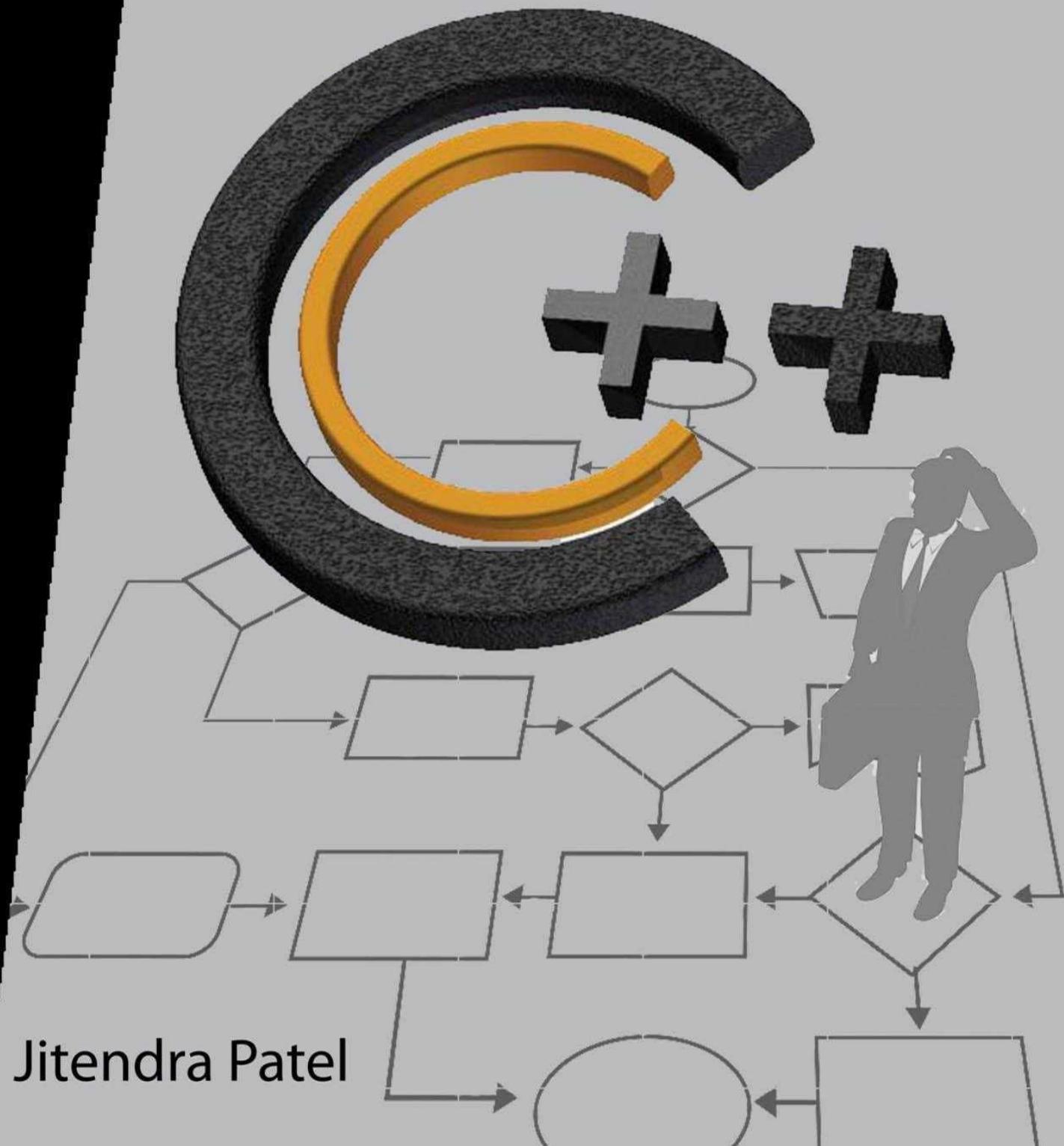


# OBJECT ORIENTED PROGRAMMING USING



Jitendra Patel

# Object Oriented Programming

## Using C++

By Jitendra Patel

### Overview

Object Oriented Programming using C++: Object Oriented Programming using C++ teaches the generic Object Oriented Programming using C++ programming language in an easy-to-follow style, without assuming previous experience in any other language. A variety of examples make learning these Concepts with C++ both fun and practical. This book is organized in such a manner that students and programmers with prior knowledge of C can find it easy, crisp and readable. Each Chapter contains many example programs throughout the book, along with additional examples for further practice.

### KEY FEATURES

- Systematic approach throughout the book
- Programming basics in C++ without requiring previous experience in another language
- Simple language has been adopted to make the topics easy and clear to the readers
- Topics have been covered with more than 100 illustrations and C++ programs
- Enough examples have been used to explain various OOPs concepts effectively. This book also consists of tested programs so as to enable the readers to learn the logic of programming
- Discusses all generic concepts of Object Oriented Programming (OOP) concepts such as Classes and Objects, Inheritance, Polymorphism using Function and Operator Overloading and Virtual Functions, Friend Functions in detail with aided examples
- Use of Various Programming terms like variables and expressions, functions are simplified
- A number of diagrams have been provided to clear the concepts in more illustrative way
- Provides exercises, review questions and exercises as the end of each chapter equipped with more than 300 questions in various patterns and more than 170 programming exercises
- Samples are presented in easy to use way through Turbo C++ 3.0.

First Edition: 2012

## **Copyright**

Object Oriented Programming Using C++ (Second Edition)

Copyright reserved by the Author

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

### Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

## Dedication

This book is dedicated to my honorable parents and beloved students who are my favorite person in the world.

## **PREFACE**

I want to thank a number of people for helping me writing the books and solving the difficulties of Object Oriented Programming Concepts and finding the solution of various critical problems I faced during writing this book.

As the reader of this book, you are my most important critic and commentator. I value your opinion and want to know what I am doing right, what I could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass my way.

As an author of this book, I welcome your comments. You can email or write me directly to let me know what you did or didn't like about this book as well as what I can do to make the book better.

When you write, please be sure to include this book's title and author as well as your name, email address, and phone number. I will carefully review your comments on the book.



# CHAPTER:1 Introduction to Object Oriented Programming

## Overview of Principles of Programming

The program is a set of statements (instructions) to perform a particular task. To write the program you must follow some mechanism and that mechanism is called as programming. Now we will discuss some mechanism of programming.

The most common mechanism for programming are:

1. Algorithm
2. Pseudo code
3. Flow chart

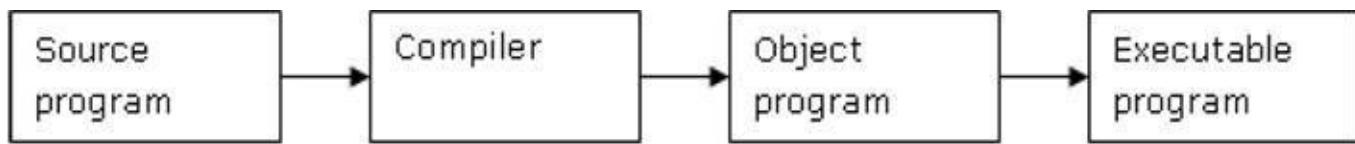
These mechanisms are general mechanism which computer cannot understand and we cannot implement algorithm or flow chart into the computer. For implementing flow chart or algorithm into computer we must use any particular language which computer can understand.

## Procedural programming language

A procedural program is written as a list of instructions, telling the computer, step-by-step, what to do.

Pascal, C, BASIC, COBOL, Fortran and similar traditional programming languages are procedural languages. i.e each statement in the language tells the computer to do something.

Program units include the main block, subroutines, functions, procedures; file scoping; includes/modules; libraries. We can use English words for variable names, English for statement also. Because of this feature human being can read program easily.



[Fig: Steps of procedural language]

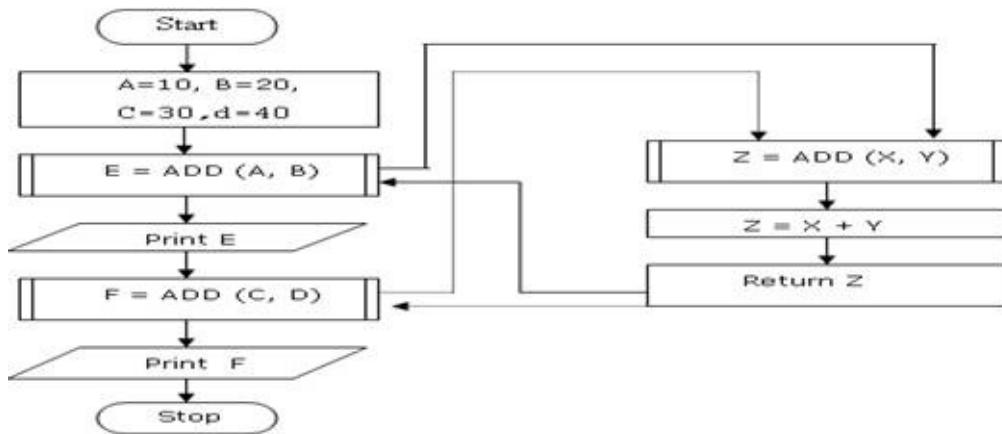
In all procedural languages you have to write code in English using the syntax of language you understand.

The compiler or interpreter will translate source code into object code, which computer can understand.

When you run object code, it makes executable code and you can see the output of your program by executing the executable code.

All procedural languages provide the feature of using functions. With this feature we can divide our code into small modules. When program become large in size, a single list of instructions becomes unreadable. Because of this reason functions are used as a way of making programs more comprehensible. The functions are also called as subroutines, subprograms or procedures.

Let us see an example of a program using function. Here is the flow chart of adding two numbers using function.



**[Fig: Adding two numbers using function]**

As you can see in above Figure the flow chart first we have declared variables A, B, C and D with values 10, 20, 30 and 40 respectively. After that we have called the ADD ( ) function with variables A and B and store the result in E.

When we call the ADD ( ) function, the pointer will go to that function and receives arguments A, B as X, Y. Then the addition of X and Y will stored into variable Z and function ADD ( ) will return value Z to variable E.

In the same way we can add the value of C and D into the variable F by calling ADD ( ) function with arguments C and D.

This is a small example of function used in structured languages. If a program is very large then by dividing the program into this type of functions, we can easily understand the code and we can reduce the size of code.

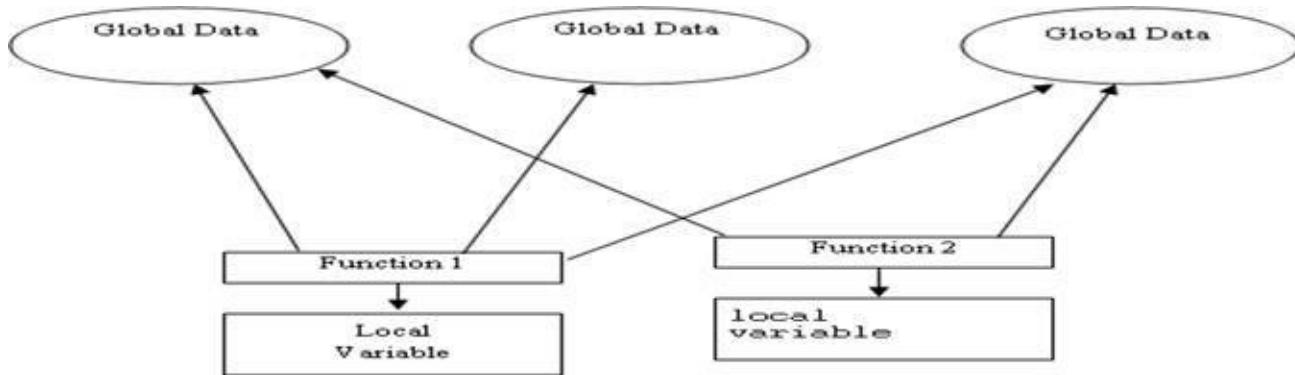
Procedural programming is fine for small projects. It is the most natural way to tell a computer what to do, and the computer processor's own language, machine code, is procedural, so the translation of the procedural high-level language into machine code is straightforward and efficient.

### **Some characteristics of procedure-oriented programming are as follows:**

1. Emphasis(Attention) is on doing things (tasks), i.e. it follows the algorithm.
2. Large programs are divided into smaller programs known as functions or procedures or subroutines.
3. Most of the functions are share global data.
4. Data move openly around the system from function to function.
5. Functions transform data from one form to another.

## Drawbacks

As time goes, the programmers found some problems while developing projects in structured languages. One of the most crucial reason of facing problems with structured language is data. As we have seen one of the characteristic of structured programming is on doing things anyway. The subdivision of a program into function continues this emphasis. Functions do thing, just as single program statements do.



**[Fig: Relationship of data and function in procedural programming]**

Data is, after all, the reason for a program's existence. The important part of any system is data not the functions. In procedure language, data is considered as second aspect rather than first.

Most of the data are treated as global in procedural languages. By global means that the variables that constitute the data are declared out side of any function so they are accessible to all functions. These functions perform various operations on the data. They read it, update it, rearrange it, display it, write it back to the disk, and so on. Where as in case of local variables, they are hidden within a single function. But local variables are not useful for important data that must be accessed by many different functions. The concept of local and global variable will more clear from above figure.

Another problem is that, because many functions access the same data, the way the data is stored(type of data, size of the field, range, format etc...) becomes critical.

The procedural language cannot model the real world problem very well because the problems are functions and data structures of procedural language.

Creating new data types is difficult using procedural languages. Computer languages typically have several built in data types like integer, floating point numbers, characters etc. but if you want to create new data type like complex number, date, two dimension coordinate then it is difficult create those new data types.

C provides to make your own data type using structure but here also structure has a

drawback that we cannot protect data of structure, anyone can access it. This type of problem is called as data hiding.

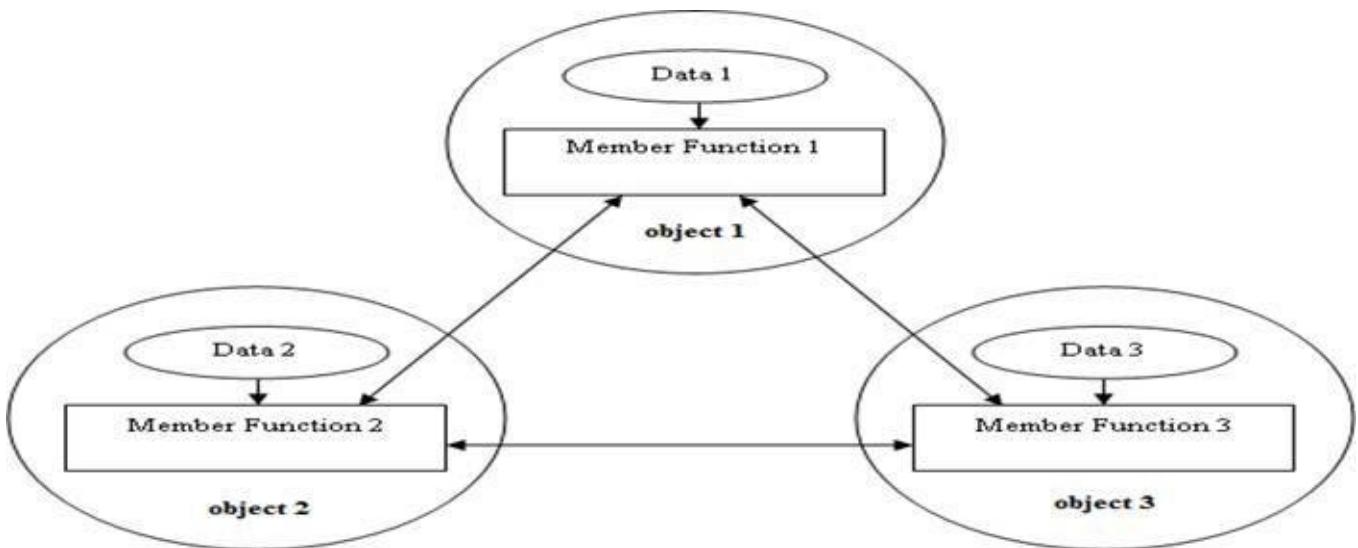
## Object oriented programming.

The fundamental idea behind an object-oriented language is to combine both data and the functions that operate on the data into a single unit. Such a unit is called an object.

Definition: “Object-oriented programming is an approach of modularizing programs by creating classes for both data and functions that can be used as templates for creating copies of such objects on demand.” Thus, an object is considered to store data and set of operations that can access that data. The objects can be used in a variety of different programs without modifications.

The fundamental need of Object Oriented Programming is data hiding, reusability of code, creating your own data type with more flexibility, etc. Data members of an object can be accessed only by using the functions of the object. You cannot access the data of the object directly; you can access data using functions of the object only. The outside functions (like main() function) cannot access the data of the object. Therefore, the data is hidden and is safe from accidental alteration.

The functions of an object are called the member functions. The objects of a C++ program can communicate with each other by calling one another’s member functions. As you can see in following figure three objects object 1, object 2 and object 3 are connected with each other using their member functions. Here data 1 can be accessed by member function 1 only, member function of object 2 and object 3 can not access data 1, so data 1 is hidden for all the objects rather than object 1, same for data 2 and data 3.



[Fig: Organization of data and functions in OOP]

C++ & Java are the examples of Object Oriented Programming language.

In 1980 Bjarne Stroustrup added several extensions to C language. The most important

addition was the concept of CLASS. The addition made were mainly aimed at extending the language in such a way that it supports object-oriented programming.

### **Some of the striking aspects of Object Oriented Programming are:**

1. Emphasis is on data rather than procedure. i.e how data is accessed rather than only how work is done.
2. Programs are divided into what are known as objects.
3. Data structures are designed such that they characterize the objects.
4. Functions that operate on the data of an object are tied together in the data structure.
5. Data is hidden and cannot be accessed by external function like main() function.
6. Objects may communicate with each other through member functions of objects.
7. New data and functions can be easily added whenever necessary.

## **Features of Object Oriented Programming**

The important features of OOPs are as follows:

### **Class**

The set of data and code of an object can be made a user defined data type with the help of a class. Class is a user defined data type which contains data member & member function. It is defined by “class” keyword. It is an important feature of object oriented programming language. Once a class has been defined, we can create any number of objects belonging to that class. Thus a class is a collection of objects of similar type. It is an abstract representation or blueprint of the real world objects.

For Example, “fruit” is a class and “apple”, “mango”, “banana” are its objects.

Same way,

1. student, teacher are objects of class college and
2. laborer, manager are objects of class employee etc.

You can declare an object student of class college in C++ as follows. college student;

### **Object**

Objects are variables of type(user defined) class. An object is a basic run time entity. Object represents a Physical or real world entity. An object is simply something you can

give a name. All the objects have some characteristics and behavior. The state of an object represents all the information held within it and behavior of an object is the set of action that it can perform to change the state of the object.

**All real world objects have three characteristics:**

**1. State**

Determines or tells how object reacts or what an object has?

**2. Behaviour**

Indicates what we can do with this object?

**3. Identity**

Used to give difference between one object to another object?

**For example:** Our bike object has following,

**1. State**

(gear, speed, fuel)

**2. Behaviour**

(changing speed, applying brakes)

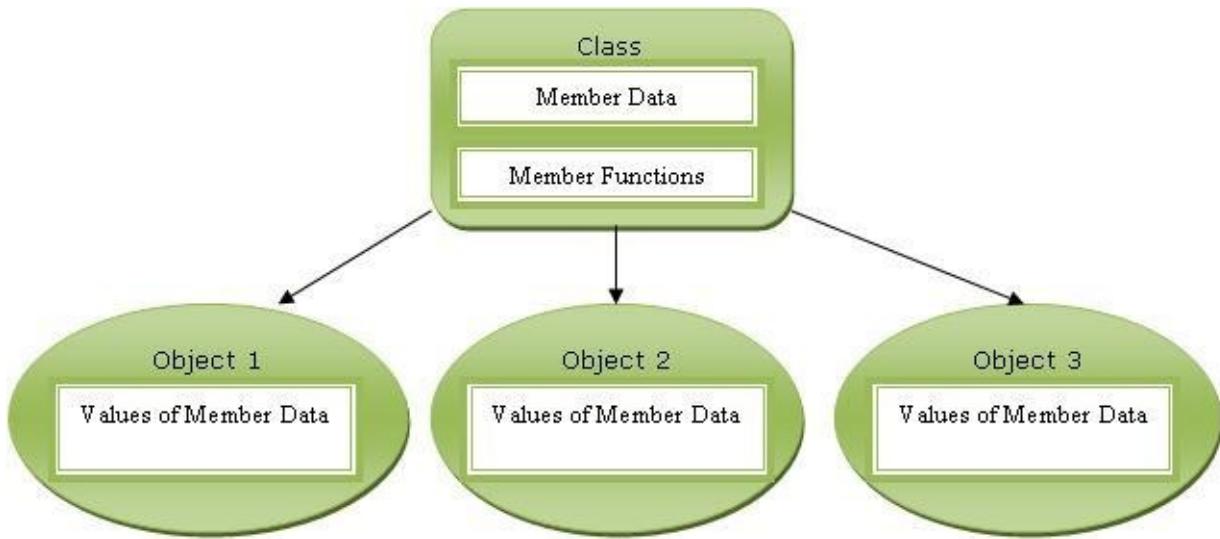
**3. Identity**

(registration number, engine number)

**For example,** in the following statement:

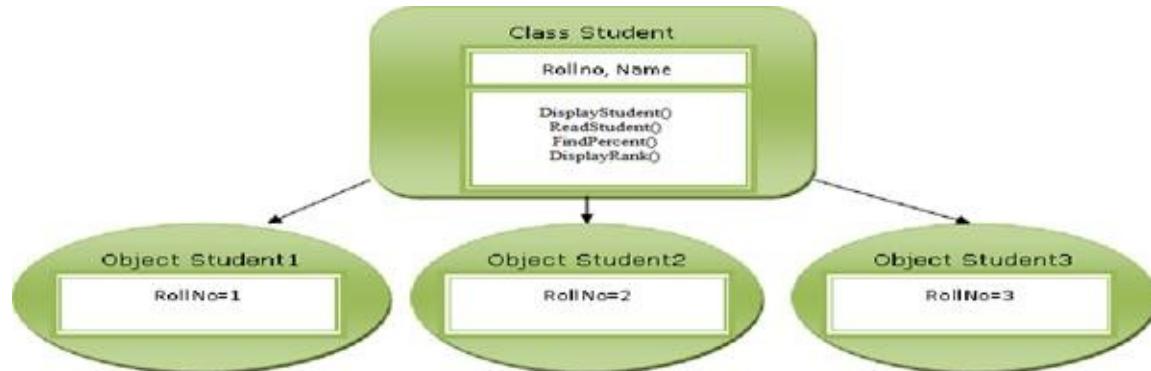
```
int Num1, Num2;
```

Num1, Num2 are two variables of type int. Similarly, you can define many objects of the same class as shown in following figure:



**[Fig: Class and Objects of the Class]**

**For example**, for a student class we can have following illustrations:



**[Fig: An example of an object]**

Object may represent a person, a place, a table of data, a bank account or any item that the program must handle.

It is important to understand the distinction between a class and an object. The two terms are often used interchangeably, however there are noteworthy differences. The differences are summarized below:

Class	Object
• Defines a real world model	• An instance of a class or model
• Declares attributes	• It Has state
• Declares behavior	• It Has behavior
• It is an ADT	• There can be many <i>unique</i> objects of the same class

## Data abstraction

Abstraction is the art of representing the essential features of some thing without including much detail. An abstraction describes a set of objects in terms of an encapsulated or hidden data and operations on that data. This is the property in which only necessary information is extracted. This property is implemented by using the classes in

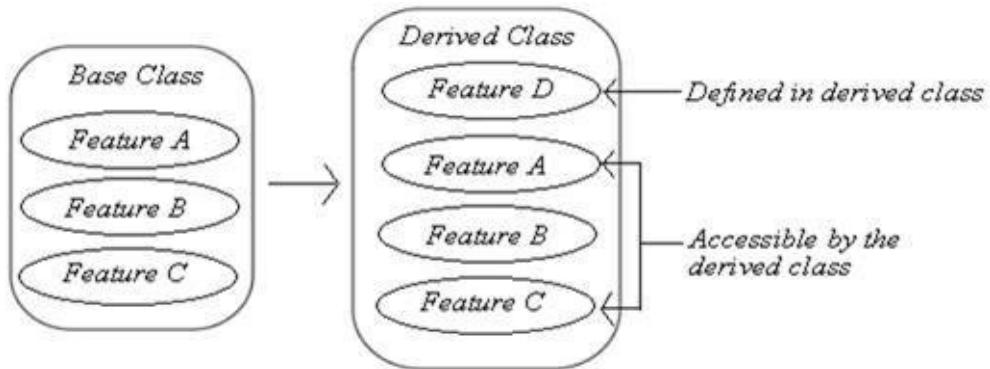
C++.

## Encapsulation

Data encapsulation is the most striking feature of a OOP. Combining data and functions into a single class is known as encapsulation. It is used to hide the data as well as for the binding of a data members and member functions. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called data hiding.

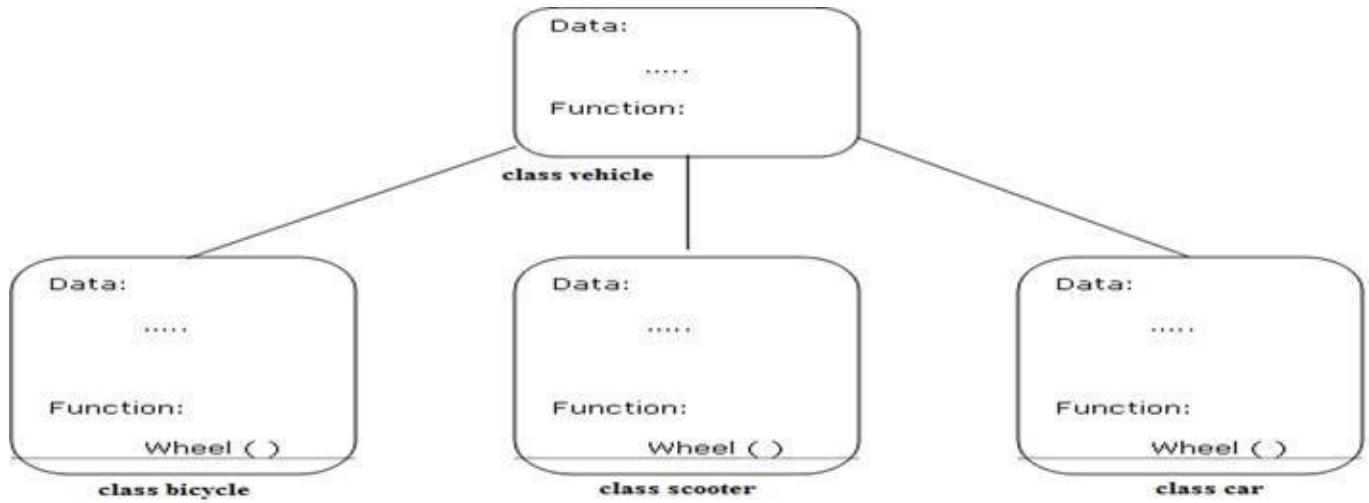
## Inheritance

Inheritance is the process by which one class inherits the properties of another Class. It is the process of creating a new class, called the derived class(subclass), from the existing class, called the base class(superclass). A derived class inherits all attributes and behavior of a base class, i.e., it provides access to all data members and member functions of the base class, and allows additional members and member functions to be added if necessary.



[Fig: Inheritance of a Derived Class From a Base Class]

**For example,** In following figure the base class is named as vehicle and it contains functions named wheel( ), Seats( ) and Horn( ). This class is derived into classes named bicycle, scooter and car. So the functions from vehicle are common to all derived classes. when we call a function wheel( ) from class bicycle then this function will have data of bicycle class, same for scooter and car. The functions paddle( ),kick( ) and door( ) are declared in bicycle, scooter and car respectively and they are not derived from vehicle class, so the function paddle( ) cannot be used by any classes rather than bicycle, same for functions kick( ) and door( ).



**[Fig: Inheritance]**

The base class and derived class have an “is a” relationship. **For example,**

1. Baseball (a derived class) is a Sport (a base class)
2. Pontiac (a derived class) is a Car (a base class)

## Reusability

The concept of inheritance provides an important feature to the object-oriented language called reusability. A programmer can take an existing class and, without modifying it, add additional features and capabilities to it. This is done by deriving a new class from an existing class.

## Data hiding

Data hiding is the property in which some of the members of object are restricted from outside access. This is implemented by using private and protected access specifies while declaring the class.

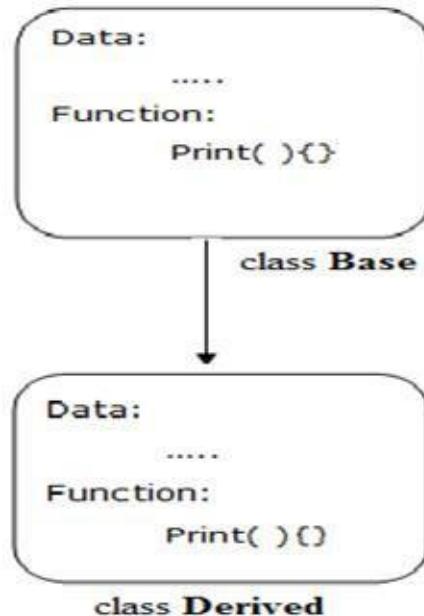
## Polymorphism

The word polymorphism is derived from two Latin words poly (many) and morphos (forms). Poly means many and morphs means form, so polymorphism means one name multiple form.

Polymorphism is the ability of different objects to respond differently to virtually the same function.

**For example,** a base class provides a function to print the current contents of an object. Through inheritance, a derived class can use the same function without explicitly defining its own. However, if the derived class must print the contents of an object differently than

the base class, it can override the base class's function definition with its own definition.



**[Fig: Polymorphism]**

In order to invoke polymorphism, the function's return type and parameter list must be identical. Otherwise, the compiler ignores polymorphism.

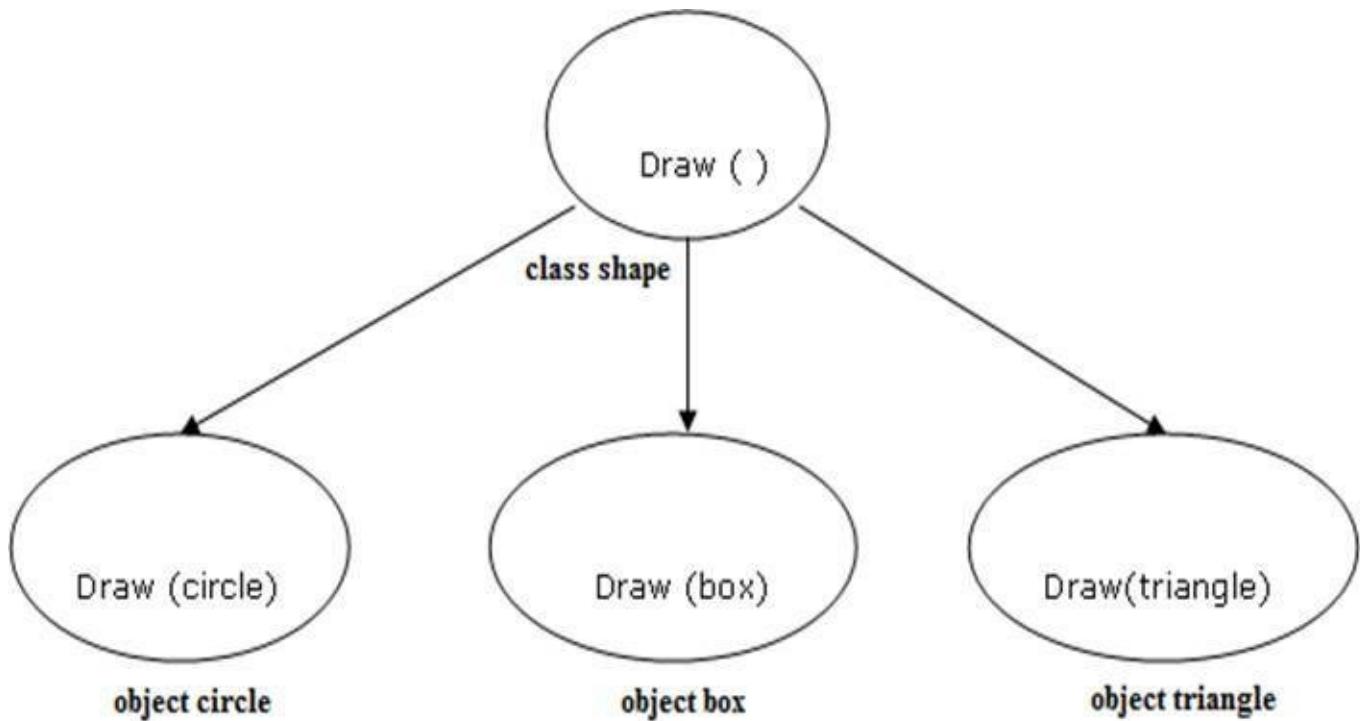
**There are two types of polymorphism :**

- 1. Compile time polymorphism**
  - A. Using Function overloading or**
  - B. Using Operator Overloading**

The concept of using operators or functions in different ways, depending on what they are operating on, is called compile time polymorphism.

- 2. Run time polymorphism.**
  - A. Using Virtual Functions**

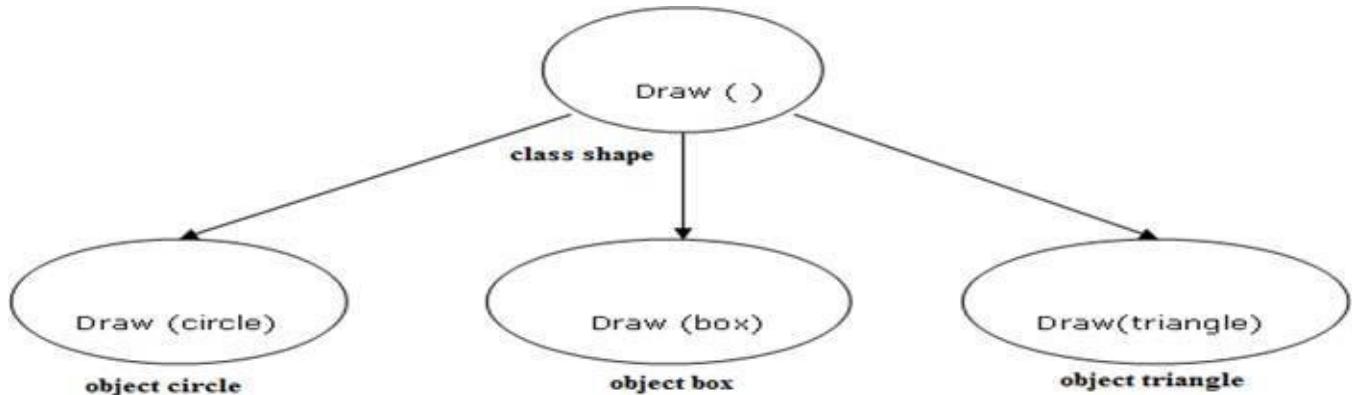
As you can see in the following figure by inheritance, every object will have the draw( ) procedure of base class shape as well as its own. The draw( ) procedure will be redefined in each class that draws that particular object. At runtime, the code matching the object under current reference will be called. Using the object of shape class we can call any of the four draw() function by assigning the reference of related object.



**[Fig: Polymorphism]**

## Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a object depends on the dynamic type of that reference.



**[Fig: Dynamic Binding of objects with different draw() function]**

**E.g.** Consider the procedure “draw” in above figure. By using inheritance, every object of all four classes will have this procedure. The draw procedure will be redefined in each subclass that defines the specific shape drawing functionality. Now consider following code,

Shape s1; Circle c1; Box b1;

```
s1.draw() //will bind the draw function of Shape class
```

```
s1=&c1;
```

```
s1.draw() //will bind the draw function of Circle class
```

```
s1=&b1;
```

```
s1.draw() //will bind the draw function of Box class
```

Therefore at run-time, which draw function will be executed depends on the object of the class we use.

## Message Passing

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:

1. Creating classes that define objects and their behavior,
2. Creating objects from class definitions, and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. A message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired result. Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent. Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

## Advantages or Benefits of oop

1. Development of secure programs that cannot be accessed by code in other parts of the program.
2. Using OOP, existing code within an application doesn't have to be modified to accommodate changes in the implementation.
3. Encourages modularity in an application development.
4. Offers better maintainability of code.
5. Existing code can be reused in other applications.
6. Promises greater programmer productivity, better quality of software and lesser maintenance cost.
7. We can build programs from the standard existing and working code thru inheritance, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
8. Object-oriented systems can be easily upgraded from small to large systems.
9. It is easy to partition the work in a project based on objects.
10. Message passing makes the interface descriptions with external systems much simpler.
11. The data-centered design approach enables us to capture more details of a model in implementable form.
12. With OOP software complexity can be easily managed.

## Applications of Oop

1. Graphical user interface design such as windows. Hundreds of windowing systems have been developed, using the OOP techniques. Real-business systems are often much more complex and contain many more objects with complicated attributes and methods. OOP is useful in these types of applications because it can simplify a complex problem.
2. Real-time systems
3. Simulation and modeling
4. Object-oriented databases
5. Hypertext, hypermedia and expertext

## 6. AI and expert systems

7. Neural networks and Parallel programming Decision support and office automation systems

## 8. CIM/CAM/CAD systems

9. Object-oriented technology is certainly going to change the way the software engineers think, analyze, design and implement future systems.

### Difference Between POP and OOP

Procedural Oriented Programming Language	Object Oriented Programming Language
POP follows Top-Down approach	OOP follows Bottom-Up approach
With POP effort is on doing things (Algorithms), i.e. it follows the algorithm	With OOP effort is on data hiding rather than creating procedures.
Large programs are divided into smaller programs known as functions or procedures or subroutines. In POP, most of the functions share global data.	Programs are divides into what are known as objects.  In POP, Data is hidden and cannot be accessed by external function(non-member function).
Functions transform data from one form to another.	Functions can not directly transform data from one to another.
POP identifies tasks; how something is to be done Data is considered as second aspect rather than first in POP.	POP hides how an object performs its tasks  Data is considered first and most important aspect in OOP.
Large program become excessively complex with POP.	It is very easy to solve large and complex programs using OOP.
The procedural language cannot model the real world problem very well. Creating new data types is difficult using procedural languages.  <i>Pascal, C, BASIC, COBOL, Fortran and similar traditional programming languages are procedural languages</i>	OOP models the real world problem very well.  Using class like concepts, it is very easy to create any new complex type of user requirement. <i>C++, JAVA, Visual basic, ADA, Smalltalk are some of the OOP languages.</i>

## Exercise

- 1.** Which of the following is a feature of Object oriented programming?
  - a.** Data Encapsulation.
  - b.** Inheritance.
  - c.** Polymorphism.
  - d.** All.
- 2.** Define the following terms:
  - a.** Algorithm **b.** Flow chart **c.** Machine language **d.** Data Encapsulation **e.** Inheritance
  - f.** Polymorphism
- 3.** Explain the drawbacks of Assembly language.
- 4.** Explain the characteristics of Procedural Language.
- 5.** Write algorithm and draw flow chart for finding maximum of three numbers.
- 6.** Explain various features of Object Oriented Programming features.
- 7.** Give difference between POP and OOP.



## CHAPTER:2 Fundamentals of C++

### History of C++ Language

C++ is an Object Oriented Programming language. Initially named ‘C with classes’. C++ was developed by Bjarne Stroustrup at AT&T Bell laboratories, USA, in 1980 and is based on the C language.

1 <sup>st</sup> Generation	2 <sup>nd</sup> Generation	3 <sup>rd</sup> Generation	Generation Gap (1970- 1990)	Object-Based & OO- Generation 1990-
Fortran I	Fortran II	PL/1		
ALGOL 58	ALGOL 60	ALGOL 68		Ada (Contribn. from Alphard CLU)
	COBOL	PASCAL		Object PASCAL, Eiffel Smalltalk
	LISP	SIMULA		C++ (Contribn. from C) CLOS (Contribution from LOOPS+ & Flavors)

**[Fig:- Table: Evolution of Generation of Languages]**

The first object-oriented language was Simula and SmallTalk. Stroustrup wanted to combine the best of Simula and strong supporter of C languages and to create a more powerful language that could support object oriented programming features and the result was C++.

C++ is also called as extension of C. In 1983 the name was changed to ‘C++’ from ‘C with classes’ because of increment operator ++ of C. The “++” is a syntactic construct used in C (to increment a variable), and C++ is intended as an incremental improvement of C .

C++ provides three important features of OOP namely classes, inheritance, polymorphisms. These features enable us to create abstract data type, inherit properties from existing data type and overloading of functions and operators. C++ implements “data abstraction” using a concept called “classes”.

## C++ is a “Better C”

C++ is a better C because C++ retains C as a subset, it gains many of the attractive features of the C language, such as efficiency, closeness to the machine, and a variety of built-in types. A number of new features were added to C++ to make the language even more robust, many of which are not used by novice programmers.

Most of these features can be summarized by two important design goals:

1. Strong compiler type checking and
2. A user-extensible language.

By enforcing stricter type-checking, the C++ compiler makes us actually aware of data types in our expressions. Stronger type checking is provided through several mechanisms, including: function argument type checking, conversions, and a few other features.

C++ also enables programmers to incorporate new types into the language, through the use of classes. A class is a user-defined type. The compiler can treat new types as if they are one of the built-in types. This is a very powerful feature. In addition, the class provides the mechanism for data abstraction and encapsulation, which are key to object-oriented programming.

## Some Additional Features of C++

C++ is an object-oriented programming (OOP) language. It offers most of the advantages of OOP by allowing the developer to create user-defined types for modeling real world situations. However, the real power within C++ is contained in its features.

### 1. Reference variable

Using referene variable we can create an alias to the variable memory location in C++. So the same memory location can be reffered by two or more names. Any changes to the location's value will be reflected for all the alias. This feature was added to C++ so that references to data types (user-defined or built-in) could be specified. This allows passing a complex data structure as an argument to a function without having to precede it with the address operator.

### 2. Operator Overloading

Operator overloading allows the developer to define basic operations (such as arithmetic operations) for objects of user-defined data types as if they were built-in data types. For example, a conditional expression of two string objects such as:

```
if(s1 == s2)
{
    ...
}
```

is much easier to read than

```
if(strcmp(s1.getStr(),s2.getStr()) == 0)
{
    ...
}
```

Operator overloading is often referred to as “syntactic sugar.”

### 3. Generic Programming

One benefit of generic programming is that it eliminates code redundancy. Consider the following function:

```
void swap(int &first,int &second)
```

```
{  
int ts = second;  
second = first;  
first = ts;  
}
```

This function is sufficient for swapping elements of type int. If it is necessary to swap two floating-point values, then the same function must be rewritten using type float for every instance of type int. The basic algorithm is the same. The only difference is the data type of the elements being swapped. Additional functions must be written in the same manner to swap elements of any other data type. This is, of course, very inefficient. The template mechanism was designed for generic programming so that we can use same function for different data type.

#### 4. Exception Handling

The exception handling mechanism is a more robust method for handling errors . It is a convenient means for handling the errors rather than crashing the program without knowing the reason for the crash. In C++ we can use exception handling mechanism.

## C++ Advantages and Claims

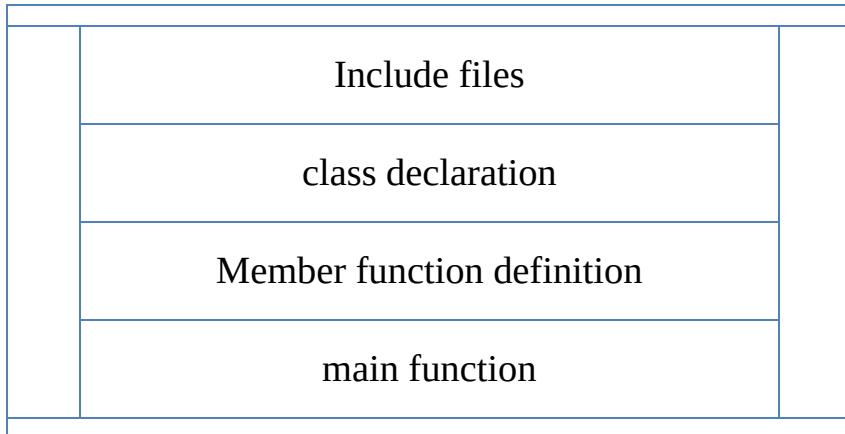
Often it is said that programming in C++ leads to 'better' programs. Some of the claimed advantages of C++ are:

1. New programs would be developed in less time because old code can be reused.
2. Creating and using new data types in C++ would be easier than in C.
3. The memory management under C++ would be easier and more transparent to use.
4. Programs would be less bug-prone, as C++ uses a stricter syntax and type checking.
5. 'Data hiding', the usage of data by one program part while other program parts cannot access the data, would be easier to implement with C++.

C++ in particular (and OOP in general) is of course not the solution to all programming problems. However, the language does offer various new and elegant facilities which are worth investigating. At the same time, the level of grammatical complexity of C++ has increased significantly compared to C. This may be considered a serious disadvantage of the language.

## Structure of a C++ program

Probably the best way to start learning a programming language is by writing a program. A typical c++ program would contain four sections as shown in figure.

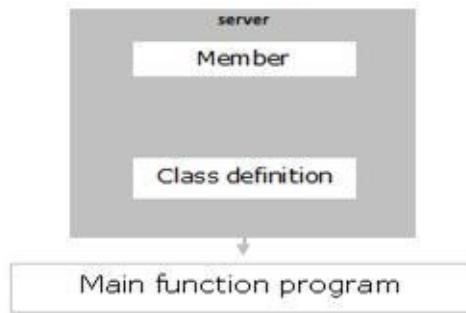


It is common practice to organize the program into 3 separate files.

1. One header file for class declaration
2. Second header file for member function definition
3. Third file for main function program which includes the above two header files.

This approach enables the programmer to separate the abstract specification of the interface (class definition) from implementation details (member function definitions).

This approach is based on client server model as shown in below figure.



The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

Therefore, here is our first program:

```
// my first program in C++  
#include <iostream.h>  
int main ()
```

```
{  
    cout << "Hello World!";  
    return 0;  
}
```

It is one of the simplest programs that can be written in C++. We are going to look line by line at the code we have just written:

### **// my first program in C++**

This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not have any effect on the behavior of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

### **#include <iostream.h>**

Lines beginning with a hash sign (#) are directives for the preprocessor. In this case the directive `#include <iostream.h>` tells the preprocessor to include the `iostream` standard file. This specific file (`iostream`) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

### **int main ()**

The `main` function is the entry point for execution of the C++ program. The instructions contained within this function's definition will always be the first ones to be executed in any C++ program. For that same reason, it is essential that all C++ programs must have a `main` function. The word `main` is followed in the code by a pair of parentheses (()). That is because it is a function declaration. Optionally, these parentheses may enclose a list of function parameters. After these parentheses we can find the body of the `main` function enclosed in braces ({}). What is contained within these braces is what the function does when it is executed.

### **cout << "Hello World";**

This line is a C++ statement. `cout` represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the "Hello World" sequence of characters) into the standard output stream (which usually is the screen) to display onto the screen. `cout` is declared in the `iostream` standard file, that's

the reason why we need to include that file in our program. Notice that every C++ statement ends with a semicolon character (;) as like C statement. This character marks the end of the C++ statement.

**return 0;**

The return statement causes the main function to finish. return may be followed by a return code (in our example is followed by the return code 0). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

## C++ Comments

Comments are those statements of the source code ignored by the compiler. They simply do nothing. Their purpose is only to allow the programmer to insert explanation of the code and additionally date of creation of the program etc... C++ supports two ways to insert comments:

```
// Single line comment  
/* block  
comment */
```

The first of them, known as line comment, discards everything from where the pair of slash signs (//) is found up to the end of that same line.

The second one, known as block comment, discards everything between the /\* characters and the first appearance of the \*/ characters, with the possibility of including more than one line.

We are going to add comments to our second program:

```
/* my second program in C++  
with more comments */  
  
#include <iostream.h>  
  
int main ()  
{  
    cout << "Hello World! "; // prints Hello World!  
    cout << "I'm a C++ program"; // prints I'm a C++ program  
    return 0;  
}
```

### Output:

Hello World! I'm a C++ program

If you include comments within the source code of your programs without using the comment characters combinations //, /\* or \*/ , the compiler will take them as if they were C++ expressions, most likely causing one or several error messages when you compile it.

# Creating, Compiling and Linking C++ program

## Creating

We can create a c++ program using a simple editor like a notepad or a rich and powerful editor of C++ like Turbo C++ or MS VC++.

## Compiling

To turn your source code into a program, you use a compiler. How you invoke your compiler, and how you tell it where to find your source code, will vary from compiler to compiler.

In Borland's Turbo C++ you pick the RUN menu command or type

```
tc <filename>
```

from the command line, where <filename> is the name of your source code file (for example, test.cpp). Other compilers may do things slightly differently.

For the Borland C++ compiler:

```
bcc <filename>
```

For the Borland C++ for Windows compiler:

```
bcc <filename>
```

For the Borland Turbo C++ compiler:

```
tc <filename>
```

For the Microsoft compilers:

```
cl <filename>
```

After your source code is compiled, an object file is produced. This file is often named with the extension .OBJ. This is still not an executable program, however. To turn this into an executable program, you must run your linker.

## Linking and executing

C++ programs are typically created by linking together one or more OBJ files with one or more libraries. A library is a collection of linkable files with ready made functionalities like standard input and output streams to be used in program. All C++ compilers come with a library of useful functions (or procedures) and classes that you can include in your program.

The steps to create an executable file are:

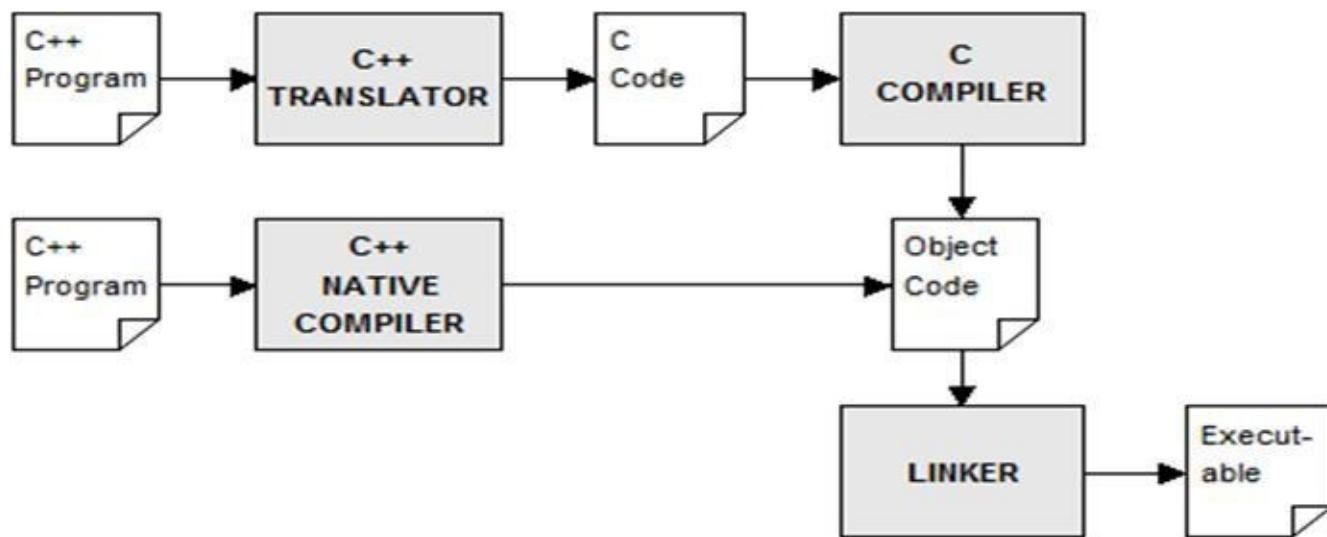
1. Create a source code file, with a .CPP extension.
2. Compile the source code into a file with the .OBJ extension.
3. Link your OBJ file with any needed libraries to produce an executable program.

## How C++ Compilation Works

Compiling a C++ program involves a number of steps (most of which are transparent to the user):

1. First, the C++ preprocessor goes over the program text and carries out the instructions specified by the preprocessor directives (e.g., #include). The result is a modified program text which no longer contains any directives.
2. Then, the C++ compiler translates the source program code. The compiler may be a true C++ compiler which generates native (assembly or machine) code, or just a translator which translates the code into C. In the latter case, the resulting C code is then passed through a C compiler to produce native object code. In either case, the outcome may be incomplete due to the program referring to library routines which are not defined as a part of the program. For example, << operator which is actually defined in a separate IO library.
3. Finally, the linker completes the object code by linking it with the object code of any library modules that the program may have referred to. The final result is an executable file.

Following figure illustrates the above steps for both a C++ translator and a C++ native compiler. In practice all these steps are usually invoked by a single command (e.g., CC) and the user will not even see the intermediate files generated.



[Figure:C++ Compilation]

## Comparison of C and C++ language

C Language	C++ Language
C is a Procedure Oriented Programming Language.	C++ is an Object Oriented Programming Language.
C is a Middle Level Language.	C++ is a Higher Level Language.
C uses a top down approach for problem solving methods.	C++ uses a bottom up approach for problem solving methods.
In C, file extension is .C	In C++, file extension is .cpp
In C program we can not use // for comments	In C++ program we can use // for comments
C does not provide strict type checking.	C++ provides strict type checking.
In C, scanf() and printf() are used as I/O functions in C. The header file required to include for this purpose is "stdio.h".	In C++, cout and cin are used as input/output facility in C++. The header file required to include for this purpose is "iostream.h".
C requires format specifier for printing & scanning variables.	C++ does not require format specifier for printing & scanning variables.
With C we can not pass parameters by reference.	With C++ we can pass parameters by reference.
C doesn't support object oriented features.	C++ supports object oriented features such as class, object, data hiding, data abstraction, inheritance, polymorphism, message passing and dynamic binding.
In C basis is on developing procedures.	C++ basis is on the data and objects rather than only functions.
In C structures can not have functions as members.	In C++ structures are like classes, so declaring functions is legal and allowed.

## Input and output operators in c++

In C Language `scanf()` and `printf()` are used for input and output respectively, whereas in C++ Language, `cin` and `cout` operators are used for input and output respectively. C++ Language I/O operators can be used by including `iostream.h` header file in the program.

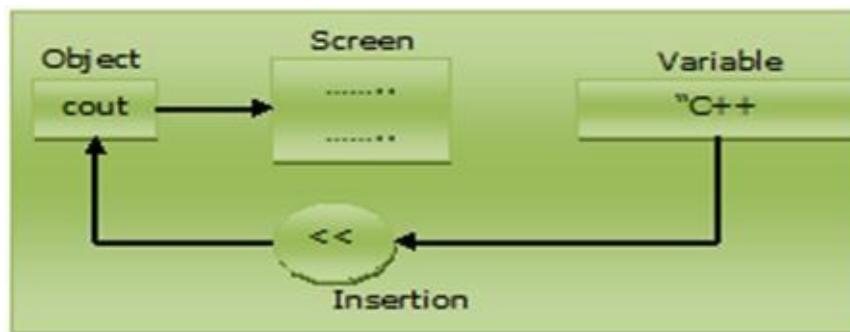
### Output(Insertion) Operator

The `cout` operator can be used with an Insertion Operator or put to Operator (`<<`).

#### Syntax:

```
cout << variable_name;
```

The Insertion Operator inserts the contents of the variable on its right to the object of its left. As shown in figure.



#### Example:

```
#include<iostream.h>

#include<conio.h>

void main()

{

char str[]="hello world";

int integer=10;

cout<< "C++ Language";

cout<< str;

cout<< 15;

cout<< integer;

cout<< "integer";
```

```
getch();
```

```
}
```

### Output:

C++ Language

hello world

15

10

Integer

Here, no need to specify data types of any string or integer variable (like %d, %s, %c, etc).

It is important to notice that cout does not add a line break after its output unless we explicitly indicate it, therefore, the following statements:

```
cout << "This is a sentence.;"
```

```
cout << "This is another sentence.;"
```

will be shown on the screen without any line break between them:

This is a sentence.This is another sentence.

even though we had written them in two different cout.

In order to perform a line break on the output we must explicitly insert a new-line character into cout. In C++ a new-line character can be specified as \n (backslash, n) or endl manipulator:

```
cout << "First sentence.\n ";
```

```
cout << "Second sentence.\nThird sentence.";
```

### This produces the following output:

First sentence.

Second sentence.

Third sentence.

Additionally, to add a new-line, you may also use the endl manipulator. **For example:**

```
cout << "First sentence." << endl;
```

```
cout << "Second sentence." << endl;
```

**would print out:**

First sentence.

Second sentence.

The endl manipulator produces a newline character, exactly as the insertion of '\n' does, but it also has an additional behavior when it is used with buffered streams: the buffer is flushed. Anyway, cout will be an unbuffered stream in most cases, so you can generally use both the '\n' escape character and the endl manipulator in order to specify a new line without any difference in its behavior.

## Input(Extraction) Operator

The cin operator can be used with an Extraction Operator or get from Operator (<<).

**Syntax:**

```
cin >> variable_name;
```

The Extraction Operator extracts the value from the keyboard and assigns it to the variable on its right. As shown in figure.



**Example:**

```
#include<iostream.h>
```

```
#include<conio.h>
```

```
void main()
```

```
{
```

```
int integer;
```

```
cout << "Enter value : ";
```

```
cin >> integer;
```

```
cout<<integer;  
getch();  
}
```

### Output:

Enter value : 20

20

### cin and strings

We can use cin to get strings with the extraction operator (>>) as we do with fundamental data type variables:

```
cin >> mystring;
```

However, cin extraction stops reading as soon as it finds any blank space, so in this case we will be able to get just one word for each extraction. This behavior may or may not be what we want; for example if we want to get a sentence from the user, this extraction operation would not be useful. In order to get entire lines, we can use the function getline, which is the more recommendable way to get user input with cin:

```
// cin with strings  
  
#include <iostream.h>  
  
#include <string>  
  
int main ()  
{  
    string mystr;  
  
    cout << "What's your name? ";  
  
    getline (cin, mystr);  
  
    cout << "Hello " << mystr << ".\n";  
  
    cout << "What is your favorite game? ";  
  
    getline (cin, mystr);  
  
    cout << "I like " << mystr << " too!\n";  
  
    return 0;
```

}

## Output:

What's your name? Jitendra Patel

Hello Jitendra Patel.

What is your favorite game? Hockey

I like The Hockey too!

Notice how in both calls to getline we used the same string identifier (mystr). What the program does in the second call is simply to replace the previous content by the new one that is introduced.

## Cascading I/O Operators

The insertion operator (<<) may be used more than once in a single statement:

```
cout<<"Hello,"<<"I am "<<"a C++ statement";
```

This last statement would print the message Hello, I am a C++ statement on the screen. The utility of repeating the insertion operator (<<) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

```
cout<<"Hello,I am"<<age<<"years old and my zipcode is"<< zipcode;
```

If we assume the age variable to contain the value 29 and the zipcode variable to contain 384002 the output of the previous statement would be:

Hello, I am 29 years old and my zipcode is 384002

In C++ the statements

```
int Total = 5;
```

```
cout<<"The value of Total variable is “;
```

```
cout<< Total;
```

**will display the output as**

The value of Total variable is 5.

It can be written in single cout statements as:

```
cout<<"The value of Total variable is “<< Total;
```

You can also use cin to request more than one datum input from the user:

```
cin >> a >> b;
```

is equivalent to:

```
cin >> a;
```

```
cin >> b;
```

In both cases the user must give two data, one for variable a and another one for variable b that may be separated by any valid blank separator: a space, a tab character or a newline.

Using single cin or cout operator, any number of variables can be read or write.

### **Example:**

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int height, width;
    cout<< "Enter Height and Width : ";
    cin>> height<< width;
    cout<<"Height = "<<height<<"\nWidth = "<<width;
    getch();
}
```

### **Output:**

```
Enter Height and Width : 5 50
```

```
Height = 5
```

```
Width = 50
```

## C++ Header Files

As previously mentioned, a header is what is used to call the type of library. It is wrapped between < > signs. The following table lists this down:

Stream	File to include	Description
Iostream	#include <iostream.h>	Allows for variable declarations for Input/Output Streams
Fstream	#include <fstream.h>	Contains ifstream (similar to istream) and ofstream (similar to ostream)
Cmath	#include <cmath.h>	Cmath allows for the use of functions that contain upper-level math manipulation such as sin, cos, tan, etc.
String	#include <string.h>	Includes functions for string manipulations
Iomanip	#include <iomanip.h>	Allows for input/output manipulation

Iomanip allows the user to modify the display to the screen using manipulators:

iomanip.h		
setprecision(n)	cout << setprecision(n);	Sets the precision of the data via the amount of decimal places. Requires the header iomanip.
ios::fixed	cout << ios::fixed;	Sets a fixed decimal format for output devices
ios::showpoint	cout << ios::fixed << ios::showpoint;	Force the output device to show the decimal point and trailing zeros.
setw(n)	cout << setw(5) << X	Sets the amount of space per column in bits. If the amount of space is greater than the variable, it will be right-justified.
ios::flush	cout << ios::flush;	Clears the buffer and prompts dialog to the next line.
setfill (char)	cout << setfill ('#')	Fills all extra space in the column with the ASCII character.
ios::left	cout << ios::left	Left justifies output
ios::right	cout << ios::right	Right justifies output

### Example:

```
#include<iostream.h>
```

```
#include<conio.h>
```

```
#include<iomanip.h>
```

```
void main()
```

```
{
```

```
float value;
```

```
clrscr();
```

```
value=12.34;  
  
cout<< setfill('0')<<setw(8)<<value<<"\n";  
  
cout<< setfill('#')<<setw(8)<<value<<"\n";  
  
value=12.34567;  
  
cout<< setprecision(2)<<value<<"\n";  
  
getch();  
}
```

**Output:**

00012.34

##12.34

12.35

## C++ Tokens

The smallest individual units in a program are known as tokens, C++ has the following tokens:

**Example of tokens:-** } , { , " " , int

1. Identifiers
2. Keywords
3. Constants

## Identifier

In our everyday, we give names to different things so they can be referred easily. Similarly, in C++, we use identifiers to name user created entities. Which may be:

1. Variables e.g. i,j,sum,avg,height,area etc...
2. Functions e.g. add(), display() etc...
3. Type e.g. a class

### Rules to declare identifiers:

1. An identifier can be combination of letters, numbers, and underscores with following restrictions:
  - a. It should start with a letter or underscore. E.g. height, my\_height,\_myHeight are allowed but not 1isGod
  - b. If it starts with a underscore then the first letter should not be capital because such names are reserved for implementation. E.g. \_Height not allowed
2. It should be unique in a program taking care that C++ is case sensitive. E.g. age and Age are different variables
  - a. A keyword cannot be used as an identifier.
3. There is no restriction on length of the identifier. E.g. h and h\_represents\_my\_height are both valid.

Besides restrictions, there are certain guidelines which you should follow:

1. Use meaningful descriptive names.  
E.g. int Age is better than int a.

2. If description makes identifier name too long then put a comment before identifier and make identifier shorter
3. Be consistent in your naming convention.
4. Use small letters for single word identifier name.
5. For multiword identifiers, either use underscore separated or intercropped notation.  
E.g. `get_my_height ()` or `getMyHeight ()`
6. Use Hungarian notation.  
**E.g.** `double dFlowRate`, `int value`, `bool check`.
7. Don't use similar identifier names in a program like `Speed`, `speed`, and `Speedy`
8. Don't use capitalized version of a keyword like `Return`

## Keywords

Keywords are predefined reserved identifiers that have special meanings. They cannot be used as identifiers in your program. Keyword is a word that the compiler already knows, i.e. when the compiler sees a keyword somewhere in the program it knows what to do automatically.

**For example**, when the compiler encounters the keyword ‘int’, it knows that ‘int’ stands for an integer type. Or if the compiler reads a ‘break’, then it knows that it should break out of the current loop. Some common keywords are-

asm	auto	break	bool	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	extern
false	float	for	friend	goto
if	inline	int	long	mutable
namespace	new	operator	private	protected
public	register	reinterpret_cast	return	short
signed	sizeof	static	static_cast	struct
switch	template	this	throw	true
try	typedef	typeid	typename	union
unsigned	using	virtual	void	volatile
wchar_t	while			

## Constants/Literals

As the name suggests, a variable is something whose value can be changed throughout the program. It is not fixed. On the other hand, a constant is one whose value remains the same (constant) throughout the program. So constants are expressions with a fixed value.

Literals are used to express particular values within the source code of a program. **For example**, when we write:

```
a = 5;
```

the 5 in this piece of code is a literal constant.

Literal constants can be divided in Integer Numerals, Floating-Point Numerals, Characters, and Strings constants.

## Integer Numerals

Numerical constants identify integer decimal values. **For example**:

```
1776
```

```
707
```

```
-273
```

Notice that to express a numerical constant we do not have to write quotes ("") nor any special character. There is no doubt that it is a constant: whenever we write 1776 in a program, we will be referring to the value 1776.

C++ allows the literal constants of octal numbers (base 8) and hexadecimal numbers (base 16). If we want to express an octal number we have to precede it with a 0 (zero character). And in order to express a hexadecimal number we have to precede it with the characters 0x (zero, x). For example, the following literal constants are all equivalent to each other:

```
75 // decimal
```

```
0113 // octal
```

```
0x4b // hexadecimal
```

All of these represent the same number: 75 (seventy-five) expressed as a base-10 numeral, octal numeral and hexadecimal numeral, respectively.

Literal constants, like variables, are considered to have a specific data type. By default, integer literals are of type int. However, we can force them to either be unsigned by appending the u character to it, or long by appending l:

```
75 // int
```

```
75u // unsigned int
```

75l // long

75ul // unsigned long

In both cases, the suffix can be specified using either upper or lowercase letters.

## Floating Point Numbers

They express numbers with decimals and/or exponents. They can include either a decimal point, an e character (that expresses “by ten at the Xth height”, where X is an integer value that follows the e character), or both a decimal point and an e character:

3.14159 // 3.14159

6.02s23 // 6.02 x 10<sup>23</sup>

1.6e-19 // 1.6 x 10<sup>-19</sup>

3.0 // 3.0

These are four valid numbers with decimals expressed in C++. The first number is PI, the second one is the number of Avogadro, the third is the electric charge of an electron (an extremely small number) -all of them approximated- and the last one is the number three expressed as a floating-point numeric literal.

The default type for floating point literals is double. If you explicitly want to express a float or long double numerical literal, you can use the f or l suffixes respectively:

3.14159L // long double

6.02s23f // float

Any of the letters than can be part of a floating-point numerical constant (e, f, l) can be written using either lower or uppercase letters without any difference in their meanings.

## Character and string literals

There also exist non-numerical constants, like:

Single character constants, For example:

‘z’

‘p’

String literals composed of several characters. For example:

“Hello world”

“How do you do?”

Notice that to represent a single character we enclose it between single quotes (‘) and to express a string (which generally consists of more than one character) we enclose it between double quotes (“).

When writing both single character and string literals, it is necessary to put the quotation marks surrounding them to distinguish them from possible variable identifiers or reserved keywords. Notice the difference between these two expressions:

x

‘x’

x alone would refer to a variable whose identifier is x, whereas ‘x’ (enclosed within single quotation marks) would refer to the character constant ‘x’.

Character and string literals have certain peculiarities, like the escape codes. These are special characters that are difficult or impossible to express otherwise in the source code of a program, like newline (\n) or tab (\t). All of them are preceded by a backslash (\). Here you have a list of some of such escape codes:

ESCAPE CHARACTER	DESCRIPTION
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\b	backspace
\f	form feed (page feed)
\a	alert (beep)
\'	single quote (‘)
\"	double quote (“”)
\?	question mark (?)
\\\	backslash (\)

**For example:**

‘\n’

‘\t’

“Left \t Right”

“one\ntwo\ntthree”

Additionally, you can express any character by its numerical ASCII code by writing a backslash character (\) followed by the ASCII code expressed as an octal (base-8) or hexadecimal (base-16) number. In the first case (octal) the digits must immediately follow

the backslash (for example \23 or \40), in the second case (hexadecimal), an x character must be written before the digits themselves (for example \x20 or \x4A).

String literals can extend to more than a single line of code by putting a backslash sign (\) at the end of each unfinished line.

“string expressed in \

two lines”

You can also concatenate several string constants separating them by one or several blank spaces, tabulators, newline or any other valid blank character:

“this forms” “a single” “string” “of characters”

## Symbolic Constants

### Using #define directive

You can define your own names for constants that you use very often without having to resort to memory-consuming variables, simply by using the #define preprocessor directive. Its format is:

```
#define identifier value
```

#### For example:

```
#define PI 3.14159265
```

```
#define NEWLINE '\n'
```

This defines two new constants: PI and NEWLINE. Once they are defined, you can use them in the rest of the code as if they were any other regular constant, **for example:**

```
// defined constants: calculate circumference
```

```
#include <iostream.h>
```

```
#define PI 3.14159
```

```
#define NEWLINE '\n'
```

```
int main ()
```

```
{
```

```
    double r=5.0; // radius
```

```
    double circle;
```

```
circle = 2 * PI * r;  
cout << circle;  
cout << NEWLINE;  
return 0;  
}
```

### Output:

31.4159

When compiler preprocessor encounters `#define` directives it replaces any occurrence of their identifier (in the previous example, these were `PI` and `NEWLINE`) by the code to which they have been defined (3.14159265 and ‘\n’ respectively).

The `#define` directive is not a C++ statement but a directive for the preprocessor; therefore it assumes the entire line as the directive and does not require a semicolon (`;`) at its end. If you append a semicolon character (`;`) at the end, it will also be appended in all occurrences within the body of the program that the preprocessor replaces.

### Using Const keyword

In C++, any value declared as `const` cannot be modified in the program. The syntax of `const` is as follows:

```
const <data type> <variable name> = <value>;
```

#### For example,

```
const float PI = 3.14159;
```

```
float area = PI * r * r;
```

Preceding a variable definition by the keyword `const` makes that variable read-only (i.e., a symbolic constant). A constant must be initialized to some value when it is defined. For example:

```
const int maxSize = 128;
```

```
const double pi = 3.141592654;
```

Once defined, the value of a constant cannot be changed:

```
maxSize = 256; // illegal!
```

If we don't specify data type while declaring constant, by default it takes that variable as int.

```
const maxSize = 128; // maxSize is of type int
```

## Variable

A variable is the storage location in memory that is stored by its value. A variable is identified or denoted by a variable name. The variable name is a sequence of one or more letters, digits or underscore.

## Variable declaration

A variable in C++ must be declared (the type of variable) and defined (values assigned to a variable) before it can be used in a program. Following example shows how to declare a variable.

```
int a;
```

means a is declared as integer variable

## Rules of variable declaration

1. A variable name can have one or more letters or digits or underscore.
2. White space, punctuation symbols or other characters are not permitted to denote variable name.
3. A variable name must begin with a letter or underscore.
4. Variable names cannot be keywords or any of the reserved words of the C++ programming language.
5. C++ is a case-sensitive language. Variable names written in capital letters differ from variable names with the same name but written in small letters. For example, the variable name CIST is different from the variable name cist.

## Variable Definition vs Declaration

<b>Declaration</b> <code>int a</code>	Describe information "about" the variable, doesn't allocate memory cell for the variable
<b>Definition</b> <code>int a=5</code>	Tell the compiler about the variable: its type and name, as well as allocated a memory cell for the variable

## The difference of declaring variables in C and C++

In C and C++, all the variables we must declare before they are used. C requires all the variables to be defined at the beginning of a scope where as in C++ whenever we need a variable to use, declare at that time and assign the values. Sometimes in between the program we need a variable which we have not declared in the beginning so in C, first we

have to declare that variable in the beginning and then we use it, whereas in C++ wherever we need a variable, declare there and assign a value to it. **For example,**

C program	C++ program
<pre>main() {     float avg;     int sum, j;     sum = 0;     for(j=1; j&lt;=5; ++j)         sum += j;     avg = sum / j;     printf("%f", avg); }</pre>	<pre>main() {     int sum = 0;     for( int j=1; j&lt;=5; ++j)         sum += j;     float avg = sum / j;     cout &lt;&lt; avg; }</pre>

As you can see in C program, first we have declared all the variables then we initialize variable sum with value 0, after for loop we have assigned the value of sum / j to avg and printed to screen that value. Whereas in C++ program we have written same program, we have declared and initialized variable sum with value 0 in single statement, in for loop we need a variable j and we declare and initialize variable j with value 1 there. After for loop we want to assign the value of sum / j to variable avg but we have not declared that variable in beginning so we have declared and initialized that variable after for loop.

## Scope of variables

All the variables that we intend to use in a program must have been declared with its type specifier at an earlier point in the code before we use them.

A variable can be either of global or local scope.

A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.

Global variables can be referred from anywhere in the code, even inside functions after its declaration. The scope of local variables is limited to the block enclosed in braces ({} ) where they are declared. For example, if they are declared at the beginning of the body of a function (like in function main) their scope is between its declaration point and the end of that function. For example, local variables declared in main could not be accessed from the other function and vice versa.

## Reference Variables

References are a new type of variable introduced by C++. References are mainly used to pass parameters to functions and to return values from them.

A reference is not a copy of the variable to which the reference is made. Instead, it is

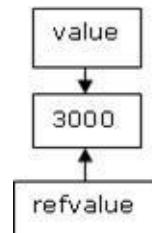
the same variable, disguised with a different name.

In contrast to pointers, once a reference is associated to a variable, that association stays permanently (within the block where the association was made).

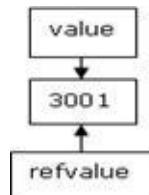
References are defined using the unary operator &

```
int value = 3000;
```

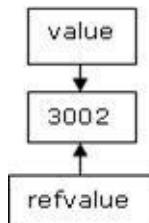
```
int &refValue = value;
```



```
value++;
```



```
refValue++;
```



```
// both, value and refValue have 3002.
```

The address of the memory region being referenced is the same for both names:

```
cout << &value << ' ' << &refValue; //should be the same
```

It is possible to make one variable to be another using reference:

```
#include <iostream.h>
```

```
int main ()
```

```
{
```

```
double a = 3.1415927;
```

```
double &b = a; // b is a
```

```

b = 89;

cout << "a contains: " << a << endl; // Displays 89.

return 0;

}

```

If you are used at pointers and absolutely want to know what happens, simply think double &b = a is translated to double \*b = &a and all subsequent b are replaced by \*b. The value of REFERENCE b cannot be changed after its declaration. For example you cannot write, a few lines further, &b = c expecting now b is c. It won't work. Everything is said on the declaration line of b. Reference b and variable a are married on that line and nothing will separate them.

## References as Function Parameters

References can be used to pass arguments between functions in a call-by-reference style (by default, C and C++ pass arguments using call-by-value).

### Example:

Call-by-reference using pointer	Call-by-reference using reference
<pre> #include &lt;iostream.h&gt; void negate(int *i); main() {     int x;      x = 123;     cout &lt;&lt; x &lt;&lt; " negated is ";     negate(&amp;x);     cout &lt;&lt; x &lt;&lt; "\n";     return 0; } void negate(int *i) {     *i = -*i; } </pre>	<pre> #include &lt;iostream.h&gt; void negate(int &amp;i); main() {     int x;      x = 123;     cout &lt;&lt; x &lt;&lt; "negated is ";     negate(x);     cout &lt;&lt; x &lt;&lt; "\n";     return 0; } void negate(int &amp;i) {     i = -i; // don't need * } </pre>
Output: 123 negated is -123	123 negated is -123

References can be used to allow a function to modify a calling variable:

```

#include <iostream.h>

void change (double &r, double s)

{
    r = 100;
    s = 200;
}

```

```
int main ()  
{  
    double k, m;  
    k = 3;  
    m = 4;  
    change (k, m);  
    cout << k << ", " << m << endl; // Displays 100, 4.  
    return 0;  
}
```

If you have used pointers in C and wonder how exactly the program above works, here is how the C++ compiler would translate it to C:

```
#include <iostream.h>  
  
void change (double *r, double s)  
{  
    *r = 100;  
    s = 200;  
}  
  
int main ()  
{  
    double k, m;  
    k = 3;  
    m = 4;  
    change (&k, m);  
    cout << k << ", " << m << endl; // Displays 100, 4.  
    return 0;  
}
```

A reference can be used to let a function return a variable:

```
#include <iostream.h>

double &biggest (double &r, double &s)

{

if (r > s) return r;

else return s;

}

int main ()

{

double k = 3;

double m = 7;

cout << "k: " << k << endl; // Displays 3

cout << "m: " << m << endl; // Displays 7

cout << endl;

biggest (k, m) = 10;

cout << "k: " << k << endl; // Displays 3

cout << "m: " << m << endl; // Displays 10

cout << endl;

biggest (k, m) ++;

cout << "k: " << k << endl; // Displays 3

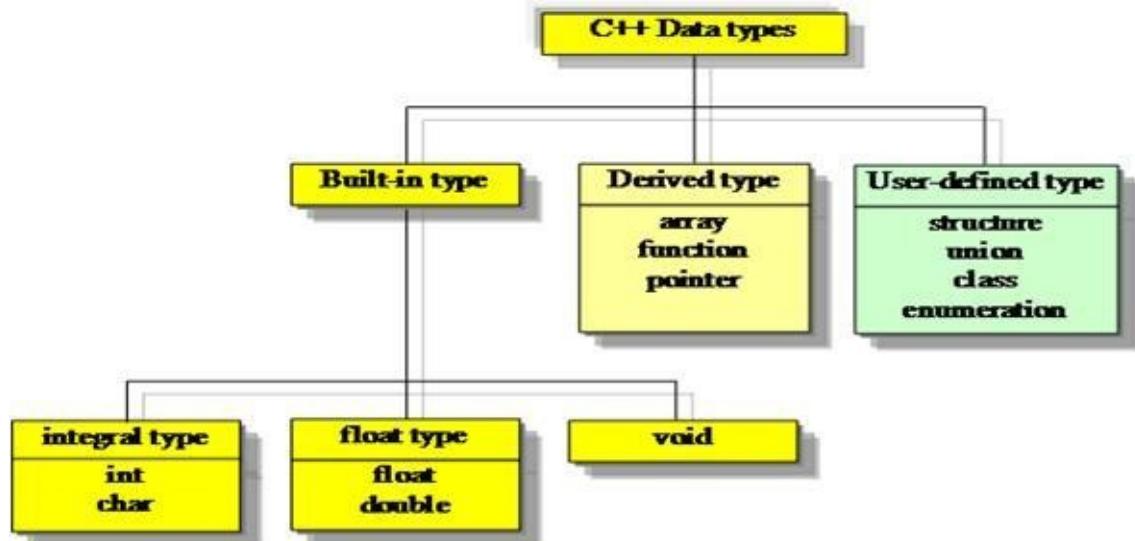
cout << "m: " << m << endl; // Displays 11

cout << endl;

return 0;

}
```

## C++ DATA TYPES



[Fig: Hierarchy of C++ data types]

As you can see in above figure various categories of C++ data types are shown. The signed, unsigned, long and short may be applied to character and integer basic data types.

Name	Description	Size	Range
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	3.4e +/- 38 (7 digits)
double	Double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
long double	Long double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
wchar_t	Wide character.	2 or 4 bytes	1 wide character

## Enumerated Data Type

It is a user-defined type, which provides a way for attaching names to numbers. The enum keyword automatically enumerates a list of words by assigning them 0, 1, 2 and so on. The syntax of enum statement is similar to struct statement. **For example,**

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

```
enum month{January, February, March, April, May, June, July, August,  
September, October, November, December };  
enum logical{Yes, No};
```

In C++, the tag names day, month and logical become new type names. That means we can declare new variables using this tag names. **For example,**

```
day today;  
month currentmonth;
```

Like C, C++ does not permit an int value to be automatically converted to an enum value.

```
day today = Friday; // valid in C++  
day today = 5; // Invalid in C++  
day today = (day) 5; // valid in C++
```

We can store the int value of enumerated value in integer variable as follows:

```
int d = Friday; //The value of d will be 5.
```

As we have seen that by default, the enumerators are assigned integer values starting with 0, 1, 2 and so on. We can assign explicitly integer values to the enumerators. **For example,**

```
enum day{Sunday, Monday, Tuesday = 4, Wednesday = 6, Thursday,  
Friday, Saturday = 11};
```

As you can see in the given example, Sunday will have value 0 by default, Monday will also have value 1 by default, we have explicitly assign a value 4 and 6 to Tuesday and Wednesday respectively, we have not explicitly assign value for Thursday and Friday so by default it will assign values 7 and 8 resp. because we have assign value 6 to Wednesday, and for last variable Saturday we have assign 11 explicitly.

## Operators of c++

All the operators of C are also available in C++. In addition, there are some special operators for special tasks. Operator can be unary (involve 1 operand) , binary(involve 2 operands),and ternary(involve 3 operands). All the C operators we have seen in earlier chapter are valid in C++ and some operators included in C++, which are as follows:

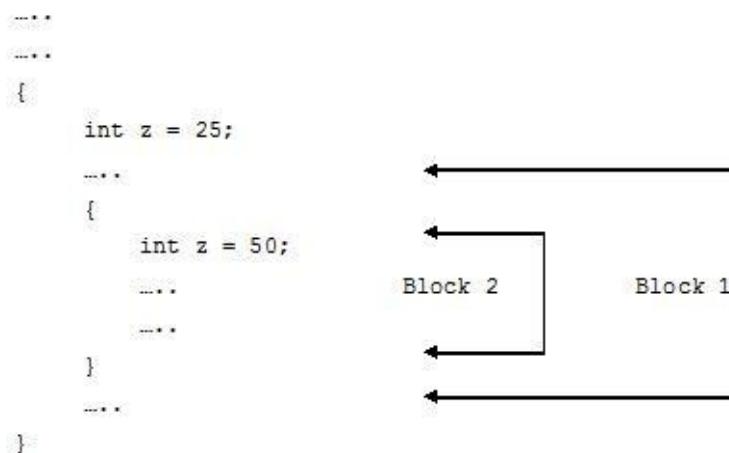
Operator	Description
<code>&lt;&lt;</code>	Insertion Operator
<code>&gt;&gt;</code>	Extraction Operator
<code>::</code>	Scope Resolution Operator
<code>::*</code>	Pointer to Member Declarator
<code>-&gt;*</code>	Pointer to Member Operator
<code>.*</code>	Pointer to Member Operator
<code>Delete</code>	Memory Release Operator
<code>endl</code>	Line feed Operator
<code>New</code>	Memory allocation Operator

[Table: C++ Operators Not available in C]

## Scope Resolution Operator

In C, we can declare two variables with same name in different blocks so they will have different value in their block; that means in one block we can not use a variable value of another block with same name declared in that block. Having local and global variables introduces the problem of defining the scope of the variables. The general rule is that local variables have precedence over global ones. C++ provides a scope resolution operator `::`. With this operator, local variable names can be distinguished from global names.

**For example,**



Here both declaration of variable `z` will have different memory locations containing different values. The statement in Block2 cannot refer to variable `z` declared in the Block1. If we want to use a value of variable `z` of Block2 in Block1, the scope resolution operator

is used in C++. A global variable can be accessed even if another variable with the same name has been declared inside the function using scope resolution operator as follows:

```
#include <iostream.h>

double a = 128;

int main ()
{
    double a = 256;
    cout << "Local a: " << a << endl;
    cout << "Global a: " << ::a << endl;
    return 0;
}
```

**Example 2:**

```
int g = 9.81; // gravity in mts/sec

void main ()
{
    int g = 3.27; // gravity at the moon

    char origin;

    cout << "Please enter your origin [E = Earth, M=Moon] ";
    cin >> origin;

    cout << "The gravity at your location is: ";

    if (origin == 'M')
        cout << g; // print the local variable
    else
        cout << ::g; // print the global variable
}
```

The need for scope resolution derived from the requirement of having name spaces, used to recognize function having synonym names, but belonging to different classes.

## Pointer to member operators

C++ provides us to access the class members through pointers. C++ provides a set of three pointer to member operators those are as follows:

1. `::*` - is used to declare a pointer to a member of a class.
2. `.*` - is used to access a member using object name and a pointer to that member.
3. `->*` - is used to access a member using a pointer to the object and a pointer to that member.

## Memory management operators: new and delete

We use dynamic allocation of memory whenever we don't know in advance that how much memory space is needed. In C, for dynamic allocation of memory we can use malloc and calloc functions and to release the memory we are using free function.

For the same task in C++, C++ introduces two additional operators for the dynamic memory management they are: new and delete operator. These two operators perform the task of allocating and freeing the memory in a better and easier way. The keywords new and delete can be used to allocate and deallocate memory. They are much sweeter than the functions malloc and free from standard C. They are also known as free store operator.

### The new operator

The new operator can be used to create objects of any type. The syntax of new operator is as follows:

```
pointer-variable = new data type;
```

Here, pointer variable is a pointer of type data-type. The new operator allocates memory to hold a data object of type data-type and returns the address of the object. The data-type may be of any valid data type. **For example,**

```
int *p; float *q;
```

```
p = new int ;
```

```
q = new float;
```

Here, p is a pointer of type int and q is a pointer of type float. p and q must have already been declared as pointers of appropriate types or we can combine the declaration of pointers and their assignments as follows:

```
int *p = new int;  
float *q = new float;  
*p = 55;  
*q = 33.33;
```

We can also initialize the memory using the new operator. The general form of using new operator to initialize memory is as follows:

```
pointer-variable = new data type (value);
```

The given allocation of memory to p and q with initialization is now as follows:

```
int *p = new int (55);  
float *q = new float (33.33);
```

We can also allocate the memory to the array using new operator as follows:

```
pointer-variable = new data type [size];
```

**For example,**

```
int *p = new int [20];
```

## The delete operator

When a data object is no longer needed, it is destroyed to release the memory space for reuse in C++ for performing this task delete operator is used. The general form of delete operator is as follows:

```
delete pointer-variable;
```

**For example,**

```
delete p;
```

```
delete q;
```

If you want to release the memory of an array, this task also can be performed by delete operator as follows:

```
delete [ ] p;
```

## Advantages of using new and delete operator over malloc and free function

The new and delete operators have several advantages over the function malloc.

1. It automatically computes the size of the data object. We need not use the operator `sizeof`.
2. It automatically returns the correct pointer type, so that there is no need to use a type cast.
3. It is possible to initialize the object while creating the memory space.
4. Like any other operator, `new` and `delete` can be overloaded.

### **Example using new and delete operator**

```
#include <iostream.h>

#include <cstring.h>

int main ()

{

double *d;

d = new double; // new allocates a zone of memory

// large enough to contain a double

// and returns its address.

// That address is stored in d.

*d = 45.3; // The number 45.3 is stored

// inside the memory zone

// whose address is given by d.

cout << "Type a number: ";

cin >> *d;

*d = *d + 5;

cout << "Result: " << *d << endl;

delete d; // delete deallocates the

// zone of memory whose address

// is given by pointer d.

// Now we can no more use that zone.
```

```
d = new double[15]; // allocates a zone for an array
// of 15 doubles. Note each 15
// double will be constructed.
// This is pointless here but it
// is vital when using a data type
// that needs its constructor be
// used for each instance.

d[0] = 4456;
d[1] = d[0] + 567;
cout << "Content of d[1]: " << d[1] << endl;
delete [] d; // delete [] will deallocate the
// memory zone. Note each 15
// double will be destructed.
// This is pointless here but it
// is vital when using a data type
// that needs its destructor be
// used for each instance (the ~
// method). Using delete without
// the [] would deallocate the
// memory zone without destructing
// each of the 15 instances. That
// would cause memory leakage.

int n = 30;
d = new double[n]; // new can be used to allocate an
// array of random size.
for (int i = 0; i < n; i++)
{
```

```
d[i] = i;  
}  
delete [] d;  
char *s;  
s = new char[100];  
strcpy (s, "Hello!");  
cout << s << endl;  
delete [] s;  
return 0;  
}
```

### Output:

Type a number: 123

Result: 128

Content of d[1]: 5023

Hello!

## Precedence of operators

When writing complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. For example, in this expression:

$a = 5 + 7 \% 2$

we may doubt if it really means:

$a = 5 + (7 \% 2)$  // with a result of 6, or

$a = (5 + 7) \% 2$  // with a result of 0

The correct answer is 6. There is an established order with the priority of each operator of C++. From greatest to lowest priority, the priority order is as follows:

Level	Operator	Description	Grouping
1	::	scope	Left-to-right
2	() [] . -> ++ --	postfix	Left-to-right
3	++ -- ~ ! sizeof new delete	unary (prefix)	Right-to-left
	* &	indirection and reference (pointers)	
	+ -	unary sign operator	
4	(type)	type casting	Right-to-left
5	. * -> *	pointer-to-member	Left-to-right
6	* / %	multiplicative	Left-to-right
7	+ -	additive	Left-to-right
8	<< >>	shift	Left-to-right
9	< > <= >=	relational	Left-to-right
10	== !=	equality	Left-to-right
11	&	bitwise AND	Left-to-right
12	^	bitwise XOR	Left-to-right
13		bitwise OR	Left-to-right
14	&&	logical AND	Left-to-right
15		logical OR	Left-to-right
16	?:	conditional	Right-to-left
17	= *= /= %= += -= >>= <<= &= ^=  =	assignment	Right-to-left
18	,	comma	Left-to-right

Grouping defines the precedence order in which operators are evaluated in the case that there are several operators of the same level in an expression. All these precedence levels for operators can be manipulated by removing possible ambiguities using parentheses signs ( and ), as in this **example**:

`a = 5 + 7 % 2;`

might be written either as:

`a = 5 + (7 % 2);`

or

`a = (5 + 7) % 2;`

depending on the operation that we want to perform.

So if you want to write complicated expressions and you are not completely sure of the precedence levels, always include parentheses. It will also become a code easier to read.

C++ Program to demonstrate the operator's hierarchy.

```
#include<iostream.h>
```

```
main ()
```

```
{
```

```
float a, b, c x, y, z;
```

```
a = 9;
```

```
b = 12;
```

```
c = 3;
```

```
x = a - b / 3 + c * 2 - 1;
```

```
y = a - b / (3 + c) * (2 - 1);
```

```
z = a - ( b / (3 + c) * 2) - 1;
```

```
cout<<"x ="<<x;
```

```
cout<<"y ="<<y;
```

```
cout<<"z ="<<z;
```

```
}
```

## Output

```
x = 10.00
```

```
y = 7.00
```

```
z = 4.00
```

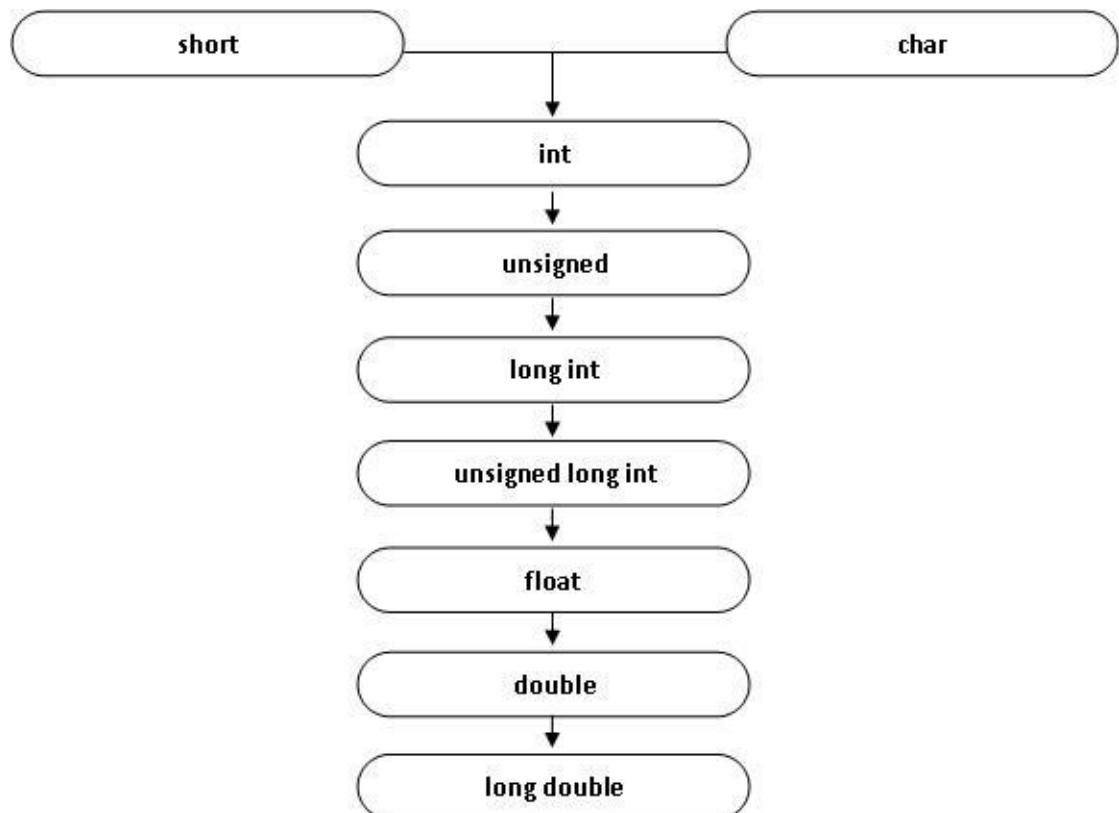
## Expressions and implicit conversions

An expression is a combination of operators, constants and variables. It may also include function calls which return values. Expression may be of constant expressions, integer expressions, float expressions and pointer expressions. **For example,**

$20 + 5 / 2.5$

$x * y -10$

Whenever data-types are mixed in an expression, C++ performs the automatic conversion. It divides the expression into sub expressions consisting of one operator and one or two operands. The compiler converts one of them to match with other using certain rule. The rule is that the smaller type is converted to the wider type. This rule will be more clear using following figure.



[Fig: Hierarchy of type conversion ]

## Excesize

### Short Questions

- 1.** What is C++? What is OOP?
- 2.** What are some advantages of C++?
- 3.** Does C++ run on machine 'X' running operating system 'Y'?
- 4.** What C++ compilers are available?
- 5.** Is there a translator that turns C++ code into C code?
- 6.** Are there any C++ standardization efforts underway?
- 7.** Where can I ftp a copy of the latest ANSI-C++ draft standard?
- 8.** Is C++ backward compatible with ANSI-C?
- 9.** Explain the use of following terms:
  - a.** cin, cout
  - b.** enum
  - c.** const
- 10.** Describe the meaning of sentence "dynamic initialization of variable is valid in C++".
- 11.** Explain the memory management operators in brief.
- 12.** How can a '::' operator be used as unary operator?
- 13.** Explain enumerated data types. Demonstrate the usage of enumeration constants.
- 14.** What are enumerated data types? Explain with an example.
- 15.** List the different phases of C++ program execution.
- 16.** What is a variable in C++? How to form a variable?

### True/ False

- 1.** Every C++ program must have a function named main.
- 2.** In C++, a block (compound statement) is not terminated by a semicolon.
- 3.** If a program compiles successfully, it is guaranteed to execute correctly.
- 4.** Every program must have a main function

5. A programming language is said to be case-sensitive if it considers uppercase letters to be different from lowercase letters.
6. During the compilation phase, a C++ source program is processed first by the preprocessor program and then by the compiler.
7. In C++ subprograms are referred as “functions”.
8. In a C++ function, the statements enclosed by a {} pair are known as the body of the function.
9. Some C++ reserved words can also be used as programmer-defined identifiers.
10. A variable is a location in memory, referenced by an identifier, in which a data value that can be changed is stored.
11. Any constant value written in a program is called a literal.
12. An arrangement of identifiers, literals, and operators that can be evaluated to compute a value of a given type.
13. The C++ statement

alpha /= beta + 25;

is equivalent to the statement

alpha = alpha / (beta + 25);

14. In C++, the expressions beta++ and ++beta can always be used interchangeably.
15. Alphanumeric characters are stored in the computer as integers.

16. The statement

char someChar = “”;

stores the apostrophe (single quote) character into someChar.

17. The code segment

string str = “HI!”;

cout << str[1];

outputs the single character I.

18. In the data type defined by

enum Colors {RED, GREEN, BLUE};

the enumerators are ordered such that RED > GREEN > BLUE.

19. Assuming that the internal representation of the character 'e' is the integer 101, the output of the following code fragment is e 101.

```
char ch = 'e';  
cout << ch << ' ' << int(ch) << endl
```

20. When passing an argument of one data type to a parameter of a different data type, either promotion or demotion occur.

21. You are writing a program that will add three floating-point variables x, y, and z. Their likely value ranges are as follows:

x: 1.0 to 1.9

y: 2.1E-10 to 3.0E-10

z: -3.5E-1 to -2.2E-1

You should add z and x, then add y to obtain the most accurate answer.

22. If int variable someInt contains a value from 0 through 9, then the statement

```
someChar = char('0' + someInt);
```

stores the corresponding digit character into someChar.

## Multiple Choice Questions

1. Which of the following statement is incorrect.

- a) A constant variable must be initialized at the time of its declaration.
- b) An escape sequence represents a single character.
- c) A string literal is a sequence of characters surrounded by double quotes.
- d) All correct.

2. The multiple use of input or output operators in one statement is called.

- a) multi-user
- b) Cascading
- c) Double operator
- d) Operands

3. Which of the following is not a valid identifier?

- a)** Hi\_There
- b)** Top40
- c)** 3Blind
- d)** CAPs

4. Which of the following is a valid string assignment?

- a)** "name" = Jones;
- b)** Name = "Jones" ;
- c)** Name = 'D' + "Jones" ;
- d)** b and c above

5. Which of the following can be assigned to a character variable?

- a)** 't'
- b)** '2'
- c)** '\$'
- d)** All of the above

6. In C++, the phrase "standard output device" usually refers to:

- a)** The keyboard
- b)** A floppy disk drive
- c)** The display screen
- d)** A CD-ROM drive

7. What will be the output after the following program is executed.

```
#include<iostream.h>
```

```
main()
```

```
{
```

```
int m=66,n;
```

```
n=++m;
```

```
n=m++;
```

```
cout<<m<< “,”<<n<<endl;  
return 0;  
}
```

**a)** 68, 67

**b)** 67, 68

**c)** 67,67

**d)** 60,67

**8.** Which of the following is the scope resolution operator

**a)** ;

**b)** \*

**c)** ->

**d)** None of the above

**9.** C++ has the following tokens

**a)** Keywords

**b)** Identifiers

**c)** Literals

**d)** Objects

Which one from above is not a token.

**10.** A sequence of digits starting with 0 (digit zero)

**a)** Octal

**b)** Decimal

**c)** Hexadecimal

**d)** Binary

**11.** Which of the following is known as insertion operator?

**a)** <<

**b)** >>

**c)** <

**d) >**

**12.** Which of the following is built in type in C++?

- a) int**
- b) float**
- c) double**
- d) All**

**13.** Which of the following is right declaration?

- a) Enum color {red, blue};**
- b) enum color (red, blue);**
- c) enum color {red = 5, blue};**
- d) enum color {red, blue = 0};**

**14.** Which of the following is pointer to member operator?

- a) ->\***
- b) .\***
- c) Both**
- d) None of the above**

**15.** The declaration for the object cin, cout, << and >> is in \_\_\_\_\_ header file.

- a) stdio.h**
- b) iostream.h**
- c) fstream.h**
- d) conio.h**

**16. main()**

```
{  
int x=20, y=30, z=10, i;  
i=x<y<z;  
cout<<i;  
}
```

**a)** 1

**b)** 20

**c)** 30

**d)** 40

## Predict Output

**1.** What is printed by the following statements?

```
cout << "My dog's name is" << " Maggie ";
```

```
cout << "I also have a cat named Senatra" << endl;
```

```
cout << ".:";
```

**2.** What is printed by the following statements?

```
cout << "My dog's name is " << " Maggie" << endl;
```

```
cout << "I also have a cat named Senatra" << endl;
```

**3.** Explain the following expressions

**a)**  $a + b \neq c$

$(a+b) \text{ NOT} = c$

**b)**  $x < 20 \ \&\& x \geq 10$

$x < 20 \text{ AND } x \geq 10 \text{ i.e } 10 < x < 20$

**c)**  $\text{sum} += i;$

$\text{sum} = \text{sum} + i$

**4.** Evaluate the expressions

**a)** Evaluate the expression  $(x > 5 \ \&\& !(x < 9) \ || \ x \leq 14)$  for  $x = 10$

For  $x = 10$  the expression will be true - (T  $\&\&$  T  $\|$  T)

**b)** Evaluate the expression  $!((a \parallel b) \ \&\& (b \&\& b))$  for  $a = 1$  and  $b = 0$

For  $a = 1$  and  $b = 0$ , the expression will be true - Not ((T or T) and F) = NOT(T and F) = NOT(F) = T.

## Programming Exercises

**1.** Write a program for print “ET&T computer education and training center” five times

using cout.

2. Declare two enumerator type named day and month as given in the chapter and get input day number and month number from the user and give output day name and month name using the enumerators.
3. Write a C++ program to convert ASCII value to its equivalent character.
4. Write an interactive program that users inputs a name from the user in the format of:

last, first middle

The program should then output the name as follows:

last first middle

5. Write a program that receives radius and height as inputs and calculates

$$\frac{1}{3} * \pi * \text{Radius}^2 * \text{Height}$$

6. Write a program to print “Hello World” on the screen.
7. Write a program to Program to read one number and display it. (With messages)
8. Write a program that will obtain the length and width of a rectangle from user and calculate its area, perimeter and diagonal.
9. The straight line method of computing the yearly depreciation of the value of an item is given by Depreciation= (Purchase Price-Salvage Value)/ Years of Service. Write a program to determine the salvage value of an Item when the purchase price, years of service and the annual depreciation are given
10. Area of a triangle is given by the formula

$$A = \sqrt{S(S-a)(S-b)(S-c)}$$

Where a,b and c are sides of the triangle and  $2S=a+b+c$ . Write a program to compute the area of the triangle given the values of a,b and c.

11. Write a program to Convert temperature in Celsius to Fahrenheit / Fahrenheit to Celsius.

$$F = \frac{9}{5} * C + 32$$

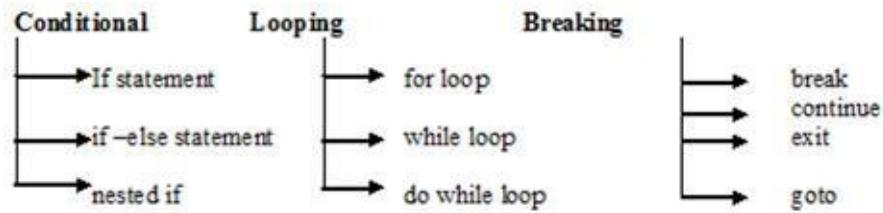
12. Write a program to find the sum of the digits of a 3-digit integer constant.
13. Write a program to interchange the values of two variables.

- 14.** Write a program to Assign value of one variable to another using post and pre increment operator and print the results.
- 15.** Write a program to Read the price of item in decimal form e.g. 12.50. Separate rupees and paisa from the given value e.g. 12 rupees and 50 paisa.
- 16.** Write a program in C++ which read name, age of student using class and display it.
- 17.** Implement a C++ program using scope resolution operator. (The program should declare three variable with same name, one should be global and the rest two should be declared in two different blocks.)



## CHAPTER:3 Control structure

Control structure determines the flow of the program based on the conditions or results of some expression. We already have studied all the control structures in C in depth so here I am just giving overview in C++:



## If statement

The if statement provides a selection control structure to execute a section of code if and only if an explicit run-time condition is met. The condition is an expression which evaluates to a boolean value, that is, either true or false.

### Syntax:

```
if ( <expression> )
{
    Statement
}
```

### Semantics when using if

1. The if statement provides selection control.
2. The expression is evaluated first.
3. If the expression evaluates to true, the statement part of the if statement is executed.
4. If the expression evaluates to false, execution continues with the next statement after the if statement.
5. A boolean false, an arithmetic 0, or a null pointer are all interpreted as false.
6. A boolean true, an arithmetic expression not equal to 0, or a non-null pointer are all interpreted as true.

### Example:

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int a ;
    cout<<“enter the no”;
    cin>>a;
    if(a%2==0)
```

```
cout<<“it is even no.”;  
getch();  
}
```

**Output:**

enter the no12  
it is even no.

## If –else

In this statement, if the expression evaluated to true, the statement or the body of if statement is executed, otherwise the body of if statement is skipped and the body of else statement is executed.

### Syntax

```
if (condition)
```

```
{
```

```
    statement1;
```

```
}
```

```
else
```

```
{
```

```
    statement2;
```

```
}
```

### Example:

```
# include<iostream.h>
```

```
#include<conio.h>
```

```
void main()
```

```
{
```

```
clrscr();
```

```
int n;
```

```
cout<<“enter the no”;
```

```
cin>>n;
```

```
if (n%2==0)
```

```
cout<<“it is even no”;
```

```
else
```

```
cout<<“it is odd no”;
```

```
getch();
```

}

**Output:**

enter the no13

it is odd no

## Switch

It provide multiple branch selection statement .if –else provide only two choices for selection and switch statement provide multiple choice for selection.

### Syntax:

```
switch(expression)
```

```
{
```

```
case exp1:
```

```
First case body;
```

```
break;
```

```
case exp2:
```

```
Second case of body;
```

```
break;
```

```
default:
```

```
Default case body;
```

```
}
```

### Example:

```
#include<iostream.h>
```

```
#include<conio.h>
```

```
int main()
```

```
{
```

```
int a ;
```

```
clrscr();
```

```
cout<<“enter the no”;
```

```
cin>>a;
```

```
switch(a)
```

```
{
```

```
case 1:
```

```
cout<<“sunday\n”;  
break;  
case 2:  
cout<<“monday\n”;  
break;  
case 3:  
cout<<“tuesday\n”;  
break;  
case 4:  
cout<<“wednesday\n”;  
break;  
case 5:  
cout<<“thrusday\n”;  
break;  
case 6:  
cout<<“friday\n”;  
break;  
case 7:  
cout<<“satday\n”;  
break;  
default:  
cout<<“wrong option”;  
}  
getch();  
return 0;  
}
```

**Output:**

enter the no 4

wednesday

## For loop

In this, first the expression or the variable is initialized and then condition is checked. if the condition is false ,the loop terminates

**Syntax:-**

```
for (initillization;condition;increment)
```

**Example: program to print the no from 1 to 100.**

```
#include<iostream.h>
# include<conio.h>
void main ()
{
    int i;
    for(i=1;i<=100;i++)
        cout<<i<<"\n";
    getch ();
}
```

## While loop

This loop is an entry controlled loop and is used when the number of iteration to be performed are known in advance. The statement in the loop is executed if the test condition is true and execution continues as long as it remains true.

### Syntax:-

```
initialization;  
while (condition)  
{  
    statement;  
    increment;  
}
```

### Example:A program to print a table.

```
#include<iostream.h>  
#include<conio.h>  
void main ()  
{  
    int n ,i;  
    cout<<"enter the no whose table is to be printed “;  
    cin>>n;  
    i=1;  
    while (i<=10)  
    {  
        cout<<n<<"x”<<i<<”=”<<nxi<<”\n”;  
        i++;  
    }  
    getch();  
}
```

## Do-while

It is bottom controlled loop. This that a do-while loop always execute at least once.

### Syntax:

```
initillization  
do  
{  
statement ;  
increement;  
}while(condition);
```

### Example: A program to print the multiplication table of number.

```
# include<iostream.h>  
#include<conio.h>  
int main()  
{  
int n,i;  
clrscr();  
cout<<“enter the no. whose table is to be printed:”;  
cin>>n;  
i=1;  
do  
{  
cout<<n<<“x”<<i<<”=”<<n*i<<”\n”;  
}while(i++<10);  
getch();  
return 0;  
}
```

**Output:**

enter the no. whose table is to be printed:5

5x1=5

5x2=10

5x3=15

5x4=20

5x5=25

5x6=30

5x7=35

5x8=40

5x9=45

5x10=50

## Break statement

The term break means breaking out of a block of code. The break statement has two use, you can use it to terminate a case in the switch statement, and you can also use it to force immediate termination of loop, bypassing the normal loop condition test.

### Example 1:-

```
#include<iostream.h>
#include<conio.h>
int main()
{
    int a ;
    clrscr();
    cout<<“enter the no”;
    cin>>a;
    switch(a)
    {
        case 1:
        cout<<“sunday\n”;
        break;
        case 2:
        cout<<“monday\n”;
        break;
        case 3:
        cout<<“tuesday\n”;
        break;
        case 4:
        cout<<“wednesday\n”;
        break;
    }
}
```

```
case 5:  
cout<<“thrusday\n”;  
break;  
case 6:  
cout<<“friday\n”;  
break;  
case 7:  
cout<<“satday\n”;  
break;  
default:  
cout<<“wrong option”;  
}  
getch();  
return 0;  
}
```

### **Output:**

```
enter the no 4  
wednesday
```

### **Example 2:**

```
# include<iostream.h>  
#include<conio.h>  
int main()  
{  
int i,j;  
clrscr();  
for(i=1;i<3;i++)  
{
```

```
for(j=1;j<3;j++)  
{  
    cout<<i<<" "<<j<<" ";  
    if(i==j)  
        break;  
}  
cout<<"\n";  
}  
getch();  
return 0;  
}
```

**Output:**

1 1 1 2  
2 1 2 2

## Continue

The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are going to skip the number 5 in our countdown:

### Example

```
#include <iostream.h>
#include<conio.h>
void main ()
{
for (int n=10; n>0; n--)
{
if (n==5) continue;
cout << n << ", ";
cout << "fire!\n";
getch();
}
```

### Output-

10, 9, 8, 7, 6, 4, 3, 2, 1, fire!

### Example 2:

```
# include<iostream.h>
#include<conio.h>
int main()
{
int i,j;
clrscr();
```

```
for(i=1;i<3;i++)  
{  
for(j=1;j<3;j++)  
{  
cout<<i<<" "<<j<<" ";  
if(i==j)  
continue;  
}  
cout<<"\n";  
}  
getch();  
return 0;  
}
```

**Output:**

```
1 1  
2 1 2 2
```

## Exit() Function

Exit is a function defined in the stdlib library means (stdlib.h).

The purpose of exit is to terminate the current program with a specific exit code. Its prototype is:

```
exit (exitcode);
```

The exitcode is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means that the program finished normally and any other value means that some error or unexpected results happened

### Example

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
void main ()
{
    int n;
    cout<<“enter the no”;
    cin>>n;
    if (n%2==0)
        cout<<“it is even no”;
    else
    {
        cout<<“it is odd no”;
        exit(0);
    }
    getch();
}
```

### Output:

enter the no 23

it is odd no

## Goto statement

Goto statement allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations. The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:).

Generally speaking, this instruction has no concrete use in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using goto:

### // goto loop example

```
#include <iostream.h>
```

```
int main ()
```

```
{
```

```
int n=10;
```

```
loop:
```

```
cout << n << ", ";
```

```
n--;
```

```
if (n>0) goto loop;
```

```
cout << "fire!\n";
```

```
return 0;
```

```
}
```

### Output:

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, fire!
```

## Exercise

### Short Questions

1. What happens when you forget to include the Break statements in a switch statement?

2. How many times the inner loop is executed in the following nested loop?

```
for (int x = 1; x <= 10; x++)
```

```
for (int y=1; y <= 10; y++)
```

```
for (int z = 1; z <=10; z++)
```

```
cout << x+y+z;
```

3. Which looping statement would you choose for a problem in which the decision to repeat an event depends on an event, and the event cannot occur until the process is executed at least once?

4. Which logical operator (op) is defined by the following table? (T and F denote TRUE and FALSE.)

P	Q	P op Q
<hr/>		
T	T	T
T	F	F
F	T	F
F	F	F

5. Mention the data types that are supported by C++ but not by c.

6. Distinguish between l value and r-value of a variable. Given an example.

7. List out the different operates that are available in C + + along with its precedence and associativity.

8. What is types casting? What is the value of result in each of cout statement? Give reason for the difference in value.

```
double result;
```

```
int whole-number=7;
```

```
result = whole- number/2;
```

```
cout << "\n the result of this division is " << result;
```

```
result = (double) whole – number/2;
```

```
cout << "\n the result of this division is ' << result;
```

**9.** Write the syntax of while loop and Do-while loop. Show the difference between the two loops with examples.

**10.** The following statement is written to check the condition of a variable x to see if it was either 0 or 1.

If (x == 0 || 1)

When the program was run the statement was never found to be false. Why? Give reason. Also give the correct statement.

**11.** Mention the syntax of all the loops available in C ++ and their utility under different contexts. Explain with program examples.

**12.** Suggest one C ++ program to show how if-else statement and conditional operator work identically.

**13.** We do not see the values of 17 and 25 a and b, when we run the below program.

### True/ False

**1.** Any Switch statement that has a break statement at the end of each case alternative can be replaced by a nested If-Then-Else structure.

**2.** The statement

```
switch (n){  
    case 8 : alpha++; break;  
    case 3 : beta++; break;  
    default: gamma++; break;  
}
```

is equivalent to the following statement.

```
if (n == 8)  
    alpha++;  
else if (n == 3)  
    beta++;  
else  
    gamma++;
```

3. The body of a Do-While loop will always execute at least once, whereas the body of a For loop may never execute.
4. If a loop containing a break statement is nested within a Switch statement, execution of that break statement causes an exit from the loop but not from the Switch statement.
5. The code segment

```
cout << "Do you wish to continue? ";  
cin >> response;  
while (response != 'Y' && response != 'N')  
{  
    cout << "Do you wish to continue? ";  
    cin >> response;  
}
```

is equivalent to the following code segment.

```
do{  
    cout << "Do you wish to continue? ";  
    cin >> response;  
} while (response != 'Y' && response != 'N');
```

6. In the following code fragment, the output is ABC.

```
cout << 'A';  
for (count = 1; count <= 3; count++);  
cout << 'B';  
cout << 'C';
```

7. The output of the following code fragment if the input value is 'G' is 5.

```
cin >> inputChar;  
switch (inputChar)  
{  
    case 'A': cout << 1; break;
```

```
case 'Q' : cout << 2; break;  
case 'G' :  
case 'M' : cout << 3; break;  
default : cout << 4;  
}
```

**8.** The output of the following code fragment if the input value is 4 is 19.

```
int num;  
int alpha = 10;  
cin >> num;  
switch (num)  
{  
case 3 : alpha++;  
case 4 : alpha = alpha + 2;  
case 8 : alpha = alpha + 3;  
default : alpha = alpha + 4;  
}  
cout << alpha << endl;
```

**9.** An equivalent for loop for the following code

```
count = -5;  
while (count <= 15)  
{  
sum = sum + count;  
count++;  
}  
is  
for (count = -5; count <= 15; count++)  
sum = sum + count;
```

**10.** The output of the following code fragment is 32.

```
n = 2;  
for (loopCount = 1; loopCount <= 3; loopCount++)  
do  
n = 2 * n;  
while (n <= 4);  
cout << n << endl;
```

**11.** After the execution of the following code, alpha has 33.

```
char ch = 'D';  
int alpha = 3;  
switch (ch)  
{  
case 'A' : alpha = alpha + 10; break;  
case 'B' : alpha = alpha + 20; break;  
case 'C' : alpha = alpha + 30;  
}
```

**12.** A break statement is not allowed in a For loop, but a Continue statement is.

**13.** The code

```
testScore >=90 && testScore <= 100
```

is a C++ logical expression that is true if the variable testScore is greater than or equal to 90 and less than or equal to 100:

**14.** The code

```
x != 5 && y != 5
```

is a C++ logical expression that is true if the variable testScore is greater than or equal to 90 and less than or equal to 100:

**15.** When floating-point numbers are involved in calculations, errors occur in the rightmost decimal places because the representation of a real number in a computer is not always exact.

**16.** The equivalent expression for !(one == two && three > four) is

one != two || three <= four

**17.** An example of a logical (Boolean) expression is an arithmetic expression followed by a relational operator followed by an arithmetic expression.

**18.** Syntactically, the only expressions that can be assigned to Boolean variables are the literal values true and false.

**19.** If ch1 contains the value ‘C’ and ch2 contains the value ‘K’, the value of the C++ expression

ch1 <= ch2 is true.

**20.** If P and Q are logical expressions, the expression P AND Q is TRUE if either P or Q is TRUE or both are TRUE.

**21.** The expression !(n < 5) is equivalent to the expression n > 5.

**22.** According to DeMorgan’s Law, the expression

!(x <= y || s > t)

is equivalent to

x <= y && s > t

**23.** The statement

```
if (grade == 'A' || grade == 'B' || grade == 'C')
```

```
cout << "Fail";
```

```
else
```

```
cout << "Pass";
```

prints Pass if grade is ‘A’, ‘B’, or ‘C’ and prints Fail otherwise.

**24.** If a C++ If statement begins with

```
if (age = 30)
```

the If condition is an assignment expression, not a relational expression.

**25.** The code segment

```
if (speed <= 40)
```

```
cout << "Too slow";
```

```
if (speed > 40 && speed <= 55)
    cout << "Good speed";
if (speed > 55)
    cout << "Too fast";
```

could be written equivalently as

```
if (speed <= 40)
    cout << "Too slow";
else if (speed <= 55)
    cout << "Good speed";
else
    cout << "Too fast";
```

**26.** If DeMorgan's Law is used to negate the expression

$(i < j) \&\& (k == l)$  then the result is  $(i >= j) \parallel (k != l)$

**27.** Given a Boolean variable isEmpty, then the following statement

`isEmpty = !isEmpty;`

is a valid C++ assignment statement?

**28.** In the following code, if the value of the input variable *angle* is 10, after the execution of the code the value of the *angle* variable is 15.

```
cin >> angle;
if (angle > 5)
    angle = angle + 5;
else if (angle > 2)
    angle = angle + 10;
```

**29.** In the following code, if the value of the input variable *angle* is 10, after the execution of the code the value of the *angle* variable is 15.

```
cin >> angle;
if (angle > 5)
```

```
angle = angle + 5;  
if (angle > 2)  
angle = angle + 10;
```

30. In the following code, if the value of the input variable *angle* is 10, after the execution of the code the value of the *angle* variable is 15.

```
cin >> angle;  
if (angle > 5)  
angle = angle + 5;  
else if (angle > 2)  
angle = angle + 10;  
else  
angle = angle + 15;
```

31. In the following code, if the value of the input variable *angle* is 0, after the execution of the code the value of the *angle* variable is 15.

```
cin >> angle;  
if (angle > 5)  
angle = angle + 5;  
else if (angle > 2)  
angle = angle + 10;
```

32. Consider the following If statement, which is syntactically correct but uses poor style and indentation:

```
if (x >= y) if (y > 0) x = x * y; else if (y < 4) x = x - y;
```

Assume that *x* and *y* are **int** variables containing the values 3 and 9, respectively, before execution of the above statement. After execution of the statement, *x* contains 27. True or False? The output of the following code fragment if the input value is 20 is “Larry Curly”.

```
cin >> someInt;  
if (someInt > 30)
```

```
cout << "Moe ";
cout << "Larry ";
cout << "Curly";
```

**33.** Assuming alpha and beta are int variables, the output of the following code is "There".

```
alpha = 3;
beta = 2;
if (alpha < 2)
if (beta == 3)
cout << "Hello";
else
cout << "There";
```

**34.** Given the following code segment

```
name1 = "Maryanne";
name2 = "Mary";
```

the value of the relational expression string1 <= string2 is False.

**35.** Assuming no input errors, an execution of the >> operator leaves the reading marker at the character immediately following the last data item read.

**36.** When working at the keyboard, the user generates a newline character by pressing the Enter or Return key.

**37.** If the reading marker is in the middle of an input line of 25 characters, execution of the statement

```
cin.ignore(500, '\n');
```

leaves the reading marker at the character following the next newline character.

**38.** The >> operator skips leading whitespace characters when looking for the next data value in the input stream.

**39.** Reading input from a file is considered interactive I/O because the CPU is required to interact with a disk drive or other device.

**40.** Using an editor program to edit a file requires interactive I/O.

**41.** In a functional decomposition, a concrete step is one in which some of the implementation details remain unspecified.

**42.** Two methods to store a value into a variable is by executing a) an input statement b) an assignment statement.

**43.** When an integer data value is read into a float variable, the value is first converted into floating-point form.

**44.** Assume the data in the following two lines is input:

A B

CDE

‘C’ is read into ch3 by the following code (All variables are of type char.)

```
cin >> ch1 >> ch2 >> ch3;
```

**45.** Assume the data in the following two lines is input:

A B

CDE

‘B’ is read into ch3 by the following code

```
cin.get(ch1);  
cin.get(ch2);  
cin.get(ch3);
```

**46.** Given the two lines of input data

ABC

DEF

A blank character is read into ch by the following code? (str is of type string, and ch is of type char.)

```
cin >> str;  
cin.get(ch);
```

**47.** Given the two lines of input data

ABC

## DEF

A blank character is read into ch by the following code? (str is of type string, and ch is of type char.)

```
getline(cin, str);  
cin.get(ch);
```

**48.** The C++ standard library defines a data type named ifstream that represents a stream of characters coming from an input file.

**49.** In a C++ floating-point constant, a decimal point is not required if exponential (E) notation is used.

**50.** In a C++ expression, all additions are performed before any subtraction.

**51.** In a C++ expression without parentheses, all operations are performed in order from left to right.

**52.** If someFloat is a variable of type float, the statement

```
someFloat = 395;
```

causes someFloat to contain an integer rather than floating-point value.

**53.** In C++, the expression  $(a + b / c) / 2$  is implicitly parenthesized as  $((a + b) / c) / 2$ .

**54.** If the int variable someInt contains the value 26, the statement

```
cout << "someInt";
```

outputs the value 26.

**55.** The setprecision manipulator is used to increase or decrease the precision of floating-point numbers that are stored in the computer's memory unit.

**56.** The setw manipulator is used only for formatting numeric values and strings, not char data.

**57.** If myString is a string variable and the statement

```
myString = "Harry";
```

is executed, then the value of the expression myString.find('r') is 2.

**58.** The setw manipulator is used only for formatting numeric values and strings, not char data.

**59.** Among the C++ operators +, -, \*, /, and %, “+ and –“ have the lowest precedence.

**60.** The value of the C++ expression  $3 / 4 * 5$  is 5.

**61.** Assuming all variables are of type float, the C++ expression for  $(a + b)c / d + e$  is:  $(a + b)*c / (d + e)$ .

**62.** The value of the C++ expression  $11 + 22 \% 4$  is 1.

**63.** If the int variables int1 and int2 contain the values 4 and 5, respectively, then the value of the expression  $\text{float}(int1 / int2)$  is 1.0.

**64.** If the int variables int1 and int2 contain the values 5 and 4, respectively, then the value of the expression  $\text{float}(int1 / int2)$  is 1.0.

**65.** The expression  
`float((int1 + int2 + int3) / 3)`  
does not correctly compute the mathematical average of the int variables int1, int2, and int3.

**66.** If testScore is an int variable containing the value 78, the following program fragment  
`cout << "1234567890" << endl << "Score:" << setw(4) << testScore << endl;`  
outputs the following output:  
1234567890  
Score: 78

**67.** `plant = "Dandelion";`  
is executed, then the value of the expression `plant.find('d')` is 3.

**68.** If name is a string variable, the output of the following code  
`name = "Marian";  
cout << "Name: " << name.substr(1, 3) + "sta";`  
is Name: arista

**69.** The expression `int(someFloat)` is an example of a(n) cast operation.

**70.** In the statement  
`y = SomeFunc(3);`  
the number 3 is known as the function’s actual parameter or argument.

## Multiple Choice Questions

1. What will be the output of the following program if your integer input is 1234567

```
main()
{
    long m,d,n=0;
    cout<<"enter a positive integer";
    cin>>m;
    while(m>0)
    {
        d=m%10;
        m/=10;
        n=10*n+d;
    }
    cout<<n<<endl;
}
```

- a)** 123564
- b)** 456123
- c)** 654321
- d)** 123456

2. When a break statement is encountered in a switch statement, program execution jumps to the

- a)** Next line
- b)** Outside the body of switch statement.
- c)** Starting of the switch statement.
- d)** Jumps to the main statement.

3. What is the value of sum when you run the following piece of code? Give reason.

```
Sum = 0;
```

```
For (I = 1 ; I <= 5 ; I + + );  
Sum = sum + I;  
cout << " value of sum = " << sum ;
```

Question:-

```
int I, j;  
string s;  
cin >> I >> j >> s >> s >> I;  
cout << I << " " << j << " " << s << " " << I;
```

Referring to the sample code above, what is the displayed output if the input string given were: "5 10 Sample Word 15 20"?

- a)** 5 10 Sample Word 15 20
- b)** 15 20 Sample Word 15
- c)** 5 10 Sample Word 15
- d)** 15 10 Word 15
- e)** 15 20 Sample Word 20

4. Which type of conditional expression is used when a developer wants to execute a statement only once when a given condition is met?

- a)** switch-case
- b)** for
- c)** if
- d)** do-while
- e)** while

5. which one from the given below is the output of this code

```
main()  
{  
int f=1, i=2;  
do
```

```
{  
f*=i;  
}while(++i<5);  
cout<<f;  
}
```

**a)** 20

**b)** 24

**c)** 6

**d)** 22

6. which one from the given below is the output of this code

```
main()  
{  
int f=1, i=2;  
while (++i<5);  
{  
f*=i;  
}  
cout<<f;  
}
```

**a)** 12

**b)** 6

**c)** 24

**d)** 20

## Predict Output

1. What is the output of the following code segment?

```
n = 0;
```

```
while (n < 5)
```

```
{
```

```
    n++;
```

```
    cout << n << ' ';
```

```
}
```

2. What is the output of the following program segments?

a) int x = 1;

```
while ( x <= 10)
```

```
{
```

```
    cout << "Hello\n";
```

```
    x = x + 1;
```

```
}
```

```
cout << x << endl; //What's the value of x ____
```

b) i = 0;

```
while ( i < 10)
```

```
{
```

```
    if (i % 2 == 1)
```

```
        cout << i << " ";
```

```
    ++i;
```

```
}
```

c) num = 0; // Initialization

```
for ( ; num < 5; ) { // Condition
```

```
    cout << "Num is " << num << endl;
```

```
    num++; // update
```

```
}
```

3. Given the input data

5 10 20 -1

what is the output of the following code fragment? (All variables are of type int.)

```
sum = 0;  
cin >> number;  
while (number != -1)  
{  
    sum = sum + number;  
    cin >> number;  
}  
cout << sum << endl;
```

4. What is the output of the following code?

```
int counter, sum; // declare loop counter and sum  
counter = 1; // initialize the loop counter  
sum = 0; // initialize the sum  
while ( counter < 100){ // iterate the loop 99 times  
    sum = sum + counter; // add the value of counter to the existing  
    ++counter; // increment the counter  
}  
cout << "The sum is " << sum << endl; // Display the sum
```

5. Explanation:

The following is an **end-of-file-controlled** ( generated by holding Ctrl key and pressing on D key) while loop code segment. The loop stops when Ctrl+D key are pressed. Read the comments carefully to understand what's happening and then answer the question following this code.

```
float number; // storage for the number read  
cin >> number; // read the first number (priming read)  
while ( cin ) { // loop until failed data stream read  
    // This happens when you typed ^D and hit return
```

```
cout << "The number read is " << number << endl; // display the
number read
cin >> number; // read next number
}
```

6. Given the following input data followed by control\_d (which is the end of file character):

1 2 3 4

what is the output of the following code fragment? (All variables are of type int.)

```
sum = 0;
cin >> number;
while (cin)
{
    sum = sum + number;
    cin >> number;
}
cout << sum << endl;
```

7. What will be the value of result after execution of the following code segment?

```
a = 5; result=10;
while ( a++ < 10 ) { result ++; }
```

8. Given the following code segment, what is the value of the variable “done” after while loop exits?

```
sum = 0;
done = false;
while (!done)
{
    cin >> number;
    if(number < 0)
```

```
done = true;  
else  
sum = sum + number;  
}
```

**9.** Convert the following while loop to a for loop

```
a = 8; sum=0;  
while ( a > 0 )  
{ sum = sum + a; a= a - 1; }
```

**10.** What is the output of the following code segments?

**a)** for (int i=1; i < 10; i=i+2)

```
cout << i << endl;
```

**b)** for (int n=10; n>0; n—)

```
{  
cout << n << endl;  
}
```

**c)** for (k=2; k<=1024; k = k \* 2)

```
cout << k << endl;
```

**d)** for (k=2; k<=1024; k = k \* 2);

```
cout << k << endl;
```

**e)** for (int k=-5; k<0; k++)

```
cout<< k + 2 << endl;
```

```
cout<<“HELLO”;
```

**f)** for (len=9; len>=1; len—)

```
{  
for (num=1; num<=len; num++)  
cout << num;  
cout << endl;
```

}

**11.** What is the output of the following code

```
count =1;  
for( ; ; count++)  
if ( count < 3 )  
cout << count;  
else  
break;
```

**12.** What is the output of the following code?

```
for (int i=0 ; i < 3; i++) {  
for (int j=0; j < 4; j++)  
cout << "*" ;  
cout << endl;  
}
```

**13.** What is the output of the following code?

```
i = 0;  
while (i < 3)  
{  
j = 1;  
while (j < 3)  
{  
cout << "i = " << i << " j = " << j;  
j++;  
}  
cout << endl;  
i++;  
}
```

14. What is the output of the following code segment if wood contains ‘O’?

```
switch(wood){  
    case 'P': cout << "Pine" ;  
    case 'F': cout << "Fit";  
    case 'C': cout << "Cedar";  
    case 'O': cout << "Oak";  
    case 'M': cout << "Maple";  
    default : cout << "Error";  
}
```

15. What is the output of the following program?

```
#include <iostream>  
using namespace std;  
int integer1 = 1;  
int integer2 = 2;  
int integer3 = 3;  
int main()  
{  
    int integer1 = -1;  
    int integer2 = -2;  
    {  
        int integer1 = 10;  
        cout << "integer1 == " << integer1 << "\n";  
        cout << "integer2 == " << integer2 << "\n";  
        cout << "integer3 == " << integer3 << "\n";  
    }  
    cout << "integer1 == " << integer1 << "\n";  
    cout << "integer2 == " << integer2 << "\n";  
    cout << "integer3 == " << integer3 << "\n";  
    return 0;  
}
```

**16.** What is the output of the following code?

```
#include <iostream>

using namespace std;

int main(void) {

    cout << "First Loop" << endl << "_____" << endl;

    int x = 3;

    cout << "Initial value of x (before loop): " << x << endl << endl;

    for (x = 1; x < 42; x = x * 2)

        cout << "Inside loop, value of x: " << x << endl;

    cout << endl << "Final value of x (after loop): " << x << endl <<

    endl;

    cout << "Second Loop" << endl << "_____" << endl;

    for (int i = 0; i < 3; i++)

        for (int j = 0; j < 3; j += 1)

            for (int k = 0; k < 3; k = k + 1)

                cout << "[" << i << " " << j << " " << k << " ]" << endl;

    return 0;

}
```

**17.** What is the output of the following program segments?

**a)** int x = 1;  
do{  
 cout << "Hello\n";  
 x = x + 1;  
}while (x <= 10);  
cout << x << endl; //What's the value of x \_\_\_\_

**b)** i = 0;  
do {

```
if (i % 2 == 1) {  
    cout << i << " "  
}  
++i;  
} while (i < 10);
```

c) num = 0; // Initialization

```
for (; num < 5; ) { // Condition  
    cout << "Num is" << num << endl;  
    num++; // update  
}
```

**18.** What is the output of the following code

```
count = 1;  
for( ; ; count++)  
if ( count < 3 )  
    cout << count;  
else  
    break;
```

**19.** What is the output of the following C++ code fragment? (Be careful here.)

```
int1 = 120;  
cin >> int2; // Assume user types 30  
if ((int1 > 100) && (int2 = 50))  
    int3 = int1 + int2;  
else  
    int3 = int1 - int2;  
cout << int1 << ' ' << int2 << ' ' << int3;
```

**20.** Given the following input data

123.456 A 789

what value is read into inputChar by the following code?

```
cin >> alpha >> inputChar >> beta;
```

**21.** Given that x is a float variable containing the value 84.7 and num is an int variable, what will num contain after execution of the following statement:

```
num = x + 2;
```

**22.** What is the output of the following program fragment?

```
age = 29;
```

```
cout << "Are you" << age << "years old?" << endl;
```

**23.** The output of the following program fragment? (x is a float variable.)

```
x = 25.6284; cout << "##" << setw(6) << setprecision(1) << x << endl;
```

**24.** Explain the following expressions

```
x < 20 && x >= 10
```

```
sum +=i;
```

**25.** Evaluate the expression  $(x > 5 \&\& !(x < 9) \parallel x \leq 14)$  for  $x = 10$

**26.** Evaluate the expression  $!((a \parallel b) \&\& (b \&\& b))$  for  $a = 1$  and  $b = 0$

**27.** Consider the following program segment. What will be the output of the program for code=1.

```
if(code !=1)  
if(code !=2)  
if (code !=3)  
printf("Yellow \n");  
else  
printf("White \n");  
else  
printf("Green \n");  
else
```

```
printf("Red \n");
```

**28.** What will be the output of the program for i=9 and n=10.

```
void main()
{
    int i=?, n=?, sum=0;
    do{
        sum = sum + i;
        i++;
    }while(i<n);
    cout<<"Sum = " << sum;
}
```

**29.** A considerable amount of time can be saved by evaluation of complex conditions. As an illustration of the first benefit, consider the expression below

```
if (n !=0 && x<1/n)
{
    y=1/n-x;
    cout << y;
}
```

Explain what benefit we get from the expression above.

**30.** Explain the following program segment

```
for(; ;)
{
    cout<<"";
}
```

**31.** Assuming that i and k are integer variables, describe the output produced by the following program segments

```
int k=4, i=-1;
```

```

while(i<=k)
{
    i+=2;
    k--;
    cout << (i+k) << "\n";
}

```

## Find Error in the Code

1. What is wrong with the following code?

```

while ( n <= 100 )
    sum += n* n;

```

## Programming Exercises

1. Write C++ program to sum the following numbers [notice it is multiplication of 5]:  

$$[5, 10, 15, 20, 25, 30, \dots, 490, 495, 500]$$
2. Write a program that reads and sums until it has read ten data values or until a negative value is read, whichever comes first.
3. Write a value returning function that accepts two int parameters, base and exponent, and returns the value of base raised to the exponent power. Use a For loop in your solution.
4. Write a program that reads n numbers from the terminal and prints the average of those numbers (use while or for loop).
5. Write a program that calculates  $n!$ . Hint:  $n! = 1 * 2 * 3 * \dots * n$  . For example if n is 4, the output will be 24.
6. Write a program that reads n integers and prints the number of positive and number of negative integers. If a value is zero, it should NOT be counted.
7. Write a program that reads n integers and prints the number of positive and number of negative integers. If a value is zero, it should NOT be counted.
8. Write a program that calculates  $n!$  [Note  $n! = 1 * 2 * 3 * \dots * (n-1) * n$  ]
9. Write a program that calculates  $1+2+3 + \dots + n$  for a given n.
10. Write a nested loop code segment that produces this output:

1

1 2

1 2 3

1 2 3 4

1 2 3 4 5

**11.** Write an algorithm to produce an n times multiplication table ( n less than or equal to 10). For example, if n is equal to four the table should appear as follows:

1 2 3 4

1 1 2 3 4

2 2 4 6 8

3 3 6 9 12

4 4 8 12 16

Convert your algorithm into a program. Use nested for loops and pay attention to formatting your output.

**12.** Write a C++ program to generate Fibonacci series.

**13.** Write a program that reads and sums until it has read ten data values or until a negative value is read, whichever comes first.

**14.** Write a value returning function that accepts two int parameters, base and exponent, and returns the value of base raised to the exponent power. Use a **For loop** in your solution.

**15.** Write a program that receives three test scores and upon calculating the average test score determines the letter grade of a student using **switch** statement:

**a)** If average is > 90 , outputs “Your letter grade is A”

**b)** If average is > 80 but less than 91 , outputs “Your letter grade is B”

**c)** If average is > 70 but less than 81 , outputs “Your letter grade is C”

**d)** If average is > 60 but less than 71 , outputs “Your letter grade is D”

**e)** If average is < 60 , outputs “Your letter grade is F”

**16.** Write a program that determines College students status (freshman, junior, senior or sophomore according to the following description using **switch** statement.

- a)** If the student has less than or equal to 30 credits, his/her status is freshman,
- b)** If the student has less than or equal to 60 credits, but more than 30, his/her status is sophomore.
- c)** If the student has less than 90 credits, but more than 60, his/her status is junior.
- d)** If the student has more than 90 credits, his/her status is senior.

**17.** Write a nested **if-else** statement that will assign a character grade to a percentage mark as follows - 90 or over A, 80-89 B, 70-79 C, 60-69 D, less than 60 F.

**18.** Could a switch statement be used to directly assign the appropriate grade given a percentage mark for question #2? If not how could you do this? Write a statement to carry out the assignment of grades.

**19.** Write a switch statement to assign grades as described in the previous question. Use the fact that mark/10 gives the first digit of the mark.

**20.** Write a program that receives three test scores as input, and upon calculating the average test score determines the letter grade of a student as following:

- a)** If average is  $> 90$  , outputs “Your letter grade is A”
- b)** If average is  $> 80$  but less than 91 , outputs “Your letter grade is B”
- c)** If average is  $> 70$  but less than 81 , outputs “Your letter grade is C”
- d)** If average is  $> 60$  but less than 71 , outputs “Your letter grade is D”
- e)** If average is  $< 60$  , outputs “Your letter grade is F”

**21.** Write a nested **if-else** statement that will assign a character grade to a percentage mark as follows - 90 or over A, 80-89 B, 70-79 C, 60-69 D, less than 60 F.

**22.** Write a program that determines College students’ status (freshman, junior, senior or sophomore) according to the following description:

- a)** If the student has less than or equal to 30 credits, his/her status is freshman,
- b)** If the student has less than or equal to 60 credits, but more than 30, his/her status is sophomore.
- c)** If the student has less than 90 credits, but more than 60, his/her status is junior.

**d)** If the student has more than 90 credits, his/her status is senior.

**23.** Modify the program in 2 such that, it is a function called from main program and you pass the credits as a parameter to it to output the status.

**24.** Write a program that implements the following math function:

$$F(x) = x^3 - 5$$

**25.** Write a program that has a function which converts miles to kilometers ( 1 mile is 1.609 kilometer)

**26.** Write a program which can right justify the output of each the following numbers in a column on the screen. Numbers are: 23.62 46.0 43.46443 100.1 98.98 ( use setprecision and formating)

**27.** Write a program that displays the strings, “Good Morning”, “Sarah”, and “Sunshine” on each line, but centered in fields of 20 characters. Do not use manipulators. Compile and Run and show your output

**28.** Repeat 4, using manipulators to help center your strings

**29.** Write a named string constant made up of your first name and last name with a blank in between. Write the statements to print out the result of applying, “length” and “size” to your named constant object.

**30.** Write a C++ program which reads values for two floats and outputs their sum, product and quotient. Include a sensible input prompt and informative output.

**31.** Write a program to convert currency from UK pounds or sterling to US dollar. Read the quantity of money in pounds and pence, and output the resulting foreign currency in dollar and cents. (There are 100 penny in a pound). Use a `const` to represent the conversion rate, which is \$1.96 to £1 at the time of writing. Be sure to print suitable headings and or labels for the values to be output.

**32.** Modify program 9 such that your program converts US dollars to UK pounds.

**33.** Write a C++ program to calculate average marks scored by a student for 3 subjects.

**34.** Write a C++ program to find the area and perimeter of a circle and rectangle.

**35.** Write a C++ program to swap two numbers.

**36.** Write a C++ program to find largest of three numbers.

**37.** Write a C++ program to find the maximum number among three numbers.

**38.** Write C++ program to read input value & based on its magnitude & sign, it must output one of following messages:

The number is positive and Even - The number is negative and Even

The number is positive and Odd - The number is negative and Odd

**39.** Write C++ program to compute the FACTORIAL of an integer input by the user?

[Factorial of N:  $N! \text{ è } 1 * 2 * 3 * 4 * \dots * N-2 * N-1 * N$ ]

**40.** Write C++ program to compute N rose to power M, where both N & M are integers?

[ $N$  raised to power  $M$ :  $N^M \text{ è } N * N * N * \dots * N$  {Without using  $\text{pow}(N,M)$ }]

**41.** Write C++ program to read an integer and to test it if it is a prime-number or composite-number.

Your program must validate the input number to be greater than 2 before testing?

**42.** Write C++ program to test a quadratic equation if it has roots, using the Discriminant ( $B^2 - 4AC$ ).

Based on that it must calculate and print both roots, or display a message: “Complex-Roots”

**43.** Using literal character strings and cout print out a large letter E as below:

XXXXX

X

X

XXX

X

X

XXXXX

**44.** Write a program to read in four characters and to print them out, each one on a separate line, enclosed in single quotation marks.

**45.** Write a program which prompts the user to enter two integer values and a float value and then prints out the three numbers that are entered with a suitable message.

**46.** Write a program to read marks from keyboard and your program should display

equivalent grade according to following table.

Marks Grade

100-80 Distinction

60-79 First class

35-59 Second class

0-34 Fail

**47.** Write a program for Solution of quadratic equation

**48.** Make Simple Calculator using switch and if ...else if

**49.** Find maximum and minimum of three numbers using ternary operator.

**50.** Check if the given year is leap year or not. ( Use full condition for leap year )

**51.** Convert the case of a given character ( i.e. upper to lower & vice versa ) ( Use getchar & putchar )

**52.** Find maximum / minimum of 3 numbers using if and also using the Ternary operator.

**53.** Write a program to find out Net salary, HRA, DA, PI of employee according to Basic salary. Do using if and also using switch statements.

Net salary=Basic+HRA+DA+PI

BASIC SALARY	HRA	DA	PI
$\geq 10,000$	20%	15%	10%
$\geq 5,000 \text{ and } < 10,000$	15%	10%	5%
$< 5000$	10%	5%	-

**54.** Write a program that will allow computer to be used as an ordinary calculator. Consider only common arithmetic operations.(+, -, \*, / ) The program should display a menu showing the different options available. Do using if and also using switch statements.

**55.** The cost of one type of mobile service is Rs. 250 plus Rs. 1.25 for Each call made over and above 100 calls. Write a program to read Customer codes and calls made and print the bill for each customer.

**56.** Write a program to Print 1st N natural numbers & calculate their sum & avg.

**57.** Write a program to Print squares / cubes of 1st N natural numbers & calculate their sum & avg.

**58.** Write a program to Print all numbers between  $-n$  &  $+n$ .

**59.** Write a program to Print 1st  $N$  odd / even numbers & calculate their sum & avg.

**60.** Write a program to Print all numbers between given two numbers  $x$  &  $y$  including  $x$  &  $y$ , & calculate their sum & avg.

**61.** Write a program to print all odd/even numbers between given two Numbers  $x$  &  $y$  including  $x$  &  $y$ , & their sum & avg.

**62.** Write a program to print every third number beginning from 2 until Number  $< 100$ , & calculate their sum & avg.

**63.** Write a program to print all numbers exactly divisible by 5 until number  $< 100$ , & calculate their sum & avg. Use modulus operator to check Divisibility.

**64.** Write a program to Print the following series: -15, -10, -5, 0, 5, 10, 15.

**65.** Write a program to Print the value of the following series : -1,  $x$ ,  $-x^2$ ,  $x^3$ ,  $-x^4$ , .....

**66.** Write a program to Print the value of the following series :  $\sin x$ ,  $\sin 2x$ ,  $\sin 3x$ ,  $\sin 4x$ , ..... $\sin nx$ .

**67.** Write a program to Print the multiplication table of given number  $X$  until  $n$ , in the Following format:  $X \times 1 = X$

**68.** Write a program To calculate the power of a number i.e.  $xy$  or  $xn$  without using `pow()` function

**69.** Write a program to Calculate the factorial of a number.

**70.** Write a program to Print all letters of the alphabet in upper & lower case.

**71.** Write a program to Print all characters between given 2 numbers  $x$  &  $y$ .

**72.** Write a program to Print 1st  $N$  numbers of the Fibonacci series.

**73.** Write a program to Check if the given number is prime or not.

**74.** Write a program for  $1 - x + x^2/2! - x^3/3! + x^4/4! \dots x^n/n!$  terms.

**75.** Write a “C ” program for following Pattern..

1

1 2

1 2 3

12 3 4

1

2 2

3 3 3

4 4 4 4

1

1 2 1

1 2 3 2 1

1 2 3 4 3 2 1

G H I J

D E F

B C

A

1

0 1

0 1 0

1 0 1 0

## CHAPTER:4 Array

It is a collection of similar type of data which may be int type, char type, float type or user-defined type such as structure or class. The significance of an array is that each array element is stored in consecutive memory locations and the array elements are accessed by their index value, which is also called subscript value.

General format of array:

```
data type array name[size];
```

## Single dimensional array

In this type of array only one sub-script(index) is used in the program.

### Syntax:

```
data type array name [size];
```

Example: Single dimensional array

```
#include<iostream.h>
#include<conio.h>
int main()
{
    clrscr();
    int a[10],i;
    for(i=1;i<=5;i++)
    {
        cout<<"enter the no.";
        cin>>a[i];
    }
    cout<<"Array elements are as follows:\n";
    for(i=1;i<=5;i++)
    {
        cout<<a[i]<<" ";
    }
    getch();
    return 0;
}
```

### Output:

enter the no:12

enter the no:34

enter the no:44

enter the no:55

enter the no:66

Array elements are as follows:

12 34 44 55 66

## Multidimensional array

In this type of array more than two subscript is used in the program. it is also known as array of array.

### Syntax-

```
data type array name [row][column];
```

**Example:** A program to add two matrix

```
#include<iostream.h>
#include<conio.h>
void main()
{
    int a[3][4],b[3][4],x[3][4];
    int r,c;
    // read value in matrices
    cout<<“enter the first matrix row wise \n”;
    for (r=0;r<3;r++)
    {
        for(c=0;c<4;c++)
        {
            cin>>a[r][c];
        }
    }
    cout<<“enter the second matrix row wise\n”;
    for (r=0;r<3;r++)
    {
        for (c=0;c<4;c++)
        {
            cin>>b[r][c];
        }
    }
}
```

```
}

}

//addition of two matrix

for (r=0;r<3;r++)

{

for (c=0;c<4;c++)

{

x[r][c]= a[r][c]+b[r][c];

}

}

//display the matrix

for (r=0;r<3;r++)

{

for (c=0;c<4;c++)

{

cout<<x[r][c]<<"\t";

}

cout<<"\n";

}

getch();

}
```

### **Output:**

enter the first matrix row wise

1 2 3 4

5 6 7 8

1 2 3 4

enter the second matrix row wise

1 2 3 4

5 6 7 8

1 2 3 4

2 4 6 8

10 12 14 16

2 4 6 8

## Anonymous unions

In C++ we have the option to declare anonymous unions. If we declare a union without any name, the union will be anonymous and we will be able to access its members directly by their member names. For example, look at the difference between these two structure declarations:

structure with regular union	structure with anonymous union
<pre>struct {     char title[50];     char author[50];     union {         float dollars;         int yens;     } price; } book;</pre>	<pre>struct {     char title[50];     char author[50];     union {         float dollars;         int yens;     }; } book;</pre>

The only difference between the two pieces of code is that in the first one we have given a name to the union (price) and in the second one we have not. The difference is seen when we access the members dollars and yens of an object of this type. For an object of the first type, it would be:

book.price.dollars

book.price.yens

whereas for an object of the second type, it would be:

book.dollars

book.yens

Once again I remind you that because it is a union and not a struct, the members dollars and yens occupy the same physical space in the memory so they cannot be used to store two different values simultaneously. You can set a value for price in dollars or in yens, but not in both.

## Exercise

### Short Questions

1. Given the declaration

```
char letter[3][3] ={  
    {'a', 'b', 'c'},  
    {'d', 'e', 'f'},  
    {'g', 'h', 'i'}  
};
```

which character is stored in letter[1][2]? \_\_\_\_\_

2. If an array has a 100 elements what is the allowable range of subscripts?

3. What is the difference between the expressions a4 and a[4]?

4. Write a declaration for a 100 element array of floats. Include an initialisation of the first four elements to 1.0, 2.0, 3.0 and 4.0.

5. An array day is declared as follows:

```
int day[] = {mon, tue, wed, thu, fri};
```

How many elements has the array day? If the declaration is changed to

```
int day[7] = {mon, tue, wed, thu, fri};
```

how many elements does day have?

### True/ False

1. The components of an array are all of the same data type.

2. The size of an array is established at compile time rather than at execution time.

3. In C++, an array can be passed as a parameter by reference.

4. C++ does not check for out-of-bounds array indices while a program is running.

5. The array declared as

```
float angle[10][25];
```

has 10 rows and 25 columns.

The array declared as

```
int bowlingScore[6][12];
```

contains 72 int components.

6. the computer's memory, C++ stores two-dimensional arrays in row order.
7. The components of an array are all of the same data type.
8. The size of an array is established at compile time rather than at execution time.
9. In C++, an array can be passed as a parameter by reference.
10. If a program has the declarations

```
enum WeatherType {SUNNY, CLOUDY, FOGGY, WINDY};  
int frequency[4];
```

then the statement

```
cout << frequency[CLOUDY];
```

is syntactically valid.

11. C++ does not check for out-of-bounds array indices while a program is running.
12. The function heading

```
void SomeFunc( float x[] )
```

causes a compile-time error because the size of the array is missing.

13. The statement

```
frequency['G']++;
```

is an example of the use of an array index with semantic content.

14. The array declared as

```
float angle[10][25];
```

has 10 rows and 25 columns.

15. The array declared as

```
int bowlingScore[6][12];
```

contains 72 int components.

16. If a program contains the declaration

```
int salePrice[100][100];
```

then the statement

```
cout << salePrice[3];
```

outputs all the values in row 3 of the array.

17. In the computer's memory, C++ stores two-dimensional arrays in row order.
18. When a two-dimensional array is declared as a parameter, the C++ compiler ignores the sizes of both dimensions.
19. When you declare a two-dimensional array as a parameter, you can omit the size of the first dimension but not the second.
20. When a two-dimensional array is passed as a parameter, the number of columns in the parameter must be identical to the number of columns in the argument.
21. When a two-dimensional array is passed as a parameter, the number of rows in the parameter must be identical to the number of rows in the argument.
22. If an array has a 100 elements what is the allowable range of subscripts?
23. What is the difference between the expressions a4 and a[4]?
24. Write a declaration for a 100 element array of floats. Include an initialization of the first four elements to 1.0, 2.0, 3.0 and 4.0.
25. An array day is declared as follows:

```
int day[] = {mon, tue, wed, thu, fri};
```

How many elements have the array day? If the declaration is changed to

```
int day[7] = {mon, tue, wed, thu, fri};
```

how many elements does day have?

## Multiple Choice Questions

1. Which of the following could be used to declare an array alpha and initialize its components to 10, 20, and 30?
  - a) `int alpha[3] = {10, 20, 30};`
  - b) `int alpha[] = {10, 20, 30};`
  - c) `int alpha[3] = {10 20 30};`
  - d) a and b above

**e)** a, b, and c above

**2.** Given the declarations

```
int status[10];
```

```
int i;
```

which of the following loops correctly zeros out the status array?

- a)** for (*i* = 0; *i* <= 10; *i*++) *status*[*i*] = 0;
- b)** for (*i* = 0; *i* < 10; *i*++) *status*[*i*] = 0;
- c)** for (*i* = 1; *i* <= 10; *i*++) *status*[*i*] = 0;
- d)** for (*i* = 1; *i* < 10; *i*++) *status*[*i*] = 0;
- e)** for (*i* = 1; *i* <= 11; *i*++) *status*[*i*] = 0;

**3.** What is the output of the following program fragment?

```
int alpha[5] = {100, 200, 300, 400, 500};
```

```
int i;
```

```
for (i = 4; i > 0; i—)
```

```
cout << alpha[i] << ' ';
```

- a)** 400 300 200 100
- b)** 500 400 300 200 100
- c)** 500 400 300 200
- d)** It cannot be answered from the information given.

**4.** What is the output of the following program fragment?

```
int alpha[5] = {100, 200, 300, 400, 500};
```

```
int i;
```

```
for (i = 4; i >= 0; i—)
```

```
cout << alpha[i] << ' ';
```

- a)** 400 300 200 100 0
- b)** 500 400 300 200 100
- c)** 500 400 300 200

**d)** It cannot be answered from the information given.

5. Given a 5000-element array beta, which of the code fragments below could be used to print out the values of beta[0], beta[2], beta[4], and so forth? (All variables are of type int.)

**a)** for (i = 0; i < 5000; i = i + 2)  
cout << beta[i] << endl;

**b)** for (i = 0; i < 2500; i++)  
cout << beta[2\*i] << endl;

**c)** for (i = 0; i < 2500; i++)  
cout << beta[i]\*2 << endl;

**d)** a and b above

6. Which of the following statements about passing C++ arrays as parameters is false?

**a)** It is impossible to pass an array by value.

**b)** When declaring an array as a formal parameter, you do not attach an ampersand (&) to the name of the component type.

**c)** When declaring an array as a parameter, you must include its size within square brackets.

**d)** At run time, the base address of the argument is passed to the function.

7. Given the declaration

```
char table[7][9];
```

which of the following stores the character ‘B’ into the fifth row and second column of the array?

**a)** table[4][1] = ‘B’;

**b)** table[1][4] = ‘B’;

**c)** table[5][2] = ‘B’;

**d)** table[2][5] = ‘B’;

**e)** table[5] = ‘B’;

8. The following program fragment is intended to zero out a two-dimensional array:

```
int table[10][20];  
int i, j;  
for (i = 0; i < 10; i++)  
for (j = 0; j < 20; j++)  
// Statement is missing here
```

What is the missing statement?

- a)** table[i][j] = 0;
- b)** table[j][i] = 0;
- c)** table[i+1][j+1] = 0;
- d)** table[j+1][i+1] = 0;
- e)** table[i-1][j-1] = 0;

#### 9. Given the nested For loops

```
for (i = 0; i < M; i++)  
for (j = 0; j < N; j++)  
cout << table[i][j];
```

what is the appropriate declaration for table?

- a)** int table[M][N];
- b)** int table[N][M];
- c)** int table[M+N];
- d)** int table[M+1][N+1];
- e)** int table[N+1][M+1];

#### 10. Given the declarations

```
float alpha[5][50];  
float sum = 0.0;  
which of the following computes the sum of the elements in row 2 of alpha?  
a) for (i = 0; i < 5; i++)  
sum = sum + alpha[i][2];
```

**b)** for ( $i = 0; i < 50; i++$ )

    sum = sum + alpha[i][2];

**c)** for ( $i = 0; i < 5; i++$ )

    sum = sum + alpha[2][i];

**d)** for ( $i = 0; i < 50; i++$ )

    sum = sum + alpha[2][i];

**11.** Given the declaration

int score[5][8];

which of the following outputs the array components in row order?

**a)** for ( $j = 0; j < 8; j++$ )

    for ( $i = 0; i < 5; i++$ )

        cout << score[i][j];

**b)** for ( $i = 0; i < 8; i++$ )

    for ( $j = 0; j < 5; j++$ )

        cout << score[i][j];

**c)** for ( $i = 0; i < 5; i++$ )

    for ( $j = 0; j < 8; j++$ )

        cout << score[i][j];

**d)** for ( $j = 0; j < 5; j++$ )

    for ( $i = 0; i < 8; i++$ )

        cout << score[i][j];

**e)** for ( $i = 0; i < 8; i++$ )

    for ( $j = 0; j < 5; j++$ )

        cout << score[j][i];

**12.** After execution of the program fragment

int table[3][3];

int i, j;

```
for (i = 0; i < 3; i++)  
for (j = 0; j < 3; j++)  
table[i][j] = i + 2*j;
```

what are the contents of the table array?

- a)** 2 4 0 5 0 6 2 6 8
- b)** 0 2 4 1 3 5 2 4 6
- c)** 1 3 5 1 3 5 2 4 6
- d)** 0 2 4 1 3 6 2 4 6

**13.** After execution of the program fragment

```
int table[3][3];  
  
int i, j;  
  
for (i = 0; i < 3; i++)  
for (j = 0; j < 3; j++)  
table[j][i] = i + 2*j;
```

what are the contents of the table array?

- a)** 0 2 4 1 3 5 2 4 6
- b)** 0 1 2 2 3 4 4 5 6
- c)** 0 2 4 1 3 5 0 0 0
- d)** 0 1 0 2 3 0 5 0 0

**14.** Your manager has asked you to integrate an existing project with an external timer. Her requirements include adding a global variable whose contents can be only read by the software but whose value will be continuously updated by the hardware clock.

Referring to the above scenario, which one of the following variable declarations satisfies all the requirements?

- a)** extern volatile clock;
- b)** extern const volatile long clock;
- c)** extern long clock;

**d)** extern const mutable long clock;

**e)** extern mutable long clock;

**15.** Which set of preprocessor directives is used to prevent multiple inclusions of header files?

**a)** #ifndef, #define and #endif

**b)** #ifdef and #endif

**c)** #define and #endif only

**d)** #\$if and #endif

**e)** #if and #define

**16.** char s[] = {'a', 'b', 'c', 0, 'd', 'e', 'f', 0};

int I = sizeof(s);

Referring to the sample code above, what value does I contain?

**a)** 3

**b)** 6

**c)** 7

**d)** 8

**e)** 9

**17.** Sample Code

```
class HasStatic
```

```
{
```

```
    static int I;
```

```
};
```

Referring to the sample code above, what is the appropriate method of defining the member variable “I”, and assigning it the value 10, outside of the class declaration?

**a)** int static I = 10;

**b)** int HasStatic::I = 10;

**c)** HasStatic I = 10;

**d)** static I(10);

**e)** static I = 10;

**18.** Which of the following statements about C++ arrays is true?

**a)** Array components cannot be of floating point types.

**b)** The index type of an array can be any data type.

**c)** An array component can be treated the same as a simple variable of its component type.

**d)** a and b above

**e)** a, b, and c above

**19.** Which of the following could be used to declare an array alpha and initialize its components to 10, 20, and 30?

**a)** int alpha[3] = {10, 20, 30};

**b)** int alpha[] = {10, 20, 30};

**c)** int alpha[3] = {10 20 30};

**d)** a and b above

**e)** a, b, and c above

**20.** The following code fragment invokes a function named InitToZero:

```
int alpha[10][20];
```

```
InitToZero(alpha);
```

Which of the following is a valid function heading for InitToZero?

**a)** void InitToZero( int beta[][] )

**b)** void InitToZero( int beta[10][20] )

**c)** void InitToZero( int beta[10][] )

**d)** void InitToZero( int beta[][20] )

**e)** b and d above

## Predict Output

**1.** What is the output of the following program fragment?

```
int gamma[3] = {5, 10, 15};
```

```
int i;
```

```
for (i = 0; i <= 3; i++)
```

```
cout << gamma[i] << ' ';
```

**2.** After execution of the code fragment

```
int arr[5];
```

```
int i;
```

```
for (i = 0; i < 5; i++){
```

```
arr[i] = i + 2;
```

```
if (i >= 3)
```

```
arr[i-1] = arr[i] + 3;
```

```
}
```

what is contained in arr[1] and arr[3]?

**3.** What is the contents of array **arr** after the for loop?

```
int j = 0;
```

```
for (i = 0; i < 5; i++){
```

```
arr[i] = i+j;
```

```
j = j + 1;
```

```
}
```

**4.** What would be output by the following section of C++?

```
int A[5] = {1 , 2, 3, 4};
```

```
int i;
```

```
for (i=0; i<5; i++)
```

```
{
```

```
    A[i] = 2*A[i];
```

```
    cout << A[i] << " ";
```

```
}
```

**5.** What is the output of the following program fragment?

```
int gamma[3] = {5, 10, 15};  
int i;  
for (i = 0; i < 3; i++)  
cout << gamma[i] << ' ';
```

**6.** After execution of the code fragment

```
int arr[5];  
int i;  
for (i = 0; i < 5; i++){  
arr[i] = i + 2;  
if (i >= 3)  
arr[i-1] = arr[i] + 3;  
}
```

what is contained in arr[1] and arr[3]?

**7.** What is the contents of array arr after the for loop?

```
int j = 0;  
for (i = 0; i < 5; i++)  
{  
arr[i] = i+j;  
j = j + 1;  
}
```

**8.** Given the declaration

```
char letter[3][3] =  
{  
{'a', 'b', 'c'},  
{'d', 'e', 'f'},  
{'g', 'h', 'i'}
```

};

which character is stored in letter[1][2]?

9. What would be output by the following section of C++?

```
int A[5] = {1 , 2, 3, 4};
```

```
int i;
```

```
for (i=0; i<5; i++)
```

```
{A[i] = 2*A[i];
```

```
cout << A[i] << " ";
```

## Find Error in the Code

1. What is wrong with the following section of program?

```
int A[10], i;
```

```
for (i=1; i<=10; i++)
```

```
cin >> A[i];
```

2. What is wrong with the following section of program?

```
int A[10], i;
```

```
for (i=1; i<=10; i++)
```

```
cin >> A[i];
```

## Programming Exercises

1. To familiarize yourself with using arrays write a program that declares two float arrays, say with 5 elements each, and carries out the following:

- a)** Input some data from the user into the two arrays.
- b)** Output the sum of the elements in each of the two arrays.
- c)** Output the inner product of the two arrays - that is the sum of the products of corresponding elements  $A[0]*B[0] + A[1]*B[1] + \dots$ etc.
- d)** Produce an estimate of how different the values in the two arrays are by evaluating the sum of squares of the differences between corresponding elements of the two arrays divided by the number of elements.

Start by only entering and printing the values in the arrays to ensure you are capturing the data correctly. Then add each of the facilities above in turn.

2. Write a program that uses `cin` method to read a number of positive integers and stores it in an integer array, until a `-1` is read. The program then outputs the array elements in reverse order and the average of the values along with the number of elements it read.
3. Statistics provide a way to characterize a data set. This exercise uses an array to store data for analysis. Read `n` from the console where `n` is the number of data values to analyze. Generate `n` elements whose value is between 0-99 using `rand()` placing these elements in a data array. Then compute:
  - a) minimum and maximum elements
  - b) average ( $\bar{x}$ ).
  - c) Standard deviation
4. To familiarize yourself with using arrays write a program that declares two float arrays, say with 5 elements each, and carries out the following:
  - a) Input some data from the user into the two arrays.
  - b) Output the sum of the elements in each of the two arrays.
  - c) Output the inner product of the two arrays - that is the sum of the products of corresponding elements  $A[0]*B[0] + A[1]*B[1] + \dots$ etc.
  - d) Produce an estimate of how different the values in the two arrays are by evaluating the sum of squares of the differences between corresponding elements of the two arrays divided by the number of elements.

Start by only entering and printing the values in the arrays to ensure you are capturing the data correctly. Then add each of the facilities above in turn.

5. A popular method of displaying data is in a Histogram. A histogram counts how many items of data fall in each of `n` equally sized intervals and displays the results as a bar chart in which each bar is proportional in length to the number of data items falling in that interval.
6. Write a program that generates `n` random integers in the range 0-99 and produces a Histogram from the data. Assume that we wish to count the number of numbers that lie in each of the intervals 0-9, 10-19, 20-29, ..... , 90-99. This requires that we hold 10 counts, use an array to hold the 10 counts. While it would be possible to check which

range a value  $x$  lies in by using if-else statements this would be pretty tedious. A much better way is to note that the value of  $x/10$  returns the index of the count array element to increment. Having calculated the interval counts draw the Histogram by printing each bar of the Histogram as an appropriately sized line of X's across the screen as below

- a)** 0 - 9 16 XXXXXXXXXXXXXXXXX
- b)** 10 - 19 13 XXXXXXXXXXXXXXXXX
- c)** 20 - 29 17 XXXXXXXXXXXXXXXXX
- d)** etc.

7. In question 1 above you should have written C++ statements to enter numbers into an array. Convert these statements into a general function for array input. Your function should indicate the number of elements to be entered and should signal an error situation if this is greater than the size of the array—think about the required parameters. Also write a function to output  $n$  elements of a given array five to a line.

8. Write a driver program to test these functions and once you are satisfied they are working correctly write functions:

- a)** To return the minimum element in the first  $n$  elements of an array.
- b)** To return a count of the number of corresponding elements which differ in the first  $n$  elements of two arrays of the same size.
- c)** Which searches the first  $n$  elements of an array for an element with a given value. If the value is found then the function should return **true** and also return the index of the element in the array. If not found then the function should return **false**.

In these functions incorporate error testing for a number of elements greater than the array size.

9. Write a program to play a game in which you try to sink a fleet of five navy vessels by guessing their locations on a grid. The ships are different length and are positioned as follows:

Frigate: 2 locations, located between (2, 4) and (2, 6).

Tender: 2 locations, located between (4, 6) and (4, 8).

Destroyer: 3 locations, located between (8, 7) and (8, 10).

Cruiser: 3 locations, located between (9, 1) and (9, 4).

Carrier: 4 locations, located between (11, 4) and (11, 8).

Your program will displace the ships on a 12 x 12 grid initially. The user will enter the coordinates in the range of 1 to 12 for rows and columns. Your program will check and report if this is a hit or a miss. If it is a hit, your program should report it is a hit and that ship will be reported sunk.

The user will be given 10 shots. If he sunk all the ships in less than 11 shots, user will be the winner, else loser.

**10.** In this assignment you are asked to write a program that reads in two  $n \times n$  matrix and after multiplying the matrices to output the resulting matrix. Recall that if the  $n \times n$  matrices are A and B, the resulting matrix would be C of size  $n \times n$ . Find out a general solution for the multiplication of two-dimensional matrices.

**11.** Write a C++ program to perform string manipulation.

**12.** Find the length of a string. Compare two strings, Concatenate two strings, Reverse a string, Copy a string to another location.

**13.** Write a C++ program to find quotient and remainder of 2 numbers.

**14.** Write a function to find the smallest element of the matrix A.

**15.** Write the declaration statement for a one-dimensional array named alpha whose index values range from 0 through 99 and whose component type is float:

**16.** Write the declaration statement for a 25-element one-dimensional array named letterGrade whose component type is char:

**17.** Given the enumeration type

```
enum FlagColors {RED, WHITE, BLUE};
```

write the declaration statement for a one-dimensional array named flagArray whose index values range from 0 through 50 and whose component type is FlagColors:

**18.** Given the enumeration type

```
enum Grades {A, B, C, D, F, AU, W};
```

write the declaration statement for a one-dimensional array named gradeCount whose index values are of type Grades and whose component type is int:

**19.** Given the declaration

```
char charArray[15];
```

**20.** write an assignment statement that stores the value 'X' into the sixth component of the array:

**21.** Given the declarations

```
enum Colors {RED, ORANGE, YELLOW, GREEN, BLUE};
```

```
float waveLength[5]; // To be indexed by values of type Colors
```

write a statement to print the third component of the waveLength array:

**22.** Write a C + + program to ask the user to enter a character string and then print the Characters in th12e reverse order.

**23.** Write a C + + program to find the minimum and maximum of an array of n numbers.

**24.** Read the value of n and the n numbers from user.

**25.** To familiarize yourself with using arrays write a program that declares two float arrays, say with 5 elements each, and carries out the following:

**a)** Input some data from the user into the two arrays.

**b)** Output the sum of the elements in each of the two arrays.

**C)** Output the inner product of the two arrays - that is the sum of the products of corresponding elements  $A[0]*B[0] + A[1]*B[1] + \dots$ etc.

**d)** Produce an estimate of how different the values in the two arrays are by evaluating the sum of squares of the differences between corresponding elements of the two arrays divided by the number of elements.

**e)** Start by only entering and printing the values in the arrays to ensure you are capturing the data correctly. Then add each of the facilities above in turn.

**26.** A popular method of displaying data is in a Histogram. A histogram counts how many items of data fall in each of n equally sized intervals and displays the results as a bar chart in which each bar is proportional in length to the number of data items falling in that interval.

Write a program that generates n random integers in the range 0-99 and produces a Histogram from the data. Assume that we wish to count the number of numbers that lie in each of the intervals 0-9, 10-19, 20-29, ..... , 90-99. This requires that we hold 10 counts, use an array to hold the 10 counts. While it would be possible to check which range a value x lies in by using if-else statements this would be pretty tedious. A much

better way is to note that the value of  $x/10$  returns the index of the count array element to increment. Having calculated the interval counts draw the Histogram by printing each bar of the Histogram as an appropriately sized line of X's across the screen as below

- a)** 0 - 9 16 XXXXXXXXXXXXXXXXX
- b)** 10 - 19 13 XXXXXXXXXXXXXXXXX
- c)** 20 - 29 17 XXXXXXXXXXXXXXXXX
- d)** etc.

**27.** In question 1 above you should have written C++ statements to enter numbers into an array. Convert these statements into a general function for array input. Your function should indicate the number of elements to be entered and should signal an error situation if this is greater than the size of the array—think about the required parameters. Also write a function to output  $n$  elements of a given array five to a line.

- a)** Write a driver program to test these functions and once you are satisfied they are working correctly write functions:
- b)** To return the minimum element in the first  $n$  elements of an array.
- c)** To return a count of the number of corresponding elements which differ in the first  $n$  elements of two arrays of the same size.
- d)** Which searches the first  $n$  elements of an array for an element with a given value. If the value is found then the function should return **true** and also return the index of the element in the array. If not found then the function should return **false**.
- e)** In these functions incorporate error testing for a number of elements greater than the array size.

**28.** Write a C++ program to generate Prime numbers between 1 and 50.

**29.** Write a C++ program to perform matrix addition and multiplication.

**30.** Write a C++ program to check whether the given matrix is a sparse matrix or not.

**31.** Write a C++ program to overload unary minus operator.

**32.** Write a C++ program to calculate total sales and average sales made by a salesman.

**33.** Write a program to read  $N$  integers and print  $N$  integers Using an array.

**34.** Write a program to find the smallest and largest number in an array of  $N$  integers.

**35.** Write a program to arrange an array of N elements into ascending order.

**36.** Write a program to read the marks of one subject of 20 students and compute the number of students in categories FAIL, PASS, FIRST CLASS and DISTINCTION.

**37.** Write a program to insert value at ith location or value entered by user using one dimensional array.

**38.** Write a program to delete value at ith location or value entered by user using one dimensional array.

**39.** Write a program to add two 3\*3 matrices, if A and B both are 3\*3 matrices then resultant matrix  $C=A+B$  where  $C_{ij}=A_{ij}+B_{ij}$  using Two-dimensional array.

**40.** Write a program to check whether given 3\*3 matrix is magic square or not.

**Note :** A square matrix is called magic square if the sum of its all rows, columns and diagonal are equal.

**41.** Write a program to read string from user and print the lenght of the string

**42.** Write a Program to reverse a string

**43.** Write a program to check whether given string is palindrome string or not

**44.** Write a program to read your name and ouput the ascii code of the first character representing your name

**45.** Write a program to print each word of the given string into seperate line

**46.** Write a program to count number of words in a given string

**47.** write a program to count number of occurence of a given character in a given string.



## CHAPTER:5 Functions in C++

A function groups a number of program statements into a unit and give them a name which can then be invoked from other parts of the program.

Dividing a program into functions is one of the major principles of structured programming.

Another reason to use functions is to reduce program size.

Any sequence of instructions that appears in a program more than once is a candidate for being made into a function. The function's code is stored in only one place in memory, even though the function is executed many times in the program.

**There are two types of function:**

- a)** Library function
- b)** User define function

## Library function

The function which are already defines or predefines in the language is known as library function.

## User defines function

The functions which are designed by user on the basis of requirement of a programmer are known as user defines function.

In c++ three terms always associated with the function are:

- a)** Function Prototype(declaration)( use semicolon;)
- b)** Function calling (use semicolon;)
- c)** Function definition

## Function prototyping

The function prototype describes the function interface to the compiler by giving details such as the number and type of arguments and type of return values. When a function is called, the compiler uses the function prototype to ensure that proper arguments are passed and the return value is treated correctly.

The standard form of declaring function is as follows:

return-type function-name (arg1, arg2, ..... , arg n);

### For example,

float area (float, float);

### Already we have seen different categories of functions in C.:

1. Function with no argument and no return value.
2. Function with no argument and return value.
3. Function with arguments and no return value.
4. Function with arguments and return value.

### Example function:

double square (double side)

{

return side\*side;

}

main()

```

{
double a=square(5.6);
}

```

Terminologies	Example
Function prototype/Declaration	double square (double side)
Function body	return side*side;
Function call	double a=square(5.6);
Function parameter	double side

### [Terminologies related to ‘function’ chapter]

**For example:** Area of rectangle using function

```

#include <iostream.h>

#include <conio.h>

int main(void)

{
    float area( float, float);

    clrscr();

    cout<<"Enter length of rectangle"<<endl;

    float length;

    cin>> length;

    cout<<"Enter width of rectangle"<<endl;

    float width;

    cin>> width;

    float rect_area = area(length, width);

    cout << "Area of rectangle is "<<rect_area;

    getch();

}

float area( float len, float wid)
{

```

```
return(len * wid);
```

```
}
```

## Output:

Enter length of rectangle

11

Enter width of rectangle

12

Area of rectangle is 132

As you can see in the program we have declared main function with return type integer because in C++ the main function always returns 0 or 1 to the compiler.

In main function first we have declared a function named area with two float arguments and a float return type. Then we have inputted from the user the value of variables length and width. Then we have declared a variable named rect\_area and initialize it's value by calling function area with arguments length and width. Here the pointer of compiler will go to the function area and return the multiplication of length and width to the rect\_area.

**For example:** Simple program of addition with function:

```
#include<iostream.h>
#include<conio.h>
void main()
{
    clrscr();
    int a,b,c;
    int add (int a,int b); // function declaration
    cout<<"enter two nos";
    cin>>a>>b;
    c=add(a,b); // function calling
    cout<<"Addition is:"<<c;
    getch();
```

```
}

int add (int x,int y) // function definition

{

int z;

z=x+y;

return z;

}
```

### **Output:**

enter two nos 12 32

Addition is:44

## Types of function parameters

There are two types of parameters associated with functions. they are:

### **1) Actual parameter:**

The parameters associated with function call are called actual parameters.

### **2) Formal parameter:**

The parameters associated with the function definition are called formal parameters.

## Function calling

### Calling function by reference

Up to this we were calling a function with call by value that means the called function create a new set of variables and copies the value of arguments into them the function dose not have the access to the actual variables in the calling program. We can provide access to the variables of calling function in program by using “call by reference “ feature.

In call by reference technique we will pass the address of variables instead of the value of variables.

### **Example:** Use of call by reference

```
#include <iostream.h>

#include <conio.h>
```

```
int main(void)
{
    int first, second;
    void interchange(int &, int &);

    clrscr();
    cout << "Enter first number";
    cin>> first;
    cout<< "Enter second number";
    cin>> second;
    interchange(first, second);
    cout<< "After Interchange"<<endl;
    cout<<"First number= "<<first << endl;
    cout<<"Second number = "<<second;
    getch();
    return(0);
}

void interchange (int &var1, int &var2)
{
    int ts;
    ts = var1;
    var1 = var2;
    var2 = ts;
}
```

### **Output:**

Enter first number 12

Enter second number 24

After Interchange

First number = 24

Second number = 12

As you can see in above program first we have declared the function named interchange() with two integer reference arguments. Then we have inputted two variable's value from the user. Now call the function interchange() with two arguments named first and second. Here we are passing the addresses of first and second not values therefore when we called the interchange() function, the function will receive the addresses of first and second with name var1 and var2. In that function we are interchanging the values of var1 and var2 and we are not returning any values to the main function then also when we printed the values of first and second, their values were changed. This shows that when we pass the addresses of variables and in function we receives their addresses in another variables, then corresponding variables will points to same address. In given program first and var1 will points to same address and second and var2 will points same address.

**Example:** Manually using a call-by-reference using a pointer.

```
#include <iostream.h>
#include <conio.h>
void neg(int *i);
int main()
{
    clrscr();
    int x;
    x = 10;
    cout << x << " negated is ";
    neg(&x);
    cout << x << "\n";
    getch();
    return 0;
}
void neg(int *i)
```

```
{
```

```
*i = -*i;
```

```
}
```

## Output:

```
10 negated is -10
```

In the above program, neg() takes as a parameter a pointer to the integer whose sign it will reverse. Therefore, neg() must be explicitly called with the address of x. Further, inside neg() the \* operator must be used to access the variable pointed to by i. This is how you generate a “manual” call-by-reference in C++, and it is the only way to obtain a call-by-reference using the C subset. Fortunately, in C++ you can automate this feature by using a reference parameter.

To create a reference parameter, precede the parameter’s name with an &. For example, here is how to declare neg() with i declared as a reference parameter:

```
void neg(int &i);
```

For all practical purposes, this causes i to become another name for whatever argument neg() is called with. Any operations that are applied to i actually affect the calling argument. In technical terms, i is an implicit pointer that automatically refers to the argument used in the call to neg(). Once i has been made into a reference, it is no longer necessary (or even legal) to apply the \* operator. Instead, each time i is used, it is implicitly a reference to the argument and any changes made to i affect the argument. Further, when calling neg(), it is no longer necessary (or legal) to precede the argument’s name with the & operator. Instead, the compiler does this automatically.

**For example:** Here is the reference version of the preceding program:

```
// Use a reference parameter.

#include <iostream.h>

#include <conio.h>

//void neg(int &i); // i now a reference

void neg(int &i)

{

    i = -i; // i is now a reference, don't need *
```

```
}

int main()
{
    int x;
    x = 10;
    cout << x << " negated is ";
    neg(x); // no longer need the & operator
    cout << x << "\n";
    return 0;
}
```

## Swapping of two numbers by call by value

**For example:**

```
#include<iostream.h>
#include<conio.h>
void swap(int n1,int n2)
{
    int ts;
    ts=n1;
    n1=n2;
    n2=ts;
    cout<<"\n"<<n1<<"\n"<<n2<<"\n";
}
void main()
{
    int m1=10,m2=20;
    clrscr();
```

```
cout<<"\n Values before invoking swap"<<m1<<"\t"<<m2;  
cout<<"\n Calling swap.....";  
swap(m1,m2);  
cout<<"\n Back to main.....values are...."<<m1<<"\t"<<m2;  
getch();  
}
```

### **Output:**

Values before invoking swap10 20

Calling swap.....

20

10

Back to main.....values are....10 20

## Swapping of two numbers by call by reference using pointer

### **For example:**

```
#include<iostream.h>  
#include<conio.h>  
void swap(int *a,int *b)  
{  
int c;  
c=*a;  
*a=*b;  
*b=c;  
cout<<"The swap function is:"<<*a<<"\n"<<*b;  
}  
void main()  
{
```

```
int a,b;  
  
clrscr();  
  
cout<<“Enter the two numbers:”;  
  
cin>>a>>b;  
  
swap(&a,&b);  
  
getch();  
}
```

**Output:**

Enter the two numbers:33

41

The swap function is:41

33

### Swapping of numbers using call by references only

**For example:**

```
#include<iostream.h>  
  
#include<conio.h>  
  
void swap(int &a,int &b)  
  
{  
  
int c;  
  
c=a;  
  
a=b;  
  
b=c;  
  
cout<<”\n In a swap function:”<<a<<”\n”<<b;  
}  
  
void main()  
{
```

```
int a,b;  
clrscr();  
cout<<“Enter the two numbers”;  
cin>>a>>b;  
swap(a, b);  
getch();  
}
```

**Output:**

Enter the two numbers33

41

In a swap function:41

33

## Return by reference

Up to now we have seen that we are returning values from the calling function. In C++ you can also return the reference of variable from the calling function.

**For example:** Let's see an example of return by reference.

```
#include<iostream.h>
#include<conio.h>
int main(void)
{
    int & min(int &, int &);

    int val1, val2;
    clrscr();
    cout<< "Enter value-1 ";
    cin>>val1;
    cout<< "Enter value-2 ";
    cin>>val2;
    min(val1, val2) = 0;
    cout<< "After calling min() function"<<endl;
    cout<< "The value of val1 is: " << val1 <<endl;
    cout<< "The value of val2 is: " << val2;
    getch();
    return(0);
}

int & min(int &var1, int &var2)
{
    if(var1<var2)
        return var1;
    else
```

```
return var2;
```

```
}
```

### **Output:**

Enter value-1 12

Enter value-2 25

After calling min() function

The value of val1 is: 0

The value of val2 is: 25

As you can see in above program first we have declared a function with two integer reference variables and a return type of integer reference. Then we got input of values of variables val1 and val2 from the user. Then we call the function min () with integer reference variable val1 and val2 and that function will return the integer reference of variable val1 or val2 according to minimum value. Note that in return by reference when we call the function the function will appear on the left hand side of an assignment statement. In given program

```
min( val1, val2) = 0;
```

The function min () will return the address of val1 or val2. If min () returns val1 then the value of val1 will be assign to 0 and if min () returns val2 then the value of val2 will be assign to 0. By seeing the output the idea will be more clear. We have assigned two values 12 and 25 to variables val1 and val2 respectively. After calling min () function we have printed the values of both variables. As you can see in the output the value of val1 is 0 that means before calling min () function the value of val1 was less than val2 and therefore min () function has returned the address of val1.

### **For example:** Second example of return by reference.

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
char &replace(int i); // return a reference
```

```
char s[80] = "Hello There";
```

```
int main()
```

```
{
```

```

clrscr();

replace(5) = 'X'; // assign X to space after Hello

cout << s;

getch();

return 0;

}

char &replace(int i)

{

return s[i];

}

```

## Differences between Pointers and References

Restrictions	Reference	Pointer
It reserves a space in the memory to store an address that it points to or reference	No	Yes
It has to be initialized when it is declared	Yes	No
It can be initialized at any time	No	Yes
Once it is initialized, it can be changed to point to another variable of the same type	No	Yes
It has to be dereferenced to get to a value it points to	No	Yes
It can be a NULL pointer/reference	No	Yes
It can point to another reference/pointer	No	Yes
An array of references/pointers can be created	No	Yes

## Static VS Dynamic Memory Allocation

Engineering problems normally require large arrays to store data for some applications. For system of limited memory, having to specify the size of all arrays to be used and allocating enough memory space for them prior to program execution could lead to insufficient memory to run the program. Hence, dynamic memory allocation is used to prevent such problem. In other words, with dynamic memory allocation whenever you need the variables, you will create them and free the memory at any time in your program when you think you don't need them anymore.

In C, you use malloc() and free() for such a purpose, but for C++, you will use new and delete operators.

Table below summarizes the differences between static and dynamic memory allocation.

Static Memory Allocation	Dynamic Memory Allocation
Memory space is allocated during compilation time	Memory space is allocated during run time
The amount of allocated memory is fixed and reserved	The amount of memory is allocated according to what is needed during execution
The memory cannot be released until the program terminates	The memory can be released anytime we want, it won't erase the pointer, just the memory space that it is being allocated to
Have the possibility of failure in executing a program when it is lacking of enough memory	More flexible in this sense, where memory is only allocated when it is needed
Example of creating a static pointer: int num(4); int* numPtr=&num;	Example of creating a dynamic pointer: int* numPtr=new int(4);
In case of static memory allocation stack is used.	A region of memory called heap is used for this purpose; a pool of free memory locations that are not being used by any program

In using dynamic memory allocation, 'memory leak' problem might occur that can cause poor memory utilization problem. It occurs when you fail to return the memory to the free store when it is no longer in use. Have a look at the example given below:

```
float* ptr=new float;
```

```
*ptr=7.9;
```

```
ptr=new float;
```

```
*ptr=5.1;
```

**For this example**, in line 1 and 2, you have allocated memory space of type float with value of 7.9 to ptr. Thus ptr holds the address of this block of memory. But without releasing the memory, you reallocate another space of type float with value of 5.1 (as seen in line 3 and 4). The first memory block is not deleted, however the address is lost since ptr now contains the address of the second block. Hence, the program will not be able to

use the first block eventhough it still occupies some memory space.

Hence, the solution to this problem is to always free the memory that it has been allocated (by using delete operator) before you need to allocate new value to the pointer.

## Inline functions

Inline functions are those functions where when the call is made to them, the actual code of the function gets placed in the calling program. We know that functions save memory space because all the calls to the function cause same code to be executed, the function body need not be duplicated in memory. When the compiler sees a function call, it normally generates a jump to the function. At the end of function it jumps back to the instruction following the call. While this sequence of events may save memory space, but it takes some extra time for following:

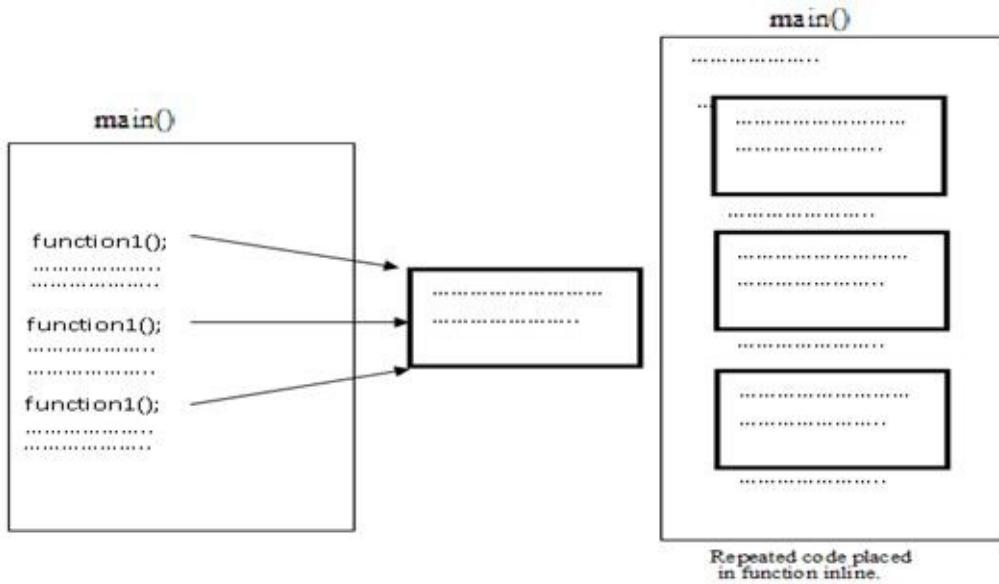
- a)** There must be an instruction for the jump to the function,
- b)** Instructions for saving registers,
- c)** Instructions for pushing arguments on to the stack in the calling program and removing them from the stack in the function,
- d)** Instructions for restoring the register and
- e)** An instruction to return to the calling program.

This would be expensive particularly when the called function body is small.

In such cases one solution is that you could simply repeat the necessary code in your program, inserting the same group of statements wherever it is needed. The trouble with repeatedly inserting the same code is that you lose the benefits of program organization and clarity with using functions. The program may run faster but it takes more space of code and it increases the complexity of code.

To save execution time and reduce the complexity of code, in short functions you may elect to put the code of the function body directly in line with the code in the calling program. That is, each time there is a function call in the source file, the actual code from the function is inserted, instead of a jump to the function.

The idea of in line functions will be more clear through the following figure.



**[Figure: Functions versus Inline Function]**

As you can see in above figure that you can place the functions as inline. The inline function behave like a normal function in the source file but compiles into inline code instead of into a function. Thus, the source file remains well organized and easy to read. Whenever you want to declare a function as inline you have to just put inline keyword before a function name. The standard form of inline function is as follows:

```
inline function-name(arg1, arg2, ...., arg n)
{
statements
}
```

All inline functions must be defined before they are called. We should use inline functions whenever there are two or three statements in the function because the speed benefits of inline function diminish as the function grows in size. Also we cannot use loop structure in inline function. The inline keyword merely sends a request, not a command, to the compiler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function. There are some cases or situations where inline may not work, those are as follows.

- a)** For functions returning values, if a loop, a switch or a goto exists.
- b)** For functions not returning values, if return statement exists.
- c)** If functions contain static variables.
- d)** If inline functions are recursive.

Whenever you declare a function as inline consider these four conditions, let's see an example of inline function.

**Example :** To find maximum of two values using inline function.

```
#include <iostream.h>
#include <conio.h>
inline int max(int a,int b)
{
    return(a>b ? a:b);
}
int main(void)
{
    int p=5,q=6,r;
    clrscr();
    r=max(p,q);
    cout<<"p= "<<p<<", q = "<<q<<", r= "<<r<<endl;
    r=max(--p,--q);
    cout<<"p= "<<p<<", q = "<<q<<", r= "<<r<<endl;
    getch();
    return(0);
}
```

**Output:**

p= 5, q= 6, r= 6

p= 4, q= 5, r= 5

## Use of inline function

**Example :** Program of addition of two values with inline function.

```
# include<iostream>
#include<conio.h>
```

```
inline int add (int a,int b); // function declaration with keyword
inline.

void main()
{
clrscr();
int a,b,c;
cout<<"enter the no";
cin>>a>>b;
c=add(a,b); // function calling
cout<<c;
getch();
}

int add (int x,int y) // function definition
{
return x+y;
}
```

### **Example 2:**

```
#include<iostream.h>
#include<conio.h>
inline float mul(float x,float y)
{
return (x*y);
}
inline double div(double p,double q)
{
return(p/q);
}
```

```
void main()
{
    clrscr();
    float a=12.34;
    float b=3.4;
    cout<<mul(a,b)<<"\n";
    cout<<div(a,b)<<"\n";
    getch();
}
```

## Reason for the need of Inline Function

Normally, a function call transfers the control from the calling program to the function and after the execution of the program returns the control back to the calling program after the function call. These concepts of function saved program space and memory space are used because the function is stored only in one place and is only executed when it is called. This concept of function execution may be time consuming since the registers and other processes must be saved before the function gets called. The extra time needed and the process of saving is valid for larger functions. If the function is short, the programmer may wish to place the code of the function in the calling program in order for it to be executed faster. This type of function is best handled by the inline function. In this situation, the programmer may be wondering “why not write the short code repeatedly inside the program wherever needed instead of going for inline function?” Although this could accomplish the task, the problem lies in the loss of clarity of the program. If the programmer repeats the same code many times, there will be a loss of clarity in the program. The alternative approach is to allow inline functions to achieve the same purpose, with the concept of functions.

## Default arguments

Whenever you declare a function with some arguments, you call the function with appropriate variables. But C++ allows us to call the functions without passing the arguments, this is done by use of default arguments while declaring a function. For example,

```
float area_rect(float height = 5, float width = 6.5);
```

As you can see in the given example when you call the function `area_rect()` without any arguments then it will consider height as 5 and width as 6.5 by default.

Default arguments are useful in situations where some arguments always have the same value.

**For example**, bank interest may remain same for all customers for a particular period of deposit.

Let's see a program of default arguments.

```
#include <iostream.h>

double test (double a, double b = 7)

{
    return a - b;
}

int main ()

{
    cout << test (14, 5) << endl; // Displays 14 - 5
    cout << test (14) << endl; // Displays 14 - 7
    return 0;
}
```

Let's see one more program of default arguments.

```
#include <iostream.h>
#include <conio.h>
#include <math.h>
```

```
int main(void)
{
int is_triangle(float x = 5, float y = 6, float z = 7);
float area(float x = 5, float y = 6, float z = 7);
float x, y, z;
clrscr();
if (is_triangle())
cout<<“Area = “<< area() << endl;
else
cout<<“Sorry, No triangle with those sides...”;
x=8, y=9, z=10;
if (is_triangle(x, y, z)) //all arguments are passed
cout<<“Area = “<< area(x, y, z) << endl;
else
cout<<“Sorry, No triangle with those sides...”;
if (is_triangle(11)) // only one argument is passed
cout<<“Area = “<< area(11) << endl;
else
cout<<“Sorry, No triangle with those sides...”;
if (is_triangle(11, 12)) //only two arguments are passed.
cout<<“Area = “<< area(11, 12) << endl;
else
cout<<“Sorry, No triangle with those sides...”;
getch();
return(0);
}

int is_triangle(float a, float b, float c)
```

```

{
cout<<“Sides are: ” << a << “, “<< b <<”, “<< c <<” “;
if ((a + b > c) && (b + c > a) && (a + c > b))
return 1;
else
return 0;
}

float area(float a, float b, float c)
{
float Area,s;
s = (a + b + c) / 2;
Area = sqrt(s * (s - a) * (s - b) * (s - c));
return Area;
}

```

### Output:

Sides are: 5, 6, 7 Area = 14.6969

Sides are: 8, 9, 10 Area = 34.197

Sides are: 11, 6, 7 Area = 18.9737

Sides are: 11, 12, 7 Area = 37.9473

### Use of default arguments

It is not necessary for all the parameters in a function's prototype to be assigned default values. But it is necessary that all parameter which are supplied default values are placed to the right of those that are not. For example, the following are erroneous prototypes for the function `is_triangle()`.

```
int is_triangle( float x = 5, float y, float z);
```

```
int is_triangle( float x , float y = 6, float z);
```

These two declarations can be corrected as follows:

```
int is_triangle( float z, float y, float x = 5);
```

```
int is_triangle( float x, float z, float y = 6);
```

## Function overloading

As we can declare two variables with same name in different blocks, we can also define two functions with same name in a program but with different no and/or type of arguments. This mechanism is called function overloading and the functions, which have same names, are called as overloaded functions. In C++, you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call. Function overloading is one way in which the C++ language implements polymorphism. Function overloading is commonly used to create several functions of the same name that perform similar tasks but on different data types or arguments.

**For example:** // overloaded function

```
#include <iostream.h>

int operate (int a, int b)
{
    return (a*b);
}

float operate (float a, float b)
{
    return (a/b);
}

int main ()
{
    int x=5,y=2;
    cout << operate (x,y);
    cout << "\n";
    float n=5.0,m=2.0;
    cout << operate (n,m);
    cout << "\n";
```

```
return 0;
```

```
}
```

### **Output:**

10

2.5

In this case we have defined two functions with the same name, operate, but one of them accepts two parameters of type int and the other one accepts them of type float. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two ints as its arguments it calls to the function that has two int parameters in its prototype and if it is called with two floats it will call to the one which has two float parameters in its prototype.

In the first call to operate the two arguments passed are of type int, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type float, so the function with the second prototype is called. This one has a different behavior: it divides one parameter by the other. So the behavior of a call to operate depends on the type of the arguments passed because the function has been overloaded.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type. Different functions can have the same name provided something allows to distinguish between them: number of parameters, type of parameters...

### **Example 2:**

```
#include<iostream.h>
#include<conio.h>
int volume(int);
float volume(float , int);
long int volume(long int,int,int);
void main( )
{
    cout<<"volume of cube"<<volume (10);
```

```

cout<<"volume of cylinder"<<volume (4.5, 5);
cout<<"volume of rectangular box"<<volume (8, 7, 3);
getch ( );
}

int volume(int a)
{
    return(a*a*a);
}

float volume(float r,int h)
{
    return (3.14 * r * r * h);
}

long int volume(long int l, int b, int h);
{
    return(l*b*h);
}

```

### **Output:**

Volume of cube (when all the sides are same) is: 3350

Volume of block (when height and width are same) is: 10100.25

Volume of block (when all sides are different) is: 1057.875

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate functions for executions. The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same and use that function.
2. If an exact match not found, the compiler uses the integral promotions to the actual arguments. For example, char to int or float to double, etc.
3. When either of them fails, the compiler tries to use the built in conversions to the

actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Consider the following two functions:

```
long absolute(long a)  
double absolute (double b)
```

4. When you call a function `absolute (10)`, the compiler will give error because int argument can be converted to either long or double, thereby creating an ambiguous situation as to which version `absolute` should be used.
5. If all of the steps fail, then the compiler will try the user defined conversions in combination with integral promotions and built in conversions to find a unique match.

## Restrictions on Function overloading in C++

You cannot overload the following function declarations if they appear in the same scope. Note that this list applies only to explicitly declared functions and those that have been introduced through using declarations.

Function declarations that differ only by return type. For example, you cannot use the following declarations:

```
int f();  
float f();
```

Member function declarations that have the same name and the same parameter types, but one of these declarations is a static member function declaration. For example, you cannot declare the following two member function declarations of `f()`:

```
struct A  
{  
    static int f();  
    int f();  
};
```

Function declarations with parameters that differ only by the use of `typedef` names that represent the same type. Note that a `typedef` is a synonym for another type, not a separate type. For example, the following two declarations of `f()` are declarations of the same

function:

```
typedef int I;  
void f(float, int);  
void f(float, I);
```

Function declarations with parameters that differ only because one is a pointer and the other is an array. For example, the following are declarations of the same function:

```
f(char*);  
f(char[10]);
```

Function declarations with parameters that differ only because their default arguments differ. For example, the following are declarations of the same function:

```
void f(int);  
void f(int i = 10);
```

There are some other cases but this information regarding to overloading is enough for the time being.

1. If parameter lists of two functions match exactly but return types differ, then second definition is treated as erroneous redeclaration of first and is flagged as compile time error.

```
unsigned int max(int, int);  
int max(int, int); //redeclaration
```

2. If parameter lists of two functions differ only in their default arguments, the second declaration is treated as a redeclaration of the first.

```
int max(int* ia, int size);  
int max(int*, int = 10); // redeclaration.
```

3. The const or volatile qualifiers are not taken into account.

```
void f(int);  
void f(const int); //redeclaration
```

4. But if const or volatile qualifiers apply to type which is a pointer or reference parameter, then the const or volatile qualifier is taken into account when declaration of different functions are identified.

```
void f(int*);  
void f(const int*); //overloaded  
void f(int&);  
void f(const int&); //overloaded
```

## Recursivity

Recursivity is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or calculate the factorial of numbers. For example, to obtain the factorial of a number ( $n!$ ) the mathematical formula would be:

$$n! = n * (n-1) * (n-2) * (n-3) \dots * 1$$

more concretely,  $5!$  (factorial of 5) would be:

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

**For example:** and a recursive function to calculate this in C++ could be:

```
// factorial calculator

#include <iostream.h>

long factorial (long a)

{
    if (a > 1)
        return (a * factorial (a-1));
    else
        return (1);
}

int main ()

{
    long number;

    cout << "Please type a number: ";
    cin >> number;

    cout << number << "!" << factorial (number);

    return 0;
}
```

**Output:**

Please type a number: 9

$$9! = 362880$$

Notice how in function factorial we included a call to itself, but only if the argument passed was greater than 1, since otherwise the function would perform an infinite recursive loop in which once it arrived to 0 it would continue multiplying by all the negative numbers (probably provoking a stack overflow error on runtime).

This function has a limitation because of the data type we used in its design (long) for more simplicity. The results given will not be valid for values much greater than 10! or 15!, depending on the system you compile it.

Some of the useful library functions:

FUNCTION	HEADER FILE	PURPOSE	PARAMETER(S) TYPE	RESULT
<code>abs(x)</code>	<code>&lt;cstdlib&gt;</code>	Returns the absolute value of its argument: <code>abs(-7) = 7</code>	<code>int</code>	<code>int</code>
<code>ceil(x)</code>	<code>&lt;cmath&gt;</code>	Returns the smallest whole number that is not less than <code>x: ceil(56.34) = 57.0</code>	<code>double</code>	<code>double</code>
<code>cos(x)</code>	<code>&lt;cmath&gt;</code>	Returns the cosine of angle <code>x: cos(0.0) = 1.0</code>	<code>double (radians)</code>	<code>double</code>
<code>exp(x)</code>	<code>&lt;cmath&gt;</code>	Returns $e^x$ , where $e = 2.718$ : <code>exp(1.0) = 2.71828</code>	<code>double</code>	<code>double</code>
<code>fabs(x)</code>	<code>&lt;cmath&gt;</code>	Returns the absolute value of its argument: <code>fabs(-5.67) = 5.67</code>	<code>double</code>	<code>double</code>
<code>floor(x)</code>	<code>&lt;cmath&gt;</code>	Returns the largest whole number that is not greater than <code>x: floor(45.67) = 45.00</code>	<code>double</code>	<code>double</code>
<code>pow(x, y)</code>	<code>&lt;cmath&gt;</code>	Returns $x^y$ ; If $x$ is negative, $y$ must be a whole number: <code>pow(0.16, 0.5) = 0.4</code>	<code>double</code>	<code>double</code>
<code>tolower(x)</code>	<code>&lt;cctype&gt;</code>	Returns the lowercase value of $x$ if $x$ is uppercase; otherwise, returns $x$	<code>int</code>	<code>int</code>
<code>toupper(x)</code>	<code>&lt;cctype&gt;</code>	Returns the uppercase value of $x$ if $x$ is lowercase; otherwise, returns $x$	<code>int</code>	<code>int</code>

## Exercise

### Short Question

1. Consider the function definition

```
void Demo( int intVal, float& floatVal )  
{  
    intVal = intVal * 2;  
    floatVal = float(intVal) + 3.5;  
}
```

Suppose that the caller has variables myInt and myFloat whose values are 20 and 4.8, respectively. What are the values of myInt and myFloat after return from the following function call?

```
Demo(myInt, myFloat);
```

2. What would the following function do?

```
void example(int n)  
{  
    int i;  
    for (i=0; i<n; i++)  
        cout << '*';  
    cout << endl;  
}
```

3. What is the life time of each of the following functions?

- a)** A global variable
- b)** A local variable in a function
- c)** A local, static variable in a function

4. If a local, static variable is initialized in its declaration within a function when does the variable get initialized?

5. What is the scope of a namespace that is specified in a using directive outside of all functions?

**6.** What is the scope of the std namespace in the following code?

```
// include directives and function prototypes here

int main()
{
    using namespace std;
    // rest of main body is here
}

// Function definitions here
```

**7.** Given the function definition

```
void Twist( int a, int& b )
{
    int c;
    c = a + 2;
    a = a * 3;
    b = c + a;
}
```

what is the output of the following code fragment that invokes Twist? (All variables are of type int.)

```
r = 1;
s = 2;
t = 3;
Twist(t, s);
cout << r << ' ' << s << ' ' << t << endl;
```

**8.** How is information supplied as input to a function? How can information be conveyed back to the calling program?

**9.** What would the following function do?

```
void example(int n)
```

```
{  
    int i;  
    for (i=0; i<n; i++)  
        cout << '*';  
    cout << endl;  
}
```

How would you call this function in a program? How would you use this function in producing the following output on the screen?

```
*  
**  
***  
****
```

**10.** Explain the use of following terms:

- a) inline function
- b) return by reference

**11.** Explain the difference between call by value and call by reference.

**12.** Write a short note on default arguments.

**13.** Explain, “The function overloading is feature of polymorphism”.

**14.** Explain the call – by – value and call – by - reference parameter passing methods with an example to each.

### True/ False

**1.** The function heading

```
void SomeFunc( float x[] )
```

causes a compile-time error because the size of the array is missing.

- 2.** When a two-dimensional array is passed as a parameter, the number of columns in the parameter must be identical to the number of columns in the argument.
- 3.** When a two-dimensional array is passed as a parameter, the number of rows in the

parameter must be identical to the number of rows in the argument.

4. True or False? If a variable alpha is accessible only within function F, then alpha is either a local variable within F or a parameter of F.

5. True or False? Suppose the first few lines of a function are as follows:

```
void Calc( /* in */ float beta ){  
    alpha = 3.8 * beta;
```

Then the variable alpha must be a local variable.

6. True or false? In the following function, the declaration of beta includes an initialization.

```
void SomeFunc( int alpha )  
{  
    static int beta = 25;  
    ...  
}
```

beta is initialized once only, the first time the function is called.

7. True or False? When the following value-returning function is executed

```
float Average (int, int, int)  
{  
    int x, y=2;  
    x= 2 * y;  
    return;  
}
```

When the function is called with the statement

```
average (alpha, 34, beta) ;
```

The function is executed, and the function input values are discarded.

8. True or False? Using a pass by reference, passing an int argument to a float parameter is acceptable to the compiler but may produce incorrect results.

**9.** True or False? With argument passage by reference, the address of the caller's argument is sent to the function.

**10.** True or False? If there are several items in a parameter list, the compiler matches the parameters and arguments by their relative positions in the parameter and argument lists.

**11.** True or False? When an argument is passed by reference, the argument can be any expression.

**12.** True or False? Given the function prototype

void Fix( int&, float );

Fix(24, 6.85); is an appropriate function call?

**13.** True or False? Given the function heading

void GetNums(int howMany, float& alpha, float& beta )

both void GetNums( int howMany, float& alpha, float& beta );

and void GetNums( int, float&, float& );

are valid function prototype for GetNums?

**14.** A function that does not return anything has return type void.

**15.** When arguments are passed by value, the function works with the original arguments in the calling program.

**16.** A function can return a value by reference.

**17.** It is not necessary to specify the variable name in the function prototype.

**18.** If function is recursive then we cannot use that function as inline function.

**19.** How are strings handled in C + + ? Discuss with appropriate examples.

**20.** What are the benefits of using functions? Define function definition, function Declaration and function parameters with an example.

## Multiple Choice Questions

**1.** The following code fragment invokes a function named InitToZero:

```
int alpha[10][20];
```

```
InitToZero(alpha);
```

Which of the following is a valid function heading for InitToZero?

- a)** void InitToZero( int beta[][] )
- b)** void InitToZero( int beta[10][20] )
- c)** void InitToZero( int beta[10][] )
- d)** void InitToZero( int beta[][][20] )
- e)** b and d above

2. Given the program fragment

```
char alpha[200];
```

```
char beta[200];
```

```
...
```

```
Copy(alpha, beta, 200); // Copy all components of beta into alpha
```

which of the following is the best function heading for the Copy function?

- a)** void Copy(/\*out\*/char arr1[], /\*in\*/char arr2[], /\*in\*/int length)
- b)** void Copy(/\*out\*/const char arr1[], /\*in\*/char arr2[], /\*in\*/int length)
- c)** void Copy(/\*out\*/char arr1[], /\*in\*/const char arr2[], /\*in\*/int length)
- d)** void Copy(/\*out\*/const char arr1[], /\*in\*/const char arr2[], /\*in\*/ int length)

3. A function can also call itself. This process is called

- a)** Calling of function
- b)** Recursion
- c)** Function prototype
- d)** Function declaration

4. Local variables are always declared inside a particular function, which of the following are true?

- I)** They cannot be accessed from outside that function
- II)** They are accessed from the point of declaration to the end of the block.
- III)** They are visible throughout the program.

**IV)** They are visible in subsequent block.

**a)** I, II, III **b)** I, II **c)** I, II, IV **d)** All

**5.** A few examples of function declaration with default value are

```
int dhoni(int b = a, int b = 10, int c = 12);
```

```
int dhoni(int b = 20, int c);
```

```
int dhoni(int a = 2, int b = 10, int c);
```

```
int dhoni (int a, int b, int c = 30);
```

which one is wrong default declaration.

**6.** A function's single most important role is to...

**a)** Give a name to a block of code  
**b)** Reduce program size  
**c)** Accept arguments and provide a return value  
**d)** Help organize a program into conceptual units.

**7.** A function argument is ...

**a)** A variable in the function that receives a value from the calling program.  
**b)** A way that functions resist accepting the calling program's values.  
**c)** A value sent to the function by the calling program.  
**d)** A value return by the function to the calling program.

**8.** which one from the given below is the output of this code

```
int global=10;  
  
void func(int &x, int y)  
{  
    x=x-y;  
    y=x*10;  
    cout<<x<<','<<y<<endl;  
}  
  
void main()
```

```
{  
    int global=7;  
    func(::global,global);  
    cout<<global<< ','<<::global<<endl;  
}
```

**a)** 3, 30

7, 3

**b)** -3, 30

-3, 7

**c)** 7, 3

3, 30

**d)** 3, 7

3, 30

**9.** main()

```
{  
    int val=5;  
    if(val++==6)  
        cout<<"six";  
    else  
        if(--val==5)  
            cout<<"five";  
    else  
        if(++ val==5)  
            cout<< "still five";  
}
```

**a)** five

**b)** six

**c)** stillfive

**d)** five still five

**10.** void main()

```
{
```

```
int num=20, result=0;
```

```
do
```

```
{
```

```
result=10;
```

```
int digit=num%10;
```

```
result+=digit;
```

```
num/=10;
```

```
}
```

```
while(num);
```

```
cout<< result;
```

```
}
```

**a)** 12

**b)** 13

**c)** 14

**d)** 10

**11.** int func (int &x, int y=10)

```
{
```

```
if (x%y==0)
```

```
return ++x;
```

```
else
```

```
return y--;
```

```
}
```

void main()

```
{  
int p=20,q=23;  
q=func(p,q);  
cout<<p<<','<<q;  
}
```

**a)** 20,23

**b)** 20,20

**c)** 20,22

**d)** 19,20

**12.** void main()

```
{  
int f,s,t;  
f=0;  
s=1;  
for(int i=2; i<10; i++)  
{  
t=f+s;  
f=s;  
s=t;  
}  
cout<<t;  
}
```

**a)** 34

**b)** 36

**c)** 21

**d)** 40

**13.** int a=3;

```
void demo(int x,int y, int &z)
{
    a+=x+y;
    z=a+y;
    y+=x;
    cout<<x<<','<<y<<','<<z<<endl;
}
```

```
void main()
{
    int a=2, b=5;
    demo(::a,a,b);
    cout<<::a<<','<<a<<','<<b <<endl;
}
```

**a)** 3,5,10

8,2,10

**b)** 8,2,10

3,5,10

**c)** 3,5,12

8,2,10

**d)** 3,2,10

8,2,10

**14.** main()

```
{
    int x=3,y,z;
    z=y=x;
    z*=y=x*x;
    cout<<x<<','<<y<<','<<z;
```

}

**a)** 3,9,7

**b)** 3,9,9

**c)** 3,3,3

**d)** 9,3,3

## Predict Output

### 1. Given the function definition

```
void Twist( int a, int& b )  
{  
    int c;  
    c = a + 2;  
    a = a * 3;  
    b = c + a;  
}
```

what is the output of the following code fragment that invokes Twist? (All variables are of type int.)

```
r = 1;  
s = 2;  
t = 3;  
Twist(t, s);  
cout << r << ' ' << s << ' ' << t << endl;
```

### 2. What would be the output from the following programs?

**a)** void change(void)

```
{  
    int x;  
    x = 1;  
}
```

```
void main()
{
    int x;
    x = 0;
    change();
    cout << x << endl;
}
```

**b)** void change(int x)

```
{
    x = 1;
}
```

```
void main()
{
    int x;
    x = 0;
    change(x);
    cout << x << endl;
}
```

## Programming Exercises

1. Write a function heading for a function which will double the first n elements of an array. If the function was amended so that it would return false if n was larger than the size of the array how should the function heading be written? If the function was to be changed so that a new array was produced each of whose elements were double those of the input array how would the heading be written?
2. Write a function prototype for a function that takes two parameters of type float and returns true (1) if the first parameter is greater than the second and otherwise returns false (0).
3. Write a function prototype for a function that takes two parameters of type int and

returns true if these two integers are a valid value for a sum of money in pounds and pence. If not valid then false should be returned.

**4.** A function named ex1 has a local variable named i and another function ex2 has a local variable named i. These two functions are used together with a main program which has a variable named i. Assuming that there are no other errors in the program will this program compile correctly? Will it execute correctly without any run-time errors?

**5.** Write a function which draws a line of n asterisks, n being passed as a parameter to the function. Write a driver program (a program that calls and tests the function) which uses the function to output an m x n block of asterisks, m and n entered by the user.

**6.** Extend the function of the previous exercise so that it prints a line of n asterisks starting in column m. It should take two parameters m and n. If the values of m and n are such that the line of asterisks would extend beyond column 80 then the function should return false and print nothing, otherwise it should output true and print the line of asterisks. Amend your driver program so that it uses the function return value to terminate execution with an error message if m and n are such that there would be line overflow.

Think carefully about the test data you would use to test the function.

**7.** Write a function which converts a sum of money given as an integer number of cent into a floating point value representing the equivalent number of dollars. For example 365 cent would be 3.65 dollars.

**8.** Write a function named Min that returns the smallest of its three integer parameters.

**9.** Write a function named Max that returns the largest of its three integer parameters.

**10.** Write a function named Min\_Max that returns the smallest and the largest of three input values, using the parameters min and max via call by reference.

**11.** Write a function that returns the fifth power of its float parameter using the parameter fifthpow via call by reference.

**12.** Write a function which converts a sum of money given as an integer number of cent into a floating point value representing the equivalent number of dollars. For example 365 cent would be 3.65 dollars. Cents are input as a parameter while dollars are returned to calling function using return statement.

**13.** Write a function

```
void floatopp(float q, int& L, int& P)
```

which converts the sum of money  $q$  in dollars into  $L$  dollars and  $P$  cent where the cents are correctly rounded. Thus if  $q$  was 24.5678 then  $L$  should be set to 24 and  $P$  should be set to 57. Remember that when assigning a real to an integer the real is truncated. Thus to round a real to the nearest integer add 0.5 before assigning to the integer.

Write a simple driver program to test the function. Think carefully about the boundary conditions.

- 14.** Modify the program in 6, such that using “substr” function, you can print your last name first, followed by a comma, and finally your first name.
- 15.** Write a C++ program to overload a function to calculate volume of cube, cylinder and rectangular box.
- 16.** Write a library of integer array functions with a header file “IntegerArray.h” and implementation file “IntegerArray.cpp”, which contains the following functions:
  - a)** A function “input\_array( $a, n$ )” which allows the user to input values for the first  $n$  elements of the array  $a$ .
  - b)** A function “display\_array( $a, n$ )” which displays the values of the first  $n$  elements of the array  $a$  on the screen.
  - c)** A function “copy\_array( $a1, a2, n$ )” which copies the first  $n$  elements of  $a2$  to the respective first  $n$  elements in  $a1$ .
  - d)** Test the functions in a suitably defined main program.

- 17.** Write a program to play a game in which you try to sink a fleet of five navy vessels by guessing their locations on a grid. The ships are different length and are positioned as follows:

Frigate: 2 locations, located between (2, 4) and (2, 6).

Tender: 2 locations, located between (4, 6) and (4, 8).

Destroyer: 3 locations, located between (8, 7) and (8, 10).

Cruiser: 3 locations, located between (9, 1) and (9, 4).

Carrier: 4 locations, located between (11, 4) and (11, 8).

Your program will displace the ships on a 12 x 12 grid initially. The user will enter the

coordinates in the range of 1 to 12 for rows and columns. Your program will check and report if this is a hit or a miss. If it is a hit, your program should report it is a hit and that ship will be reported sunk.

The user will be given 10 shots. If he sunk all the ships in less than 11 shots, user will be the winner, else loser.

18. Let  $F = (a+b)(b+c)(c+a)$ . F is to be calculated for different values of a, b and c. Write a function which calculates F and displays it i.e. define a function with formal parameters with no return value. Call the function for a=3, b=4, and c=5 from the main program.

19. Let the area of triangle is defined by  $\text{Area} = \sqrt{s(s-a)(s-b)(s-c)}$ , where s is defined as  $s = \frac{1}{2}(a+b+c)$  and a, b and c are the lengths of the sides. Write a function that returns its area, i.e. define a function with formal parameters with return value.

20. Write inline functions to ....

a) Convert Celsius temperature to Fahrenheit

b) Convert Fahrenheit temperature to Celsius

c) Convert miles to kilometers

d) Convert kilometers to miles

e) Compute the area and volume of a sphere of radius r.

f) Compute of area and volume of a cone of base radius r and height h.

21. Write a function quad(a, b, c) with default arguments for a, b and c. Your function should return the roots (if any) of the quadratic:

$$ax^2 + bx + c = 0$$

take care to distinguish between the several possible types of roots.

22. Write a program to print prime numbers less than or equal to the value of variable limit, in a loop check whether a number is less than limit or not and if yes then call the function prime() to check whether the number is prime or not.

23. Write a C + + program where you are passing an array to function to find the Maximum among n numbers.

24. Write C ++ program to swap the contents of two variables a and b using call by Address and call by reference.

**25.** area of a regular octagon is

$$\text{Area- Octagon} = 4.828a^2$$

Where a is the length of one side? Write a complete C++ program that asks the user to enter the size of the Octagon (side), and calculate and print the area to three decimal Places of accuracy. Use a # define statement for the multiplicative constant and the pow function to find side-squared value.

**26.** Write a program to find number is odd or not.

**27.** Write a program to find out factorial of given number using function.

**28.** Write a program to generate series x,x<sub>2</sub>,x<sub>3</sub>.....x<sub>n</sub>

**29.** Write a program to find maximum of three numbers.

**30.** Let the vector x be x=[1 3 4 2 5 6 7 9 2 ]. Calculates the sum of the elements of x. Use pointers to manipulate elements in arrays.

**31.** Write a C + + program to declare 4 integers a, b, c and d. use only the address of b and concepts of pointers to assign values to all the declared variables and print out the values of all the variables both using the variable name as well as their address. (assume 1byte required to store an integer)

**32.** Write a C++ program to display the student details using pointers.

**33.** Write a program to print the address of a different datatype variables along with its value.

**34.** Write a program using pointer to read in a array of integers and print its elements in reverse order

**35.** Write a program that compares two integer arrays to see whether they are identical.

**36.** Write a program to sort the array in ascending/descending order using pointers

## Find Error in the Code

**1.** What is wrong with the program? Give the correct program.

```
void main ()  
{ int a, b;  
int *a_ptr, *b_ptr;  
*a_ptr = 17; *b_ptr = 25;
```

```
count << " \n a =" << a; count << " \nb =" << b;
```

```
}
```

## CHAPTER:6 Classes and objects

We can create our own data type in both C and C++. In C we can create our own data type using structure and in C++ we can create our own data type using either structure or class. A class is an extension of structure of C. The structure of C and C++ have different capabilities. In C, structures can not have functions as members but in C++ we can also have functions as members of structures. We have already seen structures in C and all the drawbacks of structure are resolved by the class in C++. A class of C++ provides to include both data and functions (that operate on the data) in the class. To declare a structure a keyword `struct` is used same for declaring a class, a keyword `class` is used. A class provides the facility of data hiding using `private` keyword. By declaring the data as `private`, only function of that class can access the data.

Like other built in types, we cannot operate on two variables of user-defined type. **For example,**

```
struct complex
{
    float x;
    float y;
} c1, c2, c3;
```

Here the complex numbers `c1`, `c2` and `c3` can easily be assigned values using the dot (.) operator. But we cannot add or subtract two complex numbers one from the other. **For example,**

```
c3 = c1 + c2;
```

Now let's see how C++ resolve this problems. C++ has expanded its capabilities further to suit its OOP philosophy. It attempts to bring the user-defined types as close as possible to the built in data types. C++ provides user-defined type known as class.

## Structures versus classes

Syntactically, classes are similar to structures. We may say that variables can be seen as instantiations of structures while objects can be considered as instantiations of classes.

There are several differences between structures and classes.

Class	Structure
All members are private by default in class.	All members are public by default in structure.
Class provides data hiding by using private, public and protected specifiers for members.	Structure does not provide data hiding.
With classes we can use the concept of inheritance	With structures we cannot use the concept of inheritance
With classes we can use the concept of polymorphism	With structures we cannot use the concept of polymorphism

## Class

Class is an important feature of object oriented programming language. Class is a user defines data type which contains members as data & functions. It is collection of various kind of objects. A class is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions. It is define by the class keyword.

An object is an instantiation of a class. As an analogy, in terms of data type and variables, a class would be the type and an object would be the variable. For example, fruit is a class and apple, mango, banana are its objects.

The class is a specially introduced data type in C++. The class declaration is similar to a struct declaration. The general form of class declaration is as follows:

```
class class_name
{
private:
    variable declarations;
    function declarations or definitions;
public:
    variable declarations;
    function declarations or definitions;
};
```

The keyword class specifies that you are declaring your own data type and that data type is “class\_name”. The class is declared within curly braces ({} ) and after curly braces there must be a semicolon (;).

A class definition consists of two parts: header and body. The class header specifies the class name. The class body defines the class members. All the variables declared in class are known as data members and all the functions declared in class are known as member functions.

- a)** Data members specify the representation/properties/characteristics of class objects.
- b)** Member functions specify the class operations, also called the class interface.

The keywords private and public are known as access specifier used for data hiding

mechanism and these keywords are followed by a colon (:). All class members fall under one of the following three access specifiers:

There are such three types of data members used in class:

**a) Public:**

Public members are accessible by all class users. Data members and member function declared as public are accessible outside the class.

**b) Private:**

The keyword private indicates that the declaration is private and private members are only accessible by the class members or friend functions of the class. Private data members and member functions are not accessible outside the class. The data hiding concept of OOP is covered by private declaration of data in the class. The keyword private is optional because by default the members of a class are treated as private. Once you make a member variable private there is no way to change its value except by using one of the member functions. In fact the only place private member variables are used is in member functions. So the implementation of the private data is hidden.

**c) Protected:**

Protected data members and member functions are only available to derived class members(concept of inheritance).

If no keyword “private, public or protected” is written when declaring class members then all the members will be treated as private. Normal good programming practice requires that all member variables be private and that most member functions be public.

Now let's see an example of class declaration.

```
class student
{
    int rollno;
    char name[20];
    float percent;
public:
    void getdata(int, char, float);
```

```
void putdata(void);
```

```
};
```

Here **student** is class name, which contains three data members(rollno, name, percent) and two member functions. Note that we have not specify any keyword before declaring data members so all three data members will be treated as private and two member functions are declared as public. The member functions can be defined within a class or outside the class. You can define a member function of class outside the class using scope resolution operator(::). Let's see an example...

```
class student
```

```
{
```

```
int rollno;
```

```
char name[20];
```

```
float percent;
```

```
public:
```

```
void getdata(int id, char n[20], float p)//inline definition
```

```
{
```

```
rollno = id;
```

```
strcpy(name, n);
```

```
percent= p;
```

```
}
```

```
void putdata(void);
```

```
};
```

```
void student:: putdata(void) //member function defined outside
```

```
class
```

```
{
```

```
cout<<"student id is: "<< rollno;
```

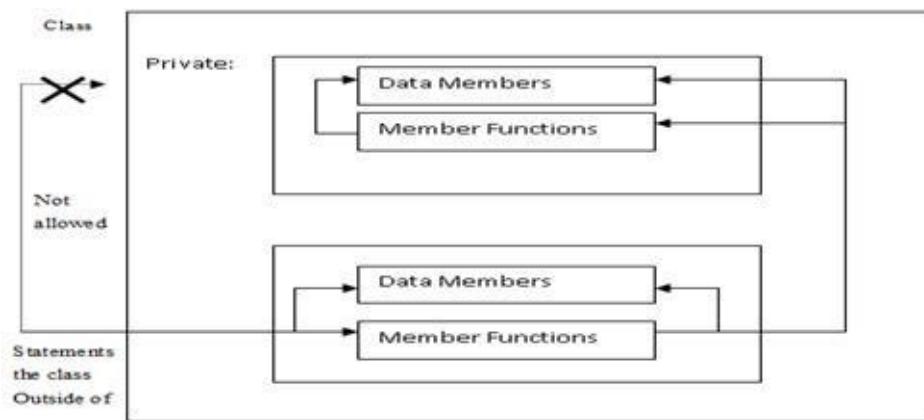
```
cout<<"student name is: "<< name;
```

```
cout<<"student percent is: "<<percent;
```

}

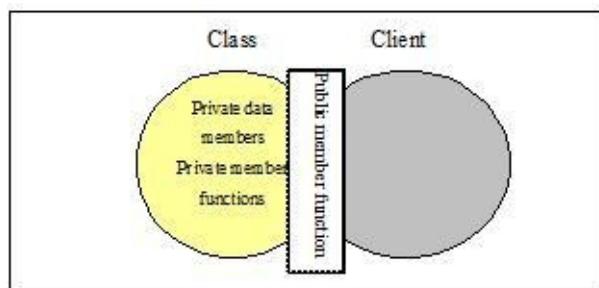
As we have seen that the member functions can be defined inside the class and outside the class. In given class student, the member function `getdata()` is defined within a class and a member function `putdata()` is defined outside the class. When you define a member function outside the class first you write the return type of that function then the class name and then scope resolution operator (`::`) and then function name with body enclosed inside `{}`.

To incorporate the data and functions in a class is called as encapsulation. The encapsulation will be more clear through following figure.



**[Figure:Data hiding mechanism in class]**

As you can see in above figure the statements outside the class can access to public data members and public member functions and through those public member functions the statements outside the class can access private data members and private member functions. Note that the statements outside the class cannot access private data members or private member functions directly.



**Figure:** Class encapsulates data members as well as member functions. Public member functions acts as interface between class and user/client of that class, providing some communication between these two entities by enabling the user to retrieve and modify the private data members

## Objects

After defining a user defined data type, when you want to create a variable of that data type the variable is called as object. An object is also called as an instance of a class. Here is the syntax of declaring objects:

```
classname object1,object2,...,objectn
```

**For example,**

```
student s1, s2;
```

The declaration of an object is similar to that of a variable of any basic type. Here student is a class and s1 and s2 are objects of class student. Objects can also be defined by placing their name immediately after the closing brace, as we do in case of structures as follows:

```
class student
{
    .....
    .....
    .....
}
s1, s2;
```

Remember that all the public member functions can be used with an object of that class. The standard form of calling member function using an object of the class is as follows:

```
object-name.member-function-name(arguments);
```

Note that after object name, to access member function dot (.) operator is used. For example, consider the student class, which has an object s1. The member function getdata() can be called as follows:

```
s1.getdata(111, "Jitendra", 60.07);
```

Here 111 will be assigned to rollno, "Jitendra" will be assigned to name and 30000 will be assigned to percent. The following statement will be illegal in C++ because rollno is private data member of the class.

```
s1.rollno = 111;
```

The member functions of the class have some characteristics, those are as follows:

- a)** More than one class can have function with same name.
- b)** Member functions can access the private data of the class which non- member function cannot.
- c)** A member function can call another member function directly, without using the dot (.) operator.

Considering all the features we have discussed up to now, we will see a simple C++ program to read and display student details using class.

```
#include<iostream.h>
#include<conio.h>
class student
{
    int rollno;
    char name[10];
    float percent; //data members will be treated as private by default
public:
    void getdata(void) //member function getdata declared and defined
    {
        cout<<"enter the student number:";
        cin>>rollno;
        cout<<"enter the name of student:";
        cin>>name;
        cout<<"enter the percent:";
        cin>>percent;
    }
    void putdata(void) //member function putdata declared and defined
    {
```

```
cout << "\nstudent number is:\t" << rollno;  
cout << "\nstudent name is:\t" << name;  
cout << "\npercent is:\t" << percent;  
}  
};// end of class  
void main()  
{  
clrscr();  
student s; //s is declared as an object of class student  
s.getdata(); //getdata function is called through s object  
s.putdata(); //putdata function is called through s object  
getch();  
}
```

### **Output:**

enter the student number:1

enter the name of student:abc

enter the percent:88.88

student number is: 1

student name is: abc

percent is: 88.88

The above example is changed some way to use two objects and the putdata member function is defined outside the class.

```
#include <iostream.h>  
#include <conio.h>  
#include <string.h>  
class student  
{
```

```
//data members will be treated as private by default

int rollno;

char name[20];

float percent;

public:

//member function getdata declared and defined

void getdata(int id, char n[], float p)

{

rollno = id;

strcpy(name, n);

percent = p;

}

//member function putdata declared

void putdata(void);

}; // end of class declaration

//member function putdata defined

void student :: putdata(void)

{

cout<<"student id is: "<< rollno << endl;

cout<<"student name is: "<< name << endl;

cout<<"student percent is: "<< percent << endl;

}

int main(void)

{

//s1 and s2 objects of class student are created

student s1, s2;

clrscr();
```

```
//getdata function is called through s1 object
s1.getdata(101, "ABC", 66.77);
//getdata function is called through s2 object
s2.getdata(102, "XYZ", 77.66);
cout<<"The value of data members using s1 object \n";
s1.putdata();
cout<<"\nThe value of data members using s2 object\n";
s2.putdata();
getch();
return(0);
} // end of main
```

### **Output:**

The value of data members using s1 object

student id is: 101

student name is: ABC

student percent is: 66.76

The value of data members using s2 object

student id is: 102

student name is: XYZ

student percent is: 77.66

In the given program, we have declared a class student. In main() function we declared two objects s1 and s2 of type student. Remember when we define an object of class, necessary memory space is allocated to an object. Then we have called getdata() function through both objects s1 and s2 with arguments. When getdata() function is called, the arguments will be stored in data members of the class. Then to print the values of data members we have called putdata() function through both objects s1 and s2.

### **To make outside member functions as inline**

As we have seen that we can define a member function outside the class and if that

function is very small then C++ provides a feature of making that function inline. To make member function as inline, just put the keyword inline before defining that function. **For example,**

```
class student
{
    int rollno;
    char name[20];
    float percent;
public:
    void getdata(int, char [], float);
};

inline void student :: getdata(int id, char n[], float p)
{
    rollno = id;
    strcpy(name, n);
    percent = p;
}
```

## Public data members

Up to now we have seen that normally we declare data members as private in the class. It does not mean that we cannot declare data member as public. We can declare data members as public and those members can be accessed directly by the objects of that class. **For example,**

```
class abc
{
private:
    int a;
public:
    int b;
    int c;
};

main()
{
    abc x;
    x.b = 10; // valid statement
    x.c = 15; // valid statement
    x.a=10; // invalid statement because a is private
}
```

## Nested member functions

Up to now we have seen that all the public member functions can be called through the objects of that class. But that is not the case, you can also call a public member function from another public member function. This is called as nesting of member functions. The following program will illustrate this feature:

```
#include <iostream.h>
#include <conio.h>
class factorial
{
private:
int x;
int facto;
public:
void fact(void)
{
facto=1;
for (int i=1; i<=x; ++i)
facto *= i;
}
void calculate_fact(int val1)
{
x = val1;
fact(); //member function called inside other member
function
}
void show_fact(void)
{
```

```
cout<<"\nFactorial of " << x << " is: " << facto;  
}  
};  
  
int main(void)  
{  
    factorial f1, f2;  
  
    clrscr();  
  
    f1.calculate_fact(5);  
  
    f2.calculate_fact(7);  
  
    f1.show_fact();  
  
    f2.show_fact();  
  
    getch();  
  
    return(0);  
}
```

### **Output:**

Factorial of 5 is 120

Factorial of 7 is 5040

The above program is of calculating factorial of data member. We have defined two objects f1 and f2 of type factorial in main() function. Then we call a member function calculate\_fact() with arguments 5 and 7 through objects f1 and f2 respectively. When a member function calculate\_fact is called, it will store the argument in data member x and call another member function fact() from that member function. So this feature or technique is called as nesting of member functions.

## Private member functions and public data members

As we can declare data members as both private and public, we can also declare member functions as both private and public. When we declare member functions as private then you cannot define it outside the class and you cannot access that member functions through objects of that class. The private member functions can be called in public member functions only. **For example,**

```
#include <iostream.h>
#include <conio.h>
class maximum
{
private:
    int x, y;
    int max(void) //private member function
    {
        return(x > y ? x : y);
    }
public:
    int z; // public data member
    void calculate_max(int val1, int val2)
    {
        x = val1;
        y = val2;
        z=max(); //call private member function and store
        //return value in public data member
    }
};
int main(void)
```

```
{  
maximum m1, m2;  
clrscr();  
m1.calculate_max(25, 34);  
m2.calculate_max(10, 7);  
cout<<“Maximum using object m1: ” << m1.z << endl;  
cout<<“Maximum using object m2: ” << m2.z;  
//print the value of z directly through m1 object  
getch();  
return(0);  
}
```

### **Output:**

Maximum using object m1: 34

Maximum using object m2: 10

As given in the above program we have declared one member function as private and one data member as public. First we have created two objects m1 and m2 of type maximum. Then we have called a public member function calculate\_max with two arguments 25 and 34 through m1 object. When public member function calculate\_max() is called it will assign arguments (25 and 34) to the x and y respectively. Then we have called a private member function max() and return value of that function will be stored in public data member z.

Note that if you write a statement like m1.max() then it will cause an error by C++ compiler because max() function is private to the class. Then we have called and printed the value of public data member z directly through an object of the class because variable z is public data member of class maximum.

## Array as data member of a class

We can also declare an array as data member of a class. One more feature is added to C++ that at the time of declaration of an array, you need not to specify the size of an array. At runtime you can specify or you can allocate the value of any variable as size of the array. **For example,**

```
#include <iostream.h>
#include <conio.h>
class sort_array
{
private:
int size;
int *arr; //a pointer to array as data member
public:
void getdata(int);
void sort(void);
void show_array(void);
};
void sort_array :: getdata(int n)
{
size=n;
arr = new int[size];//dynamic initialization of an array of size
for(int i=0; i<size; ++i)
{
cout << "Enter value- " << i+1 << " ";
cin >>arr[i];
}
}
```

```

void sort_array :: sort(void)
{
    for( int i=0; i<size; ++i)
        for(int j=0; j<size; ++j)
            if(arr[i] < arr[j])
            {
                arr[i]+=arr[j]; // if condition is true then
                arr[j]=arr[i]-arr[j]; // swap out the values of
                arr[i]-=arr[j]; // arr[i] and arr[j]
            }
    }

void sort_array :: show_array()
{
    for(int i=0;i<size; ++i)
        cout<< arr[i] << " ";
}

int main(void)
{
    sort_array s1;
    clrscr();
    s1.getdata(10);
    s1.sort();
    s1.show_array();
    getch();
    return(0);
}

```

**Output:**

Enter value-1 10

Enter value-2 20

Enter value-3 31

Enter value-4 3

Enter value-5 -25

Enter value-6 89

Enter value-7 45

Enter value-8 17

Enter value-9 34

Enter value-10 12

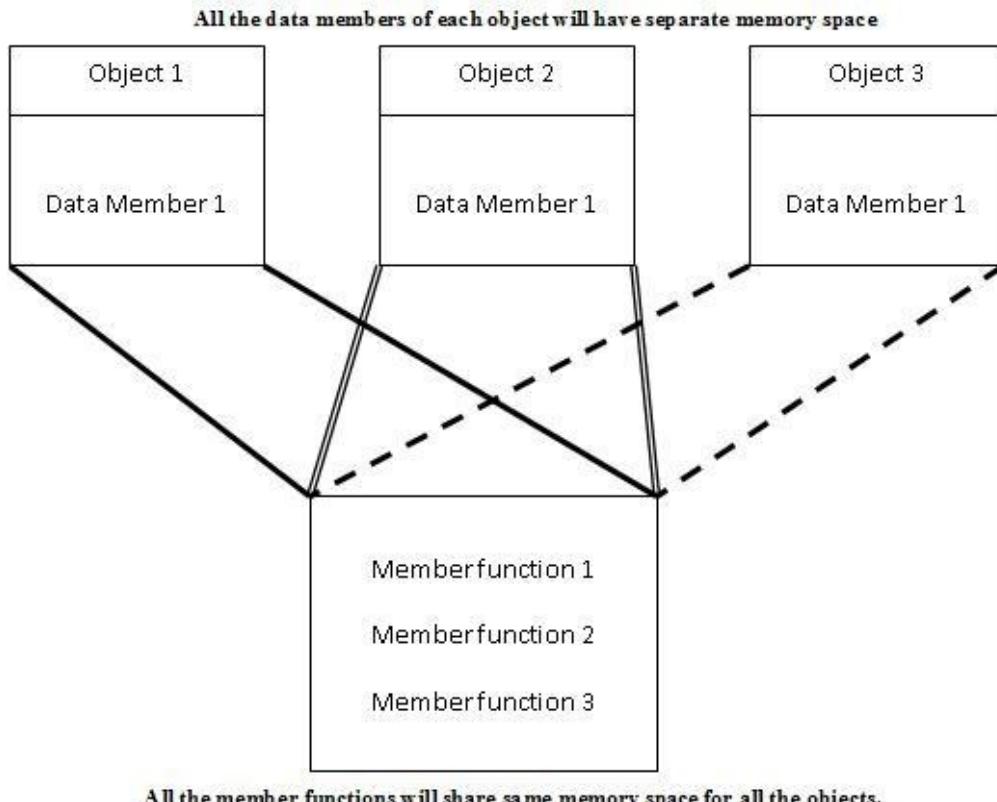
-25 3 10 12 17 20 31 34 45 89

In the above given program we have declared a pointer to integer as arr in private section of class sort\_array. We have not specified the size of an array.

In main() function we have created an object s1 of type sort\_array. Then we have called a function getdata() with argument 10 through s1 object. When a function getdata() is called the argument 10 will be assigned to data member size and at runtime an array arr of size size declared using new operator. After that we got input of 10 values from the user, these values will be stored in an array arr. Then we have called a member function sort() through s1 object. And then we have called a member function show\_array() through s1 object to show the values of an array arr.

## Memory allocation for objects

The memory space for objects is allocated when objects are created, not when class is created. Memory space for member functions will be allocated only once at the time of class creation. Memory space for data members will be allocated at the time of object creation. Each object will have its own data members so for all data members of each object will have separate memory space.



**[Figure: Memory allocation of objects]**

As you can see in above figure the memory for data members of each objects will be created when objects are defined and memory for member functions for all objects will be created when functions are defined at the creation of class.

## Static data members and member functions

A class can contain static members, either data or functions. Static data members of a class are also known as “class variables”, because there is only one unique value for all the objects of that same class. Their content is not different from one object of this class to another. Both data and function members of a class can be declared static.

### Static data members

A class' variable can be declared static. When data members of a class is declared static they are defined once and shared between all objects. A static member variable has some special characteristics, which are as follows:

- a) If a member of a class is declared as static then when an object of that class is created, the member variable will be initialized to zero (0). No other initialization is permitted.
- b) It must be initialised outside the class declaration using scope resolution operator.
- c) Only one copy of a static data member is created for the entire class and is shared by all the objects of that class.
- d) It is visible only within a class but its lifetime is the entire program.

The lifetime of a class's static member variable is the lifetime of the program. This means that a class's static member variables come into existence before any instances of the class are created.

#### For Example:

```
#include <iostream.h>
#include <conio.h>
class shared
{
public:
    static int a;
} ;
int shared::a; // define a
int main()
```

```
{  
// initialize a before creating any objects  
clrscr();  
shared::a = 99;// assigning value to a using class name  
cout << "This is initial value of a: " << shared::a;  
cout << "\n";  
shared x;  
cout << "This is x.a: " << x.a;  
getch();  
return 0;  
}
```

### Output:

This is initial value of a: 99

This is x.a: 99

Static variables are normally used to maintain values common to the entire class. Note that static member declaration is in class but a static member must be define outside the class using scope resolution operator (:) because the static data members are stored separately rather than as a part of an object. It can be accessed separately without use of any object. Here is an example to count for the objects created out of Counter class.

### Example: Private static variable

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
class Counter
```

```
{
```

```
private:
```

```
static int count;
```

```
public:
```

```
void countobject()
```

```
{  
    count++;  
}  
  
void Counter::display()  
{  
    cout<<count;  
}  
};  
  
int Counter::count;  
  
void main(void)  
{  
    clrscr();  
  
    Counter o1,o2,o3,o4;  
  
    cout << "Objects in existence: ";  
  
    o1.countobject();  
  
    o2.countobject();  
  
    o3.countobject();  
  
    o4.countobject();  
  
    o1.display();  
  
    o2.display();  
  
    o3.display();  
  
    o4.display();  
  
    getch();  
}
```

### **Output:**

Objects in existence: 4  
Objects in existence: 4

Objects in existence: 4

Objects in existence: 4

You can notice the output of the above program that all four objects display the same value for the static variable count. Also notice that the static variable is inside the private section so it can only be read by the class name. Below is an example in which static variable is in public section and accessible by the objects also.

**Example:** Public static variable

```
#include <iostream.h>
#include <conio.h>
class vector
{
public:
    int x,y;
    static int count;
    void countvector ()
    {
        x=y=0;
        x++;
        y++;
        count++;
    }
};

int vector::count = 0;
int main ()
{
    clrscr();
    cout<<"Initially number of vector objects:"<<vector::count<<endl;
```

```

vector a;

a.countvector();

cout<<“Vector Count after creating first object:”<<a.count<<endl;

cout << “Data of vector a:” << a.x <<” ” <<a.y<< endl;

vector b;

b.countvector();

cout<< “Vector Count after creating second object:” << b.count << endl;

cout << “Data of vector b:” << b.x <<” ” <<b.y<< endl;

cout << “Total number of vector objects:” << vector::count<<endl;

return 0;

}

```

### Output:

Initially number of vector objects:0

Vector Count after creating first object:1

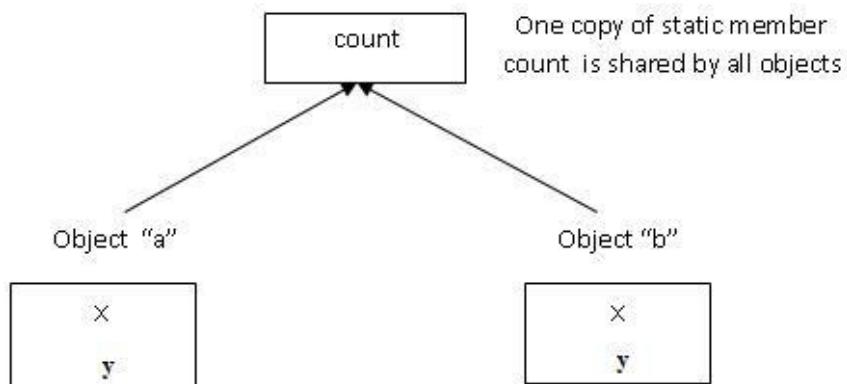
Data of vector a:1 1

Vector Count after creating second object:2

Data of vector b:1 1

Total number of vector objects:2

In above example output we can notice that only static variable value (like count) is shared between objects but not the non-static variable values (like x and y). So count is incremented twice while x and y remains 1 both objects.



**[Memory allocation for static members of objects]**

## Initializing static data members

You cannot initialize static data member in the member functions, or constructors. Hence, write the initialization outside of the class definition.

Notice that the word static is not used in initialization, like the following statement:

```
int vector::count = 0;
```

In fact, static members have the same properties as global variables but their scope is limited within the class. For that reason, and to avoid them to be declared several times, we can only include the prototype (its declaration) in the class declaration but not its definition (its initialization). In order to initialize a static data-member we must include a formal definition outside the class, in the global scope, as in the previous example.

Because it is a unique variable value for all the objects of the same class, it can be referred to as a member of any object of that class or even directly by the class name (of course this is only valid for static members).

## Static member functions

We can also declare member functions as static like data members. To declare a member function as static you have to consider following properties of static member function.

- a)** A static member function can access only other static data members or static member functions declared in the same class.
- b)** A static member function can be called using the class name not the object name of that class.

Syntax for declaring static member functions:

```
static <return type><function name>(<parameter list>);
```

Let's see an example of how to use static data members and static member functions in C++.

```
#include <iostream.h>
#include <conio.h>
class counter
{
```

```
private:
int obj_no;
static int count;

public:
void set_obj_no(void);
void show_obj_no(void);
static void show_count(void);

};

int counter :: count;

void counter :: set_obj_no(void)
{
    obj_no = ++count;
}

void counter :: show_obj_no(void)
{
    cout<<“Object number is “<< obj_no << endl;
}

void counter :: show_count(void)
{
    cout<<“The value of count is “<< count << endl;
}

int main(void)
{
    counter c1, c2;
    clrscr();
    cout<<“Value of count before initialization”<< endl;
    counter :: show_count();
```

```
c1.set_obj_no();
c2.set_obj_no();
cout<<“Value of count after setting two object no.”<< endl;
counter :: show_count();
counter c3;
c3.set_obj_no();
cout<<“Value of count after setting third object no.”<< endl;
counter :: show_count();
cout<< endl;
cout<<“The value of all obj_no....”<< endl;
c1.show_obj_no();
c2.show_obj_no();
c3.show_obj_no();
getch();
return(0);
}
```

### **Output:**

Value of count before initialization

The value of count is 0

Value of count after setting two object no.

The value of count is 2

Value of count after setting third object no.

The value of count is 3

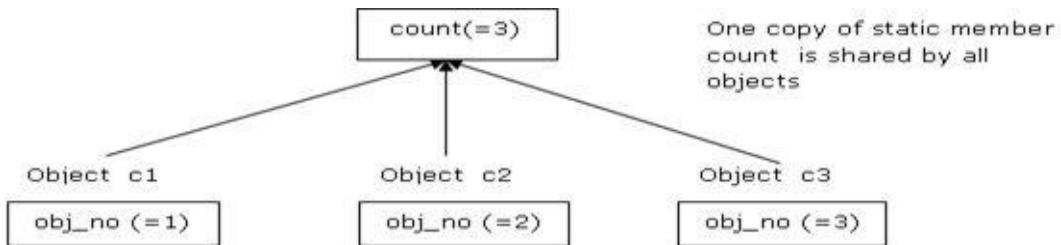
The value of all obj\_no....

Object number is 1

Object number is 2

Object number is 3

As given in the above program first we have declared a data member count as static in class counter declaration, and a member function show\_count() as static in class counter. The data member count is declared as private and a member function



**[Figure: Sharing of a static data member]**

show\_count() is declared as public in class counter. Note that we have defined count outside the class using scope resolution operator (::). In main function first we have declared two objects c1 and c2 of type counter. When we defined a static data member count, the value of count will be automatically initialize to zero (0). A static member function can access only static data members and static member functions. So in static member functions show\_count() we can access only count data because in the class count is only a static data. We cannot access obj\_no in show\_count() function. **Note that we can access static member function directly with the class name instead of object name of that class.** When you use a static member function with class name, you have to use scope resolution operator instead of dot (.) operator. As you can see in the output, the variable count will use single memory space for all the objects.

The static member functions of a class are callable before any instances of the class are created. This means that the static member functions of a class can access the class's static member variables before any instances of the class are defined in memory. This gives you the ability to create very specialized setup routines for class objects.

## Array of objects

As we have seen that we can declare an array of any built in type but in C++ you can also declare an array of variables(i.e. objects) of type class. This array of variables called as an array of objects. **For example**, consider the student class,

```
class student
{
    int rollno;
    char name[20];
    float percent;
public:
    void getdata(int id, char n[20], float p);
    void putdata(void);
};
```

Here the identifier student is user defined data type and we can create an array of objects of type student as follows:

```
student s[4];
```

Here we have created four objects s[0], s[1], s[2] and s[3] of type student. Now if you want to access any member function through object as given below,

```
s[0].putdata();
```

Now let's see a program of an array of objects.

```
#include <iostream.h>
```

```
#include <conio.h>
```

```
#include <string.h>
```

```
class student
```

```
{
```

```
private:
```

```
int rollno;
```

```
char name[20];
float percent;
public:
void getdata(void);
void putdata(void);
};

void student :: getdata(void)
{
cout<<“Enter student id “;
cin>> rollno;
cout<<“Enter student name “;
cin>> name;
cout<<“Enter student percent “;
cin>> percent;
}

void student :: putdata(void)
{
cout<<“student id is: “<< rollno << endl;
cout<<“student name is: “<< name << endl;
cout<<“student percent is: “<< percent << endl;
}

int main(void)
{
student s[3];
clrscr();
cout<<” Input data of three students” << endl;
for(int i=0;i<3; ++i)
```

```
{  
    s[i].getdata();  
    cout<< endl;  
}  
  
cout<<" Information of three students" << endl;  
  
for(i=0;i<3; ++i)  
{  
    s[i].putdata();  
    cout<<endl;  
}  
  
getch();  
return(0);  
}
```

### **Output:**

Input data of three students

Enter student id 11

Enter student name John

Enter student percent 5000

Enter student id 12

Enter student name Adams

Enter student percent 7000

Enter student id 13

Enter student name Jack

Enter student percent 10000

Information of three students

student id is: 11

student name is: John

student percent is: 5000

student id is: 12

student name is: Adams

student percent is: 7000

student id is: 13

student name is: Jack

student percent is: 10000

As you can see in the above program we have defined an array of three objects named s[0], s[1] and s[2] of type student. We access getdata() function through these three objects in for loop and then we access putdata() function to print all the data of these three objects in another for loop.

**Example:** Arrays of objects of person class

```
class person
{
    char name[30];
    int account;
    float balance;
public:
    void getdata(void);
    void display(void);
};

void person::getdata(void)
{
    cout << "Enter name: ";
    cin >> name;
    cout << "Enter account no: ";
    cin >> account;
```

```
cout << "Enter balance amount: ";
cin >> balance;
}

void person::display(void)
{
    cout << "\nName: " << name;
    cout << "\nAccount: " << account;
    cout << "\nBalance: " << balance;
}

main()
{
    person p[3]; //array of object of type person
    for(int i=0;i<3;i++)
    {
        cout << "\nGet details of person " << (i+1) << ": \n";
        p[i].getdata();
    }
    cout<<"\n";
    for(int i=0;i<3;i++)
    {
        cout << "\nDetails of person " << (i+1) << ": \n";
        p[i].display();
    }
    cout<<"\n";
    getch();
}
```

## Passing objects as function arguments

You can pass the objects as arguments and return object from the called function.

As we have seen that the arguments can be passed in two ways one is call by value and another is call by reference. In case of objects, similarly we can pass arguments in two ways. Let's see an example of passing objects as argument and returning objects from the function.

```
#include <iostream.h>
#include <conio.h>

class distance
{
private:
    int feet;
    float inches;
public:
    void get_distance(int, float);
    distance add_distance(distance);
    void put_distance(void);
};

void distance :: get_distance(int f, float i)
{
    feet = f;
    inches = i;
}

distance distance :: add_distance(distance d2)
{
    distance d;
    d.feet = feet + d2.feet; // feet is data of object d1.
```

```

d.inches = inches + d2.inches; // inches is data of object d1.

if(d.inches >= 12.0)

{
    d.inches -= 12.0;

    d.feet++;

}

return d; //Return type of function is object of distance class
}

void distance :: put_distance(void)

{
    cout<<"Feet "<< feet <<" -inches " << inches << endl;
}

int main(void)

{
    distance d1,d2,d3;

    clrscr();

    d1.get_distance(11, 9.5);

    d2.get_distance(5, 4.5);

    d3 =d1.add_distance(d2);

    d1.put_distance();

    d2.put_distance();

    d3.put_distance();

    getch();

    return(0);
}

```

## Output:

Feet 11 – inches 9.5

Feet 5 – inches 4.5

Feet 17 – inches 2

As given in the above program first we have declared a member functions `add_distance()` with an argument of object of type `distance` and that function will return an object of type `distance`. In `main()` function first we created three objects `d1`, `d2` and `d3` of type `distance`. Then we called `get_distance()` with arguments through `d1` and `d2` objects. Then we have called `add_distance()` function through `d1` object with argument `d2` and that function will return an object of type `distance`, that will be stored in object `d3`. When the function `add_distance()` will be called, in that function we are adding two data members feet of objects `d1` and `d2` and two data members inches of objects `d1` and `d2` into another data members feet and inches of object `d`. Note that this function we have called through `d1` object so there is no need of specifying object name (`d1`) in prefix of feet and inches data members. We have passed `d2` object so to access the members of `d2` we must write object name (`d2`) in front of data members. The addition of data members of `d1` and `d2` objects will be stored in the data members of object `d`. And then that function will return object `d` that will be assigned to object `d3`. Then in `main()` function we have called `put_distance()` function through all three objects `d1`, `d2` and `d3` to print the data members of all three objects.

There is another way to solve or to do the given program with more flexibility that is there is no need of defining object of type `distance` which we have defined in `add_distance()` function with name `d`. this can be done with some changes in defining `add_distance()` function in the given program. The function `add_distance()` should be defined as follows:

```
void distance :: add_distance(distance d1, distance d2)
{
    feet = d1.feet + d2.feet;
    inches = d1.inches + d2.inches;
    if(inches >= 12.0)
    {
        inches -= 12.0;
        feet++;
    }
}
```

```
}
```

```
}
```

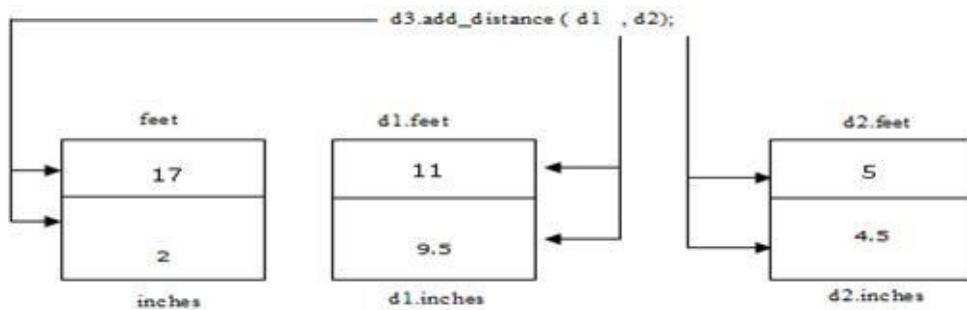
Note that here we are passing two objects d1 and d2 as arguments. And we are calling this function as follows from main() function:

```
d3.add_distance(d1, d2);
```

Instead of

```
d3 = d1.add_distance(d2);
```

Here we are calling add\_distance() function through d3 object with two arguments d1 and d2 so in the function we can access data members of d3 object, so the addition of data members of d1 and d2 will be stored directly to the data members of d3 object. The idea of this function will be more clear through the following figure.



[Figure: The idea of add\_distance() function]

## Objects as return type

A function can not only receive objects as arguments but can also return objects. The example below shows how a function returns objects to another function.

```
// Returning objects from a function.

#include <iostream.h>

#include <conio.h>

class myclass

{

int i;

public:

void set_i(int n) { i=n; }

int get_i() { return i; }

};

myclass f(); // Funciton Declaration, function f() returns object of
type myclass

int main()

{

myclass o;

o = f(); //Function call

cout << o.get_i() << "\n";

return 0;

}

myclass f()// Funciton Definition

{

myclass x;

x.set_i(1);

return x;
```

}

**Output:**

1

## Friend functions

In the real life the friend member means a member who is close to you but not your family member. In same way we can also define a function as friend in the class, which will not be the member of that particular class but that function can access even the private members of the class.

Up to now we have seen that a non-member function cannot access the private data members of that class but in some situation we need to give access to the private data member to the outside functions (or non-member functions). For this situation C++ provides the feature of friend function. A friend function can be friend of more than one class. To define a function as friend function of the class we have to write keyword “friend” in before function name at the time of declaration. The general form of friend function declaration is as follows:

```
class <class name>
{
private:
    .....
    .....
public:
    .....
    friend <return type> <function name> (arguments);
};
```

Once a function is declared as friend in the class that function must be defined outside the class, at the time of defining a friend function, we can not use scope resolution operator (::). A friend function can be declared in any number of classes.

### Friend function characteristics:

- a)** Friend function is non-member function for the class.
- b)** It is not in the scope of the class to which it has been declared as friend.
- c)** Since it is not in the scope of the class, it cannot be called using the object of that class. It can be invoked like a normal function without the help of any object.

- d)** Unlike member function, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.
- e)** It can be declared as private or public in the class without affecting its meaning.
- f)** Usually, it has the objects as arguments.

### **Example 1:** Friend Function demo

```
#include<iostream.h>
#include<conio.h>

class myclass
{
    int a, b; // note: by default a and b are private public:
public:
    void read(int i, int j)
    {
        a=i; b=j;
    }
    friend int sum(myclass x);
    // Note: sum() is not a member function of any class.
};

int sum(myclass x)
{ /* Because sum() is a friend of myclass, it can directly access
   a and b. */
    return x.a + x.b;
}

int main()
{
    myclass n;
    clrscr();
```

```
n.read(3, 4);

cout << "The addition of two nos: "<<sum(n);

//call to friend function

return 0;

}
```

**Output:**

The addition of two nos: 7

**Example 2:** Now let's see a real life example of friend function.

```
#include <iostream.h>

#include <conio.h>

class student

{

private:

int rollno;

char name[20];

float percent;

public:

void getdata(void);

friend void putdata(student);

};

void student :: getdata(void)

{

cout<<"Enter student id ";

cin>>rollno;

cout<<"Enter student name ";

cin>>name;

cout<<"Enter student percent ";
```

```
cin>>percent;  
}  
  
void putdata(student s)  
{  
    cout<<“student id “<< s.rollno << endl;  
    cout<<“student name “<< s.name << endl;  
    cout<<“student percent “<< s.percent << endl;  
}  
  
int main(void)  
{  
    student s1;  
    clrscr();  
    s1.getdata();  
    cout<<”\nCall friend function putdata() to show members of  
    s1\n\n”;  
    putdata(s1);  
    getch();  
    return(0);  
}
```

### **Output:**

```
Enter student id 1  
Enter student name Jack  
Enter student percent 10000  
Call friend function putdata() to show members of s1  
student id 1  
student name Jack  
student percent 10000
```

As given in the program we have define a member function putdata() as friend in class student. When we define that function outside the class, there is no need of specifying class name and scope resolution operator (>::). When you call the friend function putdata() from the main function, there is no need of specifying any object name or dot (.) operator. You can call putdata() function directly with its name. Now let's see how a friend function is friend of two classes. To understand this concept let's take an example of earlier class student with two more classes manager and worker.

**Example 3:** Use of friend function in more than one class

```
#include <iostream.h>
#include <conio.h>
#include <string.h>

class principal;
class professor;
class employee

{
private:
    int id;
    char name[20];
    int experience;

public:
    void getdata(void)
    {
        cout<<"Enter employee id ";
        cin>>id;
        cout<<"Enter employee name ";
        cin>>name;
        cout<<"Enter employee salary ";
        cin>>experience;
```

```
}

friend void compare(employee, principal, professor);

};

class principal

{

private:

int id;

char name[20];

int experience;

public:

friend void compare(employee, principal, professor);

};

class professor

{

private:

int id;

char name[20];

int experience;

public:

friend void compare(employee, principal, professor);

};

void compare(employee e, principal pl, professor pr)

{

if(e.experience >=15)

{

pl.id = e.id;

strcpy(pl.name, e.name);
```

```
pl.experience = e.experience;

cout<<"\nThe employee is Principal \n";
cout<<"Principal id: "<< pl.id << endl;
cout<<"Principal name: "<< pl.name << endl;
cout<<"Principal experience:"<< pl.experience << endl;
}

Else

{
pr.id = e.id;
strcpy(pr.name, e.name);
pr.experience = e.experience;
cout<<"\nThe employee is Professor \n";
cout<<"Professor id: "<< pr.id << endl;
cout<<"Professor name: "<< pr.name << endl;
cout<<"Professor experience:"<< pr.experience << endl;
}

int main(void)

{
employee e1, e2;
principal p1;
professor pr1;
clrscr();
e1.getdata();
cout<< endl;
e2.getdata();
compare (e1, p1, pr1);
```

```
compare (e2, p1, pr1);  
getch();  
return(0);  
}
```

**Output:**

```
Enter employee id 1  
Enter employee name RSah  
Enter employee salary 23  
Enter employee id 2  
Enter employee name HGRajput  
Enter employee salary 12  
The employee is Principal  
Principal id: 1  
Principal name: RSah  
Principal experience:23  
The employee is Professor  
Professor id: 2  
Professor name: HGRajput  
Professor experience:12
```

As you can see in above program first we have declared two classes professor and principal as forward declaration this means that both classes will be defined afterwards. Then we have defined student class with a member function compare() as friend with three arguments as an object of student, an object of principal and an object of professor. In any firm everybody are student of that firm, we can differentiate between principal and professor through their salaries. So through a friend function compare() we can differentiate that whether the student is principal or professor.

In declaration of class principal and professor we have defined same compare() function with same arguments as friend so this friend function can access any private data of all the three classes.

As we have discussed earlier that there is no need of scope resolution operator or any class name at the time of defining friend function, this you can see in the program that a compare() function need not scope resolution operator (::) or any class name, it can be defined as normal function. When you call a friend function from main function, there is no need of any object name or a dot operator, we can call a friend functions as normal functions.

A complex number consists of two parts – real part and an imaginary part written as  $x+ij$ , where  $x$  and  $y$  are real values.  $j$  is imaginary quantity .We can make use of friend function for the arithmetic operations with complex numbers as given below:

**Example 4:** Using friend function for the addition of two complex numbers

```
/* Friend functions*/  
class complex //a+jb  
{  
    float a;  
    float b;  
public:  
    void input(float real, float img)  
    {  
        a=real;  
        b=img;  
    }  
    friend complex sum(complex, complex);  
    void display(complex);  
};  
complex sum(complex c1, complex c2)  
{  
    complex c3;  
    c3.a=c1.a+c2.a;
```

```

c3.b=c1.b+c2.b;

return (c3);

}

void complex::display(complex c)

{

cout << c.a << “+j” << c.b <<”\n”;

}

main()

{

complex A,B,C;

//object of type complex

A.input(3.0,5.5);

B.input(2.5,1.0);

C=sum(A,B);

cout << “\n A = “; A.display(A);

cout << “\n B = “; B.display(B);

cout << “\n C = “; C.display(C);

cout<<”\n”;

getche();

}

```

**Output:**

A = 3+j5.5

B = 2.5+j1

C = 5.5+j6.5

## Friend Class

We can also declare all the member functions as friend of another class then that class will be called as friend class. The friend class can be declared as follows:

```
class a
{
private:
.....
.....
public:
.....
friend class b;
};
```

Here all the member functions of class b are friend of class a. A member function of one class can be friend function of more than one class, in such case a member function should be defined using the scope resolution operator (::) in another classes. **For example,**

```
class a
{
private:
.....
.....
public:
.....
void abc();
};

class b
{
private:
```

```
.....  
.....  
public:  
.....  
friend void a :: abc(); // abc() is friend of class b.  
};
```

A function abc() is declared in class a and if you want to declare or use that function in class b then declare that function as friend in class b using scope resolution operator (::) as shown in the given example.

**Example:** Implementation of friend **class**

```
#include<iostream.h>  
#include<conio.h>  
  
class A  
  
{  
  
int n1,n2;  
  
public:  
  
void setA(int a,int b)  
  
{  
  
n1=a;  
  
n2=b;  
}  
  
friend class B;//means all functions of B class can use n1 and n2  
};  
  
class B  
  
{  
  
public:  
  
int add(A a) //add is now friend function of A
```

```
{  
    return a.n1+a.n2;  
}  
};  
  
void main()  
{  
    clrscr();  
    A objA;  
    B objB;  
    objA.setA(23,23);  
    cout<<"\nsum of numbers in A is:"<<objB.add(objA);  
    getch();  
}
```

**Output:**

sum of numbers in A is:46

## Pointers to Objects

We know that pointers can be used to access data. Since a class is also a data type, a pointer can point to an object created by a class. The general form will be

```
class_name *pointer;
```

Object pointers are useful in creating objects at run time. The pointers can then be used to access the public members of an object. The general form is as follows:

```
pointer->memberFunction()
```

where memberFunction() is a method/function in the class.

### Example:

```
#include<iostream.h>

class item

{

int code;

float price;

public:

void getdata(int a, int b)

{

code=a;

price=b;

}

void show(void)

{

cout<<“Code:” << code<<”\n”<<“Price:” << price<<”\n”;

}

};

main()

{
```

```
item A; //object created

item *Ptr=&A; // ptr declared and initialised with address of A

Ptr->getdata(11, 2.99);

Ptr->show();

// This is same as

// A.getedata(11,2.99);

// A.show();

return 0;

}
```

### **Output:**

Code:11

Price:2

It is to be noted from this example that \*ptr is an alias of object A.

Objects can also be created using pointers and new operator as follows

```
class_name *Ptr= new class_name
```

### **e.g.**

```
item *Ptr= new item;
```

This statement allocates enough memory space for the data members in the object structure and assigns the address of the memory space to Ptr. Ptr can now be used to refer to the members, e.g.

```
Ptr->show();
```

If the class has a constructor with parameters, then parameters should be provided when creating the object. An array of objects can also be created using pointers. The following example creates an array of 10 objects of item.

```
item *Ptr= new item[10];
```

### **Example:** Pointer to Array of objects

```
#include<iostream.h>
```

```
class item
```

```
{  
int code;  
float price;  
public:  
void getdata(int a, int b)  
{  
code=a;  
price=b;  
}  
void show(void)  
{  
cout<<“Code:” << code<<”\n”<<“Price:” << price<<”\n”;  
}  
};  
const int size = 2;  
main()  
{  
item *Ptr= new item[size];  
item *d = Ptr;  
int i,x;  
float y;  
for(i=0; i<size; i++)  
{  
cout<<“Input code and price for item” << i+1;  
cin>>x>>y;  
Ptr->getdata(x,y);  
Ptr++;
```

```
}

for(i=0; i<size; i++)

{

cout<<"Item: " << i+1 <<"\n";

d->show();

d++;

}

return 0;

}
```

**Output:**

Input code and price for item11

22

Input code and price for item22

33

Item: 1

Code:1

Price:22

Item: 2

Code:2

Price:33

## Pointer to members of class

We can use the members of a class through pointer in C++. C++ provides three operators to declare and access a pointer to a member those are as follows:

::\*

.\*

->\*

These three operators are also called as dereferencing operators. The idea of these three operators will be more clear through the following program.

```
#include <iostream.h>
#include <conio.h>
class complex
{
private:
float x;
float y;
public:
void getdata(float x1, float y1)
{
x = x1;
y = y1;
}
complex add(complex com2)
{
complex ts;
float complex ::* ptr_x = &complex :: x;
//ptr_x is pointer to member x
float complex ::* ptr_y = &complex :: y;
```

```

//ptr_y is pointer to member y

complex *ptr_obj = &com2;

/*ptr_obj is pointer to object com2

ts.x = x + com2.*ptr_x;

//Accessing value of x using pointer to member and .* operator

ts.y = y + ptr_obj->*ptr_y;

//Accessing value of y using pointer to object and ->* operator

return ts;

}

void putdata(void)

{

cout<< x << " + i" << y << endl;

}

};

int main(void)

{

complex c1, c2, c3;

clrscr();

void (complex :: *ptr_fun)(float, float) = &complex :: getdata;

// creating pointer to member function getdata of class complex

(c1.*ptr_fun)(5.2, 3.7);

// calling the function thru function pointer using object c1

complex *ptr_obj = &c2;

(ptr_obj->*ptr_fun)(3.9, 7.5);

// calling the function thru pointer to object

c3 = c1.add(c2);

cout<<"First complex number: ";

```

```

c1.putdata();

cout<<“Second complex number: “;

c2.putdata();

cout<<”\nAddition of two complex numbers: “;

c3.putdata();

getch();

return(0);

}

```

**Output:**

First complex number: 5.2 + i3.7

Second complex number: 3.9 + i7.5

Addition of two complex numbers: 9.1 + i11.2

As you can see in the above program first we have declared a class named complex, in that class we have declared two private data members x and y of type float and three public member functions getdata(), putdata() and add(). For tsorary don't consider how these functions works we will see those functions when they will called in main() function.

In main() function first we have declared three objects c1, c2 and c3 of type complex. Then we have written following statement in main() function.

```
void (complex :: *ptr_fun)(float, float) = &complex :: getdata;
```

In this statement we have defined a pointer to a member function getdata() of class complex using pointer to member function operator (::\*). A pointer to a member function getdata() is declared as ptr\_fun. So wherever you want to call getdata() function, you can also call with pointer to a member function ptr\_fun.

```
(c1.*ptr_fun)(5.2, 3.7);
```

In the above statement of main() function we are calling a member function through pointer to member function with object c1. In that function we have passed two parameters with values 5.2 and 3.7. In this statement we have called getdata() function with 5.2 and 3.7 parameters to store these parameters in data members of object c1. This concept is of how to use pointer to member function (.\*).

```
complex *ptr_obj = &c2;
```

The above statement shows you that you can also use pointer to object (`->*`) instead of object name. Here in the given statement we have define a pointer `ptr_obj` of type `complex` and assigned the address of object `c2` to the `ptr_obj`. So now wherever you want to use object `c2` you can also use `ptr_obj` with pointer to object operator (`->*`). The next statement is an example of how to use pointer to object operator.

```
(ptr_obj->*ptr_fun)(3.9, 7.5);
```

Here note that we are calling `getdata()` function through its pointer `ptr_fun` with arguments 3.9 and 7.5. Earlier we have called this function with object `c1` here we want to call with `c2`. Already we have declared a pointer to object `ptr_obj` which points to object `c2` so we can use `ptr_obj` as an object to call `getdata()` function instead of `c2`.

```
c3 = c1.add(c2);
```

In the above statement we have called `add()` function with arguments as object `c2` and that function will return the addition of data members of objects `c1` and `c2` and that object will be stored in object `c3`. Now lets see how this `add()` function works. In `add()` function we have stored the argument object `c2` as object `com2`. We have declared another tsorary object `ts` of type `complex`. Already we have seen that how to define pointer to a member function. In the same way we can also define a pointer to data members. The following two statements in `add()` function we have defined two pointers named `ptr_x` and `ptr_y` to the data members `x` and `y` respectively.

```
float complex ::* ptr_x = &complex :: x;
```

```
float complex ::* ptr_y = &complex :: y;
```

Now wherever you want to use the value of data member `x` and `y` we can also use `ptr_x` and `ptr_y` respectively. In the next statement of `add()` function is declaration of pointer to an object `com2` as `ptr_obj` which already we have seen in `main()` function. The `add()` function we have called with `c1` object in `main()` function so in `add()` function we can access the data member of object `c1` directly without using dot (.) operator.

```
ts.x = x + com2.*ptr_x; //statement 1
```

```
ts.y = y + ptr_obj->*ptr_y; //statement 2
```

In the above two statements of `add()` function, we are adding data member `x` of objects `c1` and `com2` to the data member `x` of object `ts` and same for data member `y`.

Here we can access the pointer to data members of object with object name and pointer to member operator (.\* ) as given in statement 1 and we can also access the pointer to data members of object with pointer to that object name and pointer to member operator (->\*) as given in statement 2.

Next you have a summary on how can you read some pointer and class operators (\*, &, ., ->, [ ]) that appear in the previous example:

Expression	Can be read as
$*x$	pointed by x
$\&x$	address of x
$x.y$	member y of object x
$x->y$	member y of object pointed by x
$(*x).y$	member y of object pointed by x (equivalent to the previous one)
$x[0]$	first object pointed by x
$x[1]$	second object pointed by x
$x[n]$	(n+1)th object pointed by x

## Exercise

### True or False.

1. State that the following statement is True/False.
  - a) Structure struct provides data hiding facility.
  - b) By default the data members of class are private.
  - c) A function that outside the class can access public member of class only.
  - d) A function that outside the class can access private member of class.
  - e) A public member function can access private member and public member of class.
  - f) We cannot define a member function outside the class.
  - g) We cannot call a member function of a class from another member function of same class.
  - h) Each object of a class will have separate memory for member functions of that class.
  - i) A static member function can access only static data members of the class.
  - j) We can pass objects as arguments in member functions and also return an object.
  - k) There is no need of specifying object name or dot operator to access a friend function.
  - l) `.*` and `->*` both are pointer to member operators.

### Find Errors

2. Find out error in following statements (if any):

- a) `student :: putdata(void)`
- b) `void inline student :: putdata(void)`
- c) `int arr = new int[size];`
- d) `static void counter(void)`
- e) `time t[5] (time is class name)`

### Short Questions

3. Explain the use of following terms:

- a) class

- b)** object
- c)** private and public
- d)** inline
- e)** friend

4. Explain public data member with any example.

5. Explain characteristic for the static data members.

## Programming Exercises

6. Write a program for class student with variable roll\_no, name, sub1, sub2 and total\_marks and member function getdata() for take input from user and putdata() to print data member of that object.
7. Write above program using inline keyword for total() member function, which will return addition of sub1 and sub2 and store this total into data member total\_marks.
8. Write program explain in example 6 as array of ten students.
9. Write a program of class time with data members hour and minute and member function gettime(), showtime() and addtime(). In which member function addtime() with two arguments as object and return type void which add time of two object and assign in third object.
10. Write a program of class time explains in above in which member function addtime() with one argument of type object and object as return type.
11. Create two classes DM and DB, which store the value of distances. DM stores distances in meter and centimeters and DB in feet and inches. Write a program that can read values for the class objects and add one object of DM with another object of DB. The add() function must be a friend function. The object that stores the results may be a DM object or DB object, depending on the units in which the results are required.
12. Write a C++ program to manipulate the class account using classes and function. A user should be able to perform the following functions. a. Deposit money. b. Withdraw money, c. Calculate the interest d. Check the total balance in his account.
13. Write a C++ program to create a class template to find the maximum of two numbers.
14. Write a program in c++ to calculate the area of triangle using inline function.
15. Write a program in c++ to calculate the volume of cube, cylinder, and rectangle using

function overloading.

**16.** Write a program in c++ to calculate factorial using friend function.

## CHAPTER:7 Constructors and destructors

Up to now we have seen that in the class mostly we declared a `getdata()` function to assign values to the data members of that class. In this case when we declare an object of the class the data members are not initialized but when we call `getdata()` function with that object the data members will be initialized with the arguments of `getdata()` function C++ provides a facility of initializing the data members of an object at the time of object creation. This mechanism is called as **automatic initialization of objects**. In this mechanism a special type of member function we have to define in the class. That member function will have same name as of the class name. This function is called as constructor. C++ provides another member function called destructor, which destroys the objects when they are no longer required.

## Constructors

C++ provides a special member function called the constructor which enables an object to initialize itself when it is created. The task of constructor is to initialize the objects at the time of declaration. To initialize the objects at the time of creation, we must define a constructor in the class. The constructor is invoked whenever an object of its class is created. It is called constructor because it constructs the values of data members of the class' object. It is special in the sense that its name is same as the class name.

A constructor is declared and defined as follows

```
class class_name
{
    private data and functions
public:
    public data and functions
class_name () //constructor declared
.....
.....
};

class_name :: class_name () //constructor defined
{
    Initialization of private data;
}
```

There are some special characteristics of constructor, which are as follows:

- a)** Constructors should be declared in public section.
- b)** When objects are created, constructor is invoked(called) automatically.
- c)** Constructors don't have any return types, not even void and therefore they cannot return values.
- d)** Like other C++ functions, constructors can have default arguments.
- e)** We cannot refer to constructors' addresses.

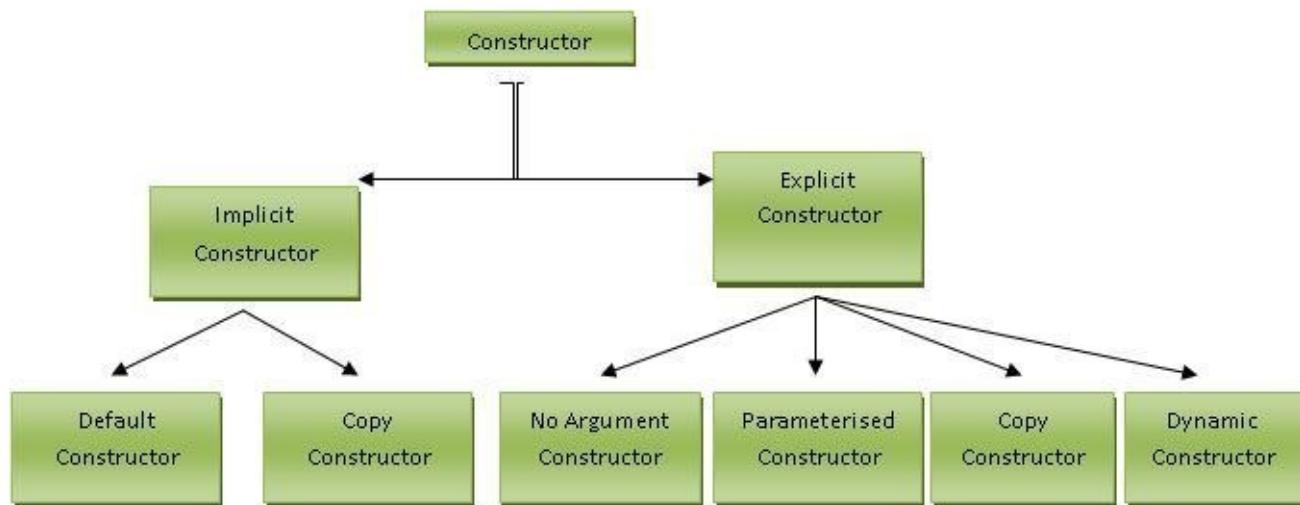
f) They make implicit calls to the operators new and delete when memory allocation is required.

There are two more characteristics, which are as follows and we will discuss these characteristics later.

a) Constructors cannot be inherited, though a derived class can call the base class constructor.

b) Constructors cannot be virtual.

There are mainly 3 types of constructor:



a) Default constructor

I) Default constructor is without arguments inside its parenthesis.

II) It is called when creating objects without passing any arguments, **for example:**

NewClass N; //Default constructor invoked implicitly

b) Argument Constructor

I) Argument Constructor accepts arguments inside its parenthesis.

II) It is called when creating objects by passing some arguments, **for example:**

NewClass N(5); //Argument constructor invoked explicitly

c) Copy constructor

I) Copy constructor is used to make copy of one object(all data members) into another object(all data members).

II) This constructor is invoked when:

Passing object as an argument to a function, **for example:**

void foo(NewClass n); //n is an object of NewClass passed as a argument

Return object from a function, **for example:**

NewClass foo(); //foo() returns an object of NewClass

Initializing an object with another object in a declaration statement

NewClass n2; //call default constructor

NewClass n1(n2); //call copy constructor

NewClass n3=n2; //call copy constructor

Now let's study the use of all the above three constructors in detail with examples.

## Default constructor

No object can be constructed without constructor. If we do not provide the constructor the compiler calls the implicit default constructor for the construction(initialization) of the object.

The default constructor is a constructor that takes no arguments or takes only default arguments. The default constructor for a class X can have either of the following forms:

X::X()

Or

X::X(const int x=0)

Default constructors allow objects to be created without passing any parameters to the constructor. For example, the declaration

Student s;

results in a Student object “s” that does not yet have a value; it is an empty object.

The default constructor usually creates an object that represents a “null” instance of the particular type the class denotes.

It is better to provide a constructor with default arguments that can serve as a default constructor or as a constructor that takes the arguments it specifies. If you do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments. Therefore, after declaring a class like this one:

```
#include <iostream.h>
#include <conio.h>
class Rectangle
{
public:
int l,b,a;
void findarea (int n, int m)
{
l=n;
b=m;
a=l*b;
}
};

int main ()
{
clrscr();
Rectangle r; //r created implicitly by default constructor
r.findarea(2,3);
cout<<"The area is:"<<r.a;
return 0;
}
```

### **Output:**

The area is:6

The compiler assumes that above class has a default constructor, so you can declare objects of this class by simply declaring them without any arguments:

```
Rectangle r;
```

In above example object declaration, eventhough no explicit constructor is defined the

compiler invokes the default constructor for the construction of the object r.

The compiler not only creates a default constructor for you if you do not specify your own, it provides three special member functions which are implicitly declared if you do not declare your own constructor. These are the:

- a)** copy constructor,
- b)** the copy assignment operator, and
- c)** the default destructor.

The copy constructor and the copy assignment operator copy all the data contained in another object to the data members of the current object.

Therefore, the two following object declarations would be correct:

```
Rectangle r (2,3);  
Rectangle r2 (r); // copy constructor (data copied from r)
```

Above program can be modified as follows:

```
#include <iostream.h>  
#include <conio.h>  
  
class Rectangle  
{  
public:  
    int l,b,a;  
  
    void findarea (int n, int m)  
    {  
        l=n;  
        b=m;  
        a=l*b;  
    }  
};  
  
int main ()
```

```
{  
clrscr();  
Rectangle r1;  
r1.findarea(2,3);  
Rectangle r2(r1); // invokes implicit copy constructor  
r2.findarea(2,3);  
cout<<“The area for object r1 is:”<<r1.a;  
cout<<”\nThe area for object r2 is:”<<r2.a;  
return 0;  
}
```

### **Output:**

The area for object r1 is:6

The area for object r2 is:6

As you can see the output for both object are same because object r2 is created implicitly by copying all the data members of the object r1.

## **Parameterized constructors**

We can also pass arguments in the constructor as like normal member function. This type of constructors is called as parameterized constructors. In practice, it is necessary to initialize various data elements of different objects with different values when they are created. C++ allows to achieve this by passing arguments to the constructor function when objects are created. These constructors are called parameterized constructors. The general form is shown below:

```
class class_name  
{  
private data and functions  
public:  
public data and functions  
class_name (parameter list); //constructor declared
```

```
.....  
.....  
};  
class_name :: class_name (parameter list) //constructor defined  
{  
Initialization of private data;  
}
```

If parameterized constructors is declared in the class then when we create an object we have to pass the arguments with that object.

As soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. Explicit constructor defined by the programmer can be argument(or Parameterised) constructor or no argument constructor. So you have to declare all objects of that class according to the constructor prototypes you have defined for the class, **for example:**

```
#include <iostream.h>  
#include <conio.h>  
class Rectangle  
{  
public:  
int l,b,a;  
Rectangle () //No argument constructor  
{  
l=b=1;  
}  
Rectangle (int n, int m)//Argument(or Parameterised) constructor  
{  
l=n;  
b=m;
```

```

}

void findarea ()
{
a=l*b;

}

};

int main ()
{
clrscr();

Rectangle r1; //invokes no argument constructor

Rectangle r2(2,3); //invokes the parameterised constructor

// or Rectangle r2=Rectangle(2,3);

r1.findarea();

r2.findarea();

cout<<"The area for r1 is:"<<r1.a;

cout<<"\nThe area for r2 is:"<<r2.a;

return 0;

}

```

**Output:**

The area for r1 is:1

The area for r2 is:6

Here we have declared two explicit constructors:

**1. No argument constructor**

```

Rectangle ()

{

l=1;

b=1;

```

```
}
```

Is no argument constructor and takes no argument as name suggests and initialize the members to value 1. And this constructor is invoked at the time of the creation of object r1 as follows:

```
Rectangle r1;//invokes no argument constructor and initialize l=b=1;
```

## 2. Argument constructor(or Parameterised constructor)

```
Rectangle (int n, int m) //invokes Argument constructor and  
initialize l=n=2, b=m=3;
```

```
{
```

```
l=n;
```

```
b=m;
```

```
}
```

takes two parameters of type int. Therefore the following object declaration would be correct:

```
Rectangle r(2,3);//invokes the parameterised constructor and  
initialize r.
```

We can pass the arguments with an object in two ways.

Call the constructor explicitly.

```
Rectangle r2=Rectangle(2,3);
```

Call the constructor implicitly.

```
Rectangle r2(2,3);
```

The implicit call is smaller and easier than explicit call.

## Copy Constructor

The sets of values of one object can be copied into the corresponding elements of another object. A copy constructor is a special constructor that can be called to copy an object. Such a constructor is called copy constructor. Copy constructor is:

- a)** a constructor function with the same name as the class
- b)** used to make deep copy(member to member copy) of objects.

There are 3 important places where a copy constructor is called.

- a)** When an object is created from another object of the same type
- b)** When an object is passed by value as a parameter to a function
- c)** When an object is returned from a function

The copy constructor for class X has the form:

X::X(const X&)

You can pass an object of a class in constructor of same class. When we copy one object with copy constructor then compiler creates a second object with all the data of the first object.

**Example:**

```
#include <iostream.h>
#include <conio.h>
class Rectangle
{
public:
int l,b,a;
Rectangle () //No argument constructor
{
l=b=1;
}
Rectangle (int n, int m)//Argument(or Parameterised)constructor
{
l=n;
b=m;
}
Rectangle (Rectangle &r) //Copy Constructor
{
```

```
l=r.l;  
b=r.b;  
}  
void findarea ()  
{  
a=l*b;  
}  
};  
int main ()  
{  
clrscr();  
Rectangle r1; //invokes no argument constructor  
Rectangle r2(2,3); //invokes the parameterised constructor  
Rectangle r3=r1; //invokes the (explicit)copy constructor  
Rectangle r4=r2; //invokes the (explicit)copy constructor  
r1.findarea();  
r2.findarea();  
r3.findarea();  
r4.findarea();  
cout<<“The area for r1 is:”<<r1.a;  
cout<<”\nThe area for r2 is:”<<r2.a;  
cout<<”\nThe area for r3 is:”<<r3.a;  
cout<<”\nThe area for r4 is:”<<r4.a;  
return 0;  
}
```

### **Output:**

The area for r1 is:1

The area for r2 is:6

The area for r3 is:1

The area for r4 is:6

The copy constructor is called more often behind the scenes whenever the compiler needs a copy of an object. These objects, appropriately called temporaries, are created and destroyed as they are needed by the compiler. The most common place this occurs is during function calls, so that the semantics of call-by-value can be preserved. For example, here is a function that takes a String argument:

Always provide a copy constructor for your classes. Do not let the compiler generate it for you. If your class has pointer data members, you must provide the copy constructor.

If you do not provide a copy constructor, the compiler will generate one for you automatically. This generated copy constructor simply performs a member-wise assignment of all of the data members of a class. This is fine for a class that does not contain any pointer variables. It is a good idea to get in the habit of always providing the copy constructor for your classes.

If a copy constructor is not defined in a class, the compiler itself defines one. This will ensure a shallow copy. If the class does not have pointer variables with dynamically allocated memory, then one need not worry about defining a copy constructor. It can be left to the compiler's discretion.

## Constructor Overloading: Multiple constructors in a class

Like normal member function, we can also declare more than one constructor with same name in a class. This mechanism is called as constructor overloading. The concept is very similar to function overloading the difference is in function overloading all functions share same name with different arguments and different class name whereas in case of constructor, all the constructors and class share same name with different arguments. Now lets see an example of constructor overloading.

### Example : Constructor overloading

```
#include <iostream.h>
```

```
class Rectangle
```

```
{
```

```
int l, b;
```

```
public:  
    Rectangle ();  
    Rectangle (int,int);  
    int findarea (void) {return (l*b);}  
};  
Rectangle ::Rectangle () //No argument constructor  
{  
    l = 5;  
    b = 5;  
}  
Rectangle ::Rectangle (int a, int b) //Parameterized constructor  
{  
    l = a;  
    b = b;  
}  
int main ()  
{  
    Rectangle r1 (3,4); //Invokes parameterized constructor  
    Rectangle rb; //Invokes No argument constructor  
    cout << "r1 area: " << r1.findarea() << endl;  
    cout << "r2 area: " << r2.findarea() << endl;  
    return 0;  
}
```

### **Output:**

r1 area: 12

r2 area: 25

In this case, r2 was declared without any arguments, so it has been initialized with the

constructor that has no parameters, which initializes both l and b with a value of 5.

If we want to use its default constructor (the one without parameters), we do not have to include parentheses ():

```
Rectangle r2; // right declaration, calls default constructor
```

```
Rectangle r2(); // wrong! declaration
```

## Constructors with default arguments

Like normal member function, we can also declare constructors with default arguments.

### Example:

```
#include <iostream.h>
#include <conio.h>

class Rectangle
{
public:
    int l,b,a;

    Rectangle () //No argument constructor
    {
        l=b=1;
    }

    Rectangle (int n=1, int m=1) //Default Argument constructor
    {
        l=n;
        b=m;
    }

    void findarea ()
    {
        a=l*b;
    }
}
```

```

}

};

int main ()
{
    clrscr();

    Rectangle r1; //invokes no argument constructor

    Rectangle r2(2,3); //invokes the parameterised constructor

    // or Rectangle r2=Rectangle(2,3);

    r1.findarea();

    r2.findarea();

    cout<<"The area for r1 is:"<<r1.a;

    cout<<"\nThe area for r2 is:"<<r2.a;

    return 0;
}

```

### **Output:**

When we compile the above program it generates a compiler error as follows:

Ambiguity between ‘Rectangle::Rectangle()’ and ‘Rectangle::Rectangle(int,int)’

This is because when we declare all arguments to be default then that constructor becomes the same as the first constructor with no argument and creates above ambiguity for the compiler. So the solution is to remove the first constructor and then run the program to get the following output.

The area for r1 is:1

The area for r2 is:6

### **Dynamic initializaton of objects**

The process of initializing the an object during run-time is called dynamic initialization. That means we can initialize the data members of object at run time. Dynamic initialization is done using overloaded constructors. It provides the flexibility of using different format of data at run time.

## Example: Dynamic Initialization of objects

```
#include <stdio.h>
#include <iostream.h>
#include <conio.h>
#include <math.h>
#include <stdlib.h>

class point
{
    int x;
    int y;
public:
    point() {}
    point(int a, int b) //constructor declaration
    { //constructor definition
        x=a;
        y=b;
    }
    void display(void);
};

void point::display(void)
{
    cout<<"(x,y) = ("<<x<<","<<y<<")\n";
}

main()
{
    // point p1,p2; //can be done here
    int x1,y1;
```

```
cout<<“Enter x value: “;  
cin>>x1;  
cout<<“Enter y value: “;  
cin>>y1;  
point p1=point(x1,y1); //dynamic initialization  
point p2=p1; //copying object p1 into p2  
cout<<“Object p1”;  
p1.display();  
cout<<“Object p2”;  
p2.display();  
getch();  
return 0;  
}
```

### **Output:**

Enter x value: 23

Enter y value: 34

Object p1(x,y)=(23,34)

Object p2(x,y)=(23,34)

Dynamic initialization is done by using an empty constructor, e.g. point() {} and then defining the constructor with parameters, **e.g.**

```
point(int a, int b)  
{  
x=a;  
y=b;  
}
```

And then call the constructor with actual parameters, **e.g.**

```
int p1=point(x1,y1);
```

The advantage of dynamic initialization is that we can provide various initializations formats using overloaded constructors. The idea of dynamic initialization of objects will be more clear through following **example**:

```
#include <iostream.h>
#include <string.h>
#include <conio.h>

class book

{
private:
    char name[20];
    float price;
    float discount;
    float mrkt_price;

public:
    book() { }

    book(char[], float, float dis = 0.15);

    book(char[], float, int);

    void putdata(void);

};

book :: book (char n[], float p, float d)
{
    strcpy(name, n);
    price = p;
    discount = d;
    mrkt_price = price - price * discount;
}

book :: book (char n[], float p, int d)
```

```
{  
strcpy(name, n);  
price = p;  
discount = float (d)/100;  
mrkt_price = price - price * discount;  
}  
  
void book :: putdata(void)  
{  
cout<<“Book name: “<< name << endl;  
cout<<“Book price: “<< price << endl;  
cout<<“Discount on book: “<< discount << endl;  
cout<<“Market price of book: “<< mrkt_price << “\n\n”;  
}  
  
int main(void)  
{  
clrscr();  
book b1, b2, b3;  
char nm[20];  
float pr, di;  
int Di;  
cout<<“Enter book name, price and discount (in percentage)\n”;  
cin>> nm >> pr >> Di;  
b1 = book(nm, pr, Di); //dynamically initialize the object b1  
cout<<”\nEnter book name, price and discount” << endl;  
cin>> nm >> pr >> di;  
b2 = book(nm, pr, di); //dynamically initialize the object b2  
cout<<”\nEnter book name and price “<< endl;
```

```
cin>> nm >> pr;  
b3 = book(nm, pr); //dynamically initialize the object b3  
cout<< "Book 1 "<< endl;  
b1.putdata();  
cout<< "Book 2 "<< endl;  
b2.putdata();  
cout<< "Book 3 "<< endl;  
b3.putdata();  
getch();  
return(0);  
}
```

**Output:**

Enter book name, price and discount (in percentage)

TurboC++ 349.99 10

Enter book name, price and discount

VisualBasic 450.00 0.10

Enter book name and price

Java2 399.95

Book 1

Book name: TurboC++

Book price: 349.99

Discount on book: 0.1

Market price of book: 314.99

Book 2

Book name: VisualBasic

Book price: 450.00

Discount on book: 0.1

Market price of book: 405

Book 3

Book name: Java2

Book price: 399.95

Discount on book: 0.15

Market price of book: 339.95

As you can see in above program we have declared a class book with three constructors one is void constructor, second is with three arguments (book name, book price and discount on book in decimal format with default value 0.15) and third constructor is also with three arguments (book name, book price and discount on book in percentage format).

In main() function first we have created three objects b1, b2 and b3 of type book. Here when you create the objects b1, b2 and b3 a void constructor will be called with those objects then we have inputted three values from the user to call a constructor with discount in percentage format for b1 object. Then we have inputted three values from the user again to call constructor with discount in decimal format for object b2 and we have inputted two values from the user to call a constructor with default value of discount 0.15 with b3 object.

## Dynamic constructors

As we have seen in C language that when we declare two variables of type integer then both variables a and b will be allocated same amount of memory space (2 bytes). C++ provides the facility that the objects of a class can have different amount of memory spaces as per the requirement. This can be done by implementing dynamic constructors in the class.

Constructors can also be used to allocate memory while creating objects. Memory is allocated with the use of new operator. Dynamic constructors enable the system to allocate the right amount of memory for each object when the objects are of different size, thus we can save the memory. This type of allocation of memory is known as dynamic construction of objects. Now lets see a program for dynamic constructors.

```
#include <iostream.h>
```

```
#include <string.h>
```

```
#include <conio.h>
```

```
class name

{
private:
char *nm; // object size will vary by this member
int length;

public:
name()
{
length = 0;
nm = new char[length + 1] ;
nm[0] = '\0';

}
name(char *s)
{
length = strlen(s);
nm = new char[length + 1];
strcpy(nm, s);
}
void join(name &);

void putdata(void);

};

void name :: join(name &n)
{
char *temp;
length += n.length;
strcpy(temp, nm);
strcat(temp, n.nm);
```

```
delete nm;

nm = new char[length + 1];

strcpy(nm, temp);

}

void name :: putdata(void)

{

cout<< nm << endl;

}

int main(void)

{

name first_name("Jitendra"), middle_name("B");

name last_name("Patel"), full_name;

clrscr();

full_name.join(first_name);

full_name.join(middle_name);

full_name.join(last_name);

cout<<"First name is: ";

first_name.putdata();

cout<<"Middle name is: ";

middle_name.putdata();

cout<<"Last name is: ";

last_name.putdata();

cout<<"\n Full name is: ";

full_name.putdata();

getch();

return(0);

}
```

## **Output:**

First name is: Jitendra

Middle name is: B

Last name is: Patel

Full name is: JitendraBPatel

As given in above program that we have declared a class name with two constructors in which dynamically we are initializing the values of data members length and nm according to parameters passed in constructor from main() function. Thus the memory allocated to the objects of class name will be according to the length of character string nm.

In given example we have created four objects first\_name, middle\_name, last\_name and full\_name. At the time of creating these four objects, a void constructor will be called and all objects will have same amount of memory space. But when we explicitly call a constructor with a character string parameter for objects first\_name, middle\_name and last\_name, all these three objects will have different amount of memory space. Then we have called join member function three times with the addresses of these three objects through full\_name object. Thus, the memory of object full\_name is increased three times when we called join() function with that object. The idea will be more clear through output of the given program. As you can see in the output, the memory allocated to all four objects is different.

## Constructors

As its name implies, the constructor is used to destroy the objects that have been created by constructor, which no longer needed. A constructor is a member function that is automatically called to destroy an object when it is no longer required or when the object is out of scope. In the same way that a constructor is called when the object is created, the destructor is automatically called when the object is destroyed. Destructor is also a member function of a class as like constructor is. Like constructor, destructor also have same name of class name but in prefix of destructor we have to specify a tilde (~) symbol. A destructor never takes any arguments nor does it return any value. It will be executed implicitly by the compiler when the program is terminated. It will clear the memory allocated to objects which are no longer accessible. It is better way of programming that wherever you define constructors in the class, you define destructor also. In the same way that a constructor sets things up when an object is created, a destructor performs shutdown procedures when an object is destroyed. Whenever an object goes out of scope it is destroyed and the memory used by that object is reclaimed. Just before the object is destroyed, an object's destructor is called to allow any clean-up to be performed. A destructor for class X has the form X::~X(). The destructor must release any resources obtained by an object during its lifetime, not just those that were obtained during construction.

Typically, some data members for an object are allocated during construction. However, some objects may allocate memory or use other resources in response to certain operations performed on the object. When implementing your destructors, make sure you release all of the resources that have been obtained by the object, not just those obtained by the constructors. Always provide a destructor for your classes. Do not let the compiler generate it for you. If you want to be in complete control of how your class operates. Do not let the compiler generate any of your code.

### Example:

```
// This program demonstrates a constructor and destructor.

#include <iostream.h>

class Demo

{

public:
```

```
Demo(); // Constructor
~Demo(); // Destructor
};

Demo::Demo()
{
    cout << "Welcome to the constructor!\n";
}

Demo::~Demo()
{
    cout << "The destructor is now running.\n";
}

void main()
{
    Demo DemoObj; // Declare a Demo object;
    cout << "This program demonstrates an object\n";
    cout << "with a constructor and destructor.\n";
}
```

### **Output:**

Welcome to the constructor!

This program demonstrates an object

with a constructor and destructor. // program ends, destructor is called

The destructor is now running.

### **Example 2:**

```
#include<iostream.h>
#include<conio.h>
class Abd
{
```

```
public:  
Abd()  
{  
cout<<“This is from the constructor.”<<endl;  
}  
~Abd()  
{  
cout<<“This is from the destructor.”<<endl;  
}  
};  
void main()  
{  
clrscr();  
Abd ob; //object will be destroyed when main() ends  
{  
Abd ob1; //object will be destroyed when block ends  
Abd ob2; //object will be destroyed when block ends  
}  
Abd ob3; //object will be destroyed when main() ends  
getch();  
}
```

### **Output:**

This is from the constructor.  
This is from the constructor.  
This is from the constructor.  
This is from the destructor.  
This is from the destructor.

## Guidelines for constructors and destructors

There are some guidelines that apply to both constructors and destructors.

Avoid calls to virtual functions in constructors and destructors.

You may call other member functions from constructors and destructors. Calling a virtual function will result in a call to the one defined in the constructor's (or destructor's) own class or its bases, but not any function overriding it in a derived class. This will guarantee that unconstructed derived objects will not be accessed during construction or destruction.

Also, calling a pure virtual function directly or indirectly for the object being constructed or destroyed is undefined. The only time it is valid is if you explicitly call the pure virtual function. For example, if class A has pure virtual function p(), then: A::p() is an explicit call. The reason this works is because explicit qualification circumvents the virtual calling mechanism.

Any data which is declared private inside a class is not accessible from outside the class. A function which is not a member or an external class can never access such private data. But there may be some cases, where a programmer will need access to the private data from non-member functions and external C++ offers some exceptions in such cases.

A class can allow non-member functions and other classes to access its own private data, by making them as friends.

Once a non-member function is declared as a friend, it can access the private data of the class

similarly when a class is declared as a friend, the friend class can have access to the private data of the class which made this a friend

## this pointer

C++ uses a unique keyword called “this” to represent an object that invokes a member function. ‘this’ is a pointer to object for which ‘this’ function was called. This unique pointer is automatically passed to a member function when it is called. The pointer “this” acts as an implicit parameter to all the member functions. If we have to reference the object itself in a member function, we can do it through an implicit pointer this. One important application of the pointer this is to return the object it points to, e.g.

```
return *this;
```

This statement assumes importance when we want to compare two or more objects inside a member function and return the invoking object as a result, e.g.

```
person & person :: greater(person & x)
```

```
{
```

```
If (x.age>age)
```

```
return x
```

```
else
```

```
return *this;
```

```
}
```

Suppose we invoke this function by the call

```
C=A.greater(B);
```

The function will return the object B if the age of the person B is greater than that of A, otherwise, it will return the object A using the pointer this.

### **Example:**

```
class person
```

```
{
```

```
char name[20];
```

```
float age;
```

```
public:
```

```
person (char *s, float a)
```

```
{
```

```
strcpy(name, s);
```

```
age=a;
```

```
}
```

```
person & person :: greater(person & x)
```

```
{
```

```
if(x.age>=age)
return x;
else
return *this;
}

void display(void)
{
cout<< "Name: " << name << "\n";
<< "Age: " << age << "\n";
}

main()
{
person P1("Paula", 37), P2("Abdul", 23), P3("Lutz", 40);
person P('\0', 0);
P=P1.greater(P3);
cout<<"The elder person is: \n";
P.display();
P=P1.greater(P2);
cout<<"The elder person is: \n";
P.display();
}
```

## Exercise

### True or False

1. State that the following statement is True/False.
  - a) A constructor is not a member of a class.
  - b) A constructor must have same name of class name.
  - c) A constructor can return value.
  - d) In a constructor we can pass more than one arguments.
  - e) A constructor cannot have default arguments.
  - f) We cannot define a constructor as inline.
  - g) We cannot overload constructors in the program.
  - h) We cannot pass an object of same class in a constructor.
  - i) We cannot initialize the objects dynamically without use of new and delete operator.
  - j) A destructor can be declared whether a constructor is declared or not in the class.

### Short Questions

1. Explain the following terms:
  - a) Constructor
  - b) Copy constructor
  - c) void constructor
  - d) Default constructor
  - e) Parameterized constructor
  - f) Destructor
2. What is Destructor? Give an example of a destructor.
3. Explain the characteristics of constructors.
4. What are the similarities and differences between constructor and destructor?
5. Explain the difference between the copy constructor and the assignment operator.

7. Is there a limit on number of constructor functions in a class?
8. How many destructor functions can a class have?
9. Which member functions are created automatically by the compiler if they are not included in the class definition?
10. Explain the ambiguity of default arguments in constructors.
11. Explain how the constructor overloading is possible in C++.
12. Explain how a '=' operator is useful in copy constructor.
13. Write a short note on dynamic constructor.

## Programming Exercises

1. Write a program for creating a class employee with void constructor that initialize the values of data members, parameterized constructor which declared with default arguments (default salary = 10000) in the class.
2. Write a program of class complex for addition of two same complex number using copy constructor.
3. Write a program, which illustrates the concept of dynamic constructor.



## CHAPTER:8 Inheritance

Inheritance is the process of creating new classes, called derived classes, from existing classes or base classes. A class that is inherited is called a base class or superclass. A class which is inheriting another class is called a derived class or subclass. When a class inherits another class, members of the base class become the members of the derived class. The derived class inherits most of the capabilities (except private) of the base class, the derived class can also add its own features, data etc., It can also override some of the features (functions) of the base class.

C++ inheritance is very similar to a parent-child relationship. When a class is inherited all the functions and data member are inherited, although not all of them (like private members) will be accessible by the member functions of the derived class.

The general form of inheritance is:-

```
class derived_name : accessSpecifier base_name
{
};
```

The derived\_name is the name of the derived class. The base\_name is the name of the base class. The accessSpecifier can be private, public or protected.

**Example:** A C++ program to explain the concepts of inheritance

```
#include<iostream.h>

class test
{
public:
    test(void) { x=0; }

    void f(int n1)
    {
        x= n1*5;
    }

    void output(void) { cout<<x; }

private:
```

```
int x;  
};  
class sample: public test  
{  
public:  
sample(void) { s1=0; }  
void f1(int n1)  
{  
s1=n1*10;  
}  
void output(void)  
{  
test::output();  
cout << s1;  
}  
private:  
int s1;  
};  
int main(void)  
{  
sample s;  
s.f(10);  
s.output();  
s.f1(20);  
s.output();  
}  
The output of the above program is
```

The access\_specifier can be private, public or protected.

If the access\_specifier is public then all public members of the base class become public members of the derived class and protected members of the base class become the protected members of the derived class.

After inheriting class A publicly, members of class B	
<pre>class A { private: int x; public: int y; protected: int z; } class B : public A { private: int a; public: int b; protected: int c; }</pre>	<pre>class B { private: int a; public: int b, y, z; protected: int c; }</pre>

If the access\_specifier is private then all public and protected members of the base class will become private members of the derived class.

After inheriting A privately members of class B	
<pre>class A { private: int x; public: int y; protected: int z; } class B : private A { private: int a; public: int b; protected: int c; }</pre>	<pre>class B { private: int a, y, z; public: int b; protected: int c; }</pre>

If the access\_specifier is protected then the public and protected members of the base

class become the protected members of the derived class. Whether access\_specifier is public, private or protected, private members of the base class will not be accessed by the members of the derived class.

	After inheriting A protected members of class B
<pre>class A { private: int x; public: int y; protected: int z; } class B protected A { private: int a; public: int b; protected: int c; }</pre>	<pre>class B { private: int a; public: int b; protected: int c; }</pre>

The access\_specifier protected provides more flexibility in terms of inheritance. The private members of the base class cannot be accessed by the members of the derived class. The protected members of the base class remain private to their class but can be accessed and inherited by the derived class. The protected members of the base class will remain private to the other elements of the program.

Base class member access specifier	Type of inheritance		
	public inheritance	protected inheritance	private inheritance
Public	public in derived class. Can be accessed directly by any non-static member functions, friend functions and non-member functions.	protected in derived class. Can be accessed directly by all non-static member functions and friend functions.	private in derived class. Can be accessed directly by all non-static member functions and friend functions.
Protected	protected in derived class. Can be accessed directly by all non-static member functions and friend functions.	protected in derived class. Can be accessed directly by all non-static member functions and friend functions.	private in derived class. Can be accessed directly by all non-static member functions and friend functions.
Private	Hidden in derived class. Can be accessed by non-static member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by non-static member functions and friend functions through public or protected member functions of the base class.	Hidden in derived class. Can be accessed by non-static member functions and friend functions through public or protected member functions of the base class.

As you can observe from above figure, different types of inheritances have different effect on how members of the base class can be accessed from the derived classes.

We can summarize the different access types for derived class according to which members can be accessed:

Access to	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
non members	yes	no	no

Where “non members” represent any access from outside the class, such as from main(), from another class or from a function.

## What is inherited from the base class?

In principle, a derived class inherits every member of a base class except:

- a) its constructor and its destructor
- b) its operator=() members
- c) its friends

Although the constructors and destructors of the base class are not inherited themselves, its default constructor (**i.e.**, its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.

If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:

```
derived_constructor_name(parameters):base_constructor_name(parameters)  
{...}
```

Here is a program which illustrates the features of inheritance.

```
#include<iostream.h>  
  
class shape  
{  
private :  
    double length;  
protected:  
    double breadth;  
public :  
    double len()  
    {  
        return(length);  
    }  
    shape(double length1,double breadth1)
```

```
{  
length=length1;  
breadth=breadth1;  
}  
//shape() { }  
};  
class shape1  
{  
public:  
double height;  
shape1(double height1)  
{  
height=height1;  
}  
//shape1() { }  
};  
class cuboid : public shape, private shape1  
{  
public:  
cuboid(double length1,double breadth1,double  
height1):shape(length1,breadth1),shape1(height1)  
{  
cout << " A constructor is called " << endl;  
}  
double volume()  
{  
return(height*breadth*len());  
}
```

```

}

double bre()
{
    return(breadth);
}

double ht()
{
    return(height);
}

};

int main()
{
    cuboid c1(2.4,3.5,6.7);

    cout << "The length of the cuboid is : " << c1.len()<<endl;
    cout << "The breadth of the cuboid is : " << c1.bre()<<endl;
    cout << "The height of the cuboid is : " << c1.ht()<< endl;
    cout << "The volume of the cuboid is : " << c1.volume()<<endl;
    return(0);
}

```

The result of the program is:-

```

C:\Documents and Settings\A\Desktop\bids\CBid\programs\Debug\inheritance
A constructor is called
The length of the cuboid is : 2.4
The breadth of the cuboid is : 3.5
The height of the cuboid is : 6.7
The volume of the cuboid is : 56.28
Press any key to continue...

```

The program has two base classes shape and shape1 and one derived class called cuboid which inherits shape as public and shape1 as private. The public and protected members of shape become public and protected members of derived class cuboid. The private members of shape remain private to the class shape. The members of shape1 class

become the private members of the derived class cuboid.

The statement

```
class cuboid : public shape, private shape1
```

states that class cuboid inherits class shape as public and class shape1 as private. The statement

```
cuboid(double length1,double breadth1,double height1):
```

```
shape(length1,breadth1),shape1(height1)
```

```
{
```

```
cout << " A constructor is called " << endl;
```

```
}
```

declares the constructor of the class cuboid. When constructor of class cuboid is called first constructor of shape is executed and then constructor of shape1 is executed and after that the constructor of cuboid is executed. The statements

```
double volume()
```

```
{
```

```
return(height*breadth*len());
```

```
}
```

calculate the volume of the cuboid. The class cuboid cannot access the private data member length of the shape class. It access the length by calling the function len() which returns the private data member length. The data member breadth becomes the protected member of the class cuboid. The height which is public member of shape1 class becomes the private member of the class cuboid as it inherits the shape 1 class as private. The statements

```
double bre()
```

```
{
```

```
return(breadth);
```

```
}
```

returns the breadth of the cuboid as data member breadth cannot be accessed outside the class as it is protected member of cuboid. The statement

```
double ht()
{
    return(height);
}
```

returns the height of the cuboid as data member height cannot be accessed outside the class as height is the private data member of the class cuboid. The statement

```
cuboid c1(2.4,3.5,6.7);
```

creates an object c1 of type cuboid. The constructor is called to initialize the values of the cuboid. The constructor of shape is executed and then constructor of shape1 is executed and then finally constructor of cuboid is executed. The statement

```
cout << "The length of the cuboid is : " << c1.len() << endl;
```

displays the length of the cuboid as c1.len() calls the len() function of class shape which is also the public member function of cuboid. The statement

```
cout << "The breadth of the cuboid is : " << c1.bre() << endl;
```

displays the breadth of the cuboid. As the data member breadth cannot be accessed directly as it is protected member of the class cuboid so the function bre() returns the breadth of the cuboid. The statement

```
cout << "The height of the cuboid is : " << c1.ht() << endl;
```

displays the height of the cuboid. The data member height cannot be accessed directly as it is private member of class cuboid so it is accessed through the function ht() which returns height.

## Types of Inheritance

Depending on the number of superclass and subclass used in the derivation we can have different types as follows:

- a)** Single Inheritance
- b)** Multiple Inheritance
- c)** Hierarchical Inheritance
- d)** Multilevel Inheritance
- e)** Hybrid Inheritance

## Single Inheritance

A derived class with only one base class is called Single Inheritance. When one class is derived only from one base class then such inheritance is called single-level inheritance. The single-level inheritance is shown below.



General syntax for single inheritance:

```
Derived_class_name : type_of_inheritance Base_class_name{};
```

**Example:** implementation of single inheritance

```
#include<iostream.h>
#include<conio.h>

class base
{
    int a;
public:
    int b;
    void get_ab(void);
    int get_a(void);
    void show(void);
};

class derived:private base
{
    int c;
public:
    void mul(void);
    void display(void);
};
```

```
void base::get_ab(void)
{
    cout<<“get”;
    a=5;
    b=6;
}

int base::get_a(void)
{
    return a;
}

void base::show()
{
    cout<<“value of a is \n”<<a<<”\n”;
}

void derived::mul(void)
{
    get_ab();
    c=b*get_a();
}

void derived::display(void)
{
    show();
    cout<<b<<”\n”;
    cout<<c<<”\n”;
}

void main()
{
```

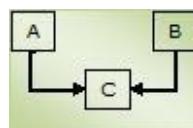
```

clrscr();
/*base b;
b.get_ab();
b.show();*/
derived d;
//d.get_ab();
//d.show();
d.mul();
d.display();
getch();
}

```

## Multiple Inheritance

A derived class with several base classes is called Multiple Inheritance. When single class inherits the properties from more than one base class, it is called the multiple inheritances. In other words we can say that multiple inheritances mean that one class can have more than one base class. It allows us to combine features of several existing classes into a single class as shown below:



General syntax for multiple inheritance:

Derived\_class\_name:type\_of\_inheritance1

Base\_class\_name1,type\_of\_inheritance2 Base\_class\_name2,...{};

**Example:** implementation of multiple inheritance

```
#include<iostream.h>
```

```
#include<conio.h>
```

```
class detail
```

```
{
```

```
protected:
```

```
int sno;  
char name[10];  
public:  
void get(void)  
{  
cout<<“enter the serialno”;  
cin>>sno;  
cout<<“enter the name”;  
cin>>name;  
}  
};
```

class level

```
{  
protected:  
char sname;  
char course[10];  
public:  
void read(void)  
{  
cout<<“enter whether student his UG(U) \ PG(P)”;  
cin>>sname;  
cout<<“enter course in UG”;  
cin>>course;  
}  
};
```

class office:public detail,public level

```

{
public:
void getdata(void)
{
if(sname=='U')
cout<<name<<"\tis doing his/her graduation
with"<<course;

if(sname=='P')
cout<<name<<"\t is completed his/her graduation
with"<<course;

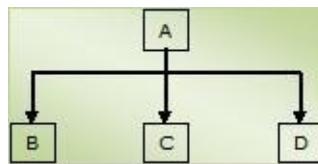
}
};

void main()
{
clrscr();
office o;
o.get();
o.read();
o.getdata();
getch();
}

```

## Hierarchical Inheritance

When many subclasses inherit properties from a single base class, it is called as hierarchical inheritance. The base class contains the features that are common to the subclass and a subclass can inherit all or some of the features from the base class as shown below:



**Example:** implementation of hierarchical inheritance.

```
#include<iostream.h>
```

```
#include<conio.h>
```

```
class student
```

```
{
```

```
protected:
```

```
int sid;
```

```
char name[20];
```

```
public:
```

```
void getdata(void)
```

```
{
```

```
cout<<“enter student ID”;
```

```
cin>>sid;
```

```
cout<<“enter student name”;
```

```
cin>>name;
```

```
}
```

```
void putdata(void)
```

```
{
```

```
cout<<“student name\t”<<name<<”\n”;
```

```
cout<<“ID\t”<<sid<<”\n”;
```

```
}
```

```
};
```

```
class arts:public student
```

```
{
```

protected:

char sub[10];

public:

void subject(void)

{

cout<<“enter subject in arts”;

cin>>sub;

}

void print(void)

{

cout<<“name \t”<<name;

cout<<”\n sid\t”<<sid;

cout<<”\n subject in arts\t”<<sub;

}

};

class science:public student

{

protected:

char sub1[10];

public:

void subj(void)

{

cout<<“enter group in science “;

cin>>sub1;

}

void put(void)

{

```

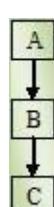
cout<<“name \t”<<name;
cout<<”\n sid\t”<<sid;
cout<<”\n group\t”<<sub1;
}
};

void main()
{
clrscr();
arts a;
a.getdata();
a.putdata();
a.subject();
a.print();
science s;
s.getdata();
s.putdata();
s.subj();
s.put();
getch();
}

```

## Multilevel Inheritance

The mechanism of deriving class from another “derived class” is known as Multilevel Inheritance.



**Example:** implementation of multilevel inheritance

```
#include<iostream.h>
#include<conio.h>
class student
{
protected:
int rollno;
public:
void get(int a)
{
rollno=a;
}
void put(void)
{
cout<<"\nROLLNO \t"<<rollno<<"\n";
}
};

class test:public student
{
protected:
float sub1;
float sub2;
public:
void get_marks(float x,float y)
{
sub1=x;
sub2=y;
}
```

```
void put_marks(void)
{
    cout<<"\nmarks in sub1\t"<<sub1;
    cout<<"\nmarks in sub2\t"<<sub2;
}

};

class result:public test
{
    float total;

public:
    void display(void)
    {
        total=sub1+sub2;
        cout<<"\ntotal marks\t"<<total;
    }
};

void main()
{
    clrscr();
    result r;
    int roll;
    float s1,s2;
    cout<<"enter the rollno";
    cin>>roll;
    r.get(roll);
    cout<<"enter marks for subject1 and subject2\n";
    cin>>s1>>s2;
```

```

r.get_marks(s1,s2);

r.put_marks();

r.display();

r.put();

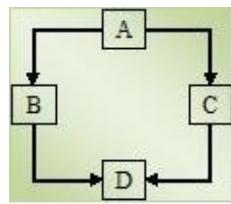
getch();

}

```

## Hybrid Inheritance

It is a combination of multiple inheritances and the hybrid inheritance. In hybrid Inheritance a class inherits from a multiple base class which itself inherits from a single base class. This form of inheritance is known as hybrid inheritance. It is shown below:



**Example:** implementation of hybrid inheritance

```

#include<iostream.h>

#include<conio.h>

class student

{

protected:

int rno;

char name[20],course[6];

public:

void read(void)

{

cout<<"enter rollno";

cin>>rno;

cout<<"enter name";

```

```
cin>>name;  
cout<<“enter course”;  
cin>>course;  
}  
};  
  
class internals:public student  
{  
protected:  
float in1,in2;  
public:  
void get(void)  
{  
cout<<“enter 1st internal marks”;  
cin>>in1;  
cout<<“enter 2nd internal marks”;  
cin>>in2;  
}  
};  
  
class externals  
{  
protected:  
float a;  
public:  
void score(void)  
{  
cout<<“enter external marks secured”;  
cin>>a;
```

```

}

};

class result:public internals,public externals

{
float total;

public:

void display(void)

{
total=in1+in2+a;

cout<<“student name is”<< name<<” with rollno “<<rno<<” and “;

cout<<“total marks secured is”<<total;

}

};

void main()

{
clrscr();

result r;

r.read();

r.get();

r.score();

r.display();

getch();

}

```

**Example:** implementation of virtual base class

```

#include<iostream.h>

#include<conio.h>

class student

```

```
{  
protected:  
int admno,year;  
char stid[10],name[20];  
public:  
void getdata(void)  
{  
cout<<“enter admission number”;  
cin>>admno;  
cout<<“enter student ID”;  
cin>>stid;  
cout<<“enter name”;  
cin>>name;  
cout<<“enter year”;  
cin>>year;  
}  
};  
class botany:virtual public student  
{  
protected:  
int marksb;  
public:  
void get(void)  
{  
cout<<“enter marks in botany”;  
cin>>marksb;  
}  
}
```

```
};
```

```
class zoology:public virtual student
```

```
{
```

```
protected:
```

```
int marksz;
```

```
public:
```

```
void gets(void)
```

```
{
```

```
cout<<“enter marks in zoology”;
```

```
cin>>marksz;
```

```
}
```

```
};
```

```
class science:public botany,public zoology
```

```
{
```

```
int total;
```

```
public:
```

```
void bdisplay(void)
```

```
{
```

```
total=marksb+marksz;
```

```
cout<<”\nadmission number\t”;
```

```
cout<<admno;
```

```
cout<<”\nstudent ID\t”;
```

```
cout<<stid;
```

```
cout<<”\nname\t”;
```

```
cout<<name;
```

```
cout<<”\nyear\t”;
```

```
cout<<year;
```

```

cout<<"\ntotal marks in biology\t"<<total;
}

};

void main()
{
    clrscr();
    science s;
    s.getdata();
    s.get();
    s.gets();
    s.bdisplay();
    getch();
}

```

A derived class can inherit one or more base classes. A constructor of the base class is executed first and then the constructor of derived class is executed. A destructor of derived class is called before the destructor of base class. The arguments to the base class constructor can be passed as follows:-

derived\_constructor (argument list): base1 (arg\_list) base2(arg\_list1)  
 baseN(arg\_list)

The derived\_constructor is the name of the derived class. The argument list is list of the data members of the derived class. The base1 is name of the base class. The arg\_list is the list of the members of the base class.

**Example:** implementation of constructor in derived class

```

#include<iostream.h>
#include<conio.h>
class base1
{
    int x;

```

```
public:  
base1(int i)  
{  
x=i;  
}  
void show(void)  
{  
cout<<“base 1 X value is\t”<<x<<”\n”;  
}  
};  
class base2  
{  
float y;  
public:  
base2(float j)  
{  
y=j;  
}  
void show1(void)  
{  
cout<<“base2 Y value is\t”<<y<<”\n”;  
}  
};  
class derive:public base1,public base2  
{  
int m,n;  
public:
```

```
derive(int a,float b,int c,int d):  
base1(a),base2(b)  
{  
m=c;  
n=d;  
cout<<“derived class conatructor\n”;  
}  
void show2()  
{  
cout<<“m value”<<m;  
cout<<”\n n value”<<n;  
}  
};  
void main()  
{  
derive d(5,1.1,7,9);  
d.show();  
d.show1();  
d.show2();  
}
```

## **Exercise**

### **Short Questions**

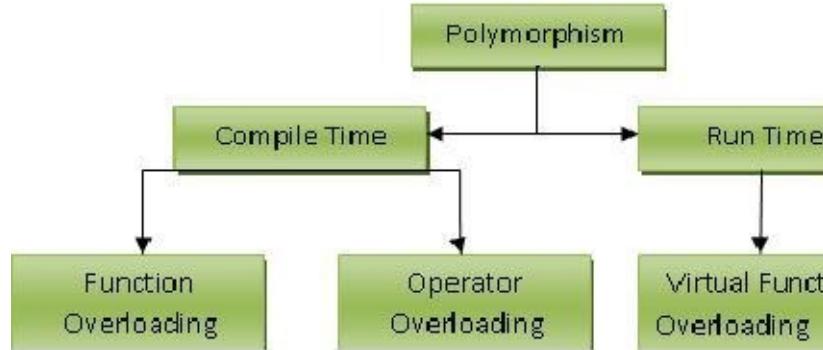
- 1) Define Inheritance.
- 2) Explain various types of inheritance with example.



## CHAPTER:9 Polymorphism

“Poly” means “many” and “morph” means “form”. Polymorphism is the ability of an object (or reference) to assume (be replaced by) or become many different forms of object.

There are two types of polymorphism one is compile time polymorphism and the other is run time polymorphism. Compile time polymorphism is functions and operators overloading. Runtime time polymorphism is done using inheritance and virtual functions.



### Function overloading

C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters (at least as far as their types are concerned). This capability is called function overloading. When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call. Function overloading is commonly used to create several functions of the same name that perform similar tasks but on different data types.

### Operator overloading

Operator overloading allows existing C++ operators to be redefined so that they work on objects of user-defined classes. Overloaded operators are syntactic sugar for equivalent function calls. They form a pleasant facade that doesn't add anything fundamental to the language (but they can improve understandability and reduce maintenance costs).

## Function Overloading

A program can consist of two functions where one can perform integer addition and other can perform addition of floating point numbers but the name of the functions can be same such as add. The function add() is said to be overloaded. Two or more functions can have same name but their parameter list should be different either in terms of parameters or their data types. The functions which differ only in their return types cannot be overloaded. The compiler will select the right function depending on the type of parameters passed. In cases of classes constructors could be overloaded as there can be both initialized and uninitialized objects.

Let's see an example of overloading function. We will see the example of calculating volume of block (it's all sides may be same, it's height and width may be same and length may be different, or all three sides may be different). So here we will declare three functions with same name but with different arguments.

### Example

```
#include <iostream.h>
#include <conio.h>
float volume(int);
float volume(float, float);
float volume(float, float, float);
int main(void)
{
    float height, width, length;
    clrscr();
    int h=15;
    float vol = volume(h);
    cout<<“Volume of cube (when all the sides are same) is:”<<vol<<endl;
    height = width = 20.10;
    length = 25;
    vol = volume(height, length);
```

```

cout<<“Volume of block (when height and width are same) is:”<<vol <<
endl;

height = 10.50;

width = 20.15;

length = 5;

vol= volume(height, width, length);

cout<<“Volume of block(when all sides are different) is:”<<vol<<ndl;

getch();

return(0);

}

float volume (int h)

{

return(h * h * h);

}

float volume (float h, float l)

{

return(h * h * l);

}

float volume (float h, float w, float l)

{

return(h * w * l);

}

```

## Output

Volume of cube (when all the sides are same) is: 3350

Volume of block (when height and width are same) is: 10100.25

Volume of block (when all sides are different) is: 1057.875

A function call first matches the prototype having the same number and type of

arguments and then calls the appropriate functions for executions. The function selection involves the following steps:

The compiler first tries to find an exact match in which the types of actual arguments are the same and use that function.

If an exact match not found, the compiler uses the integral promotions to the actual arguments. For example, char to int or float to double, etc.

When either of them fails, the compiler tries to use the built in conversions to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Consider the following two functions:

```
long absolute(long a)
```

```
double absolute (double b)
```

When you call a function absolute (10), the compiler will give error because int argument can be converted to either long or double, thereby creating an ambiguous situation as to which version absolute should be used.

If all of the steps fail, then the compiler will try the user defined conversions in combination with integral promotions and built in conversions to find a unique match.

## Operator Overloading

An operator is similar to a function: it takes an argument or arguments and returns a value as its result. However, the way in which operators are used differs from the way functions are used. For example, if we were to write a normal C++ function plus to perform addition, it would have to be used as follows:

```
x = plus (y, 2);
```

However, using the built-in addition operator, we can perform the same function as follows:

```
x = y + 2;
```

In other words, the operator symbol appears between the two arguments, instead of before them.

The +, -, \* and / operators are all binary operators. Binary operators take two arguments and return a single value. The ++ and — operators are unary operators: this means that they take a single argument and return a single value.

The built-in operators in C++ are defined only for the simple data types, i.e. int, char, float, long int, short, double. This means that if we define a new class, it will not be possible to use these operators with objects of our new class. For example, suppose we define a new class called Rational, to store and perform calculations with rational numbers (i.e. numbers that can be written as the ratio of two integers, for example 3/7). It would be nice if we could make use of the built-in arithmetic operators to perform these calculations, as in the following code:

```
Rational r1 (2, 5); //define rational number 2/5
```

```
Rational r2 (5, 7); //define rational number 5/7
```

```
Rational r3;
```

```
r3 = r1 * r2; //use built-in multiply operator
```

However, since the built-in operators are only defined for the built-in simple data types, this is not possible. Programmer can redefine the meaning of operator symbols like +, -, ... to make operators work on different data types. **For example:**

**a)** “+” is used to add two int or float data types.

**b)** But cannot be used to add arrays or complex (a + ib) numbers.

Operator overloading is a way of allowing the use of built-in operators for classes that you have written yourself.

### **Why do we need operator overloading?**

Before we proceed further, you ought to know the reason we do operator overloading. Do we really need it? For instance, compare the following syntaxes to perform addition of two objects a and b of a user defined class fraction (assume class fraction has a member function called add() which adds two fraction objects):

```
c=a.add(b);
```

```
c=a+b;
```

Which one is easier to use, line 1 or 2?

If your answer is the first statement, then our discussion ends right here. But, we assume that you are obliged to choose the second one. Only then we can continue our discussion.

Operator overloading is needed to make operation with user defined data type, i.e., classes, can be performed concisely. If you do not provide the mechanism of how these operators are going to perform with the objects of our class, you will get this error message again and again.

## **Operator Function**

How to define the behaviour of operators in our class? It is like creating a function that describes its behaviours. In C++, an operator is just one form of function with a special naming convention. As such, it can have return value as well as having some arguments or parameters. Recall the general format of a function prototype:

```
return_type function_name(type_arg1, type_arg2, ...)
```

An operator function definition is done similarly, with an exception that an operator's function name must take the form of operatorX, where X is the symbol for an operator, e.g. +, -, /, etc. For an instance, to declare multiplication operator \*, the function prototype would look like

```
return_type operator*( type arg1, type arg2)
```

Operators are overloaded by creating operator functions. The keyword operator is used to define operator function. Operator overloading doesn't allow creating new operators. The general form of operator function is:-

```
return_type operator # (arg_list)
```

```
{  
}
```

return\_type is the data type of the value returned from the function. The arg\_list is the list of the arguments to the function. The operator comes after the keyword operator. Instead of # there will be an operator.

## Rules for Operator Overloading

Although C++ allows us to overload operators, it also imposes restrictions to ensure that operator overloading serves its purpose to enhance the programming language itself, without compromising the existing entity. The followings are the restrictions:

**a)** Can not change the original behavior of the operator with built in data types.

**b)** Can not create new operator

**c)** Operators =, [], () and -> can only be defined as members of a class and not as global functions

**d)** The arity or number of operands for an operator may not be changed. For example, addition, +, may not be defined to take other than two arguments regardless of data type.

**e)** The precedence of the operator is preserved, i.e., does not change, with overloading

Operator overloading can be a very useful tool when you are developing reasonably complex classes in C++. You can overload (almost) any of the built-in operators in C++.

a) Operators that can be overloaded:

+	-	*	/	%	^	&		~	!
=	<	>	+=	-=	*=	/=	%=	^=	=
<<	>>	>>=	<<=	=	!=	<=	>=	&&	
++	--	->*	,	->	[]	0	new	delete	

b) Operators that can not be overloaded:

.	*	::	?:	sizeof
---	---	----	----	--------

Most overloaded operators should be defined outside of the class, i.e. they should not be member functions of the class. However, remember that if the data members they need

to access are protected or private they should be made friends of the class to gain access to the required data members. Also, the following operators must always be made members of the class:

= () [] ->

In addition, operators that change the values of one of their arguments are normally made member functions of the class, e.g.

+= -= \*= /= ++ —

Following table summarises the function prototypes necessary to overload some of the more common operators.

Operators	Function Prototype	Class Member?
+	Type operator+ (const Type&, const Type&)	N
-	Type operator- (const Type&, const Type&)	N
*	Type operator* (const Type&, const Type&)	N
/	Type operator/ (const Type&, const Type&)	N
=	Type& operator= (const Type&)	Y
<<	ostream& operator<< (ostream&, const Type&)	N
>>	istream& operator>> (istream&, Type&)	N
+=	Type& operator+= (const Type&)	Y
-=	Type& operator-= (const Type&)	Y
*=	Type& operator*= (const Type&)	Y
/=	Type& operator/= (const Type&)	Y
==	int operator== (const Type&, const Type&)	N
>	int operator> (const Type&, const Type&)	N
>=	int operator>= (const Type&, const Type&)	N
<	int operator< (const Type&, const Type&)	N
<=	int operator<= (const Type&, const Type&)	N
[]	Type& operator[] (int)	Y

**[Table: Overloaded function prototypes for common operators]**

When overloading operators, you can never change the unary/binary nature of the operator. For example, the < operator is a binary operator (it takes two arguments). Therefore if you overload the < operator you must define it as a binary operator. However, remember that the + and – operators have both unary and binary forms, e.g. the following statements are all valid in C++:

x = y + 2; //binary addition operator

x = +2; //unary addition operator

x = y – 2; //binary subtraction operator

```
x = -y; //unary subtraction operator
```

Finally, recall that there are built-in rules for operator precedence in C++: for example the `*` operator will always be evaluated before the `+` or `-` operators. Remember that it is never possible to override these predefined operator precedence rules.

## Overloading unary operator

The prefix `++` operator can be overloaded as such, without any change. Look at the function/operator definition.

```
class Test
```

```
{
```

```
int i;
```

```
public:
```

```
void operator ++()
```

```
{
```

```
++i;
```

```
}
```

```
};
```

The post fix `++` operator will be overloaded with a dummy integer parameter as follows.

```
class Test
```

```
{
```

```
int i;
```

```
public:
```

```
void operator ++(int)
```

```
{
```

```
i++;
```

```
}
```

```
};
```

Similar method of overloading is used for the — post/prefix operators also.

```
#include<iostream.h>
```

```
#include<conio.h>
```

```
class over
```

```
int a;
```

```
public:
```

```
void set()
```

```
{
```

```
cout<<“Enter one number”;
```

```
cin>>a;
```

```
}
```

```
void display()
```

```
{
```

```
cout<<“The number is”<<a;
```

```
}
```

```
friend void operator—(over &);
```

```
friend void operator++(over &);
```

```
};
```

```
void operator—(over &A)
```

```
{
```

```
A.a—;
```

```
}
```

```
void operator++(over &A)
```

```
{
```

```
A.a++;
```

```
}
```

```
void main()
```

```
{  
over A,B;  
A.set();  
A.display();  
cout<<“AfterA—”;  
—A;  
A.display();  
B.set();  
B.display();  
cout<<“After++”;  
++B;  
B.display();  
}
```

### Output:

Enter one number5

The number is5

AfterA—

The number is4

Enter one number

## Overloading Binary Operators

Here is a program to show binary operator overloading.

```
#include<iostream.h>  
  
class rectangle  
{  
public:  
int length;
```

```

int breadth;

rectangle(int length1,int breadth1)

{

length=length1;

breadth=breadth1;

}

int operator+(rectangle r1)

{

return(r1.length+length);

}

};

int main ()

{

rectangle r1(10,20);

rectangle r2(40,60);

int len;

len=r1+r2;

cout <<“The total length of the two rectangles is:”<<

len << endl;

return(0);

}

```

**The result of the program is:-**



```

C:\Documents and Settings\A\Desktop\bids\CBid\programs\Debug\operator overloading.
The total length of the two rectangles is : 50
Press any key to continue

```

The program consists of operator + function which is overloaded. The + operator is used to perform addition of the length of the objects. The statements

```
int operator+(rectangle r1)
```

```
{  
    return(r1.length+length);  
}
```

define the operator function whose return type is integer. The operator + is overloaded. The function is used to add the lengths of the two objects. The parameter list consists of one object of type rectangle. The operator() + takes only one parameter as the operand on the left side of the operator + is passed implicitly to the function through the this operator. The statement

```
len=r1+r2;
```

calls the operator() + function to calculate the length of the two objects. The return type is of integer. The variable len will contain the total of length of the objects.

## Advantages of Polymorphism

- a)** It allows objects to be more independent, though belong to the same family
- b)** New classes can be added to the family without changing the existing ones and they will have the basic same structure with/without added extra feature
- c)** It allows system to evolve over time, meeting the needs of a ever-changing application
- d)** ‘Polymorphism’ is an object oriented term. Polymorphism may be defined as the ability of related objects to respond to the same message with different, but appropriate actions. In other words, polymorphism means taking more than one form. Polymorphism leads to two important aspects in Object Oriented terminology - Function Overloading and Function Overriding. Overloading is the practice of supplying more than one definition for a given function name in the same scope. The compiler is left to pick the appropriate version of the function or operator based on the arguments with which it is called. Overriding refers to the modifications made in the sub class to the inherited methods from the base class to change their behavior.

## Exercise

### Short Questions

1. Explain polymorphism with an example.
2. Explain operator overloading with an example.
3. Explain function overloading with an example.

### Programming Exercise

1. Let us consider an example of an inventory of products in a store. One way of recording the details of the products is to record their names, code number, total items in the stock and the price of each item.

product\_name

product\_code

Unit\_Price

- (i) Define a class called ‘product’ with data items product\_name, product\_code, and Unit\_price. Save the class as a header file, i.e. \*\*.h.
- (ii) Define a constructor with parameter for dynamic initialization.
- (iii) Define a method for displaying prices of products.
- (iv) Write a main program to test the product system. Use the header file for the class defined in (i).

The above system has to be upgraded by adding a sales system. The sales information such as

Order\_no

Product\_code

Quantity

Total\_price

should be included.

- (v) Define a derived class called ‘Order’ which will have the above members.
- (vi) Define a method for displaying an invoice of an order as follows

| ABC Company Invoice Date: |

---

| Order no ||

| Product code ||

| Quantity @ price ||

---

| Total price | \*\*\* |

---

**(vii)** Write a main program to test the product system. Use the header file for the class defined in (v).

**2.** How many times is the copy constructor called in the following code:

```
Widget f (Widget u)
```

```
{
```

```
Widget v(u);
```

```
Widget w=v;
```

```
return w;
```

```
}
```

```
void main()
```

```
{
```

```
Widget x;
```

```
Widget y = f(f(x));
```

```
}
```

**3.** Write a SimpleCircle class declaration (only) with one member variable: itsRadius. Include a default constructor, a destructor, and accessor methods for itsRadius.

**4.** Using the class you created in Exercise 3, write the implementation of the default constructor, initializing itsRadius with the value 5.

**5.** Using the same class, add a second constructor that takes a value as its parameter and assigns that value to itsRadius.

6. Create a prefix and postfix increment operator for your `SimpleCircle` class that increments `itsRadius`.
7. Change `SimpleCircle` to store `itsRadius` on the free store, and fix the existing methods.
8. Provide a copy constructor for `SimpleCircle`.
9. Provide an `operator=` for `SimpleCircle`.
10. Write a program that creates two `SimpleCircle` objects. Use the default constructor on one and instantiate the other with the value 9. Call `increment` on each and then print their values. Finally, assign the second to the first and print its values.