;
object.doThat(2);    // Call inherited base function
object.doThat(2.5); // Call derived function
```

Figure 13-7. Inheriting a function with the same name as a function member

The second possibility is that both functions have the same function signature. You can still differentiate the inherited function from the derived class function by using the class name as a qualifier for the base class function:

```
Derived object;           // Object declaration
object.Base::doThat(3);   // Call base version of the function
```

However, there's a lot more to it than I can discuss at this point. This subject is closely related to polymorphism, which is explored in much more depth in the next chapter.

Multiple Inheritance

So far, your derived classes have all been derived from a *single* direct base class. However, you're not limited to this structure. A derived class can have as many direct base classes as an application requires. This is referred to as *multiple inheritance* as opposed to *single inheritance*, in which a single base class is used. This opens vast new dimensions of potential complexity in inheritance which is perhaps why multiple inheritance is used much less frequently than single inheritance. Because of the complexity, multiple inheritance is best avoided as much as possible. I'll just explain the basic ideas behind how multiple inheritance works.

Multiple Base Classes

Multiple inheritance involves two or more base classes being used to derive a new class, so things are immediately more complicated. The idea of a derived class being a specialization of its base leads in this case to the notion that the derived class defines an object that is a specialization of two or more different and independent class types concurrently. In practice, multiple inheritance is rarely used in this way. More often, multiple base classes are used to add the features of the base classes together to form a composite object containing the capabilities of its base classes, sometimes referred to as “mix-in” programming. This is usually for convenience in an implementation rather than to reflect any particular relationships between objects. For example, you might consider a programming interface of some kind — for graphics programming, perhaps. A comprehensive interface could be packaged in a set of classes, each of which defines a self-contained interface that provides some specific capability, such as drawing two-dimensional shapes. You can then use several of these classes as bases for a new class that provides precisely the set of capabilities you need for an application.

To explore some of the implications of multiple inheritance, I’ll start with a hierarchy that includes the `Box` and `Carton` classes. Suppose you need a class that represents a package containing dry contents, such as a carton of cereal. It’s possible to do this by using single inheritance, deriving a new class from the `Carton` class and adding a data member to represent contents, but you could also do it using the hierarchy illustrated in Figure 13-8.

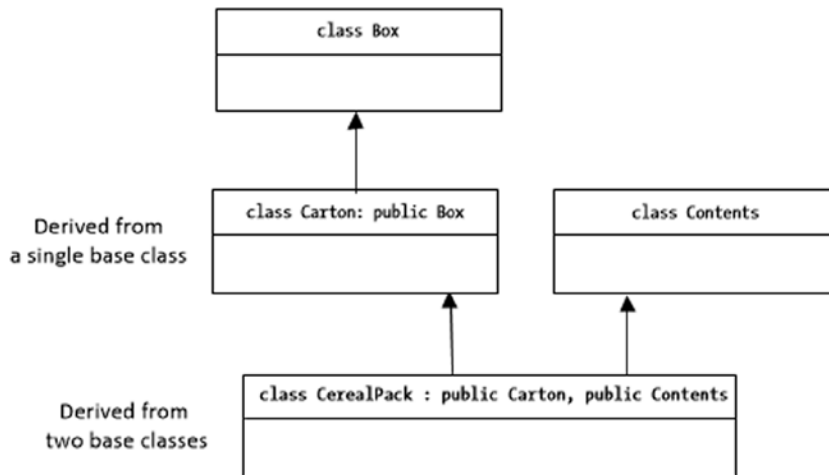


Figure 13-8. An example of multiple inheritance

The definition of the `CerealPack` class would look like this:

```
class CerealPack : public Carton, public Contents
{
    // Details of the class...
};
```

Each base class is specified after the colon in the class header, and the base classes are separated by commas. Each base class has its own access specifier and if you omit the access specifier, `private` is assumed, the same as with single inheritance. The `CerealPack` class will inherit *all* the members of *both* base classes, so this will include the members of the indirect base, `Box`. As in the case of single inheritance, the access level of each inherited member is determined by two factors: the access specifier of the member in the base class and the base class access specifier. A `CerealPack` object contains two sub-objects, a `Contents` sub-object and a `Carton` sub-object that has a further sub-object of type `Box`.

Inherited Member Ambiguity

Multiple inheritance can create problems. I'll put together an example that will show the sort of complications you can run into. The Box class is the same as in Ex13_04 but I'll extend the Carton class from that example a little:

```
class Carton : public Box
{
protected:
    string material {"Cardboard"};
    double thickness {0.125};           // Material thickness inches
    double density {0.2};               // Material density in pounds/cubic inch

public:
    // Constructors
    Carton(double lv, double ww, double hv, const string desc) : Box {lv, ww, hv}, material {desc}
    {
        std::cout << "Carton(double,double,double,string) called.\n";
    }

    Carton(const string desc) : material {desc}
    { std::cout << "Carton(string) called.\n"; }
    Carton(double side, const string desc) : Box {side}, material {desc}
    {
        std::cout << "Carton(double,string) called.\n";
    }

    Carton()
    {
        std::cout << "Carton() called.\n";
    }

    Carton(double lv, double ww, double hv, string desc, double dense, double thick) :
                                                Carton {lv, ww, hv, desc}
    {
        density = dense;
        thickness = thick;
        std::cout << "Carton(double,double,double,string, double,double) called.\n";
    }

    // Copy constructor
    Carton(const Carton& carton) : Box {carton}, material {carton.material}
    {
        std::cout << "Carton copy constructor" << std::endl;
    }

    // Destructor
    ~Carton()
    {
        std::cout << "Carton destructor. Material = " << material << std::endl;
    }
}
```

```
// "Get carton weight" function
double getWeight() const
{
    return 2.0*(length*width + width*height + height*length)*thickness*density;
}
};
```

I've added two data members that record the thickness and density of the material from which the Carton object is made, a new constructor that allows all data members to be set, and a new function member, `getWeight()`, which calculates the weight of an empty Carton object. The new constructor calls another Carton class constructor in its initialization list so it is a delegating constructor, as you saw in Chapter 11. A delegating constructor cannot have further initializers in the list so the values for density and thickness have to be set in the constructor body.

The Contents class will describe an amount of a dry product, such as breakfast cereal, which can be contained in a carton. The class will have three data members: name, volume, and density (in pounds per cubic inch). In practice, you would probably include a set of possible cereal types, complete with their densities, so that you could validate the data in the constructor, but I'll ignore such niceties in the interest of keeping things simple. Here's the class definition along with the preprocessing directives that you need in the header file, `Contents.h`:

```
// Contents.h - Dry contents
#ifndef CONTENTS_H
#define CONTENTS_H
#include <iostream>

class Contents
{
protected:
    string name {"cereal"};           // Contents type
    double volume {};                 // Cubic inches
    double unitWeight {0.03};         // Pounds per cubic inch

public:
    Contents(const string name, double wt, double vol) :
        name {name}, unitWeight {wt}, volume {vol}
    { std::cout << "Contents(string,double,double) called.\n"; }

    Contents(const string name) : name {name} { std::cout << "Contents(string) called.\n"; }

    Contents() { std::cout << "Contents() called.\n"; }

    // Destructor
    ~Contents()
    {
        std::cout << "Contents destructor" << std::endl;
    }

    // "Get contents weight" function
    double getWeight() const
    {
        return volume*unitWeight;
    }
};
#endif
```

In addition to the constructors and the destructor, the class has a public function member, `getWeight()`, to calculate the weight of the contents. Note how the name member is initialized in the constructor initializer list with the parameter value that has the same name. This is just to illustrate that this is possible - not a recommended approach. I'll define the `CerealPack` class with the `Carton` and `Contents` classes as public base classes:

```
// Cerealpack.h - Class defining a carton of cereal
#ifndef CEREALPACK_H
#define CEREALPACK_H
#include <iostream>
#include "Carton.h"
#include "Contents.h"

class CerealPack : public Carton, public Contents
{
public:
    CerealPack(double length, double width, double height, const string cerealType) :
        Carton {length, width, height, "cardboard"}, Contents {cerealType}
    {
        std::cout << "CerealPack constructor" << std::endl;
        Contents::volume = 0.9*Carton::volume();           // Set contents volume
    }

    // Destructor
    ~CerealPack()
    {
        std::cout << "CerealPack destructor" << std::endl;
    }
};
#endif
```

This class inherits from both the `Carton` and `Contents` classes. The constructor requires only the external dimensions and the cereal type. The material for the `Carton` object is set in the `Carton` constructor call, in the initialization list. A `CerealPack` object will contain two sub-objects corresponding to the two base classes. Each sub-object is initialized through constructor calls in the initialization list for the `CerealPack` constructor. Note that the volume data member of the `Contents` class is zero by default so, in the body of the `CerealPack` constructor, the value is calculated from the size of the carton. The reference to the volume data member inherited from the `Contents` class must be qualified here because it's the same as the name of the function inherited from `Box` via `Carton`. You'll be able to trace the order of constructor and destructor calls from the output statements here and in the other classes.

Let's try creating a `CerealPack` object and calculate its volume and weight with the following very simple program:

```
// Ex13_05 - doesn't compile!
// Using multiple inheritance
#include <iostream>
#include "CerealPack.h"           // For the CerealPack class

int main()
{
    CerealPack cornflakes {8.0, 3.0, 10.0, "Cornflakes"};

    std::cout << "cornflakes volume is " << cornflakes.volume() << std::endl
        << "cornflakes weight is " << cornflakes.getWeight() << std::endl;
}
```


Unfortunately, there's a problem. The program won't compile. The difficulty is that I have foolishly used some non-unique function names in the base classes. The name `volume` is inherited as a function from `Box` and as a data member from `Contents` and the `getWeight()` function is inherited from `Carton` and from `Contents` in the `CerealPack` class. There's more than one ambiguity problem.

Of course, when writing classes for use in inheritance, you should avoid duplicating member names in the first instance. The ideal solution to this problem is to rewrite your classes. If you are unable to rewrite the classes — if the base classes are from a library of some sort for example — then you would be forced to qualify the function names in `main()`. You could amend the output statement in `main()` to get the code to work:

```
std::cout << "cornflakes volume is " << cornflakes.Carton::volume() << std::endl
          << "cornflakes weight is " << cornflakes.Contents::getWeight() << std::endl;
```

With this change the program will compile and run, and it will produce the following output:

```
Box(double, double, double) called.
Carton(double,double,double,string) called.
Contents(string) called.
CerealPack constructor
cornflakes volume is 240
cornflakes weight is 6.48
CerealPack destructor
Contents destructor
Carton destructor. Material = cardboard
Box destructor
```

The working version is in the code download as `Ex13_05A`. You can see from the output that this cereal will give you a solid start to the day — a single packet weighs over 6 pounds. You can also see that the constructor and destructor call sequences follow the same pattern as in the single inheritance context: the constructors run down the hierarchy from most base to most derived, and the destructors run in the opposite order. The `CerealPack` object has sub-objects from both legs of its inheritance chain, and all the constructors for these subobjects are involved in the creation of a `CerealPack` object.

Repeated Inheritance

The previous example demonstrated how ambiguities can occur when member names of base classes are duplicated. Another ambiguity can arise in multiple inheritances when a derived object contains multiple versions of a sub-object of one of the base classes. You must not use a class more than once as a direct base class but it's possible to end up with duplication of an *indirect* base class. Suppose the `Box` and `Contents` classes in `Ex13_05` were themselves derived from a class `Common`. Figure 13-9 shows the class hierarchy that is created.

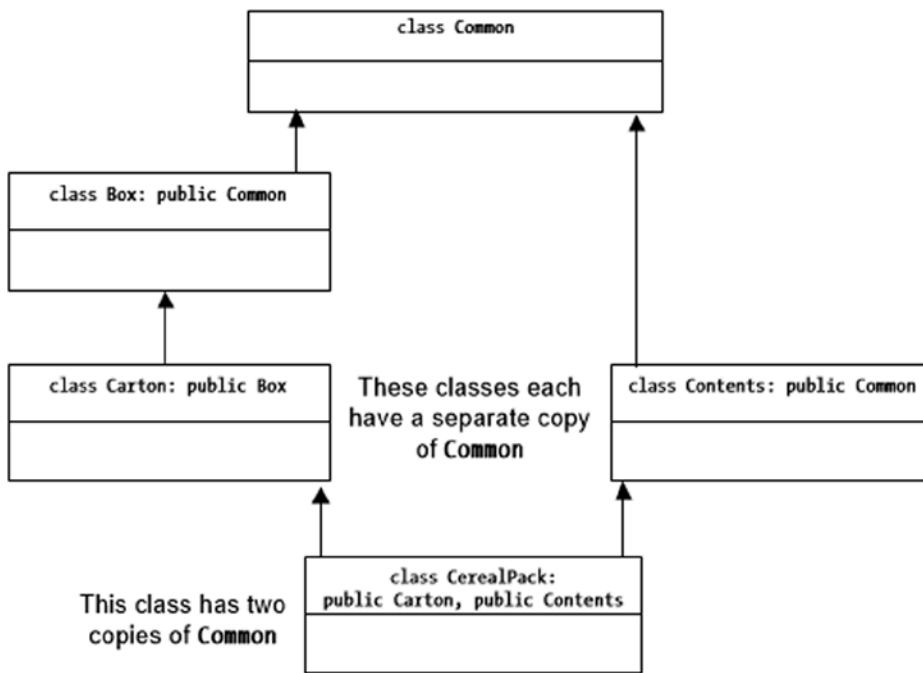


Figure 13-9. Duplicate base classes in a derived class

The CerealPack class inherits all the members of both the Contents and Carton classes. The Carton class inherits all the members of the Box class, and both the Box and Contents classes inherit the members of the Common class. Thus, as Figure 13-9 shows, the Common class is duplicated in the CerealPack class. The effect of this on objects of type CerealPack is that every CerealPack object will have two sub-objects of type Common.

It is conceivable — just — that you actually want to *allow* the duplication of the Common class. In this case, you must qualify each reference to the Common class member so that the compiler can tell which inherited member you’re referring to in any particular instance. In this case, you can do this by using the Carton and Contents class names as qualifiers because each of these classes contains a unique subobject of type Common. Of course, to call the Common class constructors when you’re creating a CerealPack object, you would also need qualifiers to specify which of the two base objects you were initializing. More typically, though, you would want to *prevent* the duplication of a base class, so let’s see how to do that.

Virtual Base Classes

To avoid duplication of a base class, you must identify to the compiler that the base class should only appear once within a derived class. You do this by specifying the class as a *virtual base class* using the `virtual` keyword. The Contents class would be defined like this:

```
class Contents: public virtual Common
{
    ...
};
```

The Box class would also be defined with a virtual base class:

```
class Box : public virtual Common
{
    ...
};
```

Now any class that uses the Contents and Box classes as direct or indirect bases will inherit the other members of the base classes as usual but will inherit only one instance of the Common class. The derived CerealPack class would inherit only a single instance of the Common base class. Because there is no duplication of the members of Common in the CerealPack class, no qualification of the member names is needed when referring to them in the derived class.

Converting Between Related Class Types

Every derived class object has a base class object inside it waiting to get out. Conversions from a derived type to its base are always legal and automatic. Here's a definition of a Carton object:

```
Carton carton {40, 50, 60, "fiberboard"};
```

You can convert this object to a base class object of type Box and store the result like this:

```
Box box;
box = carton;
```

The assignment statement converts the carton object to a new automatic object of type Box and stores a copy of it in box. Of course, only the Box sub-object part of carton is used — the Carton specific portion is sliced off and discarded. The assignment operator that is used is the default assignment operator for the Box class. Conversions up a class hierarchy (that is, toward the base class) are legal and automatic as long as there is no ambiguity. Ambiguity can arise when two base classes each have the same type of sub-object. For example, if you use the definition of the CerealPack class that contains two Common subobjects (as you saw in the previous section), and you initialize a CerealPack object, cornflakes, then the following will be ambiguous:

```
Common common {cornflakes};
```

The compiler won't be able to determine whether the conversion of cornflakes should be to the Common sub-object of Carton or to the Common sub-object of Contents.

You can't obtain automatic conversions for objects down a class hierarchy - that is, toward a more specialized class. A Box object contains no information about any class type that may be derived from Box, so the conversion doesn't have a sensible interpretation. In the next chapter you'll see that pointers are different. A pointer of a base class type can store the address of a derived class object, in which case you can cast the pointer to a derived class type.

Summary

In this chapter, you learned how to define a class based on one or more existing classes and how class inheritance determines the makeup of a derived class. Inheritance is a fundamental characteristic of object-oriented programming and it makes polymorphism possible. The important points to take from this chapter include:

- A class may be derived from one or more base classes, in which case the derived class inherits members from all of its bases.
- Single inheritance involves deriving a class from a single base class. Multiple inheritance involves deriving a class from two or more base classes; multiple inheritance is best avoided in general.
- Access to the inherited members of a derived class is controlled by two factors: the access specifier of the member in the base class and the access specifier of the base class in the derived class declaration.

- A constructor for a derived class is responsible for initializing all members of the class, including the inherited members.
- Creation of a derived class object always involves the constructors of all of the direct and indirect base classes, which are called in sequence (from the most base through to the most direct) prior to the execution of the derived class constructor.
- A derived class constructor can explicitly call constructors for its direct bases in the initialization list for the constructor.
- A member name declared in a derived class, which is the same as an inherited member name, will hide the inherited member. To access the hidden member, use the scope resolution operator to qualify the member name with its class name.
- When a derived class with two or more direct base classes contains two or more inherited subobjects of the same class, the duplication can be prevented by declaring the duplicated class as a virtual base class.

EXERCISES

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Apress website (www.apress.com/source-code), but that really should be a last resort.

Exercise 13-1. Define a base class called `Animal` that contains two private data members: a string to store the name of the animal (e.g., “Fido” or “Yogi”) and an integer member called `weight` that will contain the weight of the animal in pounds. Also include a public function member, `who()`, that outputs a message giving the name and weight of the `Animal` object. Derive two classes named `Lion` and `Aardvark`, with `Animal` as a public base class. Write a `main()` function to create `Lion` and `Aardvark` objects (“Leo” at 400 pounds and “Algernon” at 50 pounds, say) and demonstrate that the `who()` member is inherited in both derived classes by calling it for the derived class objects.

Exercise 13-2. Change the access specifier for the `who()` function in the `Animal` class to `protected`, but leave the rest of the class as before. Now modify the derived classes so that the original version of `main()` still works without alteration.

Exercise 13-3. In the solution to the previous exercise, change the access specifier for the `who()` member of the base class back to `public`, and implement the `who()` function as a member of each derived class so that the output message also identifies the name of the class. Change `main()` to call the base class and derived class versions of `who()` for each of the derived class objects.

Exercise 13-4. Define a `Person` class containing data members for `age`, `name`, and `gender`. Derive an `Employee` class from `Person` that adds a data member to store a personnel number. Derive an `Executive` class from `Employee`. Each derived class should define a function member that displays information about what it is. (Name and type will do — something like “Fred Smith is an Employee.”) Write a `main()` function to generate a vector of five executives and a vector of five ordinary employees, and display information about them. In addition, display the information on the executives by calling the member function inherited from the `Employee` class.



Polymorphism

Polymorphism is such a powerful feature of object-oriented programming that you'll use it in the majority of your C++ programs. Polymorphism requires you to use derived classes, and the content of this chapter relies heavily on the concepts related to inheritance in derived classes that I introduced in the previous chapter.

In this chapter you'll learn:

- What polymorphism is and how you get polymorphic behavior with your classes
- What a virtual function is
- When and why you need virtual destructors
- How default parameter values for virtual functions are used
- What a pure virtual function is
- What an abstract class is
- How you cast between class types in a hierarchy
- How you determine the type of an object passed to a function as the argument for a parameter that is a reference to a base class
- What pointers to members are, and how you use them

Understanding Polymorphism

Polymorphism is a capability provided by many object-oriented languages. In C++ polymorphism always involves the use of a pointer or a reference to an object to call a function member. Polymorphism only operates with classes that share a common base class. I'll show how polymorphism works by considering an example with more boxes, but first I'll explain the role of a pointer to a base class because it's fundamental to the process.

Using a Base Class Pointer

In the previous chapter, you saw how an object of a derived class type contains a sub-object of the base class type. In other words, you can regard every derived class object as a base class object. Because of this, you can always use a pointer to base class to store the address of a derived class object; in fact, you can use a pointer to any direct or indirect base class to store the address of a derived class object. Figure 14-1 shows how the `Carton` class is derived from the `Box` base class by single inheritance and the `CerealPack` class is derived by multiple inheritances from the `Carton` and `Contents` base classes. It illustrates how pointers to base classes can be used to store addresses of derived class objects.

```

CerealPack breakfast;

// You can store the address of
// the breakfast object
// in any base class pointer:

Carton* pCarton {&breakfast};
Box* pBox {&breakfast};
Contents* pContents {&breakfast};

// You can store the address of
// a Carton object in a base pointer

Carton carton;
pBox = &carton;

```

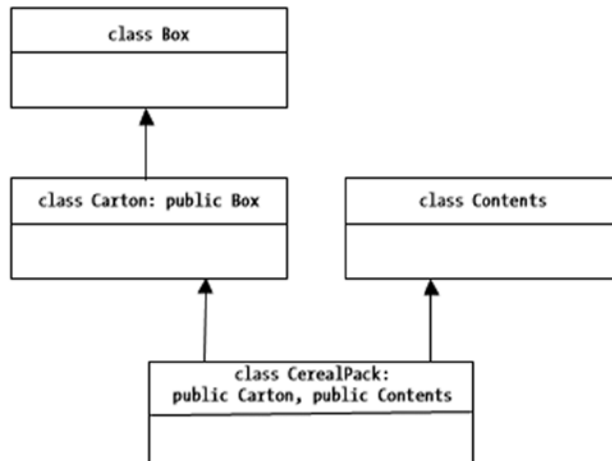


Figure 14-1. Storing the address of a derived class object in a base class pointer

The reverse is *not* true. For instance you can't use a pointer of type `Carton*` to store the address of an object of type `Box`. This is logical because a pointer type incorporates the type of object to which it can point. A derived class object is a specialization of its base - it *is a* base class object - so using a pointer to base to store its address is reasonable. However, a base class object is definitely *not* a derived class object so a pointer to a derived class type cannot point to it. A derived class always contains a complete sub-object of each of its bases, but each base class only represents a part of a derived class object.

I'll take a specific example. Suppose you derive two classes from the `Box` class to represent different kinds of containers, `Carton` and `ToughPack`. Suppose further that the volume of each of these derived types is calculated differently. For a `Carton` made of cardboard, you might just reduce the volume slightly to take the thickness of the material into account. For a `ToughPack` object you might have to reduce the usable volume by a considerable amount to allow for protective packaging. The `Carton` class definition could be of the form:

```

class Carton : public Box
{
    // Details of the class...

public:
    double volume() const;
};

```

The `ToughPack` class could have a similar definition:

```

class ToughPack : public Box
{
    // Details of the class...

public:
    double volume() const;
};

```

Given these definitions, you can declare and initialize a pointer as follows:

```
Carton carton {10.0, 10.0, 5.0};
Box* pBox {&carton};
```

The pointer `pBox`, of type pointer to `Box`, has been initialized with the address of `carton`. This is possible because `Carton` is derived from `Box`, and therefore contains a sub-object of type `Box`. You could use the same pointer to store the address of a `ToughPack` object, because the `ToughPack` class is also derived from `Box`:

```
ToughPack hardcase {12.0, 8.0, 4.0};
pBox = &hardcase;
```

The `pBox` pointer can contain the address of any object of any class that has `Box` as a base. The type of the pointer, `Box*`, is called its *static type*. Because `pBox` is a pointer to a *base* class, it also has a *dynamic type*, which varies according to the type of object to which it points. When `pBox` is pointing to a `Carton` object, its dynamic type is pointer to `Carton`. When `pBox` is pointing to a `ToughPack` object, its dynamic type is pointer to `ToughPack`. When `pBox` points to an object of type `Box`, its dynamic type is the same as its static type. The magic of polymorphism springs from this. Under conditions that I'll explain shortly, you can use `pBox` to call a function that's defined in the base class and in each derived class, and have the function that is actually called selected at runtime on the basis of the dynamic type of `pBox`. Consider these statements:

```
double vol {};  
vol = pBox->volume();           // Store volume of the object pointed to
```

If `pBox` contains the address of a `Carton` object, then this statement calls `volume()` for the `Carton` object. If it points to a `ToughPack` object, then this statement calls `volume()` for `ToughPack`. This works for any classes derived from `Box`. Thus the expression `pBox->volume()` can result in different behavior depending on what `pBox` is pointing to. Perhaps more importantly, the behavior that is appropriate to the object pointed to by `pBox` is selected automatically at runtime.

Polymorphism is a very powerful mechanism. Situations arise frequently in which the specific type of an object cannot be determined in advance — not at design time or at compile time; only at runtime. This can be handled easily using polymorphism. Polymorphism is commonly used with interactive applications, where the type of input is up to the whim of the user. For instance, a graphics application that allows different shapes to be drawn — circles, lines, curves, and so on — may define a derived class for each shape type, and these classes all have a common base class called `Shape`. A program can store the address of an object the user creates in a pointer, `pShape`, of type `Shape*` and draw the shape with a statement such as `pShape->draw()`. This will call the `draw()` function for the shape that is pointed to, so this one expression can draw any kind of shape. In order for function calls to operate in this way, the function must be a member of the base class as well as a member of the derived class. Let's take a more in-depth look at how inherited functions behave.

Calling Inherited Functions

Before I get to the specifics of polymorphism, I need to explain the behavior of inherited function members a bit further. To help with this, I'll revise the `Box` class to include a function that calculates the volume of a `Box` object, and another function that displays the resulting volume. The new version of the class definition in `Box.h` and `Box.cpp` will be:

```
// Box.h  
#ifndef BOX_H  
#define BOX_H  
#include <iostream>
```

```

class Box
{
protected:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv} {}

    // Function to show the volume of an object
    void showVolume() const
    { std::cout << "Box usable volume is " << volume() << std::endl; }

    // Function to calculate the volume of a Box object
    double volume() const { return length*width*height; }
};
#endif

```

We can display the usable volume of a Box object by calling the `showVolume()` function for the object. The data members are specified as `protected` so they can be accessed by the function members of any derived class.

I'll also define the `ToughPack` class with `Box` as a base. A `ToughPack` object incorporates packing material to protect its contents, so its capacity is only 85 percent of a basic `Box` object. Therefore, a different `volume()` function is needed in the derived class to account for this:

```

// ToughPack.h
#ifndef TOUGHPACK_H
#define TOUGHPACK_H

#include "Box.h"

class ToughPack : public Box
{
public:
    // Constructor
    ToughPack(double lv, double wv, double hv) : Box {lv, wv, hv} {}

    // Function to calculate volume of a ToughPack allowing 15% for packing
    double volume() const { return 0.85*length*width*height; }
};
#endif

```

Conceivably, you could have additional members in this derived class, but for the moment, I'll keep it simple, concentrating on how the inherited functions work. The derived class constructor just calls the base class constructor in its initializer list to set the data member values. You don't need any statements in the body of the derived class

constructor. You also have a new version of the `volume()` function to replace the version from the base class. The idea here is that you can get the inherited function `showVolume()` to call the derived class version of `volume()` when you call it for an object of the `ToughPack` class. Let's see if it works:

```
// Ex14_01.cpp
// Behavior of inherited functions in a derived class
#include "Box.h"                // For the Box class
#include "ToughPack.h"          // For the ToughPack class

int main()
{
    Box box {20.0, 30.0, 40.0};    // Define a box
    ToughPack hardcase {20.0, 30.0, 40.0}; // Declare tough box - same size

    box.showVolume();              // Display volume of base box
    hardcase.showVolume();         // Display volume of derived box
}
```

When I run the program, I get this rather disappointing output:

```
Box usable volume is 24000
Box usable volume is 24000
```

The derived class object is supposed to have a smaller capacity than the base class object, so the program is obviously not working as intended. Let's try to establish what's going wrong. The second call to `showVolume()` in `main()` is for an object of the derived class, `ToughPack`, but evidently this is not being taken into account. The volume of a `ToughPack` object should be 85 percent of that of a basic `Box` object with the same dimensions.

The trouble is that when the `volume()` function is called by the `showVolume()` function, the compiler sets it once and for all as the version of `volume()` defined in the base class. No matter how you call `showVolume()`, it will never call the `ToughPack` version of the `volume()` function. When function calls are fixed in this way before the program is executed, it is called *static resolution* of the function call, or *static binding*. The term *early binding* is also commonly used. In this example, a particular `volume()` function is bound to the call from the function `showVolume()` when the program is compiled and linked. Every time `showVolume()` is called, it uses the base class `volume()` function that's bound to it.

■ **Note** The same kind of resolution would occur in the derived class `ToughPack`. If you add a `showVolume()` function that calls `volume()` to the `ToughPack` class, the `volume()` call resolves statically to the derived class function.

What if you call the `volume()` function for the `ToughPack` object directly? As a further experiment, let's add statements in `main()` to call the `volume()` function of a `ToughPack` object directly and also through a pointer to the base class:

```
std::cout << "hardcase volume is " << hardcase.volume() << std::endl;
Box *pBox {&hardcase};
std::cout << "hardcase volume through pBox is " << pBox->volume() << std::endl;
```

Place these statements at the end of `main()`. Now when you run the program, you'll get this output:

```
Box usable volume is 24000
Box usable volume is 24000
hardcase volume is 20400
hardcase volume through pBox is 24000
```

This is quite informative. You can see that a call to `volume()` for the derived class object, `hardcase`, calls the derived class `volume()` function, which is what you want. The call through the base class pointer `pBox`, however, is resolved to the base class version of `volume()`, even though `pBox` contains the address of `hardcase`. In other words, both calls are resolved statically. The compiler implements these calls as follows:

```
std::cout << "hardcase volume is " << hardcase.ToughPack::volume() << std::endl;
Box *pBox {&hardcase};
std::cout << "hardcase volume through pBox is " << pBox->Box::volume() << endl;
```

A static function call through a pointer is determined solely by the pointer type and not by the object to which it points. The pointer `pBox` is of type pointer to `Box`, so any static call using `pBox` can only call a function member of `Box`.

■ **Note** Any call to a function through a base class pointer that is resolved statically calls a base class function.

What we want is for the `volume()` function that is to be called in any given instance to be resolved when the program executes. So, if `showVolume()` is called for a derived class object, we want the derived class `volume()` function to be called, not the base class version. When the `volume()` function is called through a base class pointer, we want the `volume()` function that is appropriate to the object pointed to to be called. This sort of operation is referred to as *dynamic binding*, or *late binding*. To make this work we have to tell the compiler that the `volume()` function in `Box` and any overrides in the classes derived from `Box` are special, and calls to them are to be resolved dynamically. We can obtain this effect by specifying that `volume()` in the base class is a *virtual function*, which will result in a *virtual function call* for `volume()`.

Virtual Functions

When you specify a function as virtual in a base class, you indicate to the compiler that you want dynamic binding for function calls in any class that's derived from this base class. A virtual function is declared in a base class by using the keyword `virtual`, as shown in Figure 14-2.

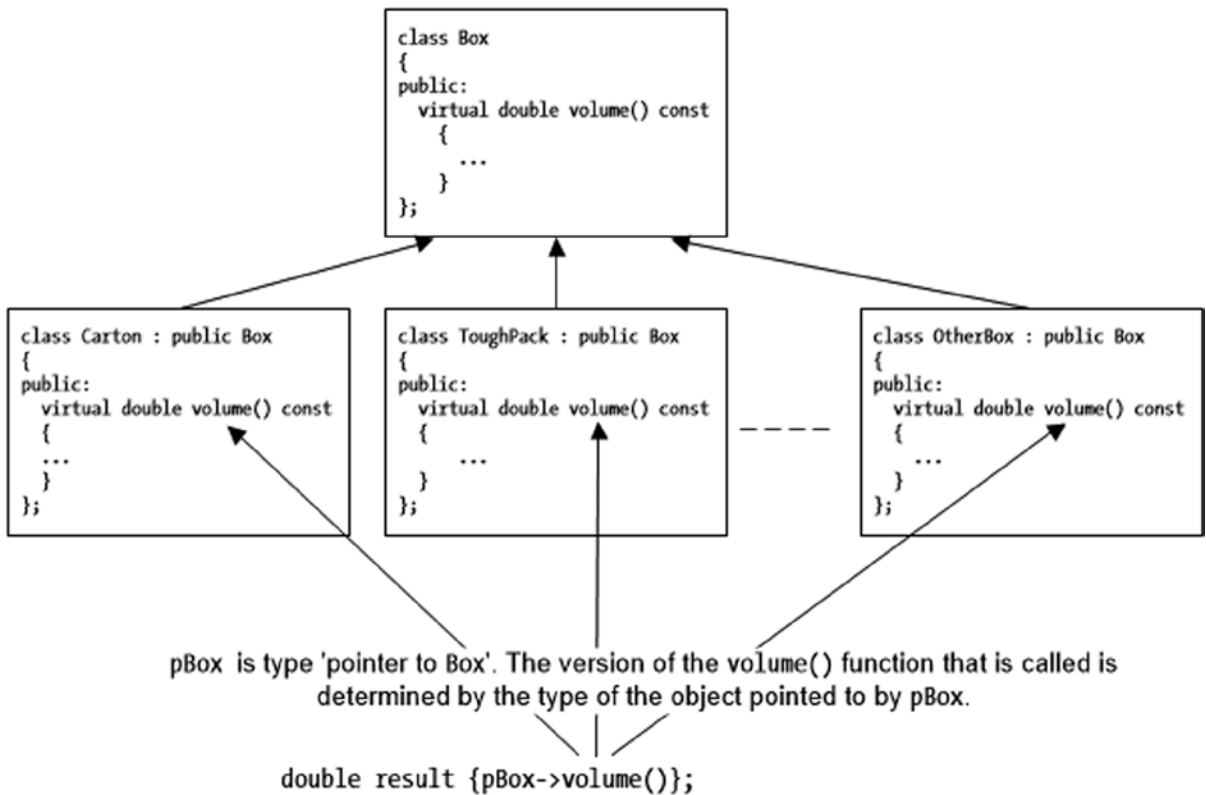


Figure 14-2. Calling a virtual function

A function that you specify as virtual in a base class will be virtual in all classes that are directly or indirectly derived from the base. This is the case whether or not you specify the function as virtual in a derived class. To obtain polymorphic behavior, each derived class may implement its own version of the virtual function (although it's not obliged to — we'll look into that later). You make virtual function calls using a variable whose type is a pointer or a reference to a base class object. Figure 14-2 illustrates how a call to a virtual function through a pointer is resolved dynamically. The pointer to the base class type is used to store the address of an object with a type corresponding to one of the derived classes. It could point to an object of any of the three derived classes shown or, of course, to a base class object. The type of the object to which the pointer points when the call executes determines which volume() function is called. Describing a class as *polymorphic* means that it is a derived class that contains at least one virtual function.

Note that a call to a virtual function using an object is *always* resolved statically. You *only* get dynamic resolution of calls to virtual functions through a pointer or a reference. Storing an object of a derived class type in a variable of a base type will result in the derived class object being sliced, so it has not derived class characteristics. With that said,

let's give virtual functions a whirl. To make the previous example work as it should, a very small change to the `Box` class is required. I just need to add the `virtual` keyword to the definition of the `volume()` function

```
class Box
{
    // Rest of the class as before...

public:
    // Function to calculate the volume of a Box object
    virtual double volume() const { return length*width*height; }
};
```

■ **Caution** If a function member definition is outside the class definition, you must *not* add the `virtual` keyword to the function definition; it would be an error to do so.

To make it more interesting, let's implement the `volume()` function in a new class called `Carton` a little differently. Here is the class definition:

```
// Carton.h
#ifndef CARTON_H
#define CARTON_H
#include <string>
#include "Box.h"
using std::string;

class Carton : public Box
{
private:
    string material;

public:
    // Constructor explicitly calling the base constructor
    Carton(double lv, double wv, double hv, string str="cardboard") : Box {lv,wv,hv}
    { material = str; }

    // Function to calculate the volume of a Carton object
    double volume() const
    {
        double vol {(length - 0.5)*(width - 0.5)*(height - 0.5)};
        return vol > 0.0 ? vol : 0.0;
    }
};
#endif
```

The `volume()` function for a `Carton` object assumes the thickness of the material is 0.25, so 0.5 is subtracted from each dimension to account for the sides of the carton. If a `Carton` object has been created with any of its dimensions less than 0.5 for some reason, then this will result in a negative value for the volume, so in such a case, the carton's volume will be set to zero.

I'll also use the ToughPack class from Ex14_01. Here's the code for the source file containing main():

```
// Ex14_02.cpp
// Using virtual functions
#include <iostream>
#include "Box.h"                // For the Box class
#include "ToughPack.h"         // For the ToughPack class
#include "Carton.h"            // For the Carton class

int main()
{
    Box box {20.0, 30.0, 40.0};
    ToughPack hardcase {20.0, 30.0, 40.0};    // A derived box - same size
    Carton carton {20.0, 30.0, 40.0, "plastic"}; // A different derived box

    box.showVolume();                // Volume of Box
    hardcase.showVolume();           // Volume of ToughPack
    carton.showVolume();             // Volume of Carton

    // Now using a base pointer...
    Box* pBox {&box};               // Points to type Box
    std::cout << "\nbox volume through pBox is " << pBox->volume() << std::endl;
    pBox->showVolume();

    pBox = &hardcase;               // Points to type ToughPack
    std::cout << "hardcase volume through pBox is " << pBox->volume() << std::endl;
    pBox->showVolume();

    pBox = &carton;                 // Points to type Carton
    std::cout << "carton volume through pBox is " << pBox->volume() << std::endl;
    pBox->showVolume();
}
```

The output that is produced should be as follows:

```
Box usable volume is 24000
Box usable volume is 20400
Box usable volume is 22722.4

box volume through pBox is 24000
Box usable volume is 24000
hardcase volume through pBox is 20400
Box usable volume is 20400
carton volume through pBox is 22722.4
Box usable volume is 22722.4
```

The virtual keyword applied to the function volume() in the base class is sufficient to determine that all definitions of the function in derived classes will also be virtual. You can optionally use the virtual keyword for your derived class functions as well, as illustrated in Figure 14-2. However, it's better not to, as I'll explain later in this chapter.

The program is now clearly doing what was wanted. The call to `showVolume()` for the `box` object calls the base class version of `volume()`, because `box` is of type `Box`. The next call to `showVolume()` for the `ToughPack` object, hardcase calls the `showVolume()` function inherited from the `Box` class but the call to `volume()` in `showVolume()` is resolved to the version defined in the `ToughPack` class because `volume()` is a virtual function. Therefore you get the volume calculated appropriately for a `ToughPack` object. The third call of `showVolume()` for the `carton` object calls the `Carton` class version of `volume()` so you get the correct result for that too.

Next, you use the pointer `pBox` to call the `volume()` function directly and also indirectly through the `showVolume()` function. The pointer first contains the address of the `Box` object `myBox`, then the addresses of the two derived class objects in turn. The resulting output for each object shows that the appropriate version of the `volume()` function is selected automatically in each case, so you have a clear demonstration of polymorphism in action.

Requirements for Virtual Function Operation

For a function to behave “virtually,” its definition in a derived class must have the same signature as it has in the base class. If the base class function is `const`, then the derived class function must also be `const`. Generally, the return type of a virtual function in a derived class must be the same as that in the base class, but there’s an exception when the return type in the base class is a pointer or a reference to a class type. In this case, the derived class version of a virtual function may return a pointer or a reference to a more specialized type than that of the base. I won’t be going into this further, but in case you come across it elsewhere, the technical term used in relation to these return types is *covariance*.

The rules for defining virtual functions imply that if you try to use different parameters for a virtual function in a derived class from those in the base class, then the virtual function mechanism won’t work. The function in the derived class will operate with static binding that is established and fixed at compile time. This is also the case if you forget to declare a derived class function as `const` when the base class function is `const`.

You can test this out by deleting the `const` keyword from the definition of `volume()` in the `Carton` class and running `Ex14_02` again. The `volume()` function signature in `Carton` no longer matches the virtual function in `Box` so the derived class `volume()` function is not virtual. Consequently, the resolution is static so that the function called for `Carton` objects through a base pointer, or even indirectly through the `showVolume()` function, is the base class version.

If the function name and parameter list of a function in a derived class are the same as those of a virtual function declared in the base class, then the return type must be consistent with the rules for a virtual function. If it isn’t, the derived class function won’t compile. Another restriction is that a virtual function can’t be a template function.

Virtual Functions and Class Hierarchies

If you want your function to be treated as virtual when it is called using a base class pointer, then you must declare it as virtual in the base class. You can have as many virtual functions as you want in a base class, but not all virtual functions need to be declared within the most basic base class in a hierarchy. This is illustrated in Figure 14-3.

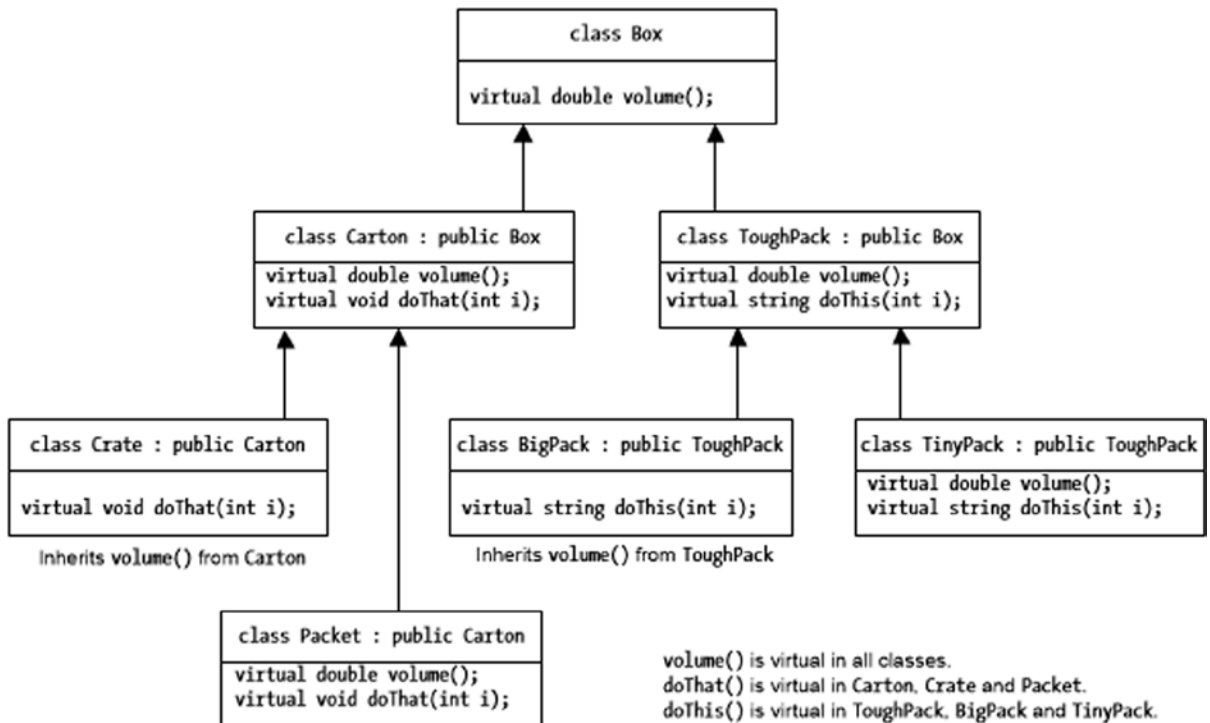


Figure 14-3. Virtual functions in a hierarchy

When you specify a function as virtual in a class, the function is virtual in all classes derived directly or indirectly from that class. All of the classes derived from the Box class in Figure 14-3 inherit the virtual nature of the volume() function. You can call volume() for objects any of these class types through a pointer of type Box* because the pointer can contain the address of an object of any class in the hierarchy:

The Crate class doesn't define volume(), so the version inherited from Carton would be called for Crate objects. It is inherited as a virtual function and therefore can be called polymorphically.

A pointer pCarton, of type Carton*, could also be used to call volume(), but only for objects of the Carton class and the two classes that have Carton as a base: Crate and Packet.

The Carton class and the classes derived from it also contain the virtual function doThat(). This function can also be called polymorphically using a pointer of type Carton*. Of course you cannot call doThat() for these classes using a pointer of type Box* because the Box class doesn't define the function doThat().

Similarly, the virtual function doThis() could be called for objects of type ToughPack, BigPack, and TinyPack using a pointer of type ToughPack*. Of course, the same pointer could also be used to call the volume() function for objects of these class types.

Using override

It's easy to make a mistake in the specification of a virtual function in a derived class. If you define `Volume()` in a class derived from `Box` it will not be virtual because the virtual function in the base class is `volume()`. This means that calls to `Volume()` will be resolved statically and the virtual `volume()` function in the class will be inherited from the base class. The code may still compile and execute, but not correctly. This kind of error can be difficult to spot. You can protect against such errors by using the `override` specifier for every virtual function declaration in a derived class, like this:

```
class Carton : public Box
{
    // Details of the class as in Ex14_02...

public:
    double volume() const override
    {
        // Function body as before...
    }
};
```

The `override` specification causes the compiler to verify that the base class declares a class member with the same signature. If it doesn't the compiler flags the definition here as an error. The `override` specification only appears within the class definition. It must not be applied to an external definition of a member function. If you always specify a virtual function override in a derived class using `override`, it's clear to anyone reading the class definition that this is a virtual function so there's no need to apply the `virtual` keyword in addition. It's a good idea to limit the use of the `virtual` keyword to base class functions and apply the `override` specification to all virtual function overrides in derived classes.

Using final

Sometimes you may want to prevent a function member from being overridden in a derived class. This could be because you want to limit how a derived class can modify the behavior of the class interface for example. You can do this by specifying that a function is `final`. You could prevent the `volume()` function in the `Carton` class from being overridden by definitions in classes derived from `Carton` by specifying it like this:

```
class Carton : public Box
{
    // Details of the class as in Ex14_02...

public:
    double volume() const override final
    {
        // Function body as before...
    }
};
```

Attempts to override `volume()` in classes that have `Carton` as a base will result in a compiler error. This ensures that only the `Carton` version can be used for derived class objects.

You can also specify a class as `final`, like this for example:

```
class Carton final : public Box
{
    // Details of the class as in Ex14_02...

public:
    double volume() const override
    {
        // Function body as before...
    }
};
```

Now the compiler will not allow `Carton` to be used as a base class. No further derivation from the `Carton` class is possible.

■ **Note** `final` and `override` are not keywords because making them keywords could break code that was written before they were introduced. This means that you could use `final` and `override` as names in your code, but don't; it only creates confusion.

Access Specifiers and Virtual Functions

The access specification of a virtual function in a derived class can be different from the specification in the base class. When you call the virtual function through a base class pointer, the access specification in the base class determines whether the function is accessible, regardless of the type of object pointed to. If the virtual function is `public` in the base class, it can be called for any derived class through a pointer (or a reference) to the base class, regardless of the access specification in the derived class. I can demonstrate this by modifying the previous example. Modify the `ToughPack` class definition from the to make the `volume()` function protected, and add the `virtual` keyword to its declaration:

```
class ToughPack : public Box
{
public:
    // Constructor
    ToughPack(double lv, double ww, double hv) : Box {lv, ww, hv} {}

protected:
    // Function to calculate volume of a ToughPack allowing 15% for packing
    double volume() const override { return 0.85*length*width*height; }
};
```

The `main()` function changes very slightly with a commented out statement added:

```
// Ex14_03.cpp
// Access specifiers and virtual functions
#include <iostream>
#include "Box.h"                // For the Box class
#include "ToughPack.h"          // For the ToughPack class
#include "Carton.h"              // For the Carton class
```

```

int main()
{
    Box box {20.0, 30.0, 40.0};
    ToughPack hardcase {20.0, 30.0, 40.0};    // A derived box - same size
    Carton carton {20.0, 30.0, 40.0, "plastic"}; // A different derived box

    box.showVolume();                        // Volume of Box
    hardcase.showVolume();                  // Volume of ToughPack
    carton.showVolume();                    // Volume of Carton

    // Uncomment the following statement for an error
    // std::cout << "hardcase volume is " << hardcase.volume() << std::endl;

    // Now using a base pointer...
    Box* pBox {&box};                      // Points to type Box
    std::cout << "\nbox volume through pBox is " << pBox->volume() << std::endl;
    pBox->showVolume();

    pBox = &hardcase;                      // Points to type ToughPack
    std::cout << "hardcase volume through pBox is " << pBox->volume() << std::endl;
    pBox->showVolume();

    pBox = &carton;                        // Points to type Carton
    std::cout << "carton volume through pBox is " << pBox->volume() << std::endl;
    pBox->showVolume();
}

```

It should come as no surprise that this code produces exactly the same output as the last example. Even though `volume()` is declared as protected in the `ToughPack` class, you can still call it for the `hardcase` object through the `showVolume()` function that is inherited from the `Box` class. You can also call it directly through a pointer to the base class, `pBox`. However, if you uncomment the line that calls the `volume()` function directly using the `hardcase` object, the code won't compile.

What matters here is whether the call is resolved dynamically or statically. When you use a class object, the call is determined statically by the compiler. Calling `volume()` for a `ToughPack` object calls the function defined in that class. Because the `volume()` function is protected in `ToughPack`, the call for the `hardcase` object won't compile. All the other calls are resolved when the program executes; they are polymorphic calls. In this case, the access specification for a virtual function in the base class is inherited in all the derived classes. This is regardless of the explicit specification in the derived class; the explicit specification only affects calls that are resolved statically.

Default Argument Values in Virtual Functions

Default argument values are dealt with at compile time so you can get unexpected results when you use default argument values with virtual function parameters. If the base class declaration of a virtual function has a default argument value and you call the function through a base pointer, you'll always get the default argument value from

the base class version of the function. Any default argument values in derived class versions of the function will have no effect. I can demonstrate this by altering the previous example to include a parameter with a default argument value for the `volume()` function in all three classes. Change the definition of the `volume()` function in the `Box` class to:

```
virtual double volume(int i = 5) const
{
    std::cout << "Box parameter = " << i << std::endl;
    return length*width*height;
}

In the Carton class it should be:
double volume(int i = 50) const override
{
    std::cout << "Carton parameter = " << i << std::endl;
    double vol {(length - 0.5)*(width - 0.5)*(height - 0.5)};
    return vol > 0.0 ? vol : 0.0;
}
```

Finally in the `ToughPack` class you can define `volume()` as follows, and make it public once more:

```
public:
    double volume(int i = 500) const override
    {
        std::cout << "ToughPack parameter = " << i << std::endl;
        return 0.85*length*width*height;
    }
```

The parameter serves no purpose here other than to demonstrate how default values are assigned.

Once you've made these changes to the class definitions, you can try out the default parameter values with the `main()` function from the previous example, in which you uncomment the line that calls the `volume()` member for the `hardcase` object directly. The complete program is in the download as `Ex14_04`. You'll get this output:

```
Box parameter = 5
Box usable volume is 24000
ToughPack parameter = 5
Box usable volume is 20400
Carton parameter = 5
Box usable volume is 22722.4
ToughPack parameter = 500
hardcase volume is 20400
Box parameter = 5

box volume through pBox is 24000
Box parameter = 5
Box usable volume is 24000
ToughPack parameter = 5
hardcase volume through pBox is 20400
ToughPack parameter = 5
Box usable volume is 20400
Carton parameter = 5
carton volume through pBox is 22722.4
Carton parameter = 5
Box usable volume is 22722.4
```

In every instance of when `volume()` is called except one, the default parameter value output is that specified for the base class function. The exception is when you call `volume()` using the `hardcase` object. This is resolved statically to `volume()` in the `ToughPack` class so the default parameter value specified in the `ToughPack` class is used. All the other calls are resolved dynamically so the default parameter value specified in the base class applies, even though the function executing is in a derived class.

Virtual Function Calls with Smart Pointers

Polymorphism works equally well with smart pointers. I'll demonstrate this using the `Box`, `Carton`, and `ToughPack` classes from `Ex14_03` and revise `main()` to use smart pointers:

```
// Ex14_05.cpp
// Virtual functions using smart pointers
#include <iostream>
#include <memory>                // For smart pointers
#include <vector>                // For vector
#include "Box.h"                // For the Box class
#include "ToughPack.h"          // For the ToughPack class
#include "Carton.h"             // For the Carton class

int main()
{
    std::vector<std::shared_ptr<Box>>boxes;
    boxes.push_back(std::make_shared<Box>(20.0, 30.0, 40.0));
    boxes.push_back(std::make_shared<ToughPack>(20.0, 30.0, 40.0));
    boxes.push_back(std::make_shared<Carton>(20.0, 30.0, 40.0, "plastic"));

    for (auto& p : boxes)
        p->showVolume();
}
```

The output from this example is:

```
Box usable volume is 24000
Box usable volume is 20400
Box usable volume is 22722.4
```

The output shows that polymorphism is alive and well with smart pointers. The elements in the `boxes` vector are of type `std::shared_ptr<Box>`, which are smart pointers to `Box` objects. The elements can store addresses for objects of `Box` or any class derived from `Box`, so there's an exact parallel with the raw pointers you have seen up to now. When you are creating objects on the heap, using smart pointers still gives you polymorphic behavior and removes the potential for memory leaks.

Using References to Call Virtual Functions

You can call a virtual function through a reference; reference parameters are particularly powerful tools for applying polymorphism. Calling a virtual function through a variable that is a reference doesn't have the same magic as calling through a pointer because a reference is initialized once and only once so it can only ever call functions for that object. Calling a function that has a reference parameter is a different matter.

You can pass a base class object or any derived class object to a function with a parameter that's a reference to the base class. You can use the reference parameter within the function body to call a virtual function in the base class and get polymorphic behavior. When the function executes, the virtual function for the object that was passed as the argument is selected automatically at runtime. I can show this in action by modifying Ex14_02 to call a function that has a parameter of type reference to Box:

```
// Ex14_06.cpp
// Using a reference parameter to call virtual function
#include <iostream>
#include "Box.h"                // For the Box class
#include "ToughPack.h"          // For the ToughPack class
#include "Carton.h"              // For the Carton class

// Global function to display the volume of a box
void showVolume(const Box& rBox)
{
    std::cout << "Box usable volume is " << rBox.volume() << std::endl;
}

int main()
{
    Box box {20.0, 30.0, 40.0};           // A base box
    ToughPack hardcase {20.0, 30.0, 40.0}; // A derived box - same size
    Carton carton {20.0, 30.0, 40.0, "plastic"}; // A different derived box

    showVolume(box);                     // Display volume of base box
    showVolume(hardcase);                 // Display volume of derived box
    showVolume(carton);                   // Display volume of derived box
}
```

Running this program should produce this output:

```
Box usable volume is 24000
Box usable volume is 20400
Box usable volume is 22722.4
```

The class definitions are the same as in Ex14_02. There's a new global function that calls `volume()` using its reference parameter to call the `volume()` member of an object. `main()` defines the same objects as in Ex14_02 but calls the global `showVolume()` function with each of the objects to output their volumes. As you see from the output, the correct `volume()` function is being used in each case, confirming that polymorphism works through a reference parameter.

Each time the `showVolume()` function is called, the reference parameter is initialized with the object that is passed as an argument. Because the parameter is a reference to a base class, the compiler arranges for dynamic binding to the virtual `volume()` function.

Calling the Base Class Version of a Virtual Function

You've seen that it's easy to call the derived class version of a virtual function through a pointer or reference to a derived class object — the call is made dynamically. However, what do you do when you actually want to call the base class function for a derived class object?

The `Box` class provides an opportunity to see why such a call might be required. It could be useful to calculate the loss of volume in a `Carton` or `ToughPack` object; one way to do this would be to calculate the difference between the volumes returned from the base and derived class versions of the `volume()` function. You can force the virtual function for a base class to be called statically by qualifying it with the class name. Suppose you have a pointer `pBox` that's defined like this:

```
Carton carton {40.0, 30.0, 20.0};
Box* pBox {&carton};
```

You can calculate the loss in total volume for a `Carton` object with this statement:

```
double difference {pBox->Box::volume() - pBox->volume()};
```

The expression `pBox->Box::volume()` calls the base class version of the `volume()` function. The class name, together with the scope resolution operator, identifies a particular `volume()` function, so this will be a static call resolved at compile time.

■ **Note** You can call the base class implementation of any member function using the scope resolution operator, provided the access specifier for the function allows it.

You can't use a class name qualifier to force the selection of a particular derived class function in a call through a pointer to the base class. The expression `pBox->Carton::volume()` won't compile because `Carton::volume()` is not a member of the `Box` class. A call of a function through a pointer is either a static call to a function member of the class type for the pointer, or it is a dynamic call to a virtual function.

Calling the base class version of a virtual function through an object of a derived class is also simple. You can use a *static cast* to convert the derived class object to the base class; then you can use the result to call the base class function. You can calculate the loss in volume for the `carton` object with this statement:

```
double difference {static_cast<Box>(carton).volume() - carton.volume()};
```

Both calls in this statement are resolved statically. Casting `carton` to type `Box` results in an object of type `Box`, so the function call will be to the `Box` version of `volume()`. Calls to virtual functions using an object are always resolved statically.

Converting Between Pointers to Class Objects

You can implicitly convert a pointer to a derived class to a pointer to a base class, and you can do this for both direct and indirect base classes. For example, let's first define a pointer to a `Carton` object:

```
Carton* pCarton {new Carton {30, 40, 10}};
```

You can convert this pointer implicitly to a pointer to a direct base class of `Carton`:

```
Box* pBox {pCarton};
```

The result is a pointer to `Box`, which is initialized to point to the new `Carton` object. You know from Ex14_05 that this also works with smart pointers. That example stored derived class smart pointers as base class smart pointers in the vector container.

Let's look at converting a pointer to a derived class type to a pointer to an indirect base. Suppose you define a `CerealPack` class with `Carton` as the public base class. `Box` is a direct base of `Carton` so it is an indirect base of `CerealPack`. Therefore you can write the following:

```
Box* pBox {pCerealPack};
```

This statement converts the address in `pCerealPack` from type pointer to `CerealPack` to type pointer to `Box`. If you need to specify the conversion explicitly, you can use the `static_cast<>()` operator:

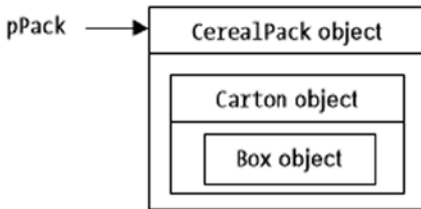
```
Box* pBox {static_cast<Box*>(pCerealPack)};
```

The compiler can usually expedite this cast because `Box` is a base class of `CerealPack`. This would not be legal if the `Box` class was inaccessible or was a virtual base class.

The result of casting a derived class pointer to a base pointer type is a pointer to the sub-object of the destination type. It's easy to get confused when thinking about casting pointers to class types. Don't forget that a pointer to a class type can only point to objects of that type, or to objects of a derived class type, and not the other way round. To be specific, the pointer `pCarton` could contain the address of an object of type `Carton` (which could be a subobject of a `CerealPack` object), or an object of type `CerealPack`. It could not contain the address of an object of type `Box`, because a `CerealPack` object is a specialized kind of `Carton`, but a `Box` object isn't. Figure 14-4 illustrates the possibilities between pointers to `Box`, `Carton`, and `CerealPack` objects.

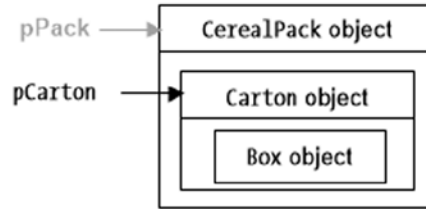
Creating the pointer:

```
CerealPack* pPack {new CerealPack};
```



Cast to the direct base:

```
pCarton = pPack;
```



Cast to the indirect base:

```
pBox = pPack;
```

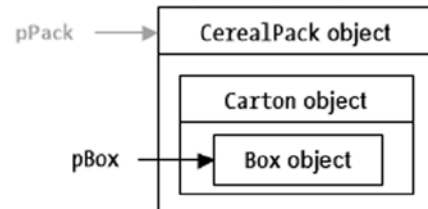


Figure 14-4. Casting pointers up a class hierarchy

Despite what I have said so far about casting pointers up a class hierarchy, it's sometimes possible to make casts in the opposite direction. Casting a pointer down a hierarchy from a base to a derived class is different; whether or not a cast works depends on the type of object to which the base pointer is pointing. For a static cast from a base class pointer such as `pBox` to a derived class pointer such as `pCarton` to be legal, the base class pointer must be pointing to a `Box` sub-object of a `Carton` object. If that's not the case, the result of the cast is undefined.

Figure 14-5 shows static casts from a pointer, `pBox`, that contains the address of a `Carton` object. The cast to type `Carton*` will work because the object is of type `Carton`. The result of the cast to type `CerealPack*` on the other hand, is undefined because no object of this type exists.

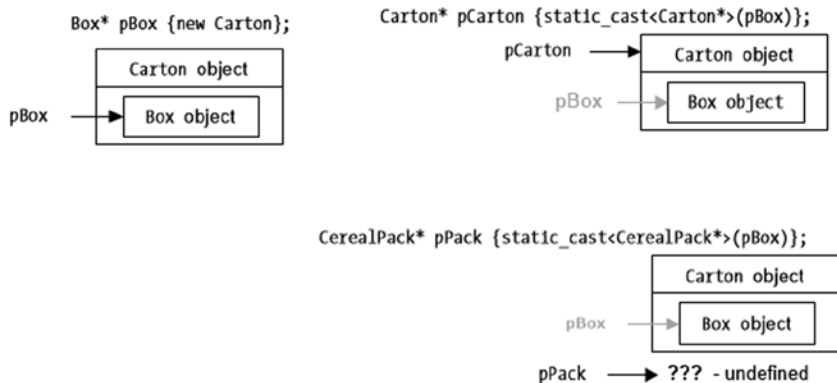


Figure 14-5. Casting pointers down a class hierarchy

If you're in any doubt about the legitimacy of a static cast, you shouldn't use it. The success of an attempt to cast a pointer down a class hierarchy depends on the pointer containing the address of an object of the destination type. A static cast doesn't check whether this is the case so if you attempt it in circumstances where you don't know what the pointer points to, you risk an undefined result. Therefore, when you want to cast down a hierarchy, you need to do it differently — in a way in which the cast can be checked at runtime.

Dynamic Casts

A dynamic cast is a conversion that's performed at runtime. The `dynamic_cast<>()` operator performs a dynamic cast. You can only apply this operator to pointers and references to polymorphic class types, which are class types that contain at least one virtual function. The reason is that only pointers to polymorphic class types contain the information that the `dynamic_cast<>()` operator needs to check the validity of the conversion. This operator is specifically for the purpose of converting between pointers or references to class types in a hierarchy. Of course, the types you are casting between must be pointers or references to classes within the same class hierarchy. You can't use `dynamic_cast<>()` for anything else. I'll first discuss casting pointers dynamically.

Casting Pointers Dynamically

There are two kinds of dynamic cast. The first is a "cast down a hierarchy," from a pointer to a direct or indirect base type to a pointer to a derived type. This is called a *downcast*. The second possibility is a cast across a hierarchy; this is referred to as a *crosscast*. Figure 14-6 illustrates these.

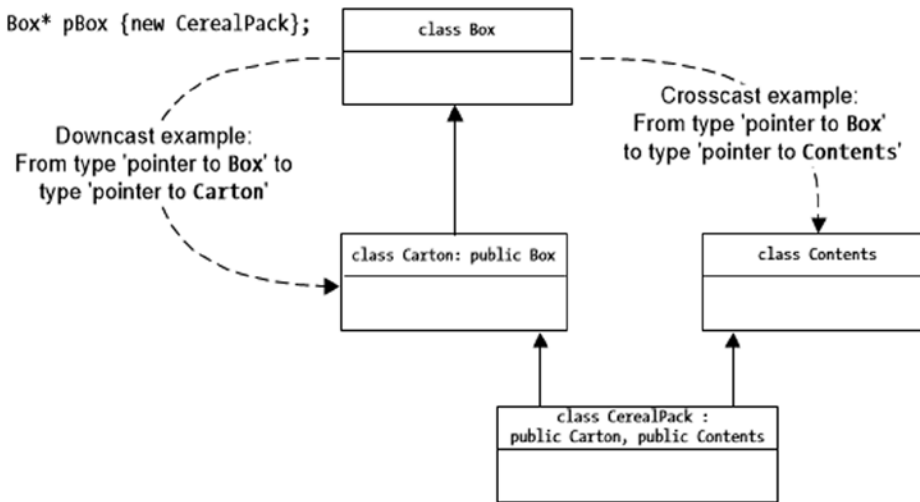


Figure 14-6. Downcasts and crosscasts

For a pointer, `pBox`, of type `Box*` that contains the address of a `CerealPack` object, you could write the downcast shown in Figure 14-6 as:

```
Carton* pCarton {dynamic_cast<Carton*>(pBox)};
```

The `dynamic_cast<>()` operator is written in the same way as the `static_cast<>()` operator. The destination type goes between the angled brackets following `dynamic_cast`, and the expression to be converted to the new type goes between the parentheses. For this cast to be legal, the `Box` and `Carton` classes must contain virtual functions, either as declared or inherited members. For the cast above to work, `pBox` must point to either a `Carton` object or a `CerealPack` object, because only objects of these types contain a `Carton` sub-object. If the cast doesn't succeed, the pointer `pCarton` will be set to `nullptr`.

The crosscast in Figure 14-6 could be written as:

```
Contents* pContents {dynamic_cast<Contents*>(pBox)};
```

As in the previous case, both the `Contents` class and the `Box` class must be polymorphic for the cast to be legal. The cast can only succeed if `pBox` contains the address of a `CerealPack` object because this is the only type that contains a `Contents` object and can be referred to using a pointer of type `Box*`. Again, if the cast doesn't succeed, `nullptr` will be stored in `pContents`.

Using `dynamic_cast<>()` to cast down a class hierarchy may fail, but in contrast to the static cast, the result will be `nullptr` rather than just “undefined.” This provides a clue as to how you can use this. Suppose you have some kind of object pointed to by a pointer to `Box`, and you want to call a non-virtual function member of the `Carton` class. A base class pointer only allows you to call the virtual function members of a derived class, but the `dynamic_cast<>()` operator can enable you to call a non-virtual function. If `surface()` is a non-virtual function member of the `Carton` class, you could call it with this statement:

```
dynamic_cast<Carton*>(pBox)->surface();
```

This is obviously hazardous. You need to be sure that `pBox` is pointing to a `Carton` object, or to an object of a class that has the `Carton` class as a base. If this is not the case, the `dynamic_cast<>()` operator returns `nullptr`, and the call fails. To fix this, you can use the `dynamic_cast<>()` operator to determine whether what you intend to do is valid; for example:

```
Carton* pCarton {dynamic_cast<Carton*>(pBox)}
if(pCarton)
    pCarton->surface();
```

Now you'll only call the `surface()` function member if the result of the cast is not `nullptr`. Note that you can't remove `const`-ness with `dynamic_cast<>()`. If the pointer type you're casting from is `const`, then the pointer type you are casting to must also be `const`. If you want to cast from a `const` pointer to a non-`const` pointer, you must first cast to a non-`const` pointer of the same type as the original using the `const_cast<>()` operator.

Converting References

You can apply the `dynamic_cast<>()` operator to a reference parameter in a function to cast down a class hierarchy to produce another reference. In the following example, the parameter to the function `doThat()` is a reference to a base class `Box` object. In the body of the function, you can cast the parameter to a reference to a derived type:

```
double doThat(Box& rBox)
{
    ...
    Carton& rCarton {dynamic_cast<Carton&>(rBox)};
    ...
}
```

This statement casts from type reference to `Box` to type reference to `Carton`. Of course, it's possible that the object passed as an argument may not be a `Carton` object and if this is the case, the cast won't succeed. There is no such thing as a null reference, so this fails in a different way from a failed pointer cast: execution of the function stops and an exception of type `bad_cast` is thrown. You haven't met exceptions yet, but you'll find out what this means in the next chapter.

Determining the Polymorphic Type

Applying a dynamic cast to a reference blind is obviously risky but there's an alternative. You can check the dynamic type of a pointer or reference at runtime using the `typeid` operator. You can apply `typeid` to a type or to an expression. The operator returns a `std::type_info` object that encapsulates the actual type; the `std::type_info` class is declared in the `typeinfo` Standard Library header. You can use `typeid` to determine whether an object passed as an argument

for a function parameter that is a reference is of a given type and avoid the risk of an illegal dynamic cast of the reference parameter. You could improve the example from the preceding section like this:

```
double doThat(Box& rBox)
{
    ...
    double area {};
    if(typeid(rBox) == typeid(Carton))
    { // rBox does reference a Carton object so the cast is legal
        Carton& rCarton {dynamic_cast<Carton&>(rBox)};
        area {rCarton.surface()};           // Call non-virtual function member
    }
    ...
}
```

The `std::type_info` class supports comparisons for equality so you can compare the `type_info` object for the `Carton` class with that produced by applying `typeid` to the parameter, `rBox`. If the `type_info` objects are equal, then calling the non-virtual member of the `Carton` class using `rBox` will work because `rBox` definitely references a `Carton` object.

The `std::type_info` class defines a `name()` member that returns the type name as a C-style string. The following statement would output the type name for the object referenced by `rBox`:

```
std::cout << "Object referenced is of type " << typeid(rBox).name() << std::endl;
```

The Cost of Polymorphism

As you know, there's no such thing as a free lunch — this certainly applies to polymorphism. You pay for polymorphism in two ways: it requires more memory, and virtual function calls result in additional overhead. These consequences arise because of the way that virtual function calls are typically implemented in practice.

For instance, suppose two classes, `A` and `B`, contain identical data members, but `A` contains virtual functions, whereas `B`'s functions are non-virtual. In this case, an object of type `A` requires more memory than an object of type `B`.

■ **Note** You can create a simple program with two such class objects and use the `sizeof` operator to see the difference in memory occupied by objects with and without virtual functions.

The reason for the increase in memory is that when you create an object of a polymorphic class type, a special pointer is created in the object. This pointer is used to call any of the virtual functions in the object. The special pointer points to a table of function pointers that gets created for the class. This table, usually called a *vtable*, has one entry for each virtual function in the class. Figure 14-7 illustrates this.

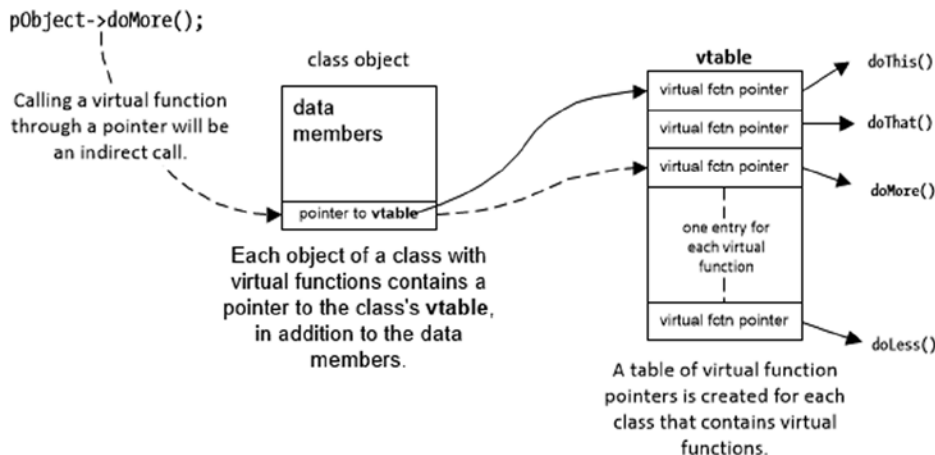


Figure 14-7. How polymorphic function calls work

When a function is called through a pointer to a base class object, the following sequence of events occurs:

1. The pointer to the vtable in the object pointed to is used to find the beginning of the vtable for the class.
2. The entry for the function to be called is found in the vtable, usually by using an offset.
3. The function is called indirectly through the function pointer in the vtable. This indirect call is a little slower than a direct call of a non-virtual function, so each virtual function call carries some overhead.

However, the overhead in calling a virtual function is small, and shouldn't give you cause for concern. A few extra bytes per object and slightly slower function calls are small prices to pay for the power and flexibility that polymorphism offers. This explanation is just so you'll know why the size of an object that has virtual functions is larger than that of an equivalent object that doesn't.

Pure Virtual Functions

There are situations that require a base class with a number of classes derived from it, and a virtual function that's redefined in each of the derived classes, but where there's no meaningful definition for the function in the base class. For example, you might define a base class, *Shape*, from which you derive classes defining specific shapes, such as *Circle*, *Ellipse*, *Rectangle*, *Curve*, and so on. The *Shape* class could include a virtual function *draw()* that you'd call for a derived class object to draw a particular shape, but the *Shape* class itself has no meaningful implementation of the *draw()* function because it doesn't define anything that can be drawn. This is a job for a *pure virtual function*.

The purpose of a pure virtual function is to enable the derived class versions of the function to be called polymorphically. To declare a pure virtual function rather than an "ordinary" virtual function that has a definition, you use the same syntax, but add `= 0` to its declaration within the class.

If all this sounds confusing in abstract terms, you can see how to declare a pure virtual function by taking the concrete example of defining the `Shape` class I just alluded to:

```
// Generic base class for shapes
class Shape
{
protected:
    Point position;                // Position of a shape

    // Constructor
    Shape(const Point& shapePosition) : position {shapePosition} {}

public:

    virtual void draw() const = 0;    // Pure virtual function to draw a shape

    virtual void move(const Point& newPosition) = 0; // Pure virtual function to move a shape
};
```

The `Shape` class contains a data member of type `Point` (which is another class type) that stores the position of a shape. It's a base class member because every shape must have a position, and the `Shape` constructor initializes it. The `draw()` function is virtual because it's qualified with the `virtual` keyword and it's pure because the `= 0` following the parameter list specifies that there's no definition for the function in this class. The `draw()` function is a pure virtual function in the `Shape` class. The `move()` function is also a pure virtual function.

A class that contains a pure virtual function is called an *abstract class*. The `Shape` class contains two pure virtual functions — `draw()` and `move()` — so it is most definitely an abstract class. Let's look a little more at exactly what this means.

Abstract Classes

Even though it has a data member and a constructor, the `Shape` class is an incomplete description of an object, because the `draw()` and `move()` functions are not defined. Therefore you're not allowed to create instances of the `Shape` class; the class exists purely for the purpose of deriving classes from it. Because you can't create objects of an abstract class, you can't use it as a function parameter type or as a return type; a parameter of type `Shape` will not compile. However, pointers or references to an abstract class can be used as parameter or return types so function parameters of type `Shape*` and `Shape&` are fine. It is essential that this should be the case to get polymorphic behavior for derived class objects.

This raises the question, "If you can't create an instance of an abstract class, then why does the abstract class contain a constructor?" The answer is that the constructor for an abstract class is there to initialize its data members. The constructor for an abstract class will be called by a derived class constructor, implicitly or from the constructor initialization list. If you try to call the constructor for an abstract class from anywhere else, you'll get an error message from the compiler.

Because the constructor for an abstract class can't be used generally, it's a good idea to declare it as a protected member of the class, as I have done for the `Shape` class. This allows it to be called in the initialization list for a derived class constructor but prevents access to it from anywhere else. Note that a constructor for an abstract class must not call a pure virtual function; the effect of doing so is undefined.

Any class that derives from the `Shape` class must define both the `draw()` function and the `move()` function if it is not also to be an abstract class. More specifically, if any pure virtual function of an abstract base class isn't defined in a derived class, then the pure virtual function will be inherited as such, and the derived class will also be an abstract class.

To illustrate this, you could define a new class called `Circle`, which has the `Shape` class as a base:

```
// Class defining a circle
class Circle : public Shape
{
protected:
    double radius;                // Radius of a circle

public:
    Circle(Point center, double circleRadius) : Shape(center), radius(circleRadius) {}

    virtual void draw() const
    {
        // Circle center is at point 'position', inherited from the base class
        std::cout << "Circle center " << position << " radius " << radius << std::endl;
    }

    virtual void move(const Point& newCenter) { position = newCenter; }
};
```

The `draw()` and `move()` functions are defined, so this class is not abstract. If either function were not defined, then the `Circle` class would be abstract. The class includes a constructor, which initializes the base class sub-object by calling the base class constructor.

■ **Note** You can *only* call the constructor of an abstract base class in the initialization list of a derived class constructor.

Of course, an abstract class can contain virtual functions that it does define and functions that are not virtual. It can also contain any number of pure virtual functions. The presence of at least one pure virtual function will make a class abstract. A derived class must have definitions for every pure virtual function in its base; otherwise, it will also be an abstract class. Let's look at a working example that uses an abstract class.

I'll define a new version of the `Box` class with the `volume()` function declared as a pure virtual function:

```
class Box
{
protected:
    double length {1.0};
    double width {1.0};
    double height {1.0};

    Box(double lv, double ww, double hv) : length {lv}, width {ww}, height {hv} {}

public:
    virtual double volume() const = 0;    // Function to calculate the volume
};
```

Because `Box` is now an abstract class, you can no longer create objects of this type. The `Carton` and `ToughPack` classes in this example are the same as in `Ex14_06`. They both define the `volume()` function, so they aren't abstract and I can use objects of these classes to show that the virtual `volume()` functions are still working as before:

```
// Ex14_07.cpp
// Using an abstract class
#include <iostream>
#include "Box.h"                // For the Box class
#include "ToughPack.h"          // For the ToughPack class
#include "Carton.h"             // For the Carton class

int main()
{
    ToughPack hardcase {20.0, 30.0, 40.0};    // A derived box - same size
    Carton carton {20.0, 30.0, 40.0, "plastic"}; // A different derived box

    Box*pBox {&hardcase};                    // Base pointer - derived address
    std::cout << "hardcase volume is " << pBox->volume() << std::endl;

    pBox = &carton;                          // New derived address
    std::cout << "carton volume is " << pBox->volume() << std::endl;
}
```

This generates the following output:

```
hardcase volume is 20400
carton volume is 22722.4
```

Declaring `volume()` to be a pure virtual function in the `Box` class ensures that the `volume()` function members of the `Carton` and `ToughPack` classes are also virtual. Therefore you can call them through a pointer to the base class, and the calls will be resolved dynamically. The output for the `ToughPack` and `Carton` objects shows that everything is working as expected. The `Carton` and `ToughPack` class constructors still call the `Box` class constructor that is now protected in their initialization lists.

Abstract Classes As Interfaces

Sometimes, an abstract class arises simply because a function has no sensible definition in the context of the class and only has a meaningful interpretation in a derived class. However, there is another way of using an abstract class. An abstract class that contains only pure virtual functions can be used to define a *standard class interface*. It would typically represent a declaration of a set of related functions that supported a particular capability — a set of functions for communications through a modem, for example. As I've discussed, a class that derives from such an abstract base class must define an implementation for each virtual function, but the way in which each virtual function is implemented is specified by whoever is implementing the derived class. The abstract class fixes the interface, but the implementation in the derived class is flexible.

Indirect Abstract Base Classes

Given a base class and a derived class, it may be that the base itself is derived from a more general base class. In this situation, the most derived class inherits indirectly from the most base class. You can create as many levels of derivation as you need. An extension of the previous example will demonstrate indirect inheritance involving an abstract class, and also illustrate the use of a virtual function across a second level of inheritance. The Carton and ToughPack classes will be the same as in Ex14_07.

You can define a `Vessel` class to represent a generic container that will be an abstract base class for the `Box` class. This will allow classes representing other types of storage containers to be derived from `Vessel` so that you can calculate volumes of these types of objects polymorphically. You could put a definition for the `Vessel` class in a new header file called `Vessel.h`:

```
// Vessel.h Abstract class defining a vessel
#ifndef VESSEL_H
#define VESSEL_H

class Vessel
{
public:
    virtual double volume() const = 0;
};
#endif
```

This is an abstract class because it contains the pure virtual function, `volume()`. You can now modify the `Box` class to use the `Vessel` class as a base:

```
// Box.h
#ifndef BOX_H
#define BOX_H

#include "Vessel.h"

class Box : public Vessel
{
protected:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv} {}

    double volume() const override { return length*width*height; }
};
#endif
```


You can add another class derived from `Vessel` that defines a can, placing the definition in `Can.h` and the implementation in `Can.cpp`. Here is the class definition:

```
// Can.h Class defining a cylindrical can of a given height and diameter
#ifndef CAN_H
#define CAN_H
#include "Vessel.h"

class Can : public Vessel
{
protected:
    double diameter {1.0};
    double height {1.0};
    constexpr static double pi {3.14159265};

public:
    Can(double d, double h) : diameter {d}, height {h} {}

    double volume() const override { return pi*diameter*diameter*height/ 4.0; }
};
#endif
```

This defines `Can` objects that represent regular cylindrical cans, such as a beer can. The class defines `pi` as a static member because it's required within a function member of the class. Qualifying the declaration of a data member with `constexpr` specifies that it is constant and that its value can be set at compile time. This allows the initial value of a static data member to be specified within the class definition. A data member that is qualified by `constexpr` is by definition `const` and can be used as a constant expression. You could replace `constexpr` by `const` and put the definition and initialization for `pi` outside the class definition.

```
// Ex14_08.cpp
// Using an indirect base class
#include <iostream>
#include <vector>           // For the vector container
#include "Box.h"           // For the Box class
#include "ToughPack.h"     // For the ToughPack class
#include "Carton.h"        // For the Carton class
#include "Can.h"           // for the Can class

int main()
{
    Box box {40, 30, 20};
    Can can {10, 3};
    Carton carton {40, 30, 20, "Plastic"};
    ToughPack hardcase {40, 30, 20};

    std::vector<Vessel*> pVessels {&box, &can, &carton, &hardcase};

    for (auto p : pVessels)
        std::cout << "Volume is " << p->volume() << std::endl;
}
```

This generates the following output:

```
Volume is 24000
Volume is 235.619
Volume is 22722.4
Volume is 20400
```

You have a three-level class hierarchy in this example, as shown in Figure 14-8.

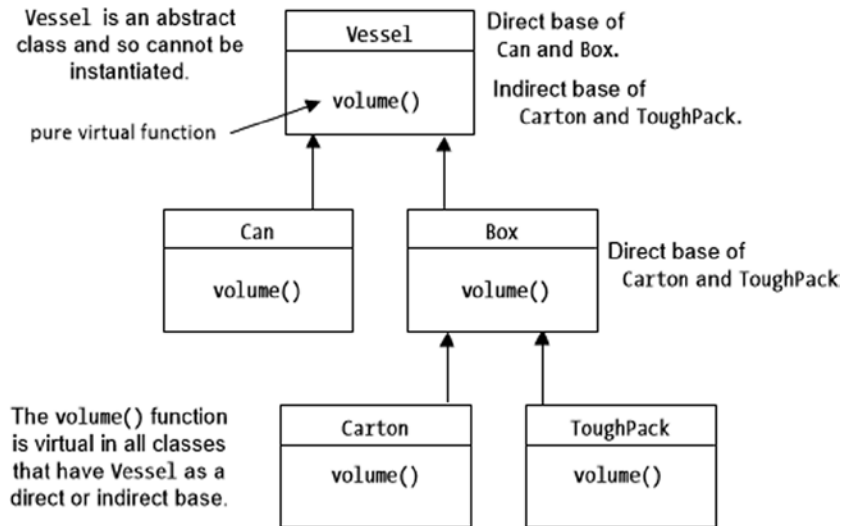


Figure 14-8. A three-level class hierarchy

If a derived class fails to define a function that's declared as a pure virtual function in the base class, then the function will be inherited as a pure virtual function, and this will make the derived class an abstract class. You can demonstrate this effect by removing the `const` declaration from either the `Can` or the `Box` class. This makes the function different from the pure virtual function in the base class, so the derived class inherits the base class version, and the program won't compile.

This time around, I used a vector of raw pointers to `Vessel` objects to exercise the virtual functions. The output shows that all the polymorphic calls of the `volume()` function work as expected.

Destroying Objects Through a Pointer

The use of pointers to a base class when you are working with derived class objects is very common, because that's how you can take advantage of virtual functions. If you use pointers to objects created on the heap, a problem can arise when derived class objects are destroyed. You can see the problem if you implement the classes in the previous

example with destructors that display a message and change `main()` to create objects in the free store. To begin, add a destructor to the `Vessel` class that just displays a message when it gets called:

```
#include <iostream>
class Vessel
{
public:
    virtual double volume() const = 0;

    ~Vessel() { std::cout << "Vessel destructor" << std::endl; }
};
```

Do the same for the `Can`, `Box`, and `ToughPack` classes, and add an output statement to the destructor for the `Carton` class. You'll need to include the `<iostream>` header into files where it's not already included.

You now need to modify `main()` from the previous example to this:

```
int main()
{
    std::vector<Vessel*> pVessels {new Box {40, 30, 20}, new Can {10, 3},
                                new Carton {40, 30, 20, "Plastic"}, new ToughPack {40, 30, 20}};

    for (auto p : pVessels)
        std::cout << "Volume is " << p->volume() << std::endl;

    // Free the memory
    for (auto p : pVessels)
        delete p;
}
```

The complete program is in the code download as `Ex14_09`. It produces the following output:

```
Volume is 24000
Volume is 235.619
Volume is 22722.4
Volume is 20400
Vessel destructor
Vessel destructor
Vessel destructor
Vessel destructor
```

Clearly we have a failure on our hands: the wrong destructors are being called in each case. The problem occurs because the binding to the destructor is being set at compile time and because the `delete` operator is being applied to an object pointed to by a pointer of type `Vessel*`, the `Vessel` class destructor is called every time. This is not a problem for the classes here because the classes do not contain members that are raw pointers to objects in the free store. If they did, it would represent a memory leak, because free store memory allocated for members of derived class objects would not be released. To get the correct destructor called for a derived class, we need dynamic binding for the destructors.

Virtual Destructors

To ensure that the correct destructor is always called for objects of derived classes that are allocated in the free store, you need *virtual class destructors*. To implement a virtual destructor in a derived class, you just add the keyword `virtual` to the destructor declaration in the base class. This signals to the compiler that destructor calls through a pointer or a reference parameter should have dynamic binding, and so the destructor that is called will be selected at runtime. This makes the destructor in every class derived from the base class virtual, in spite of the derived class destructors having different names; destructors are treated as a special case for this purpose.

You can show this effect by adding the `virtual` keyword to the destructor declaration in the `Vessel` class:

```
class Vessel
{
public:
    virtual double volume() const = 0;

    virtual ~Vessel() { std::cout << "Vessel destructor" << std::endl; }
};
```

The destructors of all the derived classes will automatically be virtual as a result of declaring a virtual base class destructor. If you run the example again, the output will confirm that this is so.

■ **Tip** When you are using inheritance and a class that has a destructor defined contains at least one virtual function, you should declare the base class destructor as `virtual`. A small overhead in the execution of the class destructors does exist, but you won't notice it in the majority of circumstances. Using virtual destructors ensures that your objects are always properly destroyed and avoids problems that might otherwise occur.

Summary

In this chapter, I've covered the principal ideas involved in using inheritance. The fundamentals that you should keep in mind are these:

- Polymorphism involves calling a function member of a class through a pointer or a reference and having the call resolved dynamically — that is, the function to be called is determined by what is referenced or pointed to when the program is executing.
- You can call function members polymorphically using raw pointers or smart pointers.
- A function in a base class can be declared as `virtual`. All occurrences of the function in classes that are derived from the base will then be virtual too. When you call a virtual function through a pointer or a reference, the function call is resolved dynamically and the type of object for which the function call is made will determine the particular function that is used.
- A call of a virtual function using an object and the direct member selection operator is resolved statically — that is, at compile time.
- If a base class contains a virtual function and any derived classes need to define a destructor, then you should always declare the base class destructor as `virtual`. This will ensure correct selection of a destructor for dynamically created derived class objects.

- A pure virtual function has no definition. A virtual function in a base class can be specified as pure by placing `=0` at the end of the function member declaration.
- A class with one or more pure virtual functions is called an abstract class, for which no objects can be created. In any class derived from an abstract class, all the inherited pure virtual functions must be defined. If they're not, it too becomes an abstract class, and no objects of the class can be created.
- You should use the `override` qualifier with function members of a derived class that override a virtual base class member. This causes the compiler to verify that the functions signatures in the base and derived classes are the same.
- Default argument values for parameters in virtual functions are assigned statically, so if default values for a base version of a virtual function exist, default values specified in a derived class will be ignored for dynamically resolved function calls.

EXERCISES

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck after that, you can download the solutions from the Apress website (www.apress.com/source-code), but that really should be a last resort.

Exercise 14-1. Define a base class called `Animal` that contains two protected data members: a `string` member to store the name of the animal (e.g., "Fido"), and an integer member, `weight`, that will contain the weight of the `Animal` in pounds. Also include a virtual public function member, `who()`, that returns a `string` object containing the name and weight of the `Animal` object, and a pure virtual function called `sound()` that in a derived class should return a `string` representing the sound the animal makes. Derive at least three classes — `Sheep`, `Dog`, and `Cow` — with the class `Animal` as a public base, and implement the `sound()` function appropriately in each class.

Define a class called `Zoo` that can store the addresses of any number of `Animal` objects of various types in a vector container. Write a `main()` function to create a random sequence of an arbitrary number of objects of classes derived from `Animal` and store pointers to them in a `Zoo` object. The number of objects should be entered from the keyboard. Define and use function members of the `Zoo` class, one to output information about each animal in the `Zoo`, and the other to output as text the sound that it makes.

Exercise 14-2. Define a class called `BaseLength` that stores a length as an integral number of millimeters, and which has a member function `length()` that returns a `double` value specifying the length. Derive classes called `Inches`, `Meters`, `Yards`, and `Perches` from the `BaseLength` class that override the base class `length()` function to return the length as a `double` value in the appropriate units. (1 inch is 25.4 millimeters; 1 meter is 1000 millimeters; 1 yard is 36 inches; 1 perch [US] is 5.5 yards.). Define a `main()` function to read a series of lengths in various units and create the appropriate derived class objects, storing their addresses in a vector of pointers of an appropriate type. Output each of the lengths in millimeters as well as the original units.

Exercise 14-3. Define conversion operator functions to convert each of the derived types in the previous example to any other derived type. For example, in the `Inches` class, define members `operator Meters()`, `operator Perches()`, and `operator Yards()`. Extend `main()` from the previous exercise to output each measurement in the four different units.

Exercise 14-4. Repeat the previous exercise using constructors for the conversions instead of conversion operators.



Runtime Errors and Exceptions

EXCEPTIONS are used to signal errors or unexpected conditions in a program. Using exceptions to signal errors is not mandatory, and you'll sometimes find it more convenient to handle them in other ways. However, it is important to understand how exceptions work, because they can arise out of the use of standard language features such as the new operator and the `dynamic_cast` operator and exceptions are used extensively within the standard library.

In this chapter you'll learn:

- What an exception is and when you should use exceptions
- How you use exceptions to signal error conditions
- How you handle exceptions in your code
- The types of exceptions defined in the Standard Library
- How to deal with exceptions that are thrown in a constructor
- How an exception being thrown can affect a destructor

Handling Errors

Error handling is a fundamental element of successful programming. You need to equip your program to deal with potential errors and abnormal events, and this can often require more effort than writing the code that executes when things work the way they should. The quality of the error-handling code determines how robust a program is and it is usually a major factor in making a program user-friendly. It also has a substantial impact on how easy it is to correct errors in the code or to add functionality to an application.

Not all errors are equal though and the nature of the error determines how best to deal with it. You don't use exceptions for errors that occur in the normal use of a program. In many cases, you'll deal with such errors directly where they occur. For example, when data is entered by the user, mistakes can result in erroneous input but this isn't really a serious problem. It's usually quite easy to detect such errors and the most appropriate course of action is often simply to discard the input and prompt the user to enter the data again. In this case, the error-handling code is integrated with the code that handles the overall input process.

More serious errors are often recoverable and can be dealt with in a manner that doesn't prejudice other activity within a program. When an error is discovered within a function, it's often convenient to return an error code of some kind to tell the caller about the error so that the caller can decide how best to proceed.

Exceptions provide you with an additional approach to handling errors. They don't replace the kinds of mechanisms I have just described. Exceptions are to deal with error conditions that you don't expect to occur in the normal course of events. A primary advantage of using exceptions to signal these errors is that the error-handling code is separated completely from the code that caused the error. Of course, the examples in this chapter will be atypical in that they are designed to cause an exception to show how things work. This means that they will not reflect the normal context for using exceptions.

Understanding Exceptions

An *exception* is a temporary object, of any type, that is used to signal an error. An exception can be of a fundamental type, such as `int` or `const char*`, but it's usually and more appropriately an object of a class type. The purpose of an exception object is to carry information from the point at which the error occurred to the code that is to handle the error. In many situations more than one piece of information is involved so this is best done with an object of a class type.

When you recognize that something has gone wrong in the code, you can signal the error by *throwing* an exception. The term “throwing” effectively indicates what happens. The exception object is tossed to another block of code that *catches* the exception and deals with it. Code that may throw exceptions must be within a special block called a *try* block if an exception is to be caught. If a statement that is not within a *try* block throws an exception, or a statement within a *try* block throws an exception that is not caught, the program terminates. I'll discuss this further a little later in this chapter.

A *try* block is followed immediately by one or more *catch* blocks. Each *catch* block contains code to handle a particular kind of exception; for this reason, a *catch* block is sometimes referred to as a *handler*. All the code that deals with errors that cause exceptions to be thrown is within *catch* blocks that are completely separate from the code that is executed when everything works as it should.

Figure 15-1 shows a *try* block, which is a normal block between braces that is preceded by the *try* keyword. Each time the *try* block executes, it may throw any one of several different types of exception. Therefore, a *try* block can be followed by several *catch* blocks, each of which handles an exception of a different type. A *catch* block is a normal block between braces preceded by the *catch* keyword. The type of exception that a *catch* block deals with is identified by a single parameter between parentheses following the *catch* keyword.

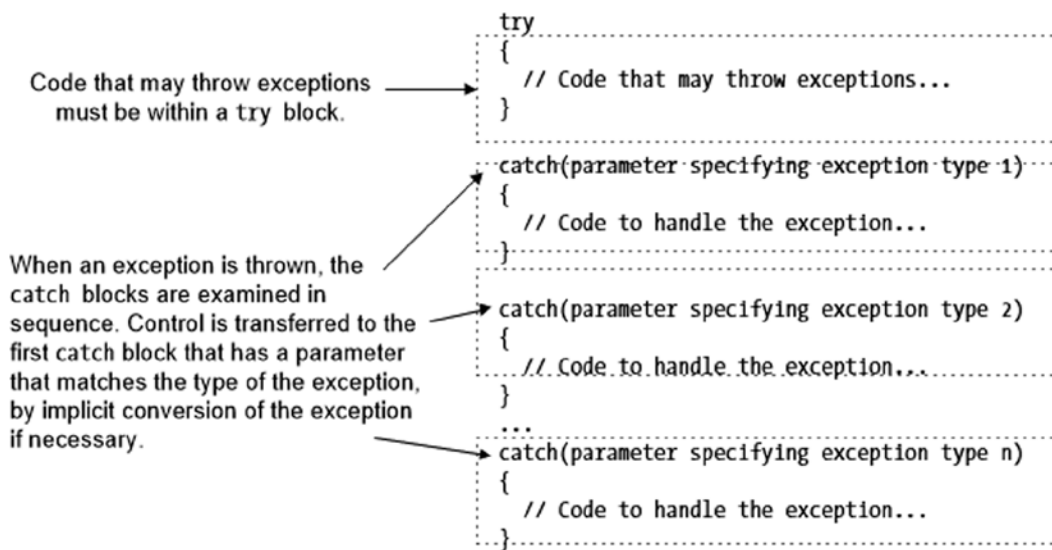


Figure 15-1. A *try* block and its *catch* blocks

The code in a *catch* block only executes when an exception of a matching type is thrown. If a *try* block doesn't throw an exception, then none of the *catch* blocks following the *try* block is executed. You can't branch into a *try* block, by using a *goto*, for instance. A *try* block always executes beginning with the first statement following the opening brace.

Throwing an Exception

It's high time you throw an exception to find out what happens when you do. Although you should always use class objects for exceptions (as you'll do later in the chapter), I'll begin by using basic types, because this will keep the code very simple while I explain what's going on. You throw an exception using a *throw expression*, which you write using the `throw` keyword. Here's an example:

```
try
{
    // Code that may throw exceptions must be in a try block...

    if(test > 5)
        throw "test is greater than 5";          // Throws an exception of type const char*

    // This code only executes if the exception is not thrown...
}
catch(const char* message)
{
    // Code to handle the exception...
    // ...which executes if an exception of type 'char*' or 'const char*' is thrown
    std::cout << message << std::endl;
}
```

If the value of `test` is greater than 5, the `throw` statement throws an exception. In this case, the exception is the literal, "test is greater than 5". Control is immediately transferred out of the `try` block to the first handler for the type of the exception that was thrown: `const char*`. There's just one handler here, which happens to catch exceptions of type `const char*`, so the statement in the `catch` block executes, and this displays the exception.

■ **Note** The compiler ignores the `const` keyword when matching the type of an exception that was thrown with the `catch` parameter type. I'll examine this more thoroughly later.

Let's try exceptions out in a working example which will throw exceptions of type `int` and `const char*`. The output statements help you see the flow of control:

```
// Ex15_01.cpp
// Throwing and catching exceptions
#include <iostream>

int main()
{
    for(size_t i {} ; i < 7 ; ++i)
    {
        try
        {
            if(i < 3)
                throw i;

            std::cout << " i not thrown - value is " << i << std::endl;
```



```

        if(i > 5)
            throw "Here is another!";

        std::cout << " End of the try block." << std::endl;
    }
    catch(size_t i)
    { // Catch exceptions of type size_t
        std::cout << " i caught - value is " << i << std::endl;
    }
    catch(const char* message)
    { // Catch exceptions of type char*
        std::cout << " \"" << message << "\" caught" << std::endl;
    }
    std::cout << "End of the for loop body (after the catch blocks) - i is "
        << i << std::endl;
}
}

```

This example produces the following output:

```

i caught - value is 0
End of the for loop body (after the catch blocks) - i is 0
i caught - value is 1
End of the for loop body (after the catch blocks) - i is 1
i caught - value is 2
End of the for loop body (after the catch blocks) - i is 2
i not thrown - value is 3
End of the try block.
End of the for loop body (after the catch blocks) - i is 3
i not thrown - value is 4
End of the try block.
End of the for loop body (after the catch blocks) - i is 4
i not thrown - value is 5
End of the try block.
End of the for loop body (after the catch blocks) - i is 5
i not thrown - value is 6
"Here is another!" caught
End of the for loop body (after the catch blocks) - i is 6

```

The try block within the for loop contains code that will throw an exception of type `size_t` if `i` (the loop counter) is less than 3, and an exception of type `const char*` if `i` is greater than 5. Throwing an exception transfers control out of the try block immediately so the output statement at the end of the try block only executes if no exception is thrown. The output shows that this is the case. You only get output from the last statement when `i` has the value 3, 4, or 5. For all other values of `i`, an exception is thrown, so the output statement is not executed.

The first catch block immediately follows the try block. All the exception handlers for a try block must immediately follow the try block. If you place any code between the try block and the first catch block, or between successive catch blocks, the program won't compile. The first catch block handles exceptions of type `size_t`, and you can see from the output that it executes when the first throw statement is executed. You can also see that the next catch block is not executed in this case. After this handler executes, control passes directly to the last statement at the end of the loop.

The second handler deals with exceptions of type `char*`. When the exception “Here is another!” is thrown, control passes from the `throw` statement directly to this handler, skipping the previous catch block. If no exception is thrown, neither the catch block is executed. You could put this catch block before the previous handler, and the program would work just as well. On this occasion, the sequence of the handlers doesn’t matter, but that’s not always the case. You’ll see examples of when the order of the handlers is important later in this chapter.

The statement that identifies the end of a loop iteration in the output is executed whether or not a handler is executed. Throwing an exception that is caught doesn’t end the program — unless you want it to of course — in which case you terminate the program in the catch block. If the problem that caused the exception can be fixed within the handler, then the program can continue.

The Exception Handling Process

From the example, you should have a fairly clear idea of the sequence of events when an exception is thrown. Some other things happen in the background though; you might be able to guess some of them if you think about how control is transferred from the `try` block to the catch block. The throw/catch sequence of events is illustrated conceptually in Figure 15-2.

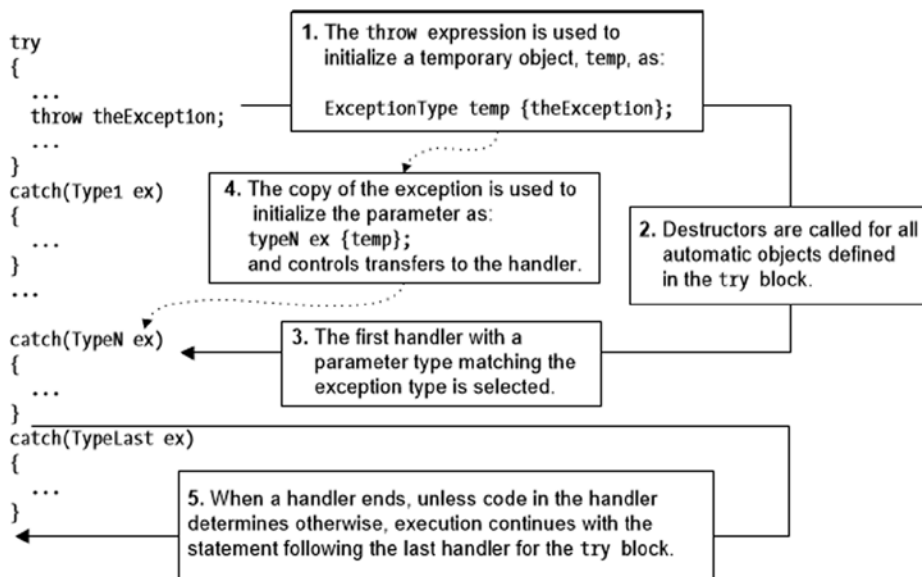


Figure 15-2. The mechanism behind throwing and catching an exception

Of course a `try` block is a statement block, and you know that a statement block defines a scope. Throwing an exception leaves the `try` block immediately, so at that point all the automatic objects that have been defined within the `try` block prior to the exception being thrown are destroyed. The fact that none of the automatic objects created in the `try` block exists by the time the handler code is executed is most important — it implies that you must not throw an exception object that’s a pointer to an object that is local to the `try` block. It’s also the reason why the exception object is copied in the throw process.

■ **Caution** An exception object must be of a type that can be copied. An object of a class type that has a private copy constructor can’t be used as an exception.

Because the `throw` expression is used to initialize a temporary object — and therefore creates a copy of the exception — you can throw *objects* that are local to the `try` block, but not *pointers* to local objects. The copy of the object is used to initialize the parameter for the `catch` block that is selected to handle the exception.

A `catch` block is also a statement block, so when a `catch` block has finished executing, all automatic objects that are local to it (including the parameter) will be destroyed. Unless you transfer control out of the `catch` block using a `goto` or a `return` statement, execution continues with the statement immediately following the last `catch` block for the `try` block. Once a handler has been selected for an exception and control has been passed to it, the exception is considered handled. This is true even if the `catch` block is empty and does nothing.

Unhandled Exceptions

If an exception is thrown in a `try` block and is not handled by any of its `catch` blocks, then (subject to the possibility of nested `try` blocks, which I'll discuss shortly) the Standard Library function `std::terminate()` is called. This function is declared in the exception header and calls a predefined *default terminate handler function*, which in turn calls the Standard Library function `std::abort()` that is declared in the `cstdlib` header. The sequence of events for an uncaught exception is shown in Figure 15-3.

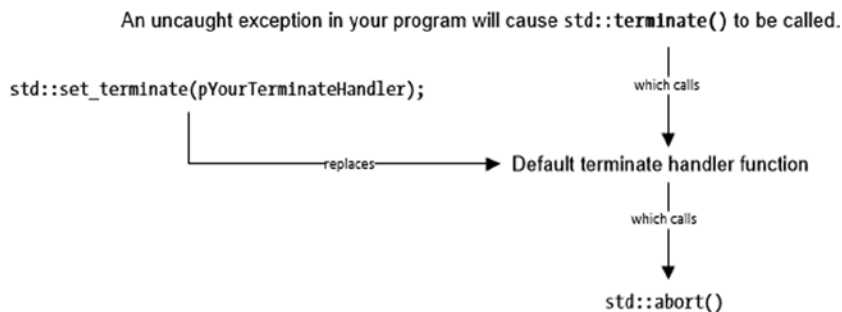


Figure 15-3. Uncaught exceptions

■ **Note** The `std::abort()` function terminates the entire program immediately; it doesn't call destructors for any automatic or static objects. `std::exit()` also terminates a program but does carry out cleanup operations, including calling destructors. When you want to terminate a program, it's better to call `std::exit()`.

The action provided by the default terminate handler can be disastrous in some situations. For example, it may leave files in an unsatisfactory state, or connection to a communications line may be left open. In such cases, you'd want to make sure that things are tidied up properly before the program ends. You can do this by replacing the default terminate handler function with your own version by calling the Standard Library function `std::set_terminate()`, as illustrated in Figure 15-3. The `set_terminate()` function accepts an argument of type `terminate_handler`, and returns a value of the same type. This type is defined in the exception header file as:

```
typedef void (*terminate_handler)();
```

`terminate_handler` is a pointer to a function that has no parameters and doesn't return a value, so your replacement function must be of this form. You can do what you want within your version of the terminate handler, but it must not return; it must ultimately terminate the program. Your definition of the function could take this form:

```
void myHandler()
{
    // Do necessary clean-up to leave things in an orderly state...
    std::exit(1);
}
```

Calling `std::exit()` is a more satisfactory way of terminating a program than calling `std::abort()`. Calling `exit()` ensures that destructors for global objects are called, and any open input/output streams are flushed if necessary and closed. Any temporary files that were created using standard library functions will be deleted. The integer argument that you pass to `exit()` is returned to the operating system as a status code. A non-zero value indicates abnormal program termination.

To set up the `myHandler()` function as the terminate handler, you could write this:

```
terminate_handler pOldHandler { std::set_terminate(myHandler)};
```

The return value is a pointer to the previous handler that was set, so by saving it, you'll be able to restore it later if necessary. The first time you call `set_terminate()`, the return value will be a pointer to the default handler. Each subsequent call to set a new handler will return a pointer to whatever handler is in effect. This means that you can have your handler in effect for a particular part of your program, and then restore the default handler when your handler no longer applies.

Of course, you can set different terminate handlers at various points in your program to provide shutdown actions that suit the particular conditions that apply at any given time. For example, if your program involves database operations, you might need to make sure that the database shuts down in an orderly fashion when a fatal error occurs. You would define a terminate handler to take care of this. Different terminate handlers accommodating different shutdown requirements can be used whenever you need them. You can obtain a pointer to the current terminate handler by calling `std::get_terminate()`.

Code That Causes an Exception to Be Thrown

I said at the beginning of this discussion that `try` blocks enclose code that may throw an exception. However, this doesn't mean that the code that throws an exception must be physically between the braces bounding the `try` block. It only needs to be logically within the `try` block. If a function is called within a `try` block, any exception that is thrown and not caught within that function can be caught by one of `catch` blocks for the `try` block. An example of this is illustrated in Figure 15-4. Two function calls are shown within the `try` block: `fun1()` and `fun2()`. Exceptions of type `ExceptionType` that are thrown within either function can be caught by the `catch` block following the `try` block. An exception that is thrown but not caught within a function may be passed on to the calling function the next level up. If it isn't caught there, it can be passed on up to the next level; this is illustrated in Figure 15-4 by the exception thrown in `fun3()` when it is called by `fun1()`. There's no `try` block in `fun1()` so exceptions thrown by `fun3()` will be passed to the function that called `fun1()`. If an exception reaches a level where no further `catch` handlers exist and it is still uncaught, then the terminate handler is called to end the program.

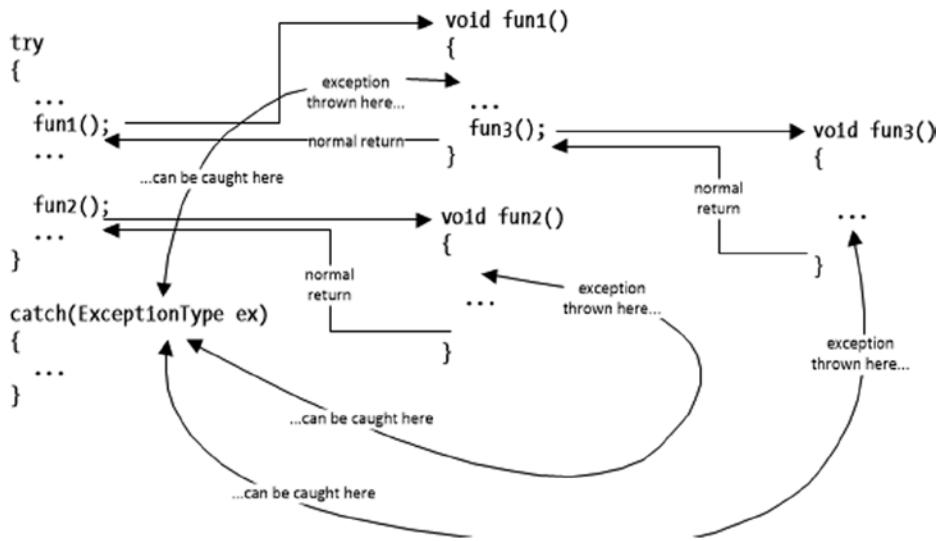


Figure 15-4. Exception thrown by functions called within a try block

Of course, if the same function is called from different points in a program, the exceptions that the code in the body of the function may throw can be handled by different catch blocks at different times. You can see an example of this situation in Figure 15-5.

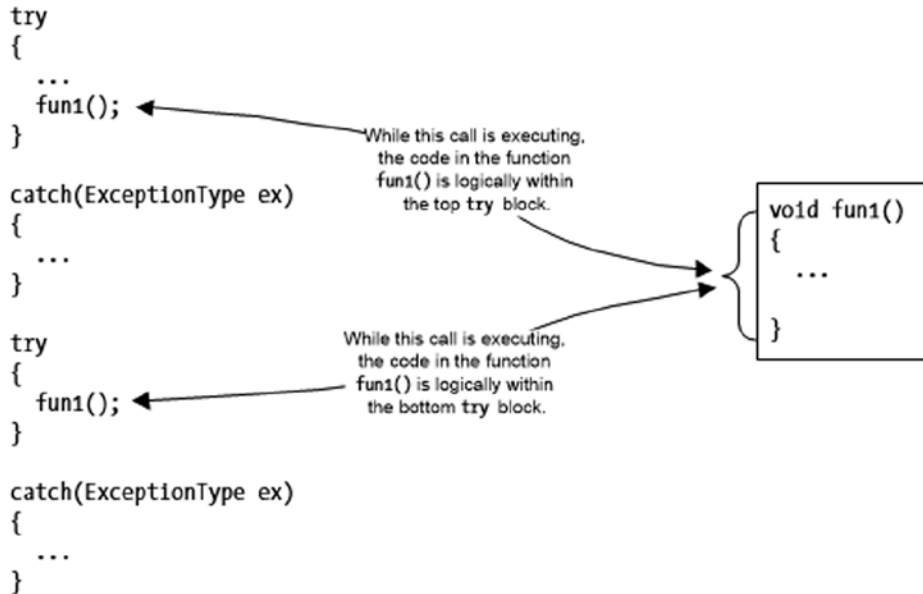


Figure 15-5. Calling the same function from within different try blocks

As a consequence of the call in the first try block, the catch block for that try block handles any exceptions of type `ExceptionType` thrown by `fun1()`. When `fun1()` is called in the second try block, the catch handler for that try block deals with any exception of type `ExceptionType` that is thrown. From this you should be able to see that you can choose to handle exceptions at the level that is most convenient to your program structure and operation. In an extreme case, you could catch all the exceptions that arose anywhere in a program in `main()` just by enclosing the code in `main()` in a try block and appending a suitable variety of catch blocks.

Nested try Blocks

You can nest a try block inside another try block. Each try block has its own set of catch blocks to handle exceptions that may be thrown within it, and the catch blocks for a try block are only invoked for exceptions thrown within that try block. This process is shown in Figure 15-6.

```
try
{
    // outer try block
    ...

    try
    {
        // inner try block
        ...
    }
    catch(ExceptionType1 ex) ← This handler catches ExceptionType1
                              exceptions thrown in the inner try block.
    {
        ...
    }
    ...
}
catch(ExceptionType2 ex) ← This handler catches ExceptionType2
                           exceptions thrown in the outer try block, as
                           well as uncaught exceptions of that type from
                           the inner try block.
{
    ...
}
```

Figure 15-6. *Nested try blocks*

Figure 15-6 shows one handler for each try block, but in general there may be several. The catch blocks catch exceptions of different types, but they could catch exceptions of the same type. When the code in an inner try block throws an exception, its handlers get the first chance to deal with it. Each handler for the try block is checked for a matching parameter type, and if none match, the handlers for the outer try block have a chance to catch the exception. You can nest try blocks in this way to whatever depth is appropriate for your application.

When an exception is thrown by the code in the outer try block, that block's catch handlers handle it, even if the statement originating the exception precedes the inner try block. The catch handlers for the inner try block can never be involved in dealing with exceptions thrown by code within the outer try block. The code within both try blocks may call functions, in which case the code within the body of the function is logically within the try block that called it. Any or all of the code within the body of the function could also be within its own try block, in which case this try block would be nested within the try block that called the function.

It all sounds rather complicated in words, but it's much easier in practice. I'll put together a simple example in which an exception is thrown and then see where it ends up. Once again, I'm going for explanation rather than gritty realism so I'll throw exceptions of type `int` and type `long` rather than objects of a class type. The code for this example demonstrates nested try blocks and throwing an exception within a function:

```

// Ex15_02.cpp
// Throwing exceptions in nested try blocks
#include <iostream>

void throwIt(int i)
{
    throw i;                      // Throws the parameter value
}

int main()
{
    for(int i {} ; i <= 5 ; ++i)
    {
        try
        {
            std::cout << "outer try:\n";
            if(i == 0)
                throw i;          // Throw int exception

            if(i == 1)
                throwIt(i);        // Call the function that throws int

            try
            {                      // Nested try block
                std::cout << " inner try:\n";
                if(i == 2)
                    throw static_cast<long>(i); // Throw long exception

                if(i == 3)
                    throwIt(i);    // Call the function that throws int
            }                      // End nested try block
            catch(int n)
            {
                std::cout << " Catch int for inner try. " << "Exception " << n << std::endl;
            }

            std::cout << "outer try:\n";
            if(i == 4)
                throw i;          // Throw int
            throwIt(i);            // Call the function that throws int
        }
        catch(int n)
        {
            std::cout << "Catch int for outer try. " << "Exception " << n << std::endl;
        }
        catch(long n)
        {
            std::cout << "Catch long for outer try. " << "Exception " << n << std::endl;
        }
    }
}

```

This produces the following output:

```

outer try:
Catch int for outer try. Exception 0
outer try:
Catch int for outer try. Exception 1
outer try:
inner try:
Catch long for outer try. Exception 2
outer try:
inner try:
Catch int for inner try. Exception 3
outer try:
Catch int for outer try. Exception 3
outer try:
inner try:
outer try:
Catch int for outer try. Exception 4
outer try:
inner try:
outer try:
Catch int for outer try. Exception 5

```

How It Works

The `throwIt()` function throws its parameter value. If you were to call this function from outside a try block, it would immediately cause the program to end, because the exception would go uncaught and the default terminate handler would be called. All the exceptions are thrown within the for loop. Within the loop, you determine when to throw an exception and what kind of exception to throw by testing the value of the loop variable, `i`, in successive if statements. At least one exception is thrown on each iteration. Entry to each try block is recorded in the output and because each exception has a unique value, you can easily see where each exception is thrown and caught.

The first exception is thrown from the outer try block when the loop variable, `i`, is 0. You can see from the output that the catch block for exceptions of type `int` that follows the outer try block catches this exception. The catch block for the inner try block has no relevance because it can only catch exceptions thrown in the inner try block.

The next exception is thrown in the outer try block when `i` is 1 as a result of calling `throwIt()`. This is also caught by the catch block for `int` exceptions that follows the outer try block. The next two exceptions, however, are thrown in the inner try block. The first is an exception of type `long`. No catch block for the inner try block for this type of exception exists, so it propagates to the outer try block. Here, the catch block for type `long` handles it, as you can see from the output. The second exception is of type `int` and is thrown in the body of the `throwIt()` function. No try block exists in this function, so the exception propagates to the point where the function was called in the inner try block. The exception is then caught by the catch block for exceptions of type `int` that follows the inner try block.

When a handler for the inner try block catches an exception, execution continues with the remainder of the outer try block. Thus, when `i` is 3, you get output from the catch block for the inner try block, plus output from the handler for `int` exceptions for the outer try block. The latter exception is thrown as a result of the `throwIt()` function call at the end of the inner try block. Finally, two more exceptions are thrown in the outer try block. The handler for `int` exceptions for the outer try block catches both. The second exception is thrown within the body of `throwIt()` and because it is called in the outer try block, the catch block following the outer try block handles it.

Although none of these was a realistic exception — exceptions in real programs are invariably class objects — they did show the mechanics of throwing and catching exceptions and what happens with nested try blocks. Let's move on to take a closer look at exceptions that are objects.

Class Objects as Exceptions

You can throw any type of class object as an exception. However, keep in mind that the idea of an exception object is to communicate information to the handler about what went wrong. Therefore it's usually appropriate to define a specific exception class that is designed to represent a particular kind of problem. This is likely to be application-specific, but your exception class objects almost invariably contain a message of some kind explaining the problem, and possibly some sort of error code. You can also arrange for an exception object to provide additional information about the source of the error in whatever form is appropriate.

Let's define a simple exception class. I'll put it in a header file with a fairly generic name, `MyTroubles.h`, because I'll be adding to this file later:

```
// MyTroubles.h Exception class definition
#ifndef MYTROUBLES_H
#define MYTROUBLES_H
#include <string>
using std::string;

class Trouble
{
private:
    string message;
public:
    Trouble(string str = "There's a problem") : message {str} {}
    string what() const {return message;}
};
#endif
```

This class just defines an object representing an exception that stores a message indicating a problem. A default message is defined in the parameter list for the constructor, so you can use the default constructor to get an object that contains the default message. The `what()` function member returns the current message. To keep the logic of exception handling manageable, you functions need to ensure that function members of an exception class don't throw exceptions. Later in this chapter, you'll see how you can explicitly prevent a member function from doing so.

Let's find out what happens when a class object is thrown by throwing a few. As in the previous examples, I won't bother to create errors. I'll just throw exception objects so that you can follow what happens to them under various circumstances. I'll exercise the exception class with a very simple example that throws some exception objects in a loop:

```
// Ex15_03.cpp
// Throw an exception object
#include <iostream>
#include "MyTroubles.h"

int main()
{
    for(int i {}; i < 2 ; ++i)
    {
        try
        {
            if(i == 0)
                throw Trouble {};
            else
                throw Trouble {"Nobody knows the trouble I've seen..."};
        }
    }
```

```

    catch(const Trouble& t)
    {
        std::cout << "Exception: " << t.what() << std::endl;
    }
}

```

This produces the following output:

```

Exception: There's a problem
Exception: Nobody knows the trouble I've seen...

```

Two exception objects are thrown in the for loop. The first is created by the default constructor for the `Trouble` class, and therefore contains the default message string. The second exception object is thrown in the else clause of the if statement and contains a message that is passed as the argument to the constructor. The catch block catches both exception objects.

The parameter for the catch block is a reference. Remember that an exception object is always copied when it is thrown, so if you don't specify the parameter for a catch block as a reference, it'll be copied a second time — quite unnecessarily. The sequence of events when an exception object is thrown is that first the object is copied to create a temporary object and the original is destroyed because the try block is exited and the object goes out of scope. The copy is passed to the catch handler — by reference if the parameter is a reference. If you want to observe these events taking place, just add a copy constructor and a destructor containing some output statements to the `Trouble` class.

Matching a Catch Handler to an Exception

I said earlier that the handlers following a try block are examined in the sequence in which they appear in the code, and the first handler whose parameter type matches the type of the exception will be executed. With exceptions that are basic types (rather than class types), an exact type match with the parameter in the catch block is necessary. With exceptions that are class objects, implicit conversions may be applied to match the exception type with the parameter type of a handler. When the parameter type is being matched to the type of the exception that was thrown, the following are considered to be a match:

- The parameter type is the same as the exception type, ignoring `const`.
- The type of the parameter is a direct or indirect base class of the exception class type, or a reference to a direct or indirect base class of the exception class, ignoring `const`.
- The exception and the parameter are pointers, and the exception type can be converted implicitly to the parameter type, ignoring `const`.

The possible type conversions listed here have implications for how you sequence the catch blocks for a try block. If you have several handlers for exception types within the same class hierarchy, then the most derived class type must appear first and the most base class type last. If a handler for a base type appears before a handler for a type derived from that base, then the base type is always selected to handle the derived class exceptions. In other words, the handler for the derived type is never executed.

Let's add a couple more exception classes to the header containing the `Trouble` class, and use `Trouble` as a base class for them. Here's how the contents of the header file `MyTroubles.h` will look with the extra classes defined:

```
// MyTroubles.h Exception classes
#ifndef MYTROUBLES_H
#define MYTROUBLES_H
#include <string>
using std::string;

class Trouble
{
private:
    string message;
public:
    Trouble(string str = "There's a problem") : message {str} {}

    virtual ~Trouble(){} // Virtual destructor
    virtual string what() const { return message; }
};

// Derived exception class
class MoreTrouble : public Trouble
{
public:
    MoreTrouble(string str = "There's more trouble...") : Trouble {str} {}
};

// Derived exception class
class BigTrouble : public MoreTrouble
{
public:
    BigTrouble(string str = "Really big trouble...") : MoreTrouble {str} {}
};

#endif
```

Note that the `what()` member and the destructor of the base class have been declared as `virtual`. Therefore, the `what()` function is also `virtual` in the classes derived from `Trouble`. It doesn't make much of a difference here, but remembering to declare a `virtual` destructor in a base class is a good habit to get into. Other than different default strings for the message, the derived classes don't add anything to the base class. Often, just having a different class name can differentiate one kind of problem from another. You just throw an exception of a particular type when that kind of problem arises; the internals of the classes don't have to be different. Using a different `catch` block to catch each class type provides the means to distinguish different problems. Here's the code to throw exceptions of the `Trouble`, `MoreTrouble`, and `BigTrouble` types, and the handlers to catch them:

```
// Ex15_04.cpp
// Throwing and catching objects in a hierarchy
#include <iostream>
#include "MyTroubles.h"

int main()
{
    Trouble trouble;
    MoreTrouble moreTrouble;
    BigTrouble bigTrouble;
```

```

for (int i {}; i < 7; ++i)
{
    try
    {
        if (i == 3)
            throw trouble;
        else if (i == 5)
            throw moreTrouble;
        else if (i == 6)
            throw bigTrouble;
    }
    catch (const BigTrouble& t)
    {
        std::cout << "BigTrouble object caught: " << t.what() << std::endl;
    }
    catch (const MoreTrouble& t)
    {
        std::cout << "MoreTrouble object caught: " << t.what() << std::endl;
    }
    catch (const Trouble& t)
    {
        std::cout << "Trouble object caught: " << t.what() << std::endl;
    }
    std::cout << "End of the for loop (after the catch blocks) - i is " << i << std::endl;
}
}

```

Here's the output:

```

End of the for loop (after the catch blocks) - i is 0
End of the for loop (after the catch blocks) - i is 1
End of the for loop (after the catch blocks) - i is 2
Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 3
End of the for loop (after the catch blocks) - i is 4
MoreTrouble object caught: There's more trouble...
End of the for loop (after the catch blocks) - i is 5
BigTrouble object caught: Really big trouble...
End of the for loop (after the catch blocks) - i is 6

```

How It Works

After creating one object of each class type, the for loop throws one or other of the exceptions for selected values of the loop variable, *i*. Each of the catch blocks contains a different message so the output shows which catch handler is selected when an exception is thrown. In the handlers for the two derived types, the inherited `what()` function still returns the message. Note that the parameter type for each of the catch blocks is a reference, as in the previous example. One reason for using a reference is to avoid making another copy of the exception object. In the next example, you'll see another good reason why you should always use a reference parameter in a handler.

Each handler displays the message contained in the object thrown, and you can see from the output that each handler is called to correspond with the type of the exception thrown. The ordering of the handlers is important because of the way the exception is matched to a handler, and because the types of your exception classes are related. Let's explore that in a little more depth.

Catching Derived Class Exceptions with a Base Class Handler

Exceptions of derived class types are implicitly converted to a base class type for the purpose of matching a catch block parameter so you could catch all the exceptions thrown in the previous example with a single handler. You can modify the previous example to see this happening. Just delete or comment out the two derived class handlers from `main()` in the previous example:

```
// Ex15_05.cpp
// Catching exceptions with a base class handler
#include <iostream>
#include "MyTroubles.h"

int main()
{
    Trouble trouble;
    MoreTrouble moreTrouble;
    BigTrouble bigTrouble;

    for (int i {}; i < 7; ++i)
    {
        try
        {
            if (i == 3)
                throw trouble;
            else if (i == 5)
                throw moreTrouble;
            else if (i == 6)
                throw bigTrouble;
        }
        catch (const Trouble& t)
        {
            std::cout << "Trouble object caught: " << t.what() << std::endl;
        }
        std::cout << "End of the for loop (after the catch blocks) - i is " << i << std::endl;
    }
}
```

The program now produces this output:

```
End of the for loop (after the catch blocks) - i is 0
End of the for loop (after the catch blocks) - i is 1
End of the for loop (after the catch blocks) - i is 2
Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 3
End of the for loop (after the catch blocks) - i is 4
Trouble object caught: There's more trouble...
End of the for loop (after the catch blocks) - i is 5
Trouble object caught: Really big trouble...
End of the for loop (after the catch blocks) - i is 6
```

The catch block with the parameter of type `const Trouble&` now catches all the exceptions thrown in the `try` block. If the parameter in a catch block is a reference to a base class, then it matches any derived class exception. So, although the output proclaims “Trouble object caught” for each exception, the output for exceptions that are caught after they first correspond to objects of classes derived from `Trouble`.

The dynamic type is retained when the exception is passed by reference so you could also obtain the dynamic type and display it using the `typeid()` operator. Just modify the code for the handler to:

```
catch(const Trouble& t)
{
    std::cout << typeid(t).name() << " object caught: " << t.what() << std::endl;
}
```

Some compilers don’t enable runtime type identification by default so if this doesn’t work, check for a compiler option to switch it on. With this modification to the code, the output shows that the derived class exceptions still retain their dynamic types, even though a reference to the base class is being used. Remember, the `typeid()` operator returns an object of the `type_info` class and calling its `name()` member returns the class name as `type const char*`. For the record, the output from this version of the program should look like this:

```
End of the for loop (after the catch blocks) - i is 0
End of the for loop (after the catch blocks) - i is 1
End of the for loop (after the catch blocks) - i is 2
class Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 3
End of the for loop (after the catch blocks) - i is 4
class MoreTrouble object caught: There's more trouble...
End of the for loop (after the catch blocks) - i is 5
class BigTrouble object caught: Really big trouble...
End of the for loop (after the catch blocks) - i is 6
```

Try changing the parameter type for the handler to `Trouble` so that the exception is passed by value rather than by reference:

```
catch(Trouble t)
{
    cout << typeid(t).name() << " object caught: " << t.what() << endl;
}
```

This version of the program produces the output:

```
End of the for loop (after the catch blocks) - i is 0
End of the for loop (after the catch blocks) - i is 1
End of the for loop (after the catch blocks) - i is 2
class Trouble object caught: There's a problem
End of the for loop (after the catch blocks) - i is 3
End of the for loop (after the catch blocks) - i is 4
class Trouble object caught: There's more trouble...
End of the for loop (after the catch blocks) - i is 5
class Trouble object caught: Really big trouble...
End of the for loop (after the catch blocks) - i is 6
```

Here, the `Trouble` handler is still selected for the derived class objects, but the dynamic type is not preserved. This is because the parameter is initialized using the base class copy constructor, so any properties associated with the derived class are lost. Only the base class sub-object of the original derived class object is retained. This is an example of *object slicing*, which results because the base class copy constructor knows nothing about derived objects. Object slicing is a common source of error when passing objects by value. That leads to the inevitable conclusion that you should *always* use reference parameters in catch blocks.

Rethrowing Exceptions

When a handler catches an exception, it can *rethrow* it to allow a handler for an outer `try` block to catch it. You rethrow the current exception with a statement consisting of just the `throw` keyword:

```
throw;                                // Rethrow the exception
```

This rethrows the existing exception object without copying it. You might rethrow an exception if a handler that catches exceptions of more than one derived class type discovers that the type of the exception requires it to be passed on to another level of `try` block. You might also want to register the point in the program where an exception was thrown, and then rethrow it for handling in a caller function.

Note that rethrowing an exception from the inner `try` block doesn't make the exception available to other handlers for the inner `try` block. When a handler is executing, any exception that is thrown (including the current exception) needs to be caught by a handler for a `try` block that encloses the current handler, as illustrated in Figure 15-7. The fact that a rethrown exception is not copied is important, especially when the exception is a derived class object that initialized a base class reference parameter. I'll demonstrate this with an example.

```

try          // Outer try block
{
    ...
    try      // Inner try block
    {
        if( ... )
            throw ex;
        ...
    }
    catch(ExType& ex)
    {
        ...
        throw;
    }
    catch(AType& ex)
    {
        ...
    }
}
catch(ExType& ex)
{
    // Handle ex...
}

```

This handler catches the `ex` exception that is thrown in the inner `try` block.

This statement rethrows `ex` without copying it so that it can be caught by a handler for the outer `try` block.

This handler catches the `ex` exception that was rethrown in the inner `try` block.

Figure 15-7. Rethrowing an exception

This example throws `Trouble`, `MoreTrouble`, and `BigTrouble` exception objects and then rethrows some of them to show how the mechanism works:

```

// Ex15_06.cpp
// Rethrowing exceptions
#include <iostream>
#include "MyTroubles.h"

int main()
{
    Trouble trouble;
    MoreTrouble moreTrouble;
    BigTrouble bigTrouble;

    for (int i {}; i < 7; ++i)
    {
        try
        {
            try
            {
                if (i == 3)
                    throw trouble;
                else if (i == 5)
                    throw moreTrouble;
                else if (i == 6)
                    throw bigTrouble;
            }
        }
    }
}

```



```

    }
    catch (const Trouble& t)
    {
        if (typeid(t) == typeid(Trouble))
            std::cout << "Trouble object caught in inner block: " << t.what() << std::endl;
        else
            throw; // Rethrow current exception
    }
}
catch (const Trouble& t)
{
    std::cout << typeid(t).name() << " object caught in outer block: "
              << t.what() << std::endl;
}
std::cout << "End of the for loop (after the catch blocks) - i is " << i << std::endl;
}

```

This example displays the following output:

```

End of the for loop (after the catch blocks) - i is 0
End of the for loop (after the catch blocks) - i is 1
End of the for loop (after the catch blocks) - i is 2
Trouble object caught in inner block: There's a problem
End of the for loop (after the catch blocks) - i is 3
End of the for loop (after the catch blocks) - i is 4
class MoreTrouble object caught in outer block: There's more trouble...
End of the for loop (after the catch blocks) - i is 5
class BigTrouble object caught in outer block: Really big trouble...
End of the for loop (after the catch blocks) - i is 6

```

The for loop works as in the previous example, but this time there is one try block nested inside another. The same sequence of exception objects as the previous example objects are thrown in the inner try block and they are all caught by the matching catch block because the parameter is a reference to the base class, `Trouble`. The if statement in the catch block tests the class type of the object passed and executes the output statement if it is of type `Trouble`. For any other type of exception the exception is rethrown and therefore available to be caught by the catch block for the outer try block. The parameter is also a reference to `Trouble` so it catches all the derived class objects. The output shows that it catches the rethrown objects and they're still in pristine condition.

You might imagine that the throw statement in the handler for the inner try block is equivalent to the following statement:

```
throw t; // Rethrow current exception
```

After all, you're just rethrowing the exception, aren't you? The answer is no; there's a major difference. If you make this modification to the program code and run it again. You'll get this output:

```
End of the for loop (after the catch blocks) - i is 0
End of the for loop (after the catch blocks) - i is 1
End of the for loop (after the catch blocks) - i is 2
Trouble object caught in inner block: There's a problem
End of the for loop (after the catch blocks) - i is 3
End of the for loop (after the catch blocks) - i is 4
class Trouble object caught in outer block: There's more trouble...
End of the for loop (after the catch blocks) - i is 5
class Trouble object caught in outer block: Really big trouble...
End of the for loop (after the catch blocks) - i is 6
```

The statement with an explicit exception object specified is throwing the exception, not rethrowing it. This results in the exception being copied, using the copy constructor for the `Trouble` class. It's the object slicing problem again. The derived portion of each object is sliced off, so you are left with just the base class sub-object in each case. You can see from the output that the `typeid()` operator identifies all the exceptions as type `Trouble`.

Catching All Exceptions

You can use an ellipsis (...) as the parameter specification for a catch block to indicate that the block should handle any exception:

```
try
{
    // Code that may throw exceptions...
}
catch(...)
{
    // Code to handle any exception...
}
```

This catch block handles an exception of any type, so a handler like this must always be last in the sequence of handlers for a try block. Of course, you have no idea what the exception is, but at least you can prevent your program from terminating because of an uncaught exception. Note that even though you don't know anything about it, you can still rethrow the exception as you did in the previous example.

You can modify the last example to catch all the exceptions for the inner try block by using an ellipsis in place of the parameter:

```
// Ex15_07.cpp
// Catching any exception
#include <iostream>
#include "MyTroubles.h"

int main()
{
    Trouble trouble;
    MoreTrouble moreTrouble;
    BigTrouble bigTrouble;
```

```

for (int i {}; i < 7; ++i)
{
    try
    {
        try
        {
            if (i == 3)
                throw trouble;
            else if (i == 5)
                throw moreTrouble;
            else if (i == 6)
                throw bigTrouble;
        }
        catch (...)                // Catch any exception
        {
            std::cout << "We caught something! Let's rethrow it. " << std::endl;
            throw;                  // Rethrow current exception
        }
    }
    catch (const Trouble& t)
    {
        std::cout << typeid(t).name() << " object caught in outer block: "
                  << t.what() << std::endl;
    }
    std::cout << "End of the for loop (after the catch blocks) - i is " << i << std::endl;
}
}

```

This produces the following output:

```

End of the for loop (after the catch blocks) - i is 0
End of the for loop (after the catch blocks) - i is 1
End of the for loop (after the catch blocks) - i is 2
We caught something! Let's rethrow it.
class Trouble object caught in outer block: There's a problem
End of the for loop (after the catch blocks) - i is 3
End of the for loop (after the catch blocks) - i is 4
We caught something! Let's rethrow it.
class MoreTrouble object caught in outer block: There's more trouble...
End of the for loop (after the catch blocks) - i is 5
We caught something! Let's rethrow it.
class BigTrouble object caught in outer block: Really big trouble...
End of the for loop (after the catch blocks) - i is 6

```

The catch block for the inner try block has an ellipsis as the parameter specification so any exception that is thrown will be caught by this catch block. Every time an exception is caught, a message displays and the exception is rethrown to be caught by the catch block for the outer try block. There, its type is properly identified and the string returned by its `what()` member is displayed.

Functions That Throw Exceptions

Any function can throw an exception, and this includes constructors. An exception thrown by a function can be caught in the calling function, as you saw in Ex15_02. For this to occur, the exception must be thrown or rethrown and not caught within the function. Of course, if you don't want a program to be terminated as a result of an exception being thrown, the exception must be caught somewhere, and for that to happen the call of a function that throws exceptions must be enclosed within a try block and a handler for the try block must catch the exception.

Of course, a function body can contain try blocks with catch blocks to handle exceptions. Any uncaught exceptions propagate to the point at which the function was called. It is sometimes convenient to make the whole body of a function a try block with its set of handlers; you can do this by using a *function try block*.

Function try Blocks

You define a function try block by putting the try keyword before the opening brace of the function body. You place the catch blocks for the function try block after the closing brace of the function body. Here's an example:

```
void doThat(int argument)
try
{
    // Code for the function...
}
catch(BigTrouble& ex)
{
    // Handler code for BigTrouble exceptions...
}
catch(MoreTrouble& ex)
{
    // Handler code for MoreTrouble exceptions...
}
catch(Trouble& ex)
{
    // Handler code for Trouble exceptions...
}
```

The entire body of the function is now a try block followed by its catch blocks. The catch blocks do not necessarily catch all the types of exception that may be thrown. Of course a function that throws exceptions doesn't need to have a function try block, or indeed any try block. However, any call of a function that throws exceptions should be enclosed by a try block at some level unless you want the uncaught exception to terminate execution of the program.

You need to take care that a catch block for a function try block executes an appropriate return statement if execution is not terminated. If the execution sequence in a catch block for a function try block runs to the end of the catch block, it is the equivalent of executing return. This will be fine if the function return type is void, but if the return type is other than void, the behavior is undefined.

Functions That Don't Throw Exceptions

You can specify that a function does not throw exceptions. You do this by appending the `noexcept` keyword to the function header. This does not mean that no exceptions are thrown within the function; it means that if an exception is thrown, it will be caught within the function and not rethrown. For example:

```
void doThat(int argument) noexcept
try
{
    // Code for the function...
}
catch( ... )
{
    // Handles all exceptions and does not rethrow...
}
```

This function handles any exceptions that may be thrown. If a function specified with `noexcept` *does* throw an exception that is not caught within the function, the exception will not be propagated to the calling function; `std::terminate()` will be called immediately.

■ **Note** Prior to the C++ 11 language standard, `throw()` could be appended to a function header to indicate that the function did not throw exceptions, and `throw(exception_type_list)` could be appended to identify a set of exception types that a function could throw. Both of these are now deprecated because they were not effective and could cause difficulties in practice so you should not use them.

Constructor try Blocks

As I indicated in the previous section, class constructors can throw exceptions. Of course, this can happen even without an explicit `throw` statement in the body of a constructor. It's kind of obvious really when you consider that Standard Library class functions can throw exceptions and these can be used in a constructor. Some member functions of `std::string` can throw exceptions for example and if a constructor allocates memory using `new`, this can result in an exception being thrown. If an exception *is* thrown and caught in a constructor, the catch block should rethrow the exception. This is because the object will not have been constructed and it is essential that the caller knows this. If execution reaches the end of a catch block for a constructor try block without rethrowing the exception, the original exception will be rethrown anyway.

If there is the potential for the initialization list for a constructor to throw an exception, you can make the constructor body, including the initialization list if there is one, a try block. You specify a constructor body that includes the initialization list as a try block by placing the `try` keyword immediately following the closing brace for the parameter list. Here's an example:

```
Example::Example(int count) try : BaseClass(count)
{
    // Could throw an exception...
}
catch(...) // Catch any exception
{
    // Handle the exception...
    rethrow;
}
```

The `try` keyword precedes the colon that appears before the initialization list so the initialization list is within the `try` block. Note that if there is an initialization list, you cannot place the `try` keyword between the initialization list and the opening brace for the constructor body. The constructor for the `Example` class calls a base class constructor, `BaseClass`, in the initialization list. If the base class constructor call or the code in the body of the `Example` constructor throws an exception, it will be caught by the `catch` block. The ellipsis specifies that any type of exception is to be caught but in general you can catch specific exception types and of course there can be multiple `catch` blocks. It's not essential that you rethrow the exception in the `catch` block. An exception thrown in a constructor `try` block will be rethrown in any event when the end of the code in a `catch` block is reached.

Exceptions and Destructors

Automatic objects that are in scope when an exception is thrown are destroyed so a class destructor may be called before the `catch` block that handles the exception is executed. Within a destructor, it can be important to know that the destructor was called because an exception was thrown rather than because the object was destroyed by going out of scope in the normal way. You can call the `std::uncaught_exception()` function that is declared in the `exception` header to detect when a destructor is called because an exception was thrown. The function returns `true` if an exception was thrown and the corresponding `catch` block hasn't been executed, so this allows a suitable course of action within your destructor to deal with this.

As a general rule, destructors shouldn't throw exceptions. Destructors are `noexcept` by default so any exception thrown within the destructor causes `std::terminate()` to be called, which ends the program immediately. If a destructor is called as a result of an exception being thrown and the destructor throws an exception, this prevents the `catch` block for the original exception from ever being reached, which could be disastrous. To avoid this situation you need to make sure that no exceptions are thrown from a destructor in order to allow the handler for the original exception to execute. Of course, to prevent exceptions from escaping beyond the bounds of a destructor, you can enclose the code in a `try` block and use a handler that catches any exception but this is rarely necessary.

Standard Library Exceptions

Several exception types are defined in the Standard Library. They're all derived from the `std::exception` class that is defined in the `exception` header and they are all defined within the `std` namespace. The derived exception class definitions are in the `stdexcept` header. None of the function members of the standard exception classes will throw an exception. For reference, the hierarchy for the standard exception classes is shown in Figure 15-8.

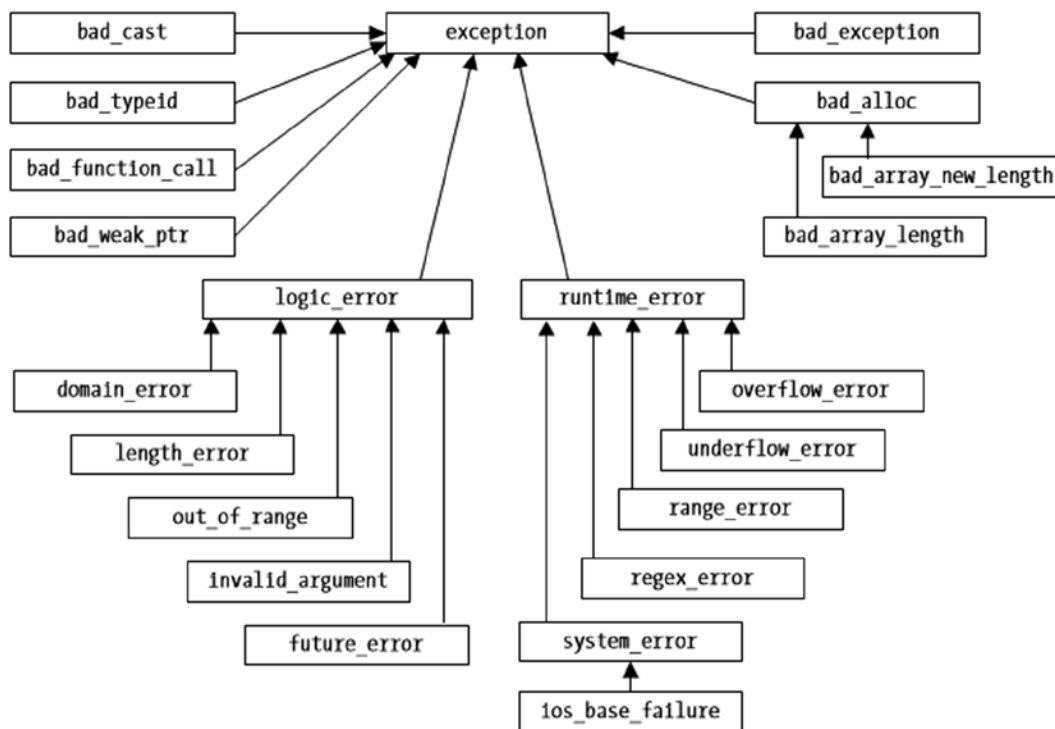


Figure 15-8. Standard exception class types

There are a lot of types in Figure 15-8 and I'm not going to grind through where they all originate. Your library documentation will identify when a function throws an exception but I'll mention a few of the standard exception types and explain a little about the thinking behind the hierarchy. The `bad_cast` exception that can be thrown by the `dynamic_cast<>()` operator and the `bad_alloc` exception can be thrown by the operator `new`. A `bad_typeid` exception is thrown if you use the `typeid()` operator with a null pointer. A `shared_ptr` constructor will throw a `bad_weak_ptr` exception if you attempt to create a `shared_ptr` object from a `weak_ptr` that has expired.

Most of the types of exceptions fall into two groups with each group identified by a base class that is derived from `exception`, either `logic_error` or `runtime_error`. For the most part, Standard Library functions do not throw `logic_error` or `runtime_error` objects directly, only objects of types derived from these, although `std::locale` class members can throw `runtime_error`. The types that have `logic_error` as a base are exceptions thrown for errors that could (at least in principle) have been detected before the program executed because they are caused by defects in the program logic. The other group, derived from `runtime_error`, is for errors that are generally data dependent and can only be detected at runtime. For instance, if you access characters in a `string` object using the `at()` member function and the index is outside the legal range for the object, an exception of type `out_of_range` is thrown; this is something you might have detected before calling `at()`. The `ios_base::failure` exception is thrown by functions in the standard library that support stream input-output. I'll discuss streams in depth in Chapter 17.

The Exception Class Definitions

You can usefully use a standard exception class as a base class for your own exception class. Since all the standard exception classes have `exception` as a base, it's a good idea to understand what members this class has because they are inherited by all the other exception classes. The exception class is defined in the exception header like this:

```
class exception
{
public:
    exception() noexcept;                // Default constructor
    exception(const exception&) noexcept; // Copy constructor
    exception& operator=(const exception&) noexcept; // Assignment operator
    virtual ~exception();                // Destructor
    virtual const char* what() const noexcept; // Return a message string
};
```

This is the public class interface specification and a particular implementation may have additional non-public members. This is true of the other standard exception classes too. The `noexcept` that appears in the declaration of the function members specifies that they do not throw exceptions, as I discussed earlier. The destructor is `noexcept` by default. Notice that there are no data members. The null-terminated string returned by `what()` is defined within the body of the function definition and is implementation dependent. This function is declared as `virtual` so it will be `virtual` in any class derived from `exception`. If you have a virtual function that can deliver a message that corresponds to each exception type, you can use it to provide a basic, economical way to record any exception that's thrown.

A catch block with a base class parameter matches any derived class exception type so you can catch any of the standard exceptions by using a parameter of type `exception&`. Of course, you can also use a parameter of type `logic_error&` or `runtime_error&` to catch any exceptions of types that are derived from these. You could provide the `main()` function with a function try block, plus a catch block for exceptions of type `exception`:

```
int main()
try
{
    // Code for main...
}
catch(exception& ex)
{
    std::cout << typeid(ex).name() << " caught in main: " << ex.what() << std::endl;
}
```

The catch block catches all exceptions that have `exception` as a base and outputs the exception type and the message returned by the `what()` function. Thus this simple mechanism gives you information about any exception that is thrown and not caught anywhere in a program. If your program uses exception classes that are not derived from `exception`, an additional catch block with ellipses in place of a parameter type catches all other exceptions, but in this case you'll have no access to the exception object and no information as to what it is. Making the body of `main()` a try block is a handy catch-all mechanism but more local try blocks provide a direct way to localize the source code that is the origin of an exception when it is thrown.

The `logic_error` and `runtime_error` classes each only add two constructors to the members they inherit from `exception`. For example:

```
class logic_error : public exception
{
public:
    explicit logic_error(const string& what_arg);
    explicit logic_error(const char* what_arg);
};
```

`runtime_error` is defined similarly and all the subclasses except for `system_error` also have constructors that accept a string or a `const char*` argument. The `system_error` class adds a data member of type `std::error_code` that records an error code and the constructors provide for specifying the error code.

Using Standard Exceptions

There is no reason why you shouldn't make use of the exception classes defined in the Standard Library in your code, and a few very good reasons why you should. You can use the standard exception types in two ways: you can throw exceptions of standard types in your code, and you can use a standard exception class as a base for your own exception types. Obviously, if you are going to throw standard exceptions, you should only throw them in circumstances consistent with their purpose. This means that you shouldn't be throwing `bad_cast` exceptions for instance because these have a very specific role already. However, you can use some of the exception classes derived from `logic_error` and `runtime_error` directly. To use a familiar example, you might throw a `range_error` exception in a `Box` class constructor when an invalid dimension is supplied as an argument:

```
Box::Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv}
{
    if(lv <= 0.0 || wv <= 0.0 || hv <= 0.0)
        throw std::range_error("Zero or negative Box dimension.");
}
```

Of course, the source file would need to include the `stdexcept` header that defines the `range_error` class. The body of the constructor throws a `range_error` exception if any of the arguments are zero or negative.

Deriving your own Exception Classes

A major point in favor of deriving your own classes from one of the standard exception classes is that your classes become part of the same family. This makes it possible for you to catch standard exceptions as well as your own exceptions within the same catch blocks. For instance, if your exception class is derived from `logic_error`, then a catch block with a parameter type of `logic_error&` catches your exceptions as well as the standard exceptions with that base. A catch block with `exception&` as its parameter type always catches standard exceptions— as well as yours, as long as your classes have `exception` as a base.

You could incorporate the `Trouble` exception class and the classes derived from it into the standard exception family quite simply, by deriving it from the exception class. You just need to modify the class definition as follows:

```
class Trouble : public std::exception
{
public:
    Trouble(const char* pStr = "There's a problem") noexcept;
    virtual ~Trouble();
    virtual const char* what() const noexcept;

private:
    const char* message;
};
```

This provides its own implementation of the virtual `what()` member defined in the base class. Your version displays the message from the class object, as before. With your new knowledge of the exception specification for functions, you've added an exception specification to each member function so that no exceptions are thrown from within them. You also need to update the member functions of the classes `MoreTrouble` and `BigTrouble` that are derived from `Trouble` in a similar fashion. Each of the definitions for the member functions must include the same exception specification that appears for the function in the class definition.

Returning to the `Box` class definition in the previous section - it would be useful to derive an exception class from `std::range_error` to provide the option of a more specific string to be returned by `what()` that identifies the problem causing the exception to be thrown. Here's how you might do that:

```
#ifndef DIMENSION_ERROR_H
#define DIMENSION_ERROR_H
#include<stdexcept>                // For derived exception classes
#include <string>                  // For string type
using std::string;
using std::range_error;

class dimension_error : public range_error
{
public:
    using range_error::range_error;          // Inherit base constructors

    dimension_error(std::string str, int dim) :
        std::range_error {str + std::to_string(dim)} {}
};
#endif
```

The `using` directive causes the base class constructors to be inherited so the class will include these two constructors:

```
explicit dimension_error( const std::string& what_arg): std::range_error {what_arg} {}
explicit dimension_error( const char* what_arg): std::range_error {what_arg} {}
```

This allows `dimension_error` objects to be created in the same way as base class objects. The additional constructor provides for an extra parameter that specifies the dimension value that caused the exception to be thrown. It calls the base class constructor with the argument as a new `string` object that is formed from appending a `string` representation of the second constructor argument, `dim`, with the `string` object passed as the first

argument. The `to_string()` function is a template function that is defined in the `string` header; it returns a string representation of its argument, which can be a value of any fundamental numeric type. The inherited `what()` function will return whatever string is passed to the constructor when the `dimension_error` object is created.

Here's how this exception class could be used in the `Box` class definition:

```
// Box.h
#ifndef BOX_H
#define BOX_H
#include <algorithm>                // For min() function template
#include "Dimension_error.h"

class Box
{
protected:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv}
    {
        if(lv <= 0.0 || wv <= 0.0 || hv <= 0.0)
            throw dimension_error {"Zero or negative Box dimension.", std::min(lv, std::min(wv, hv))};
    }

    double volume() const { return length*width*height; }
};
#endif
```

The `Box` constructor throws a `dimension_error` exception if any of the arguments are zero or negative. The constructor uses the `min()` template function from the `algorithm` header to determine the dimension argument that is the minimum of those specified - that will be the worst offender. An example to demonstrate the `dimension_error` class in action is:

```
// Ex15_08.cpp
// Using an exception class
#include <iostream>
#include "Box.h"                // For the Box class
#include "Dimension_error.h"    // For the dimension_error class

int main()
try
{
    Box box1 {1.0, 2.0, 3.0};
    std::cout << "box1 volume is " << box1.volume() << std::endl;
    Box box2 {1.0, -2.0, 3.0};
    std::cout << "box1 volume is " << box2.volume() << std::endl;
}
```

```
catch (std::exception& ex)
{
    std::cout << "Exception caught in main(): " << ex.what() << std::endl;
}
```

The output from this example is:

```
box1 volume is 6
Exception caught in main(): Zero or negative Box dimension.-2.000000
```

The body of `main()` is a try block and its catch block catches any type of exception that has `std::exception` as a base. The output shows that the `Box` class constructor is throwing a `dimension_error` exception object when a dimension is negative. The output also shows that the `what()` function that `dimension_error` inherits from `range_error` is outputting the string formed in the `dimension_error` constructor call.

Summary

Exceptions are an integral part of programming in C++. Several operators throw exceptions and you've seen that they're used extensively within the Standard Library to signal errors. Therefore it's important that you have a good grasp of how exceptions work, even if you don't plan to define your own exception classes. The important points that I've covered in this chapter are as follows:

- Exceptions are objects that are used to signal errors in a program.
- Code that may throw exceptions is usually contained within a try block, which enables an exception to be detected and processed within the program.
- The code to handle exceptions that may be thrown in a try block is placed in one or more catch blocks that must immediately follow the try block.
- A try block, along with its catch blocks, can be nested inside another try block.
- A catch block with a parameter of a base class type can catch an exception of a derived class type.
- A catch block with the parameter specified as an ellipsis will catch an exception of any type.
- If an exception isn't caught by any catch block, then the `std::terminate()` function is called, and this calls `std::abort()`.
- The Standard Library defines a range of standard exception types in the `stdexcept` header that are derived from the `std::exception` class that is defined in the exception header.
- The `noexcept` specification for a function indicates that the function does not throw exceptions.
- A function try block for a constructor can enclose the initialization list as well as the body of the constructor.
- The `uncaught_exception()` function allows you to detect when a destructor is called as a result of an exception being thrown.

EXERCISES

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck after that, you can download the solutions from the Apress website (www.apress.com/source-code/), but that really should be a last resort.

- Exercise 15-1. Derive your own exception class called `CurveBall` from the `std::exception` class to represent an arbitrary error, and write a function that throws this exception approximately 25 percent of the time. (One way to do this is to generate a random number between 1 and 20, and if the number is 5 or less, throw the exception.) Define a `main()` function to call this function 1,000 times and to record and display the number of times an exception was thrown.
- Exercise 15-2. Define another exception class called `TooManyExceptions`. Then throw an exception of this type from the `catch` block for `CurveBall` exceptions in the previous exercise when the number of exceptions caught exceeds 10.
- Exercise 15-3. Implement your terminate handler in the code for the previous example so that a message is displayed when the `TooManyExceptions` exception is thrown.
- Exercise 15-4. A `sparse array` is an array in which most of the element values are zero or empty. Define a class for a one-dimensional sparse array of elements that are values of type `double` such that only non-zero elements are stored. The potential number of elements should be specified as a constructor argument, so a sparse array to store up to 100 elements can be defined with this statement:

```
SparseArray values {100};
```

Implement the subscript operator for the `SparseArray` class so that you can use array notation to retrieve or store elements. Throw an exception that identifies the erroneous subscript if the legal index range is exceeded in the subscript operator function. (Hint: You could use a linked list of some kind internally to store the elements...). Create an example to demonstrate the operation of the `SparseArray` class including catching and recovering from exceptions thrown by the subscript operator function.



Class Templates

You learned about templates that the compiler uses to create functions back in Chapter 8; this chapter is about templates the compiler can use to create classes. Class templates are a powerful mechanism for generating new class types automatically. A significant portion of the Standard Library is built entirely on the ability to define templates, particularly the Standard Template Library, which includes many class and function templates.

By the end of this chapter, you will have learned:

- What a class template is and how it is defined
- What an instance of a class template is, and how it is created
- How to define templates for member functions of a class template outside the class template definition
- How type parameters differ from non-type parameters
- How static members of a class template are initialized
- What a partial specialization of a class template is and how it is defined
- How a class can be nested inside a class template

Understanding Class Templates

Class templates are based on the same idea as the function templates. A class template is a *parameterized type* — a recipe for creating a family of class types, using one or more parameters. The argument for each parameter is typically (but not always) a type. When you define a variable that has a type specified by a class template, the compiler uses the template to create a definition of a class using the template arguments that you use in the type specification. You can use a class template to generate any number of different classes. It's important to keep in mind that a class template is not a class, but just a recipe for creating classes, because this is the reason for many of the constraints on how you define class templates.

A class template has a name, just like a regular class, and one or more parameters. A class template must be unique within a namespace, so you can't have another template with the same name and parameter list in the namespace in which the template is defined. A class *definition* is generated from a class template when you supply an argument for each of the template's parameters. This is illustrated in Figure 16-1.

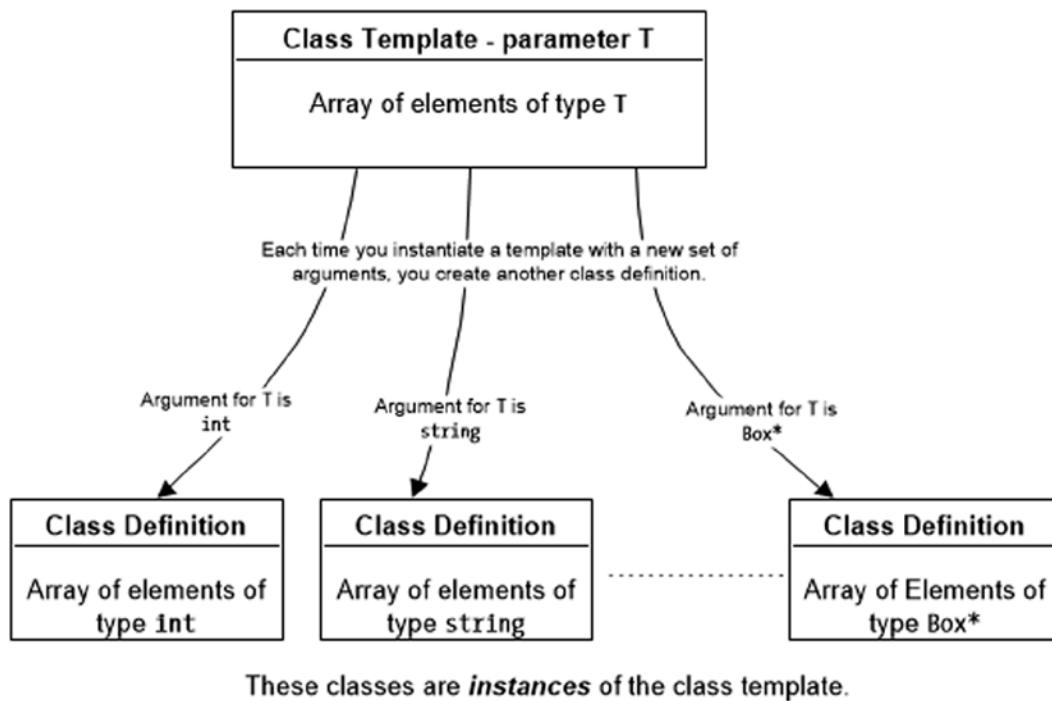


Figure 16-1. Instantiating a template

Each class that the compiler generates from a template is called an instance of the template. When you define a variable using a template type for the first time, you create an instance of the template; variables of the same type defined subsequently will use the first instance created. You can also cause instances of a class template to be created without defining a variable. The compiler does not process a class template in a source file in any way if the template is not used to generate a class.

There are many applications for class templates but they are perhaps most commonly used to define *container classes*. These are classes that can contain sets of objects of a given type, organized in a particular way. In a container class the organization of the data is independent of the type of objects stored. Of course, you already have experience of instantiating and using `std::vector` and `std::array` class templates that define containers where the data is organized sequentially.

Defining Class Templates

Class template definitions tend to look more complicated than they really are, largely because of the notation used to define them and the parameters sprinkled around the statements in their definitions. Class template definitions are similar to those of ordinary classes, but like so many things, the devil is in the details. A class template is prefixed by the template keyword followed by the parameters for the template between angled brackets. The template class definition consists of the class keyword followed by the class template name with the body of the definition between braces. Just like a regular class, the whole definition ends with a semicolon. The general form of a class template looks like this:

```
template <template parameter list>
class ClassName
{
    // Template class definition...
};
```

ClassName is the name of this template. You write the code for the body of the template just as you'd write the body of an ordinary class, except that some of the member declarations and definitions will be in terms of the template parameters that appear between the angled brackets. To create a class from a template, you must specify arguments for every parameter in the list. This differs from a function template where most of the time the compiler can deduce the template arguments from the context.

Template Parameters

A template parameter list can contain any number of parameters that can be of two kinds — *type parameters* and *non-type parameters*. The argument corresponding to a type parameter is always a type, such as `int`, or `std::string`, or `Box`; the argument for a non-type parameter can be a literal of an integral type such as `200`, an integral constant expression, a pointer or reference to an object, a pointer to a function or a pointer that is null. Type parameters are much more commonly used than non-type parameters, so I'll explain these first and defer discussion of non-type parameters until later in this chapter.

■ **Note** There's a third possibility for class template parameters. A parameter can also be a *template* where the argument must be an instance of a class template. A detailed discussion of this possibility is a little too advanced for this book.

Figure 16-2 illustrates the options for type parameters. You can write type parameters using the `class` keyword or the `typename` keyword preceding the parameter name (`typename T` in Figure 16-2 for example). I prefer to use `typename` because `class` tends to connote a class type and the type argument doesn't have to be a class type. `T` is often used as a type parameter name (or `T1`, `T2`, and so on when there are several type parameters) because it's concise and easy to identify in the template definition but you can use whatever name you want.

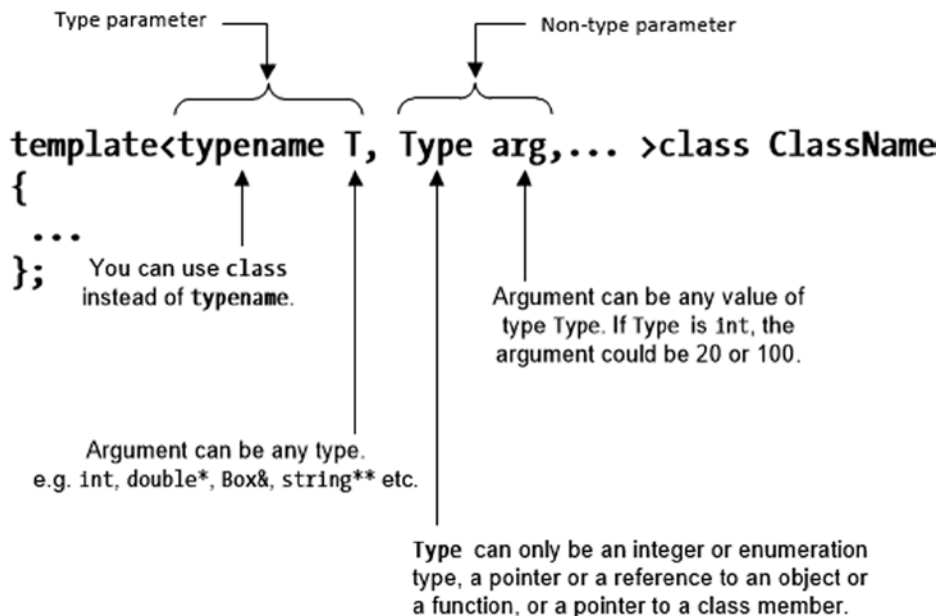


Figure 16-2. Class template parameters

A Simple Class Template

Let's take an example of a class template for arrays that will do bounds checking on index values to make sure that they are legal. The Standard Library provides a comprehensive implementation of an array template but building a limited array template is an effective basis from which you can learn how templates work. You already have clear idea of how arrays work so you can concentrate on the template specifics.

This template just has a single type parameter, so in outline, its definition will be:

```
template <typename T>
class Array
{
    // Definition of the template...
};
```

The Array template has just one type parameter, T. You can tell that it's a type parameter because it's preceded by the keyword `typename`. Whatever is “plugged in” for the parameter when you instantiate the template — `int`, `double*`, `string`, or whatever — determines the type of the elements stored in an object of the resultant class. The definition in the body of the template will be much the same as a class definition, with data and function members that are specified as `public`, `protected`, or `private`, and it will typically have constructors and a destructor. You can use T to define data members or to specify the parameters or return types for function members, either by itself or in types such as `T*` or `T&&`. You can use the template name with its parameter list — `Array<T>`, in this case — as a type name when specifying data and function members.

The very least we need by way of a class interface is a constructor; a copy constructor because the space for the array will need to be allocated dynamically; an assignment operator because the compiler will supply an unsuitable version if there isn't one defined; an overloaded subscript operator; and finally a destructor. With this in mind, the initial definition of the template looks like this:

```
template <typename T>
class Array
{
private:
    T* elements;                // Array of type T
    size_t size;                // Number of array elements

public:
    explicit Array<T>(size_t arraySize);    // Constructor
    Array<T>(const Array<T>& array);        // Copy Constructor
    ~Array<T>();                            // Destructor
    T& operator[](size_t index);            // Subscript operator
    const T& operator[](size_t index) const; // Subscript operator-const arrays
    Array<T>& operator=(const Array<T>& rhs); // Assignment operator
};
```

The body of the template looks much like a regular class definition, except for the type parameter, T, in various places. For example, it has a data member, `elements`, which is of type *pointer to T* (equivalent to *array of T*). When the template is instantiated to produce a specific class definition, T is replaced by the actual type used to instantiate the template. If you create an instance of the template for type `double`, `elements` will be of type `double*` or *array of double*. The operations that the template needs to perform on objects of type T will obviously place requirements on the definition of type T when T is a class type.

The first constructor is declared as `explicit` to prevent its use for implicit conversions. The subscript operator has been overloaded on `const`. The non-`const` version of the subscript operator applies to non-`const` array objects and can return a non-`const` reference to an array element. Thus this version can appear on the left of an assignment. The `const` version is called for `const` objects and returns a `const` reference to an element; obviously this can't appear on the left of an assignment.

The assignment operator function parameter is of type `const Array<T>&`. This type is *const reference to Array<T>*. When a class is synthesized from the template — with `T` as type `double`, for example — this is a `const` reference to the class name for that particular class, which would be `const Array<double>`, in this case. More generally, the class name for a specific instance of a template is formed from the template name followed by the actual type argument between angled brackets. The template name followed by the list of parameter names between angled brackets is called the *template ID*.

It's not essential to use the full template ID within a template definition. Within the body of the `Array` template, `Array` by itself will be taken to mean `Array<T>`, and `Array&` will be interpreted as `Array<T>&`, so I can simplify the class template definition:

```
template <typename T>
class Array
{
private:
    T* elements;                // Array of type T
    size_t size;                // Number of array elements

public:
    explicit Array(size_t arraySize);    // Constructor
    Array(const Array& array);           // Copy Constructor
    ~Array();                           // Destructor
    T& operator[](size_t index);         // Subscript operator
    const T& operator[](size_t index) const; // Subscript operator-const arrays
    Array& operator=(const Array& rhs);   // Assignment operator
    size_t getSize() { return size; }     // Accessor for size
};
```

■ **Caution** You *must* use the template ID to identify the template outside the body of the template. This will apply function members of a class template that are defined outside the template.

It's desirable that the number of elements in an `Array<T>` object can be determined so the `getSize()` member provides this. The assignment operator allows one `Array<T>` object to be assigned to another, which is something you can't do with ordinary arrays. If you wanted to inhibit this capability, you would still need to declare the `operator=()` function as a member of the template. If you don't, the compiler will create a public default assignment operator when necessary for a template instance. To prevent use of the assignment operator, just declare it as a `private` member of the class or use `=delete` in the declaration to prevent the compiler from supplying the default; then it can't be accessed. Of course, you don't need an implementation for the function member in this case, because you are not required to implement a function member unless it is used, and a `private` member will never be used outside the class. The `getSize()` member is implemented within the class template so it's inline by default and no external definition is necessary.

Defining Function Members of a Class Template

If you include the definitions for the function members of a class template within its body, they are implicitly inline in any instance of the template, just like in an ordinary class. However, you'll want to define members outside of the template body from time to time, especially if they involve a lot of code. The syntax for doing this is a little different from what applies for a normal class.

The clue to understanding the syntax is to realize that external definitions for function members of a class template are themselves templates. This is true even if a function member has no dependence on the type parameter *T*, so `getSize()` would need a template definition if it was not defined inside the class template. The parameter list for the template that defines a function member must be identical to that of the class template. Let's start by defining the constructors for the `Array` template.

Constructor Templates

When you're defining a constructor outside a class template definition, its name must be qualified by the class template name in a similar way to a function member of an ordinary class. However, this isn't a function definition, it's a *template* for a function definition, so that has to be expressed as well. Here's the definition of the constructor:

```
template <typename T>                // This is a template with parameter T
Array<T>::Array(size_t arraySize) : size {arraySize}, elements {new T[arraySize]}
{ }
```

The first line identifies this as a template and also specifies the template parameter as *T*. Splitting the template function declaration into two lines, as I've done here, is only for illustrative purposes, and isn't necessary if the whole construct fits on one line. The template parameter is essential in the qualification of the constructor name because it ties the function definition to the class template. Note that you *don't* use the `typename` keyword in the qualifier for the member name; it's only used in the template parameter list. You don't need a parameter list after the constructor name. When the constructor is instantiated for an instance of the class template — for type `double` for example — the type name replaces *T* in the constructor qualifier, so the qualified constructor name for the class `Array<double>` is `Array<double>::Array()`.

In the constructor, you must allocate memory in the free store for an `elements` array that contains `size` elements of type *T*. If *T* is a class type, a public default constructor must exist in the class *T*. If it doesn't, the instance of this constructor won't compile. The operator `new` throws a `bad_alloc` exception if the memory can't be allocated for any reason, so you might want to put the body of the `Array` constructor in a `try` block:

```
template <typename T>                // This is a template with parameter T
Array<T>::Array(size_t arraySize) try : size {arraySize}, elements {new T[arraySize]}
{ }
catch (std::bad_alloc& )
{
    std::cerr << "memory allocation failed for Array object.\n";
}
```

This will output a message to `std::cerr` if `new` fails to allocate the memory. `cerr` is the standard error output stream defined in the `iostream` header. It encapsulates the same destination as `cout` but ensures the stream is flushed so the output appears immediately. The parameter name is omitted from the `catch` block parameter list because it is not referenced; some compilers will issue a warning for local variables that are never referenced.

Of course, members defined within a class template body are `inline` by default and you can specify this too for an external template for a function member of a class template:

```
template <typename T>                // This is a template with parameter T
inline Array<T>::Array(size_t arraySize)
try : size {arraySize}, elements {new T[arraySize]}
{}
catch (std::bad_alloc& )
{
    std::cerr << "memory allocation failed for Array object.\n";
}
```

You place the `inline` keyword following the template parameter list and preceding the member name.

The copy constructor has to create an array for the object being created that's the same size as that of its argument, and then copy the latter's data members to the former. Here's the code to do that:

```
template <typename T>
inline Array<T>::Array(const Array& array)
try : size {array.size}, elements {new T[array.size]}
{
    for (size_t i {}; i < size; ++i)
        elements[i] = array.elements[i];
}
catch (std::bad_alloc&)
{
    std::cerr << "memory allocation failed for Array object copy.\n";
}
```

This assumes that the assignment operator works for type `T`. This demonstrates how important it is to always define the assignment operator for classes that allocate memory dynamically. If the class `T` doesn't define it, the default for `T` is used, with undesirable side effects if creating a `T` object involves allocating memory dynamically. Without seeing the code for the template before you use it, you may not realize the dependency on the assignment operator. Because the copy constructor also allocates memory using the `new` operator, the body is also in a `try` block and the `catch` block issues a message that identifies where the memory allocation failed.

The Destructor Template

In many cases a default constructor will be OK in a class generated from a template but this is not the case here. The destructor must release the memory for the `elements` array, so its definition will be:

```
template <typename T> inline Array<T>::~~Array()
{
    delete[] elements;
}
```

We are releasing memory allocated for an array so we must use the `delete[]` form of the operator. Failing to define this template would result in all classes generated from the template having major memory leaks.

Subscript Operator Templates

The `operator[]()` function is quite straightforward, but we must ensure illegal index values can't be used. For an index value that is out of range, we can throw an exception:

```
template <typename T> inline T& Array<T>::operator[](size_t index)
{
    if (index >= size) throw std::out_of_range {"Index too large: " + std::to_string(index)};

    return elements[index];
}
```

I could define an exception class to use here, but it's easier to borrow the `out_of_range` class type that's already defined in the `stdexcept` header. This is thrown if you index a string object with an out of range index value for example, so the usage here is consistent with that. An exception of type `out_of_range` is thrown if the value of `index` is not between 0 and `size-1`. The argument to the `out_of_range` constructor is a string object that describes the error and includes the erroneous index value. A null-terminated string (type `const char*`) corresponding to the string object is returned by the `what()` member of the exception object. The argument that is passed to the `out_of_range` constructor is a message that includes the erroneous index value to make tracking down the source of the problem a little easier. An index cannot be less than zero because it is of type `size_t`, which is an unsigned integer type.

The `const` version of the subscript operator function will be almost identical to the non-`const` version:

```
template <typename T> inline const T& Array<T>::operator[](size_t index) const
{
    if (index >= size) throw std::out_of_range {"Index too large: " + std::to_string(index)};

    return elements[index];
}
```

The Assignment Operator Template

There's more than one possibility for how the assignment operator works. The operands must be of the same `Array<T>` type with the same `T` but this does not prevent the `size` members from having different values. You could implement the assignment operator so that the left operand retains the same value for its `size` member. If the left operand has fewer elements than the right operand, you would just copy sufficient elements from the right operand to fill the array for the left operand. If the left operand has more elements than the right operand, you could either leave the excess elements at their original values or set them to the value produced by the default `T` constructor.

To keep it simple, I'll just make the left operand have the same `size` value as the right operand. To implement this, the assignment operator function must release any memory allocated in the destination object and then do what the copy constructor did, after checking that the objects are not identical of course. Here's the definition:

```
template <typename T> inline Array<T>& Array<T>::operator=(const Array& rhs)
try
{
    if (&rhs != this)                                // If lhs != rhs...
    {                                                  // ...do the assignment...
        if (elements)                                // If lhs array exists
            delete[] elements;                        // release the free store memory
    }
}
```

```

        size = rhs.size;
        elements = new T[rhs.size];
        for (size_t i {}; i < size; ++i)
            elements[i] = rhs.elements[i];
    }
    return *this;                // ... return lhs
}
catch (std::bad_alloc&)
{
    std::cerr << "memory allocation failed for Array assignment.\n";
}

```

Checking to make sure that the left operand is identical to the right is essential; otherwise you'd free the memory for the `elements` member of the object pointed to by `this`, then attempt to copy it to itself when it no longer exists. When the operands are different, you release any heap memory owned by the left operand before creating a copy of the right operand. This has the potential to throw `bad_alloc` so I have put the function body in a `try` block too. Heap memory allocation failure is a rare occurrence these days because physical memory is large and virtual memory is very large so checking for `bad_alloc` is often omitted. If `bad_alloc` is thrown, you'll definitely know about it anyway. I'll omit the `try/catch` combination for `bad_alloc` from any subsequent examples where it might apply to keep the code shorter and less cluttered.

All the function member definitions that you've written here are templates and they are inextricably bound to the class template. They are not function definitions, they're templates to be used by the compiler when the code for one of the member functions of the class template needs to be generated, so they need to be available in any source file that uses the template. For this reason, you'd normally put all the definitions of the member functions for a class template in the header file that contains the class template itself.

Instantiating a Class Template

The compiler instantiates a class template as a result of a definition of an object that has a type produced by the template. Here's an example:

```
Array<int> data {40};
```

When this statement is compiled, two things happen: the definition for the `Array<int>` class is created so that the type is identified, and the constructor definition is generated because it must be called to create the object. This is all that the compiler needs to create the `data` object so this is the only code that it provides from the templates at this point.

The class definition that'll be included in the program is generated by substituting `int` in place of `T` in the template, but there's one subtlety. The compiler *only* compiles the member functions that your program *uses*, so you do not necessarily get the entire class that would be produced by a simple substitution for the template parameter. On the basis of just the definition for the object, `data`, it is equivalent to:

```

class Array<int>
{
private:
    int* elements;                // Array of type int
    size_t size;                 // Number of array elements

public:
    Array(size_t arraySize);      // Constructor
};

```

You can see that the only function member is the constructor. The compiler won't create instances of anything that isn't required to create the object, and it won't include parts of the template that aren't needed in the program. This implies that there can be coding errors in a class template and a program that uses the template may still compile, link, and run successfully. If the errors are in parts of the template that aren't required by the program, they won't be detected by the compiler because they are not included in the code that is compiled. Obviously, you are almost certain to have other statements in a program besides the declaration of an object that use other function members — for instance, you'll always need the destructor to destroy the object — so the ultimate version of the class in the program will include more than that shown in the preceding code. The point is that what is finally in the class generated from the template will be precisely those parts that are actually used in the program, which is not necessarily the complete template.

■ **Caution** Of course, this implies that you must take care when testing your own class templates to ensure that all the function members are generated and tested. You also need to consider what the template does across a range of types so you need to test a template with pointers and references as the template type argument.

The instantiation of a class template from a definition is referred to as an *implicit instantiation* of the template, because it arises as a by-product of declaring an object. This terminology is also to distinguish it from an *explicit instantiation* of a template, which I'll get to shortly and which behaves a little differently.

As I said, the declaration of data also causes the constructor, `Array<int>::Array()`, to be called, so the compiler uses the function template that defines the constructor to create a definition for the constructor for the class:

```
inline Array<int>::Array(size_t arraySize) try : size {arraySize}, elements {new int[arraySize]}
{}
catch (std::bad_alloc & )
{
    std::cerr << "memory allocation failed for Array object creation.\n";
}
```

Each time you define a variable using a class template with a different type argument, a new class is defined and included in the program. Because creating the class object requires a constructor to be called, the definition of the appropriate class constructor is also generated. Of course, creating objects of a type that you've created previously doesn't necessitate any new template instances. The compiler uses any previously created template instances as required.

When you use the function members of a particular instance of a class template — by calling functions on the object that you defined using the template, for example — the code for each member function that you use is generated. If you have member functions that you don't use, no instances of their templates are created. The creation of each function definition is an implicit template instantiation because it arises out of the use of the function. The template itself isn't part of your executable code. All it does is enable the compiler to generate the code that you need automatically. This process is illustrated in Figure 16-3.

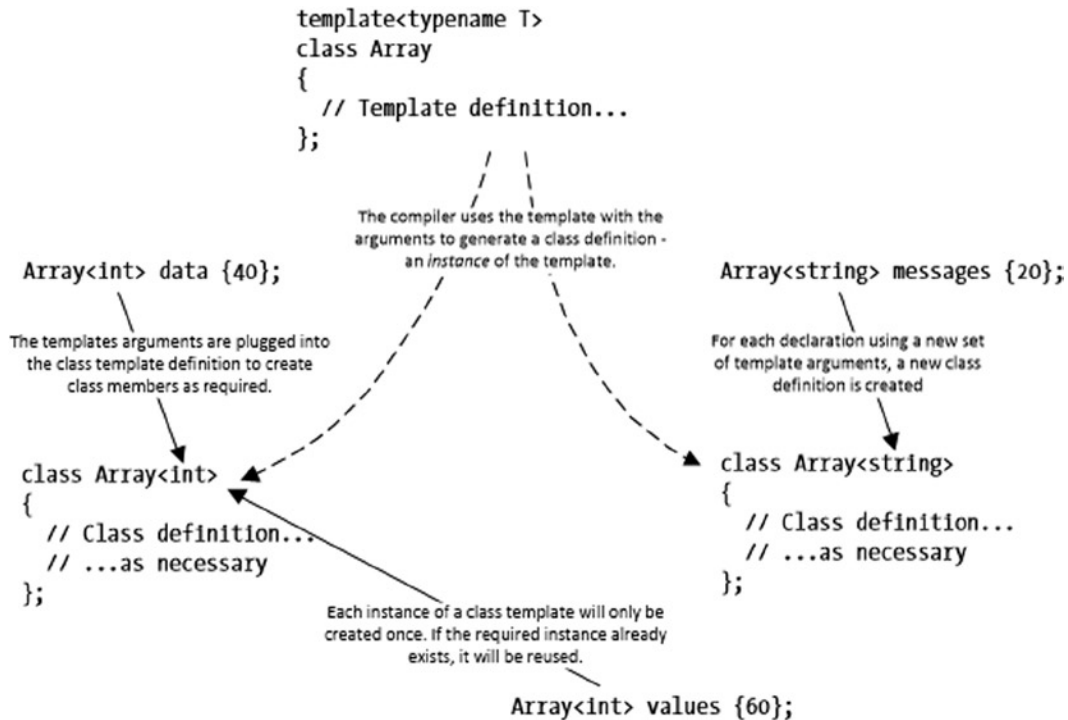


Figure 16-3. Implicit instantiation of a class template

Note that a class template is only implicitly instantiated when an object of the specific template type needs to be created. Declaring a pointer to an object type won't cause an instance of the template to be created. Here's an example:

```
Array<string*> pObj; // A pointer to a template type
```

This defines `pObj` as type *pointer to type* `Array<string*>`. No object of type `Array<string*>` is created as a result of this statement so no template instance is created. Contrast this with the following statement:

```
Array<std::string*> pMessages {10};
```

This time the compiler does create an instance of the class template. This defines an `Array<std::string*>` object so each element of `pMessages` can store a pointer to an `std::string` object. An instance of the template defining the constructor is also generated. Let's try out the `Array` template in a working example. You can put the class template and the templates defining the member functions of the template all together in a header file `Array.h`:

```

// Array class template definition
#ifndef ARRAY_H
#define ARRAY_H
#include <stdexcept> // For standard exception types
#include <string>     // For to_string()

// Definition of the Array<T> template...

// Definitions of the templates for function members of Array<T>...
#endif

```


To use the class template, you just need a program that'll declare some arrays using the template and try them out. The example will create an Array of Box objects - you can use this definition for the Box class:

```
// Box.h
#ifndef BOX_H
#define BOX_H

class Box
{
protected:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv} {}
    Box() = default;

    double volume() const { return length*width*height; }
};
#endif
```

I'll use some out-of-range index values in the example, just to show that it works:

```
// Ex16_01.cpp
// Using a class template
#include "Box.h"
#include "Array.h"
#include <iostream>
#include <iomanip>

int main()
{
    const size_t nValues {50};
    Array<double> values {nValues};           // Class constructor instance created
    try
    {
        for (size_t i {}; i < nValues; ++i)
            values[i] = i + 1;               // Member function instance created

        std::cout << "Sums of pairs of elements:";
        size_t lines {};
        for (size_t i {nValues - 1}; i >= 0; i--)
            std::cout << (lines++ % 5 == 0 ? "\n" : "")
                << std::setw(5) << values[i] + values[i - 1];
    }
    catch (const std::out_of_range& ex)
    {
        std::cerr << "\nout_of_range exception object caught! " << ex.what() << std::endl;
    }
}
```

```

try
{
    const size_t nBoxes {10};
    Array<Box> boxes {nBoxes};           // Template instance created
    for (size_t i {} ; i <= nBoxes ; ++i) // Member instance created in loop
        std::cout << "Box volume is " << boxes[i].volume() << std::endl;
}
catch (const std::out_of_range& ex)
{
    std::cerr << "\nout_of_range exception object caught! " << ex.what() << std::endl;
}
}

```

This example will produce the following output:

```

Sums of pairs of elements:
 99  97  95  93  91
 89  87  85  83  81
 79  77  75  73  71
 69  67  65  63  61
 59  57  55  53  51
 49  47  45  43  41
 39  37  35  33  31
 29  27  25  23  21
 19  17  15  13  11
 9   7   5   3
out_of_range exception object caught! Index too large: 4294967295
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
Box volume is 1
out_of_range exception object caught! Index too large: 10

```

The `main()` function creates an object of type `Array<double>` that implicitly creates an instance of the class template with a type argument of `double`. The number of elements in the array is specified by the argument to the constructor, `nValues`. The compiler will also create an instance of the template for the constructor definition.

Within the first try block, the elements of the `values` object with are initialized values from 1 to `nValues` in a for loop. The expression `values[i]` results in an instance of the subscript operator function being created. This instance is called implicitly by this expression as `values.operator[](i)`. Because `values` is not `const`, the non-`const` version of the operator function is called. A second for loop in the try block outputs the sums of successive pairs of elements, starting at the end of the array. The code in this loop also calls the subscript operator function, but because the instance of the function template has already been created, no new instance is generated. Clearly, the expression `values[i-1]` has an illegal index value when `i` is 0, so this causes an exception to be thrown by the `operator[]()` function. The catch block catches this and outputs a message to the standard error stream. The `what()` function for the `out_of_range` exception returns a null-terminated string that corresponds to the `string` object passed to the

constructor when the exception object was created. You can see from the output that the exception was thrown by the overloaded subscript operator function and that the index value is very large. The value of the index suggests that it originated by decrementing an unsigned zero value.

When the exception is thrown by the subscript operator function, control is passed immediately to the handler, so the illegal element reference is not used and nothing is stored at the location indicated by the illegal index. Of course, the loop also ends immediately at this point.

The next try block defines an object that can store Box objects. This time, the compiler generates an instance of the class template, `Array<Box>`, which stores an array of Box objects, because the template has not been instantiated for Box objects previously. The statement also calls the constructor to create the `boxes` object so an instance of the function template for the constructor is created. The constructor for the `Array<Box>` class calls the default constructor for the Box class when the `elements` member is created in the free store. Of course, all the Box objects in the `elements` array have the default dimensions of $1 \times 1 \times 1$.

The volume of each Box object in `boxes` is output in a for loop. The expression `boxes[i]` calls the overloaded subscript operator, so again the compiler uses an instance of the template to produce a definition of this function. When `i` has the value `nBoxes`, the subscript operator function throws an exception because an index value of `nBoxes` is beyond the end of the `elements` array. The catch block following the try block catches the exception. Because the try block is exited, all locally declared objects will be destroyed, including the `boxes` object. The `values` object still exists at this point because it was created before the first try block and it is still in scope.

Static Members of a Class Template

A class template can have static members, just like an ordinary class. Static function members of a template class are quite straightforward. Each instance of a class template instantiates the static function member as needed. A static function member has no `this` pointer and therefore can't refer to non-static members of the class. The rules for defining static function members of a class template are the same as those for a class, and a static function member of a class template behaves in each instance of the template just as if it were in an ordinary class.

A static data member is a little more interesting because it needs to be initialized outside the template definition. Suppose the `Array<T>` template contained a static data member of type `T`:

```
template <typename T>
class Array
{
private:
    static T value;           // Static data member
    T* elements;             // Array of type T
    size_t size;             // Number of array elements

public:
    explicit Array(size_t arraySize); // Constructor
    Array(const Array& array);         // Copy Constructor
    ~Array();                         // Destructor
    T& operator[](size_t index);       // Subscript operator
    const T& operator[](size_t index) const; // Subscript operator-const arrays
    Array& operator=(const Array& rhs); // Assignment operator
    size_t getSize() { return size; }  // Accessor for size
};
```

The initialization for the value member is accomplished through a template in the same header file:

```
template <typename T> T Array<T>::value;    // Initialize static data member
```

This initializes value with the equivalent of 0 so it effectively calls the default constructor for type T. A static data member is always dependent on the parameters of the template of which it is a member regardless of its type, so you must initialize value as a template with parameter T. The static variable name must also be qualified with the type name Array<T> so that it's identified with the instance of the class template that is generated. You can't use Array by itself here, because this template for initializing value is outside the body of the class template, and the template ID is Array<T>. Suppose you wanted a static member of Array<T> to define a minimum number of elements:

```
template <typename T>
class Array
{
private:
    static size_t minSize;                // Minimum number of elements

    // Rest of the class as before...
};
```

Even though minSize is of a fundamental type - size_t is an alias for an unsigned integer type - you still must initialize it in a template and qualify the member name with the template ID:

```
template<typename T> size_t Array<T>::minSize {5};
```

minSize can only exist as a member of a class generated from the Array<T> template and each such class has its own minSize member. It is therefore inevitable that it can only be initialized through another template. Note that creating an instance of a template does not guarantee that a static data member is defined. A static data member of a class template will only be defined if it is used because the compiler will only process the template that initializes the static data member when the member is used.

Non-Type Class Template Parameters

A non-type parameter looks like a function parameter — a type name followed by the name of the parameter. Therefore, the argument for a non-type parameter is a value of the given type. However, you can't use just any type for a non-type parameter in a class template. Non-type parameters are intended to be used to define values that might be useful in specifying a container, such as array dimensions or other size specification, or possibly as upper and lower limits for index values.

A non-type parameter can only be an integral type, such as size_t or long; an enumeration type; a pointer or a reference to an object, such as string* or Box&; a pointer or a reference to a function; or a pointer to a member of a class. You can conclude from this that a non-type parameter can't be a floating point type or any class type, so types double, Box, and std::string are not allowed, and neither is std::string*. Remember that the primary rationale for non-type parameters is to allow sizes and range limits for containers to be specified. Of course, the argument corresponding to a non-type parameter can be an object of a class type, as long as the parameter type is a reference. For a parameter of type Box&, for example, you could use any object of type Box as an argument.

A non-type parameter is written just like a function parameter, with a type name followed by a parameter name. Here's an example:

```
template <typename T, size_t size>
class ClassName
{
    // Definition using T and size...
};
```

This template has a type parameter, *T*, and a non-type parameter, *size*. The definition is expressed in terms of these two parameters and the template name. If you need it, the *type name* of a type parameter can also be the *type* for a non-type parameter:

```
template <typename T,                // T is the name of the type parameter
          size_t size,
          T value>                  // T is also the type of this non-type parameter
class ClassName
{
    // Definition using T, size, and value...
};
```

This template has a non-type parameter, *value*, of type *T*. The parameter *T* must appear before its use in the parameter list, so *value* couldn't precede the type parameter *T* here. Note that using the same symbol with the type and non-type parameters implicitly restricts the possible arguments for the type parameter to the types permitted for a non-type argument (in other words, *T* can only be an integral type).

To illustrate how you could use non-type parameters, suppose you defined the class template for arrays as follows:

```
template <typename T, size_t arraySize, T value>
class Array
{
    // Definition using T, size, and value...
};
```

You could now use the non-type parameter, *value*, to initialize each element of the array in the constructor:

```
template <typename T, int arraySize, T value>
Array<T, arraySize, value>::Array(size_t arraySize) : size {arraySize}, elements {new T[size]}
{
    for(size_t i {} ; i < size ; ++i)
        elements[i] = value;
}
```

This is not a very intelligent approach to initializing the members of the array. This places a serious constraint on the types that are legal for *T*. Because *T* is used as the type for a non-type parameter it is subject to the constraints on non-type parameter types. A non-type parameter can only be an integral type, a pointer, or a reference, so you can't create *Array* objects to store double values or *Box* objects, so the usefulness of this template is somewhat restricted.

To provide a more credible example, I'll add a non-type parameter to the Array template to allow flexibility in indexing the array:

```
template <typename T, int startIndex>
class Array
{
private:
    T* elements;                // Array of type T
    size_t size;                // Number of array elements

public:
    explicit Array(size_t arraySize);    // Constructor
    Array(const Array& array);           // Copy Constructor
    ~Array();                           // Destructor
    T& operator[](int index);            // Subscript operator
    const T& operator[](int index) const; // Subscript operator-const arrays
    Array& operator=(const Array& rhs);  // Assignment operator
    size_t getSize() { return size; }    // Accessor for size
};
```

This adds a non-type parameter, `startIndex` of type `int`. The idea is that you can specify that you want to use index values that vary over a given range. For example, to create an `Array<>` object that allows index values from -10 to +10, you would specify the array with the non-type parameter value as -10 and the argument to the constructor as 21 because the array would need 21 elements. Index values can now be negative so the parameter for the subscript operator functions has been changed to type `int`.

Because the class template now has two parameters, the templates defining the member functions of the class template must have the same two parameters. This is necessary even if some of the functions aren't going to use the non-type parameters. The parameters are part of the identification for the class template, so to match the template, they must have the same parameter list.

There are some serious disadvantages to what I have done here. A consequence of adding the `startIndex` template parameter is that different values for the argument generate different template instances. This means that an array of double values indexed from 0 will be a different type from an array of double values indexed from 1. If you use both in a program, two independent class definitions will be created from the template, each with whatever member functions you use. This has at least two undesirable consequences: first, you'll get a lot more compiled code in your program than you might have anticipated (a condition often known as *code bloat*); second (and far worse), you won't be able to intermix elements of the two types in an expression. It would be much better to provide flexibility for the range of index values by adding a parameter to the constructor rather than using a non-type template parameter. Here's how that would look:

```
template <typename T>
class Array
{
private:
    T* elements;                // Array of type T
    size_t size;                // Number of array elements
    int start;                  // Starting index value
};
```

```

public:
    explicit Array(size_t arraySize, int startIndex=0); // Constructor
    T& operator[](int index);                        // Subscript operator
    const T& operator[](int index) const;            // Subscript operator-const arrays
    Array& operator=(const Array& rhs);              // Assignment operator
    size_t getSize() { return size; }                // Accessor for size
};

```

The extra member, `start`, stores the starting index for the array specified by the second constructor argument. The default value for the `startIndex` parameter is 0, so normal indexing is obtained by default.

In the interests of seeing how the function members are defined when you have a non-template parameter, let's ignore the better definition and complete the set of function templates that you need for the version of the `Array` class template with the additional non-type parameter.

Templates for Function Members with Non-Type Parameters

Because you've added a non-type parameter to the class template definition, the code for the templates for all function members needs to be changed. The template for the constructor is:

```

template <typename T, int startIndex>
inline Array<T, startIndex>::Array(size_t arraySize) :
    size {arraySize}, elements {new T[arraySize]}
{}

```

The template ID is now `Array<T, startIndex>`, so this is used to qualify the constructor name. This is the only change from the original definition apart from adding the new template parameter to the template and omitting the try/catch blocks to deal with `bad_alloc`.

For the copy constructor, the changes to the template are similar:

```

template <typename T, int startIndex>
inline Array<T, startIndex>::Array(const Array& array) :
    size {array.size}, elements {new T[array.size]}
{
    for (size_t i {} ; i < size ; ++i)
        elements[i] = array.elements[i];
}

```

Of course, the external indexing of the array doesn't affect how you access the array internally; it's still indexed from zero here.

The destructor only needs the extra template parameter:

```

template <typename T, int startIndex>
inline Array<T, startIndex>::~~Array()
{
    delete[] elements;
}

```

The template definition for the non-const subscript operator function now becomes:

```
template <typename T, int startIndex>
T& Array<T, startIndex>::operator[](int index)
{
    if (index > startIndex + static_cast<int>(size) - 1)
        throw std::out_of_range {"Index too large: " + std::to_string(index)};

    if(index < startIndex)
        throw std::out_of_range {"Index too small: " + std::to_string(index)};

    return elements[index - startIndex];
}
```

Significant changes have been made here. The index parameter is of type `int` to allow negative values. The validity checks on the index value now verify that it's between the limits determined by the non-type template parameter and the number of elements in the array. Index values can only be from `startIndex` to `startIndex+size-1`. Because `size_t` is usually an unsigned integer type you must explicitly cast it to `int`; if you don't, the other values in the expression will be implicitly converted to `size_t`, which will produce a wrong result if `startIndex` is negative. The choice of message for the exception and the expression selecting it has also been changed.

The const version of the subscript operator function changes in a similar fashion:

```
template <typename T, int startIndex>
const T& Array<T, startIndex>::operator[](int index) const
{
    if (index > startIndex + static_cast<int>(size) - 1)
        throw std::out_of_range {"Index too large: " + std::to_string(index)};

    if(index < startIndex)
        throw std::out_of_range {"Index too small: " + std::to_string(index)};

    return elements[index - startIndex];
}
```

Finally, you need to alter the template for the assignment operator, but only the template parameter list and the template ID that qualifies the operator name need to be modified:

```
template <typename T, int startIndex>
Array<T, startIndex>& Array<T, startIndex>::operator=(const Array& rhs)
{
    if (&rhs != this)                                // If lhs != rhs...
    {                                                  // ...do the assignment...
        if (elements)                                // If lhs array exists
            delete[] elements;                        // release the free store memory

        size = rhs.size;
        elements = new T[rhs.size];
        for (size_t i {}; i < size; ++i)
            elements[i] = rhs.elements[i];
    }
    return *this;                                     // ... return lhs
}
```


There are restrictions on how you use a non-type parameter within a template. In particular, you must not modify the value of a parameter within the template definition. Consequently, a non-type parameter cannot be used on the left of an assignment or have the increment or decrement operator applied to it — in other words, it's treated as a constant. All parameters in a class template must always be specified to create an instance, unless there are default values for them. I'll discuss the use of default argument values for class template parameters later in the chapter.

In spite of the shortcomings of the Array template with a non-type parameter, let's see it in action in a working example. You just need to assemble the definitions for the function member templates into a header file together with the Array template definition with the non-type parameter. The following example will exercise the new features using Box.h from Ex16_01:

```
// Ex16_02.cpp
// Using a class template with a non-type parameter
#include "Box.h"
#include "Array.h"
#include <iostream>
#include <iomanip>

int main()
try
{
    try
    {
        const size_t size {21};                // Number of array elements
        const int start {-10};                 // Index for first element
        const int end {start + static_cast<int>(size) - 1}; // Index for last element

        Array<double, start> values {size};      // Define array of double values

        for (int i {start}; i <= end; ++i)      // Initialize the elements
            values[i] = i - start + 1;

        std::cout << "Sums of pairs of elements: ";
        size_t lines {};
        for (int i {end}; i >= start; --i)
            std::cout << (lines++ % 5 == 0 ? "\n" : "")
                << std::setw(5) << values[i] + values[i - 1];
    }
    catch (const std::out_of_range& ex)
    {
        std::cerr << "\nout_of_range exception object caught! " << ex.what() << std::endl;
    }

    const int start {};
    const size_t size {11};

    Array<Box, start - 5> boxes {size};        // Create array of Box objects

    for (int i {start - 5}; i <= start + static_cast<int>(size) - 5; ++i)
        std::cout << "Box[" << i << "] volume is " << boxes[i].volume() << std::endl;
}
```

```

catch (const std::exception& ex)
{
    std::cerr << typeid(ex).name() << " exception caught in main()! "
               << ex.what() << std::endl;
}

```

This displays the following output:

```

Sums of pairs of elements:
  41  39  37  35  33
  31  29  27  25  23
  21  19  17  15  13
  11   9   7   5   3
out_of_range exception object caught! Index too small: -11
Box[-5] volume is 1
Box[-4] volume is 1
Box[-3] volume is 1
Box[-2] volume is 1
Box[-1] volume is 1
Box[0] volume is 1
Box[1] volume is 1
Box[2] volume is 1
Box[3] volume is 1
Box[4] volume is 1
Box[5] volume is 1
class std::out_of_range exception caught in main()! Index too large: 6

```

The body of `main()` is a try block that catches any uncaught exceptions that have `std::exception` as a base class so `std::bad_alloc` will be caught by this. The nested try block, starts by defining constants that specify the range of index values and the size of the array. The size and start variables are used to create an instance of the `Array` template to store 21 values of type `double`. The second template argument corresponds to the non-type parameter and specifies the lower limit for the index values of the array. The size of the array is specified by the constructor argument.

The for loop that follows assigns values to the elements of the `values` object. The loop index, `i`, runs from the lower limit `start`, which will be `-10`, up to and including the upper limit `end`, which will be `+10`. Within the loop the values of the array elements are set to run from 1 to 21.

Next the sums of pairs of successive elements are output starting at the last array element and counting down. The `lines` variable is used to output the sums five to a line. As in the earlier example, sloppy control of the index value results in the expression `values[i-1]` causing an `out_of_range` exception to be thrown. The handler for the nested try block catches it and displays the message you see in the output.

The statement that creates an array to store `Box` objects is in the outer try block that is the body of `main()`. The type for `boxes` is `Array<Box, start-5>`, which demonstrates that expressions are acceptable as argument values for non-type parameters in a template instantiation. Such an expression must either evaluate to a value that has the type of the parameter, or it must be possible to convert the result to the appropriate type by means of an implicit conversion. You need to take care if such an expression includes the `>` character. Here's an example:

```
Array<Box, start > 5 ? start : 5> boxes;    // Will not compile!
```

The intent of the expression for the second argument that uses the conditional operator is to supply a value of at least 5, but as it stands, this won't compile. The `>` in the expression is paired with the opening angled bracket, and closes the parameter list. Parentheses are necessary to make the statement valid:

```
Array<Box, (start > 5 ? start : 5)> boxes; // OK
```

Parentheses are also likely to be necessary for expressions for non-type parameters that involve the arrow operator (`->`), or the shift right operator (`>>`).

The next for loop throws another exception, this time because the index exceeds the upper limit. The exception is caught by the catch block for the body of `main()`. The parameter is a reference to the base class and the output shows that the exception is identified as type `std::out_of_range`, thus demonstrating there is no object slicing occurring with a reference parameter. There's a significant difference between the ways the two exceptions were caught. Catching the exception in a catch block for the body of `main()` means that the program ends at this point. The previous exception was caught inside the body of `main()` in the catch block for the nested try block so it was possible to allow program execution to continue.

You must always keep in mind that non-type parameter arguments in a class template are part of the type of an instance of the template. Every unique combination of template arguments produces another class type. As I indicated earlier, the usefulness of the `Array<T, int>` template is very restricted compared to the original. You can't assign an array of ten values of a given type to another array of ten values of the same type if the starting indexes for the arrays are different — the arrays will be of different types. The class template with an extra data member and an extra constructor parameter is much more effective. You should always think twice about using non-type parameters in a class template to be sure that they're really necessary. Often you'll be able to use an alternative approach that will provide a more flexible template and more efficient code.

Arguments for Non-Type Parameters

An argument for a non-type parameter that is not a reference or a pointer must be a compile-time constant expression. This means that you can't use an expression containing a non-`const` integer variable as an argument, which is a slight disadvantage, but the compiler will validate the argument, which is a compensating plus. For example, the following statements won't compile:

```
int start {-10};
Array<double, start> values(21);           // Won't compile because start is not const
```

The compiler will generate a message to the effect that the second argument here is invalid. Here are correct versions of these two statements:

```
const int start {-10};
Array<double, start> values(21);           // OK
```

Now that `start` has been declared as `const`, the compiler can rely on its value, and both template arguments are now legal. The compiler applies standard conversions to arguments when they are necessary to match the parameter type. For example, if you had a non-type parameter declared as type `const size_t`, the compiler converts an integer literal such as 10 to the required argument type.

Pointers and Arrays as Non-Type Parameters

The argument for a non-type parameter that is a pointer must be an address, but it can't be any old address. It must be the address of an object or function with external linkage; so for example, you can't use addresses of array elements or addresses of non-static class members as arguments. This also means that if a non-type parameter is of type `const char*`, you can't use a string literal as an argument when you instantiate the template. If you want to use a string literal as an argument in this case, you must initialize a pointer variable with the address of the string literal, and pass the pointer as the template argument.

Because a pointer is a legal non-type template parameter, you can specify an array as a parameter, but an array and a pointer are not always interchangeable when supplying arguments to a template. For example, you could define a template as follows:

```
template <long* numbers>
class MyClass
{
    // Template definition...
};
```

You can now create instances of this template with the following code:

```
long data[10];                // Global
long* pData {data};          // Global

MyClass<pData> values;
MyClass<data> values;
```

Either an array name or a pointer of the appropriate type can be used as an argument corresponding to a parameter that is a pointer. However, the converse is not the case. Suppose that you have defined this template:

```
template <long numbers[10]>
class AnotherClass
{
    // Template definition...
};
```

The parameter is an array with 10 elements of type long, and the argument must be of the same type. In this case, you can use the data array defined earlier as the template argument:

```
AnotherClass<data> numbers;    // OK
```

However, you *can't* use a pointer, so the following won't compile:

```
AnotherClass<pData> numbers;   // Not allowed!
```

The reason is that an array type is quite different from a pointer type; an array name by itself represents an address but it is not a pointer and cannot be modified in the way that a pointer can.

Default Values for Template Parameters

You can supply default argument values for both type and non-type parameters in a class template. This works in a similar way to default values for function parameters - if a given parameter has a default value, then all subsequent parameters in the list must also have default values specified. If you omit an argument for a template parameter that has a default value specified, the default is used, just like with default parameter values in a function. Similarly, when you omit the argument for a given parameter in the list, all subsequent arguments must also be omitted.

The default values for class template parameters are written in the same way as defaults for function parameters — following an = after the parameter name. You could supply defaults for both the parameters in the version of the Array template with a non-type parameter. Here’s an example:

```
template < typename T = int, int startIndex = 0>
class Array
{
    // Template definition as before...
};
```

You don’t need to specify the default values in the templates for the member functions; the compiler will use the argument values used to instantiate the class template.

You could omit all the template arguments to declare an array of elements of type `int` indexed from 0.

```
Array<> numbers {101};
```

The legal index values run from 0 to 100, as determined by the default value for the non-type template parameter and the argument to the constructor. You must still supply the angled brackets, even though no arguments are necessary. The other possibilities open to you are to omit the second argument or to supply them all, as shown here:

```
Array<string, -100> messages {200};           // Array of 200 string objects indexed from -100
Array<Box> boxes {101};                       // Array of 101 Box objects indexed from 0
```

If a class template has default values for any of its parameters, they only need to be specified in the first declaration of the template in a source file, which usually will be the definition of the class template.

Explicit Template Instantiation

So far in this chapter, instances of a class template have been created *implicitly* as a result of defining a variable of a template type. You can also *explicitly* instantiate a class template without defining an object of the template type. The effect of an explicit instantiation of a template is that the compiler creates the instance determined by the parameter values that you specify.

You have already seen how to explicitly instantiate *function* templates back in Chapter 8. To instantiate a class template, just use the `template` keyword followed by the template class name and the template arguments between angled brackets. This statement explicitly creates an instance of the Array template:

```
template class Array<double, 1>;
```

This creates an instance of the template that stores values of type `double`, indexed from 1. Explicitly instantiating a class template generates the class type definition and it instantiates all of the function members of the class from their templates. This happens regardless of whether you call the function members so the executable may contain code that is never used.

Special Cases

You’ll encounter many situations where a class template definition won’t be satisfactory for every conceivable argument type. For example, you can compare `string` objects by using overloaded comparison operators, but you can’t do this with null-terminated strings. If a class template compares objects using the comparison operators, it will work for type `string` but not for type `char*`. To compare objects of type `char*`, you need to use the comparison functions that are declared in the `cstring` header.

To deal with this sort of problem, you have two options. The first possibility is to avoid the problem. You can use `static_assert()` that you met way back in Chapter 10 to test one or more of the type arguments. The second possibility is to define a *class template specialization*, which provides a class definition that is specific to a given set of arguments for the template parameters. I'll explain how you use `static_assert()` first.

Using `static_assert()` in a Class Template

You can use `static_assert()` to cause the compiler to output a message and fail compilation when a type argument in a class template is not appropriate. `static_assert()` has two arguments; when the first argument is false, the compiler outputs the message specified by the second argument. To protect against misuse of a class template, the first argument to `static_assert()` will use one or more of the templates from the `type_traits` header. These test the properties of types and classify types in various ways. The templates in `type_traits` have the `std::integral_constant` template as a base, which defines a static member, `value`, which is a constant expression that is implicitly convertible to type `bool`. There are a lot of templates in the `type_traits` header so I'll just mention a few in Table 16-1 to give you an idea of the possibilities, and leave you to explore the rest in your Standard Library documentation. These templates are all defined in the `std` namespace.

Table 16-1.

Template	Result
<code>is_default_constructible<T></code>	The value member is only true if type <code>T</code> is default constructible, which means for a class type that the class has a no-arg constructor.
<code>is_copy_constructible<T></code>	The value member is true if type <code>T</code> is copy constructible, which means for a class type that the class has a copy constructor.
<code>is_assignable<T></code>	The value member is true if type <code>T</code> is assignable, which means for a class type that it has an assignment operator function.
<code>is_pointer<T></code>	The value member is true if type <code>T</code> is a pointer type and false otherwise.
<code>is_null_pointer<T></code>	The value member is true only if type <code>T</code> is of type <code>std::nullptr_t</code> .
<code>is_class<T></code>	The value member is true only if type <code>T</code> is a class type.

It's easy to get confused about what is happening with these templates. Keep in mind that these templates are relevant at *compile time*. A template such as `is_assignable<T>` will be compiled each time an instance of a template that uses it in a `static_assert()` is instantiated. The result therefore relates to the argument that was used to instantiate the template so the test applies to the template type argument. An example should make it clear how you use these.

Let's amend Ex16_01 to show this in operation. First, comment out the line in the `Box` class definition in `Box.h` that generates the default constructor:

```
class Box
{
protected:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv} {}
    // Box() = default;
    double volume() const { return length*width*height; }
};
```

Next, add an `#include` directive for the `type_traits` header to `Array.h` and add one statement following the opening brace in the body of the `Array` template:

```
#include <stdexcept>                // For standard exception types
#include <string>                    // For to_string()
#include <type_traits>

template <typename T>
class Array
{
    static_assert(std::is_default_constructible<T>::value, "A default constructor is required.");

    // Rest of the template as before...
};
```

You can now recompile the example, which will fail of course. The first argument to `static_assert()` is the value member of the instance of `is_default_constructible<T>` for the current type argument for `T`. When this is type `Box`, the value member of the template will be `false`, triggering the message you'll see in the output from your compiler and the compilation will fail. Removing the commenting out of the `Box` default constructor will allow the compilation to succeed. The complete example is in the code download as `Ex16_03`.

Defining a Class Template Specialization

A class template specialization is a class definition, not a class template. Instead of using the template to generate the class from the template for a particular type, `char*` say, the compiler uses the specialization you define for that type instead. Thus a class template specialization provides a way to predefine instances of a class template to be used by the compiler for specific sets of argument for the template parameters.

Suppose it was necessary to create a specialization of the first version of the `Array` template for type `char*`. You'd write the specialization of the class template definition as follows:

```
template <>
class Array<char*>
{
    // Definition of a class to suit type char*...
};
```

This definition of the specialization of the `Array` template for type `char*` must be preceded by the original template definition, or by a declaration for the original template. Because all the parameters are specified, it is called a *complete specialization* of the template, which is why the set of angle brackets following the `template` keyword are empty. The compiler will always use a class definition when it is available, so there's no need for the compiler to consider instantiating the template for type `char*`.

It may be that just one or two function members of a class template need to have code specific to a particular type. If the function members are defined by separate templates outside the class template, rather than within the body of the class template, you can just provide specializations for the function templates that need to be different.

Partial Template Specialization

If you were specializing the version of the template with two parameters, you may only want to specify the type parameter for the specialization, leaving the non-type parameter open. You could do this with a *partial specialization* of the Array template that you could define like this:

```
template <int start>                // Because there is a parameter...
class Array<char*, start>          // This is a partial specialization...
{
    // Definition to suit type char*...
};
```

This specialization of the original template is also a template. The parameter list following the `template` keyword must contain the parameters that need to be specified for an instance of this template specialization — just one in this case. The first parameter is omitted because it is now fixed. The angled brackets following the template name specify how the parameters in the original template definition are specialized. The list here must have the same number of parameters as appear in the original, unspecialized template. The first parameter for this specialization is `char*`. The other parameter is specified as the corresponding parameter name in this template.

Apart from the special considerations you might need to give to a template instance produced by using `char*` for a type parameter, it may well be that pointers in general are a specialized subset that need to be treated differently from objects and references. For example, to compare objects when a template is instantiated using a pointer type, pointers must be dereferenced, otherwise you are just comparing addresses, not the objects or values stored at those addresses.

For this situation, you can define another partial specialization of the template. The parameter is not completely fixed in this case, but it must fit within a particular pattern that you specify in the list following the template name. For example, a partial specialization of the Array template for pointers would look like this:

```
template <typename T, long start>
class Array<T*, start>
{
    // Definition to suit pointer types other than char*...
};
```

The first parameter is still `T`, but the `T*` between angled brackets following the template name indicates that this definition is to be used for instances where `T` is specified as a pointer type. The other two parameters are still completely variable, so this specialization will apply to any instance where the first template argument is a pointer.

Choosing between Multiple Partial Specializations

Suppose both the partial specializations of the Array template that I just discussed were defined — the one for type `char*`, and the one for any pointer type. How can you be sure that the version for type `char*` is selected by the compiler when this is appropriate for any particular instantiation? For example, consider this declaration:

```
Array<Box*, -5> boxes {11};
```

Clearly, this only fits with the specialization for pointers in general, but both partial specializations fit the declaration if you write this:

```
Array<char*, 1> messages {100};
```


In this case, the compiler determines that the `char*` partial specialization is a better fit because it is more specialized than the alternative. The partially specialized template for `char*` is determined to be more specialized than the specialization for pointers in general because although anything that selects the `char*` specialization — which happens to be just `char*` — also selects the `T*` specialization, the reverse is not the case.

One specialization is more specialized than another when every argument that matches the given specialization matches the other, but the reverse is not true. Thus you can consider a set of specializations for a template to be ordered from most specialized to least specialized. When several template specializations may fit a given declaration, the compiler will select and apply the most specialized specialization from these.

Friends of Class Templates

Because a class can have friends, you won't be surprised to learn that a class template can also have friends. Friends of a class template can be classes, functions, or other templates. If a class is a friend of a class template, then all its function members are friends of every instance of the template. A function that is a friend of a template is a friend of any instance of the template, as shown in Figure 16-4.

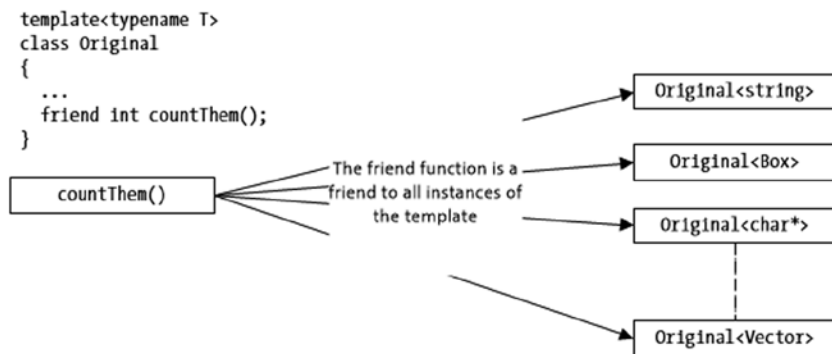


Figure 16-4. A friend function of a class template

Templates that are friends of a template are a little different. Because they have parameters, the parameter list for the template class usually contains all the parameters to define the friend template. This is necessary to identify the instance of the friend template that is the friend of the particular instance of the original class template. However, the function template for the friend is only instantiated when you use it in your code. In the Figure 16-5, `getBest()` is a function template.

```
template<typename T>
class Original
{
    ...
    friend Original<T>* getBest(Original<T>* pObjects);
}
```

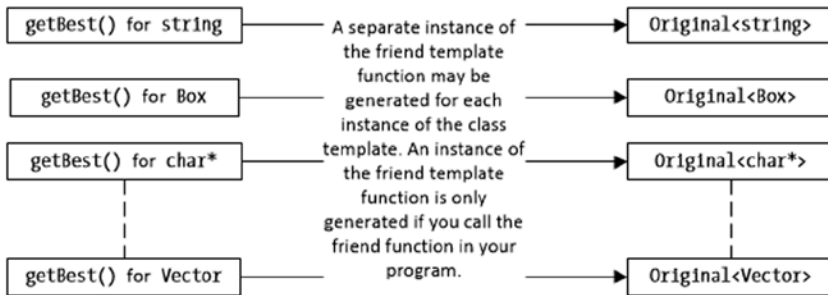


Figure 16-5. A function template that is a friend of a class template

Although each class template instance in Figure 16-5 could potentially have a unique friend template instance, this is not necessarily the case. If the class template has some parameters that aren't parameters of the friend template, then a single instance of the friend template may service several instances of the class template.

Note that an ordinary class may have a class template or a function template declared as a friend. In this case, all instances of the template are friends of the class. With the example in Figure 16-6, every member function of every instance of the Thing template is a friend of the Box class because the template has been declared as a friend of the class.

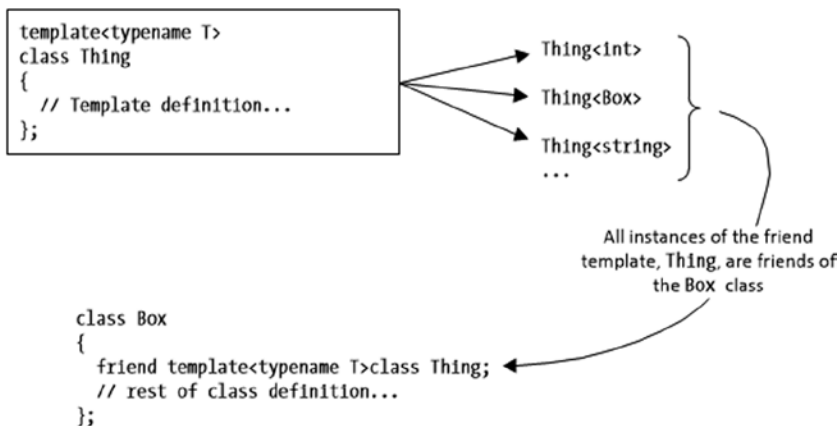


Figure 16-6. A class template that is a friend of a class

The declaration in the Box class is a *template* for a friend declaration and this effectively generates a friend declaration for each class generated from the Thing template. If there are no instances of the Thing template then the Box class has no friends.

Class Templates with Nested Classes

A class can contain another class nested inside its definition. A class template definition can also contain a nested class or even a *nested class template*. A nested class template is independently parameterized, so inside another class template it creates a two-dimensional ability to generate classes. Dealing with templates inside templates is outside the scope of this book, but I'll introduce aspects of a class template with a nested class.

Let's take a particular example. Suppose you want to implement a stack, which is a "last in, first out" storage mechanism. A stack is illustrated in Figure 16-7. It works in a similar way to a plate stack in a self-service restaurant. It has two basic operations. A *push operation* adds an item at the top of a stack and a *pop operation* removes the item at the top of the stack. Ideally a stack implementation should be able to store objects of any type so this is a natural for a template.

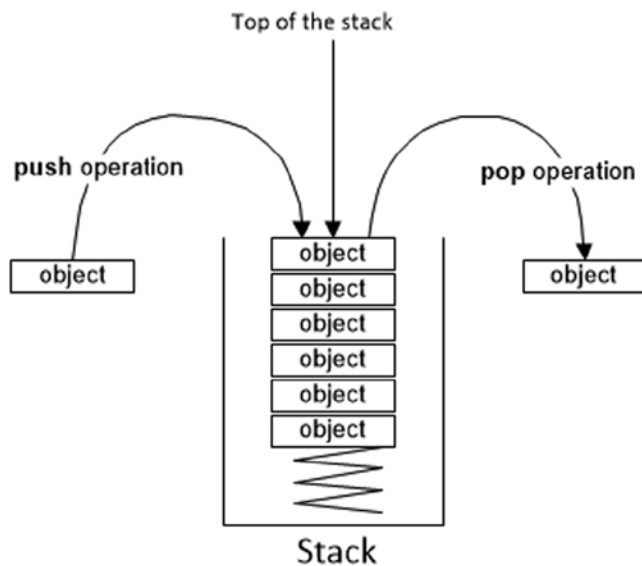


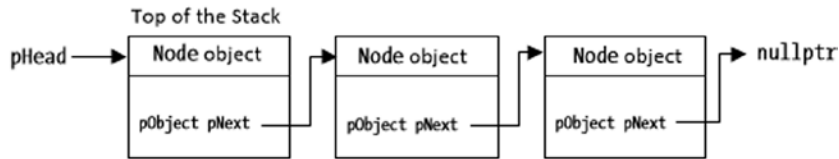
Figure 16-7. The concept of a stack

The parameter for a Stack template is a type parameter that specifies the type of objects in the stack, so the initial template definition is going to be:

```
template <typename T>
class Stack
{
    // Detail of the Stack definition...
};
```

If you want the stack's capacity to grow automatically, you can't use fixed storage for objects within the stack. One way of providing the ability to automatically grow and shrink the stack as objects are pushed on it or popped off it, is to implement the stack as a linked list. The nodes in the linked list can be created in the free store, and the stack only needs to remember the node at the top of the stack. This is illustrated in Figure 16-8.

A Stack object only needs to store a pointer to the node at the top.



In an empty stack, the pointer to the top of the stack will be null, because no nodes exist.

```
pHead = nullptr;
```

Figure 16-8. A stack as a linked list

When you create an empty stack, the pointer to the head of the list is `nullptr`, so you can use the fact that it doesn't contain any `Node` objects as an indicator that the stack is empty. Of course, only the `Stack` object needs access to the `Node` objects that are in the stack. The `Node` objects are just internal objects used to encapsulate the objects that are stored in the stack so there's no need for anyone outside the `Stack` class to know that type `Node` exists.

A nested class that defines nodes in the list is required in each instance of the `Stack` template, and because a node must hold an object of type `T`, the `Stack` template parameter type, you can define it as a nested class in terms of `T`. We can add this to the initial outline of the `Stack` template:

```
template <typename T>
class Stack
{
private:
    // Nested class
    class Node
    {
    public:
        T* pItem {};                // Pointer to object stored
        Node* pNext {};             // Pointer to next node

        Node(T& item) : pItem {&item} {}    // Create a node from an object
    };

    // Rest of the Stack class definition...
};
```

The `Node` class is declared as `private` so we can afford to make all its members `public` so that they're directly accessible from function members of the `Stack` template. Objects of type `T` are the responsibility of the user of the `Stack` class so only a pointer to an object of type `T` is stored in a `Node` object. The constructor is used when an object

is pushed onto the stack. The parameter to the constructor is a reference to an object of type `T` and the address of this object is used to initialize the `pItem` member of the new `Node` object. The rest of the `Stack` class template to support the linked list of `Node` objects shown in Figure 16-8 is:

```
template <typename T>
class Stack
{
private:
    // Nested Node class definition as before...

    Node* pHead {};                // Points to the top of the stack
    void copy(const Stack& stack);  // Helper to copy a stack
    void freeMemory();              // Helper to release free store memory

public:
    Stack() = default;              // Default constructor
    Stack(const Stack& stack);       // Copy constructor
    ~Stack();                       // Destructor
    Stack& operator=(const Stack& stack); // Assignment operator

    void push(T& item);              // Push an object onto the stack
    T& pop();                       // Pop an object off the stack
    bool isEmpty() {return !pHead;} // Empty test
};
```

As I explained earlier, a `Stack` object only needs to “remember” the top node so it has only one data member, `pHead`, of type `Node*`. There’s a default constructor, a copy constructor, a destructor, and an assignment operator function. The destructor is essential because nodes will be created dynamically using `new` and the addresses stored in raw pointers. The `push()` and `pop()` members transfer objects to and from the stack and the `isEmpty()` function returns true if the `Stack` object is empty. The private `copy()` member of the `Stack` class will be used internally in the copy constructor and the assignment operator function to carry out operations that are common to both; this will reduce the number of lines of code necessary and the size of the executable. Similarly, the private `freeMemory()` function is a helper that will be used by the destructor and the assignment operator function. We just need the templates for the member functions of the `Stack` template to complete the implementation.

Function Templates for Stack Members

I’ll start with the definition of the template for the `copy()` member:

```
template <typename T>
void Stack<T>::copy(const Stack& stack)
{
    if(stack.pHead)
    {
        pHead = new Node {*stack.pHead}; // Copy the top node of the original
        Node* pOldNode {stack.pHead};    // Points to the top node of the original
        Node* pNewNode {pHead};           // Points to the node in the new stack
    }
}
```

```

while(pOldNode = pOldNode->pNext)           // If it's nullptr, it's the last node
{
    pNewNode->pNext = new Node {*pOldNode}; // Duplicate it
    pNewNode = pNewNode->pNext;           // Move to the node just created
}
}
}

```

This copies the stack represented by the stack argument to the current Stack object, which is assumed to be empty. It does this by replicating pHead for the argument object, then walking through the sequence of Node objects, copying them one by one. The process ends when the Node object with a null pNext member has been copied.

The freeMemory() helper function will release the heap memory for all Node objects belonging to the current Stack object:

```

template <typename T>
void Stack<T>::freeMemory()
{
    Node* pTemp {};
    while(pHead)
    {
        pTemp = pHead->pNext; // While current pointer is not null
        delete pHead;         // Get the pointer to the next
        pHead = pTemp;        // Delete the current
    }                         // Make the next current
}

```

This uses a temporary pointer to a Node object to hold the address stored in the pNext member of a Node object before the object is deleted from the free store. At the end of the while loop, all Node objects belonging to the current Stack object will have been deleted and pHead will be nullptr.

The default constructor is defined within the template as the default that the compiler should supply. The copy constructor must replicate a Stack<T> object, which can be done by walking through the nodes and copying them, which is exactly what the copy() function does. This makes the template for the copy constructor trivial:

```

template <typename T>
Stack<T>::Stack(const Stack& stack)
{
    copy(stack);
}

```

The assignment operator is similar to the copy constructor, but two extra things must be done. First, the function must check to see whether or not the objects involved are identical. Second, it must release memory for nodes in the object on the left of the assignment before copying the object that is the right operand. Here's the template to define the assignment operator so that it works in this way:

```

template <typename T>
Stack<T>& Stack<T>::operator=(const Stack& stack)
{
    if (this != &stack) // If objects are not identical
    {
        freeMemory(); // Release memory for nodes in lhs
        copy(stack);   // Copy rhs to lhs
    }
    return *this // Return the left object
}

```

Using the helper functions, the implementation becomes very simple. If the operands are the same, it's only necessary to return the object. If the objects are different, the first step is to delete all the nodes for the left-hand object, which is done by calling `freeMemory()`, then replace them with copies of the nodes from the right operand by calling `copy()`.

The code for the destructor template just needs to call `freeMemory()`:

```
template <typename T>
Stack<T>::~~Stack()
{
    freeMemory();
}
```

The template for the `push()` operation is very easy:

```
template <typename T>
void Stack<T>::push(T& item)
{
    Node* pNode {new Node(item)};           // Create the new node
    pNode->pNext = pHead;                   // Point to the old top node
    pHead = pNode;                         // Make the new node the top
}
```

The `Node` object encapsulating `item` is created by passing the reference to the `Node` constructor. The `pNext` member of the new node needs to point to the node that was previously at the top. The new `Node` object then becomes the top of the stack so its address is stored in `pHead`.

The `pop()` operation is slightly more work because you must delete the top node:

```
template <typename T>
T& Stack<T>::pop()
{
    T* pItem {pHead->pItem};               // Get pointer to the top node object
    if(!pItem)                             // If it's empty
        throw std::logic_error {"Stack empty"}; // Pop is not valid so throw exception

    Node* pTemp {pHead};                   // Save address of top node
    pHead = pHead->pNext;                   // Make next node the top
    delete pTemp;                          // Delete the previous top node
    return *pItem;                         // Return the top object
}
```

It is possible that there could be a `pop` operation on an empty stack. Because the function returns a reference, you can't signal an error through the return value, so you have to throw an exception in this case. After storing the pointer to the object in the top node in the local `pItem` variable, the function deletes the top node, promotes the next node to the top, and returns a reference to the object.

That's all the templates you need to define the stack. If you gather all the templates into a header file, `Stacks.h`, you can try it out with the following code:

```
// Ex16_04.cpp
// Using a stack defined by nested class templates
#include "Stacks.h"
#include <iostream>
#include <string>
using std::string;

int main()
{
    const char* words[] {"The", "quick", "brown", "fox", "jumps"};
    Stack<const char*> wordStack;           // A stack of C-style strings

    for (size_t i {}; i < sizeof(words)/sizeof(words[0]) ; ++i)
        wordStack.push(words[i]);

    Stack<const char*> newStack {wordStack}; // Create a copy of the stack

    // Display the words in reverse order
    while(!newStack.isEmpty())
        std::cout << newStack.pop() << " ";
    std::cout << std::endl;

    // Reverse wordStack onto newStack
    while(!wordStack.isEmpty())
        newStack.push(wordStack.pop());

    // Display the words in original order
    while(!newStack.isEmpty())
        std::cout << newStack.pop() << " ";
    std::cout << std::endl;

    std::cout << std::endl << "Enter a line of text:" << std::endl;
    string text;
    std::getline(std::cin, text);           // Read a line into the string object

    Stack<const char> characters;           // A stack for characters

    for (size_t i {}; i < text.length(); ++i)
        characters.push(text[i]);          // Push the string characters onto the stack

    std::cout << std::endl;
    while(!characters.isEmpty())
        std::cout << characters.pop();      // Pop the characters off the stack

    std::cout << std::endl;
}
```


Here's an example of the output:

```
jumps fox brown quick The
The quick brown fox jumps
```

Enter a line of text:

Never test for errors that you don't know how to handle.

```
.eldnah ot woh wonk t'nod uoy taht srorre rof tset reveN
```

You first define an array of five objects that are null-terminated strings, initialized with the words shown. Then you define an empty `Stack` object that can store `const char*` objects. The `for` loop then pushes the array elements onto the stack. The first word from the array will be at the bottom of the `wordStack` stack and the last word at the top. You create a copy of `wordStack` as `newStack` to exercise the copy constructor.

In the next `while` loop, you display the words in `newStack` in reverse order by popping them off the stack and outputting them in a `while` loop. The loop continues until `isEmpty()` returns `false`. Using the `isEmpty()` function member is a safe way of getting the complete contents of a stack. `newStack` is empty by the end of the loop, but you still have the original in `wordStack`.

The next `while` loop retrieves the words from `wordStack` and pops them onto `newStack`. The `pop` and `push` operations are combined in a single statement, where the object returned by `pop()` for `wordStack` is the argument for `push()` for `newStack()`. At the end of this loop, `wordStack` is empty and `newStack` contains the words in their original sequence — with the first word at the top of the stack. You then output the words by popping them off `newStack`, so at the end of this loop, both stacks are empty:

The next part of `main()` reads a line of text into a string object using the `getline()` function and then creates a stack to store characters:

```
Stack<const char> characters;           // A stack for characters
```

This creates a new instance of the `Stack` template, `Stack<const char>`, and a new instance of the constructor for this type of stack. At this point, the program contains two classes from the `Stack` template each with a nested `Node` class.

You peel off the characters from `text` and push them onto the new stack in a `for` loop. The `length()` function of the `text` object is used to determine when the loop ends. Finally the input string is output in reverse by popping the characters off the stack. You can see from the output that my input was not even slightly palindromic, but you could try, “Ned, I am a maiden” or even “Are we not drawn onward, we few, drawn onward to new era.”

Summary

If you understand how class templates are defined and used, you'll find it easy to understand and apply the capabilities of the Standard Template Library. The ability to define class templates is also a powerful augmentation of the basic language facilities for defining classes. The essential points I've discussed in this chapter include:

- A class template defines a family of class types.
- An instance of a class template is a class definition that generated by the compiler from the template using a set of template arguments that you specify in your code.
- An implicit instantiation of a class template arises out of a definition for an object of a class template type.
- An explicit instantiation of a class template defines a class for a given set of arguments for the template parameters.

- An argument corresponding to a type parameter in a class template can be a fundamental type, a class type, a pointer type, or a reference type.
- The type of a non-type parameter can be an integral or enumeration type, a pointer type, or a reference type.
- A partial specialization of a class template defines a new template that is to be used for a specific, restricted subset of the arguments for the original template.
- A complete specialization of a class template defines a new template for a specific, complete set of parameter arguments for the original template.
- A friend of a class template can be a function, a class, a function template, or a class template.
- An ordinary class can declare a class template or a function template as a friend.

EXERCISES

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck after that, you can download the solutions from the Apress website (www.apress.com/source-code/), but that really should be a last resort.

- Exercise 16-1. Define a *template* for one-dimensional sparse arrays that will store objects of any type so that only the elements stored in the array occupy memory. The potential number of elements that can be stored by an instance of the template should be unlimited. The template might be used to define a sparse array containing pointers to elements of type `double` with the following statement:

```
SparseArray<double> values;
```

Define the subscript operator for the template so that element values can be retrieved and set just like in a normal array. If an element doesn't exist at an index position, the subscript operator should return an object created by the default constructor for the object class. Exercise the template with a `main()` function that stores 20 random element values of type `int` at random index positions within the range 32 to 212 in a sparse array with an index range from 0 to 499, and output the values of element that exist along with their index positions.

Exercise 16-2. Define a template for a linked list type that allows the list to be traversed backward from the end of the list, as well as forward from the beginning. Apply the template in a program that stores individual words from some arbitrary prose or poetry as `std::string` objects in a linked list, and then displays them five to a line in sequence and in reverse order.

Exercise 16-3. Use the linked list and sparse array templates to produce a program that stores words from a prose or poetry sample in a sparse array of up to 26 linked lists, where each list contains words that have the same initial letter. Output the words, starting each group with a given initial letter on a new line. (Remember to leave a space between successive `>` characters when specifying template arguments—otherwise, `>>` will be interpreted as a shift right operator.)

Exercise 16-4. Add an `insert()` function to the `SparseArray` template that adds an element following the last element in the array. Use this function and a `SparseArray` instance that has elements that are `SparseArray` objects storing `string` objects to perform the same task as the previous exercise.



File Input and Output

THE C++ LANGUAGE has no provision for input and output. The subject of this chapter is the input and output (I/O) capabilities that are available in the Standard Library, which provides support for device-independent input and output operations. You've used elements of these facilities to read from the keyboard and output to the screen in all the examples so far. In this chapter I'll expand on that and explain how you can read and write disk files. By the end of this chapter, you'll have learned:

- What a stream is
- What the standard streams are
- How binary streams differ from text streams
- How to create and use file streams
- How errors in stream operations are recorded, and how you can manage them
- How to use unformatted stream operations
- How to write numerical data to a file as binary data
- How objects can be written to and read from a stream
- How to overload the insertion and extraction operators for your classes
- How to create string streams

Input and Output in C++

You'll need many different kinds of I/O capabilities in your programs. An application might need to store and retrieve data in a database, to create application windows and display graphics on the screen, to communicate over a phone line or over a network. All of these examples have one thing in common. They're totally outside the remit of C++ and its Standard Library.

This implies that in the majority of situations you'll use I/O facilities that aren't part of C++, although they may well be provided as part of your C++ development environment. Of course, the capabilities that are provided in C++ are still very important; they represent a substantial Standard Library facility with extensive functionality. In addition to file I/O capability, they provide facilities for data formatting using string-based I/O.

Understanding Streams

The I/O functionality provided by the standard library involves using streams. A *stream* is an abstract representation of an input device or an output device that is a sequential source or destination for data in your program and of course, a stream is represented by a class type. You can visualize a stream as a sequence of bytes flowing between an external device and the main memory of your computer. You can write data to an *output stream* and read data from an *input stream*; some streams provide the capability for both input and output. Fundamentally, all input and output is a sequence of bytes being read from, or written to, some external device.

When you're reading data from an external device, it's up to you to interpret the data correctly. When you read bytes from an external source, the bytes could be a sequence of 8-bit characters, a sequence of UCS characters, binary values of various types, or a mixture of all of them. There's no way to tell from the data in the stream what it is. You have to know the structure and type of data in advance, and read and interpret it accordingly.

Data Transfer Modes

There are two modes for transferring data to and from a stream: *text mode* and *binary mode*. In *text mode*, the data is interpreted as a sequence of characters that is usually organized as one or more lines terminated by the newline character, '\n'. Numerical values are separated by one or more whitespace characters. In text mode, a stream may transform newline characters as they're read from or written to the physical device. Whether this occurs, and how characters are changed, is system dependent. On some systems such as Microsoft Windows, a single newline character is written to a stream as *two* characters: a carriage return and a line feed. When a carriage return and a line feed are read from a stream, they're mapped into a single character, '\n'. On other systems such as Unix, the newline will remain a single character. In binary mode, there are no transformations of data. The original bytes are transferred between memory and the stream without conversion.

Text Mode Operations

In text mode, you can read and write various types of data using the extraction and insertion operators, >> and <<, exactly as you've been doing throughout the book when reading from `std::cin` and writing to `std::cout`. These are *formatted I/O operations* that occur in *text mode*. Binary numerical data such as integers and floating-point values are converted to a character representation before they're written to the stream, and the inverse process occurs when numerical values are read. Of course, data written in text mode is still just a sequence of bytes in the file. You could still read it in binary mode if you wanted; whether the data written in text mode would make sense when it is read in binary mode is doubtful though.

Binary Mode Operations

In *binary mode* you read or write bytes, regardless of the type of data. A binary value of type `int` occupying four bytes in memory is written as four bytes, as is a 4-byte `float` value or a sequence of four characters of type `char`. This means that the data is recorded exactly as it is inside your computer, with none of the small errors that can creep in as a result of the conversions of floating-point data in text mode. Once the bytes have been written to the file, they are just bytes, there's no indication of what they were. Because you are reading bytes, how you interpret the bytes is up to you. You must know what type of data was written to a file in order to read it sensibly. There is nothing to prevent a sequence of four bytes from being read as a 4-byte integer, a `float` value, or as four characters.

A binary read or write operation can be for a single byte, a given number of bytes, or a sequence of bytes terminated by a delimiter of some kind. You always read or write a sequence of *bytes* exactly as they are in memory. These are *unformatted input/output operations* because data transferred between memory and the stream without modification. Data that you write to a stream may originate as any combination of character strings and binary numerical values of various types, but whatever they are, it's the bytes that make up the data values in memory that are written to the stream.

Advantages of Using Streams

The primary reason for using streams for I/O operations is to make the code for the operations independent of the physical device. This has a couple of advantages. First, you don't have to worry about the detailed mechanics of each device; that is all taken care of behind the scenes. Second, a program will work with a variety of disparate physical devices without necessitating changes to the source code.

The physical reality of an *output stream* — in other words, where the data goes when you write to it — can be any device to which a sequence of bytes can be transferred. File stream I/O will typically be to a file on a disk or on a solid state drive. The Standard Library defines three standard output stream objects, `cout`, `cerr`, and `clog`, all of which are typically associated with the display screen. `std::cout` is the standard output stream. `std::cerr` and `std::clog` are both connected to the *standard error stream*, which is used for error reporting. The difference between the last two streams is that `cerr` is unbuffered (so data is written immediately to the output device), whereas `clog` is buffered (so data will only be written when the buffer is full). Figure 17-1 shows some devices and the kinds of streams that they would represent.

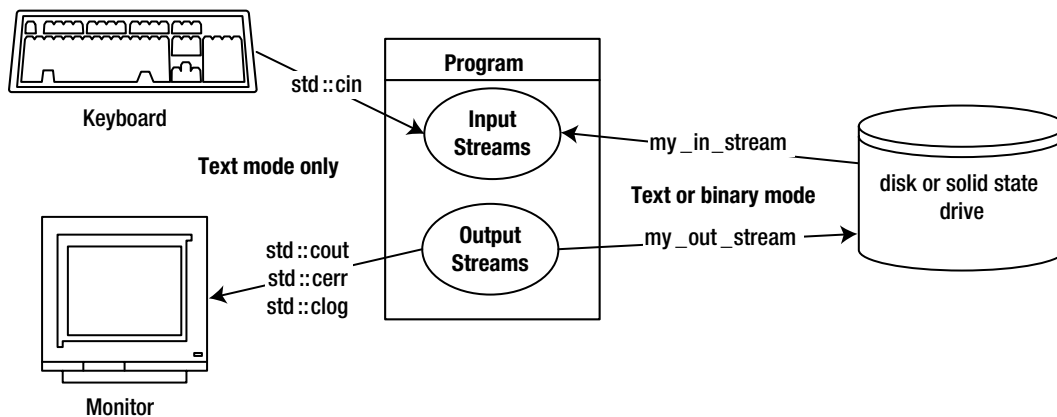


Figure 17-1. I/O devices and streams

In principle, an *input stream* can be any serial source of data, but it's typically a disk file or the keyboard. As I indicated earlier, streams are objects of class types, and the standard streams are predefined objects that are associated with specific external devices on your system. When you've been reading objects from `cin` using the extraction operator, `>>`, or writing objects to `cout` using the insertion operator, `<<`, you've been using overloaded versions of the operator `>>()` and operator `<<()` functions for these objects. For other sources or destinations for data, you define a stream object and associate it with a file on a particular device. Figure 17-1 shows `my_in_stream` as the object for reading from a file on disk and `my_out_stream` as the object for writing to a file. Let's look at the classes that represent streams.

■ **Caution** Writing files is a hazardous business and you run the examples from this chapter at your own risk. It's very easy to overwrite files that matter. To avoid this possibility, set up a separate directory (folder) on your system and use this for examples from this chapter and your solutions to exercises. Before executing any of the examples, make sure that file paths in the code correspond to the directory you have set up for the purpose.

The code in this chapter uses Microsoft Windows path specifications. I recommend that you change the file paths to suit your environment in any event. To encourage this I have used paths on the D: drive in the examples, which will usually be the DVD/CD drive unless you have a real or virtual second disk. Of course, the examples won't work with the DVD/CD drive.

Stream Classes

There are quite a few classes involved in stream I/O, and these are mostly defined by class templates. The main classes that I'll be discussing in this chapter and the relationships between them are illustrated in Figure 17-2. Figure 17-2 is a simplified representation, but it's all you need to understand the principles. `ios_base` is an "ordinary" class, and the others are instances of templates. The `istream` type for example, is an instance of the `basic_istream` template, `basic_istream<char>`, and the `ios` type is the instance of the `basic_ios` template, `basic_ios<char>`. The stream classes share a common base, `ios`, which defines flags that record the state of a stream and the formatting modes in effect. Thus, all the stream classes that provide the I/O operations that you'll be using share a common set of status and formatting flags, and the functions to query and set them.

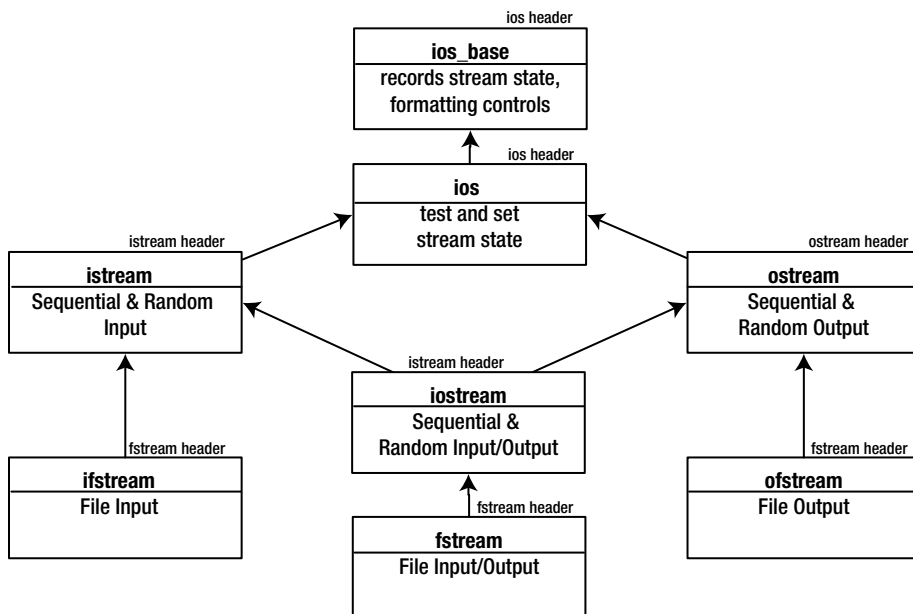


Figure 17-2. The main classes that represent streams

Figure 17-2 doesn't show the `streambuf` type that provides buffering in memory for stream I/O because this is an instance of standalone template type `basic_streambuf`, which is defined in the `streambuf` header; you don't need to concern yourself with this type - it is used internally by stream objects as necessary.

The `iostream` header just defines the standard stream objects but to do this it includes the `ios`, `istream`, `ostream`, and `streambuf` headers so you get all these when you include `iostream` into a source file. The standard input stream `cin` is an object of type `istream`, and the standard output streams `cout`, `cerr`, and `clog` are `ostream` objects. You can see that the stream classes for file handling, `ifstream`, `fstream`, and `ofstream`, all have `istream`, `ostream`, or both as base classes, so the facilities that you've been using with the standard streams `cin` and `cout` are going to be available with file streams too. Note that `fstream` is derived from `iostream` and does not have `ifstream` or `ofstream` as a base class; thus the classes for file streams are completely independent of one another.

The class templates that generate the stream classes generally have type parameters that specify the character sets for a particular stream, and the class names in Figure 17-2 apply to streams that deal with characters of type `char`. The names in Figure 17-2 are aliases for predefined instances of these templates. There are also aliases for predefined template instances for streams that handle characters of type `wchar_t`, called `wistream`, `wostream`, `wiostream`, `wifstream`, `wofstream`, and `wfstream`. I won't discuss these wide-character stream classes specifically in this chapter, but they work in the same way as the byte stream classes.

You'll sometimes see references to the original templates rather than the aliases that appear in Figure 17-2 so you need to be aware of the template names. The aliases for the stream class template are defined by these typedefs:

```
typedef basic_ios<char>      ios;
typedef basic_istream<char> istream;
typedef basic_ostream<char> ostream;
typedef basic_iostream<char> iostream;
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
typedef basic_fstream<char> fstream;
```

The corresponding aliases for wide-character streams have `wchar_t` as the type argument, in place of `char`. I'll use the type aliases in the rest of this chapter rather than the full template names, as they involve considerably less typing!

Standard Stream Objects

The standard streams are defined in the `iostream` header as objects within the `std` namespace like this:

```
extern istream cin;
extern ostream cout;
extern ostream cerr;
extern ostream clog;
```

The `iostream` header file also defines the equivalent wide-character stream objects:

```
extern wistream wcin;
extern wostream wcout;
extern wostream wcerr;
extern wostream wclog;
```

You've already made extensive use of the standard input stream `cin` and the standard output stream `cout`. The `cerr` and `clog` streams are used in exactly the same way as `cout`. I won't repeat what I covered in previous chapters about reading from and writing to the standard streams. I'll concentrate on the background to how they work; the same techniques and mechanisms apply to other stream types. To begin with, I'll revisit the formatted stream operations that you're familiar with, and show how you can use these with files. I'll then delve into unformatted stream operations and how and when you can use those to your advantage.

Stream Insertion and Extraction Operations

The insertion and extraction operators that you've been using with standard stream objects work just the same with other types of stream objects. As I indicated in Figure 17-1, the standard streams operate only in text mode, because the data source and destination are essentially character-based. The insertion and extraction operators are principally concerned with converting between internal binary representations of data and their external character representations. When you use these operators with streams other than the standard streams, you only use them in text mode because they work with a character-based representation of the data. Text mode is concerned with the visual presentation of the data, which can be an approximation of the binary data when it is numeric.

Stream Extraction Operations

The `operator>>()` function is implemented as a set of templates for function members of the `istream` class to support reading the following types of data from any input stream:

<code>short</code>	<code>int</code>	<code>long</code>	<code>long long</code>
<code>unsigned</code>	<code>unsigned int</code>	<code>unsigned long</code>	<code>unsigned long long</code>
<code>float</code>	<code>double</code>	<code>long double</code>	
<code>bool</code>	<code>void*</code>		

The `operator>>()` function that supports the `void*` type enables you to read address values into a pointer of any type, *except* for a pointer to type `char`, which refers to a null-terminated string and is treated as a special case. Overloads of `operator<<()` are defined by non-member templates to read characters into variables of types `char`, `signed char`, and `unsigned char` and to read character sequences that do not contain whitespace as C-style strings into arrays of type `char*`, `signed char*`, and `unsigned char*`.

Let's revisit how the code you've been writing connects to these operator functions by considering what happens when you write statements such as these:

```
int i {};  
double x {};  
std::cin >> i >> x;
```

`std::cin` is an object of type `istream` so the last statement that reads values for the two variables from the stream translates to this:

```
(std::cin.operator>>(i)).operator>>(x);
```

The `operator>>()` function is called once for each use of the extraction operator. The function returns a reference to the stream object for which it was called (in this case `std::cin`), so you can use the return value to call the next operator function. The parameter for `operator>>()` has to be a reference to allow the function to store the data value that it reads from the stream in the variable that is passed as the argument.

Whitespace characters are regarded as delimiters between values and ignored, so you can't read whitespace characters using any of the `operator>>()` functions for the `istream` class. You'll recall that you used the `std::getline()` function for `cin` when you wanted to read a line of text that includes whitespace; you can use this in the same way for any `istream` object.

Stream Insertion Operations

The `operator<<()` function is overloaded in the `ostream` class for formatted stream output of values of the fundamental types. Output to `cout` works analogously to input with `cin`. You can write the values of `i` and `x` to `cout` with this statement:

```
std::cout << i << ' ' << x;
```


This statement translates to three `operator<<()` function calls:

```
std::cout.operator<<(std::cout.operator<<(i), ' ').operator<<(x);
```

All versions of the `operator<<()` function return a reference to the stream object for which they're called, so you can use the return value to call the next `operator<<()` function. The `operator<<()` functions that write single characters and null-terminated strings to the stream are implemented as non-member functions, which is why the operator function call to write `i` to the stream appears as the first argument to the operator function call to write the space. The non-member function returns the stream object, and that's used to call the member function that writes the value of `x`.

`operator<<()` is overloaded in the `ostream` class for the same set of types as `operator>>()` in the `istream` class. Outputting a single character or a null-terminated string is catered for by non-member versions of `operator<<()`. The functions that output a single character to an output stream have the following prototypes:

```
ostream& operator<<(ostream& out, char ch);
ostream& operator<<(ostream& out, signed char ch);
ostream& operator<<(ostream& out, unsigned char ch);
```

There are similar functions defined that will output null-terminated strings:

```
ostream& operator<<(ostream& out, const char* str);
ostream& operator<<(ostream& out, const signed char* str);
ostream& operator<<(ostream& out, const unsigned char* str);
```

You can see now why you can output a string using a pointer to a `char` type, but pointers to other types are always written to a stream as an address. Because these functions are defined, sending a variable of type `const char*` to an output stream writes the string to which the pointer points to the stream, rather than the address stored in the pointer variable. If you want the address contained in the pointer to be output rather than the string, you must explicitly cast it to type `void*`. Then, the member function of `ostream` that has a parameter of that type will be called. Thus, these statements output a message

```
const char* message {"More is less and less is more."};
std::cout << message;
```

To output the address contained in `message`, you can use this statement:

```
std::cout << static_cast<void*>(message);
```

Stream Manipulators

You've already made extensive use of manipulators to control formatting of stream output in text mode. The basic manipulators shown in Table 17-1 can be inserted into any stream in text mode; these are defined in the `ios` header.

Table 17-1. *Basic Stream Manipulators*

Manipulator	Effect
dec	Sets the default radix for integers to decimal.
oct	Sets the default radix for integers to octal.
hex	Sets the default radix for integers to hexadecimal.
fixed	Outputs floating-point values in fixed-point notation without an exponent.
scientific	Outputs floating-point values in scientific notation with an exponent.
hexfloat	Enables hexadecimal floating-point formatting.
defaultfloat	Restores the default formatting for floating-point values.
boolalpha	Represents bool values as alphabetic; true and false in English.
noboolalpha	Represents bool values as 1 and 0, which is the default.
showbase	Indicates the base for octal (0 prefix) and hexadecimal (0x prefix) integers.
noshowbase	Omits the base indication for octal and hexadecimal integers.
showpoint	Always outputs floating-point values to the stream with a decimal point.
noshowpoint	Outputs integral floating-point values without a decimal point.
showpos	Displays a + prefix for positive integers.
noshowpos	Does not display a + prefix for positive integers.
skipws	Skips whitespace on input.
noskipws	Does not skip whitespace on input.
uppercase	Uses uppercase for hexadecimal digits A to F, and E for an exponent.
lowercase	Uses lowercase for hexadecimal digits a to f, and e for an exponent.
internal	Inserts “fill characters” to pad the output to the field width.
left	Aligns values left in an output field.
right	Aligns values right in an output field.
endl	Writes a newline character to the stream buffer and writes the contents of the buffer to the stream.
flush	Writes data from the stream buffer to the stream.

All of the manipulators in Table 17-1 can be inserted directly into the stream using the << operator, for example:

```
int i {1000};
std::cout << std::hex << std::uppercase << i << std::endl;
```

This will write the value of the integer, *i*, as a hexadecimal value using uppercase hexadecimal digits followed by a newline. In other words, you’d see 3E8 in the output.

It’s interesting and educational to see how this works. You know that using an insertion operator results in a version of `operator<<()` being called, but none of the versions I’ve described so far can be involved here, because they only deal with values of specific types being output to the stream. The effect of these manipulators doesn’t

involve sending data to a stream, so they can't be data values. In fact, all of the manipulators in Table 17-1 are pointers to functions of the same type. When you use one of these manipulators, a version of `operator<<()` that accepts a pointer to a function as the argument is called and the manipulator is passed as the argument.

To make this clearer, I'll take an example. The hex manipulator is a pointer to this function that is defined in the `ios` header:

```
ios_base& hex(ios_base& str);
```

You might use the hex manipulator in a statement such as this:

```
std::cout << std::hex << i;
```

This translates to:

```
(std::cout.operator<<(std::hex)).operator<<(i);
```

The first call of `operator<<()` has the pointer to function, `std::hex`, as an argument. Within the `operator<<()` function, the `hex()` function will be called using the function pointer to set the output formatting to transfer the value of `i` to the stream in hexadecimal format.

The `ios_base` class that cropped up in the prototype for `hex()` is the base of the `ios` class, as you saw at the beginning of this chapter. Because this is the base for all of the stream classes, the type `ios_base&` can reference any stream object. All of the manipulators are pointers to functions that have a parameter and a return type of type *reference to ios_base*, so they all result in the same version of `operator<<()` being called, which will call the function pointed to by the argument. `ios_base` defines flags that control the stream, and the function that is called when you use a manipulator modifies the appropriate flags to produce the desired result. You can modify these flags directly using the functions `std::setf()` and `std::unsetf()` for a stream object, but it's much easier to use the manipulators.

Manipulators with Arguments

There are some manipulators that accept an argument. These are defined in the `iomanip` header. You use these manipulator functions in the same way as the other manipulators, by effectively inserting a function call into the stream. Table 17-2 shows the ones that are most commonly used.

Table 17-2. Manipulators That Accept an Argument

Manipulator	Effect
<code>setprecision(int n)</code>	Sets the precision for floating-point output to <code>n</code> digits. This remains in effect until you change it.
<code>setw(int n)</code>	Sets the field width for the next output value to <code>n</code> characters. This will reset on each output to the default setting, which outputs a value in a field width that is just sufficient to accommodate the value.
<code>setfill(char ch)</code>	Sets the fill character to be used as padding within the output field to <code>ch</code> . This is modal, so it remains in effect until you change it again.
<code>setbase(int base)</code>	Sets the output representation for integers to octal, decimal, or hexadecimal, corresponding to values for the argument of 8, 10, or 16. Any other values will leave the number base unchanged.

The return type of each of the manipulators in Table 17-2 is implementation defined. Here's an example of using them:

```
std::cout << std::endl << std::setw(10) << std::setfill('*') << std::left << i << std::endl;
```

This statement outputs the value `i` left-justified in a field that's ten characters wide. The field will be padded with `'*'` in any unused character positions to the right of the value. The fill character will be in effect for any following output values, but you must set the field width explicitly prior to each output value.

■ **Caution** It's an error to include parentheses for the manipulators that do not require an argument such as `std::left`, so don't confuse them with the manipulators in `iomanip` that do require an argument.

The `iomanip` header also declares the `setiosflags()` and `resetiosflags()` functions that set or reset the flags that control stream formatting by specifying a mask. You construct the mask using the bitwise OR operator to combine flags that are defined in the `ios_base` class. The name of each flag is the same as the name of the manipulator that sets it, so you could set the flags for left-justified, hexadecimal output like this:

```
std::cout << std::endl << std::setw(10)
    << std::setiosflags(std::ios::left | std::ios::hex) << i << std::endl;
```

This will output `i` as a left-justified, hexadecimal value in a field that's ten characters wide. You can use `ios` instead of `ios_base` as the qualifier for the flag names here, because the flags are inherited in the `ios` class from `ios_base`.

File Streams

There are three types of stream objects for working with files: `ifstream`, `ofstream`, and `fstream`. As you saw earlier, these classes have `istream`, `ostream`, and `iostream` as base classes, respectively. An `istream` object represents a file input stream so you can only read it, an `ofstream` object represents a file output stream that you can only write to it, and `fstream` is a file stream that you can read or write.

You can associate a file stream object with a physical file when you create it. You can also create a file stream object that isn't associated with a file, and then call a function member to establish the connection with a specific file. In order to read or write a file, you must "open" the file; this attaches the file to your program via the operating system with a set of permissions that determine what you can do with it. If you create a file stream object with an initial association to a file, the file is opened and available for use immediately. It's possible to change the file that is associated with a file stream object so you can use a single `ofstream` object for example to write to different files at different times.

A file stream has some important properties. It has a *length*, which corresponds to the number of characters in the stream; it has a *beginning*, which is index position of the first character in the stream; and it has an *end*, which is the index position one beyond the last character in the stream. It also has a *current position*, which is the index position of the character in the stream where the next read or write operation will start. The first character in a file stream is at index position 0. These properties provide a way for you to move around a file to read the particular parts that you're interested in or to overwrite selected areas of the file.

Writing a File in Text Mode

To begin investigating file streams, let's look at how you can write to a file stream. An output file will be represented by an `ofstream` object, which you can create like this:

```
std::ofstream outFile {"filename"};
```

The argument to the `ofstream` constructor is the file name, which can be a string literal, a variable of type `char*` pointing to a C-style string, or a `string` object. The file name can include the file path, but if it doesn't, the file should be in the current directory. Text mode is the default so the file identified by `filename` will automatically be opened for writing in text mode and you can write to it immediately. I'll explain how you open a file in binary mode later. If `filename` doesn't exist, a file with this name and path will be created. If the file exists, the file is opened with the file position at 0, the beginning of the file, so whatever you write to the file will overwrite any existing contents. The `outFile` object has an `ostream` sub-object, so the stream operations that I've discussed for the standard output stream apply to an output file stream in text mode. Let's see text mode output to a file working in an example that finds prime numbers and writes them to a file:

```
// Ex17_01.cpp
// Writing primes to a file
#include <cmath>                // For sqrt() function
#include <fstream>              // For file streams
#include <iomanip>              // For stream manipulators
#include <iostream>            // For standard streams
#include <string>              // For string type
#include <vector>              // For vector container
using ulong = unsigned long long;

int main()
{
    size_t max {};              // Number of primes required
    std::cout << "How many prime would you like (at least 4)? : ";
    std::cin >> max;
    if (max < 4) max = 4;

    std::vector<ulong> primes {2ULL, 3ULL, 5ULL}; // First three primes defined
    ulong trial {5ULL};                // Candidate prime
    bool isprime {false};              // true when a prime is found
    ulong limit {};                    // Maximum divisor

    while (primes.size() < max)
    {
        trial += 2;                    // Next value for checking
        limit = static_cast<ulong>(std::sqrt(trial));
        for (auto prime : primes)
        {
            if (prime > limit) break;    // Only check divisors < square root
            isprime = trial % prime > 0; // false for exact division...
            if (!isprime) break;         // ...if so it's not a prime
        }
        if (isprime)                    // If we found one...
            primes.push_back(trial);     // ...save it
    }
}
```

```

std::string filename {"d:\\Example_Data\\primes.txt"};
std::ofstream outFile {filename};           // Define file stream object

// Output primes to file
size_t perline {5};                        // Prime values per line
size_t count {};
for (auto prime : primes)
{
    outFile << std::setw(10) << prime;
    if (++count % perline == 0)            // New line after every perline primes
        outFile << std::endl;
}
outFile << std::endl;
std::cout << max << " primes written to " << filename << std::endl;
}

```

Here's some sample output:

```

How many prime would you like (at least 4)? : 1000
1000 primes written to d:\\Example_Data\\primes.txt

```

All the primes are written to the file. If you intend to use the path specified in the code, you must create the ExampleData directory on drive D: before you execute the program; otherwise change the initial value for filename to the path you have set up for examples. Note that you can use a single forward slash, '/', instead of '\\\\' as a delimiter in a string specifying a file path. The Standard Library does not provide a way to create directories or folders because these are system dependent. The library that comes with your compiler may provide functions to create directories. You should be able to view the contents of the file created by the example using any text editor.

I defined `ulong` as an alias for type `unsigned long long` to save typing and reduce the length of some of the statements. Most of the detail of determining whether or not an integer is prime you have seen before. The change here is that this code only checks prime divisors for `trial` up the square root of its value, so execution will be faster when `trial` is a prime; without the square root as the upper limit for divisors, division by all primes in the vector would be tried. An integer that is not prime has two factors whose product is the non-prime value. If the two factors are identical, they are the square root of the value. If they are different, one has to be less than the square root. Thus any non-prime integer always has at least one factor that is less than or equal to its square root.

`std::sqrt()` is defined in the `cmath` header by a template that with integral arguments returns a value of type `double`. Without the explicit cast of the result to type `ulong`, you are likely to get a compiler warning because of the potential loss of data from an implicit cast of a floating-point value to an integer. The primes are saved in a vector container because we need the known primes in memory for use in checking a prime candidate. Of course, it would be possible to hold a subset of the primes in memory with the rest in a file, but that would need the capability to read the file, which I haven't explained yet.

The `fstream` header defines the `ofstream` type as well as the `ifstream` and `fstream`, so including this header into a source file provides for creating stream objects for all combinations of file read and write operations. The `ofstream` object is created by passing filename that identifies the file name and path to the constructor. You could pass a literal to the constructor here, but frequently you'll need to allow the file to be used to be identified by user input. The output stream is in text mode by default. The primes are written to the file using the `<<` operator in exactly the same way as if you were writing them to `cout`. You can see that the stream manipulators work in the same way too. The `setw()` manipulator is important here. Without the use of `setw()`, there would be no whitespace between one prime and the next.

The file is closed automatically when the `ostream` object is destroyed - at the end of the program in this case. You can close a file by calling the `close()` member of the `ostream` object. For example:

```
outFile.close(); // Close the file associated with the stream
```

It is best practice to explicitly close a file when it is no longer required, even when you know it will eventually be closed for you. After executing this statement you can no longer write to the file. You can verify that the file contents is overwritten by running the program twice with a lower number of primes in the second instance and looking at the contents after each run.

■ **Note** You don't have to overwrite the file contents; that happens to be the default setting. I'll explain how you control the way a file is written a little later in this chapter.

Reading a File in Text Mode

To read a file, you create an `ifstream` object that encapsulates the file, for example:

```
string filename {"D:\\Example_Data\\primes.txt"};
std::ifstream inFile {filename};
```

This defines an `ifstream` object, `inFile`, that encapsulates the `primes.txt` file that was created in the `Example_Data` directory on the D: drive, and opens the file ready for reading in text mode with the file position as 0. If you're going to read a file, it must already exist and should not be empty, but we all know that things don't always go as planned. What happens if you try to read from a file that you haven't prepared earlier?

Checking the State of a File Stream

As far as the definition of the `ifstream` object is concerned, the answer is absolutely nothing: you just have a file stream object that won't work. To find out if everything is as it should be, you must test the status of the file, and there are several ways to do this. One possibility is for you to call the `is_open()` member of the `ifstream` object, which returns `true` if the file is open and `false` if it isn't; if the file doesn't exist, clearly it can't be open. Another option is to call the `fail()` function that is inherited in the file stream classes from the `ios` class; this returns `true` if any file error occurred.

You can also use the `!` operator with a file stream object. This operator is overloaded in the `ios` class to check the stream status indicators. When applied to a stream object, it returns `true` if the stream isn't in a satisfactory state. Using the `!` operator function is equivalent to calling `fail()` for the stream object. To make sure a stream object is in a satisfactory condition and ready for use, you can write this:

```
if(!inFile)
{
    std::cout << "Failed to open file " << filename << std::endl;
    return 1;
}
```

You can test whether or not an output file stream object is available for use in exactly the same way, because the `ofstream` class inherits from `ios` too. It also inherits the `fail()` function and implements the `is_open()` function.

The stream classes also inherit an overload of operator `bool()` from the `basic_ios` class. This returns `true` if a stream object is ready for I/O to be performed. The overload is explicit, so you can only use this to test a stream using an explicit cast. For example:

```
if(static_cast<bool>(inFile))
{
    // No error states so we can read the file...
}
else
{
    std::cout << "Failed to open file " << filename << std::endl;
    return 1;
}
```

Even though casting to type `bool` offers a positive check mechanism, I think it's clearer to use the expression `!inFile.fail()`. However, there's an even simpler option. The `basic_ios` class defines operator `void*()`. This operator function overloads a conversion of an object to a pointer of type `void*`; it is not defined as explicit so the compiler can insert it as an implicit conversion. The function returns `nullptr` if calling `fail()` for the stream object returns `true`, and a non-null pointer otherwise. You know that a pointer as an `if` or `while` loop expression is implicitly converted to type `bool`. Therefore you can validate an input file stream object like this:

```
if(inFile)
{
    // No error states so we can read the file...
}
else
{
    std::cout << "Failed to open file " << filename << std::endl;
    return 1;
}
```

This is equivalent to the previous code fragment but less typing. I'll look further into stream error states a little later in this chapter.

Reading a file in text mode is just like reading from `cin` —you use the extraction operator in exactly the same way. However, you don't necessarily know how many data values there are in a file, so how do you know when you've reached the end? The `eof()` function that is inherited from `basic_ios` in the `ofstream` class provides a neat solution. It returns `true` when the end of file (EOF) is reached, so you can just continue to read data until that happens.

Reading the File

You now know enough about how input file streams work to read the file that was written by the previous example. This example reads the file and outputs the primes to the screen. Here's the code:

```
// Ex17_02.cpp
// Reading the primes file
#include <fstream>
#include <iostream>
#include <iomanip>
#include <string>
using ulong = unsigned long long;
```



```

int main()
{
    std::string filename {"D:\\Example_Data\\primes.txt"};    // Input file name
    std::ifstream inFile {filename};                          // Create input stream object

    // Make sure the file stream is good
    if (!inFile)
    {
        std::cout << "Failed to open file " << filename << std::endl;
        return 1;
    }

    ulong aprime {};
    size_t count {};
    size_t perline {6};
    while (true)                                              // Continue until EOF is found
    {
        inFile >> aprime;                                     // Read a value from the file
        if (inFile.eof()) break;                             // Break if EOF reached

        std::cout << (count++ % perline == 0 ? "\n" : "") << std::setw(10) << aprime;
    }
    std::cout << "\n" << count << " primes read from " << filename << std::endl;
}

```

The output will be a listing of the primes written by Ex17_01.cpp. They were written 5 to a line in the file but the output here is 6 to a line, just to be different. A newline is a whitespace character that is ignored by the << operator function for stream input so it has no effect on the output to cout. However, although this file is easy to view in a text editor in this instance, there's a potential problem with reading it in general. Each value is written to the file with a field width of 10. This is fine when the values are less than 10 digits, but there will be no whitespace between successive values if one of them is 10 or more digits. This is unlikely here, but possible in general. The absence of whitespace between successive values would prevent the file from being read correctly. The >> operator depends on whitespace to separate one value from the next. If it isn't there, the input process cannot determine where one value ends and the next begins; a contiguous sequence of digits will be read as a single input value. It would be better to ensure there is at least a newline character between successive values in the file. You need to keep this in mind whenever you are writing a file in text mode. We'll fix this in the next example.

The file input stream object is type `ifstream` and the file name is passed to the constructor analogous to creating an `ofstream` object. Assuming it exists, the file is opened ready to read with the file position as 0, which is the beginning of the file contents. After creating `inFile`, the `!` operator is applied to the object in the `if` statement to verify that the file exists. You can see this check failing if you introduce an error into the file name - spell it `prrimes.txt` for example.

The primes are read from `inFile` in an indefinite while loop using the >> operator for the stream object. The `if` statement that calls `eof()` for `inFile` to determine when the end of file has been reached ends the loop. Note that the EOF condition is set when you read a file *after* the last data item has been read—the file position will be end, one beyond the last byte in the file. EOF is *not* set by reading the last data item. This means that the check for EOF will only be true if all the data has been read (or the file position has been set to end by some other means) and you execute another read operation. When EOF is detected by a read operation, the EOF flag is set in the stream object but no data is transferred, so `aprime` will be left unchanged by the read operation that executes after the last prime has been read.

The file will be closed when the stream object is destroyed, but you can close an input stream in the same way as an output stream:

```
inFile().close(); // Close the input stream
```

The file can no longer be read after `close()` has been called.

Setting the Stream Open Mode

The *open mode* for an `ifstream` or `ofstream` object determines what you can do with a file. You define an open mode as a combination of bit mask values of type `openmode` that are defined in `ios_base` and inherited in the stream classes via the `ios` class. The constructors for `ifstream` and `ofstream` objects have a second parameter of type `openmode` that has a default value that we have used up to now. The `openmode` mask values are shown in Table 17-3.

Table 17-3. Stream open mode values

Value	Meaning
<code>ios::app</code>	Moves to the end of the file before each write (a ppend operation). This ensures that you can only add to what is in a file; you can't overwrite it.
<code>ios::ate</code>	Moves to the end of the file after opening it (a t the end). You can move the current position to elsewhere in the file subsequently.
<code>ios::binary</code>	Sets binary mode. In binary mode, all characters are unchanged when they're transferred to or from the file. If binary mode is not set, the mode is text mode.
<code>ios::in</code>	Opens the file for reading.
<code>ios::out</code>	Opens the file for writing.
<code>ios::trunc</code>	Truncates the existing file to zero length.

An open mode specification consists of one or more of these `openmode` values. Because each `openmode` setting is a single bit, you generate an open mode that is a combination of these by bitwise-ORing them. For a file that is to be opened for writing in binary mode, such that data can only be added at the end of the file, you would specify the mode by the expression `ios::out|ios::app|ios::binary`. You can open a file for reading *and* writing by specifying both `ios::in` and `ios::out`, but this is only legal with an object of type `fstream`, as you'll see shortly.

The default open mode for an `ifstream` object is `ios::in`, which just opens the file for input. The default for an `ofstream` object is `ios::out`, which specifies that the file is to be opened for output. You can set the file open mode by supplying it as a second argument to a file stream constructor. For example, you can specify the open mode for a file output stream such that you can append data to the file rather than overwrite the contents like this:

```
std::string filename {"D:\\Example_Data\\primes.txt"};
std::ofstream outFile {filename, std::ios::out|std::ios::app};
```

The combination of `ios::out` and `ios::app` as the open mode causes the file to be opened for output so that all writes are at the end of the file; this means that the file position is set to end. It's not essential to specify `ios::in` for an input file stream and `ios::out` for an output file stream because it is implicit in the type of stream. Thus the definition of `outFile` could be:

```
std::ofstream outFile {filename, std::ios::app}; // Same as previous definition
```

If you explicitly close a file stream by calling `close()` for the stream object, you can reopen the stream with a different open mode by calling `open()` for the stream object. The `open()` member accepts two arguments, the file name and the open mode. The second parameter to `open()` class has a default value that is the same as for the constructor. You could close the `outFile` stream and reopen it with a different open mode setting with the following statements:

```
outFile.close();
outFile.open(filename);
```

This reopens the file to overwrite the original contents, because `ios::out` is the default value for the second parameter. To reopen the file to append data you would use this statement:

```
outFile.open(filename, std::ios::out|std::ios::app);
```

Let's create a variation on `Ex17_01` that generates and displays the number of primes that you request, but if the `primes.txt` file already exists, the primes in the file are accessed and displayed as necessary and new primes that need to be found are added to the file. The program will only need to generate primes in excess of those already in the file. Any that are already in the file can be displayed immediately.

The logic is a little complicated so I'll go through the code in detail. The example will need to determine whether or not there is an existing file, and if it doesn't exist, it must create it. If it does exist, the program will output primes from the file up to the number required, if they are all in the file. If there are fewer primes in the file than the number requested, the example must discover the additional primes and write them to the file, as well as outputting them to the standard output stream. Here's the code to do that:

```
// Ex17_03.cpp
// Reading and writing the primes file
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cmath>
#include <string>
using std::ios;
using std::string;
using ulong = unsigned long long;

ulong nextprime(ulong aprime, const string filename); // Find the prime following aprime

int main()
{
    string filename {"D:\\Example_Data\\more_primes.txt"};
    size_t nprimes {}; // Number of primes required
    size_t count {}; // Count of primes found
    ulong lastprime {}; // Last prime found
    size_t perline {5}; // Number output per line

    // Get number of primes required
    std::cout << "How many primes would you like (at least 4)? ";
    std::cin >> nprimes;
    if (nprimes < 4) nprimes = 4;
```

```

std::ifstream inFile;                                // Create input file stream
inFile.open(filename);                                // Open the filename file for input

if (!inFile.fail())                                   // If there is a file...
{                                                       // ...read primes from it
    while (true)
    {
        inFile >> lastprime;
        if (inFile.eof()) break;

        std::cout << std::setw(10) << lastprime << (++count % perline == 0 ? "\n" : "");
        if (count == nprimes) break;
    }
    inFile.close();                                    // Reading is finished
    if (count == nprimes)                               // Check if we found them all
    {
        inFile.close();                                // We are done with the file
        std::cout << std::endl << count << "primes found in file." << std::endl;
        return 0;
    }
}

// If we get to here, we need to find more primes
inFile.clear();                                        // Clear EOF flag
inFile.close();                                        // Reading is finished
try
{
    size_t oldCount {count};                            // The number that were in the file
    std::ofstream outFile;                               // Create an output stream object

    if (oldCount == 0)
    { // The file is empty
        outFile.open(filename);                          // Open file to create it
        if (!outFile.is_open())
            throw ios::failure {string {"Error opening output file "} + filename + " in main()"};

        outFile << "2\n3\n5\n";                          // Write 1st three primes to file
        outFile.close();
        std::cout << std::setw(10) << 2 << std::setw(10) << 3 << std::setw(10) << 5;
        lastprime = 5;
        count = 3;
    }
}

while (count < nprimes)
{
    lastprime = nextprime(lastprime, filename);
    outFile.open(filename, ios::out | ios::app); // Open file to append data
}

```

```

    if (!outFile.is_open())
        throw ios::failure {string {"Error opening output file "} + filename + " in main()"};
    outFile << lastprime << '\n';
    outFile.close();
    std::cout << std::setw(10) << lastprime << (++count % perline == 0 ? "\n" : "");
}
std::cout << std::endl << nprimes << " primes found. "
    << nprimes-oldCount << " added to file." << std::endl;
}
catch (std::exception& ex)
{
    std::cout << typeid(ex).name() << ": " << ex.what() << std::endl;
    return 1;
}
}

```

main() calls the nextprime() function to obtain the prime that follows the prime that is the first argument. The second argument is the name of the file that contains the primes found so far. The definition of this function is:

```

ulong nextprime(ulong last, const string filename)
{
    bool isprime {false};                // true when we have a prime
    ulong aprime {};                     // Stores a prime from the file
    std::ifstream inFile;                // Local file input stream object

    // Find the next prime
    ulong limit {};
    while (true)
    {
        last += 2ULL;                    // Next value for checking
        limit = static_cast<ulong>(std::sqrt(last));

        // Try dividing the candidate by all the primes up to limit
        inFile.open(filename);            // Open the primes file
        if (!inFile.is_open())
            throw ios::failure {string {"Error opening input file "} + filename + " in nextprime()"};

        do
        {
            inFile >> aprime;
        } while (aprine <= limit && !inFile.eof() && (isprime = last % aprime > 0));
        inFile.close();
        if (isprime)                      // We got one...
            return last;                  // ...so return it
    }
}

```

This program will output the number of primes that you request and write any new ones to the file `more_primes.txt`. Here some sample output from an execution that is not the first:

```
How many primes would you like (at least 4)?: 31
    2         3         5         7         11
   13        17        19        23        29
   31        37        41        43        47
   53        59        61        67        71
   73        79        83        89        97
  101       103       107       109       113
  127
31 primes found. 2 added to file.
```

Because all the primes that are found are stored in the file, it's no longer necessary to keep them in memory; they can be read from the file when required. Primes are found by the `nextprime()` function, but before exploring that, let's look at how the code in `main()` works.

After getting the number of primes required from the user, the `more_primes.txt` file is opened as an input stream. The `ifstream` object, `inFile`, is created using the default constructor. The stream object will not be associated with a file at this point. To use the stream object to open the `more_primes.txt` file, the `open()` function is called with the file name as the argument. The function can accept a second argument — the open file mode — but because it's not specified, the default value of `ios::in` will apply. Of course, the first time you run the example, the file won't exist, so it will be created. Note that if the path to the file does not exist, an exception will be thrown in `main()`. You can verify that this is the case by changing the directory name to a non-existent directory.

Of course, it may be that the file could not be opened for some reason so it's a good idea to verify that the file stream is in a good state before you try to read it. This is done by calling `fail()` for the file stream object in the `if` statement. `fail()` returns true if the file could not be opened so if the expression `!inFile.fail()` is true, all is well.

Within the `if` block, up to `nprimes` values are read from the file in the indefinite `while` loop. Calling `eof()` for `inFile` after each read operation checks whether the end of file has been reached. After each prime read successfully has been written to `cout`, there's a check whether the number required have been found. If they have, there's nothing more to do beyond closing the file and outputting a suitable message. When this occurs, the file obviously contains more primes than were required and therefore no new primes are added.

If the `while` loop ends because EOF was reached, the condition remains set until it is reset. Calling the `clear()` function for the stream object resets all errors flags, including the flag that indicates EOF has been recognized. I'll discuss what else you can do with the `clear()` function a little later in the chapter.

After closing the file, the remainder of the code in `main()` comes within a `try` block because an exception will be thrown if there's a problem with opening the file. The `catch` block will catch any exception type that has `exception` as a base so it will catch `ios::failure` objects.

At this point, the value of `count` represents the number of primes in the file; you save this in `oldCount` for use in a message after all the required primes have been found. Of course, if the value of `oldCount` is 0, no primes were read from the file so you must write the file from scratch. This involves opening the file and writing the first three primes, 2, 3, and 5, to it as seed values. Of course, this is a text file so you can write the three values as a single string. You also write these values to `cout` because they are first three of those required. Subsequent primes will follow 5 so `lastprime` is set to this value.

Subsequent primes are found in the `while` loop that continues as long as `count` is less than `nprimes`. The next prime is found by calling the `nextprime()` function with `lastprime` as the first argument; the second argument is the file name because the function will use the file contents as divisors.

Calling `open()` for `outFile` with `ios::out|ios::app` as the argument opens the file to append data. After verifying the file was opened successfully, `lastprime` is written to the file followed by a newline as separator. The file is then closed because it will need to be opened to read it on the next iteration to find the next prime.

The `nextprime()` function finds a new prime using the primes from the file as divisors, so it creates a local `ifstream` object. The process for finding the next prime is in the indefinite `while` loop. The first value to be checked is obtained by incrementing `lastprime` by 2. The value passed to the function as `lastprime` will be the last prime number found, so you don't need to check that it is odd. To determine whether value is a prime, you try dividing by all the primes up to the square root of the value in the `do-while` loop. The loop continues as long as the prime read is less than limit and it not an exact divisor and as long as EOF has not been reached. If the loop ends because of an exact divisor, `isprime` will be `false` so the outer `while` loop will continue with the next candidate in `last`.

The program works, but this process of opening and closing the file every time you go round the loop in the `nextprime()` function is very inefficient. It would be much better if you could just read the file from the beginning each time, so let's see how you could do that.

Managing the Current Stream Position

You can access and change the current stream position using function members of the `istream` and `ostream` classes; these don't apply to the standard streams because the standard streams don't relate to physical devices for which a stream position would be meaningful. They *do* apply to objects of all the file stream classes. The functions in `istream` and `ostream` are inherited in `ifstream` and `ofstream` respectively, and both sets of functions are inherited by the `fstream` class via `iostream`.

There are two things that you can do in relation to the stream position: you can obtain the current position in the stream, and you can change the current position. The current position is returned by the `tellg()` function for input stream objects and by the `tellp()` function for output stream objects. The *g* in `tellg()` is for "get" and the *p* in `tellp()` is for "put," so this indicates whether you get data from the stream the function relates to or you put data into it. Both functions return a value of type `pos_type` that represents an absolute position in a stream. `pos_type` is an integer type that is defined within the `basic_ios` class and inherited by the stream classes.

For example, you could obtain the current position in an input file stream object called `inFile` using this statement:

```
std::ifstream::pos_type here {inFile.tellg()}; // Record current file position
```

This initializes `here` with the current file position, the type for `here` is cumbersome because `pos_type` must be qualified with the name of the class in which it is defined. You can avoid having to remember the specific type for a file position by using the `auto` keyword:

```
auto here = inFile.tellg(); // Record current file position
```

Don't forget. *always* use `=` when specify a variable type as `auto`; if you use an initializer list, the variable will be the wrong type.

You define a new position in a stream by calling `seekg()` for an input stream object or `seekp()` for an output stream object. The argument is typically a position that you recorded using either `tellg()` or `tellp()`. For example, you could reset the stream position for `inFile` back to `here` with this statement:

```
inFile.seekg(here); // Set current position to here
```

A stream position is an integral value that is the index position of a character in the stream, where the first character is at index position 0. It's therefore possible to use integer values to move to specific positions in a stream. This can be hazardous in text mode because it's difficult to keep track of where data is in a text file, not least because the number of characters in the stream may be different from the number of characters written. However, seeking to

position 0 is always going to move to the beginning of the stream, so it could be used in the `nextprime()` function to avoid closing and reopening the file on each iteration of the while loop:

```

ulong nextprime(ulong last, const string filename)
{
    bool isprime {false};           // true when we have a prime
    ulong aprime {};                // Stores a prime from the file
    std::ifstream inFile(filename); // Local file input stream object
    if (!inFile.is_open())
        throw ios::failure {string {"Error opening input file "} + filename + " in nextprime()"};

    // Find the next prime
    ulong limit {};
    while (true)
    {
        last += 2ULL;                // Next value for checking
        limit = static_cast<ulong>(std::sqrt(last));

        // Try dividing the candidate by all the primes up to limit
        do
        {
            inFile >> aprime;
        } while (aprime <= limit && !inFile.eof() && (isprime = last % aprime > 0));

        inFile.seekg(0);
        if (isprime)                 // We got one...
        {
            inFile.close();          // ...close the file...
            return last;             // ...and return the prime
        }
    }
}

```

With this version of the function, you open the file when you create the `ifstream` object and simply reset the file position to the first character in the file at the end of the while loop. Only when a prime has been found do you close the input file.

An alternative to moving to an absolute position in a stream is to move to a position specified as an offset relative to one of three specific positions in a stream. The offset can be positive or negative. You can define a new position relative to the first character in the stream (the offset must be positive), relative to the last character in the stream (the offset must be negative), or relative to the current position. The offset from the current position can be positive or negative, as long as the position is not at either end. Figure 17-3 illustrates how you seek to a relative position.

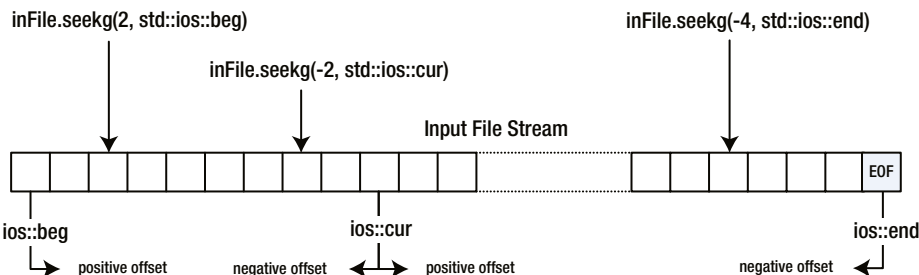


Figure 17-3. Moving to a relative stream position

You use versions of `seekg()` or `seekp()` that accept two arguments when moving to a relative position. The first argument is the offset, which is an integral value of type `off_type`, and the second argument must be one of the values in Table 17-4 that are defined in the `ios` class.

Table 17-4. Constants identifying the base position for a relative stream position

Value	Description
<code>beg</code>	Offset is relative to the first character in the file.
<code>cur</code>	Offset is relative to the current file position.
<code>end</code>	Offset is relative to one beyond the last character in the file.

The constants in Table 17-4 are *not* stream positions. They are constants of type `std::ios_base::seekdir`, not the stream position type. As I explained, the offset value must be positive relative to `ios::beg` and negative relative to `ios::end`. An offset of 0 relative to `ios::end` sets the position to the end of the file, so you can use this when you want to append to a file. An offset can be in either direction relative to `ios::cur` by using positive or negative values for the offset. You can specify the offset as an integer constant, or as an expression that evaluates to an integer. Of course, these standard position values are inherited in the file stream classes so you can access them as a member of such objects.

The `seekg()` functions return a reference to the file stream object, so you can combine a seek operation with an input operation by using an extraction operator, for example:

```
inFile.seekg(10, inFile.beg) >> value;
```

This statement moves the file position to an offset of ten characters from the beginning of the file and read a data item from that point into `value`.

Similarly, you can use the `seekp()` function to move to a position where you want to start the next *output* operation; the function arguments are exactly the same as for `seekg()`. The next write to the stream will overwrite characters, starting with the character at the new position. Relative seek operations don't work in text mode, and absolute seeks are dubious so I recommend that you avoid seeking in text mode.

You know that a file position is the index of a character position in the stream and `std::ios::end` is the position one beyond the last character written to the stream. You can therefore determine the length of a stream like this:

```
auto stream_length = inFile.seekg(0, inFile.end).tellg();
```

The `seekg()` function returns the file stream, so this is used to call `tellg()` for the same stream object. Because `seekg()` moves the position to one beyond the last character, the `tellg()` function returns a value that will be the number of characters in the stream.

■ **Note** I'm not done with the operations I've discussed here. I'll return to them later in this chapter when I discuss random read/write operations on a stream.

Unformatted Stream Operations

In addition to the insertion and extraction operators for formatted I/O in text mode, there are function members of the stream classes for transferring data without any formatting. The extraction operator treats whitespace characters as delimiters, but otherwise ignores them. The unformatted stream input functions don't skip whitespace characters — they're read and stored just like any other characters.

Unformatted Stream Input

The `istream` class defines a wealth of unformatted input functions that are inherited in the file stream classes. For a start, there are two varieties of the `get()` function member to read a single character from a stream:

```
std::istream::int_type get();
```

Reads a single character from the stream and returns it. `std::istream::int_type` is an implementation defined integral type that will be capable of storing any character. It will usually correspond to type `int`. If the end of file is reached, the function returns a character representing end of file, which is the character returned by `std::char_traits::eof()` and by the `EOF` macro in the `cstdio` header. If a character can't be read from the stream for any reason, the error flag `ios::failbit` will be set. (I'll discuss this and other error flags later in the chapter, when I discuss I/O errors.)

```
std::istream& get(char& ch);
```

Reads a single character from the stream and stores it in `ch`. The function returns a reference to the stream object, so you can combine calling this function with other member function calls. As with the previous function, if a character can't be read from the stream, the `failbit` error flag is set and `EOF` is stored in `ch`.

The `peek()` function member is similar to the first `get()` function in that it reads the next character from the stream; the difference is that it leaves the character in the stream so you can read it again. `peek()` returns the character read as a value of type `std::istream::int_type`.

You can return the last character read from a stream back to the stream by calling the `unget()` function member. For a file, this just moved the file position back to the character that was read. `unget()` returns a reference to the `istream` object for which it was called. It's typically used in combination with a `get()` function that reads a single character when parsing input. For example, you might need a function to skip characters that are not digits in a stream so as to position the stream at the next digit; you could define it like this:

```
std::ifstream& skipnondigits(std::ifstream& in)
{
    std::istream::int_type ch {};
    while (true)
    {
        ch = in.get();
        if(ch == EOF) break;
        if(std::isdigit(ch))
        {
                                                    // When a digit is read...
                                                    // ...put it back in the stream
            in.unget();
            break;
        }
    }
    return in;
                                                    // Return the stream
}
```

The effect of this function is to move the file position until a digit is found, or until `EOF` if no digits are present. Returning the file stream is a convenience for the caller to call a stream function member in the same statement. The caller could test for `EOF` for example, which would indicate that no digits were found:

```
if(!skipnondigits(in).eof())
{
    // Read a numerical value...
}
```

The `putback()` function member has a similar effect to `unget()`, but in this case you specify the character to put back in the stream as an argument. In the preceding example, instead of the statement calling `unget()`, you could have written this:

```
in.putback(ch);                                // ...put back the digit...
```

The argument to `putback()` must be the last character read; otherwise, the result is undefined. The `putback()` function also returns a reference to the stream.

There are two `get()` functions that read a sequence of characters as a null-terminated string:

```
std::istream& get(char* pArray, std::streamsize n);
```

Reads up to `n-1` characters from the stream and stores them in the array `pArray`, adding a null terminator at the end to make `n` characters in total. Characters are read until a newline character is read, the end of file is reached, or `n-1` characters have been read. If a newline is reached, it isn't stored, but `'\0'` is always appended to the sequence of characters read. The effect is to read a line of text without storing the `'\n'` character that marks the end of the line. The `'\n'` character remains in the stream as the next character to be read. Because up to `n` characters may be stored, the array pointed to by `pArray` should have at least `n` elements. `ios::failbit` is set if no characters are stored. `std::streamsize` is a signed integer type that is implementation defined.

```
std::istream& get(char* pArray, std::streamsize n, char delim);
```

This works in the same way as the previous function, except that you can specify a delimiter, `delim`, that will be used in place of newline to end the input process. If the delimiter is found, it isn't stored, but is left in the stream.

There are two `getline()` function members that are almost equivalent to the `get()` functions that read a line of text:

```
std::istream& getline(char* pArray, std::streamsize n);
std::istream& getline(char* pArray, std::streamsize n, char delim);
```

The difference between `getline()` and the corresponding `get()` function is that `getline()` removes the delimiter from the stream, so the next character to be read is the character following the delimiter.

After calling an unformatted input function, you can determine the number of characters that were read by calling `gcount()` for the stream object. This returns the count of characters read by the last unformatted input operation as a value of type `std::streamsize`.

You can read a specified number of characters from a stream, assuming they're available, with the `read()` function member:

```
std::istream& read(char* pArray, std::streamsize n);
```

This function reads `n` characters into `pArray` if they are available, including any newlines and null characters. If the end of the file is reached before `n` characters have been read, `ios::failbit` is set in the input object.

You'll typically use the `read()` function when you know that `n` characters *are* available in the stream. There's another member function that you can use when this isn't the case. The `readsome()` function operates similarly to `read()`, but it returns the count of the number of characters read:

```
std::streamsize readsome(char* pstr, std::streamsize n);
```

If fewer than `n` characters are available in the stream, the function sets the `ios::eofbit` flag so calling `eof()` for the stream object will return true.

You can skip over characters in an input stream with this function:

```
std::istream& ignore(std::streamsize n, std::istream::int_type delim);
```

Up to *n* characters are read from the stream and discarded. Reading the *delim* character or reading *n* characters ends the process. There are default values for the parameters *n* and *delim* of 1 and the character representing EOF, respectively. Thus, you can skip a single character with the following statement:

```
inFile.ignore(); // Skip one character
```

To skip 20 characters, you would write this:

```
inFile.ignore(20); // Skip 20 characters up to the end of the file
```

Reading the end of file will stop the process in this case, but you could skip 20 characters in the current line with the following statement:

```
inFile.ignore(20, '\n'); // Skip 20 characters up to the end of the current line
```

Unformatted Stream Output

In stark contrast to the plethora of unformatted input functions, you have just two functions available for unformatted output to a stream in text mode: `put()` and `write()`. The `put()` function takes the following form:

```
std::ostream& put(char ch);
```

This writes the single character, *ch*, to the stream and returns a reference to the stream object. To write a sequence of characters to a stream, you use the `write()` function, which has this form:

```
std::ostream& write(const char* pArray, std::streamsize n);
```

This writes *n* characters to the stream from the array, *pArray*. Any kind of characters can be written, including null characters. Generally, output to a stream is buffered and sometimes you want the contents of the stream buffer written to the stream regardless of whether the buffer is full. Calling the `flush()` member of an `ostream` object will do this. `flush()` writes the contents of the stream buffer to the device and returns a reference to the stream object.

Errors in Stream Input/Output

All the stream classes store the state of the stream in a data member that is an integer. This member is normally 0 but when a stream error occurs, the state is set to a combination of one or more error flags. These flags are defined by integer constants that are bit masks of type `std::ios_base::iostate`; this type is inherited along with the constants that are flags in all classes derived from `ios_base`. Table 17-5 shows the meanings of these flags.

Table 17-5. *Stream State Flags*

Flag	Meaning
<code>ios::badbit</code>	A single bit that is set when a stream is in a state in which it can't be used further—if an I/O error occurred, for example. This isn't recoverable.
<code>ios::eofbit</code>	A single bit that is set when the end of file is reached.
<code>ios::failbit</code>	A single bit that is set if an input operation didn't read the characters expected or an output operation failed to write characters successfully. This is typically due to a conversion or formatting error or reading beyond the end of the stream. Any subsequent operations will fail while the bit is set, but the situation may be recoverable.
<code>ios::goodbit</code>	Defined as 0. If the stream state is <code>goodbit</code> , there are no errors.

If EOF is read, the stream state will be `ios::eofbit`. If a serious error occurred while reading from a stream and no characters could be read, both the `ios::badbit` and `ios::failbit` flags will be set, so the stream state would be the result of `ios::badbit|ios::failbit`.

Once a flag is set, it remains set unless you reset it. You'll sometimes want to reset the flags — when you reach the end of a file while reading it, for example — because you may subsequently want to read the file again. Calling `clear()` for a stream object resets all the error flags:

```
infile.clear();                // Clear all error states
```

You can test the state of a stream by applying a cast to type `bool` or the `!` operator to the stream object, by calling its `fail()` member, or by using just the object; this can be in an `if` statement or a loop condition. `fail()` returns `true` if `badbit` and/or `failbit` have been set, the same as the `!` operator. For example:

```
while(!infile.fail())
{
    // Read from inFile...
}
infile.clear();                // Clear any error states
```

You can use this kind of loop to read to the end of a file, because when the end of file is reached, the read operation will set `failbit` as well as `eofbit`. You could also write this as:

```
while(infile)
{
    // Read from inFile...
}
infile.clear();                // Clear any error states
```

This is implicitly invoking operator `void*()` for the stream object followed by an implicit conversion of the pointer to type `bool`.

The stream classes inherit function members that test the state of individual flags. As Table 17-6 shows, they each return a value of type `bool`.

Table 17-6. *Functions for Testing Stream State Flags*

Function	Action
<code>bad()</code>	Returns true if <code>badbit</code> is set in the stream object.
<code>eof()</code>	Returns true if <code>eofbit</code> is set in the stream object.
<code>fail()</code>	Returns true if <code>failbit</code> or <code>badbit</code> is set in the stream object.
<code>good()</code>	Returns true if <code>goodbit</code> is set in the stream object which implies the other flags are not set.

Instead of just calling `clear()` at the end of the previous code fragment, you might do a more detailed analysis of the stream state:

```
while(inFile)
{
    // Read from inFile...
}
if(inFile.bad())
{
    std::cout << "Non-recoverable file input error." << std::endl;
    std::exit(1);
}
inFile.clear();
```

The `fail()` function will return true if either `failbit` or `badbit` is set, so there could have been a non-recoverable read error. This now terminates the program if `badbit` is set. The other error bits will be set by reading beyond the end of the file so it's reasonable to clear the error state and continue as long as `badbit` was not set.

Input/Output Errors and Exceptions

When errors occur in I/O operations, exceptions may be thrown. The exceptions for stream errors are of type `std::ios_base::failure`. This type is a nested class that is inherited in `ios` from `ios_base`, so you can use `ios::failure` as the exception type. A mask that is a member of a stream object determines whether an exception will be thrown when a particular stream state flag is set. You can set this mask by passing a mask to the `exceptions()` member of the stream object, with bits set to specify which flags you want to throw exceptions. For example, if you would like to have exceptions thrown when *any* of the flag bits is set for a stream called `inFile`, you could enable this with the following statement:

```
inFile.exceptions(ios::badbit | ios::eofbit | ios::failbit);
```

Now if anything goes wrong, even when just the end of a file is reached, an exception of type `ios::failure` will be thrown.

Generally, it's better to test the error flags in one of the ways I've discussed, rather than to use exceptions for handling I/O errors, at least as far as the `eofbit` and `failbit` flags are concerned. Most of the time, you'll be involved in dealing with `failbit` and `eofbit` flags, because these are a part of the normal process of handling stream input and output. The default position in most development environments is that exceptions are *not* thrown for stream errors.

You can check whether exceptions will be thrown by calling a version of `exceptions()` with no arguments that returns a value of type `ios::iostate`. The value returned reflects which error flags will result in an exception being thrown, so you can test whether a given flag being set will throw exceptions as follows:

```
std::ios::iostate willthrow {inFile.exceptions()};
if(willthrow & std::ios::badbit)
    std::cout << "Causing badbit to be set will throw an exception" << std::endl;
```

The result of ANDing `ios::badbit` with `willthrow` will be 0 unless `ios::badbit` is set in `willthrow`.

Stream Operations in Binary Mode

There are many situations in which text mode isn't appropriate or convenient, and it can sometimes cause difficulties. The transformation of newline characters into two characters on some systems and not others makes relative seek operations unreliable for programs that are to run in both environments. By using *binary mode*, you avoid these complications and make the stream operations much simpler. You've already seen how to open a stream in binary mode: you just need to specify the open mode flags appropriately. You can read and write a file in binary mode using functions that you have already seen, `put()` and `get()`. Let's try it in an example.

We can write a program that will copy any file. The copying can be done using the `get()` and `put()` function members for a stream that read and write a single character:

```
// Ex17_04.cpp
// Copying files
#include <iostream>                                // For standard streams
#include <cctype>                                    // For character functions
#include <fstream>                                  // For file streams
#include <string>                                    // For string type
#include <stdexcept>                                // For standard exceptions
using std::string;
using std::ios;

void validate_files(string source, string target); // Validate the files
int main(int argc, char* argv[])
try
{
    // Verify correct number of arguments
    if (argc != 3)
        throw std::invalid_argument {"Input and output file names required.\n"};

    // Check for output file identical to input file
    const string source {argv[1]};                // The input file
    const string target {argv[2]};                // The destination for the copy
    if (source == target)
        throw std::invalid_argument {string("Cannot copy ") + source + " to itself.\n"};
    validate_files(source, target);

    // Create file streams
    std::ifstream in {source, ios::in | ios::binary};
    std::ofstream out {target, ios::out | ios::binary | ios::trunc};
```

```

// Copy the file
char ch {};
while (in.get(ch))
    out.put(ch);

if (in.eof())
    std::cout << source << " copied to " << target << " successfully." << std::endl;
else
    std::cout << "Error copying file" << std::endl;
}
catch (std::exception& ex)
{
    std::cout << std::endl << typeid(ex).name() << ": " << ex.what();
    return 1;
}

// Verify input file exists and check output file for overwriting
void validate_files(string infile, string outfile)
{
    std::ifstream in {infile, ios::in | ios::binary};
    if (!in) // Stream object
        throw ios::failure {string("Input file ") + infile + " not found"};

    // Check if output file exists
    std::ifstream temp {outfile, ios::in | ios::binary};
    if (temp)
    { // If the file stream object is ok then the output file exists
        temp.close(); // Close the stream
        std::cout << outfile << " exists, do you want to overwrite it? (y or n): ";
        if (std::toupper(std::cin.get()) != 'Y')
        {
            std::cout << "Destination file contents to be kept. Terminating..." << std::endl;
        }
    }
}
}

```

This program requires the name of the input file and the name of the output file as command-line arguments. I entered this on the command line on my Microsoft Windows system with the current directory as the one containing Ex17_04.exe:

```
Ex17_04.exe D:\Example_Data\primes.txt D:\Example_Data\primes_copy.txt
```

I got the following output:

```
D:\Example_Data\primes.txt copied to D:\Example_Data\primes_copy.txt successfully.
```

I also tried this:

```
Ex17_04.exe Ex17_04.exe Ex17_04_copy.exe
```

This generated the following output:

```
Ex17_04.exe copied to Ex17_04_copy.exe successfully.
```

The duplicate .exe file was in the current directory after executing the copy too, which is most encouraging! This execution shows that using a file name without a path refers to a file in the current directory. Of course, it's more than likely that your system environment already provides a copy function but it's nice to be able to create one for yourself.

The array `argv` will have `argc` elements, the first of which will contain the program name. Thus, `argv` should have three elements, accommodating the program name plus the two file names, so you first verify that to be the case in `main()`. If there are no command-line arguments, you throw a standard exception of type `invalid_argument` that will be caught by the catch block at the end of `main()`. Having checked the arguments, you then assign them to a pair of string objects that will make for easier manipulation and recognition in the remainder of the code.

You don't want to be copying a file to itself, so you check that the files aren't the same by comparing them. If they are the same, you don't continue—you throw another exception. Once you're past the validity checks on the command-line arguments, there still could be problems with the source and destination files so the `validate_files()` function is called to check them out.

The `validate_files()` function first verifies that the source file exists by creating an `ifstream` object for it. If the file does not exist, the `!` operator applied to the stream object will return `true`, in which case an exception is thrown that will be caught by the catch block in `main()`. It's possible that the output file exists, so it's important to verify that it can be overwritten when that is the case; it could be disastrous the blindly go ahead with overwriting an existing file. Creating an `ofstream` object will create the file if it doesn't exist so an output file stream is no help. However, you can determine whether a file exists by creating an input stream object for it and using the stream object as an `if` expression. As you saw earlier, this implicitly calls operator `void*()` for the object, which returns a non-null pointer if the file exists; the resultant pointer is implicitly converted to type `bool`. If the file does exist, you offer the option of overwriting the file or terminating the copy operation.

If there are any problems with the source and target files for the copy operation, the `validate_files()` function throws an exception. It only returns normally to `main()` when all is well. The code for copy operation is very straightforward. After creating the `ifstream` and `ofstream` objects for the input and output files respectively, the copy occurs in the while loop. Because the copying is character by character, regardless of what the characters are, the operation works with any file contents. The `get()` function call for the `in` object will return `false` when EOF is read, thus ending the copy process.

Writing Numeric Data in Binary

In binary mode data is written and read as a sequence of bytes, regardless of what the data is. Numerical data can be written as binary values with no conversion to a string representation. Writing binary avoids the errors that can be introduced by converting binary floating-point values to decimal representation, and the data takes up less space in the file. Because you don't need whitespace to separate data items in the file, the file will be shorter and therefore faster to read. As long as you know what kind of data was written, you can read exactly what you wrote.

You need to be cautious when you're reading binary data on a different sort of computer from where it was written. There can be differences in the representations of binary floating-point values, and binary integers can be stored differently on different machine architectures; you learned in Chapter 1 about big-endian and little-endian ways of representing binary integers.

There are no stream functions that write numerical values of fundamental types to a file in binary, or read them back; the binary operations only read and write bytes. However, you can write your own, as I'll demonstrate. The file streams have the `read()` and `write()` function members that just read and write bytes:

```
basic_istream& read(char* str, std::streamsize count);    // Read count bytes into str
basic_ostream& write(const char* str, std::streamsize count); // Write count bytes from str
```

`std::streamsize` is an integer type that is defined in the `ios` header. These functions read or write `count` bytes starting at `str`. Obviously, `read()` is an `ifstream` member and `write()` is an `ofstream` member and the `fstream` class has both members.

You can use the `read()` and `write()` members to implement a set of functions that will write any of the numerical types as binary data. The best way to see how this might work is, as ever, to consider an example.

Suppose you want to write values of type `double` to a file. You could implement your own `write()` function to do this:

```
void write(std::ostream& out, double value)
{
    out.write(reinterpret_cast<char*>(&value), sizeof(double));
}
```

To write a floating-point value to a file, you write the sequence of bytes that the value occupies in memory to the file. This is illustrated in Figure 17-4, which assumes that type `double` occupies 8 bytes.

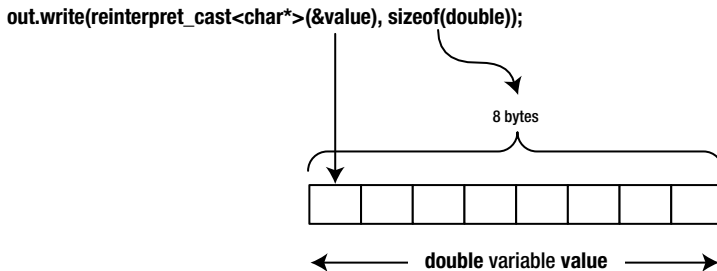


Figure 17-4. Writing a double value to a stream in binary mode

You force the conversion of the address of the first byte of the `double` value to type `char*`, and pass that to the `write()` member of the stream object, `out`. The `reinterpret_cast<>()` operator only alters the *interpretation* of what the pointer points to, without changing the data it points to in any way. The `sizeof` operator provides the number of bytes to be written for type `double`, so you pass that value to the `write()` member of the stream object as the count of the number of bytes to be written.

Clearly, you can write any of the numeric types to a stream in exactly the same way, so you could define a template that will generate these functions when required. Such a template could potentially create functions for class types, creating the illusion that the templates would work with objects of any class type. Unfortunately, this isn't the case. For example, a class object that contained a pointer as a data member wouldn't be valid when it was read back

from a file because the address in the pointer would certainly be invalid. You can prevent the template from being misused by applying `static_assert()` in the way you saw in Chapter 16. The `type_traits` header defines templates that you can use to ensure the template is only instantiated for numeric types. Here's the function template for binary output of numeric data:

```
#include <type_traits>                                // For is_arithmetic

template <typename T>
void write(std::ostream& out, T value)
{
    static_assert(std::is_arithmetic<T>::value, "Only for use with numeric types.");
    out.write(reinterpret_cast<char*>(&value), sizeof(T));
}
```

The `is_arithmetic<T>` template results in `true` only if `T` is an integer or floating-point type. It will result in `false` causing an assertion for any other type, including class types, and pointer and reference types.

A function template to generate functions to read binary values from a file is also easily defined:

```
template <typename T>
void read(std::istream& in, T& value)
{
    static_assert(std::is_arithmetic<T>::value, "Only for use with numeric types.");
    in.read(reinterpret_cast<char*>(&value), sizeof(T));
}
```

This reads the number of bytes that a variable of type `T` occupies into the memory locations occupied by `value`. The second parameter is a reference to allow the bytes read from the file to be stored at the location identified by the argument.

Notice that you use a reference to an `istream` object as the first parameter type here, rather than a reference to an `ifstream` object. Similarly, with the `write()` function you use `ostream&` as the type for the stream parameter. Although you'll only use these functions with file streams, making the parameters references to file stream types has a distinct disadvantage. The `fstream` class is derived from `iostream`, and so has `istream` and `ostream` as indirect base classes. It isn't derived from `ifstream` or `ofstream`. By using `istream&` and `ostream&` as the parameter types in the templates, you ensure that both will work with `fstream` as well as `ifstream` in the case of `read<T>()`, or `ofstream` in the case of `write<T>()`.

Defining function templates to read and write *arrays* of values of the basic types is not difficult, although you must include a function parameter to specify the array length when the array is written, because the array length cannot be deduced from an argument:

```
template <typename T>
void write(std::ostream& out, T* values, size_t length)
{
    write(out, length);
    for(size_t i {} ; i< length ; ++i)
        write(out, values[i]);
}
```

This first writes the array length, followed by the array elements using instances of the function template for writing a single numeric value. If `T` is not numeric, the instances of the function template for writing single value will cause an assertion, so the instance of this template will not compile. You could just write the element values but writing the array length first will allow the `read<T>()` template to avoid reading more elements than were written to the file. It is up to the user to ensure that the values array has sufficient elements.

Here the template for functions to read an array of elements of any numeric type:

```
template <typename T>
size_t read(std::istream& in, T* values)
{
    size_t length {};
    read(in, length);
    size_t i {};
    for( ; i < length ; ++i)
    {
        read(in, values[i]);
        if(!in) break;
    }
    return i;
}
```

This uses the templates for reading single values. The `static_assert()` in those will prevent instances of this template from compiling when `T` is not a numeric type. I have included a check for a stream error in the loop. This will include a check for a premature EOF that would cause fewer elements to be read than were expected. This should not happen, but returning the number of values read will allow the caller to recognize when this occurs and know how many there are.

Templates for functions to write and read `std::vector<T>` and `std::array<T,N>` containers are easy. Here are templates for array containers:

```
#include <array>
#include <type_traits>

// Write arrays of numeric values
template <typename T, size_t N>
void write(std::ostream& out, std::array<T, N>& values)
{
    write(out, N);
    for(size_t i {} ; i < N ; ++i)
        write(out, values[i]);
}

// Read arrays of numeric values
template <typename T, size_t N>
size_t read(std::istream& in, std::array<T, N>& values)
{
    size_t length {};
    read(in, length);
    if(N < length) throw std::invalid_argument {"Too few elements in array container."};
    size_t i {};
    for( ; i < length ; ++i)
    {
        read(in, values[i]);
        if(!in) break;
    }
    return i;
}
```

The user is responsible for creating the array container for both input and output operations. It's also up to the user to verify that it's OK to overwrite an output file when it already exists. Simple, isn't it?

The templates for `vector<T>` containers are also simple:

```
#include <vector>
#include <type_traits>

// Write vectors of numeric values
template <typename T>
void write(std::ostream& out, std::vector<T>& values)
{
    write(out, values.size());
    for(auto x : values)
        write(out, x);
}

// Read vectors of numeric values
template <typename T>
size_t read(std::istream& in, std::vector<T>& values)
{
    size_t length {};
    read(in, length);
    T value {};
    size_t i {};
    for( ; i < length ; ++i)
    {
        read(in, value);
        if(!in) break;
        values.push_back(value);
    }
    return i;
}
```

If you put all these templates together in a header file with `#include` directives for the `array`, `vector`, `fstream`, `type_traits` and `stdexcept` headers, you can try it out with an example. To exercise all the templates need quite a lot of code. Although the code is a bit repetitive, I'll explain the contents of `main()` piecemeal. The program will assume that the `primes.txt` file that was created by `Ex17_03.cpp` still exists in the `Example_Data` directory and contains at least 20 primes. First, the header files and using directives in `Ex17_05.cpp`:

```
// Ex17_05.cpp
// Using function templates for numeric I/O
#include <cctype> // For character functions
#include <fstream> // For file streams
#include <iostream> // For standard streams
#include <iomanip> // For stream manipulators
#include <array> // For array container template
#include <vector> // For vector container template
#include <string> // For string type
#include "Binary_Numeric_IO.h" // For numeric I/O function templates
using std::ios;
using std::string;
using ulong = unsigned long long;
```

The templates for numeric I/O functions should be in the `Binary_Numeric_IO.h` header. The `main()` function will make use of a helper function that checks whether a file exists. The code for it is:

```
void check_output_file(string filename)
{
    std::ifstream in(filename);
    if (in)
    {
        std::cout << filename << " exists. Overwrite(Y or N)?: ";
        char reply {};
        std::cin >> reply;
        std::cin.ignore();                // Remove EOF
        if (std::toupper(reply) != 'Y')
            std::exit(1);
    }
}
```

To check whether the file identified by the argument exists, a file input stream is created from the file name. If the file exists, the implicit cast of the `in` object will result in `true`. In this case there's a prompt for permission to overwrite the file. If the answer is not 'y' or 'Y', the program is terminated by calling `exit()`. Reading a single character from a stream using `get()` leaves the EOF that results from pressing the Enter key in the stream. If you place the code above after `main()` in `Ex17_05.cpp`, don't forget to put a prototype for the function before `main()`.

The `check_output_file()` function will be called several times in `main()`. A subsequent read from `cin` will fail if EOF is left in the stream; calling `ignore()` for the stream object removes the EOF. The function template to read from a file into an `array<T,N>` container can throw exceptions, so the body of `main()` will be a try block:

```
int main()
try
{
    // Code for main()...
}
catch (std::exception& ex)
{
    std::cout << ex.what() << std::endl;
}
```

The first block of code in `main()` reads the `primes.txt` file:

```
const size_t primes_count {20};
ulong primes[primes_count];
string primesfile {"D:\\Example_Data\\primes.txt"};
std::ifstream in {primesfile};
if (!in) throw ios::failure{primesfile + " not found in main()."};
for (size_t i {} ; i < primes_count ; ++i)
{
    in >> primes[i];
    if (in.eof()) throw std::runtime_error {string {"not enough primes in "} +primesfile};
}
in.close();
```

This defines an array of `primes_count` elements of type `unsigned long long` and fills the array from the data in `primes.txt`. The file was written in text mode so it is read using the `>>` operator for the stream. It's possible that the file with the name and path specified by `primesfile` does not exist - a typo in the initial value for `primesfile` would cause that for example, so there's a check to verify that the file does indeed exist. If it doesn't, an exception is thrown that will be caught by the catch block for `main()`.

The next block of code in `main()` writes the contents of the `primes` array to a file in binary mode:

```
string binaryfile {"D:\\Example_Data\\primes.bin"};
check_output_file(binaryfile);
std::ofstream out {binaryfile};
write(out, primes, primes_count);
out.close();
std::cout << binaryfile << " written." << std::endl;
```

This calls the helper function to verify that it's OK to overwrite the output file if it exists. The data in the array is written to the file by an instance of the `write<T>()` function template for arrays. After writing the file, the stream and the file are closed, so the file can be reopened subsequently in a different mode.

Just to demonstrate that the binary file contains the same data as the original text file, the next block of code reads it back into another array and output the data to `cout`:

```
ulong primesback[primes_count];
in.open(binaryfile, ios::binary | ios::in);
if (!in) throw ios::failure {binaryfile + " not found in main() ."};
read(in, primesback);
in.close();
size_t perline {6};
std::cout << "Primes read into array:\n";
for (size_t i {}; i < primes_count; ++i)
{
    std::cout << std::setw(10) << primesback[i];
    if (!((i + 1) % perline)) std::cout << '\n';
}
std::cout << std::endl;
```

This uses the same file input stream object, `in`, that was used to read the text file to open a binary file to read it. The file is read using an instance of the function template for reading from a binary file into an array. This template will use an instance of the function template that reads a single value from a binary file. There's a check that the input file does exist before reading it. The contents of the array are written to `cout`, so the output will show whether or not everything is working as it should.

The next block of code reopens the binary file and reads the contents into a `vector<T>` container:

```
in.open(binaryfile, ios::binary|ios::in);
if (!in) throw ios::failure {binaryfile + " not found in main() ."};
std::vector<ulong> primes_vector;
read(in, primes_vector);
in.close();
```

This uses the same file input stream object to reopen the binary file to read it. The data is read into the vector using an instance of another of the function templates. The stream is closed, which closes the file, so the file can be reopened later to read it or write it.

The data in the vector is written to a new binary file by the next block of code in `main()`:

```
string vectorfile {"D:\\Example_Data\\primesvector.bin"};
check_output_file(vectorfile);
out.open(vectorfile, ios::binary | ios::out);
write(out, primes_vector);
out.close();
std::cout << vectorfile << " written." << std::endl;
```

This is similar to what happened previously, but uses an instance of another function template that writes the contents of a `vector<T>` container to a binary file. The file is closed and then reopened to read it by the next code in `main()`:

```
in.open(vectorfile, ios::binary | ios::in);
if (!in) throw ios::failure {vectorfile + " not found in main()."};
std::array<ulong, primes_count> primesarray;
read(in, primesarray);
in.close();
```

Here, the data that was written from the vector to the file is read back into an `array<T,N>` container. This results in yet another template instance created from the `read<T,N>()` template for reading into an array container. Of course, to show that the `write<T,N>()` template also works, we write the array container to another file:

```
string arrayfile {"D:\\Example_Data\\primesarray.bin"};
check_output_file(arrayfile);
out.open(arrayfile, ios::binary | ios::out);
write(out, primesarray);
out.close();
std::cout << arrayfile << " written." << std::endl;
```

Finally, to show that nothing was lost along the way, we read the file that was just written and output the contents:

```
size_t count {};
in.open(arrayfile, ios::binary | ios::in);
if (!in) throw ios::failure {arrayfile + " not found in main()."};
read(in, count);
ulong prime {};
for (size_t i {}; i < count; ++i)
{
    read(in, prime);
    std::cout << std::setw(10) << prime;
    if (!(i + 1) % perline) std::cout << '\n';
}
std::cout << std::endl;
```


We know that the first item in the file is the count of the number of prime values that follow, so that is read first and used to control reading of the prime values in the `for` loop. That's a lot of code but all the templates for transferring numerical data to and from a binary file have been used, so the output will show whether all the templates work. Remember, if you don't create a template instance, the template will not be compiled and could contain coding errors. I got this output:

```
D:\Example_Data\primes.bin written.
Primes read into array:
    2      3      5      7      11      13
   17     19     23     29     31     37
   41     43     47     53     59     61
   67     71
D:\Example_Data\primesvector.bin written.
D:\Example_Data\primesarray.bin written.
    2      3      5      7      11      13
   17     19     23     29     31     37
   41     43     47     53     59     61
   67     71
```

This shows that all the templates work. If you run the example a second time, you will get prompts to ask whether you want to overwrite the binary files that now exist.

File Read/Write Operations

You can open a stream so that you can carry out both input and output operations with a file. `fstream` objects specifically support both input and output operations. This will sometimes involve changing the file position frequently. As you saw early on in this chapter, `fstream` inherits from `iostream`, which in turn inherits from `istream` and `ostream`, so all the input and output functions discussed so far are available for an `fstream` object. You can create an `fstream` object with a file name as the constructor argument, just like an `ifstream` or `ofstream` object. For example:

```
string filename {"D:\\Example_Data\\primes.txt"};
std::fstream bothways {filename};
```

The default open mode is `ios::in|ios::out` and like the other file streams, it will be in text mode by default. To create a stream operating in binary mode, you specify an argument for the second parameter, exactly as you've done previously. For example:

```
string filename {"D:\\Example_Data\\primes.bin"};
std::fstream bothways {filename, std::ios::in|std::ios::out|std::ios::binary};
```

This opens the file for both input and output operations in binary mode. If the file can't be opened for any reason, `ios::failbit` will be set. Note that you can't use `ios::app` with `fstream` objects, but you can use `ios::trunc`, which will discard any previous file contents. In fact, you can't use `ios::app` in combination with `ios::in` at all, so you can't use it for `ifstream` objects either. This isn't unreasonable, because `ios::app` implies that you'll *write* at the end of the file, which isn't particularly meaningful for an input stream.

The default constructor creates a stream object with no associated file. You can then open a particular physical file by using the `open()` member of the `fstream` object, for example:

```
std::fstream inout;
inout.open(filename, std::ios::in|std::ios::out|std::ios::binary|std::ios::trunc);
```

The first statement creates an `fstream` object without associating it with a file. The second statement opens the file specified by `filename` for both input and output in binary mode and discards any existing file contents. If you omit the second argument, the default open mode is the same as for the constructor: `ios::in|ios::out`. If you're both writing and reading a file, then almost by definition you'll want to read and/or write at random positions within the file, so let's look at how you do that.

Random Access to a File

I'll discuss random access to a file in the context of using an `fstream` object in binary mode but you can apply the same techniques to reading an `ifstream` or to updating an `ofstream` in binary mode. Once you have opened a file stream for input and output, you can read from or write to any position in the stream. However, you do need to know where you are in the stream, and where you want to go next.

Random Access to a Binary Stream

As I have said, you can open an `fstream` for both reading and writing, however, there's a slight catch: Whenever you switch from reading the file to writing it, and vice versa, you must either flush the stream or execute a seek to set the file position; executing a seek carries less overhead so it's usually better to seek. You call `seekp()` to switch from read to write operations and `seekg()` to switching from writing a file to reading it. If you want to retain the current position when switching between reads and writes, you can obtain the current position by calling `tellg()` or `tellp()` and passing that as the argument to the seek function. You know all you need to know to be able to read and write a file at arbitrary position but an example will show how it all hangs together.

Random File Operations in Practice

We can write another example to write prime numbers to a file. This time we'll use a binary file and provide the option of requesting a particular prime—the 25th or 432nd prime in sequence, for instance. The idea is that if the prime is in the file, the program should fetch it and display it. If it isn't, the program should calculate new primes up to the one required and add them to the file. All the examples of file I/O in this chapter have been procedural, so this time I'll take an object-oriented approach and define a `Primes` class to provide the capability:

```
#include <fstream>
#include <string>
using std::string;
using ulong = unsigned long long;

class Primes
{
private:
    std::fstream primesfile;           // The file stream - input & output
    size_t nprimes {};                 // Number in the file
    ulong lastprime {};                // Last prime in the file
};
```

```

bool file_exists(string file);           // Returns true if a file exists
bool isprime(ulong n);                  // Returns true if n is prime

public:
    Primes(string filename = "D:\\Example_Data\\primes_cache.bin");
    ~Primes() { primesfile.close(); }

    ulong prime_after(ulong n);          // Next prime after an integer
    ulong next_prime(ulong prime);       // Next prime after a prime
    ulong operator[](size_t n);          // nth prime (indexed from 0)
};

```

The data members that record the number of primes in the file and the last prime in the file are for convenience and to improve efficiency a little; they'll be recorded in the file too so it's not absolutely essential to store them as data members. Knowing the last prime without having to read the file will make it easy to decide whether or not a prime that is requested is in the file. The approach for finding primes will not be particularly efficient in any event though because the class interface hides a lot of complexity.

The `fstream` member will encapsulate the file containing the primes. The constructor parameter has a default specification for the file so if you don't supply an argument, the same file will be used each time. The file contents will be a little different from previous examples. The number of primes in the file will be recorded as the first item of data. This will allow the `Primes` constructor to initialize the data members easily. All data will be written in binary mode. The destructor just closes the file.

The application of the function members is fairly self-explanatory. You can obtain a prime using an index with the subscript operator. The primes are indexed from 0 for consistency with indexing generally but there are arguments for indexing from 1. It would be easy to implement this as an option. The `next_prime()` member will return the prime that follows the argument, which must be a prime. The `prime_after()` member will return the next prime that is greater than an arbitrary integer. The `isprime()` and `file_exists()` members are for use by other function members and therefore private.

All the input and output operations will be carried out by instances of the templates that were defined for `Ex17_05` in the `Binary_Numeric_IO.h` header, so that will be included in this program.

Implementing the Constructor

The constructor is responsible for creating the `fstream` object that encapsulates the file specified by the argument and initializing the data members. It will be necessary for the constructor to first decide whether the file already exists. Here's the code:

```

Primes::Primes(string filename)
{
    if (!file_exists(filename))           // If no file...
    {
        std::ofstream out {filename, ios::binary|ios::out};    // ...create it
        nprimes = 3;
        lastprime = 5ULL;
        write(out, nprimes);
        write(out, 2ULL);
        write(out, 3ULL);
        write(out, 5ULL);
        out.close();
        primesfile.open(filename, ios::binary | ios::in | ios::out);
    }
}

```

```

{
    primesfile.open(filename, ios::binary | ios::in | ios::out);
    read(primesfile, nprimes);
    primesfile.seekg(sizeof(nprimes) + (nprimes-1)*sizeof(ulong), primesfile.beg);
    read(primesfile, lastprime);
}
}

```

The `file_exists()` member checks whether there is a file. If the file doesn't exist, it's created by the `ofstream` object, `out`, and the first three primes are written preceded by the prime count. After closing `out`, the file is opened using the `fstream` object that is a member by calling `open()` for it. The file is then ready for use by the function members.

Checking for the Existence of a File

The code in the `else` block executes when the file does exist. The `nprimes` member by immediately reading the file because the file is opened at the beginning. To initialize `lastprime`, the `seek` operation sets the position as an offset from the first bytes in the file before reading the value. The offset is the length of the prime count value plus the lengths of the two prime values that precede the last value.

The `file_exists()` member is very simple:

```

bool Primes::file_exists(string file)
{
    std::ifstream stream {file};
    return static_cast<bool>(stream);
}

```

The process is to create a file input stream object. If the file does exist, casting the `ifstream` object to `bool` will result in `true`; if it does not exist the result will be `false`.

The member to find a prime that is greater than a given integer is not too difficult:

```

ulong Primes::prime_after(ulong n)
{
    ulong prime {};
    if (n < lastprime)
    {
        primesfile.seekg(sizeof(nprimes), primesfile.beg);
        while (prime < n) read(primesfile, prime);
    }
    else
    {
        while((prime = next_prime(lastprime)) < n);
    }
    return prime;
}

```

The value to be returned will be stored in the local variable, `prime`. If the argument is less than `lastprime`, which is the last prime in the file, we just need to search the file to find the first prime that is greater than the argument. After moving the file position to the first byte following the count, the `while` loop reads primes from the file until one is found that is greater than the argument. If the argument is not less than `lastprime`, it is necessary to generate more primes until one is discovered that is greater than `n`. Successive primes are created by calling the `next_prime()` member. This function will find the prime that is greater than the argument, write it to the file, and update the count recorded in the file. It will also update the `nprimes` and `lastprime` members. We end up with all new primes added to the file.

Finding a Prime that follows a Prime

The `next_prime()` member implementation looks like this:

```

ulong Primes::next_prime(ulong prime)
{
    if (prime < lastprime)
    { // Next prime must be in the file
        primesfile.seekg(sizeof(nprimes), primesfile.beg);
        ulong current {};
        for (size_t i {}; i < nprimes; ++i)
        {
            read(primesfile, current);
            if (prime < current) return current;
        }
        throw std::logic_error {string {"next_prime() fail. File contents incorrect or "} +
                                   std::to_string(prime) + " not a prime"};
    }

    // The next prime is not in the file
    ulong trial {lastprime + 2ULL};
    do
    {
        while (!isprime(trial)) trial += 2ULL;
        lastprime = trial;
        primesfile.seekp(0, primesfile.end);           // File position to the end
        write(primesfile, lastprime);
        ++nprimes;
        primesfile.seekp(0);                           // File position to the beginning
        write(primesfile, nprimes);
        trial += 2ULL;
    } while (lastprime <= prime);
    return lastprime;
}

```

There's more code to this, but it's not difficult. The `if` block deals with the situation when the argument `prime` is less than `lastprime`. In this case the next prime must be in the file - in the worst case it's `lastprime`. The `for` loop search for the first prime in the file that is greater than `prime` and returns it. In the unlikely event of a prime not being found in the file, an exception is thrown that indicates that there must be something wrong with the file or the function argument.

The `do-while` loop deals with the situation when `lastprime` is not greater than the argument. In this case, more primes must be found until one is found that is greater than the argument. This process is essentially what you have seen before. The prime candidate is in `trial`, which is `lastprime+2` initially. The `isprime()` member is called to check whether `trial` is prime. Each new prime is written to the file and the count at the beginning of the file is updated, along with the `nprimes` and `lastprime` members. The loop continues until `lastprime` is greater than the argument, `prime`.

Checking for a Prime

We just need a definition for the `isprime()` member:

```
bool Primes::isprime(ulong n)
{
    ulong root_n {static_cast<ulong>(std::sqrt(n))};
    ulong prime {};
    primesfile.seekg(sizeof(nprimes), primesfile.beg);

    while (primesfile)
    {
        read(primesfile, prime);
        if ((n % prime) == 0) return false;
        if (prime > root_n) return true;
    }
    throw std::logic_error {string {"isprime() fail. Could not determine primeness of "} +
                             std::to_string(n)};
}
```

This assumes that `lastprime` is always greater than the square root of the argument, `n`. Barring errors in the code, this should always be the case because it is a `private` function that is only used inside the class. Before checking begins, the file position is moved to the first byte of the first prime in the file, which follows the count. The `while` loop determines whether `n` is prime by reading primes and using them as divisors. This continues until either there is an exact division - in which case `n` is not prime, or until all primes less than the root of `n` have been tried as divisors, in which case `n` is prime. The `throw` statement is belt and braces - it should never be executed. There's a serious program error somewhere if it is.

Implementing the Subscript Operator

The subscript operator will return the prime that is the index, where the index value for the first prime is 0. Of course, the `n`th prime may or may not be in the file, so the operator function must deal with this:

```
ulong Primes::operator[](size_t n)
{
    ulong prime {};
    if (n < nprimes)
    { // The nth prime is in the file
        primesfile.seekg(sizeof(nprimes) + (n*sizeof(ulong)), primesfile.beg);
        read(primesfile, prime);
        return prime;
    }

    // If we get to here, the nth prime is not in the file
    while (nprimes < n + 1)
    {
        lastprime = next_prime(lastprime);
    }
    //We have found the nth prime
    return lastprime;
}
```

It's a surprisingly easy function to implement. If `nprimes` is greater than the index, the prime is in the file, so we just move the file position to the first byte of the *n*th prime and read it. You can see the expression for the offset for the seek operation; it's the index times the size of a prime value, plus the size of the count. If there are too few primes in the file, `next_prime()` is called to add a prime until there are *n*+1 primes in the file (this is because the index starts at 0). The last prime in the file is then returned.

Using the Primes Class

We just need a program to try out the `Primes` class:

```
// Ex17_06.cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include "Primes.h"

int main()
{
    Primes primes;
    std::cout << "3rd prime is " << primes[2] << std::endl;
    std::cout << "5th prime is " << primes[4] << std::endl;
    ulong prime {primes.prime_after(50)};
    std::cout << "Prime greater than 50 is " << prime << std::endl;
    std::cout << "Prime following " << prime << " is " << primes.next_prime(prime) << std::endl;
    std::cout << "Primes between 100 and 300 are:\n";
    std::vector<ulong> values;
    prime = primes.prime_after(100);
    while (prime < 300)
    {
        values.push_back(prime);
        prime = primes.next_prime(prime);
    }
    size_t count {};
    size_t perline {6};
    for (auto p : values)
    {
        std::cout << std::setw(12) << p;
        if (
            (++count % perline) == 0) std::cout << std::endl;
    }
    std::cout << std::endl;
    std::cout << "500th prime is " << primes[499] << std::endl;
}
```

The complete code for the example is in the code download. This tries the various ways of using a `Primes` object and produces the following output:

```

3rd prime is 5
5th prime is 11
Prime greater than 50 is 53
Prime following 53 is 59
Primes between 100 and 300 are:
    101      103      107      109      113      127
    131      137      139      149      151      157
    163      167      173      179      181      191
    193      197      199      211      223      227
    229      233      239      241      251      257
    263      269      271      277      281      283
    293
500th prime is 3571

```

The code in `main()` is very straightforward so you should have no problem seeing what is happening. The output demonstrates that the `Primes` class works as it should. The example shows how easy it is to encapsulate file operations within a class and how simple random read and writes are, as long as you obey the rules for file I/O.

String Streams

There are three *string stream classes* that connect a stream to a `string` object in memory. This enables you to read or write a `string` as though it was a file. The string stream classes are `istringstream`, `ostringstream`, and `stringstream`, which have `istream`, `ostream`, and `iostream` as base classes, respectively. Operations on these classes are essentially the same as for the file streams, except of course that the input and output operations are to `string` objects.

Although they can use any of the I/O functions that are inherited from their corresponding base class, string streams are used most often with the insertion and extraction operators. The reason for this is that their primary application is formatting data in memory, or analyzing input. For example, you might have an application in which the format of the input isn't known in advance. In such a case, you could read the data as a sequence of characters into a `string` object, and then use the stream input operations with an `istringstream` object attached to your `string` object to carry out formatted read operations on it. This provides the possibility to read the input as many times as necessary to figure out its format.

Suppose you read a line of input from `cin` with these statements:

```

string buffer;
getline(std::cin, buffer);

```

Having read the input into `buffer`, you can create an `istringstream` object with the following statement:

```

std::istringstream inStr {buffer};

```

You can now read from `buffer` via the `inStr` stream just like any other stream and make use of the conversion capability from character representation to binary:

```

long value {};
double data {};
inStr >> value >> data;

```


You can use an `ostringstream` object to format data into a string. For instance, you could create a string object and an output string stream with these statements:

```
string outBuffer;
std::ostringstream outStr {outBuffer};
```

You can now use the insertion operators to write to `outBuffer` via `outStr`:

```
double number {2.5};
outStr << "number = " << (number/2.0);
```

As a result of the write to the string stream, `outBuffer` will contain `"number = 1.25"`. The string `outBuffer` will automatically expand to accommodate however many characters you write to the stream, so it is a very flexible way of forming strings or complex output messages.

The string parameter to the string stream constructors is a reference in each case, so write operations for the `ostringstream` and `stringstream` objects act directly on the string object. There is also a default constructor for each of the string stream classes. When you use these, the string stream object will maintain a string object internally, and you can obtain a copy of this using the `str()` member, for example:

```
std::ostringstream outStr;
double number {2.5};
outStr << "number = " << (3 * number/2);
string output {outStr.str()};
```

After these statements have been executed, `output` will contain the string `"number = 3.75"`.

Objects and Streams

In previous chapters I've been telling you how great object-oriented programming is, so inevitably the question of writing objects to a file arises. The process of writing objects to a stream so they can be retrieved is called *serialization*. *Serializing* an object is writing it to a stream and you *deserialize* it to get it back. So what assistance is there for you to implement input and output of objects? As far as the C++ Standard Library is concerned - nil, zip, nothing - you're on your own.

Although this is inconvenient, it's not altogether surprising. By nature, defining a class is completely open ended so inevitably an object of an arbitrary class type is an unknown quantity. Data members of a class can be of fundamental types, class types and pointer types. Input and output operations for objects are going to be class specific, whether we like it or not.

Serialization is a large and complex topic. However, I think it's important to have an inkling of what is involved when you are starting out in C++ so this section provides that. In spite of the difficulties implicit in generalized serialization, some C++ development systems do provide a framework for supporting it. If your development environment doesn't, you *can* do something yourself. Serialization can mean different things depending on the context. Writing an object to a stream so that an environment that is not C++ can access and use it is a completely different problem from writing an object so you can read it back later in the same program. I'll illustrate how simple the latter can be through examples.

Using the Insertion Operator with Objects

You have already seen how to implement an overloaded version of `operator<<()` to write `Box` objects to a stream; this was for writing an object to the standard output stream. In general this can be done for any class, either as a global function if there are accessory functions for the data members, or as a friend of the class if there are not. In most cases it's quite straightforward. The fundamental question is usually *what* you will output for a given class type,

and this depends on *why* you are implementing text mode output for it. The answer may be different depending on whether you are writing an object to a stream in order to read it back later, or whether you just want a human readable representation of an object on the standard output stream. The latter is usually a primary reason to implement operator<<() for a class. In general, what you write to cout to represent an object is unlikely to be what you need in a file stream to read the object back.

Let's consider the specific case of the Primes class in the previous example. Writing a Primes object to the standard output stream by overloading the insertion operator is probably not a good solution. A Primes object may encapsulate a file that contains a large number of primes. You are unlikely to need a general capability for outputting these en masse as text, and if you do, it's likely to be better to provide the capability by defining a member function such as list_primes(). It also likely to be a good idea to in this case to supplement this with a get_count() member to return the number of primes in the file. You really don't want to initiate output to your display if there are millions of them.

It's sometimes better to write objects to a file in binary mode when you want to recover the objects from a stream, especially when a class has one or more data members of a floating-point type. Binary mode preserves the integrity of floating-point data, which is not the case in text mode. Binary floating-point values are not always precisely representable in decimal form so the conversion of a value for output in text mode can introduce small errors. This implies that an object read back in text mode may not be identical to the object that was written. In a way, the Primes class already provides for serializing an object in binary mode. When you create a Primes object, the constructor synthesizes the object from data in the file if the file is present. I'll have more to say about object I/O in binary mode a little later. In the meantime, let's look at serializing objects in text mode.

Using the Extraction Operator with Objects

I'll show how you can implement the extraction operator using a variation on the Box class that you saw Chapter 12. The >> operator function needs two parameters, the istream object that is the source of the data, and a reference to the Box object where the data is to go. The left operand of the >> operator corresponds to the first parameter and the right operand is the second parameter. It will be a friend of the Box class so it can access the data members. Here's the Box class definition, including the operator functions:

```
class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructors
    Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv} {}

    Box()=default;                                     // Default constructor

    Box(const Box& box)                                 // Copy constructor
        : length {box.length}, width {box.width}, height {box.height} {}

    double volume() const                               // Calculate the volume
    { return length*width*height; }

    friend std::ostream& operator<<(std::ostream& stream, const Box& box);
    friend std::istream& operator>>(std::istream& in, Box& box);
};
```

The second parameter for `operator>>()` has to be a reference; it must not be `const` because the input operation will change it. I'll define `operator<<()` like this to make sure the input operation works reliably:

```
inline std::ostream& operator<<(std::ostream& out, const Box& box)
{
    return out << std::setw(10) << box.length << ' '
               << std::setw(10) << box.width << ' '
               << std::setw(10) << box.height << '\n';
}
```

Writing a space or a newline after each output value guarantees that each value will be separated by at least one whitespace character. This guards against the possibility that a value could be more than the specified field width. If there is no whitespace between successive values in the stream, they will be read by `operator>>()` as a single value. The final newline that is output ensures that each object is written in a separate line.

We can define the function for the extraction operator like this:

```
inline std::istream& operator>>(std::istream& in, Box& box)
{
    return in >> box.length >> box.width >> box.height;
}
```

This uses the standard extraction operator function to read values from the stream into the data members of `box`. The standard `operator>>()` function for extraction data of fundamental types returns a reference to the stream object so we can return that from our overload. The definitions for both functions are inline so they can go in `Box.h`, following the class definition. Simple, isn't it?

This will work with any stream include a file stream, so let's see it in action:

```
// Ex17_07.cpp
// Writing Box objects to a file and reading them back
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include "Box.h"
using std::string;

int main()
try
{
    std::vector<Box> boxes {Box {1.0, 2.0, 3.0}, Box {2.0, 2.0, 3.0},
                           Box {3.0, 2.0, 2.0}, Box {4.0, 2.0, 3.0},
                           Box {1.0, 4.0, 3.0}, Box {2.0, 2.0, 4.0}};

    const string filename {"D:\\Example_Data\\boxes.txt"};
    std::ofstream out {filename};
    if (!out)
        throw std::ios::failure {string {"Failed to open output file "} + filename};

    for (auto& box : boxes)
        out << box;                                // Write a Box object
    out.close();                                     // Close the input stream

    std::cout << boxes.size() << " Box objects written to the file." << std::endl;
```

```

std::ifstream in {filename};           // Create a file input stream
if (!in)                               // Make sure it's valid
    throw std::ios::failure {string("Failed to open input file ") + filename};

std::cout << "Reading objects from the file.\n";
std::vector<Box> newBoxes;
Box newBox;
while (true)
{
    in >> newBox;
    if (!in) break;
    newBoxes.push_back(newBox);
}
in.close();                           // Close the input stream
std::cout << newBoxes.size() << " objects read from the file:\n";
for (auto& box : newBoxes)
    std::cout << box;
}
catch (std::exception& ex)
{
    std::cout << typeid(ex).name() << ": " << ex.what() << std::endl;
}

```

Here's the output:

```

6 Box objects written to the file.
Reading objects from the file.
6 objects read from the file:
    1      2      3
    2      2      3
    3      2      2
    4      2      3
    1      4      3
    2      2      4

```

Clearly the objects read from the input file are those that were written, so file I/O in text mode for Box objects is working satisfactorily. There's a few points to note here. The while loop that reads objects from the file reads each object into the same variable, `newBox`. This is passed to the `push_back()` member of the vector by value so the Box copy constructor is called to create the object that is stored in the vector. The range-based for loop that lists the boxes in the `newBoxes` vector uses our `operator<<()` function with `cout`, so the example demonstrates that it works with different types of output stream. The loop to access the objects in the `newBoxes` vector has a reference variable, so the objects in the vector are not duplicated; the objects in the vector are accessed by reference to write them to the standard output stream.

Object I/O in Binary Mode

There are pros and cons for using binary mode to transfer objects to and from a file. Space in the file is minimized in binary mode and you are assured of getting back what was written - assuming the operations are implemented correctly. However, there are no delimiters for data in binary mode so reading a file where the sequence of data items can vary can be tricky to say the least.

You have seen how you can implement functions that read and write data of fundamental types. Supporting object input and output is essentially more of the same. Each class type needs functions that perform the read and write operations. Writing an object to a stream will ultimately boil down to writing data of fundamental types. Reading an object from a stream will always involve reading a sequence of data items of fundamental types, then reconstructing the original object. An essential requirement for writing an object to a file is that all its data members can be written. Of course, data members of an object can themselves be objects of class types and there must be provision for writing these objects and reconstructing them when the file is read. This can lead to a lot of complexity.

Let's take a simple example of binary file I/O with Box objects. Here's the Box class definition:

```
class Box
{
private:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructors
    Box(double lv, double wv, double hv) : length {lv}, width {wv}, height {hv} {}

    Box() = default;                                // Default constructor

    Box(const Box& box)                               // Copy constructor
        : length {box.length}, width {box.width}, height {box.height} {}

    double volume() const                            // Calculate the volume
    {
        return length*width*height;
    }

    friend std::ostream& operator<<(std::ostream& stream, const Box& box);
    friend std::istream& operator>>(std::istream& in, Box& box);

    friend std::ifstream& read(std::ifstream& in, Box& box);
    friend std::ofstream& write(std::ofstream& out, Box& box);
};
```

The binary file I/O operations are provided by friend functions, `read()` and `write()`. The first parameter is a file stream so these cannot be used for other stream types. Both functions return a reference to the stream, so the caller can use the return value to check the stream state. I left the text mode I/O functions in because we will need the text mode output in `main()` to output Box objects to `cout`. Here's the definition for the binary input function:

```
inline std::ifstream& read(std::ifstream& in, Box& box)
{
    in.read(reinterpret_cast<char*>(&box.length), sizeof(box.length));
    in.read(reinterpret_cast<char*>(&box.width), sizeof(box.width));
    in.read(reinterpret_cast<char*>(&box.height), sizeof(box.height));
    return in;
}
```

The code is very simple. There are three calls for the `read()` function for the stream that read the number of bytes occupied by each of the data members. The address of each data member is reinterpreted as a pointer of type `char*` because all binary input is read as a sequence of characters. The `write()` function is very similar:

```
inline std::ofstream& write(std::ofstream& out, Box& box)
{
    out.write(reinterpret_cast<char*>(&box.length), sizeof(box.length));
    out.write(reinterpret_cast<char*>(&box.width), sizeof(box.width));
    out.write(reinterpret_cast<char*>(&box.height), sizeof(box.height));
    return out;
}
```

Both functions are `inline` so the definitions should be in `Box.h`, following the class definition. Here's a `main()` function to test this:

```
// Ex17_08.cpp
// Writing & reading Box objects in binary mode
#include <fstream>                                // For file streams
#include <iostream>                                // For standard streams
#include <string>                                   // For string type
#include <vector>                                   // For vector container
#include "Box.h"
using std::string;

int main()
try
{
    std::vector<Box> boxes {Box {1.0, 2.0, 3.0}, Box {2.0, 2.0, 3.0},
                           Box {3.0, 2.0, 2.0}, Box {4.0, 2.0, 3.0},
                           Box {1.0, 4.0, 3.0}, Box {2.0, 2.0, 4.0}};

    const string filename {"D:\\Example_Data\\boxes.bin"};
    std::ofstream out {filename, std::ios::binary};
    if (!out)
        throw std::ios::failure {string {"Failed to open output file "} + filename};

    for (auto& box : boxes)
        write(out, box);                          // Write a Box object
    out.close();                                    // Close the output stream

    std::cout << boxes.size() << " Box objects written to " << filename << std::endl;

    std::ifstream in {filename, std::ios::binary};  // Create a file input stream
    if (!in)                                        // Make sure it's valid
        throw std::ios::failure {string {"Failed to open input file "} + filename};
}
```

```

std::cout << "Reading objects from the file.\n";
std::vector<Box> newBoxes;
Box newBox;
while (true)
{
    if(!read(in, newBox)) break;
    newBoxes.push_back(newBox);
}
in.close(); // Close the input stream
std::cout << newBoxes.size() << " objects read from " << filename << ":\n";
for (auto& box : newBoxes)
    std::cout << box;
}
catch (std::exception& ex)
{
    std::cout << typeid(ex).name() << ": " << ex.what() << std::endl;
}

```

This uses a vector containing the same Box objects as the previous example as the objects to be written to a binary file. The output file is opened in binary mode by specifying the open mode as `std::ios::binary`; `std::ios::out` is implicit in the stream type and does not need to be specified. Each object is written to the file by calling the `write()` function that is a friend of the Box class in a range-based for loop. The objects are read from the file in a similar loop. The `read()` function is called in the if expression that checks for EOF being reached to end the loop. The output demonstrates that everything is working:

```

6 Box objects written to D:\Example_Data\boxes.bin
Reading objects from the file.
6 objects read from D:\Example_Data\boxes.bin:
    1      2      3
    2      2      3
    3      2      2
    4      2      3
    1      4      3
    2      2      4

```

You can try append mode with this example. You can change the definition of the output file stream object to:

```
std::ofstream out {filename, std::ios::binary|std::ios::app};
```

Each time you execute the example, the objects will be appended to the file. The output of what is in the file will increase on each execution.

More Complex Objects in Streams

With the Box object, I deliberately chose a simple case. Handling derived class objects gets more complicated. You need a virtual input and output mechanism to ensure that derived class objects are handled properly, but there's no way you can make the insertion and extraction operator functions class members. The `operator>>()` and `operator<<()` functions cannot be member functions of your class because they overload the operator functions in the stream classes. They are binary operators, and binary operator functions that are class members can only have

one parameter - the right operand. You have no way to pass the stream object *and* the right operand to the function implemented as a member of your class. However, a global operator function *could* call a virtual class member that implements an I/O operation. I'll add function members to the Box class to perform I/O in text mode:

```
class Box
{
protected:
    double length {1.0};
    double width {1.0};
    double height {1.0};

public:
    // Constructors
    Box(double lv, double ww, double hv) : length {lv}, width {ww}, height {hv} {}

    Box() = default;                                // Default constructor

    Box(const Box& box)                               // Copy constructor
        : length {box.length}, width {box.width}, height {box.height} {}

    virtual double volume() const                    // Calculate the volume
    {
        return length*width*height;
    }

    // Member stream I/O functions
    virtual std::ostream& put(std::ostream& out) const // Stream output
    {
        return out << std::setw(10) << length << ' ' << std::setw(10) << width << ' '
            << std::setw(10) << height << '\n';
    }

    virtual std::istream& get(std::istream& in)        // Stream input
    {
        return in >> length >> width >> height;
    }
};
```

I've added two public virtual members to the class that perform the stream I/O operations. They use the operator functions implemented by the stream objects for fundamental types to transfer the data members, in the same way as the previous friend operator functions. The insertion and extraction operator functions no longer need to be friend functions because they will now call these functions that are in the public class interface.

You can implement the operator<<() function like this:

```
inline std::ostream& operator<<(std::ostream& out, const Box& box)
{
    return box.put(out);
}
```


This calls the virtual `put()` function member of the `Box` class to perform the output operation and returns the stream object. Similarly, the implementation of the `operator>>()` function will be:

```
inline std::istream& operator>>(std::istream& in, Box& box)
{
    return box.get(in);
}
```

To provide stream support for a class that is derived from `Box`, such as the `Carton` class in `Ex14_08`, you just need to define overrides in the `Carton` class for the virtual `get()` and `put()` functions in the base class. The derived class function members can call the base class versions where necessary.

You'll remember the `Carton` class that is derived from `Box` implemented a different `volume()` function and added a string data member that contained the name of the material from which the `Carton` was made. The `get()` and `put()` function member for this class could be defined like this:

```
class Carton : public Box
{
private:
    string material;

public:
    // Constructor explicitly calling the base constructor
    Carton(double lv, double ww, double hv, string str = "material") : Box {lv, ww, hv}
    {
        material = str;
    }

    Carton() = default;

    // Function to calculate the volume of a Carton object
    double volume() const override
    {
        double vol {(length - 0.5)*(width - 0.5)*(height - 0.5)};
        return vol > 0.0 ? vol : 0.0;
    }

    // Stream output
    std::ostream& put(std::ostream& out) const override
    {
        out << std::left << std::setw(15) << material << ' '; // Write the material
        return Box::put(out); // Write the sub-object
    }

    // Stream input
    std::istream& Carton::get(std::istream& in) override
    {
        in >> material; // Read the string
        return Box::get(in); // Read the sub-object
    }
};
```

The Carton class overrides the `get()` and `put()` members of the Box class; the `override` keyword will cause the compiler to verify that a corresponding virtual base class function exists. The overrides call the base class member in the return statement to deal with the inherited data members. When you call `operator<<()`, the function will select the appropriate virtual `put()` function for the object because the object is passed to the function as a reference. This will work for any class that has Box as a direct or indirect base as long as the `get()` and `put()` overrides of the base class functions exist in a derived class. Let's see if it works:

```
// Ex17_09.cpp
// Writing & reading base and derived class objects
#include <fstream>           // For file streams
#include <iostream>          // For standard streams
#include <string>             // For string type
#include <vector>             // For vector container
#include "Box.h"
#include "Carton.h"
using std::string;

int main()
try
{
    std::vector<Box> boxes {Box {1.0, 2.0, 3.0}, Box {2.0, 2.0, 3.0},
        Box {3.0, 2.0, 2.0}, Box {4.0, 2.0, 3.0}};
    std::vector<Carton> cartons {Carton {6.0, 7.0, 8.0, "plastic"}, Carton {5.0, 7.0, 9.0, "wood"},
        Carton {5.0, 6.0, 5.0}};

    const string filename {"D:\\Example_Data\\containers.txt"};
    std::ofstream out {filename};
    if (!out)
        throw std::ios::failure {string {"Failed to open output file "} + filename};

    for (auto& box : boxes)
        out << box;           // Write a Box object
    out.close();               // Close the output stream

    std::cout << boxes.size() << " Box objects written to " << filename << std::endl;

    std::ifstream in {filename}; // Create a file input stream
    if (!in)                     // Make sure it's valid
        throw std::ios::failure {string {"Failed to open input file "} + filename};

    std::cout << "Reading Box objects from the file:\n";
    Box box;
    while (true)
    {
        in >> box;             // Read an object from the file
        if (!in) break;        // End if EOF
        std::cout << box;      // Output the object read
    }
    in.close();                // Close the input stream
}
```

```

out.open(filename);                                // Open the output file
for (auto& carton : cartons)
    out << carton;                                // Write a Carton object
out.close();                                       // Close the output stream
std::cout << cartons.size() << " Carton objects written to " << filename << std::endl;

in.open(filename);
std::cout << "Reading Carton objects from the file:\n";
Carton carton;
while (true)
{
    in >> carton;                                // Read a Carton object from the file
    if (!in) break;                              // End if EOF
    std::cout << carton;                          // Output the object read
}
in.close();                                       // Close the input stream
}
catch (std::exception& ex)
{
    std::cout << typeid(ex).name() << ": " << ex.what() << std::endl;
}

```

This populates a vector with Box objects and writes them to a file in text mode using the overloaded operator<>>() function. The objects are then read back and written to cout using the same overloaded operator function that was used to write to the file. The process is repeated with Carton objects to demonstrate that the output operations are selected at runtime. The output from the example shows that it all works:

```

4 Box objects written to D:\Example_Data\containers.txt
Reading Box objects from the file:
    1      2      3
    2      2      3
    3      2      2
    4      2      3
3 Carton objects written to D:\Example_Data\containers.txt
Reading Carton objects from the file:
plastic      6      7      8
wood         5      7      9
cardboard    5      6      5

```

You may have noticed that we didn't *really* write objects to a file in any of the previous examples. We just wrote the data that the program needed to recreate an object of a given type - the dimensions in the case of a Box object. Nothing in the file identified the contents as representing an object. The typeid operator results in a type_info object that you could use to obtain the type name so you could write this to the stream too. In theory this would allow the input process to verify that the data in the stream related to the type of object you are expecting. However, there's no guarantee that the name() member of the type_info class will always return the same string for a given class type from one execution to the next so if you do need to identify the class type in the file, it would be better to devise your own scheme for this.

The difficulties increase with classes that are more complex than Box or Carton. Suppose that you want to serialize objects of classes that contain pointers to objects of other classes. A fundamental prerequisite is that the class type of the object being pointed to defines functions that enable serialization. These are essential to write and retrieve the objects pointed to. Writing the address that a pointer contains to a stream is futile because the address won't be valid when it's read back. The object pointed to will certainly be located at a different address when it's reconstructed. However, you can still deal with it. Figure 17-5 shows one possible approach for writing an object that contains a linked list to a stream.

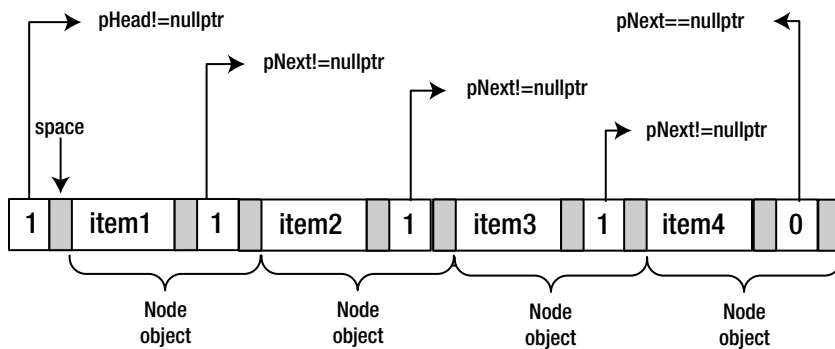


Figure 17-5. Serializing an object containing a linked list

The only thing that is important about a pointer when serializing it is whether or not it's `nullptr`. If it's `nullptr`, it doesn't point to anything so there's no object to write to the stream. If it's not `nullptr`, then there is an object that must be written to the stream. You can represent the state of a pointer in the stream as a bool value; if it's `nullptr` you can write it as `false`; otherwise you write it as `true`, and record the object to which it points in the stream. Figure 17-7 illustrates the contents of the stream after writing an object that encapsulates a linked list of Node objects each of which contains an Item object. The pointer to the first object in the list is written as `true` if the list is not empty; if the list isn't empty the Item object that is contained in the first Node object is written to the stream. If the `pNext` pointer member of a Node is not `nullptr`, `true` is written to the stream followed by the Item object from the next Node object. The last Node will have `nullptr` as its `pNext` member, which will be written to the stream as `false`; this ends the output. You don't need to store Node object in the stream; they are just wrappers for the items that are the real data. I'm leaving it as a final exercise for you to implement this in an example.

■ **Note** An alternative approach to serialization is to use a mark-up language to express the data and structure for an object. XML — the eXtended Markup Language — is often the base for languages that are used to specify objects in a file. This approach has the merit that it is independent of the programming language used to read and write the data, but retains the object definitions. A discussion of this is outside the scope of this book.

Summary

In this chapter, I've covered the basics of stream operations and how you can apply them to writing and reading files. The important elements that you've explored in this chapter include:

- The standard library supports input and output operations on character streams, binary (byte) streams, and string streams.
- The standard streams for input and output are `std::cin` and `std::cout`. The standard error streams are `std::cerr` and `std::clog`.
- The Standard Library defines file stream classes that encapsulate a file for input, for output, or for both. These classes are `std::ifstream`, `std::ofstream` and `std::fstream`.
- The open mode that you specify when creating or opening a stream determines whether you can read from a stream and/or write to it. It also determines whether the stream contents are to be overwritten on output, and whether the stream is binary or text.

- A file only contains bytes, regardless of the open mode. There is nothing to prevent you from reading a binary stream as text or vice versa.
- Opening a file output stream with a file name for which no file exists causes a file to be created.
- A file has a beginning, an end, and a current position.
- You can alter the current position in a file stream to a position that was recorded previously. This can be a position that is a positive offset from the beginning of the stream, a position that is a negative offset from the end of a stream, or a position that is a positive or negative offset from the current position.
- The extraction and insertion operators provide formatted stream input and output operations for data of fundamental types.
- To support stream operations for your objects, you can overload the insertion and extraction operators with operator functions that are friends of your class.
- Stream classes provide `read()` and `write()` function members for reading and writing a stream in binary mode. Binary mode operations always read and write a sequence of bytes.
- The string stream classes provide stream I/O operations to or from `std::string` objects in memory.

EXERCISES

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Apress website (<http://www.apress.com/source-code>), but that really should be a last resort.

- Exercise 17-1. Write a `Time` class that stores hours, minutes, and seconds as integers. Provide an overloaded insertion operator (`<<`) that will print the time in the format `hh:mm:ss` to any output stream.
- Exercise 17-2. Provide a simple extraction operator (`operator>>()`) for the `Time` class that will read time values in the form `hh:mm:ss`. How are you going to cope with the `:` characters?
- Exercise 17-3. Write a program to log time values to a file. Write a matching program to read a file of time values and output them to the screen.
- Exercise 17-4. Write a program that reads lines of text from standard input and writes them to standard output, removing all leading whitespace and converting multiple spaces to single spaces. Test it on input from the keyboard and on characters read from a file. Write a second program that converts lowercase characters to uppercase, and test that too.
- Exercise 17-5. Define a class that encapsulates a linked list of `Box` objects. Implement `operator>>()` and `operator<<()` for the class so the object can be stored in a file and read back. Define a `main()` function to show that your class works.

This is not quite the end of the book. In my previous C++ book there was a sizable project for you to attempt. It is too long for inclusion here but it is still available in the download for the exercise solutions for this book. There is a project description for you to read before you implement the project. There's also my solution that you can look at if you get stuck. Have fun!

Index

■ A

Abstract class
 Circle class creation, 454
 constructor, 453
 data member, 457
 definition, 453
 interface, 455
 three-level class hierarchy, 458
 Vessel class, 456–457
 volume() function, 454–455
Access specifiers, 407–408
Address-of operator, 153
Aggregation, 402
AND operator, 61, 92
Arithmetic operations, 29
Array<T,N> template, 140
Arrays
 of characters, 131
 definition, 105
 dynamic allocation, 169
 initial values, 111
 of pointers, 158
 runtime, 119
 size, 111
 usage, 106
assert() macro, 311
average() function, 223

■ B

Beans array argument, 225
Binary literals, 28
Binary mode
 get() and put() function, 561–562
 main() function, 563
 numerical data, 563
 array containers, 566
 function template, 565
 #include directives, 567

 is_arithmetic<T> template, 565
 main() function, 568
 primes_count elements, 569
 read() function, 564
 static_assert(), 565–566
 vector<T> container, 567, 569–570
 write() function, 564–565
 validate_files() function, 563
Binary notation, 11
Bitwise operators
 AND, 61
 exclusive OR
 flags, 67
 setfill() manipulator, 67
 setw() manipulator, 67
 flags, 57
 OR, 62
 record information, 57–58
 shift operators, 59
 definition, 58
 logical operations, 60
 signed and
 unsigned integers, 60
Bubble sort, 130
Bubble up, 130

■ C

Call stack, 213
Capture clause, 279
Casting pointers, 447–449
change() function, 276
change_it() function, 220
Character set, 17
 escape sequences, 18
 control characters, 19
 double quotes, 20
 problem characters, 19
 string literal, 20
trigraph sequences, 18

- C++ language
 - characters
 - ASCII codes, 16
 - escape sequence, 19
 - trigraph sequence, 18
 - UCS codes, 17
 - Unicode, 17
 - classes, 7
 - code presentation styles, 8
 - compile and link process, 8–9
 - definition, 1
 - header files, 7
 - numbers
 - big-endian and little-endian systems, 14
 - binary numbers, 9
 - floating-point numbers, 15
 - hexadecimal numbers, 11
 - negative binary numbers, 12
 - octal integers, 14
 - object-oriented approach, 20
 - objects, 7
 - procedural programming, 20
 - programming concepts
 - comments, 2
 - functions, 3
 - header files, 3
 - input/output streams, 5
 - keywords, 6
 - names, 6
 - namespaces, 5
 - preprocessing directives, 3
 - return statement, 5
 - statements, 4
 - structure, 2
 - whitespace, 3
 - source files, 7
 - Standard Library, 1, 7
 - templates, 7
- Class, 315
 - access specifiers, 320
 - addBox() function, 360
 - arrays, 340
 - const member function, 339
 - constructors
 - Box class, 323
 - copy constructor, 330
 - default constructor, 322, 326
 - definition, 324
 - delegating constructor, 329
 - explicit keyword, 327
 - initialization list, 326
 - data hiding, 317
 - data members, 321
 - definition, 320
 - destructors, 347
 - encapsulation, 316
 - friend functions, 333
 - boxSurface() function, 335
 - main() function, 335
 - unrestricted access, 336
 - function member, 321
 - inheritance, 317
 - manipulators, 539–540
 - ios header, 541
 - with arguments, 541–542
 - nested class, 359
 - listBoxes() member, 361
 - Truckload class, 360
 - with public access, 362
 - non-const function, 339
 - object size, 341
 - pointers, 350
 - Box object, 351
 - listBox() member, 351
 - Package class, 353
 - package objects, 352
 - setNext() function, 353
 - Truckload object, 352
 - (see also Truckload class)
 - polymorphism, 318
 - private data members, accessing, 331
 - references, 350
 - standard stream objects, 536–537
 - static data members, 342
 - accessing, 345
 - Box class constructor, 346
 - constructor, 344
 - getObjectCount() function, 344–345
 - objectCount, 343
 - static function members, 342
 - accessing, 347
 - getObjectCount() function, 346
 - streambuf type, 536
 - stream extraction operations, 538
 - stream insertion operations, 538–539
 - struct keyword, 321
 - terminology, 319
 - this pointer
 - return type, 337
 - volume() function, 336
 - typedefs, 537
 - volume() function, 338
- Class templates, 495
 - array template, 498
 - assignment operator, 499, 502
 - constructor, 498, 500
 - container classes, 496
 - default values, 518
 - definitions, 496
 - destructor, 501

- friends function, 522
- getSize() member, 499
- instantiation, 496, 503
 - Box class, 506
 - explicit instantiation, 504, 518
 - implicit instantiation, 504–505
 - main() function, 507
 - member functions, 504
 - out-of-range index values, 506
 - subscript operator, 508
 - try block, 508
 - what() function, 507
- member function, 500
- nested class (*see* Stack)
- non-type parameters, 509
 - arguments, 516
 - array template, 511, 514
 - assignment operator, 513
 - conditional operator, 516
 - copy constructor, 512
 - destructor, 512
 - exception, 516
 - function parameter, 510
 - member function, 511
 - pointers, 517
 - subscript operator, 513
- parameters
 - non-type parameter, 497
 - type parameter, 497
- specialization
 - complete specialization, 520
 - partial specialization, 521
- static_assert(), 519
- static members, 508
- subscript operator, 499, 502
- Comma operator, 116
- Constant max, 166
- Constant pointers, 160
- Const function parameter, 223
- Constructor
 - Box class, 415
 - copy constructor
 - Carton class, 412
 - definition, 412–413
 - creation, 409–411
 - default constructor, 414
 - definition, 409, 412
- Container
 - array<>, 142
 - deleting elements from vector, 147
 - vector<T>, 144
- Copy constructor
 - Carton class, 412
 - definition, 412–413
- Crosscast, 448
- C-style string, 131

■ D

- Data hiding, 317
- Data types. *See* Class
- Debugging
 - assert() macro, 311
 - description, 306
 - preprocessing directives, 307
 - tools for, 306
- Decimal integer literals, 26
- Decimal notation, 10
- Decision making
 - arbitrary calculation, 101
 - boolean literals, 80
 - character classification, 86
 - comparison operators, 80
 - conditional operator
 - definition, 95
 - if statement, 95
 - increment operator, 95
 - program, 96
 - Converting Character, 87
 - floating-point values, 82
 - if-else statement
 - char variable, 88
 - definition, 88
 - isalnum() function, 88
 - logic, 88
 - program, 89
 - if statement
 - declaration, 83
 - logic, 82
 - program, 83–84
 - semicolon (;), 83
 - locale header, 87
 - logical operators
 - AND, 92
 - negation, 93–94
 - OR, 93
 - usage, 92
 - Nested if-else statements, 89
 - Nested if statements, 85
 - relational operators, 79
 - standard library functions, 87
 - switch statement, 101–102
 - break statement, 97
 - case label, 97
 - cases, 97
 - case values, 99
 - creation, 97
 - default label, 97
 - integer constant expression, 97
 - isalpha() function, 99
 - program, 98
 - tolower() function, 99
 - unconditional branching, 100

- Default argument values, 231
- Default capture clause, 279
- Default constructor, 414
- Delete operator, 169
- Destructors, 416
- Dimensions Setup, 137
- doThat() function, 439
- doThis() function, 439
- Do-while loop, 121, 166–167
- Downcast, 448
- draw() function, 431, 453
- dynamic_cast<>() operator, 450
- Dynamic memory allocation, 167
 - arrays, 169
 - hazards, 171

■ E

- Encapsulation, 316
- Error handling, 463
- Exceptions, 463
 - class objects
 - base class type, 478
 - catch block, 475, 483
 - code implementation, 474
 - default constructor, 474
 - dynamic type, 479
 - header file, 474
 - parameter type, 479
 - rethrow exceptions, 480
 - working principles, 477
 - definition, 464
 - functions
 - constructor try block, 486
 - destructor, 487
 - exceptions thrown, 486
 - function try block, 485
 - handling process, 467
 - nested try blocks, 471
 - Standard Library, 487
 - Box class constructor, 490
 - catch block, 489
 - dimension_error
 - objects, 491–492
 - exception class, 489
 - logic_error classes, 490
 - runtime_error classes, 490
 - std::range_error function, 491
 - Trouble exception class, 491
 - throw exception, 465, 469
 - functions, 470
 - try blocks, 470
 - throwIt() function, 473
 - try block, 464

- uncaught exception, 468
 - std::abort() function, 468
 - std::exit() function, 469
 - std::terminate() function, 468
- Explicit type conversion
 - compiler, 47
 - explicit cast, 48
 - old-style casts, 49
 - program, 47–48
 - static_cast keyword, 47
- Extension namespace, 296
- External linkage
 - const variable, 289
 - definition, 288
- Extern declaration, 289
- Extern keyword, 75
- extract_words() function, 263

■ F

- File read/write operations
 - fstream object, 572
 - ifstream/ofstream object, 571
 - random access
 - binary stream, 572
 - constructor, 573
 - data members, 573
 - file_exists() member, 574
 - fstream member, 573
 - isprime() member, 576
 - next_prime() member, 575
 - object-oriented approach, 572
 - Primes class, 577
 - subscript operator, 576
- File streams
 - properties, 542
 - reading file, text mode
 - close(), 548
 - coding implementation, 546–547
 - eof() function, 546–547
 - explicit cast, 546
 - fail() function, 545
 - ifstream object, 545
 - void*(), 546
 - types, 542
 - writing file, text mode
 - close(), 545
 - cmath header, 544
 - coding implementation, 543
 - fstream header, 544
 - ofstream object, 543
 - unsigned long, 544
- Fill() function, 141
- find_words(), 228

- For loop, 108, 110
 - floating-point values, 113
 - range-based, 116
- Fragmentation, memory, 172
- Function header, 4
- Functions, 3, 213
 - characteristics of, 214
 - in classes, 214
 - declaration, 218
 - definition, 214
 - for loops, 216
 - inline, 239
 - main(), 230
 - overloading
 - const pointer parameters, 245
 - const reference parameters, 246
 - default parameter values, 246
 - largest() function, 241
 - pointer parameters, 243
 - reference parameters, 243
 - pass-by-reference, 225
 - pass-by-value mechanism, 219
 - array, 222
 - pointer, 221
 - pointer to function, 255
 - prototypes, 218
 - recursive (*see* Recursive function)
 - returning values
 - pointer, 233
 - reference, 238
 - return statement, 217
 - static variables, 239
 - template
 - definition, 247
 - explicit type argument, 250
 - instances, 248
 - overloading, 251
 - specialization, 250
 - typename, 248
- Fundamental data types, 23
 - arithmetic operations, 29
 - assignment operations
 - const variables, 32
 - conversion, 32
 - << operators, 32
 - lvalue, 30
 - program, 31
 - rvalue, 30
 - auto keyword, 52
 - automatic variables, 71
 - binary literals, 28
 - binary operators, 28
 - bitwise operators
 - (*see* Bitwise operators)
 - character variables
 - arithmetic expressions, 50
 - definition, 50
 - escape sequences, 50
 - unicode character, 51
 - cmath header
 - numerical functions, 40
 - program, 42
 - radians calculation, 41–42
 - sqrt() function, 43
 - trigonometric functions, 41
 - decimal integer literals, 26
 - declaration, 34
 - using directive, 34
 - dynamic variable, 71
 - enumerators
 - compile-time constants, 68
 - creation, 67
 - definition, 67
 - explicit value, 68
 - old-style enumerations, 70
 - program, 68–69
 - standard output stream, 68
 - explicit type conversion
 - compiler, 47
 - explicit cast, 48
 - old-style casts, 49
 - program, 47–48
 - static_cast keyword, 47
 - extern keyword, 75
 - floating-point calculations
 - NaN and $\pm$ infinity operands, 40
 - operands, 38
 - pitfalls, 38
 - floating-point literal, 38
 - floating-point variables, 37
 - global variables
 - advantages, 72
 - and automatic variables, 73–74
 - definition, 71
 - Example.cpp, 72
 - increment operator, 74
 - main(), 72
 - scope resolution operator, 74
 - hexadecimal literals, 27
 - implicit conversions, 46
 - increment and decrement operators
 - definition, 35
 - postfix decrement, 35
 - postfix increment, 35
 - limits standard library header, 49–50
 - Lvalues and Rvalues, 52–53
 - octal literals, 27
 - op= assignment operators, 32–33

Fundamental data types (*cont.*)

- operator precedence and associativity
 - C++ operators, 56
 - definition, 55
 - expression, 56
- sizeof operator, 34
- static keyword, 71
- static variables, 75
- stream manipulators
 - integer values, 45
 - io manip header, 44
 - setprecision() parameter, 44
- typedef keyword, 70
- unary operators, 28
- variables (*see* Variables)

■ G

- getWeight() function, 422, 424
- Global namespace, 295
- Global variables
 - advantages, 72
 - and automatic variables, 73–74
 - definition, 71
 - Example.cpp, 72
 - increment operator, 74
 - main(), 72
 - scope resolution operator, 74

■ H

- Header file
 - description, 292
 - preventing duplication in, 293
- Heap/free store, 167
- Hexadecimal literals, 27

■ I, J, K

- Implicit conversions, 46
- Indefinite loops, 127
- index_min, 235
- Indirection operator, 154
- Indirect member selection operator, 171
- Inheritance, 317, 399, 426
 - vs.* aggregation, 401
 - class
 - access level of, 406
 - access specifiers, 407–408
 - Box class, 402–403
 - Carton class, 403–404
 - constructor (*see* Constructor)
 - data member, 418
 - definition, 399, 401, 415
 - destructors, 416

- hierarchies, 400
- main() function, 404
- member function, 419
- protected keyword, 405
- multiple inheritance (*see* Multiple inheritance)
- Inline functions, 239
- Interface, 455
- Internal linkage, 288
- isalnum() function, 88
- is_arithmetic<T> template, 312
- isprime function, 167

■ L

- Lambda expressions, 271
 - capture clause, 279
 - definition, 272
 - function parameter, 275
 - function templates, 274
 - naming of, 272
 - recursion in, 283
 - std::function template type, 276
 - in template, 281
- Linkage
 - definition, 288
 - for name, 288
- listVector<T>(), 282
- list_words() function, 228
- Loop
 - definition, 107
 - do-while loop, 121
 - for loop, 108
 - indefinite, 127
 - nested loops, 123
 - skipping loop iterations, 125
 - while loop, 117
- Loop control expression, 114–115

■ M

- main(), 230
- Max_word_length() function, 265
- Member selection, pointer, 171
- Memory leaks, 172
- Multidimensional arrays, 134
 - character, 138
 - dimensions setup, 137
 - initializing, 136
- Multiple inheritance
 - ambiguity problems, 421, 424
 - Carton class, 421
 - CerealPack class, 420, 423
 - class hierarchy, 420, 424–425
 - data members, 422
 - definition, 419

- getWeight() function, 422, 424
- header file, 422
- interface, 420
- main().function, 424
- virtual base class, 425–426

Myprog, 230
 myRegion, 295
 mysort() function, 279

■ N

Namespace, 6, 294

- aliases, 301
- declaration, 298
- definition, 295
- and functions, 298
- nested, 302
- unnamed, 301

Nested if-else statements, 89
 Nested if statements, 85
 Nested loops, 123
 Nested namespaces, 302
 New operator, 168
 nextFibonacci() function, 241
 non-const reference, 238
 Non-type template parameters, 253
 normalize() function, 302
 Null character, 131
 nullptr, 229
 Numerical functions, 40

■ O

Object-oriented approach, 20–21
 Object-oriented programming (OOP), 315
 Octal literals, 27
 One-dimensional array, 134
 Operator overloading

- arithmetic operators, 380
 - Box objects, 381
 - listBox() member, 382
 - main() function, 384
 - max() and min() functions, 382
 - memory management, 381
 - operator function, 386
 - output statements, 384
- assignment operator, 369
- binary operator, 367
- Box class, 365
- class members
 - assignment operator, 375, 377
 - copy constructor, 374, 377
 - destructor, 375
 - move constructor, 375, 379

Class_Type operators, 373

- definition, 366
- function members, 368
- function object, 396
- global operator, 369
- inline function, 370
- main() function, 369
- member operator function, 370
- operators, 366
- reference function parameter, 367
- relational operators, 372
- Return_Type operators, 373
- show() function, 371
- subscript operator
 - do-while loop, 387
 - friend function, 388
 - getBox() member, 392
 - listBoxes() member, 388, 390
 - operator<<() function, 388
 - temporary copy, 392
 - Truckload class, 387
- type conversions, 394
 - conversion operators, 395
 - increment and decrement operators, 395
- unary operators, 374

OR operator, 62, 93

■ P

Pass-by-reference mechanism, 225
 Pass-by-value mechanism, 219

- array, 222
- pointer, 221

peek() function, 556
 Pointer

- definition, 458
- pvalue, 168
- Vessel class destructor, 459
- virtual destructors, 460

Pointer arithmetic operation, 162
 Pointers, 151

- address-of operator, 153
- arithmetic operation, 162
- arrays of, 158
- constant, 160
- definition, 151
- difference in, 164
- indirection operator, 154
- member selection, 171
- notation, array name, 164
- shared_ptr<T> object, 175, 178
- to char, 156
- unique_ptr<T>, 174
- uses, 156
- weak_ptr<T>, 179

- Pointer to function, 255
- Polymorphism, 318, 429
 - Base Class Pointer, 430
 - Box.h and Box.cpp, 431
 - Carton class definition, 430
 - casting pointers, 448–449
 - cost of, 451–452
 - default argument values, 443
 - dynamic binding/late binding, 434
 - dynamic cast, 448
 - pointer
 - definition, 458
 - Vessel class destructor, 459
 - virtual destructors, 460
 - pointer to derived class conversion
 - Casting pointers, 447–448
 - CerealPack class, 447
 - compiler, 447
 - pure virtual functions
 - (see Pure virtual functions)
 - reference parameter, 444–445, 450
 - showVolume() function, 432
 - smart pointers, 444
 - static binding, 433
 - ToughPack class, 431
 - typeid operator, 450
 - usage, 431
 - virtual functions
 - access specifiers, 441
 - class hierarchies, 438
 - declaration, 434
 - final, 440
 - overriding, 440
 - requirements, 438
 - showVolume() function, 438
 - ToughPack class, 437
 - volume() function, 436
 - volume() function, 432–433
- Postfix decrement operator, 35
- Postfix increment operator, 35
- power() function, 215–216, 218
- Preprocessing directives, 287, 290
- Preprocessing identifiers, 291
- Pure virtual functions
 - abstract class
 - Circle class creation, 454
 - constructor, 453
 - data member, 457
 - definition, 453
 - interface, 455
 - three-level class hierarchy, 458
 - Vessel class, 456–457
 - volume() function, 454–455

- purpose of, 452
 - Shape class creation, 453
- putback() function, 557
- put() function, 558
- pvalue, 168
- pWords, 228
- PWords alias, 229

■ Q

- Quicksort algorithm, 261

■ R

- Range-based for loop, 116
- Raw pointers, 173
- Raw string literal, 209
- readsome() function, 557
- Recursive function, 258
 - extract_words() function, 263
 - main() function, 262
 - max_word_length() function, 265
 - Quicksort algorithm, 261
 - show_words() function, 266
 - sort() function, 264
 - sorting operation, 261
 - swap() function, 264
- Reference cycles, 173
- References, 180
 - lvalue, 181
 - rvalue, 182
 - variable, range-based for loop, 181
- Relational operators, 79
- resetiosflags() functions, 542
- reverse() function, 274

■ S

- seekg() function, 555
- seekp() function, 555
- setiosflags() function, 542
- setValues<T>(), 282
- shift_range() function, 235
- show_data() function, 233
- show_error() function, 246
- showHCF function, 284
- showVolume() function, 432
- Show_words() function, 266
- Size() function, 141
- Sizeof operator, 34
- Smart pointer, 173
- sort() function, 264, 276
- sqrt() function, 43

- Stack, 167
 - assignment operator, 527
 - code implementation, 529
 - concept, 524
 - copy() function, 526–527
 - default constructor, 527
 - destructor template, 528
 - freeMemory() function, 526–527
 - getline() function, 530
 - isEmpty() function, 530
 - length() function, 530
 - linked list, 525
 - node object, 525
 - pop() operation, 528
 - push() operation, 524, 528
- Standard error stream, 535
- Standard Template Library (STL), 7
- Statement, C++, 4
- Static assertion, 312
- Static variables, 75, 239
- Stream I/O
 - advantages of, 535
 - binary mode, 534 (*see also* Binary mode)
 - class (*see* Class)
 - data transferring, 534
 - definition, 534
 - errors in
 - clear(), 559
 - exceptions, 560–561
 - fail() function, 559–560
 - state flags, 559–560
 - file read/write operations (*see* File read/write operations)
 - file streams (*see* File streams)
 - objects
 - in binary mode, 583
 - Carton class overrides, 588–589
 - extraction operator, 580
 - insertion operator, 579–580
 - linked list, 590
 - operator>>() and
 - operator<<() functions, 586
 - public virtual members, 586
 - serialization, 579
 - volume() function, 587
 - open mode
 - clear() function, 552
 - close(), 549
 - definition, 548
 - fail() function, 552
 - file overwrite, 548
 - nextprime() function, 551–554
 - open (), 549
 - outFile, 548
 - seekg() functions, 555
 - seekp() function, 555
 - standard output stream, 549–551
 - tellg() function, 553
 - values, 548
 - text mode, 534
 - unformatted input functions
 - EOF, 556
 - gcount(), 557
 - getline(), 557
 - null-terminated string, 557
 - peek() function, 556
 - putback(), 557
 - readsome() function, 557
 - single character, 556
 - unset() function, 556
 - unformatted output functions, 558
- Strings
 - character access, 190
 - compare() function
 - substr() function, 197
 - with substring, 196
 - word, 195
 - comparison operators, 193
 - cstring header, 185
 - erase() function, 207
 - find() function
 - getline() function, 200
 - set of characters, 200
 - string::npos, 198
 - while loop, 200
 - insert() function, 204
 - international characters
 - char16_t, 208
 - char32_t, 208
 - wchar_t characters, 208–209
 - raw string literals, 209
 - replace() function, 205
 - rfind() function, 203
 - string object
 - concatenation, 188
 - define and initialize, 188
 - length() function, 186
 - proverb, 186–187
 - string type, 185
 - substr() function, 192
 - Unicode characters, 209
- String stream classes, 578–579
- surface() function, 450
- swap() function, 264
- Switch statement, 101–102
 - break statement, 97
 - case label, 97
 - cases, 97
 - case values, 99
 - creation, 97

Switch statement (*cont.*)

- default label, 97
- integer constant expression, 97
- isalpha() function, 99
- program, 98
- tolower() function, 99

■ T

- tellg() function, 553
- Ternary operator, 95
- TESTFUNCTION identifier, 308
- Three-dimensional array, 134
- total() function, 418
- Trailing return type, 255
- Translation unit, 287
- Tree structure, 258
- TReturn, 252
- Trigonometric functions, 41
- Truckload class
 - constructor, 355
 - definition, 354
 - deleteBox() function, 355
 - getFirstBox() function, 356
 - getNextBox() function, 354, 356
 - listBox(), 357
 - main() function, 359
 - Package object, 355
- Two-dimensional array, 134
- type_traits header, 313

■ U

- unset() function, 556
- Unicode
 - character, 51
 - UTF-8, 17
 - UTF-16, 17
- Universal Modelling Language (UML), 400
- Unnamed namespaces, 301

■ V

Variables

- const keyword, 26
- definition, 23
- integer
 - definition, 24
 - functional notation, 24
 - initializer list, 24
 - narrowing conversion, 24
 - program, 25
 - signed integer types, 25
 - unsigned integer types, 26

Vector

- capacity of, 145–147
- container, 147
- size of, 145–147

Vessel class, 456

Virtual base class, 425–426

Virtual functions

- access specifiers, 441
- class hierarchies, 438
- declaration, 434
- final, 440
- overriding, 440
- requirements, 438
- showVolume() function, 438
- ToughPack class, 437
- volume() function, 436

void keyword, 215

volume() function, 408–409, 432–433, 441–442, 454

■ W, X

- While loop, 117
- write() function, 558

■ Y, Z

- yield() function, 224

Beginning C++



Ivor Horton

Apress®

Beginning C++

Copyright © 2014 by Ivor Horton

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4842-0008-7

ISBN-13 (electronic): 978-1-4842-0007-0

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director: Welmoed Spahr

Lead Editor: Steve Anglin

Development Editor: Matthew Moodie

Technical Reviewer: Michael Thomas

Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Louise Corrigan, Jim DeWolf,

Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham,

Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft,

Gwenan Spearing, Matt Wade, Steve Weiss

Coordinating Editor: Melissa Maldonado

Copy Editor: Lori Cavanaugh

Compositor: SPi Global

Indexer: SPi Global

Artist: SPi Global

Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

This is for Alexander and Henry who are both going to learn programming soon. If their amazing expertise with Minecraft is anything to go by, they will be brilliant at it.

Contents

About the Author	xxiii
About the Technical Reviewer	xxv
Introduction	xxvii
■ Chapter 1: Basic Ideas	1
Modern C++	1
C++ Program Concepts.....	2
Comments and Whitespace	2
Preprocessing Directives and Header Files	3
Functions	3
Statements	4
Data Input and Output.....	5
return Statements.....	5
Namespaces	5
Names and Keywords.....	6
Classes and Objects	7
Templates.....	7
Program Files	7
Standard Libraries.....	7
Code Presentation Style	7
Creating an Executable	8
Representing Numbers.....	9
Binary Numbers.....	9
Hexadecimal Numbers	11

Negative Binary Numbers.....	12
Octal Values	14
Big-Endian and Little-Endian Systems	14
Floating-Point Numbers.....	15
Representing Characters.....	16
ASCII Codes	16
UCS and Unicode	17
C++ Source Characters.....	17
Trigraph Sequences.....	18
Escape Sequences	18
Procedural and Object-Oriented Programming	20
Summary.....	21
■ Chapter 2: Introducing Fundamental Types of Data	23
Variables, Data, and Data Types	23
Defining Integer Variables.....	24
Defining Variables with Fixed Values	26
Integer Literals	26
Decimal Integer Literals.....	26
Calculations with Integers.....	28
More on Assignment Operations	30
The op= Assignment Operators.....	32
using Declarations and Directives.....	34
The sizeof Operator	34
Incrementing and Decrementing Integers.....	35
Postfix Increment and Decrement Operations.....	35
Defining Floating-Point Variables	36
Floating-Point Literals	37
Floating-Point Calculations	38
Pitfalls.....	38
Mathematical Functions	40

Formatting Stream Output.....	43
Mixed Expressions and Type Conversion.....	45
Explicit Type Conversion.....	47
Old-Style Casts	49
Finding the Limits.....	49
Working with Character Variables	50
Working with Unicode Characters	51
The auto Keyword	52
Lvalues and Rvalues.....	52
Summary.....	53
■ Chapter 3: Working with Fundamental Data Types	55
Operator Precedence and Associativity.....	55
Bitwise Operators.....	57
The Bitwise Shift Operators.....	58
Using the Bitwise AND	61
Using the Bitwise OR	62
Using the Bitwise Exclusive OR	63
Enumerated Data Types.....	67
Old-Style Enumerations.....	70
Synonyms for Data Types	70
The Lifetime of a Variable.....	71
Positioning Variable Definitions.....	71
Global Variables	71
Static Variables	75
External Variables.....	75
Summary.....	76
■ Chapter 4: Making Decisions	79
Comparing Data Values	79
Applying the Comparison Operators.....	80
Comparing Floating Point Values.....	82

The if Statement.....	82
Nested if Statements.....	85
Code-Neutral Character Handling.....	86
The if-else Statement.....	88
Nested if-else Statements.....	89
Understanding Nested ifs	90
Logical Operators	92
Logical AND	92
Logical OR.....	93
Logical Negation.....	93
The Conditional Operator.....	95
The switch Statement	97
Unconditional Branching	100
Statement Blocks and Variable Scope.....	101
Summary.....	102
■ Chapter 5: Arrays and Loops.....	105
Arrays	105
Using an Array	106
Understanding Loops	107
The for Loop	108
Avoiding Magic Numbers	110
Defining the Array Size with the Initializer List	111
Determining the Size of an Array	111
Controlling a for Loop with Floating-Point Values	113
More Complex for Loop Control Expressions.....	114
The Comma Operator	116
The Ranged-based for Loop	116
The while Loop	117
Allocating an Array at Runtime.....	119

The do-while Loop.....	121
Nested Loops.....	123
Skipping Loop Iterations.....	125
Breaking Out of a Loop.....	127
Indefinite Loops	127
Arrays of Characters.....	131
Multidimensional Arrays.....	134
Initializing Multidimensional Arrays.....	136
Multidimensional Character Arrays	138
Alternatives to Using an Array	140
Using array<T,N> Containers	140
Using std::vector<T> Containers.....	144
The Capacity and Size of a Vector	145
Deleting Elements from a Vector container	147
Summary	148
■ Chapter 6: Pointers and References.....	151
What Is a Pointer?	151
The Address-Of Operator.....	153
The Indirection Operator.....	154
Why Use Pointers?.....	156
Pointers to Type char.....	156
Arrays of Pointers	158
Constant Pointers and Pointers to Constants.....	160
Pointers and Arrays	162
Pointer Arithmetic.....	162
Using Pointer Notation with an Array Name	164
Dynamic Memory Allocation.....	167
The Stack and the Heap	167
Using the new and delete Operators	168

Dynamic Allocation of Arrays	169
Member Selection through a Pointer	171
Hazards of Dynamic Memory Allocation.....	171
Memory Leaks	172
Fragmentation of the Free Store.....	172
Raw Pointers and Smart Pointers	173
Using <code>unique_ptr<T></code> Pointers	174
Using <code>shared_ptr<T></code> Pointers	175
Comparing <code>shared_ptr<T></code> Objects.....	178
<code>weak_ptr<T></code> Pointers	179
Understanding References	180
Defining lvalue References.....	181
Using a Reference Variable in a Range-Based for Loop	181
Defining rvalue References	182
Summary	183
■ Chapter 7: Working with Strings.....	185
A Better Class of String	185
Defining string Objects	186
Operations with String Objects.....	188
Accessing Characters in a String.....	190
Accessing Substrings	192
Comparing Strings.....	193
Searching Strings	198
Searching a String Backwards	203
Modifying a String	204
Strings of International Characters	208
Strings of <code>wchar_t</code> Characters.....	208
Objects that contain Unicode Strings	209
Raw String Literals	209
Summary	210

■ Chapter 8: Defining Functions.....	213
Segmenting Your Programs.....	213
Functions in Classes.....	214
Characteristics of a Function.....	214
Defining Functions	214
The Function Body	216
Function Declarations.....	217
Passing Arguments to a Function.....	219
Pass-by-Value.....	219
Pass-by-Reference	225
Arguments to main()	230
Default Argument Values	231
Multiple Default Parameter Values	231
Returning Values from a Function	233
Returning a Pointer.....	233
Returning a Reference.....	238
Inline Functions	239
Static Variables.....	239
Function Overloading	241
Overloading and Pointer Parameters.....	243
Overloading and Reference Parameters.....	243
Overloading and const Parameters	245
Overloading and Default Argument Values	246
A Sausage Machine for Functions.....	247
Creating Instances of a Function Template.....	248
Explicit Template Argument.....	249
Function Template Specialization	250
Function Templates and Overloading.....	251
Function Templates with Multiple Parameters	252
Non-Type Template Parameters.....	253

Trailing Return Types	254
Pointers to Functions	255
Defining Pointers to Functions	256
Recursion	258
Applying Recursion.....	261
The Quicksort Algorithm	261
The main() Function	262
The extract_words() Function	263
The swap() Function.....	264
The sort() function.....	264
The max_word_length() Function	265
The show_words() Function.....	266
Summary.....	267
■ Chapter 9: Lambda Expressions.....	271
Introducing Lambda Expressions	271
Defining a Lambda Expression.....	272
Naming a Lambda Expression.....	272
Passing a Lambda Expression to a Function.....	274
Function Templates that Accept Lambda Expression Arguments	274
A Function Parameter Type for Lambda Arguments	275
Using the std::function Template Type	276
The Capture Clause	279
Capturing Specific Variables.....	280
Using Lambda Expressions in a Template.....	281
Recursion in Lambda Expressions	283
Summary.....	284
■ Chapter 10: Program Files and Preprocessing Directives.....	287
Understanding Translation Units.....	287
The “One Definition” Rule.....	288
Program Files and Linkage.....	288

Determining Linkage for a Name.....	288
External Names	289
const Variables with External Linkage.....	289
Preprocessing Your Source Code	289
Defining Preprocessing Identifiers	291
Undefining an Identifier	292
Including Header Files.....	292
Preventing Duplication of Header File Contents	293
Namespaces.....	294
The Global Namespace	295
Defining a Namespace.....	295
Applying using Declarations	298
Functions and Namespaces	298
Unnamed Namespaces.....	301
Namespace Aliases	301
Nested Namespaces.....	302
Logical Preprocessing Directives	303
The Logical #if Directive	303
Testing for Specific Identifier Values	303
Multiple Choice Code Selection	304
Standard Preprocessing Macros	305
Debugging Methods	306
Integrated Debuggers.....	306
Preprocessing Directives in Debugging.....	307
Using the assert() Macro	311
Switching Off assert() Macros.....	311
Static Assertions	312
Summary	313

■ Chapter 11: Defining Your Own Data Types	315
Classes and Object-Oriented Programming	315
Encapsulation	316
Inheritance.....	317
Polymorphism.....	318
Terminology	319
Defining a Class	320
Constructors	322
Defining Constructors Outside the Class	324
Default Constructor Parameter Values.....	326
Using a Constructor Initialization List.....	326
Use of the explicit Keyword	327
Delegating Constructors	329
The Copy Constructor	330
Accessing Private Class Members	331
Friends	333
The Friend Functions of a Class	333
Friend Classes	336
The this Pointer	336
Returning this from a Function.....	337
const Objects and const Member Functions	338
Casting Away const	339
Arrays of Class Objects	340
The Size of a Class Object.....	341
Static Members of a Class	342
Static Data Members	342
Accessing Static Data Members.....	345
A Static Data Member of the Class Type.....	346
Static Function Members.....	346
Destructors.....	347

Pointers and References to Class Objects.....	350
Using Pointers As Class Members.....	350
Defining the Package Class.....	353
Defining the Truckload Class	354
Implementing the Truckload Class.....	355
Nested Classes.....	359
Summary.....	362
■ Chapter 12: Operator Overloading.....	365
Implementing Operators for a Class.....	365
Operator Overloading.....	366
Operators That Can Be Overloaded.....	366
Implementing an Overloaded Operator.....	367
Global Operator Functions	369
Implementing Full Support for an Operator	370
Implementing All Comparison Operators in a Class.....	371
Operator Function Idioms.....	373
Default Class Members	374
Defining the Destructor	375
When to Define a Copy Constructor.....	377
Implementing the Assignment Operator	377
Implementing Move Operations.....	379
Overloading the Arithmetic Operators	380
Improving Output Operations.....	384
Implementing One Operator in Terms of Another	386
Overloading the Subscript Operator	387
Lvalues and the Overloaded Subscript Operator	392
Overloading Type Conversions	394
Overloading the Increment and Decrement Operators.....	395
Function Objects	396
Summary.....	397

■ Chapter 13: Inheritance	399
Classes and Object-Oriented Programming	399
Hierarchies	400
Inheritance in Classes	401
Inheritance vs. Aggregation.....	401
Deriving Classes	402
protected Members of a Class	405
The Access Level of Inherited Class Members	405
Choosing Access Specifiers in Class Hierarchies	407
Changing the Access Specification of Inherited Members	408
Constructor Operation in a Derived Class.....	409
The Copy Constructor in a Derived Class.....	412
The Default Constructor in a Derived Class	414
Inheriting Constructors	415
Destructors Under Inheritance	416
The Order in Which Destructors Are Called.....	417
Duplicate Data Member Names	418
Duplicate Function Member Names	419
Multiple Inheritance	419
Multiple Base Classes.....	420
Inherited Member Ambiguity	421
Repeated Inheritance	424
Virtual Base Classes	425
Converting Between Related Class Types	426
Summary.....	426
■ Chapter 14: Polymorphism.....	429
Understanding Polymorphism	429
Using a Base Class Pointer	429
Calling Inherited Functions	431
Virtual Functions.....	434

Default Argument Values in Virtual Functions.....	442
Virtual Function Calls with Smart Pointers	444
Using References to Call Virtual Functions.....	444
Calling the Base Class Version of a Virtual Function.....	445
Converting Between Pointers to Class Objects.....	446
Dynamic Casts.....	448
Converting References	450
Determining the Polymorphic Type.....	450
The Cost of Polymorphism	451
Pure Virtual Functions	452
Abstract Classes.....	453
Indirect Abstract Base Classes	456
Destroying Objects Through a Pointer	458
Virtual Destructors.....	460
Summary.....	460
■ Chapter 15: Runtime Errors and Exceptions	463
Handling Errors	463
Understanding Exceptions.....	464
Throwing an Exception	465
The Exception Handling Process	467
Code That Causes an Exception to Be Thrown	469
Nested try Blocks	471
How It Works.....	473
Class Objects as Exceptions.....	474
Matching a Catch Handler to an Exception.....	475
How It Works.....	477
Catching Derived Class Exceptions with a Base Class Handler.....	478
Rethrowing Exceptions.....	480
Catching All Exceptions	483

Functions That Throw Exceptions.....	485
Function try Blocks.....	485
Functions That Don't Throw Exceptions	486
Constructor try Blocks	486
Exceptions and Destructors.....	487
Standard Library Exceptions	487
The Exception Class Definitions	489
Using Standard Exceptions.....	490
Summary	493
■ Chapter 16: Class Templates	495
Understanding Class Templates	495
Defining Class Templates	496
Template Parameters.....	497
A Simple Class Template	498
Defining Function Members of a Class Template	500
Instantiating a Class Template	503
Static Members of a Class Template	508
Non-Type Class Template Parameters.....	509
Templates for Function Members with Non-Type Parameters	512
Arguments for Non-Type Parameters	516
Pointers and Arrays as Non-Type Parameters	516
Default Values for Template Parameters	517
Explicit Template Instantiation	518
Special Cases	518
Using static_assert() in a Class Template	519
Defining a Class Template Specialization	520
Partial Template Specialization.....	521
Choosing between Multiple Partial Specializations.....	521

Friends of Class Templates.....	522
Class Templates with Nested Classes	524
Function Templates for Stack Members	526
Summary.....	530
■ Chapter 17: File Input and Output	533
Input and Output in C++	533
Understanding Streams	534
Advantages of Using Streams.....	535
Stream Classes	536
Standard Stream Objects.....	537
Stream Insertion and Extraction Operations.....	537
Stream Manipulators	539
File Streams	542
Writing a File in Text Mode	543
Reading a File in Text Mode.....	545
Setting the Stream Open Mode	548
Managing the Current Stream Position.....	553
Unformatted Stream Operations	555
Unformatted Stream Input.....	556
Unformatted Stream Output	558
Errors in Stream Input/Output	558
Input/Output Errors and Exceptions.....	560
Stream Operations in Binary Mode	561
Writing Numeric Data in Binary	563
File Read/Write Operations	571
Random Access to a File	572
String Streams	578

Objects and Streams	579
Using the Insertion Operator with Objects.....	579
Using the Extraction Operator with Objects.....	580
Object I/O in Binary Mode.....	582
More Complex Objects in Streams	585
Summary.....	590
Index.....	593

About the Author



Ivor Horton graduated as a mathematician and was lured into information technology with promises of great rewards for very little work. In spite of the reality being a great deal of work for relatively modest rewards, he has continued to work with computers to the present day. He has been engaged at various times in programming, systems design, consultancy, and the management and implementation of projects of considerable complexity.

Ivor has many years of experience in designing and implementing systems for engineering design and manufacturing control. He has developed occasionally useful applications in a wide variety of programming languages, and has taught primarily scientists and engineers to do likewise. His currently published works include tutorials on C, C++, and Java. At the present time, when he is not writing programming books or providing advice to others, he spends his time fishing, traveling, and enjoying life in general.

About the Technical Reviewer



Michael Thomas has worked in software development for more than 20 years as an individual contributor, team lead, program manager, and Vice President of Engineering. Michael has more than 10 years of experience working with mobile devices. His current focus is in the medical sector, using mobile devices to accelerate information transfer between patients and health care providers.