2022

Object Oriented Programming using C++

Dr. Babasaheb Ambedkar Open University

Expert Committee

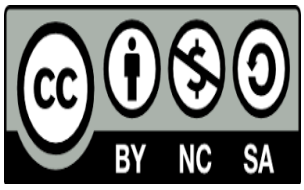
Prof. (Dr.) Nilesh K. Modi Professor and Director, School of Computer Science, Dr. Babasaheb Ambedkar Open University, Ahmedabad	(Chairman)
Prof. (Dr.) Ajay Parikh Professor and Head, Department of Computer Science Gujarat Vidyapith, Ahmedabad	(Member)
Prof. (Dr.) Satyen Parikh Dean, School of Computer Science and Application Ganpat University, Kherva, Mahesana	(Member)
M. T. Savaliya Associate Professor and Head Computer Engineering Department Vishwakarma Engineering College, Ahmedabad	(Member)
Mr. Nilesh Bokhani Assistant Professor, School of Computer Science, Dr. Babasaheb Ambedkar Open University, Ahmedabad	(Member)
Dr. Himanshu Patel Assistant Professor, School of Computer Science, Dr. Babasaheb Ambedkar Open University, Ahmedabad	(Member Secretary)

Course Writer

Dr. Sanjib Kr. Kalita	Krishna Kanta Handiqui State Open University
Arabinda Saikia	Krishna Kanta Handiqui State Open University
Dr. Tapashi Kashyap Das	Krishna Kanta Handiqui State Open University

Content Editor

Nilesh N. Bokhani	Assistant Professor, School of Computer Science, Dr. Babasaheb Ambedkar Open University, Ahmedabad



Acknowledgement: The content in this book is modifications based on the work created and shared by the Krishna Kanta Handiqui State Open University for the subject Object Oriented Programming through C++ used according to terms described under Creative Commons Attribution-Non Commercial-Share Alike 4.0 License

ISBN:

Printed and published by: Dr. Babasaheb Ambedkar Open University, Ahmedabad

Dr. Babasaheb Ambedkar Open University

BSCIT-204

Object Oriented Programming using C++

BLOCK-1:

BLOCK-2:

BLOCK-3:

BLOCK-4:

COURSE INTRODUCTION

Programming is a skill not a science. To develop this skill one needs to practice this programming fora few months. After practicing for a few months this skill will develop automatically. Moreover, programming is exercised with the help of some programming languages. Like natural languages programming languages also have grammar and syntax. Initially, the beginners need to develop two skills, first, the grammar and syntax of the language and, secondly, the programming skill i.e. the logic of a program. Every programming language has unique grammar, syntax and approach.

C++ is a case sensitive object oriented programming language. This course is designed and developed in such a way that it does not require any prior programming knowledge. Moreover, it does not require any knowledge of mathematics. The aim of this course is to provide knowledge of programming. All the features of OOPs are discussed with adequate examples. Before starting this course learners need to have one computer with C++ compiler, so that they can practice the programming practically.

There are two blocks in this course:

Block 1 deals with the basic concept of OOPs and C++ programming. After going through this block learners will be comfortable to learn block 2.

Block 2 concentrates on the implementation of the features of OOPs. All the features like class, objects, constructors, destructors, operator overloading, inheritance and polymorthism are discussed in this block.

Each unit of these blocks includes some along-side boxes to help you know some of the difficult, unseen terms. Some “EXERCISES” have been included to help you apply your own thoughts. You may find some boxes marked with: “LET US KNOW”. These boxes will provide you with some interesting and relevant additional information. Again, you will get “CHECK YOUR PROGRESS” questions. These have been designed to self-check your progress of study. It will be helpful for you if you solve the problems put in these boxes immediately after you go through the sections of the units and then match your answers with “ANSWERS TO CHECK YOUR PROGRESS” given at the end of each unit.

BLOCK INTRODUCTION

This course contains thirteen units in two blocks. **Block I** contains 7 units. These units discuss the basic concept of C++ programming. The Unit 1 deals with the basic concept of object oriented programming and introduces the learners to the concept of procedural programming, object oriented programming, benefits of OOPs etc. Unit 2 introduces object oriented programming features like abstraction, encapsulation, inheritance and polymorphism. The basic of C++ programming like data type, keywords, character set, variables, headers files are discussed in Unit 3. Operators and their precedence are discussed in Unit 4. Unit 5 discusses the control statements used for writing C++ program. Both the decision and loop control structures are discussed in this unit. Array, pointers and structures are discussed in unit 6. Unit 7 discusses the details about functions like user defined function, built in function, function call, function definition, declaration, actual parameter, formal parameter etc.

UNIT 1: INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

UNIT STRUCTURE

1.1 LEARNING OBJECTIVES

After going through this unit you will be able to:

- describe object and class in real world as well as in Object Oriented programming.
- define procedural programming and Object Oriented programming(OOP)
- describe relationship between data and functions.
- describe benefits of OOP
- describe OOPs programming languages
- describe how to write, save, compile and execute C++ program

1.2 INTRODUCTION

The real world entities are called **objects**, which are derived from some classes. An object may be physical or logical. For example, an wooden table is a physical object with several attributes or properties such as number of legs, color, weight, height etc. derived from class wood. But proper classification of class and object is a little bit confusing. Because, one may say that woods are objects derived from class tree, while another

may say that different trees are objects derived from class forest and so on. However, a **class** is a set of members or attributes or properties. The members of a class do not have any value, but they are capable of holding values or data. For example, if man is a class, then properties such as name, address, sex, age do not have any particular value. But, when objects are created from the class 'man' such as Hari, Jadu, Madhu etc. then each object will contain some attributes value. For example name is Hari, address is Guwahati, sex is male and age is 32. Attributes value for each and every objects may be different or same. The number of attributes of each object are same. There are two main categories of programming. They are

a. Procedural programming
b. Object oriented programming (OOP).

a) **Procedural programming** : Procedural programming is comparatively older than object oriented programming. The programming languages like BASIC, COBOL, Pascal, FORTRAN, C are based on procedural programming concept. In procedure oriented programming the problem to be solved is treated as a sequence of steps. It basically depends on function. For each activity a specific function is written. A function is a self contained program to perform some specific task. It is defined somewhere in the program and called from other location when it is required. Functions are normally used to manipulate or process data. Data may be declared as local or global. Local data are accessible from within the function only whereas global data are accessible from any function within that program. In a multi function program, many data items are placed as a global data so that they may be accessed by all the functions. In a large program, it is difficult to manage global data as many functions use the same data or the same variable. Figure 1 depicts the relationship between data and functions in procedure-oriented programming.

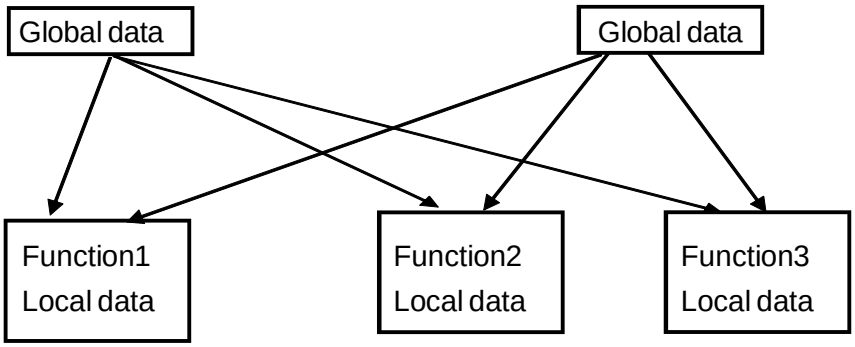


Fig. 1.1 : Relationship between data and functions

From Figure 1.1 it is clear that global data are accessible from all the functions of that program whereas local data are accessible within that function only. This reduces the reliability of global data. Another disadvantage of procedural programming is that it is not based on real world problems. The following are some characteristics of procedural programming

a. Emphasis is given on step by step solution
b. Programs are divided into subprograms known as functions
c. Functions shares global data
d. There is no concept of data hiding
e. Functions transform data from one form to another form
f. Top-down approach is used in program design

b) **Object Oriented Programming** : Like hardware industry, software industry is also changing day by day. Due to the technological advancement, programming platform has also shifted from procedural programming to object oriented programming. The primary concern of procedural programming was that it views data and functions in two separate entities, but in object oriented programming data and functions are viewed in a single entity. As a result, reusability of code is possible in OOP. The main advantages of OOP is that the programmer can create modular, reusable code. The modular program allows the programmers to modify a particular module without affecting the other modules. This increases the programmer productivity.

The basic objective of object oriented programming is to reduce the drawbacks of procedural oriented programming. OOPs consider data as a critical element and does not allow it to flow freely around the program. The access specifiers (private, public, protected) introduced in OOPs protect data from its external access. In OOPs data and functions are tightly bound to each other. Due to this reason data are not accessible from outside functions. OOPs permit to decompose a problem into subproblems depending upon the entities of the application. As an example, in a library of a college the common entities are books, borrower, issue, return etc. In OOPs data structure is designed based on the entities and the corresponding operations in the form of functions. The relationship between data and functions in object oriented programming have been depicted in Figure 1.2.

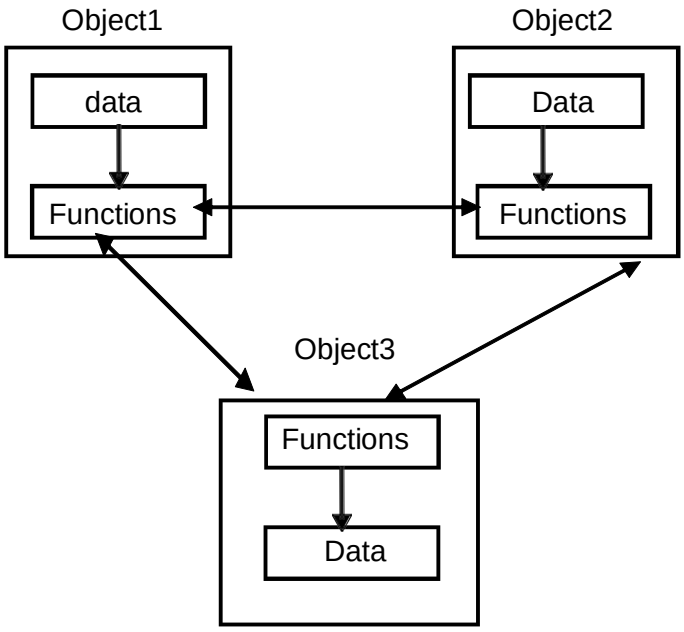


Fig. 1.2 : Relationship between data and function

Objects are created from class. As an example if animal is the class then tiger, lion, cow are the objects. A class is normally combination of data and function. A function associated with a particular class can be accessed through the object created from that class. Probably this features makes OOPs more secure. The following are the some common features of OOPs.

a. Focused on data rather than algorithm
b. Programs are based on objects
c. Data structures are based on entity
d. Functions and data are tied together
e. Hidden data are not accessible from external function
f. Objects may communicate to each other through functions
g. New data and functions can be added easily without affecting the existing code.

Note : In C++ programming, a class is a set of variables and functions. Variables are called **data member** and functions are called **member function**. Data members are used to store data and member functions are used to operate on data. The binding of data member and member function into a single unit is called **encapsulation**.

1.3 BENEFITS OF OOP

OOPs provide several benefits to both the system designer and user. Since the objects are independent entities and share with other objects through functions, so it provides better communication among objects. The independence of each object provides easy development and maintenance of a program. The main benefits of OOPs are

- New data and methods can be added at any stage of program development without affecting the existing system
- Code can be reused through inheritance.
- Data hiding is one of the major benefits of OOPs. It helps the programmer to build a secure program.
- Most of the OOP languages provide a standard class library that can be extended by the users, hence it can reduce programming time and effort. As a result programming productivity increases.
- Many OOP languages provide dynamic initalization of objects
- It is possible for multiple instances of an object to co-exist without any interference.
- It is easy to partition the work into a number of modules based on objects.

CHECK YOUR PROGRESS

1. Write True or False
   (i) Members of a class can have some values
   (ii) COBOL is object oriented programming language
   (iii) In procedural programming emphasis is given on step by step solution
   (iv) In OOP data and functions are same
   (v) In OOPs Objects may communicate to each other through functions

1.4 OOP LANGUAGES

All programming languages are based on some programming techniques. In OOP programming the attention is given on objects. A

language that is specially designed to support the OOP concepts makes it easier to implement them. A language that supports several OOP concepts is called object oriented language. OOP languages are classified into two categories based upon the features they support.

a. Object based programming languages

b. Object oriented programming languages.

a. ***Object based programming*** style basically supports data encapsulation and object identity. The following are the major features supported by object based programming languages

i. Data encapsulation

ii. Data hiding and access mechanism

iii. Automatic and dynamic initialization of objects

iv. Operator overloading

b. ***Object oriented programming*** language incorporates all the object based programming features along with the following two features.

i. Inheritance

ii. Dynamic binding

The following are some object oriented languages

1.4.1 C++

The C++ language was developed at AT & T Bell laboratories as an extension of C language in 1980s. Bjarne Stroustrup was the founder of C++. Initally it was not so powerful as it is today. The language was developed taking the idea from C, Simula67 and ALGOL68 programming languages. Initially C++ was known as “C with Classes”. In the initial version of “C with Classes” features like operator overloading, references and virtual functions are not available. Later, it was added in C++. The name C++(pronounced as C plus plus) was proposed by Rick Mascitti in 1983. Some of the common features of C++ are *classes, operator and function overloading, free storage management, references, inline functions, inheritance, virtual functions, streams for console and file manipulation, templates and exception handling*. In C++, variables are known as data member and functions are known as member function. Both the data members and member functions are grouped into a single unit known as class. Memebers of a class

may be declared as *private, protected and public*. These three keywords are called ***access specifiers***. By default all members are private. The private members of a class are accessible by the member function of that class. In other words, private members are not accessible from outside the class definition. Public members are accessible from outside the class definiton, but it violates the ***data hiding*** property. The term '**data hiding**' means that data is there but not accessible. Protected members are also not accessible from outside the class definiton, but it is inheritable. Inheritance is the feature through which code and members can be reused in other classes. Another special member function whose name is exactly same with class name is called ***constructor***. Constructors are used to initialize new objects. Like constructor, another special member function used to destroy objects is called ***destructor***.

1.4.2 Smalltalk

Smalltalk is the first object oriented language. It was developed at Xerox's Palo Alto Research Centre(PARC) in around 1970. It introduces the basic concept of OOPs like objects, class, message, method and inheritance. Smalltalk is efficient, portable, easy to use and reliable. Smalltalk introduces the concept of mouse programming, bit-mapped graphics display, windowing systems, object oriented programming, user interface etc. Due to the licensing restrictions and high cost the original Smalltalk-80 was not circulated to the common people. It was basically used in research and academic circles. The concept of functions and objects are similar to C++.

1.4.3 Java

Java is another object oriented programming language. It was developed by Sun Microsystems in 1995. The latest release of the Java Standard Edition is Java SE 8. The primary objective of Java was **Write Once, Run Anywhere**. The following are some common features of Java:

a. **Object Oriented:** Java is pure object oriented. All programming should be written within a class and it has to be executed with the help of objects of that class.

b. **Platform Independent:** Java is platform independent. Its means that the program written in one machine(specific architecture) can execute in other machine(different architecture). This is because when Java is compiled, it is not compiled into platform specific machine; rather, it produced byte code. This byte code is distributed over the web and interpreted by the Virtual Machine.

c. **Simple:** It is designed to be easy to learn. Anyone who has the programming idea in C++ can learn Java easily.

d. **Secure:** You can use java compatible web browser to safely download java applets without fear of viral infection or malicious intent.

e. **Architecture-neutral:** Java compiler generates a machine independent object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.

f. **Portable:** In addition to the architecture-neutral features java programs are also portable. This is one of the important design aspect of java that it can be portable.

g. **Multithreaded:** Java is multithreaded programming languages. With this feature it is possible to write programs that can perform many tasks simultaneously.

h. **Distributed:** Java is designed for the distributed environment of the internet.

1.5 ELEMENTS OF OBJECT ORIENTED PROGRAMMING

The following are some components of object oriented programming

1.5.1 Objects

Any logical or physical entities are called objects. The entities which can touch through our hand are called physical entities. On the

other hand logical entities are not touchable with our hand like bank account, salary, a job etc. Figure 1.3 depicts some example of physical objects. Each object has some specific attributes. Figure 1.4 shows some object names and their corresponding attributes.

BOY

DOG

BOOK

Fig 1.3 : Examples of physical objects

Object : BOY
name
Weight
Height

Object : DOG
Type
Weight
Color

Object : BOOK
Author
Name
Publisher

Fig 1.4 : Object names and attributes

Every object has some attributes and and associated operations. If we consider *account* as an object then *Accountnumber, Accounttype, name, balance* are the attributes and *deposite, withdraw* are the operations. Objects, attributes and operations are created from real world applications.

1.5.2 Classes

A class is a data structure that consists of a set of attributes and operations. Attributes are called data member and operations are called member function.

1.6 HOW TO WRITE/SAVE/COMPILE and EXECUTE C++ PROGRAMS

Most compilers have their own interface and editor except a few such as COBOL, JAVA. C++ compiler has also its own editor. Writing a program and execution is done in the following sequences-

1. Writing the program.
2. Saving the program.(a file name with **.cpp** extension)
3. Compile the program.(generate two more file with **.obj** and **.exe** extension)
4. Linking the program with functions and
5. Executing the program.

Although these steps remain the same irrespective of the operating system, the system commands used are different for different systems. If you have **MSDOS** operating system and TURBO C compiler, then the required steps are

C:\>
C:\>cd tc [or turboc]
C:\tc>tc(Here press enter key from the keyboard)

To highlight the menu option press F10 and select File->New. After writing the program press F2 to save the program. Before executing the program one should save the newly created program so that there is no chance of loosing the program.The following keys are pressed to compile, execute the program.

F9 - Compile
Ctrl+F9- Execute.
While using **Microsoft Windows** the steps would be
Start->program->MS DOS Prompt
C:\windows>
C:\windows>cd..
C:\>cd tc[or turboc]
C:\tc>tc(Here press enter key from the keyboard)

The program written by the programmer is known as **source program.** The source program is then converted to **object program** provided that there is no syntax error. The object programs are generated automatically after successful compilation of the source program.

Working with UNIX or LINUX environment is a little bit different from DOS and Windows environments. The commercial C-compiler for DOS and UNIX are different with same functional capabilities. In UNIX operating system programs are created using some text editor like **ed** or **vi.** UNIX text editors have two working mode- one is **text mode** and the other is **command mode**. Programs are written using text mode and saved by using command mode. **Esc** key is used to switch from text mode to

command mode and *i* or *a* is used to get the text mode. Capital letters I or A are not used as UNIX command- since the UNIX commands are case sensitive. In UNIX compilation is performed with **cc** command as follows—

cc *test.cpp* (Where cc UNIX command and test.cpp is the file name)

After successful compilation of the source program **object file** or **program** is generated with **.o** extension. Object files are then converted to **executable object file** with the name **a.out**. The command for executing a C program in UNIX environment is **a.out**.

1.7 LET US SUM UP

- A class is a set of data member and member function
- Objects are run time entity of class
- There are two main categories of programming. They are
  a. Procedural programming
  b. Object oriented programming(OOP)
- For procedural programming emphasis is on data and function
- For object oriented programming(OOP) emphasis is on objects.
- The features of OOP's are data encapsulation, data hiding, automatic and dynamic initialization of objects, operator overloading, inheritance, dynamic binding, polymorphism etc.

1.8 FURTHER READING

- Venugopal, K.P. (2013). *Mastering C++.* Tata McGraw-Hill Education
- Balagurusami, E. (2001). *Object Oriented Programming with C++,* 6e. Tata McGraw-Hill Education

1.9 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q. No. 1

(i) False

(ii) False

(iii) True

(iv) False

(v) True

1.10 MODEL QUESTIONS

1. What is object oriented language?
2. The name C++(C plus plus) was proposed by .....
3. Is it possible to access private data member from outside the class definiton?
4. What are the features of C++?
5. What are the features of Java?
6. Why OOP's are preferable to procedural oriented language?
7. Give some real life examples of class, objects and operations.

****

UNIT 2: FEATURES OF OBJECT ORIENTED PROGRAMMING

UNIT STRUCTURE

2.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- learn about the different features like abstraction, encapsulation, inheritance, polymorphism.
- describe the different forms of Inheritance
- explain the need and advantages of Inheritance

2.2 INTRODUCTION

In the previous unit we have learnt about the basic concept of programming languages like procedural or object oriented languages. Every language has its own properties. Most of the object oriented languages have some common properties like inheritance, encapsulation, polymorphism etc. In this unit you will be able to learn about these properties in brief. In the subsequent units you will be able to learn these features in detail

2.3 INHERITANCE

Inheritance is a technique of creating a new class from an existing one. It is like copying the existing codes from one location to another location during run time. In real life, attributes of one class can be inherited by another new class. For example, the face appearance of father may be inherit ed by his child. Are all properties of the father inheritable to his child? Definately no. For example, the qualification of the father is not inheritable to his child. In C++, a class is a set of data member and member functions. These members can be inherited from one existing class to another new class. The existing class is called the **base class** or **parent class** and the new one is called the **derived class**. The derived class may contain some members from the base class and some from its own. A class can also inherit properties from more than one class or from more than one level. The derived class can inherit one or all members of the base class. The main advantage of inheritance is to reduce the number of instructions in the program and hence it reduces the programming time, effort and required memory space.

2.3.1 Forms of Inheritance

Inheritance is classified into the following forms based on the levels of inheritance.

(i) Single Inheritance
(ii) Multiple Inheritance
(iii) Hierarchical Inheritance
(iv) Multilevel Inheritance
(v) Hybrid Inheritance

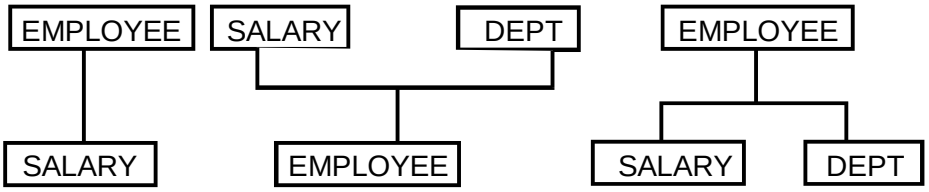
(i) Single Inheritance : A derived class with only one base class is called single inheritance.

(ii) Multiple Inheritance : A derived class with several base classes is called multiple inheritance.

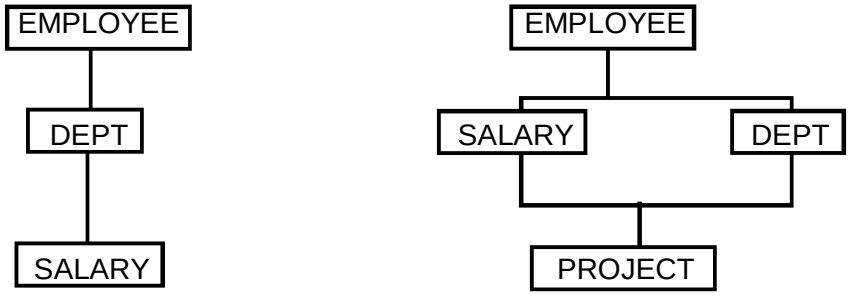
(iii) Hierarchical Inheritance : Several derived classes with only one base class is called hierarchical inheritance.

(iv) Multilevel Inheritance : The techniques of deriving a class from another derived class is called multilevel inheritance.

(v) Hybrid Inheritance : Inheritance that involves two or more forms of inheritance is called hybrid inheritance.



(i) Single inheritance (ii) Multiple inheritance (iii) Hierarchical inheritance



(iv) Multilevel inheritance (v) Hybrid inheritance

Fig 2.1 Forms of Inheritance

2.4 DEFINIING THE DERIVED CLASS

A derived class is a class that contains its own data members and member functions and some from its base class. Derived class is defined by specifying its relationship with the base class. The general form or syntax of defining a derived class is:

```
class derived_class_name : Visibility_mode base_class_name
{
        data members and member functions of derived class;
};
```

There are two visibility mode in C++ **private** and **public**. It specifies, whether members of the base class are derived privately or publicly. The use of visibility mode in derived class definition is **optional**. The **default** visibility mode is private.

Examples

class salary:**private employee** //private derivation of members from class employee

```
{
    data members and member functions of class salary;
};
```

```
class salary : public employee//public derivation of members from class
employee
{
        data members and member functions of class salary;
};

class salary : employee//private derivation of members from  class
employee
{
        data members and member functions of class salary;
};
```

As discussed earlier, members of a base class can be derived privately or publicly. When public members of a base class are inherited as public, then public members of the base class become public in the derived class and they are accessible from derived class objects. On the other hand, if public members of a base class are inherited as private, then the public members of the base class become private in the derived class and they are not accessible from derived class objects. Private members of a base class are not inherited directly. Inheritance of derived class members to its base class is impossible.

2.4.1 MAKING PRIVATE MEMBER INHERITABLE

As discussed earlier, private members of the base class are not inheritable to its derived class. One way to inherit a member is by changing the limit of visibility from private to public. Public members are inheritable to its derived class but violating the data hiding property. Fortunately, C++ provides another visibility modifier, **protected**, whose scope is in between private and public. Protected members are inheritable like public members but not accessible from outside the class definition like private member. When a **protected** member is inherited in **public** mode, it becomes **protected** in the derived class, and accessible by the member functions of the derived class. It is also ready for further inheritance. On the other hand, when a protected member is inherited in **private** mode, it becomes **private** in the derived class. It is available to the member functions of the derived class, but not ready for further inheritance.

Table-2.1 : Visibility of inherited members

Base class visibility	Derived class visibility	
	Public derivation	Private derivation
Private	Not inherited	Not inherited
Protected	Protected	Private
Public	Public	Private

The keywords **private**, **protected** and **public** may appear in any order and in any number of times in a program.

2.5 VIRTUAL BASE CLASS

A base class that is derived with the keywork 'virtual' is called virtual base class. The virtual base class concept is essential when one uses multiple and multilevel inheritance in a single program.Consider a situation where some employees works for a particular project and they belong to some department. At the end of the project it is required to process their bill. This situation may be depicted as in Fig. 2.2. The BILL has two direct base classes 'PROJECT' and 'DEPT' which have a common base class EMPLOYEE. The BILL class inherits the members of class EMPLOYEE through PROJECT and DEPT class. Inheritance by the BILL class will create a problem of duplicate data. All the public and protected memebrs of EMPLOYEE class are inherited into BILL twice via PROJECT and DEPT. This will generate duplicate data or ambigity in BILL class.This ambiguity can be removed by making the common base class as virtual base class. Remember virtual is a keyword. Virtual base class takes the necessary action to remove this ambiguity. We will discuss virtual base class in subsequent units.

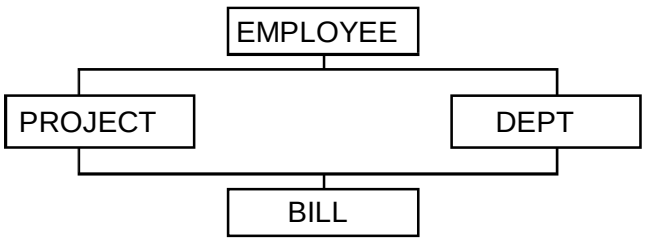


Fig 2.2 : Multiple base class

2.6 ENCAPSULATION

The grouping of data and functions into a single unit is called **encapsulation**. Data encapsulation is the most powerful features of C++. The functions which are grouped with the data are permitted to access that data. No outside function can access data.

2.7 POLYMORPHISM

Polymorphism means one name many forms. It is one of the important features of OOPs. It can be classified as depicted in Figure 2.3

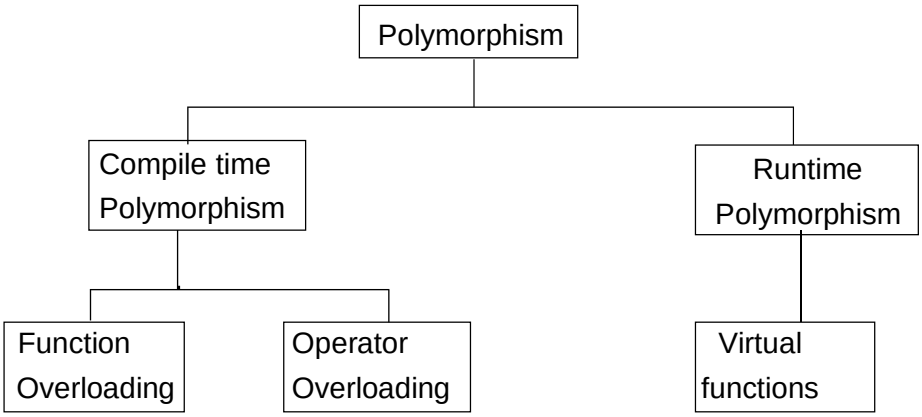


Fig. 2.3 : Forms of Polymorphism

2.7.1 Function Overloading

Overloading means the same word or symbol with different meanings. C++ supports the concept of **function overloading** or **function polymorphism** i.e. different functions with the same function name which performs different tasks. To differentiate one function form another one must consider the list of arguments. The function would perform different operations depending on the argument list in the function call. This fact can be explained with the help of an example. Suppose there are three students with same name and title(say Dilip Medhi) in a class room. The class teacher is decided to call one particular student for a particular task. The basic problem for the teacher is that he can not call a particular student name sharing the same with the two others because it

causes confusion to the students. The teacher has to take some extra parameters such as color of shirt, fathers name, address etc. Atleast one parameter must be different from one student to another to identify them. Similarly, in C++ function overloading, atleast one parameter must be different from one function to another to make them unique. The overloaded functions must declare **globally**. When calling a function, the number of actual parameter and the number of formal parameter should match in type and their order. In some situation, type of parameters may not match, then the function selection involves the following steps:

(a) First, the compiler tries to find out the match for type of actual parameter and formal parameter and their order.

(b) If match is not found, then the compiler tries to match as follows
char to **int**(i.e. char in actual and int in formal)
float to **double**(i.e. float in actual and double in formal)

(c) When either of them fails, the compiler tries to use the built-in conversions to the actual arguments.

2.7.2 Operator Overloading

Overloading means many forms for different activities with one common name. Operator overloading is one of the key features of C++ language. It has the capability to provide the operators such as +, - etc. with a special meaning for a data type. The technique of giving such special meanings to an operator is known as operator overloading. However, there are certain operators which cannot be overload. C++ supports several operators to be overloaded as stated in Table 2.2

Table 2.2 : List of operators to overload

Operators can be overloaded	Operators can not be overloaded
Unary : ++ -- Arithmetic : + - / * Relational : < > == Shorthand :+= -= *= /= Stream :<< >>	Members access operators (., .*) Scope resolution operator(::) Size operator(sizeof) Conditional operator(?:)

Subscript :( ) [ ] Memory : new and delete Arrow : ->* Comma : ,	

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special member function, called operator function. These functions should be either member function or friend function. The basic difference between them is that a friend function accepts only one argument for unary operators and two for binary operators, while a member function has no argument for unary operators and only one for binary operators.

Overloading without explicit arguments to an operator function is known as *unary operator overloading* and overloading with a single explicit argument is known as *binary operator overloading*.

Syntax for operator overloading:

```
Return type operator  operatorsymbol  ([arg1, [arg2]])
```

Examples

(a) Unary operators

(i) test operator +()

(ii) int operator –()

(iii) void operator ++()

(iv) void operator – –()

(v) int operator *()

(b) Binary operator overloading

(i) complex operator +(complex c1);

(ii) int operator –(int a);

(iii) void operator *(complex c1);

(iv) void operator /(complex c1);

(v) complex operator +=(complex c1);

CHECK YOUR PROGRESS

1.Write True or False

(i) All members of a class are inheritable

(ii) Private members can be inherited

(iii) Binary operator can be overloaded

(iv) Unary operator can not be overloaded

(v) Virtual is a keyword

(vi) Virtual base class is used to remove ambiguity.

2.8 LET US SUM UP

- Most of the object oriented languages have some common properties like inheritance, encapsulation, polymorphism etc.
- Inheritance is the process of creating a new class from the existing one.
- Private data are not inheritable.
- There are different types of inheritance like
  - (i) Single Inheritance
  - (ii) Multiple Inheritance
  - (iii) Hierarchical Inheritance
  - (iv) Multilevel Inheritance and
  - (v) Hybrid Inheritance
- In inheritance the parent class is called base class and the child class is called derived class.
- There are three visibility modes like ***private, protected and public***.
- The grouping of data and functions into a single unit is called encapsulation.
- A base class that is derived with the keywork 'virtual' is called virtual base class.
- The basic idea of operator overloading and function overloading

2.9 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education

2.10 ANSWERS TO CHECK YOUR PROGRESS

1. (i) False
   (ii) False
   (iii) True
   (iv) False
   (v) True
   (vi) True

2.11 MODEL QUESTIONS

1. Explain the different features of object oriented languages
2. What is inheritance? What are the different forms of inheritance?
3. What is virtual base class? Why is it necessary?
4. What is polymorphism? What are the different types of polymorphism?
5. Mention the names of operators that can be overloaded
6. Mention the names of operators that cannot be overloaded
7. What is function overloading? Mention the implicit rules in function overloading.

UNIT 3 : ELEMENTS OF C++ LANGUAGE

UNIT STRUCTURE

3.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- learn about the basic elements of C++, viz. character set, token, identifier and keywords
- declare variables and constants
- describe dynamic initialization of variable, and reference variables

3.2 INTRODUCTION

Every programming language has its own syntax and rules. So, before writing any error free program it is necessary to know the rules of syntax of the language.

In this unit, we will introduce you to the some basic elements of C++ language including character set, token, identifier, keywords etc. We will also discuss the variables, constants and reference variable. In addition, we will also see how a variable can be initialized dynamically.

3.3 TOKEN, IDENTIFIER AND KEYWORDS

TOKEN

A C++ program contains various components. The individual elements in a program is identified as a token by the compiler. Tokens are classified in the following types:

- Keywords
- Variables
- Constants
- Special character
- Operators

Keywords are a set of reserved words with fixed meanings e.g., *int, switch, char, class* etc.

Variables are used to hold data temporarily, e.g., *marks, age, name* etc.

Constants are fixed values like 3.2, 9.3 etc.

Special characters are symbols like #, ~ are known as special character

Operators are used to perform different operations such as arithmetic or logic etc. e.g. +, -, :, ?, >, < etc.

IDENTIFIER

Identifiers are the names that are given to various program elements, such as variables, functions and arrays. Identifiers consist of letters and digits, in any order, except that the first character must be a letter. To construct an identifier you must obey the following points:

- only alphabet, digit and lender scores are permitted.
- an identifier cannot start with a digit.
- identifiers are case sensitive, i.e. upper case and lower case letters are distinct.

In C++, there is no limit to the length of an identifier and at least the first 1024 characters are significant. Some correct and incorrect identifiers name given here :

Correct	Incorrect
count	1 count
names	#$sum
tax_rate	order-no
_temp	high.....balance

An identifier cannot be the same as a C or C++ keyword, and should not have the same name as functions that are in the C or C++ library.

KEYWORDS

Reserved words are the essential part of language definition. The meaning of these words has already been explained to the complier. So, you can't use these reserved words as a variable name. All C keywords are valid in C++. There are 63 keywords in C++.

The common keywords in C and C++ are listed in Table 3.1. Table describes the additional keywords of C++.

Table 3.1 : C & C++ common keywords

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Table 3.2 : Additional C++ keywords

asm	private	bool	wchar_t
catch	protected	mutable	explicit
class	public	typename	static_cast
delete	template	const_cast	export
friend	this	namespace	true
inline	throw	using	false
new	try	dynamic_cast typeid	
oprator		virtual reinterpret_cast	

3.4 CHARACTER SET AND SYMBOLS

Using the valid character set and symbols a source program is created. The following is the valid list of character set and symbols in C++.

Alphabets	A to Z,
	a to z and _(under score)
Digits	0 to 9
Special symbols	# , & | ! ? ~ ^ { } [ ] ( ) < > . : ; $ ‘ “ + - /
	* = % blank \

3.5 BASIC DATA TYPES IN C++

C++ supports a wide variety of data types. We can choose the appropriate type for writing error free programs. The data types in C++ can be classified in the following categories.

Basic Data Type	Derived Data Type	User Defined Data Types
Char	*Array*	*Structure*
Int	*Function*	*Union*
Float	*Pointer*	*Class*
Double	*Reference*	

We will concentrate on the discussion of basic data types in this unit. The derived data types such as arrays and pointers and the user defined data types such as structure, union and classes are being discussed in next units.Table 3.3 lists all combinations of the basic data types and modifiers along with their size and ranges:

Table 3.3 : Basic data types and size

Type	Size (Bytes)	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767

unsigned short int	2	0 to 65535
signed sort int	4	-32768 to 32767
long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
signed long int	4	-2147483648 to 2147483647
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E-308
long float	10	3.4E-4932 to 1.1E+4932

3.6 VARIABLES

A variable is an entity that is used to represent some specified type of information within a program and whose value can be changed during the execution of the program. A variable is denoted by giving a name to it. A variable declaration associates a memory location to the variable name. It means whenever we declare a variable, a definite amount of memory location is reserved for it. The main factors associated with a variable are as follows–

- Date type – type of the data it stores i.e. char, int, float, date (user defined) etc.
- Variable Name – the name that is given to the variable by programmer.
- Address – address of the memory location.
- Value – data stored in memory locations.

We have already learnt in C programming, how to give names to a variable, how to declare a variable and how to initialize values to a variable. Let us discuss it briefly.

Variable Name

Variable names are indentifiers used to name variables. They are the symbolic names assigned to memory locations. A variable name consists of a sequence of letters and digits. The first one must be a letter. Examples of some valid variable names are:

x	sum	count
name	student_name MAX	
age	_num	dept_num

Some invalid variable names are:

2 slum	–	first character should be a letter
date birth	–	blank is not allowed
Emp, record	–	, is not allowed
student--age	–	illegal character (-)

Variable Declaration :

We know that a variable must be declared before using it in a program. Actually this declaration process reserves memory depending on the type of the variable. Syntax for declaring a variable is shown below -

Data type VarName1, ... VarName n;

The following are some valid variable declaration statements:

```
int x;          // x is an integer variable
int m, n, q;    // m, n, q are integer variables
float root1, root2   // root1, root2 are floating point variables
```

Variables can also be declared at the point of their usage as follows :

```
for ( int i = 0; i < 10; i++ )
    count << i;
int d = 10;
```

Here, variable *i* and *d* are defined at the point of their usage.

Variable Initialization

A variable can be assigned with a value during its declaration. The assignment operator (=) is used in this case. The following syntax shows how a variable initialized.

Data-type VariableName = constant value;

The following are the valid initialization statements :

int a = 20

float x = 2.25, y = 6.0925;

The following program demonstrates the initialization of variables.

// Program 3.1 : initialization of variables

```
#include<iostream.h>
#include<conio.h>
void main( )
{
 int x, y;// x and y are integer type variables
 int z = 75;//75 is initialized to integer variables z
```

```
float average;
clrscr ( );
x = z;
y =z+50;// value of z is added with 50 and assigned to y
average = 5.125;
cout <<"x=" <<x << "\n";
cout << "y =" <<y <<"\n";
cout<<"z="<<z<<"\n";
cout <<"average=" <<average <<"\n";
getch ( );
}
```

OUTPUT : x = 75
y = 125
z = 75
average = 5.125

Here, the statement cout << "x=" <<x <<"\n";
displays a message 'x=' followed by the contents of the variable x and then a new line. We will discuss the input and output operations of C++, in the next section of this unit.

3.7 CONSTANTS

The constants in C++ are applicable to the values which do not change during the execution of a program. C++ has two types of constants

(i) literal constants
(ii) symbolic constants

i) Literal constant

A literal constant is just a value. For example, 10 is a literal constant. It does not have a name, just a literal value.

For example, `int x = 100;`

where x is a variable of type int and 10 is a literal constant. We cannot use 10 to store another integer value and its value cannot be altered. The literal constant does not hold memory location. Depending on the type of data, literal constants are of the following types shown with examples below

Example	Constant Type
547	Integer constant
65.125	Floating point constant
0x98	Hexadecimal integer constant
0175	Octal integer constants
'a'	Character constant
"Student Name"	String constant
"1024"	String constant

Remember that a character constant is always enclosed with single quotation mark, whereas a string constant is always enclosed with a double quotation mark. Another point to remember is that an octal integer constant always starts with *0* and a hexadecimal integer constant with *0x*.

ii) Symbolic constant

A symbolic constant is defined in the same way as variable. However, after initialization of constants the assigned value cannot be altered. The constant can be defined in the following three ways :

a) # define

b) The *const* keyword

c) The *enum* keyword

a) # *define* *:* The # define preprocessor directive can be used for defining constants as

define Maximum 100

define PI 3.142

define AGE 30

In the above example, *Maximum, PI, AGE* symbolic constants contain the value 100, 3.142 and 30 and here it is not mentioned whether the type is *int, float or char*. Every time when the preprocessor finds the word Maximum, PI, AGE, it will just substitute it with the values 100, 3.142 and 30 respectively.

The following program demonstrates the use of #define–

```
// Program 3.2
#include<iostream.h>
#include<conio.h>
#define PI 3.142
void main ( )
```

```
    {
        float radius, area;
        clrscr ( );
        cout <<"Enter the radius :";
        cin >> radius;
        area = PI * radius * radius;
        cout << "Area of the circle =" <<area <<"\n";
        getch ( );
    }
```

Output :

Enter the radius : 2.5
Area of the circle = 19.6375

In the above program the statement

area = PI * radius * radius;

is translated by the preprocessor as

area = 3.142 * radius * radius;

and are calculated result is stored in the variable 'area' which is displayed in the next statement.

***b) const* keyword** : The syntax of defining variables with the ***const* keyword** is shown below :

const [data type] variable name = constant value;

The following examples illustrate the declaration of constant variable :

```
const float p1 = 3.142;
const int TRUE = 1;
const int FALSE = 0;
```

The following program demonstrates the use of constant variable and its declaration :

// Program 3.3

```
#include< iostream.h >
#include< conio.h >
const int MAX = 5;
void main ( )
    {
    int i ;
    clrscr ( ) ;
    for ( i = 1; i < = MAX ; i+ + )
```

```
    cout << " The loop runs for =" << i << "times" <<
"/n" ;
            getch( ) ;
        }
```

Output:

The loop runs for 1 times
The loop runs for 2 times
The loop runs for 3 times
The loop runs for 4 times
The loop runs for 5 times

In the above program, the *for* loop will run for 5 times because the MAX variable contains the constant value 5.

c) *enum keyword* : enum is a keyword to assign constant to a variable. Constants can be defined using enumeration as given below :

Example :

enum { a,b,c };

Here a,b and c are declared as integer constants with value 0,1 and 2.

We can also assign new values to a, b and c

enum { a = 5,b =10, c = 15 } ;

Here, a, b and c are declared as integer constants with value 5,10 and 15.

3.8 DYNAMIC INITIALIZATION OF VARIABLE

The declaration and initialization of variable in a single statement at any place in the program is called as *dynamic initialization*. The dynamic initialization is always accomplished at run time i.e., when program execution is going on . Dynamic means process carried out at run time, for example, dynamic initialization, dynamic memory allocation etc.

The C++ compiler allows declaration and initialization of variables at any place in the program. In C initialization of variables can be done at the beginning of the program, but in C++ initialization of variables can be done anywhere in the program.

The following program illustrates the dynamic initialization of variables in C++.

// Program 3.4

```
#include<iostream.h>
#include<conio.h>
void main()
    {
        clrscr ( ) ;
        cout << "Enter radius : / n";
        int r ;
        cin >> r ;
        float area = 3.14 * r * r ;
        cout << "/n Area = "<< area ;
        getch ( ) ;
    }
```

Output :

Enter radius = 3

Area = 28.26

In the above program, variable 'r' and area are declared inside the program. In the statement *float area = 3.14 * r * r ;* variable *area* is declared and initialize with the value *3.14 * r * r*. This assignment is carried out at run time. Such type of declaration and initialization of a variable is called as *dynamic initialization*.

3.9 REFERENCE VARIABLE

C++ supports another type of variable called *reference variable*. A reference variable acts as an alternative (alias) name for a previously defined variable. Recall that a variable holds only a value and we have already learnt from C programming that pointer variables are used to hold the address of some other variables. A reference variable behaves similar as an ordinary variable and also as a pointer variable. Inside a program code it is used as an ordinary variable but it acts as a pointer variable. The syntax for declaring a reference variable is shown below:

Data type & Reference variable name = variable name;

Example :

```
int sum = 100;
int & totalsum = sum;
```

Here, the variable sum is already declared and initialized. The second

statement defines an alternative variable name i.e. *totalsum* to variable sum. Both the variables will display the same value, any change made in one of the variables causes change in both the variables.

The following are some examples of reference variable–

```
char ch;            float   m;
char & ch 1 = ch    float & n = m;
```

The following program illustrates the use of reference variables.

//Program 3.5

```
#include<iostream.h>
#include<conio.h>
void main( )
    {
        int x = 10, y = 11, z = 12;
        clrscr ( );
        int & m = x; // variable m becomes alias of x
cout <<"x =" <<x <<"y =" <<y <<"z=" <<z <<"m=" <<m<<"\n";
m = y;//changes value of x to value of y
Cout<<"x="<<x<<"y="<<y<<"z="<<z<<"m="<<m<<"\n";
m = z;  // changes value of x to value of z

cout<<"x="<<x<<"y="<<y<<"z="<<z<<"m="<<m<<"\n";
        getch ( );
    }
```

Output:

x = 10 y = 11 z = 12 m = 10
x = 11 y = 11 z = 12 m = 11
x = 12 y = 11 z = 12 m = 12

From the above program we have seen that any change made to the reference variable *m* also reflects in the variable *x.*

3.10 STREAMS IN C++

Generally, every program involves in the process reading data from input device - computation is done on the data - sending the result to output devices. Hence, to control such operations every language provides a set

of built-in functions. C++ also supports a rich set of functions for performing input and output operations. These C++ I/O functions make it possible for the user to work with different types of devices such as keyword, monitor, disk, tape drives etc. It is designed to provide a consistent and devise independent interface. These I/O functions are part of standard library. A library is nothing but a set of ***.obj*** (object) files.

Now, we have come to know that the data flows from an input device to programs and from programs to output device. In C++, a stream is used to refer to the flow of data in bytes in sequence. If data is received from input devices in sequence then it is called as ***source stream*** and when the data is passed to output devices then it is called as ***destination stream***. The data is received from keyword or disk and can be passed on to monitor or to the disk. The following figure describes the concept of stream with input and output devices.

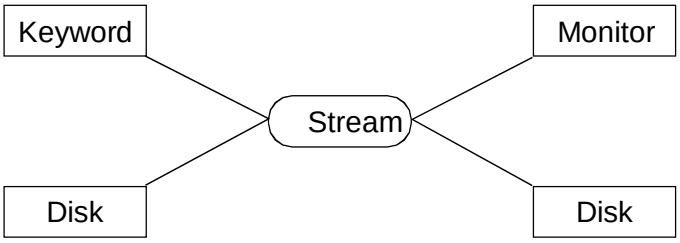


Fig. 3.1 : Streams with I/O devices

Data in source stream can be used as input data by program. So the source stream is also called as ***input stream***. The destination stream that collects output data from the program is known as ***output stream***. The mechanism of input and output stream is illustrated in figure 3.2.

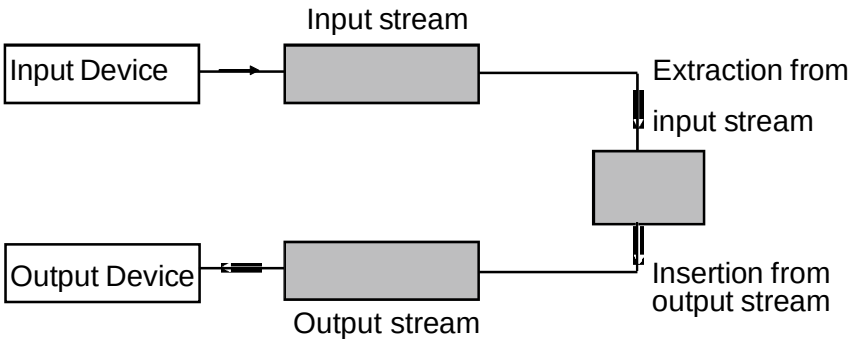


Fig. 3.2 : Input and output streams

Thus, the stream acts as an intermediator or interface between I/O devices

and the user. The input stream pulls the data from keyword or storage devices such as hard disk, floppy disk etc. The data present in output stream is passed on to the output devices such as monitor, printer etc.

C++ has a number of predefined streams that are also called standard I/O objects. These streams are automatically activated when the program execution starts. The four standard streams ***cin, cout, cerr*** and ***clog*** are automatically opened before the function *main( )* is executed; they are closed after *main( )* has completed. These predefined stream objects are declared in the header file ***iostream.h***. In this unit, we will concentrate on the discussion about *cin and cout*.

Output stream

The output stream allows to perform write operation on output devices such as monitor, disk etc. Output on the standard stream is performed using the ***cout*** object. The syntax for standard output operation is as follows:

```
cout << variable;
```

The *cout* object is followed by the symbol **<<** which is called the ***insertion operator*** and then the items (it may be variable/constants/expressions) that are to be displayed.

The following are examples of stream output operation:

```
cout << "KKHSOU" ;
cout << "BCA 3rd semester" ;
float area ;
cout << area ;
char code ;
cout << code ;
```

More then one item can be displayed using a single *cout* output stream object. Such output operations are called *cascaded output operations*. As an example in the above programs 3.1, 3.2, 3.4 we have already used cascaded output operation. The following are some statements which we have used in our previous programs

cout << "Area =" << area ; and

cout << "x = "<< x << " y = " << y << " z = "<< z << " m = "<< m << " \n ";

The *cout* object will display all the items from left to right, we have shown in RUN portion of the program. In the first statement it will first display " Area = " and then will display the value of the `area` variable which will be finally

Area = 28.26

The second statement will be displayed as–

x = 10 y = 11 z = 12 m = 10

Where 10,11,12,10 are the value of the variable x, y, z and m.

The complete syntax of standard output stream operation is as follows :

cout << variable1 << variable2 << .....<< variableN ;

Input stream :

The input stream allows to perform read operation through input devices such as keyboard, disk etc. Input from the standard stream is performed using the ***cin*** object. The syntax for standard input operation is as follows :

cin >> variable ;

The *cin* object is followed by the symbol **>>** which is called the ***extraction operator*** and then the variable, into which the input data is to be stored. The following are some example of standard input operations :

```
int r ;
cin >> r ;
float radius ;
cin>>radius;
char name [25] ;
cin >> name ;
```

Using the *cin* input stream object inputting of more then one item can also be performed. The complete syntax of the standard input stream operation is as follows:

cin >> variable 1 >> variable 2 >>....>>variable N;

Following are some valid input statements;

```
cin >> i >> j >> k ;
cin >> name >> age >> address ;
```

The following program illustrates the use of *cin* and *cout* object :

***//* Program 3.6**

```
#include<iostream.h>
#include<conio.h>
void main ( )
  {
int marks1, marks2, marks3 ;
char name [25] ;
char semester [15] ;
clrscr ( ) ;
```

```
cout << "=====================================";
cout << " \n " ;
cout << " Enter Marks : " ;
cin >> marks1 >> marks2 >> marks3 ;
cout << " Enter Name : " ;
cin >> name ;
cout << "Enter Semester : ;
cin >> semester ; << "/n";
cout<< Marks 1 = "<<marks1<<"\n"<< "Marks 2 = "marks2
<<"\n"
        <<"Marks 3 = " <<marks 3 << "\n" ;
cout << "/n =======The End ====================" ;
getch ( ) ;
    }
```

Output :

```
=======================================
Enter Marks : 61 71 59
Enter name : Bikash Bora
Enter Semester : 3rd Semester
Marks 1 = 61
Marks 2 = 71
Marks 3 = 59
===============The End ==================
```

The following figure shows flow of input and output stream :

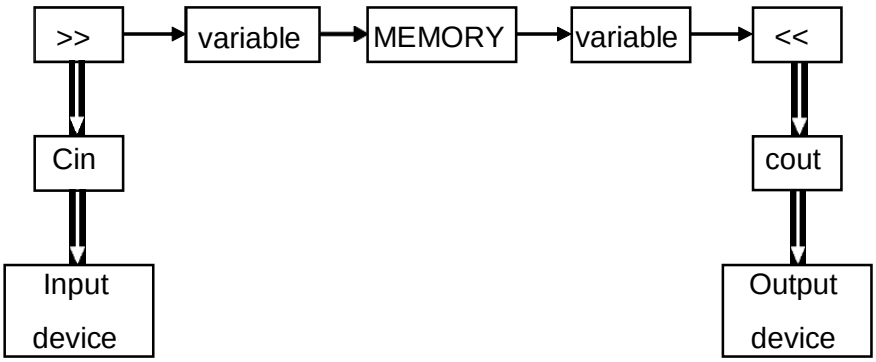


Fig. 3.3 : Working of *cin* and *cout* statement

CHECK YOUR PROCESS

1. State whether the following statements are TRUE or FALSE
   (a) A variable name can consist of letters, digits and underscore (-) but no other special characters.
   (b) In dynamic initialization, we initialize a variable at compile time.
   (c) In C++, an identifier must be initialized using constant expressions.
   (d) $age is a valid variable name.
   (e) Cin is also called extraction operator.
2. Choose the correct answer from the following:
   (i) Which of the following is a reserved word in C++:
   (a) template (b) throw
   (c) this (d) all of the above
   (ii) A variable defined within a block is visible :
   (a) within a block (b) within a function
   (c) both (a) and (b) (d) none of the above
   (iii) The cin and cout functions require the header file to include:
   (a) isotream.h (b) stdio.h
   (c) iomanip.h (d) none of the above
   (iv) The streams is a :
   (a) flow of data (b) flow of integers
   (c) flow of statements (d) none of the above
   (v) Which of the following is C++ standard stream :
   (a) cin (b) cout
   (c) cerr (d) All of the above

3.11 LET US SUM UP

- In C++, tokens are the various elements present in a program. Tokens can be classified as - keyword, variable, constants, special character and operators.
- Identifiers are names of variables, function and arrays. They are user-

defined names, consisting of sequence of letters and digits, with a letter as a first character,

- The C++ keywords are reserved words by the compiler. All C language keywords are valid in C++ and a few additional keywords are added.
- Variables are used to store value i.e. information. A variable is a sequence of memory locations, which are used to store assigned values.
- C++ permits declaration of variables anywhere in the program.
- The constants in C++ are applicable to those values, which do not change during the execution of a program. The two types of constants are *literal and symbolic*.
- The initialization of variable at run-time is called as *dynamic initialization*.
- In C++, a reference variable acts as an alternative (alias) name for a perviously defined variable.
- C++ supports all data type in C.
- A stream is a series of bytes that acts as a source and destination for data. The source stream is called input stream and the destination stream is called output stream.
- The *cin, cout*, *cerr* and *clog* are predefined streams.
- The header file *iostream.h* must be include when we use *cin* and *cout* functions.

3.12 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education

3.13 ANSWER TO CHECK YOUR PROGRESS

1. (a) T, (b) F, (c) F, (d) F, (e) T.
2. (i) d, (ii) a, (iii) a, (iv) a, (v) d.

3.14 MODEL QUESTIONS

1. What are identifiers, variables and constants?
2. What is the difference between a keyword and an identifier?
3. List the rules of naming an identifier in C++?
4. Which are the two types of constants? Describe them with suitable examples ?
5. What is dynamic initialization? Is it possible in C?
6. What is the difference between reference variable and normal variable?
7. Write short note on the following:
   (a) Dynamic initialization of variable
   (b) Reference variable
   (c) Input stream
   (d) Output stream
   (e) Constants
   (f) Variables

UNIT 4: OPERATORS AND MANIPULATORS

UNIT STRUCTURE

4.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- describe about operators and manipulators in C++
- describe different types of operators like arithmetic, relational, logical, bitwise etc.
- describe to use scope resolution, insertion, extraction operators
- describe precedence and associativity of operators

4.2 Introduction

In the earlier units, we become familiar with the elements of C++ language. These variables, constants and other elements can be joined together by various operators to form expressions. All C language operators are valid in C++. In addition, C++ introduces some new operators.

This unit describes the opereators used in C++ language. It also introduces the concept of manipulators which are required to format the display of data.

4.3 Operators

An ***operator*** consists of symbols or words which help the user to command the computer to do a certain mathematical or logical manipulations. They are used in programs to manipulate data and variables. Operators are essential to form *expressions*.

4.4 Types of Operators

C++ has a rich set of operators which can be classified based on their utility and action. There are different types of operators like *arithmetic, relational*, *logical*, *bitwise, increment and decrement, assignment , conditional comma etc*. C++ also provides the facility to give several meanings to an operator depending upon the type of arguments used.

4.4.1 Arithmetic Operator

The operators used to perform arithmetic operations like plus, minus, multiplication, division etc. are known as *arithmetic operators*. The basic arithmetic operators in C++ are listed in Table 4.1.

Table 4.1 : Arithmetic Operators

Operator	Action
–	Subtraction
+	Addition
*	Multiplication
/	Division
%	Modulus

The operators +, -, * and / all work the same way as they do in other programming languages like C. We can apply them to almost any built-in data type. When we apply / (division) to an integer or character, any remainder will be truncated.

For example, 7/2 will equal 3 in integer *division*. But the *modulus* operator (%) produces the remainder of an integer division, i.e., 7%2 will give 1.

/***Program 4.1***: Program showing summation, subtraction, multiplication, division, modulo division, increment, decrement of two integer numbers*/

Operand:
The data items that operators act upon are called operands.

```
#include<iostream.h>
#include<conio.h>
int main( )
      {
      int a, b, sum, sub, mul, div, mod;
      clrscr( );
      cout<<"Enter two integer numbers:";
      cin>>a>>b;   //inputs the operands
      sum = a+b;
      cout<<"\n The sum is ="<<sum;
      sub = a-b;
      cout<<"\n The difference is= "<<sub;
      mul = a*b;
      cout<<"\n The product is = "<<mul;
      div = a/b;   // will give the quotient
      cout<<"\n The division is = "<<div;
      mod = a%b;   //will give the remainder
      cout<<"\n The modulus is = "<<mod;
      getch( );
      return 0;
     }
```

If we enter 7 and 2 as input number then the output of the above program will be like this:

Enter two integer number: 7 2
The sum is = 9
The difference is = 5

The product is = 14
The division is = 3
The modulus is = 1

4.4.2 Relational and Logical Operators

Often it is required to compare the relationship between operands and bring out a decision and program accordingly. The operands can be literals(constants), variables or expressions. For example, we may compare the age of two persons, marks of students, salary of persons, or the price of two items, and so on. These comparisons can be done with the help of ***relational operators*.**

With the help of *logical operators* these relationships can be connected. The relational and logical operators often work together and they are discussed together here. The relational and logical operators supported by C++ are listed in Table 4.2. The relational and logical operator introduce the idea of ***true*** and ***false*** value. We have learned earlier that true is any value other than zero and false is zero.

Table 4.2 : Relational and Logical Operators

Relational Operators	**Action**
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
= =	Equal
!=	Not equal
Logical Operators	**Action**
&&	AND
||	OR
!	NOT

Expressions that use relational or logical operators return 0 for false and 1 for true. C++ supports this zero/nonzero concept. However, it also defines the ***bool*** data type and the Boolean constants *true* and *false*. In C++, a 0 value is automatically converted into *false,* and a non-zero value is automatically converted into *true*.

A simple relational expression contains only one relational operator and takes the following form:

operand1 ***relational operator*** operand2

where *operand1* and *operand2* are expressions, which may be simple constants, variables or combination of them i.e., expression. Some examples of relational expressions and their evaluated values are listed below:

3.5 <= 12	TRUE
–6 > 0	FALSE
10 < 7 + 5	TRUE

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. The truth table for logical operators cosidering 1 for true and 0 for false is shown in Table 4.3.

AND (&&)

The logical AND operator is used to evaluate two conditions or expressions with relational operators simultaneously. If both the expressions to the left and to the right of the logical operator are true then the whole compound expression is true. For example :

```
a>b && x = = 8
```

The whole expression is true only if both expressions a>b and x==8 are true i.e., if *a* is greater than *b* and *x* is equal to 8.

OR (||)

The logical OR operator is used to combine two expressions or the condition evaluates to true if any one of the two expressions is true. For example:

a < m || a < n

The expression evaluates to true if any one of them is true or if both of them are true. It evaluates to true if *a* is less than either *m* or *n* and when *a* is less than both *m* and *n*.

NOT (!)

The logical NOT operator takes single expression and evaluates to true if the expression is false and evaluates to false if the expression is true. In other words it just reverses the value of the expression.

Table 4.3 : Truth table for Logical AND, OR, NOT

operand1	operand2	NOT	NOT	AND	OR
a	**b**	**!a**	**!b**	**a&&b**	**a||b**
T	T	F	T	T	T
T	F	F	T	F	T
F	T	T	F	F	T
F	F	T	T	F	F

(T indicates *True* and F indicates *False*)

CHECK YOUR PROGRESS

1. Choose the appropriate option for the correct answer:

   (i) 15%6 yields a result of

   (a) 3 (b) 2

   (c) 0 (d) none of these

   (ii) !(-6>0) results

   (a) False (b) True

   (c) Both a) and b) (d) none of these

2. Give the output of the following code:

```
void main()
  {     int i=2;
        cout<<“Output :”<<endl<<i+5<<endl<<i-3<<endl<<i;
  }
```

3. What is the final value of x if initially x has the value 1, after execution of the following codes:

   (a)
```
if(x>=0)
    x+=10;
else if(x>=10)
    x+=2;
```

   (b)
```
if(x>=0)
    x+=10;
if(x>=10)
    x+=2;
```

4.4.3 Assignment Operator

C and C++ use a single equal sign (=) to indicate assignment operation. The general form of assignment operator is:

```
variable_name = expression;
```

where an *expression* can be a single variable or literal, or a combination of variables, literals, and operators. The left part of the assignment must be a variable or a pointer. It cannot be a function or a constant. For example, in the following assignment statement:

```
sum = a + b ;
```

the value of *a* + *b* is evaluated first and then substituted to the variable *sum*. Like C, shortcut of assignment operator is possible in C++ also. For example, in place of the following statement:

```
a = a+1;
```

we may use the shorthand form : a += 1;

It is also possible to assign many variables with the same value by using multiple assignments in a single statement. For example, the following statement assigns a,b and c the value 0:

```
a = b = c = 0;
```

4.4.4 Increment and Decrement Operator

The ***increment operator*** (++) adds 1 to its operand and the ***decrement operator*** (- -) substracts 1. Increment and decrement operators require single variable as their operand. These operators are used in a program as follows:

```
++ i;    or   i ++;
- - i;    or   i - -;
```

where ***i*** is an integer type variable. When an increment or decrement operator precedes its operand, then they are in ***prefix*** form. The increment or decrement operation in prefix notation is performed before obtaining the value of the operand for use in the expression. If the operator follows its operand i.e., in ***postfix*** form, the value of the operand is obtained before incrementing and decrementing it. For example, consider the following statements:

```
+ + a ;
a + +;
```

In the above statements, it does not matter whether the increment operator is pre incremented or post incremented; it will produce the same result. However, in the following statements, it does make a difference:

```
int x = 0, y = 5;
x = ++y;
```

Here, the value of x after the execution of this statement will be 6, since y is incremented first and then assigned. If we write

```
x = y++;
```

instead of the previous, then the value of x will be 5, since it is assigned first and then incremented.

4.4.5 Bitwise Operator

Bitwise operators are used for manipulation of data at bit level. These operators are used for testing, complementing, setting, or shifting bits to the right or left in a *byte* or *word*. Bitwise operators can be used in **char** and **int** type data. We cannot use bitwise operations on **float**, **double**, **long double**, **void**, or **bool** types of data. The bitwise AND, OR, NOT(one's complement) possesses the same truth table as their logical equivalents, except that they work bit by bit. Bitwise operators used in C++ are listed in table 4.4.

Table 4.4 : Bitwise Operators

Operator	Action
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR
~	Bitwise NOT
<<	Bitwise left shift
>>	Bitwise right shift

Relational and logical operators always produce a result that is either true(1) or false(0), whereas the similar bitwise operations may produce any arbitrary value in accordance with the specific opera-

tion. The truth table of Bitwise AND (&), OR(|) and NOT(~) operators are same with logical AND(&&), OR(||) and NOT(!) respectively(table 4.3).

Following is an example of ***bitwise AND*** operator :

```
unsigned int a = 60;        // 60 = 0011 1100
unsigned int b = 13;        // 13 = 0000 1101
unsigned int c = 0;         //  &
c = a & b;                  //12 = 0000 1100
```

The ***bitwise OR*** operator places a 1 in the resulting value's bit position if either operand has a bit *set* (i.e.,1) at the position. Operation of bitwise OR(|) is shown with the following statements:

```
unsigned int a = 60;        // 60 = 0011 1100
unsigned int b = 13;        // 13 = 0000 1101
unsigned int c = 0;         //          |
c = a | b;                  // 61 = 0011 1101
```

The ***bitwise exclusive OR*** *(XOR)* operator sets a bit in the resulting value's bit position if either operand (but not both) has a bit *set* (i.e.,1) at the position. Bitwise exclusive OR operation can be understood with the following statement:

```
unsigned int a = 60;        // 60 = 0011 1100
unsigned int b = 13;        // 13 = 0000 1101
unsigned int c = 0;         //          ^
c = a ^ b;                  // 49 = 0011 0001
```

The ***bitwise NOT*** (~) operator reverses each bit in the operand. That is, all 1s are set to 0, and all 0s are set to 1.

The ***bit-shift*** operators, >> and <<, move all bits in a value to the right or left as specified. As bits are shifted off one end, 0's are brought in the other end. The bits shifted off one end do not come back around to the other. The bits shifted off are lost. In the case of a signed, negative integer, a right shift will cause a 1 to be brought in so that the sign bit is preserved.

The ***left shift*** operator shifts bits to the left. Let us consider the following declaration:

```
unsigned int a = 5;     // 5 in binary              = 00000000 00000101
a = a<<1;               // after left shift by 1    = 00000000 00001010
                                                    = 10 in decimal
a=a<<15;                // after left shift by 15   =10000000 00000000
                                                    = 32768 in decimal
```

Shifting once to the left multiplies the number by 2. Multiple shifts of 1 to the left, results in multiplying the number by 2 over and over again. In other words, multiplying by a power of 2.

For example, let us evaluate 4 << 2. In binary 16-bit format, 4 is 00000000 00000100. To evaluate 4<<2, we can add 2 zeros to the end of its binary equivalent. It gives 00000000 000010000, which is 16, i.e., $4*2^2 = 4*4 = 16$. Similarly, 4 << 3 can be evaluated by adding 3 zeros to get 00000000 00100000, which is $4*2^3 = 4*8 = 32$. Some other examples are:

$5 << 3 = 5*2^3 = 5*8 = 40$

$8 << 4 = 8*2^4 = 8*16 = 128$

$1 << 2 = 1*2^2 = 1*4 = 4$

The ***right shift*** operator shifts bits to the right. As bits are shifted toward low-order position, 0 bits enter the high-order positions, if the data is unsigned. If the data is signed and the sign bit is 0, then 0 bits also enter the high-order positions. For example, let us consider the statement

unsigned int x = 40960;

and x in bibary 16-bit format is 10100000 00000000. Now if we apply right shift, then

x >> 1 is 01010000 00000000 or 20480 decimal

and x >> 15 is 00000000 00000001 or 1 decimal

/* ***Program 4.2*****:** Program for the demonstrating of left shift and right shift */

```
#include<iostream.h>
#include<conio.h>
int main()
{
    unsigned int a,b,c;
    a=5;
    b=8;
    c=40960;
    clrscr();
    cout<<"\na ="<<a;
    a=a<<1;                    //left shift by 1
    cout<<"\na after left shift by 1:"<<a;  //
    output will be 10
```

```
      cout<<"\n\nb ="<<b;
      b=b>>1;    //right shift by  1
      cout<<"\nb after right shift by 1:"<<b;
//output will be 4
      cout<<"\n\nc ="<<c;
      c=c>>15;   //right shift by  15
      cout<<"\nc after right shift by 15:"<<c;
//output will be 1
      getch();
      return 0;
   }
```

4.4.6 Conditional Operator

The conditional operator consists of two symbols, the question mark (?) and the colon (:) . The syntax is

exp1 **?** exp2 **:** exp3 ;

where *exp1*, *exp2*, and *exp3* are expressions. *exp1* is evaluated first. If the expression is true then *exp2* is evaluated and its value becomes the result of the expression. If *exp1* is false, *exp3* is evaluated and its value becomes the result of the expression. For example, let us consider the program segment :

```
a = 10;
b = 15;
x = (a > b) ? a : b;
```

Here *x* is assigned the value 15. As the condition (a>b) is false for a= 10 and b=15, therefore the value of *b* will be assigned to *x*.

//***Program 4.3 :*** Program showing the use of conditional operator?

```
#include<iostream.h>
#include<conio.h>
int main( )
{
     int age;
     clrscr();
     cout<<"Enter your age in years: ";
```

```
cin>>age;
(age>=18)?(cout<<"\nYou can vote\n"):(cout<<"You
cannot vote");
 getch();
 return 0;
}
```

If we enter *age* as 26 then the output will be:

Enter your age in years: 26
You can vote

Again, if we run the program by entering *age* as 15, then the output will be:

Enter your age in years:15
You cannot vote

4.4.7 Comma Operator

The comma operator can be used to link related expressions together. The comma allows for the use of multiple expressions to be used where normally only one would be allowed. The comma operator forces all operations that appear to the left to be fully completed before proceeding to the right of comma. Let us consider the following declaration:

```
num1 = num2 + 1, num2 = 2;
```

The comma ensures that num2 will not be changed to 2 before num2 has been added to 1 and the result placed into num1. We will observe the use of comma operator while we discuss loop in later units. For example, in exchanging(swap) values we can use comma operator like this :

```
temp = x, x = y, y = temp;
```

4.4.8 sizeof Operator

The ***sizeof*** operator returns the number of bytes required to represent a data type or variable. It can be used with built-in as well as user-defined data types. The general form of writing sizeof operator is:

```
sizeof(data type);
sizeof(variable);
```

For example, the following statements:

```
double a;
char c;
cout<<sizeof(c);    //returns 1
cout<<sizeof(int); //returns 2
cout<<sizeof(a);    //returns 8
```

will give the result 1 2 8 as the size of character, integer and double are 1, 2 and 8 bytes respectively.

EXERCISE-1

Q. What will be the output of the following code :

```
void main()
{
    char *p;
    cout<<sizeof(p);
}
```

4.4.9 Scope Resolution Operator(::)

We can use nested blocks in C++. For example, we can write nested blocks as follows:

```
....
{
    int a = 5;        // a is global for inner block
    ....
    ....
    {
        int a = 2;
        ....
        ....
    }
    ....
    ....
}
```

inner block **outer block**

Declaration of variable in an inner block hides a declaration of the same variable in an outer block. A variable declared inside a block is said to be local to that block. In C, a global version of a variable can not be accessed from within the inner block. But in C++, we can resolve this problem using a new operator **::** called the ***scope resolution operator***. It can be written as:

:: variable_name;

// ***Program 4.4 :*** Use of scope resolution operator

```
#include<iostream.h>
#include<conio.h>
int a=20; // global a
int main()
{
    int a=5;    // a is local to main()
    clrscr();
    {
        int a=15;     // a is local to inner block
        cout<<"In inner block a is = "<<a; / /
here, a will be 15
    }
    cout<<"\nIn outer block a is = "<<a;  / /
here, a will be 5
    cout<<"\nOutside main function a is = "<<::a;
//here, a will be 20
    getch();
    return 0;
}
```

The output of the above program will be like this:

In inner block a is = 15
In outer block a is = 5
Outside main function a is = 20

We have used **::a** to display the value of global variable. If we use simply **a** then it will give 5 instead of 20.

One major application of the scope resolution operator is to identify the class to which a member function belongs. The role of scope resolution operator will be discussed in more detail in later units when classes and objects are introduced.

4.4.10 Insertion and Extraction Operator

We have already used the insertion and extraction operator in many input/output statements of our program. The ***insertion operator*** *(<<)* is used with ***cout*** object to carry out output operations. Similarly, the ***extraction operator*** (>>) is used with ***cin*** object to carry out input operations. These can be written as:

```
cout<<variable;
cin>>variable;
```

For example, cout<<v_1<<v_2<<$v_{3.....}$ <<v_n;

cin>>v_1>>v_2>>$v_{3.....}$ >>v_n;

where v_1 ,v_2, $v_{3, ...}$ v_n are variables.

4.4.11 Address and Indirection Operator

All the variables defined in a program reside at specific addresses in memory. It is possible to obtain the address of a variable used in a program by using the ***address operator*** (&). When used as a prefix to a variable name, **&** operator returns the address of that variable.

The ***indirection operator*** (*****) return the value of the variable located at the address that follows it. For example, let us consdier the following program.

```
// Program 4.5 : Demonstration of * and & operator
#include<iostream.h>
#include<conio.h>
int main()
{
  int a, *p ;
  a=50;
  p= &a;
  clrscr();
  cout<<"Value of a :"<<a<<endl;
  cout<<"Value of a: "<<*p<<endl;
  cout<<"Value of a: "<<*(&a)<<endl;
  cout<<"Address of a: "<<&a<<endl;
  cout<<"Address of a: "<<p;
```

```
    getch();
    return 0;
}
```

Output will be like this:

4.4.12 Memory Management Operator

In C language, we have studied the function ***malloc()***, ***calloc()*** and ***realloc()*** which are used to allocate memory dynamically at run time. Similarly, ***free()*** function is used to release memory which is allocated by these functions. Although C++ supports these functions, it also defines two operators for allocation and deallocation of memory in an easier way. These two operators are ***new*** and ***delete***.

new operator

The *new* operator allocates memory of specified type and returns back the starting address to the pointer variable. The general form of *new* operator is:

pointer_variable = ***new*** data_type[size];

Here, the *pointer_variable* is a pointer to the *data_type.* The *size* is optional. We can specify the size when we want to allocate memory space for user defined data types such as arrays, structure and classes. If the *new* operator fails to allocate memory, it returns NULL. For example, let us consider the following declaration:

```
int *p ;
p = new int ;
char *q = new char ;
```

where *p* is the pointer of type *int* and *q* is a pointer of type *char.*

For allocation of memory for user defined data type such as array, we can use the foolowing form:

pointer_variable = ***new*** data_type[size] ;

The statement int *p = new int[10]; creates memory space for an array of 10 integers (i.e.,20 bytes). p[0] will refer to the first element, p[1] to the second element and so on.

We can also initialize the memory using the *new* operator like this:

pointer_variable = ***new*** data_type(value);

For example, int *ptr = new int(5); where 5 is assigned to pointer variable *ptr*.

delete operator

The delete operator releases the memory allocated by the new operator. Following are the syntax of delete operator:

```
delete pointer_variable;
delete [size] pointer_variable;
```

For example, delete p;
delete [10] p;

CHECK YOUR PROGRESS

4. Find the output of the following program segment:

(a)
```
int main()
{
    int x = 50;
    cout<<x++<<endl;
    cout<<++x;
    getch();
    return 0;
}
```

(b)
```
void main()
{
    float a[4],s;
```

```
            clrscr();
            s=sizeof(a);
            cout<<"\nSize is "<<s<<" bytes";
            getch();
        }
(c)     int p=100;
        void main(){
            int p=5;
            clrscr();
            cout<<p<<endl;
            {
                p=20;
                cout<<p<<endl;
                cout<<::p+5;
            }
            getch();
        }
```

5. Choose the appropriate option for the correct answer:

(i) The scope resolution operator is denoted by the symbol

(a) **::** (b) **:** (c) **->** (d) **!**

(ii) The *new* operator

(a) releases memory

(b) allocates memory statically

(c) allocates memory dynamically

(d) none of these

(iii) float *ptr = new float[15]; allocates memory of

(a) 30 bytes (b) 40 bytes

(c) 10 bytes (d) 60 bytes

(iv) Bitwise operators can be used only with

(a) int and *float* datatype

(b) *char* and *int* datatype

(c) *float*, *double*,*long double* datatype

(d) none of these

4.5 Precedence and Associativity

There are two important characteristics of operators which determine how operands group with operators. These are *precedence* and *associativity*.

Every operator in C++ language has a precedence associated with it. ***Precedence*** rules help in removing the ambiguity about the order of performing operations while evaluating an expression. An operator's precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first. The grouping of operands can be forced by using parentheses.

With each operator, there is an ***associativity*** factor that tells in what direction the operands associated with the operator are to be evaluated. It is the *left-to-right* or *right-to-left* order for grouping operands to operators that have the same precedence. For example, in the following statements,

```
b = 9;
c = 18;
a = b = c;
```

the value 18 is assigned to both ***a*** and ***b*** because of the right-to-left associativity of the assignment = operator. The value of ***c*** is assigned to ***b*** first, and then the value of ***b*** is assigned to ***a***. Similarly, in the expression

```
x =  a + b * c / d;
```

As precedence of * and / is more than binary +, therefore multiplication and division are performed before +. Again, ***b*** is multiplied by ***c*** before it is divided by ***d*** because of their left-to-right associativity. A list indicating the precedence and associativity of operators is shown in table 4.5. C++ operators are classified into 16 categories based on their precedence. The operators within each category have equal precedence. In the table, operators are arranged from highest(in top) to lowest(in bottom) precedence.

Table 4.5 : Operators from highest to lowest precedence

Precedence	Operator	Description	Associativity
1	()	Function call	L-to-R
	[ ]	Array subscript	L-to-R
	–>	C++ indirect component selector	L-to-R
	::	Scope Resolution	L-to-R
	.	C++ direct component selector	L-to-R

Prece-ence	Operator	Description	Associ-ativity
	!	Logical NOT	R-to-L
	~	Bitwise NOT(1's complement)	R-to-L
	+	Unary Plus	R-to-L
	–	Unary Minus	R-to-L
	++	Increment	R-to-L
2	--	Decrement	R-to-L
	&	Address	R-to-L
	*	Indirection	R-to-L
	sizeof	Returns size of operand in bytes	R-to-L
	new	Dynamically allocates memory	R-to-L
	delete	Reseases memory	R-to-L
3	.* –>*	C++ dereference	L-to-R
	*	Multiply	L-to-R
4	/	Divide	L-to-R
	%	Modulus	L-to-R
5	+	Binary plus	L-to-R
	–	Binary minus	L-to-R
6	<< >>	Left and Right shift	L-to-R
7	< <= > >=	Relational	L-to-R
8	== !=	Equality	L-to-R
9	&	Bitwise AND	L-to-R
10	^	Bitwise XOR	L-to-R
11	|	Bitwise OR	L-to-R
12	&&	Logical AND	L-to-R
13	||	Logical OR	L-to-R
14	?:	Conditional	R-to-L
	= *= /= %=	Assignment	R-to-L
15	+= –= >>=		
	<<= &= ^= |=		
16	,	Comma	R-to-L

4.6 MANIPULATORS

Manipulators are used to format the display of data. They are specifically designed to be used in conjunction with the insertion (<<) and extraction (>>) operators on stream objects.

We often use two important manipulators which are ***endl*** and ***setw***.

For example, suppose we are to display the following text in three different lines: KKHSOU

Guwahati

Assam

We have already learnt how to use the newline character "\n" for displaying a value or text in different lines. The effect of ***endl*** is also same. The manipulator ***endl*** when used in an output statement, causes a linefeed to be inserted. It can be used as follows:

```
cout<<"KKHSOU"<<endl;
cout<<"Guwahati"<<endl;
cout<<"Assam";
```

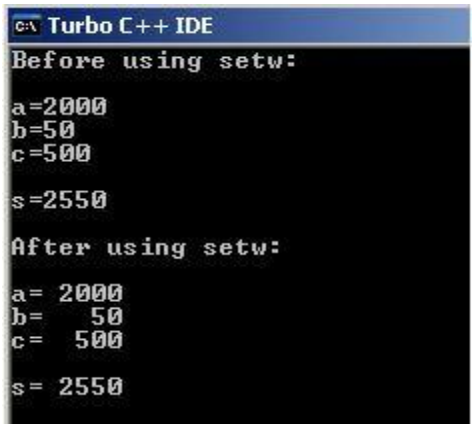
The manipulator ***setw(n)*** sets the field width to ***n*** for displaying the value of a variable. The use of these two manipulators are demonstrated in the following program where sum of three values are displayed in two different format.

// ***Program 4.6 :*** Program for the demonstration of *setw* and *endl*

```
#include<iostream.h>
#include<iomanip.h> //for manipulator set
#include<conio.h>
int main()
{
    int a,b,c,s;
    clrscr();
    a = 2000, b = 50, c = 500;
    cout<<"Before using setw:"<<endl<<endl;
    cout<<"a="<<a<<endl;
    cout<<"b="<<b<<endl;
    cout<<"c="<<c<<endl;
    s=a+b+c;
    cout<<endl<<"s="<<s<<endl;
    cout<<endl<<"After using setw:"<<endl<<endl;
    cout<<"a="<<setw(5)<<a<<endl;
    cout<<"b="<<setw(5)<<b<<endl;
    cout<<"c="<<setw(5)<<c<<endl;
    cout<<endl<<"s="<<setw(5)<<s;
```

```
        getch();
    return 0;
}
```

In the output screen content will be like this:



Here we see that after using *setw(5)* the display is in ideal format.

The most commonly used manipulators are listed below :

Table 4.6 : List of Manipulators

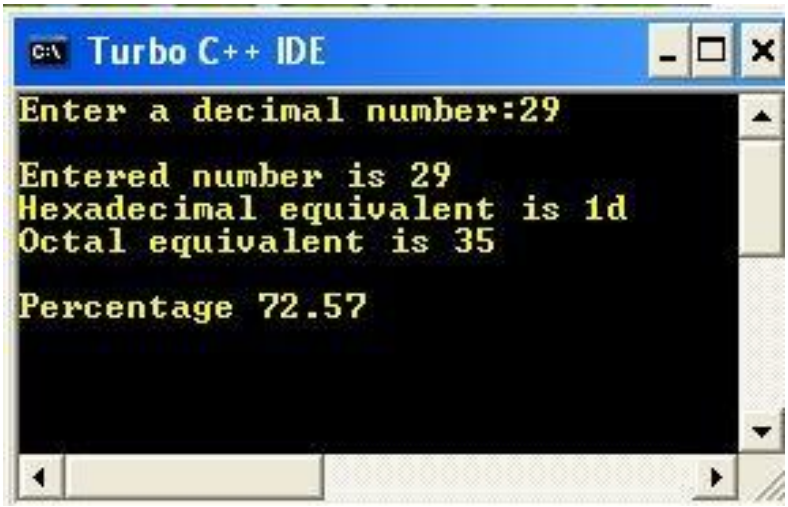
Manipulators	Function
setw(int n)	Sets the field width to *n.*
setprecision(int d)	Sets the floating point precision to *d.*
setfill(char f)	The fill character is set to character. Stored in variable *f.*
setiosflags(long f)	Sets the format flag *f* .
resetiosflags(long f)	Clears the flag indicated by *f*.
endl	Inserts new line.
skipws	Omits white space on input.
noskipws	Does not omit white space on input.
flush	Flushes the buffer stream.
ws	Used to omit the leading white. spaces present before the first field.
dec, oct, hex	Displays the number system in decimal, octal and hexadecimal format.

The manipulator dec, oct, hex, ws, endl, flush are defined in header file ***iostream.h***. The manipulator like *setw()*, *setfill()*, etc. which require an argument are defined in ***iomanip.h***.

// ***Program 4.7*** **:** Program showing some use of manipulators

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
int main()
{
     int n;
     float per=72.56888;
     clrscr();
     cout<<"Enter a decimal number:";
     cin>>n;
     cout<<endl<<"Entered number is "<<n;
     cout<<endl<<"Hexadecimal equivalent is "<<hex<<n;
     cout<<endl<<"Octal equivalent is "<<oct<<n<<endl;
          cout<<endl<<"Percentage
"<<setprecision(2)<<per;
     getch();
     return 0;
     }
```

If we enter 29, the output screen will be as follows :



CHECK YOUR PROGRESS

6. Name the header files that shall be needed for the following code:

```
void main()
{
    int number= 105;
    clrscr();
    cout<<setw()<<number;
    getch();
}
```

(b) For *endl*, the header file is_____.

7. State which of the following statements are true(T) or False(F):
   (i) *endl* is an operator.
   (ii) setw is an operator with a parameter.
   (iii) Presedence of *comma* operator is less than *assignment* operator.
   (iv) Associativity factor gives in what direction the operands associated with the operator are to be evaluated.
   (v) 7*2/3 yields 4 because of their left-to-right associativity.
   (vi) 7*2/3 yields 2 because of their right to-left associativity.

4.7 LET US SUM UP

The key points to keep in mind in this unit are:

- ***Operators*** are special characters or symbols or some specific words in C++ which instruct the compiler to perform operation on some operands. Operands can be a variables, expressions etc.
- Some operators operate on a single operand and they are called ***unary*** operators. Most operators act between two operands and they are called ***binary*** operators.
- C++ supports all operators of C. There are some other operators introduced by C++ which are listed in table 4.5.

- The ***assignment operator***(=) evaluates an expression on the right of the expression and substitutes it to the value or variable on the left of the expression.
- The five ***arithmetic operators*** supported by C++ are +(addition), -(subtraction), *(multiplication), /(division),%(modulus) .
- ***Relational operators*** are used to make comparisons between expressions. All relational operators are binary and require two operands. These operators are used to compare logically related data for taking decisions.
- ***Logical operators*** are useful in combining one or more conditions.
- ***Bitwise operators*** are used to manipulate integer and character operand at bit level.
- The ***increment*** and ***decrement*** operator increases or decreases the value of a variable on which they operate by one.
- The ***conditional operator*** is an alternate method of using a simple *if-else* construct.
- The ***sizeof*** operator accepts one parameter that can be either a data type of a variable or a variable itself and returns the size in bytes of that type or object.
- ***Comma*** operator links the related expressions together. We can use the comma operator to build a compound expression by putting several expressions inside a set of parentheses.
- To allocate memory dynamically, C++ uses the ***new*** operator. The ***delete*** operator is used to release the memory which is dynamically allocated. The *new* and *delete* operators are easy in writing as compared to *malloc()*, *calloc()*, *realloc()*, *free()* functions whch are also supported by C.
- The ***scope resolution operator*** :: plays significant role in C++. It is useful for accessing class members. It allows a programmer to access a global name even if it is hidden by a local redeclarion of that name.
- ***Precedence*** of operators help in removing the ambiguity about the order of performing operations while evaluating an expression.
- ***Associativity*** of operators specifies the direction in which the operators are evaluated in an expression.
- C++ provides some ***manipulators*** which are used to format the display of data.

4.8 FURTHER READING

- Venugopal, K.P. (2013). *Mastering C++.* Tata McGraw-Hill Education
- Balagurusami, E. (2001). *Object Oriented Programming with C++,* 6e. Tata McGraw-Hill Education
- Kamthane, Ashok. *Object-Oriented Programming with ANSI and Turbo C++*. Pearson Education India.

4.9 ANSWERS TO CHECK YOUR PROGRESS

1. (i) (a) 3 (ii) (b) True
2. Output : 7
   -1
   2
3. (a) 11 (b) 13
4. (a) 50 (b) Size is 16 bytes (c) 5
   52 20
   105
5. (i) (a) **::**
   (ii) (c) allocates memory dynamically
   (iii) (d) 60 bytes
   (iv) (b) char and int datatype
6. (a) The header files needed for the code are: iostream.h
   conio.h
   iomanip.h
   (b) iostream.h
7. (i) False (ii) False (iii) True
   (iv) True (v) True (vi) False

4.10 MODEL QUESTIONS

1. What are logical and relational operators? Give the truth table for the logical operators.
2. What is bitwise operator? Explain with example.
3. Write a C++ program to allocate memory using new operator for 15 integers. Input and dispaly the integers.
4. Write a C++ program to evaluate the following expression and display their result (initialize a and b with some value)
   (i) x = a*(++b);
   (ii) y = a*(b++); where a,b,x,y are intergers.
5. Write a C++ program to find the largest of two numbers.
6. Write a C++ program to display 1 if the input taken from the keyboard is character, otherwise display 0.
7. Write short notes on:
   (i) Precedence of operators in C++
   (ii) Division and Modulus operator
8. Write a C++ program to display the size of integer, float, character, float, long and double.
9. What is the use of manipulators in C++? Give the function of any 3 manipulators.
10. What is scope resolution operator? Explain with example.

****

UNIT 5: DECISION AND CONTROL STRUCTURES

UNIT STRUCTURE

5.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- learn about conditional statements
- learn how to deal with multiple choice situations
- use loops to perform some repetition of task in a program
- alter the sequence of program execution with the help of *break* , *continue* and *goto* statement.

5.2 INTRODUCTION

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

In this unit, we will look at how to add decision-making capabilities to our programs. We will also learn how to make our programs repeat a set of actions until a specific condition is met.

5.3 DECISION MAKING STATEMENTS

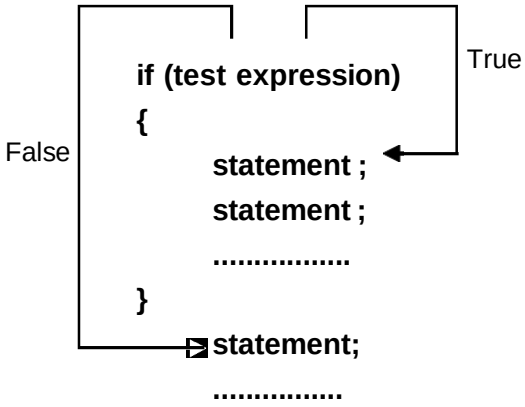
Decision-making is an important concept in any programming language and to accomplish this, C++ uses the following decision making statements:

- ***if*** statement
- ***if-else*** statement
- ***switch*** statement

5.3.1 if statements

***Block** : In C++ and some other languages like C, Java, blocks are collection of one or more than one statements enclosed by curly braces { }. Block statements are also referred to as compound statement.*

The ***if*** statement is a powerful decision making statement and is used to control the flow of execution of statements. The syntax of *if* state-ment is as follows:



Whenever an *if* statement is encountered in a program, it evaluates the expression first and if the expression returns *true (non-zero)* value then all the statements inside the braces of *if* block are executed.

Otherwise, if the expression returns *false (zero)*, then the statement outside the *if* block is executed. If there is only one statement to be executed when the expression returns *true* in an *if* statement, then it can be written without the curly braces. The flow chart of *if* statement is given in figure 5.1.

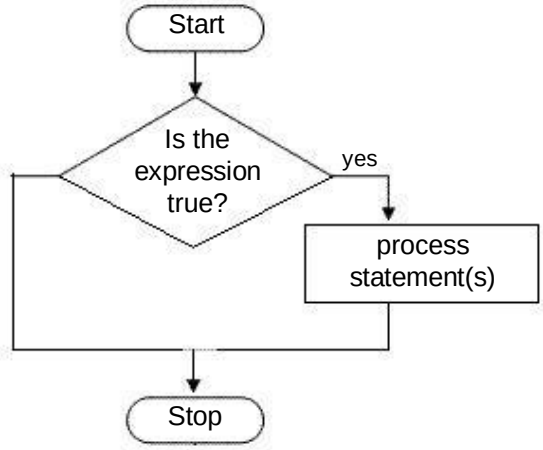


Fig. 5.1 : Flow chart of *if* statement

For example, let us consider a program where the programmer desires to display “Pass” if the marks obtained by the student is greater than or equal to 150. Thus, this can be performed as follows:

```
/*Program 5.1: Demonstration of if statement */
    #include<iostream.h>
    #include<conio.h>
    int main()
    {
        float marks;
        clrscr();
        cout<<"\nEnter the marks obtained by the
student:";
        cin>>marks;
        if(marks>=150)
           cout<<"Pass";
        getch();
        return 0;
        }
```

There is only one statement to execute if the statement **if(marks>=150)** returns *true* value, therefore, the braces are not used. They are optional in such cases.

Suppose, the programmer wants to display the percentage of the student if he/she is passed, then we can modify the above program with the following lines of code. Here braces are used for the *if* block as follows:

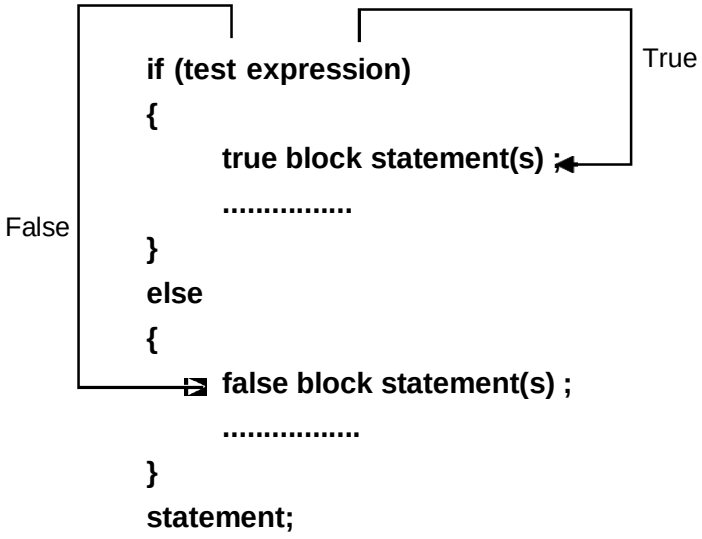
/***Program 5.2*:** Demonstration of *if* statement (more than one statements inside if statement)*/

```
#include<iostream.h>
#include<conio.h>
int main()
{
    float totalmarks=500,marks,per;
    clrscr();
    cout<<"\nEnter the marks obtained by the stu-
dent:";
    cin>>marks;
    if(marks>=150)
    {
         cout<<"Pass";
         per=(marks/totalmarks)*100;
         cout<<"\nPercentage of the student is
"<<per<<"%";
    }              //end of if statement
    getch();
    return 0;
}
```

5.3.2 *if-else* statements

The ***if-else*** statement is an extension of the simple *if* statement. It performs some action even when the test expression fails.

In the syntax, we can see if the test expression is *true*, then the true block statement(s), immediately following the *if* statement are executed; otherwise the false block statement(s) are executed. The syntax of if-else statement is as follows:



The flow chart of *if-else* state-ment is given in figure 5.2 :

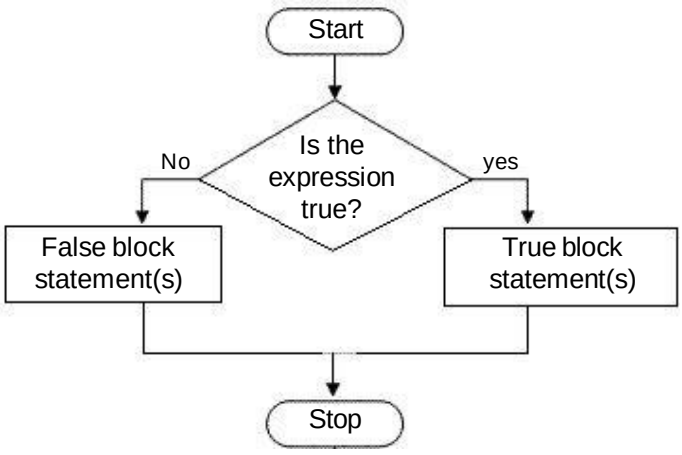


Fig. 5.2 : Flow chart of *if-else* statement

/****Program 5.3:*** Program to display “Pass” or “Fail” along with student's percentage. If marks obtained by the student is greater than or equal to 150 then the result will be Pass otherwise Fail. There are 5 subjects and total marks is 500. */

```
#include<iostream.h>
#include<conio.h>
int main()
{
    float totalmarks=500,marks,per;
    cout<<"\nEnter the marks obtained by the
student:";
```

```
        cin>>marks;
        if(marks>=150)
        {
            cout<<"\nPass";
            per=(marks/totalmarks)*100;
            cout<<"\nPercentage is:"<<per<<"%";
        }
        else
        {
            cout<<"\nFail";
            per=(marks/totalmarks)*100;
            cout<<"\nPercentage is: "<<per<<"%";
        }
        getch();
        return 0;
}
```

Sometimes, our program will need to check many conditions. The syntax of *if-else* statement for such a situation is:

```
if (condition)
{
    statement 1;
    statement 2;
    ..............
}
else if (condition)
{
    statement 1;
    statement 2;
    ..............
}
..............
..............
else
{
    statement 1;
    statement 2;
    ..............
}
```

The compiler will check the first condition. If the condition is true, then all the statements inside the *if* block will be executed and the *else if* will be ignored. If the condition is false, control passes to else block where condition is again checked with the *if* statement. This process continues till there is no *if* statement in the last *else* block. If one of the *if* statements statisfies the condition, other nested *if-else* will not be executed. This is referred to as a ***if-else-if ladder***. The use of nested *if-else* is shown in program 5.4.

/***Program 5.4*:** Program to display division and percentage of student if he/she has passed the examination. There are 5 papers and each paper carrying 100 marks. If marks obtained by the student is greater than or equal to 300 then First division, if it is greater than or equal to 225 then Second division and if it is greater than or equal to 150 then Third division. If marks obtained by the student is less than 150 then the result should be Fail. (Demonstration of *if-else-if ladder*)*/

```
#include<iostream.h>
#include<conio.h>
int main()
{
    float totalmarks=500,marks,per;
    clrscr();
    cout<<"\nEnter the marks obtained by the stu-
dent:";
    cin>>marks;
    if(marks>=300)
    {
            cout<<"\nPass. Student has got First Di-
vision";
        per=(marks/totalmarks)*100;
        cout<<"\nPercentage is: "<<per<<"%";
    }
    else if(marks>=225)
    {
        cout<<"\nPass. Student has got Second Divi-
sion";
        per=(marks/totalmarks)*100;
```

```
            cout<<"\nPercentage is: "<<per<<"%";
        }
        else if(marks>=150)
        {
            cout<<"\nPass. Student has got Third Divi-
sion";
            per=(marks/totalmarks)*100;
            cout<<"\nPercentage is: "<<per<<"%";
        }
        else
            cout<<"\nFail";
            getch();
            return 0;
        }
```

5.3.3 *switch* statements

When there are many conditions, it becomes too difficult and complicated to use the *if* and *if-else* constructs. The ***switch*** statement is suitable when many conditions are being tested for. The general form of a ***switch*** statement is:

```
switch (expression)
{
    case constant1:
            statements1
            break;
    case constant2:
            statements2
            break;
    case constant3:
    case constant4:
    case constant5:
            statements3     // multiple values can share the
            break;          // same statement
    .....
    .....
```

```
        .....
        default:
                statements      // execute if expression != any
of the
    }                           //  above case constant
```

The ***expression*** value must be always integral. The expression to be tested comes after the ***switch*** statement and it can be a variable, an expression or the result of a function. The keyword ***case*** is followed by an *integer* or *character constant*. Every case constant terminates with a colon (**:**). *Switch* statement evaluates *expression* and checks if it is equivalent to *constant1.* If it is, it executes group of *statements1* until it finds the ***break*** statement. When it finds this *break* statement the control jumps to the end of the *switch* block. While studying unconditional statements in this unit you will be able to learn about the *break* statement elaborately.

If *expression* was not equal to *constant1* it will be checked against *constant2*. If it is equal to this, it will execute group of *statements2* until a *break* keyword is found, and then will jump to the end of the *switch* block. Finally, if the value of *expression* did not match any of the previously specified *constants*, the program will execute the *statements* included after the *default* statement, if it exists. If *default* is not present since it is optional, then simply control flows out of the *switch* block.

As we have seen, the statements to be executed for a particular *case* are terminated by a *break* statement. The *break* transfers execution to the statement after the *switch*. If we donot include it, all the statements for the *cases* following the one selected will be executed. For example if *constant1* was met with the value of the *expression*, and there was no *break* statement at the end of the *case*, then all other *cases* like *case constant2*, *case constant3*, etc. and even *default* would all be executed. The flow chart of switch statement is given as in Fig 5.3

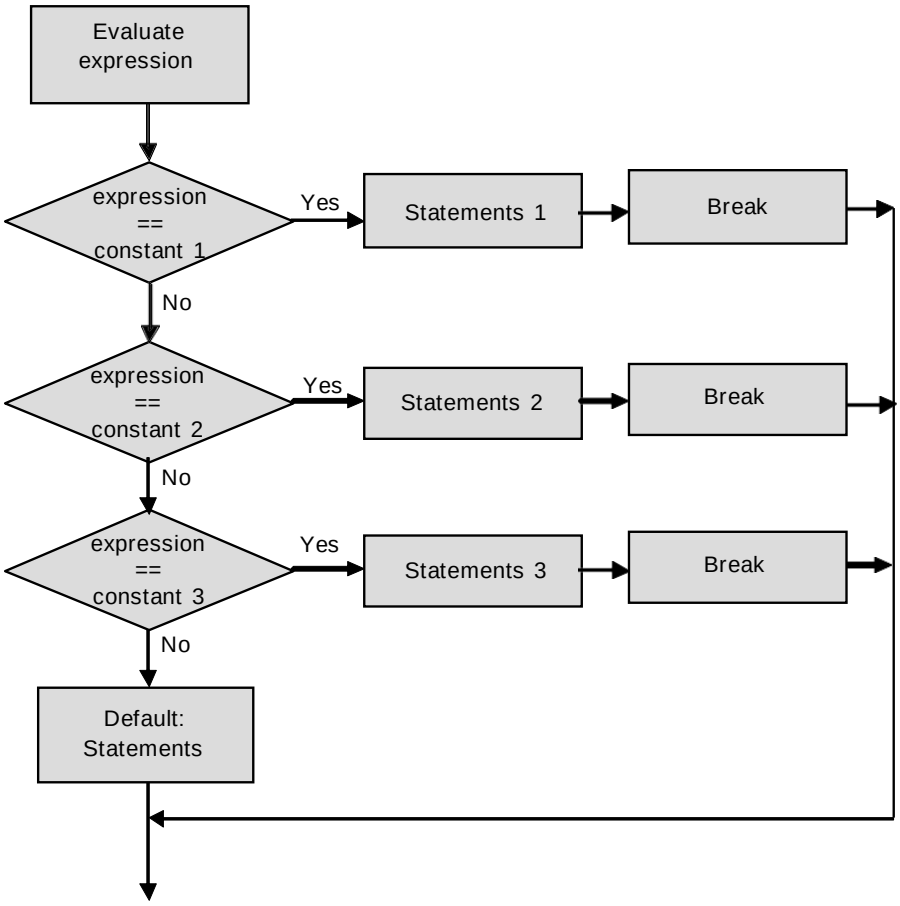


Fig. 5.3 : Flow chart of *switch* statement

It is important to know that no two *case constants* in the same switch can have identical values. Of course, a switch statement enclosed by an outer switch may have case constants that are the same.

Here is an example where the programmer wishes to calculate the area of any one the following:

1. Circle

2.Rectangle

3.Triangle.

If 1 is pressed in the keyword, then area of circle should appear. Similarly, for 2 and 3 area of rectangle and triangle should appear. Other parameters like radius, length, breadth, base etc. should be entered through keyboard.

```
//Program 5.5:
#include<iostream.h>
```

```
#include<conio.h>
int main()
{
 float  area,radius,length,breadth,base,height;
 int ch;
 clrscr();
 cout<<"\n1.Area of Circle::";
 cout<<"\n2.Area of Rectangle::";
 cout<<"\n3.Area of Triangle::";
 cout<<"\n\nEnter your choice(1,2,or 3):";
 cin>>ch;
 switch(ch)//here expression inside ( ) is an
integer variable
 {
 case 1: // followed by an integer constant,
which is 1
      cout<<"\nEnter the radius of the circle:";
      cin>>radius;
      area=3.1416*radius*radius; // p = 3.1416
      cout<<"\nArea is "<<area<<" sq. units";
      break;
 case 2: // followed by an integer constant,
which is 2
      cout<<"\nEnter the length & breadth :";
      cin>>length>>breadth;
      area=length*breadth; //area=length*breadth
      cout<<"\nRectangle  area  is  "<<area<<"
sq.units";
          break;
      case 3:
          cout<<"\nEnter the base and height of
the                        triangle:";
          cin>>base>>height;
          area=(base*height)/2.0;//
area=(base*height)/2.0
          cout<<"\nArea  of  the  triangle  is
"<<area<<" sq.units";
          break;
```

```
        default:
            cout<<"\nInvalid choice";
            break;
    }
    getch();
    return 0;
    }
```

EXERCISE-1

1. Write a C++ program to find the largest of the three numbers using *if-else* statement. Numbers should be entered through the keyboard.

2. Write a C++ program using *switch* statement that will examine the value of a floating point variable called *temperature* and display one of the following messages, depending on the value assigned to *temperature.*

(a) ICE, if the value of *tempetarure* is less than 0(zero),

(b) WATER, if the value of *temperature* lies between 0 and 100

(c) STEAM, if the value of *temperature* exceeds 100.

3. What will be output of the following code segment:

```
int main(){
int a=300, b=100, c;
if(a>= 250){
          b=200;  c=b+100;
        }
  cout<<"a=" <<a<<" b= "<<b<<" c= "<<c;
return 0;
}
```

CHECK YOUR PROGRESS

1. Answer the following by selecting the appropriate option:
   (i) In a simple *if* statement with no *else*, what happens if the condition following the *if* is false?
   (a) The program searches for the last *else* in the program
   (b) Both (a) and (c)
   (c) Control falls through to the statement following the *if* statement
   (d) None of these.
   (ii) The advantage of a switch statement over an *if-else* contruct is:
   (a) Several different statements can be executed for each case in a *switch.*
   (b) A *default* condition can be used in the *switch.*
   (c) The *switch* is easier to understand.
   (d) Several different conditions can cause one set of statements to be executed in a *switch.*
2. State whether the following statements are true(T) or false(F):
   (i) No two *case constants* in the same *switch* statement can have identical values.
   (ii) When a *break* is encountered in a *switch,* the control jumps to the end of the *switch* block.
   (iii) It is mandatory to write a *default* statement within a *switch* statement.
   (iv) In case of *if-else-if ladder*, the final *else* is not associated with an *if*.
   (v) The *switch* statement is used in place of many *if-else* statements.

5.4 LOOPS

In programming we often come across some situation where we are to perform some tasks repeatedly. C++ provides loop structures to perform

those tasks which are repetitive in nature. A loop lets you repeat lines of code as many times as you need instead of having to type out the code a whole lot of times. Loops in C++ are mainly of three types:

- ***while*** loop
- ***do-while*** loop
- ***for*** loop

For those who have studied C language, this is not very new as the syntax for the loops are exactly the same.

5.4.1 while loop

The ***while*** loop has the form:

```
while (condition)
{
        statements;
}
```

Here, the given ***condition*** (or ***expression***) is evaluated and if the condition is true then the body of the loop is executed. After the execution of the body, the test condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the test condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the statements immediately after the body of the loop. The body of the loop may have one or more statements. The braces "{" and "}" are needed only if the body contained two or more statements. When using a *while* loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever.

Let us take a small example of a program which displays KKHSOU four (4) times. Here we have written the statement **i++;** with the body of while loop, that increases the value of the variable 'i' that is being evaluated in the condition by one. This will eventually make the condition (i<=4) to become false after a certain number of loop iterations.

//***Program 5.6:*** Demonstration of *while* loop

```
#include<iostream.h>
#include<conio.h>
int main( )
{
     int i=1;
     clrscr( );
     while(i<=4)
     {
         i++;
         cout<<"KKHSOU"<<endl;
     }
     getch();
     return 0;
}
```

The output of the above code will be :

KKHSOU
KKHSOU
KKHSOU
KKHSOU

Let us consider another example where we are using *while* loop. Here we are to display the multiplication table of a number which is entered through the keyboard.

/* ***Program 5.7:*** Displaying multiplication table of a number using while loop*/

```
#include<iostream.h>
#include<conio.h>
int main()
{
     int n,i=1;
     clrscr();
     cout<<"\nEnter a number to display it's
multiplication table:";
     cin>>n;
     while(i<=10)
     {
         cout<<n<<" * "<<i<<" = " <<n*i<<endl;
```

```
            i++;
        }
        getch();
        return 0;
    }
```

If we enter 5 then the output will be:

5*1 = 5
5*2 = 10
5*3 = 15
5*4 = 20
5*5 = 25
5*6 = 30
5*7 = 35
5*8 = 40
5*9 = 45
5*10= 50

5.4.2 *do-while* loop

The ***do-while*** loop is also a kind of loop, which is similar to the *while* loop. In contrast to *while* loop, the *do-while* loop tests the condition at the bottom of the loop after executing the body of the loop. Since the body of the loop is executed first and then the loop condition is checked we can be assured that the body of the loop is executed at least once. The syntax of *do-while* loop is:

```
do
{
    statement block;
} while (condition);
```

Here the statements are executed, then condition is evaluated. If the condition is true then the body is executed again and this process continues till the condition becomes false. Here we have modified *Program 5.7* for demonstration of *do-while* loop.

//Program 5.8: Displaying multiplication table using *do-while* loop.

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

```
int main()
{
    int n,i=1;
    clrscr();
    cout<<"\nEnter a number for multiplication
table:";
    cin>>n;
    do
    {
        cout<<n<<" * "<<i<<" = " <<n*i<<endl;
        i++;
    } while(i<=10);  //end of do-while loop
    getch();
    return 0;
}
```

5.4.3 for loop

The ***for*** loop provides a more concise loop control structure. The general form of the *for* loop is:

```
        for (initialization; condition; increment or decrement)
    {
        statements;
    }
```

Initialization is executed first. Generally, it is an initial value setting for a counter variable. This is executed only once. After *initialization* the ***condition*** is checked and if it is true (i.e., satisfied), the statements inside the loop will be executed. As usual, it can be either a single statement or a block of statements enclosed in braces { }. After that the value of the counter variable will be ***incremented*** or ***decremented***. Then again, the *condition* is checked with the new value of the counter variable. If the *condition* is true, the body of the loop will be executed again. This process continues until the *condition* becomes false. If the *condition* becomes false (i.e., not satisfied), then the statements inside the loop are not executed and the control is transferred to the end of the loop. For demostration of *for* loop let us take an example of displaying all odd numbers between 1 to 20.

//***Program 5.9:*** Displaying odd numbers between 1 to 20

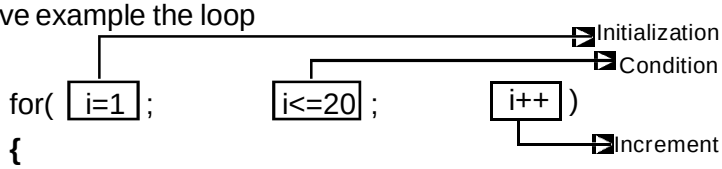
```
#include<iostream.h>
#include<conio.h>
int main()
{
    int i;
    cout<<"The odd numbers between 1 to 20 are:"<<endl;
    for(i=1;i<=20;i++)
    {
        if(i%2!=0)
            cout<<i<<"\t";
        }
        getch();
        return 0;
    }
```

The output will be like this:

The odd numbers between 1 to 20 are:

1 3 5 7 9 11 13 15 17 19

In the above example the loop




```
{
    if(i%2!=0)
    cout<<i<<"\t";
}
```

will execute 20 times. The variable i is incremented by one in each iteration.

The *initialization* part of a *for* loop can also include a declaration statement for a loop variable. Using our previous example, we could have written the loop to include the declaration for the loop counter **i**:

```
for(int i = 1; i <= 20; i++)// i is declared and initialized
    {
      if(i%2!=0)
```

```
    cout<<i<<"\t";
}
```

We can also omit the *initialization* part altogether. If we initialize i appropriately in the declaration, we can write the loop as:

```
int i = 1;
for(  ; i <= 20 ; i++)
{
   if(i%2!=0)
   cout<<i<<"\t";
}
```

But we need the semicolon that separates the *initialization* from the test *condition* for the loop. In fact, both semicolons must be in place. This flexibility also applies to the contents of the *increment/decrement* part.

/***Program 5.10:*** Combining two strings without using string library functions.(Example to show *for* loop without any body)*/

```
#include<iostream.h>
#include<conio.h>
int main()
{
   char a[20],b[20],c[40];
   int length1,length2,i,j;
   clrscr();
   cout<<"Enter the first string:";
   cin.getline(a,20);
   cout<<"Enter the second string:";
   cin.getline(b,20);
   for(length1=0;a[length1]!='\0';length1++);

/*for loop ending with a semicolon */
   for(length2=0;b[length2]!='\0';length2++);
//
   for(i=0;i<length1;i++)
   c[i]=a[i];
   c[i]=' ';
   for(j=0;j<length2;j++)
       c[length1+j]=b[j];
```

```
    c[length1+length2]='\0';
    cout<<"The combined string is: "<<c;
    getch();
    return 0;
}
```

In the statements

```
for(length1=0;a[length1]!='\0';length1++);
for(length2=0;a[length2]!='\0';length2++);
```

we have used the semicolon after the closing parentheses, to indicate that the loop statement are empty. If we omit the semicolon, the statement immediately following the loop will be interpreted as the loop statement. In the above program, these two for loops are simply used to determine the length of the first and the second string.

The infinite loop

When a test *condition* specified in a loop evaluates and returns true forever then statements within the loop are executed infinitely. This form of loop construct is called ***infinite*** loop. Here is an example of infinite loop:

***//*Program 5.11:** Example of infinite loop (Program will run indefinitely)

```
#include<iostream.h>
#include<conio.h>
int main()
{
     int i=1;
     clrscr();
     while(i)
     {
         cout<<i;
         i++;
     }
     getch();
     return 0;
}
```

As we have not specified the limit of 'i' in the *while* loop, so the loop will execute indefinite number of times.

5.5 UNCONDITIONAL BRACHING STATEMENTS

The statements which tranfer the control from one place to another within the program unconditionally are known as ***jump*** or ***unconditional*** statements. The following are three important jump statements:

- ***break*** statement
- ***continue*** statement
- ***goto*** statement

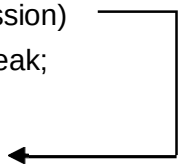
5.5.1 break statement

The *break* statement is used to break out of a loop statement i.e., stop the execution of a looping statement, even if the loop condition has not become false or the sequence of items has been completely iterated over. The *break* statement can be written as:

```
break;
```

Within nested loop, the *break* statement terminates only the *inner* loop that immediately encloses it. The *break* is also used with the *switch* statement to terminate the processing of a particular *case* within a *switch* statement. It passes the control out of the *switch* body to the next statement outside the *switch* statement. The control flow in *for* loop with *break* statement is shown below:

```
for (initialization; condition; increment or decrement)
{
        .....
        if(expression)
              break;
        .....
}
statements;
```



Break statements are necessary within the *switch* statement (Used in *Program 5.5).* Here is another example for the demonstration of *break* statement :

//***Program 5.12*:** Displaying the summation of all entered numbers

```
#include<iostream.h>
#include<conio.h>
int main( )
{
    int i,n,sum=0;
    clrscr();
    cout<<"\nEnter as many number as you wish(0
to Quit): ";
    while(1)
    {
         cin>>n;
         if(n==0)
              break; //use of break statement
         sum=sum+n;
    }
    cout<<"\nThe summation of all entered
nos:"<<sum;
    getch();
    return 0;
}
```

The above program takes as many number as we enter and displays their summation.

5.5.2 continue statement

The *continue* statement passes control to the next iteration of the innermost loop *(while*, *do-while* or *for*) in which it appears, bypassing any remaining statements in the loop body. During program execution, when the *continue* statement is encountered, the control automatically passes to the *condition* which is evaluated; if it is true, the loop is executed again. The *continue* statement can be written as:

```
continue;
```

Usually, *continue* statement is associated with an *if* statement. The control flow in *for*, *while* and *do-while* loop with *continue* statement are indicated with arrows as follows:

```
for (initialization; condition;  increment or decrement)
{
        .....
        if(expression)
              continue;
        .....
}
```

```
        while(condition)
        {
              .................
              if(expression)
                    continue;
              .................
        }
```

```
        do
        {
              if(expression)
                    continue;
              .................
        } while(condition);
```

The following example demonstrates the use of *continue* statement.

/****Program 5.13:*** Program to find the summation of all even numbers between 1 to 10 */

```
#include<iostream.h>
#include<conio.h>
int main()
{
    int n,i,sum=0;
    clrscr();
    cout<<"\nThe even numbers between 1 to 10
are: ";
    for(i=1;i<=10;i++)
    {
    if(i%2!=0)          //if i%2!=0 i.e.,  i is an
```

```
odd number
        continue;// it will skip the odd no. and
increment i
    cout<<"\t"<<i;
    sum=sum+i;
    }
    cout<<"\nThe summation of all even numbers
is "<<sum;
    getch();
    return 0;
}
```

5.5.3 *goto* statement

With the help of *goto* statement one can transfer the control to anywhere in the program. The destination point is identified by a *label*, The *label* is a valid identifier followed by a colon and placed before the statement requiring labeling. The general syntax of *goto* statement is shown below :

```
goto label;
.....
.....
.....
label: statement
```

A program may contain a number of *goto* statements with different label names. Each labelled statement within the program must have a unique label i.e., no two statements can have the same label. Let us demostrate the *goto* statement with an example.

/***Program 5.13:*** Calculating the summation of first *n* natural numbers using *goto* statement. (value of *n* should be entered through key-board) */

```
#include<iostream.h>
#include<conio.h>
int main()
{
    int limit, num,sum=0;
    clrscr();
```

```
    cout<<"\nEnter the limit for addition of
natural numbers\n ";
    cin>>limit;
    num=1;
    target : sum=sum+num;   //target used as
label
    if(num<limit)
    {
        num++;
        goto target;     //goto statement
    }
    cout<<"\nSum of first "<<limit<<" natural
number                   is "<<sum;
    getch();
    return 0;
}
```

In the above example control is transferred repeatedly out of *if* statement to the statement whose *label* is *target* until *num<limit*. If we enter the limit as 10, then the following lines of code behaves like a loop form 1 to 9 in the above program.

```
        num=1;
        target : sum=sum+num;    //target used as
label
        if(num<limit)
        {
                        num++;
                        goto target; / / g o t o
statement
        }
```

The use of *goto* statement should generally be avoided as a number of *goto* statements in a program make the program difficult to understand. Occational situations do arise, however, in which the goto statement can be useful. In a difficult programming situation it seems so easy to use a *goto* to take the control where you want to.

5.6 LET US SUM UP

The key points to keep in mind in this unit are:

- C+ + provides three major decision making statements: ***if***, ***if-else*** and ***switch***.
  - The ***if*** statement by itself will execute a statement or a group of state-ments, when the condition following *if* evaluates *true* (non zero value). When the condition evaluates *false* (zero value) the control falls through to the statement following the *if* statement.
  - In case of ***if-else*** statement, if the expression evaluates *true* then the statement or group of statements following *if* will be executed and if the expression evaluates *false,* then the statement or group of statements following *else* will be executed.
  - The ***switch*** statement is suitable when many conditions are being tested for. It is mainly used to replace multiple *if-else* sequence which is difficult to maintain in a program.
  - The ***switch*** statement successively tests the value of an *expression* against a list of integer or character constants associated with ***case*** and when a match is found, the statement associated with that constants are executed.
  - A program may require that a group of instructions be executed re-peatedly until some logical condition has been satisfied. This is known as ***looping***. Loops are basically means to do a task multiple times, without actually coding all statements over and over again.
  - There are three types of loop: ***while***, ***do-while*** and ***for*** loop. ***While*** loop does not execute when the given condition is *false*. ***Do-while*** loop executes atleast one time even if the given condition is *false*. The ***for*** loop is most popular and it allows us to specify the three things : *initialization*, *condition* and *increment / decrement* in a single line.
  - The ***break*** statement exits out of a loop and transfers the control to the statement immediately following the control structure.
  - The ***continue*** statement skips the remainder of the current iteration in a loop and initiates the execution of the next iteration. The loop does not terminate when a *continue* statement is encountered.
  - The ***goto*** statement unconditionally transfers control to the statement labeled by the specified identifier.

5.7 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education

5.8 ANSWERS TO CHECK YOUR PROGRESS

1. (i) (c) Control falls through to the statement following the *if* statement
   (ii) (c) The *switch* is easier to understand.
2. (i) True (ii) True (iii) False (iv) True (v) True
3. (i) (b) do-while (ii) (a) always true
   (iii) (b) the inner most loop is completed first
   (iv) (c) a label (v) (b) allows the programmer to terminate the loop
   (vi) (d) continue

5.9 MODEL QUESTIONS

1. What is the purpose of *if-else* statements?
2. What are the differences between *break* and *continue* statements? Explain with examples.
3. What is the purpose of the *break* statement? Within which control statements can the *break* statement be included?
4. What is the purpose of *continue* statement? Compare it with the *break* statement.
5. What is the purpose of the *goto* statement? How are the assotiated target statements identified?

6. What happens when the value of the expression in the *switch* statement matches the value of one of the *case* labels? What happens when the value of this expression does not match any of the case labels?
7. What is loop? What are the different types of loops in C++. Give their syntax with example.
8. What is the purpose of *do-while* loop? What is the minimum number of times that a *do-while* loop can be executed?
9. What are the differences between *while* and do-while loop?
10. What is an infinite loop?
11. Write a C++ program to check whether a given number is prime or not?
12. Write a C++ program to determine the sum of all digits of an entered number.

UNIT 6: ARRAY, POINTER AND STRUCTURE

UNIT STRUCTURE

6.1 LEARNING OBJECTIVES

After going through this unit, you will able to :

- describe manipulation of an array
- describe the concept of pointers and its implementation
- describe how to use the structures

6.2 INTRODUCTION

Till now, we have discussed about the elements of C++ language including operators and conditional statements. In this unit, we will concentrate on the dicussion of array, pointer, structure and union.

6.3 ARRAYS

Arrays are the basic building blocks for more complex data structures. We have already been familiar with the array. An array is a similar collection of series of data elements (or variables) in which all the elements are of same type and are stored consecutively in memory. Each array element (i.e., each individual data item) is referred to by specifying the array name followed by one or more ***subscripts,*** with each subscript being enclosed in square brackets. The syntax for declaration of an one dimensional array is

Storage Class datatype arrayname [expression];

Here, *storage class* may be *auto, static or extern. Storage class* refers to the permanence of a variable, and its scope within the program, i.e. the portion of the program over which the variable is recognized. If the storage class is not given then the compiler assumes it is an *auto* storage class.

The one dimensional array can be declared as :

```
int x[15];//x is a 15 element integer array
char name[25];//name is a 25 element character array
```

In the array x, the array elements are x[0], x[1],....., x[14] as illustrated in the fig. 6.1.

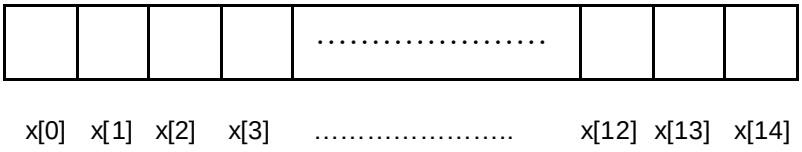


Fig 6.1 : A one dimensional array data structure

Array can be initialized at the time of the declaration of the array. For example,

int marks [5] ={ 85, 79, 60, 87, 70 };

Then, the ***marks*** array can be represented as follows :

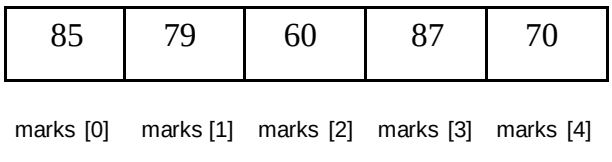


Fig. 6.2 : the marks array after initialization

The number of the subscripts determines the ***dimensionality of the array.*** For example,

marks [i],

refers to an element in the ***one dimensional array***. Similarly, **matrix [i] [j]** refers to an element in the ***two dimensional array***.

Two dimensional arrays are declared in the same way as that one dimensional arrays. For example,

```
int matrix[3][5];
```

is a two dimensional array consisting of 3 rows and 5 columns for a total of 20 elements. Two dimensional array can be initialized in a manner analogous to the one dimensional array :

```
int matrix [3][5] =     {
                        { 10, 5,  -3, 9, 2 },
                        {  1 , 0, 14, 5, 6 },
                        { -1, 7,  4,  9,  2 }
                      };
```

The matrix array can be represented as follows: :

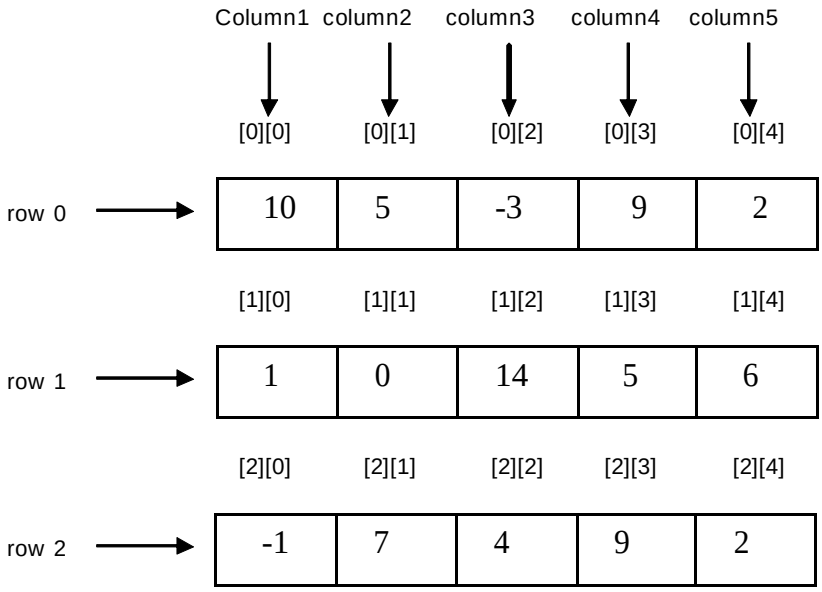


Fig. 6.3 : Matrix array after initialization

The above statement can be written as follows :

```
int matrix[3][5]={10,5,-3,9,2,1,0,14,5,6,-1,7,4,9,2};
```

A statement such as

```
int matrix [ 3 ][ 5 ] = {
                        { 10, 5,  -3 },
                        {  1 , 0, 14 },
                        { -1, 7,  4  }
                      };
```

only initializes the first three elements of each row of the two dimensional array. The remaining values are set to 0.

The following program demonstrating the use of two dimensional matrix. The program read a matrix and calculate the sum of the upper diagonal elements and lower diagonal elements.

// Program 6.1

```
#include<iostream.h>
#include<conio.h>
void main()
  {
    clrscr();
    int a[20][20],i,j,m,n,csum=0,dsum=0;
    cout<<"Enter the size of the Matrix\n";
    cin>>m>>n;
    if(m!=n)
    cout<<"You cannot get a proper Diagonal with this
size \n";
    else
    {
    cout<<"Enter elements of the Matrix\n";
    for(i=0; i<m; i++)
    {
        for(j=0; j<n; j++)
        cin>>a[i][j];
    }
    cout<<"Matrix is :\n";
    for(i=0; i<m; i++)
    {
        cout<<"\n";
        for(j=0; j<n; j++)
        cout<<" "<<a[i][j];
    }
    csum=0;
    for(i=0; i<m; i++)
    for(j=0; j<n; j++)
    {
        if(i<j)
        csum += a[i][j];
    }
    cout<<"\nSum of the Elements\n Above the diagonal
:";
    cout<<csum<<"\n";
    dsum=0;
```

```
for(i=0; i<m; i++)
for(j=0; j<n; j++)
{
    if(i>j)
    dsum += a[i][j];
}
cout<<"Below the diagonal :";
cout<<dsum<<"\n";
}
```

OUTPUT

Enter the size of the Matrix : 3 3
Enter elements of the matrix : 1 2 3 4 5 6 3 2 1
Matrix is : 1 2 3
4 5 6
3 2 1
Sum of the Elements
Above the diagonal : 11
Below the diagonal : 9

String Manipulations

Strings are used in programming language for storing and manipulating text, such as word, names and sentences. We have already been familiar with the declaration of a character array for storing which looks like–

char array-name [size];

e.g.char name [30];

which stores 30 bytes of memory for storing a set of characters.

In C++, we can also initialize the characters or strings at the time of declaration of the variables like the following -

char name[4]={'a', 'b', 'c', 'd'};

It can also be declared without specifying the size of the array as follows :

char name [ ] = { "krishna" } ;

K	r	i	s	h	n	a	\0
name [0]	name [1]	name [2]	name [3]	name [4]	name[5]	name[6]	name[7]

Fig. 6.4 : name array after initialization

Remember that every character string is terminated by a null character (\0). Some more declarations of arrays with initial values are given below :

```
char vowels [ ] = { 'A', 'E', 'I', 'O', 'U' };
char colour [ ] = { 'v','I','B','G','Y','O','U','R'};
```

In the above case, the compiler assumes that the array size is equal to the number of elements enclosed in the curly braces. Thus, in the above statements, size of array would automatically be assumed to be 5. If the number of elements in the initializer list is less than the size of the array, the rest of the elements of the array may remain uninitialized or may be initialized to zero or garbage value depending on the compiler.

In C++ also, you will get several built-in functions like *strlen(), strcat(), strcpy()* etc.

Here, *strlen()* – returns the length of a given function

strcat() – concatenates two strings resulting in a single string

strcpy() – copies the contents of one string to another

strcmp() – compares lexicographically two string and returns integer based on the outcome shown below

greater than zero, if string1>string2
zero, if string1==string2
less than zero, if string1<string2

To use these functions, the header file *string.h* must be included in the program using the statement

#include<string.h>

The following program demonstrates the use of string functions.

//Program 6.2

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
void main()
  {
    char str1[20], str2[20];
    clrscr();
    cout<<"Enter 1st string:";
    cin>>str1;
    cout<<"Enter 2nd string:";
```

```
    cin>>str2;
    cout<<"\n";
    cout<<"Length of 1st string:"<<strlen(str1)<<"\n";
    cout<<"Length of 2nd string:"<<strlen(str2)<<"\n";
    int flag=strcmp(str1,str2);
    if(flag==0)
    cout<<str1<<"Equals  to"<<str2<<"\n";
    else
if(flag>0)
cout<<str1<<"Greater  than"<<str2<<"\n";
else
cout<<str1<<"Less  than"<<str2<<"\n";
cout<<"String  after  concatenation  :"<<  strcat(str1,
str2);
getch();
}
```

OUTPUT

Enter 1st string : computer
Enter 2nd string : programming
Length of 1st string : 8
Length of 2nd string : 11
computer Less than programming
Resultant string after concatenation : computerprogramming

Arrays of Strings

Let us consider a two dimensional character array as–

char name[15][20];

Here, the name array can store 15 names, each of length of 19 character, because we know that the last character is used to represent '\0'. The first name is accessed by the expression *name[0]*, and second name by *name[1]*, and so on. If we write *name[5][2]*, then it will represent the 3rd character of the 5th name, similarly the expression *name[0][4]* will represent the 5th character of the first name. Such types of two dimensional array of characters are known as array of strings. The array ***char colour[3][5]*** is represented in Figure 6.5.

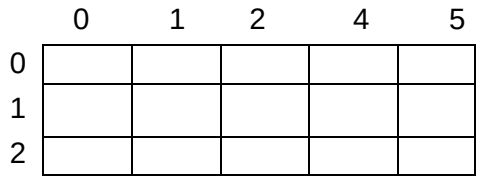


Fig. 6.5 : An array of strings representation

6.4 POINTERS

Each memory location that you use to store a data value has an address. A pointer is a variable that stores **an address** of another variable(not the **value**) of a particular type. A pointer has a variable name just like any other variable and also has a type which designates what kind of variables its contents refer to.

Suppose we define a variable called ***sum*** as follows :

int sum = 25;

Let's now define another variable, called ***pt_sum*** in the following way

int *pt_sum;

It means that ***pt_sum*** is a pointer variable pointing to an integer, where * is a unary operator, called the ***indirection operator,*** that operates only on a pointer variable.

We have already used the '**&**' *unary operator* as a part of a *scanf* statement in our C programs. This operator is known as the ***address operator***, that evaluates the address of its operand.

Now, let us assign the address of sum to the variable **pt_sum** such as

pt_sum = ∑

Now the variable **pt_sum** is called a ***pointer*** to **sum**, since it "points" to the location or address where **sum** is stored in memory. Remember, that ***pt_sum*** represents ***sum's*** address, not its value. Thus, ***pt_sum*** referred to as a ***pointer variable***.

The relationship between pt_sum and sum is illustrated in Figure 6.6

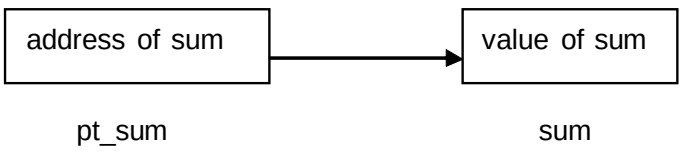


Fig. 6.6 : Relationship between pt_sum and sum

The data item represented by **sum** (i.e., the data item stored in sum's memory cells) can be accessed by the expression ***pt_sum***.

Therefore, ***pt_sum** and **sum** both represent the same data item i.e. 25.

Several typical pointer declarations in C program are shown below

```
int *alpha ;
char *ch ;
float *s ;
```

Here, alpha, ch and s are declared as pointer variables, i.e. variables capable of holding addresses. Remember that, addresses (location nos) are always going to be whole numbers, therefore pointers always contain whole numbers.

The declaration **float *s** does not mean that **s** is going to contain a floating-point value. What it means is, **s** is going to contain the address of a floating-point value. Similarly, **char *ch** means that **ch** is going to contain the address of a char value. The following program shows the use of pointers in a program.

// Program 6.3 : Use of pointers in a program

```
#include <iostream.h>
#include<conio.h>
void main( )
  {
       int a = 5;
       int *b;
       b = &a;
       clrscr();
       cout<<"value of a:"<<a;
       cout<<"\n value of a:"<<*(&a);
       cout<<"\n value of a:"<<*b+1;//notice how value of a
changes
       cout<<"\n address of a:"<<&a;
       cout<<"\n address of a:"<<b;
       cout<<"\n address of b:"<<&b;//address of b is
different
       getch();
  }
```

Output

value of a : 5
value of a : 5
value of a : 6
address of a : 0x8fa9fff4
address of a : 0x8fa9fff4
address of a : 0x8fa9fff2

Void pointer

Let us consider the following pointer declarations :

```
float *ptr1; // pointer to float
char str;    // character variable
```

Now, if we write an assignment statement like :

```
ptr1 = &str;
```

the compiler will generate an error, because the type of both the variable is not same i.e. incompatible. Such types of compatibility problems can be overcome by using a general purpose pointer called ***void pointer***. The format for declaring a void pointer is as follows :

```
void *vd_ptr;        // declaring a void pointer
```

The reserved word **void** is used for specifying the type of the pointer. Pointers defined in this manner do not have any type associated with them and can hold the address of any type of variable.

The following are some valid C++ statements :

```
Void *vd_ptr;
int *num_ptr;
int marks;
char name;
float percnt;
vd_ptr = &marks;
vd_ptr = &name;
vd_ptr = &percnt;
num_ptr=&marks;
```

The following are some invalid statements :

```
num_ptr = &name;
num_ptr = &percnt;
```

Pointer to Pointer

A pointer to a pointer is a technique used frequently in more complex

programs. To declare a pointer to a pointer, place the variable name after two successive asterisks (*). In this case one pointer variable holds the address of the other pointer variable. The following statement shows a pointer to pointer :

```
int  **x;
```

The following program shows the use of pointer to pointer techniques :

//Program 6.4 : Use of pointer to pointer

```
#include<iostream.h>
#include<conio.h>
void main()
{
  int i;
  int *ptr;//declaration of pointer variable
  int  **pptr;//declaration  of  pointer  to  pointer
variable
  clrscr();
  i=6;
  ptr=&i;            // address of i stored in ptr
  pptr=&ptr;         // address of ptr is stored in
pptr
  cout<<"i="<<i<<"\n";
  cout<<"*ptr="<<*ptr<<"\n";// *ptr means value of i
  cout<<"**pptr="<<**pptr<<"\n";// **ptr means value
of i
      getch();
  }
```

RUN :

```
i=6
*ptr=6
**ptr=6
```

Array of pointers

An array of pointers is an array, that contains a collection of addresses. The elements of such arrays (i.e. addresses) are stored in memory just like the elements of any kind of arrays. We can declare an array of pointers in the same way as we declare a normal array. The syntax for declaring an

array of pointers is as follows :

datatype *ArrayName[size];

With an array of pointers of type char, each element can point to an independent string, and the lengths of each of the strings can be different. The following program shows the use of array of pointers technique.

// Proogram 6.5 : Use of array of pointers

```
#include<iostream.h>
#include<conio.h>
void main()
  {
       int flag = 0;
       clrscr();
       // Initializing a pointer array
       char *pstr[ ] = {"Red",
                        "Green",
                        "Yellow",
                        "Orange",
                        "Blue",
                        "violet"
                       };
   char *pstart = "Your favourite colour is :: ";
       /*notice the new style of cout statement*/
   cout<< " Pick a favourite colour!\n"
   << " Enter a number between 1 and 6: ";
   cin >> flag;
   cout << "\n";
   if(flag >= 1 && flag <= 6)    //  Check  input
validity
   cout  <<  pstart  <<  pstr[flag-1];//  Output  colour
name
   else
       // Invalid input
   cout << "Sorry, you haven't got a colour.";
   cout << endl;
   getch();
}
```

Output

Pick a favourite colour!
Enter a number between 1 and 6 : 2
Your favourite colour is :: Green

In the program, in *char *pstr[ ]* we have not specified the size of the array. It means that we can allocate memory as we need. The array *char *pstr[ ],* and how it will act in the program, is pictorially shown in Figure 6.7 below.

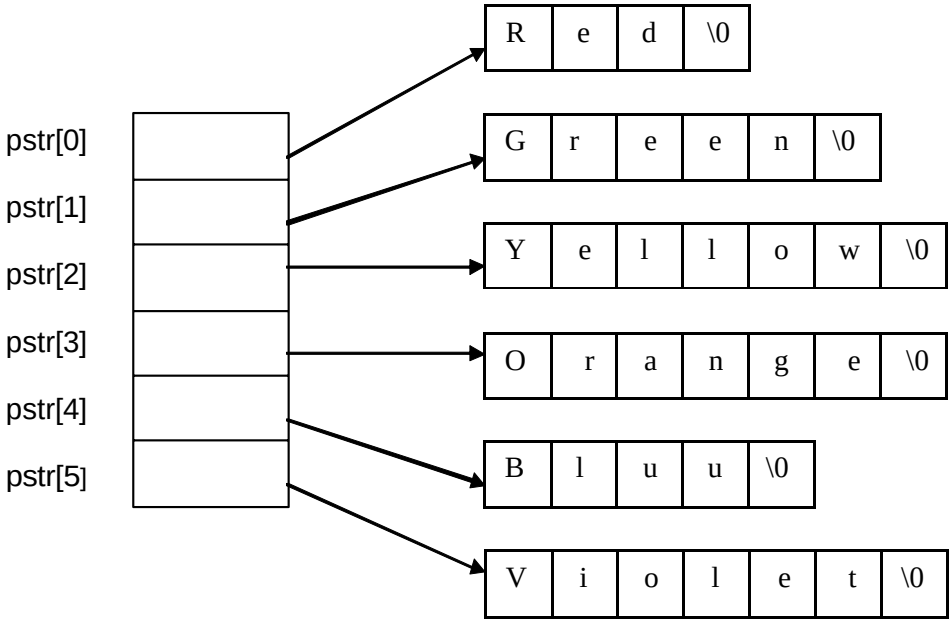


Figure 6.7 : Representation of array of pointers *char *pstr[ ]*

Pointers to functions

A pointer-to-function can also be defined to hold the starting address of a function. Even though a function is not a variable, it still has a physical location in memory that can be assigned to a pointer. This address is the entry point of the function and it is the address used when the function is called. Once a pointer points to a function, the function can be called through that poiinter. Function pointers also allow functions to be passed as arguments to other functions. The syntax of defining a pointer to a function is shown below :

returntype (*ptr_to_func) (arguments);

Note that in the definition the return type of the function(to which the pointer variable points) and the argument lists of the function also has to be specified. It should be remembered that the function prototype or

definition should be known before its address is assigned to a pointer.

After defining a pointer to a function, it will point to any function with which its return type and arguments list will match. As an example, let us declare a pointer to a function such as - its return type is integer and there are two arguments as follows -

*int (*point) (int, int);*

Now, it points to the following functions :

int sum(int x, int y);
int max(int m, int y);

The statement of the form–

point = sum;
or
point = max;

will assign the address of the function to pointer variable ***point.***

For invoking the function, say *int sum(int x, int y);* we will have to write a statement as follows–

(*point)(a,b);

where a,b are two parameter passing to the function.

The following program shows the use of pointer to function.

//**Program 6.6 :** Use of pointer to function

```
#include<iostream.h>
#include<conio.h>
int maximum(int a, int b)
{
  int max;
  if(a>b)
    max=a;
  else
    max=b;
  return max;
}
void main()
{
  int x, y, z;
```

```
  clrscr();
  int (*point)(int, int);//declaration of pointer to
function
  point=maximum;          //assigns the address of
function
  cout<<"Enter the two numbers:";
  cin>>x>>y;
  z=(*point)(x,y);        //calls the function maximum
  cout<<"The Maximum is :"<<z;
  getch();
}
```

OUTPUT

Enter two numbers : 12 6

Maximum is : 12

CHECK YOUR PROGRESS

1. Describe the output generated by the following program :

```
#include<iostream.h>
void main()
  {
    int a, b=0;
    static int c[10] = { 1,2,3,4,5,6,7,8,9,0}
    for(a=0; a<10; ++a)
      if((a%2)==0) b+=c[a];
    cout<<"b="<<b;
  }
```

2. Explain the meaning of each of the following declarations

   a) float a, b;
      float *pa, *pb;

   b) float a = -0.167;
      float *pa = &a;

   c) char *d[4] = {“north‘, ‘south”, “east”, “west”};

   d) float (*x)(int *a);

6.5 STRUCTURES

Structure is a user defined data type in C++. A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure. But in an array, all the data items are of the same type. The individual variables in a structure are called *member variables*. A structure is a convenient way of grouping several pieces of related information together.

Here is an example of a structure declaration.

```
struct student
  {
       char name[25];
       char course[20];
       int age;
       int year;
  };
```

Declaration of a structure always begins with the key word ***struct*** followed by a user given name, here the **student**. Recall that after the opening brace there will be the member of the structure containing different names of the variables and their data types followed by the closing brace and the semicolon.

Graphical representation of the structure *student* is shown in Fig. 6.8 :

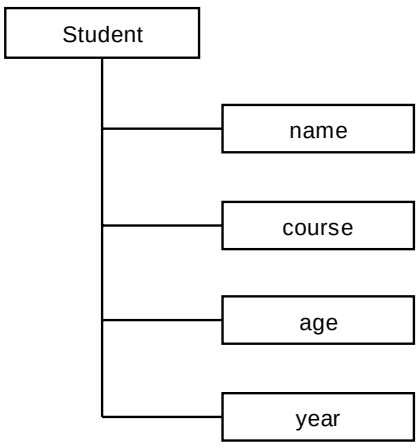


Figure 6.8 : A structure named Student

After declaring the structure of any type, the next step is to create structure variables. At the time of creation of the structure variables, storage space are also allocated for them. The syntax for structure variable definition

is shown below–

structurename var1, var2..... ;

e.g. Student s1, s2;

where s1, s2 are structure variable of type student.

We can create structure variables during the structure declaration as follows :

```
struct student
  {
       char name[25];
       char course[20];
       int age;
       int year;
  } s1, s2;
```

Accessing Structure Members :

C++ provides the *period operator or dot (.)* operator to access the members of a structure independently. The dot operator connects a structure variable and its member. The member variables of the structure *student* are accessed by the variable ***s1*** as shown below :

s1.name	*s1.course*
s1.age	*s1.year*

The following program uses the structure data type to read and display the information about a book.

***//* Program 6.7**

```
#include<iostream.h>
#include<conio.h>
    struct book
      {
           char name[20] ;
           float price ;
           int pages ;
    } ;
     void main( )
    {
      struct book b1;
      clrscr();
```

```
    cout<<"\nEnter Data for Book :\n";
    cout<<"Book  Name  :";
    cin>>b1.name;
    cout<<"\nPrice :";
    cin>>b1.price;
    cout<<"\nPage :";
    cin>>b1.pages;
    cout<<"==========================";
    cout<< "\nYou have entered - \n";
    cout<<"Book Name:"<<b1.name<<"\n";
    cout<<"Price :"<<b1.price<<"\n";
    cout<<"Pages :"<<b1.pages<<"\n";
    getch();
}
```

OUTPUT

```
Enter Data for Book :
Book Name : Data Structlure through C++
Price :500
Page : 250
==============================
You have entered -
Book Name: Data Structlure through C++
Price :500
Page : 250
```

Array of structure

Now let us see how to declare an array of structure. In the following we have declared a variable **st_rec** of type student :

student st_rec[100];

In this declaration **st_rec** is a 100 element array of structures. Each element of **st_rec** *i.e st_-rec[1], st_rec[2], st_rec[3].....st_rec[99]* are separate structure of type *student* it means each element of **st_rec** represents an individual student record.

The representation of st_rec[100] is shown in the Figure. 6.9.

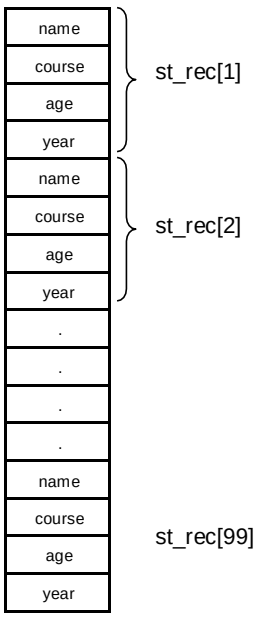


Figure 6.9 : Array of structure representation

Let us see how the elements of such structure can be accessed. As an example, if we want to access the name of the 10th student (i.e. **st_rec[9]**) from the above structure then we will have to write

st_rec[9].name

Similarly, course and age of the 10th student can be accessed by writing

st_rec[9].course and **st_rec[9].age**

The members of a structure variable can be assigned initial values in much the same manner as the elements of an array. Example of assigning the values for the 10th student record is shown in the following:

struct student **st_rec[9]** = { “Arup Deka”, “BCA”, 21, 2008 };

Now let us try to write a program using array of structure to display *id_no, name, address, age* of 20 voters whose age exceed 45 by assuming suitable data types.

***//* Program 6.8**

```
#include<iostream.h>
#include<conio.h>
struct voter
    {
      int  id_no;
      char name[25];
      char address[50];
      int age;
    } ;
    void main()
    {
      struct voter ward[20];//declaration of array
of structure
      clrscr();
      for(int i=0; i<2; i++)
      {
           cout<<"\n"<<"Voter No."<<i+1;
           cout<<"\n"<<"Enter Id_No :";
           cin>>ward[i].id_no;
    cout<<"\n Enter Name :";
```

```
        cin>>ward[i].name;
        cout<<"\n Enter Address :";
        cin>>ward[i].address;
        cout<<"\n Enter Age :";
        cin>>ward[i].age;
}
for(i=0; i<20; i++)
{
        if(ward[i].age>45)
        {
          cout<<"Id_No :";
          cout<<ward[i].id_no;
          cout<<"\n Name :";
          cout<<ward[i].name;
          cout<<"\n Address :";
          cout<<ward[i].address;
          cout<<"\n";
        }
}
getch();
}
```

CHECK YOUR PROGRESS

3. Give appropriate declaration for the following :
   (i) An array ***temp*** to store 30 temperatures along with their corresponding dates dd/mm/yyyy.
   (ii) Give the output of the following program :

```
#include<iostream.h>
struct point
   {
     int X, Y;
   };
void show(point P)
   {
     cout<<P.X<<":"<<P.Y<<"\n";
```

```
    }
void main()
   {
     Point U= {20,10}, V, W;
     V=U;
     V.X +=20;
     W=V;
     U.Y +=10;
     U.X +=5;
     W.X -=5;
     show(U);
     show(V);
     show(W);
   }
```

6.6 UNIONS

We have come to know that, in the case of structures, the amount of memory required to store a structure variable is the sum of the size of all the members. In the fig. 6.9 each variable i.e. st_rec[1], st_rec[2] etc are occupying 49 (i.e. 25+20+2+2) bytes, so the amount of memory occupied by the whole structure is 49 x 100=4900bytes (approximately 5MB).

C++ offers another data type called *union* which is similar to structure datatype except that the members of a union variable occupy the same locations in memory i.e. share the storage space. The declaration of a union type must specify all the possible different types that may be stored in the variable. The form of such a declaration is similar to declaring a structure.

An example of declaring a union type is shown below :

```
union student
  {
    char name[25];
    char course[20];
    int age;
    int year;
  };
```

The union variables of the above ***union student*** can be defined as follows :

```
union student s1;
```

Here, **s1** is the union variable and the memory requirement to store **s1** is equal to the member variable having maximum size. In our above example, the member variable having maximum size is *char name[25]* which is 25 bytes. So, the size of **s1** will be 25 bytes. At any point of time, the union variable can hold data of any one of its members. The following program illustrates the memory requirements for variables of the structure and union types.

// Program 6.9

```
#include<iostream.h>
#include<conio.h>
struct
  {
    char name[25];
    char course[20];
    int age;
    int year;
  } s1;
union
  {
    char name[25];
    char course[20];
    int age;
    int year;
  } s2;
void main()
  {
    clrscr();
    cout<<"The size of Structure is
    :"<<sizeof(s1)<<"\n";
    cout<<"The size of Union is :"<<sizeof(s2)<<"\n";
    getch();
  }
```

RUN :

The size of Structure is : 49

The size of Union is : 25

Unions obey the same syntactic rules as structures. We can access elements with either the dot operator (.) or the right arrow operator (->).

The following program demonstrates the use of union data type :

***//* Program 6.10**

```
#include<iostream.h>
#include<conio.h>
void main()
{
  union data
  {
    int a;
    float b;
  };
  union data d;
  clrscr();
  d.a=20;            // assigns value to member
variables
  d.b= 195.25;
  cout<<"First member is :"<<d.a;
  cout<<"\nSecond member is :"<<d.b;
  getch();
}
```

OUTPUT

First member is : 20

Second member is : 195.5

6.7 LET US SUM UP

- An array is a collection of two or more adjacent memory locations containing same types of data.
- A pointer is a memory variable that stores a memory address of

another variable. It can have any name that is valid for other variable and it is declared in the same way as any other variable. It is always denoted by '*'.

- A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure.
- A structure which contains a member field that points to the same structure type is called a self-referential structure.
- Unions are similar as structures and the only difference is that the members of a union variable occupy the same locations in memory.

6.8 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education

6.9 ANSWERS TO CHECK YOUR PROGRESS

1. 25, sum of the even array elements
2. (a) a and b are floating point variables, pa and pb are pointers to floating point quantities (though not necessarily to a & b)
   (b) a is a floatinig point variable whose initial value is -0.167; pa is a pointer to a floating point quantity, the address of a is assigned to pa as an intial value.
   (c) d is a one dimensional array of pointers to the string 'north', 'south', 'east' and 'west'.
   (d) x is a pointer to function that accepts an argument which is a pointer to an integer quantity and returns a floating point quantity.
3. a) struct date

```
{
    int dd;
    int mm;
    int yyyy;
```

```
    };
  struct temp_det
    {
       float temp;
       date dt;
    };
  temp_det temp[30];
```

b) 25 : 20
40 : 10
35 : 10

6.10 MODEL QUESTIONS

1. What are arrays? Discuss how the elements of one-dimensional array are stored and accessed?
2. Write a program in C++, to find the sum of diagonal elements of a square matrix.
3. Write a program by applying bubble sort algorithm on an array ARR containing 8 elements : 71, 31, 41, 9, 84, 11, 60, 51.
4. Write a C++ program to search for an item in a sorted array using linear search algorithm.
5. Define a pointer. What is the relationship between an array and a pointer?
6. Differentiate between– *p and **p
   &p and *p
7. What is a structure? How is a structure different from an array?
8. What is meant by array of structure?
9. Using pointer concept write a function that takes two string arguments and returns a string which is the larger of the two. Also show how this function will be invoked from main().
10. Write a program in C++ to prepare the marksheet of a college examination and the following items will be read from keyboard
    student name, subject, internal marks & external marks
    Assume that a student will fail if total marks (internal+external marks) is <50%
    Prepare a list of students showing separately those who have failed and those who have passed in the examination.
11. What is union? Differentiate between structure and union.

****

UNIT 7 : FUNCTIONS

UNIT STRUCTURE

7.1 LEARNING OBJECTIVE

After going through this unit you will be able to learn about

- describe library function and user defined function
- describe function declaration, definition, call, actual parameter, formal parameter, return type.
- define inline function, default arguments, macro function

.

7.2 INTRODUCTION

Reading input data, processing data and writing results are the three important components of a computer program. In our previous discussion we have used some standard functions such as **`getch( ), clrscr(),`**

pow(). A function is a self contained program to perform some specific tasks. It is very easy to write C++ program using function. Because most of the functions are already defined and have to be used only. C++ functions can be classified into two categories - **Library function** and **user defined function**. main() is an example of user defined function. Library functions are predefined and attached with the C++ compiler. It reduces complexcity of program and programming time.

7.3 LIBRARY FUNCTION

A library function is a program to perform some specific task. C++ is rich in library functions. All the related functions are put together in a single group known as **library**. For example, all mathematical functions are stored into a single file known as **math library**. C++ standard functions are classified into different categories. Some of these are given below.

* I/O functions
* Mathematical functions
* Data conversion functions
* Character functions
* String functions
* Memory allocation functions
* Graphics functions
* Time related functions and
* DOS interface functions

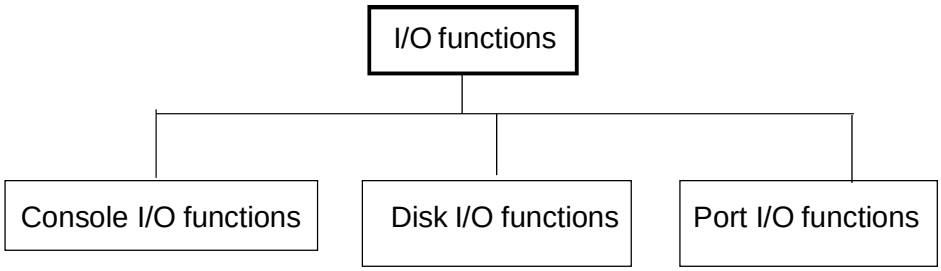
7.4 I/O FUNCTIONS

A computer program is generally used to accept data, process these data and to display the result onto some media. Data are stored into variables to process them. Key board is the most common input device through which data can be supplied to the computer program and monitor(VDU) is the most commonly used output device onto which data can be displayed. Apart from keyboard, there are other devices such as mouse, scanner, disk ect. through which data can be supplied to the program. Though C has no provision for I/O, still with the help of standard I/O library functions it is possible to do it. Library functions are not part of C++'s formal definition, they have become a standard feature of C++

language. I/O functions take a major role in the data input and output process. Based on the input and output devices, I/O functions can be classified into three categories. In this section we will discuss I/O functions related to keyboard and monitor only. The other functions will be discussed in the subsequent sections.

(I/O functions related to Keyboard and monitor) (I/O functions related to Floppy disk and Hard disk) (I/O functions related to various ports)

The I/O functions are classified into three categories—



7.5 UNFORMATTED I/O FUNCTIONS

Unformatted functions are easier to use. They do not need to use any control string. There are several input and output functions through which we can accept data and display them on the monitor. We can accept a single character or a set of characters. Similarly, we can display a single character as well as a set of characters using output functions.

Input functions

getch( )
getche( )
getchar( )
gets( )

Output functions

putch( )
putchar( )
puts( )

getch( ) : This function accepts a character from the keyboard and does not echo the typed character on the screen. It will read a character just after it is typed without waiting for the enter key to be pressed. In most of our previous examples, we have used this function. One

might think that C++ program always ends with getch( ) - which is not true. The function getch() can be used anywhere in a C++ program to accept a single character. Most often this function is used to hold the execution of a program, or to see intermediate results or to stop rolling the screen.

getche() : The function getche( ) is similar to getch( ) except that getche( ) will echo the typed character on the screen whereas getch( ) does not echo the typed character on the screen.

getchar( ) : getchar( ) also reads a character from the keyboard and echo it on the screen just after it is typed. It requires the enter key to be pressed.

gets( ) : It accepts a string or a group of words from the keyboard at a time which is not possible by cin or getch( ) . It is used to read a sentence from the keyboard. The reading process will terminate when an enter key is hit.

Program 7.1 : Program to show the use of library functions

```
void main()
{
      char a, b, c;
      clrscr();
      cout"Press any key to continue : \n";
      a=getch();
      cout<<"Type any character \n";
      b=getche();
      cout<<"Press any key and press enter key \n";
      c=getchar();
      cout<<"Outputs are : ";
      putch(a);
      putch(b);
      putch(c);
      getch();
}
```

putch( ) and **putchar( )** are just opposite to **getch( ).** They are used to display a single character on the screen. **puts( )** is also just opposite to **gets( )**. It is used to display a string or a sentence on the screen.

Program 7.2 : Program to read your name and title from the key-board and display it

```
void main()
{
      char name[30];
      clrscr();
      cout<<"Your name please: ";
      gets(name);
      puts("Hello ");
      puts(name);
      getch();
}
```

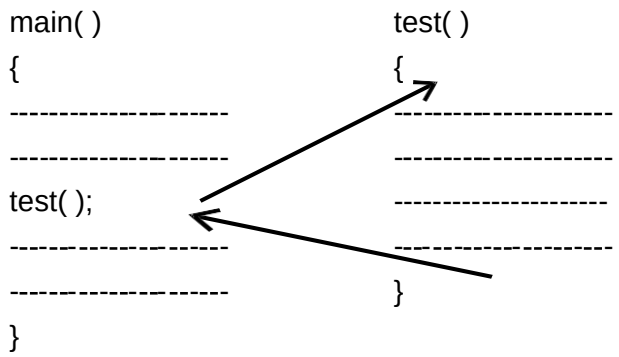
Output

```
Your name please: Ankita Talukdar
Hello Ankita Talukdar
```

7.6 USER DEFINED FUNCTIONS

As already mentioned C and C++ are rich in functions. Functions are normally used to perform certain task. A function may contain a single statement or a set of statements or there may be a function without any statement inside the body of the function. Every C/C++ program has at least one user defined function that is **main()**, which is responsible for the final execution of the program. One may be confused why user defined functions are needed, since C++ is rich in standard functions. It is because of the variety of applications. For example, if we have to calculate salary of 100 employees, C++ standard function can not do this job, because the formula for calculating salary is different for different organizations. Moreover, we can do this job without writing functions, but we have to write same line of codes 100 times. So it increases the size of the program and program complexity unnecessarily. An application may contain several activities. For example in a college library there are so many activities such as maintaining books stock, issuing books to students, return books, checking validity for library member etc. For each and every activity we can write functions to make complete the application. Though for the beginners it may be little bit difficult to write functions themselves, still perhaps all professional programmers prefer program development by using only user defined functions due to their simplicity. Functions are just

like co-workers for the main worker. The general form of a function is given below.



7.7 KEY TERMS RELATED TO FUNCTIONS

To be familiar with functions, one should understand and remember the following key terms.

- Function declaration
- Function call
- Function definition
- Actual parameter
- Formal parameter
- Return type

7.8 RULES FOR WRITING FUNCTION

(a) Every function should have a unique name which must be self-explanatory.
(b) A function should be declared before it is used.
(c) Function must be defined.
(d) The header of a function definition and a function declaration must be same.
(e) Function declaration is to be terminated with semicolon(;)
(f) The parent or owner function is called a **calling function** and the child function is called a **called function**.
(g) A function accepts many parameters and returns only single value through **return** statement.
(h) Parameters used in function definition or declaration are called **formal argument** and the parameters used to call a function are called

actual arguments.

(i) The number of formal arguments, type of formal arguments and order of formal arguments must be same with the number of actual arguments, type of actual arguments and order of actual arguments.

(j) A function may be called many times and from other functions using the function name.

(k) If a function does not return any value, then the return type of the function declaration must be **void**. Void is a keyword to indicate that the function does not return any value.

(l) A same function may be a calling function and/or a called function.

7.9 SYNTAX FOR FUNCTION DECLARATION

```
<return type> <function name>(argument list);
```

Function declaration statements contain three parts i.e. **return type**, **function name** and **argument list** of which return type and argument list are optional. The default return type is **int**. Return type may be any valid C/C++ data type. A function declaration is not always mandatory in C++. In absence of function declaration, the compiler always assumed that the function would return an integer value.

Example

```
void sum( );        //function does not return value and no parameter passing
int sum( );         //function return integer value and no parameter passing
int sum(int a, int b);      //function return one integer value and accept two integer parameters.
int sum(int, int);
float sum(int a, float b);
```

7.10 CATEGORIES OF FUNCTIONS

Functions are categorised into four ways based on the parameters passed and the return type.

(a) Function with no parameter and no return type.

(b) Function with parameter(s) and no return type.

(c) Function with no parameter(s) and return type

(d) Function with parameter(s) and return type.
It may be described in tabler view as shown below

	Passing Parameters	Return type
Category I	No	No
Category II	Yes	No
Category III	No	Yes
Category IV	Yes	Yes

One may be confused with the category of functions to be used and when it is as per the convenience of the programmer. The C/C++ compiler does not restrict the programmers on the use of any category of functions.

(a) Function with no parameter and no return type.

Program 7.3: Function without declaration

```
            void main()
            {
            clrscr();
            india();   //   Function call
            getch();
      }
india()     //Function definition
{
cout<<"Hello I am inside India ";//Called function
}
```

Program 7.4 : Called and calling function

```
      void main()
      {
            clrscr();
            india();
            getch();
      }
      india()
      {
            printf("Hi! I am inside india\n ");
            assam();
```

```
}
assam()
{
      printf("Hi! I am inside Assam\n ");
      guwahati();
}
guwahati()
{
      printf("Hi! I am inside Guwahati ");
}
```

Program 7.5 : Addition of two numbers using functions

```
void main()
{
      clrscr();
      sum();
      getch();
}
sum()
{
      int a=10, b=20,c;
      c=a+b;
      cout<<"Sum= "<<c;
}
```

Output Sum=30

Here the function sum() calculates the sum of two numbers and does not return any value to the calling program i.e. main().

(b) Function with parameter(s) and no return type

A function may accept any number of parameters of any type. When calling a function one must be careful about the number, type and order of parameters which must match with the formal paramers of the called function.

Program 7.6 : Addition of two numbers

```
void main()
{     int a=10, b=20;
      clrscr();
      sum(a,b);
      getch();
}
```

```
sum(int p, int q)
{
      int c;
      c=p+q;
      cout<<"Sum= "<<c;
}
```

Here a and b are called **actual parameters** and p and q are called **formal parameters**. When the statement(function call) sum(a,b) is executing inside main(), then the control is transffered to the function definition and the value of a and b will be copied to p and q. In place of p and q one can use a and b and so on, but the compiler will treat them as different from that of a and b of main().

Program 7.7 : Program to find whether a number is prime or not.

//A prime number is a number that is divisible only by that number and 1.

```
void main()
{     int a;
      clrscr();
      cout<<"Enter a number ";
      cin>>a;
      prime(a);
      getch();
}
prime(int p)
{
      int i, flag=1;
      for(i=2;i<=p/2;i++)
      if((p%i)==0)
      flag=0;
      if(flag==1)
            cout<<"Number is prime";
      else
            cout<<"Number is not prime";
}
```

(c) Function with no parameter(s) and return type

In some situation, function may not accept any parameter(s) and may still return some value.

Program 7.8 : Addition of two numbers

```
void main()
{     int s;
      clrscr();
      s=sum();
      cout<<"Sum= "<<s;
      getch();
}
sum()
{
      int a=10, b=20,c;
      c=a+b;
      return c;
}
```

Output Sum=30

Here, the function sum() calculates a+b, and then the value is returned to the calling function main( ) and assigned to the variable s. Look, how interesting it is? Calculation is done by the function i.e. **sum( )**, but the value is displayed by the function main( ).

Program 7.9: Addition of two floating point numbers

```
void main()
{     float  s;
      clrscr();
      s=sum();
      cout<<"Sum= "<<s;
      getch();
}
sum()
{
      float a=10.2, b=20.3,c;
      c=a+b;
      return c;
}
```

Output Sum=30.000000(**incorrect result**)

Here, the correct result is 30.5, but it is displayed as 30.0. It is because of the absence of function declaration. Since the normal tendency of a function is to return an integer value not a float value. Though the function sum( ) calculates the correct result it returns only the integer part,

discarding the fractional portion. If any one is interested for the correct result then he/she must declare the function at the beginning of main( ) as follows-

```
float sum( );
```

(d) Function with parameter(s) and return type

Function can accept any number of parameters and can return only a single value through return statement.

7.11 MATHEMATICAL LIBRARY FUNCTIONS

C/C++ support the following mathematical standard functions

Function	Meaning
Trigonometric	
acos(x)	Arc cosine of x
asin(x)	Arc sin of x
atan(x)	Arc tangent of x
atan2(x,y)	Arc tangent of x/y
cos(x)	Cosine of x
sin(x)	Sine of x
tan(x)	Tangent of x
Hyperbolic	
cosh(x)	Hyperbolic cosine of x
sinh(x)	Hyperbolic sine of x
tanh(x)	Hyperbolic tangent of x
Other functions	
ceil(x)	x rounded up to the nearest integer
exp(x)	e to the power x (e^x)
fabs(x)	Absolute value of x
floor(x)	x rounded down to the nearest integer
fmod(x,y)	Remainder of x/y
log(x)	Natural log of x, x>0
log10(x)	Base 10 base of x, x>0
pow(x,y)	x to the power y (x^y)
sqrt(x)	Square root of x, x>=0

Note : 1. x any y should be declared as double

2. In trigonometric and hyperbolic functions , x and y are in radians

3. All the functions return a double.

7.12 INLINE FUNCTION

A function is a self contained program to perform some specific task. Function execution involves jumping of control from the calling program to called function. This is an overhead for the program and increases the time in execution. To eliminate this overhead, C++ provides a mechanism called **inline function**. An inline function is a function that is declared with the keyword **inline**. The keyword inline is a request to the compiler, not a command. It is applicable only in case of simple functions, where there is no decision or loop construct. In case of complex function the compiler may ignore this request without any error message to the programmer and treated it as a normal function. The compiler replaces the function call with the corresponding function body at the time of compilation. The general form of an inline function is described below.

```
inline returntype functionname(parameters)
{
        //body of the function
}
```

Program 7.10 : Addition of two numbers using inline function

```
#include<iostream.h>
#include<conio.h>
inline int sum(int p, int q)
{
      return p+q;
}
void main()
{
      clrscr();
      int s=sum(10,20);
      cout<<"Sum="<<s;
      getch();
}
```

Here, the function body contains only one statement i.e. **`return p+q`** and it replaces the function call **`sum(10,20)`** at the time of compilation.

7.13 FUNCTION OVERLOADING

Overloading means the same word or symbol with different meaning. C++ supports the concept of **function overloading** or **function polymorphism** i.e. different functions with same function name which performs different tasks. To differentiate one function from another one must consider the list of arguments. The function would perform different operations depending on the argument list in the function call. This fact can be explained with the help of an example. Suppose there are three students with the same name and title(say Dilip Medhi) in a class room. The class teacher has decided to call one particular student for a particular task. The basic problem for the teacher is that he cannot call a particular student sharing the same name with others because it makes confusion to the students. The teacher has to take some extra parameters such as color of shirt, father's name, address etc. Atleast one parameter must be different from one student to another to identify them. Similarly, in C++ function overloading atleast one parameter must be different from one function to another to make them unique. The overloaded functions must declare **globally**. When calling a function, the number of actual parameter and the number of formal parameter should match in type and their order. In some situation, type of parameters may not matche, then the function selection involves the following steps:

(a) First, the compiler tries to find out the match for type of actual parameter and formal parameter and their order.

(b) If match is not found, then the compiler tries to match as follows
- **char** to **int**(i.e. char in actual and int in formal)
- **float** to **double**(i.e. float in actual and double in formal)

(c) When either of them fails, the compiler tries to use the built-in conversions to the actual arguments.

Program 7.11 //overload.cpp : Program to overload sum( ) functions

```
#include<iostream.h>
#include<conio.h>
void sum();
void sum(int p, int q);//Global declaration of
void sum(int p, float q);//overloaded function sum()
void main()
{
      clrscr();
```

```
        sum();
        sum(1,2);  //Overloaded function call
        sum(5, 3);
        getch();
}
void sum()
{
        cout<<"Sum(without parameters) ="<<10+20<<"\n";
}
void sum(int p, int q)
{
        cout<<"Sum(with   int,int   parameters)
="<<p+q<<"\n";
}
void sum(int p, float q)
{
cout<<"Sum(with int,float parameters) ="<<p+q<<"\n";
}
```

Here,

void sum() ---------- > No parameter

void sum(int p, int q) -- > Two integer parameters

void sum(int p, float q) > One integer and one float parameter

7.14 DEFAULT ARGUMENTS

C/C++ functions may or may not have arguments at the time of definition. When a function is defined with certain arguments, then it should call with some values which are known as actual parameters. C++ provides some flexibility at the time of function definition. Default values can be assigned to its arguments at the time of function declaration. In such a situation, when the actual parameter is missing , the compiler substitutes the default values in that place. Default values must be initialized from **right to left**, not from left to right or at middle.

Example

Valid	Invalid
void sum(int p=10, int q=20); void sum(int p, int q=20);	void sum(int p=10, int q);

Default arguments are applicable in case of some fixed values such as rate of interest in a bank, examination fees of a college student etc.

Program 7.12 : Program to show the use of default arguments

```
#include<iostream.h>
#include<conio.h>
void main()
{
      void sum(int p=10, int q=20);
      clrscr();
      sum();
      sum(1);
      sum(5, 3);
      getch();
}

void sum(int p, int q)
{
      cout<<"Sum ="<<p+q<<"\n";
}
```

Output

Sum=30
Sum=21
Sum=8

Here, the function **sum( )** is called three times. First without actual parameters, second with one parameter and thirdly with both parameters. In the first case **sum()**, parameters p and q use default values 10 and 20, in the second case sum(1), the value of p is replaced by 1 and in the third case **sum(5, 3)**, value of p and q are replaced by the value 5 and 3.

7.15 MACROS or MACRO FUNCTIONS

C and C++ provides a mechanism which substituted the function body at the point of function call during compilation called **marcos** or **macro functions**. The advantage of macro function is that there will be no explicit function call during execution, since the function body is substituted at the point of macro call during compilation. Thereby the runtime overhead for function linking is reduced. However, it takes up more memory because the statements that define the macro function are substituted at each point

where the function is called. The pre-processor directives #define, indicates the start of a macro or macro function.

Examples

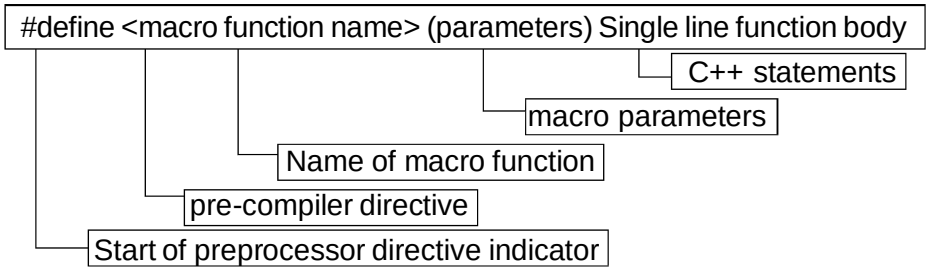
Valid macro declaration	Invalid macro declaration

`#define MAX_SIZE 10`
`#define TRUE 1`
`#define FALSE 0`
`#define begin {`
`#define end }`
	`#define MAX SIZE 10`
	`#define TRUE 1;`

Program 7.13 //readdata.cpp : Program to read 5 numbers in an array and display it

```
#include<iostream.h>
#include<conio.h>
#define SIZE 5
void main()
{
      int a[SIZE],i;
      cout<<"Enter 5 numbers one by one: ";
      for(i=0;i<=4;i++)
            cin>>a[i];
      for(i=0;i<=4;i++)
            cout<<a[i]<<" ";
      getch();
}
```

Syntax for declaring macro function



Program 7.14 //add.cpp : Program to add two numbers using macro function

```
#include<iostream.h>
#include<conio.h>
#define SUM(a,b)  (a+b)
void main()
```

```
{
        cout<<"The sum is "<<SUM(10,20);
        getch();
}
```

Here, SUM is the name of the macro function and a, b its parameter.

CHECK YOUR PROGRESS

1. State whether the following are True or False
   a. Library function and user defined functions are same
   b. Function can return only one value
   c. Function can accept only one parameter
   d. Parameter used to call a function is called actual parameter
   e. Library functions are defined in header file.
   f. sqrt() is an user defined function.
2. Write a function to accept a number and find the sum of the digits
3. write a function to accept a string and find the length of the string without using string function.

7.16 LET US SUM UP

- main() is an example of user defined function
- A function should be declared before using it.
- Function must defined.
- The header of a function definition and a function declaration must be same.
- Function declaration is to be terminated with semicolon(;)
- The parent or owner function is called a **calling function** and the child function is called a **called function**.
- A function accepts many parameters and returns only single value through **return** statement.
- Parameters used in function definition or declaration are called **formal argument** and parameters used to call a function are called **actual arguments**.
- The number of formal arguments, type of formal arguments and order of formal arguments must be same with number of actual arguments, type of actual arguments and order of actual arguments.

- A function may be called many times and from other functions using the function name.
- If a function does not return any value, then the return type of the function declaration must be **void**. Void is a keyword to indicate that the function does not return any value.
- An inline function is a function that is declared with the keyword **inline**. The keyword inline is a request to the compiler, not a command.

7.17 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education

7.18 ANSWER TO CHECK YOUR PROGRESS

Answer: 1

a. False
b. True
c. False
d. True
e. True
f. False

Answer: 2

```
#include<iostream.h>
#include<conio.h>
int digitsum(int p)
{
      int sum=0,r;
      while(p>0)
      {
            r=p%10;
            p=p/10;
            sum=sum+r;
      }
      return sum;
```

```
}
void main()
{
      int x,z;
      cout<<"Enter the number";
      cin>>x;
      z=digitsum(x);
      cout<<"Sum of digit "<<z;
      getch();
}
```

Answer 3

```
#include<iostream.h>
#include<conio.h>
int stlen(char *s)
{
      int count=0;
      while(*s!='\0')
      {
            count++;
            s++;
      }
      return count;
}
void main()
{
      char name[20];
      int c;
      cout<<"Enter a string";
      cin>>name;
      c=stlen(&name[0]);
      cout<<"The length is "<<c;
      getch();
}
```

7.19 MODEL QUESTIONS

1. Write the output of the following program segment

```
main()
{
      int i=10, m,n;
      m=check(0);
      n=check(i);
```

```
        cout<<m<<",”<<n;
}
ckeck(int p)
{
        if(p>=10)
                return(0);
        else
                return(1);
}
```

2. State whether the following statements are true or false:
   (i) The variables commonly used in C++ functions are available to all the functions in a program.
   (ii) The same variable names can be used in different functions without any conflict.
   (iii) To return the control back to the calling function we must use the keyword return.
   (iv) Every return statement in a function may return a different value.
   (v) A function can return more then one value.
3. Write C++ functions for the following:
   (i) The basic pay of an employee is input through the keyboard. His dearness allowance is 67% of basic pay, and house rent allowance is 12% of basic pay. Write a program to calculate his gross salary.
   (ii) The distance between two cities(in kms) is input through the keyboard. Write a program to convert and print this distance in meters, inches and centimeters.
   (iii) Temperature of a city in Fahrenheit degrees is input through the keyboard. Write a program to convert this temperature into Centigrade degrees.
   (iv) If a five-digit number is input through the keyboard, write a program to reverse the number.
   (vii) Write a program to check whether an inputed integer is odd or even.
   (viii) Any year is input through the keyboard. Write a program to determine whether the year is a leap year or not.
   (ix) A number is entered through the keyboard. Write a program to find the reversed number and to determine whether the original and reversed numbers are equal or not.
   (x) Any character is entered through the keyboard, write a program to determine whether the character entered is a capital letter, a small case letter, a digit or a special symbol.
   (xi) Write a program to calculate overtime pay of an employee. Over time is paid at the rate of Rs. 10.00 per hour for every hour worked avove 8 hours. Assume that employees do not work for fractional part of an hour.

UNIT 8: INTRODUCTION TO CLASSES AND OBJECT

UNIT STRUCTURE

8.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- identify the basic components of a class
- define a class and create objects
- define member function of a class
- describe array of objects
- use objects as function arguments

8.2 INTRODUCTION

So far, we have learnt that C ++ lets you create variables which can be of a range of basic data types : *int, long, double* and so on. However, the variables of the basic type do not allow you to model realworld objects (or even imaginary objects) adequately. We come to know that the basic theme of the object oriented approach is to model the real –world problems. So, object oriented programming language C++ introduces a new data type called ***class*** by which you can define your own data types as *class*. Defining the variables of a class data type is known as a class instantiation and such variables are called ***objects***. In this unit, we will introduce you how to declare a class and how to create objects of a class.We will also discuss how a member function declare inside a class or outside a class and how it can be accessed. Moreover, the techniques of passing objects as function arguments are also illustrated in this unit.

8.3 CLASSES IN C++

We are already familiar with the term encapsulation which is a fundamental feature of OOP. The encapsulation is nothing but a mechanism that binds together the data and functions into a single component, and keeps both safe from outside interference and misuse.

The data cannot be accessible by outside functions. With encapsulation data hiding can be accomplished.

In C++, the encapsulation is supported by a construct called- "*class*". First, let us think a bit about-what is an object . From the general concept, we can say that an object is something that has fixed shape or well defined boundary. In other words, an object can be a person, a place, or a thing with which the computer must deal. If you look at your surroundings some objects may correspond to real-world entities such as– student, bank account, book, cars, bags, computer, lock, watch etc. You will observe two characteristics about objects–

(i) each objects has certain attributes.

(ii) each objects has some behaviours or actions or operations associated with it.

For example, the objects 'computer' & 'car' have the following attributes and operations–

- Object : car

  Attributes: company, model, colour, & capacity

  Operation: speed ( ), average ( ), & break ( )

- Object: Computer

  Attributes: brand, price, monitor resolution, hard disk and RAM size

  Operation: Processing( ), display( ) & printing ( )

Each object will have its own identity though its attributes and operation are same, the objects will never become equal. In case of person object, for instance, two person have the same attributes like name, age and sex, but they are not equal. Objects are the basic run time entities in an object – oriented system. Thus, in C++, an object is a collection of **related variables** and **function** bound together to form a high level entity. Thus,

- The variable defines – the state of the object
- function defines – the action or operation that can be performed on the object.

Now, let us come back to the discussion of **class**.

A **class** is a grouping of objects having identical attributes and common behaviour (operations). It means all objects possessing similar attributes or properties are grouped into the same unit which is called a class. A class encloses both the data and function into a single unit as shown in the following Figure 8.1

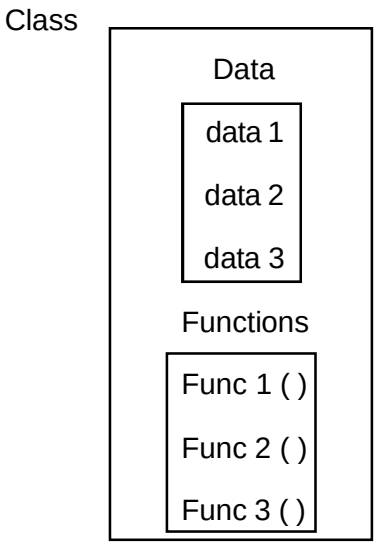
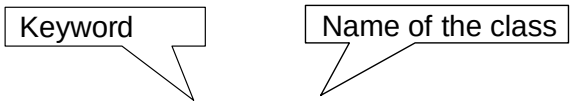


Figure 8.1 Grouping of data and function in a class

Thus, the entire group of data and code of an object can be built as a user-define data type using class. It is obvious that classes are the basic language construct in C++ for creating the user defined data types. Once a class has been declared, the programmer can create a number of objects associated with that class. Objects are nothing but the variable of the class data type. Defining variables of a class data type is known as a **class instantiation.** The syntax used to create an object of the class data type is similar to the syntax used to create an integer variable in C. In the next section, we will learn how to declare a class and an object.

8.4 CLASS DECLARATION

We have come to know that classes contain not only data but also functions. Data and functions within a class are called members of a class. The data inside a class is called a **data member** and the functions are called **member function**. The member functions define the set of operations that can be performed on the data member of a class. The syntax of a class declaration is shown below—




```
class <Class Name>
    {
    private :
        data member;
        member function;
    public:
        data member;
        member function;
    };
```

The keyword **'class'** indicates the name which follows class name, is an abstract data type. The declaration of a class is enclosed with curly braces and terminated by a semi-colon. The body of a class contains declaration of data members and member functions.

The members of a class are usually grouped in two sections i.e. *private* and *public*, which define the visibility of members.

The following declaration illustrates a class specification:

```
class employee
    {
    Private :
        char name[30];
        float age;
    Public :
        void insert_data (void);
        void show_data (void);
    };
```

A class name should be meaningful, reflecting the information it holds. Here in our example, the class 'employee' contains two data members and two member functions. The member function *insert_data( )* is used to assign value to the member variable or data member 'name' and 'age'. The member function *show_data( )* is used to display the values of the

data members. The data member of the class 'employee' cannot be accessed by any other functions that are defined outside the class. It means only the member functions of a class are permitted to access its data members.

In general, the data members are declared as private and member functions are declared as public. In our example, though, we have not specifed the data members as private; yet they are treated as private by default.

Figure 8.2 represent the class 'employee'.

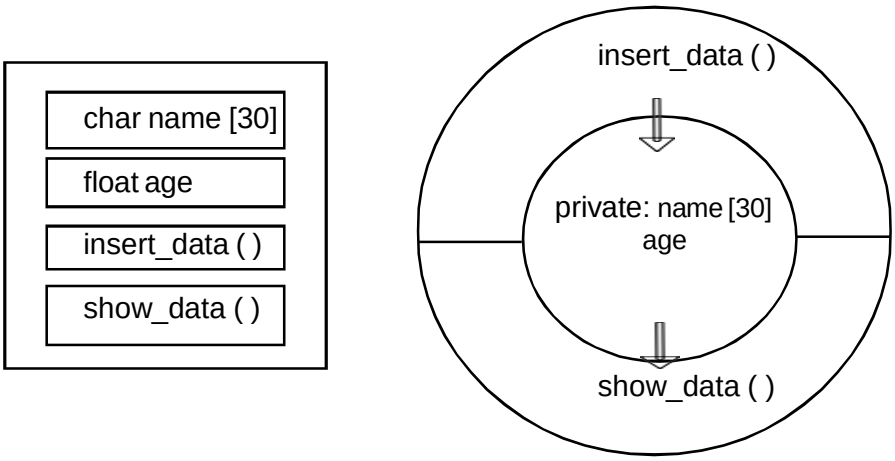


Figure 8.2 Representation of 'employee' class

8.4.1 Access Control in a Class

The members of a class are generally grouped into three sections by using the following keyword–

- private
- public
- protected

These keywords are called ***access control specifiers***. These control specifiers are written inside the class and terminated by this **':'**

symbol. All the members that follow a keyword (upto another keyword) belong to that type. If no keyword is specified, then the members are assumed as private member. We will discuss later about the protected keyword. Let us now briefly discuss about private and public keywords.

- **Private**: Private members are accessible to their own class members only. They cannot be accessed from outside the class by any member functions. The members at the beginning of class without any access specifier are private by default. Hence, writing the keyword 'private' at the beginning of a class is optional.

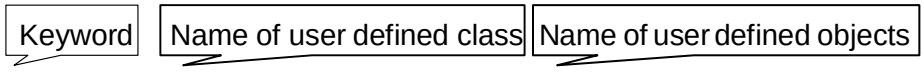
- **Public:** Public members are visible (accessible) outside the class, they should be declared in public section. All data members are not only accessible to their own members of a class but also can be accessible from anywhere in the program, either by functions that belong to the class or by those external to the class.

8.5 DECLARING OBJECTS

A class declaration only builds the structure of objects. In our example, the class *employee* does not define any objects of employee but only specifies what it will contain. Once a class has been declared, we can create variable of that type by using the class name (like any other built-in type variable). The process of creating objects (variables) of the class is called *class **instantiation***. For example–

```
int x, y, z;        // declaration of integer variables
float m, n; // declaration of float variables
employee a; // declaration of object or class variable
```

The syntax for creating objects are shown below :



class classname object name, ……

Remember, the use of the keyword *'class'* is optional at the time creating objects.

For example, ***class employee e1,***

or

employee e1;

In a single statement we can create more than one object as shown below.

employee e1, e2, e3, e4;

As in the case of structures, we can create objects by placing their names immediately just after the closing braces as follows –

```
class employee
{
    // body of the class
} e1, e2, e3;
```

This practice is rarely followed because we would like to define the objects as and when required, or at the point of their use.

Always remember that, at the time of declaration of object, necessary memory space is allocated for an object. Suppose, we have declared two objects as –

employee e1, e2;

Both ***e1*** and ***e2*** have the same data members and it is illustrated by the following figure.

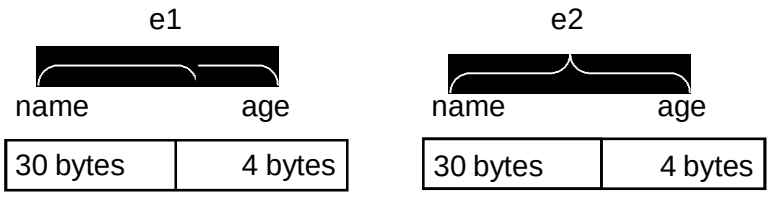


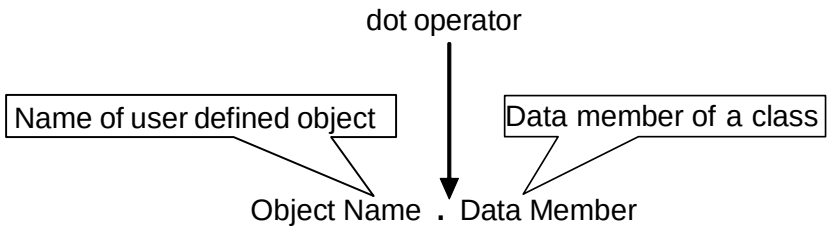
Fig. 8.3 : Allocation of memory for objects

Here, in the figure the objects **e1** and **e2** occupy the memory area. They are not initialized to anything; however the data members of each object will simply contain junk values. So, our main task is to access the data members of the object and setting them to some specific values.
An object is a conceptual entity having the following properties:

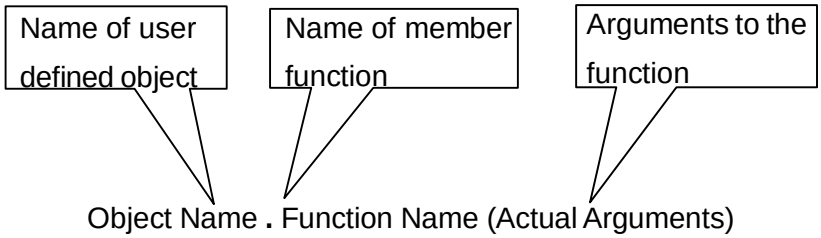
- it is individual
- it points to a thing, either physical or logical that is identifiable by the user.
- it holds data as well as operation method that handles data.
- its scope is limited to the block in which it is defined.

8.5.1 Accessing Class Members

After creating an object of a class, it is the time to access its members. This is the achieved by using the member access operator, dot (.). The syntax for accessing members (data and functions) of a class is shown below–



(a) Accessing data member of a class



(b) Accessing member function of a class

The following program demonstrates how the objects are used for accessing the class data members.

//Program 8.1

```
# include <iostream.h>
# include <string.h>
# include <conio.h>
class employee
{
     private:
     char name [30];          // name of an employee
     float age; // age of an employee
     public :   // initializing data members
     void insert_data (char * name1, float age)
     { strcpy (name, name1);
            age = age1;
     }
     void show_data ( )       //displaying the data members
     {
       cout << "Name :"<<name<<endl;
       cout << "Age :"<<age<<endl;
     }
};
void main ( )
{
     employee e1; //first object of class employee
     employee e2; //second object of class employee
     clrscr( );
     e1.insert_data("Hemanga",30);
     //e1 calls member insert_data ( )
     e2.insert_data ("Prakash",32);
     //e2 calls member insert_data ( )
     cout << "Employee Details:"<<endl;
     e1.show_data ( ); // e2 calls member show_data (
)
     e2.show_data ( ); // e2 calls member show_data (
)
       getch ( );
     }
```

OUTPUT : Employee Details:

Name : Hemanga
Age : 30
Name : Prakash
Age : 32

In the above program, in main( ) we have declared two objects through the statements

```
employee e1;      and  employee e2;
```

The statements

```
e1.insert_data ("Hemanga", 30);
e2.insert_data ("Prakash", 32);
```

initialize the data members of object e1 and e2. The object e1's data member 'name' is assigned 'Hemanga' and age is assigned 30. Similarly, the object e2's data member 'name' is assigned 'Prakash' and age is assigned 32.

The statements

```
e1.show_data ( );
e2.show_data ( );
```

call their member show_data ( ) to display the contents of data members namely, 'name' and 'age' of employee objects e1 and e2. The two objects e1 and e2 will hold different data values.

CHECK YOUR PROGRESS

1. Answer the following by selecting the appropriate option:
   a) The members of a class are by default.
      (i) Private (ii) Public
      (iii) Protected (iv) None of these
   b) The private data of any class is accessed by -
      (i) Only public member function
      (ii) Only private member function

(iii) Boths (i) & (ii) (iv) None of these

c) Encapsulation means

(i) Protecting data (ii) Allowing global access

(iii) Data hiding (iv) Both (i) & (iii)

d) The size of object is equal to

(i) Total size of member data variables

(ii) Total size of member functions

(iii) Both (i) & (ii) (iv) None of these

e) In a class, only member function can access data which is not accessible to out side. This feature is called

(i) data security (ii) data hiding

(iii) data manipulation (iv) data definition

8.6 DEFINING MEMBER FUNCTIONS

We have already come to know that a class holds both the data and functions which are called *data members and member functions*. The data member of a class must be declared within the body of the class. The member functions of a class can be defined in two places

- inside the class definition
- outside the class definition

It is obvious that a function should perform the same task, no matter where it is defined. But the syntax of a member function definition changes depending on the place of its definition, i.e. inside a class or outside a class. We will now discuss both the approaches.

8.6.1 Member Function Inside a Class

In this method, the function is defined inside the class body. When a function is defined inside a class, it is treated as an **inline** function. We will discuss inline function in the next section.

In the program 8.1 we have defined the member functions–

```
void insert_data (char * name1, float age);
```

and

void show_data ( );

inside the class 'employee'. We have seen that these function definition are similar to the normal function definition except that they are enclosed within the body of a class. They will be treated as an inline function. Remember that if a function contains loop instruction i.e. *for, while do, while* ...etc. then that function will not be treated as inline function.

8.6.2 Member Function outside a Class

In this method of defining a member function outside a class - first you will have to declare a function prototype within the body of the class and then define the function outside the body of the class.

The function defined outside the body of a class have the same syntax as normal functions i.e. they should have a function header and a function body. But, there must have a mechanism of binding the functions to the class to which they belong. This is done by using the **scope resolution operation (: :)**, in the header of the function. The scope resolution operator acts as an 'identity-label' and tells the compiler the class to which the function belongs. The general form of a member function definition is shown below–

```
ClassName
  {  .....
     .....
     ReturnType MemberFunction (arguments); // Function Prototype
     .....
     .....
  } ;
Return Type ClassName : : MemberFunction (arguments)
  {
        // body of the function
    }
```

Here, the label **ClassName : :** tells the compiler that the function **MemberFunction** is the member of class **ClassName**. The scope of the function is restricted to only the objects and other members of the class. We can modify the Program 8.1 by defining the member functions outside the class body, as shown below:

//Program 8.2

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class employee
{
    private:
      Char name [30];
      float age;
    public:
      void insert_data (char *name1, float age1);
      void show_data ( );// Member Function
};
void employee::insert_data (char *name1, float age1)
{
    strcpy (name, name1);
    age = age1;
}
void employee ::show_data ( )// Function definition
{
    cout <<"Name :"<<name<<endl;
    cout <<"Age:"<<age<<endl;
}
void main ( )
{
    employee e1, e2;
    clrscr( );
    e1 . insert_data ("Hemanga", 30);
    e2 . insert_data ("Prakash", 32);
    cout<<"EMPLOYEE DETAILS ..."<<endl;
    e1 . show_data ( );
          e2 . show_data ( );
          getch ( );
      }
```

OUTPUT : EMPLOYEE DETAILS:

Name : Hemanga
Age : 30
Name : Prakash
Age : 32

In the above definitions, the label '***employee : :***' informs the compiler that the functions insert_data ( ) and show_data ( ) are the members of the employee class.

The member functions have some special characteristics. There are–

- A program can have several different classes and they can use the same name for different member functions. The 'membership label' (ClassName : :) resolves the ambiguity of the compiler in deciding which function belong' to which class.
- Member functions can access the private data of the class, whereas non-member functions are not allowed to access. But 'friend function' can access the private data member of a class we will discuss later the friend functions.
- o Member functions of the same class can access all other members of their own class without the use of dot operator.

8.7 INLINE MEMBER FUNCTION

Let us first see what an inline function is. C++ provides a mechanism called *inline function*. When a function is declared as inline, function body is inserted in place of function call during compilation. In this mechanism, passing of control between *caller* and *callee* functions is avoided. The use of the inline function is most effective when calling function is small. If the calling function is very large, in such a case also, compiler copies the content of the inline function in the called function which reduces the program's execution speed. So, in such a case inline function should not be used.

Now, let us see how an inline function behaves with class specification. We have come to know that we can define a member function outside the class definition. The same member function can be defined as inline function by just using the qualifier inline in the header line of the function defining. In the followng, the syntax for defining inline function outside the class declaration is shown

Keyword indicates function defined outside a class body is inline

inline ReturnType ClassName : : FunctionName (arguments)

In fact, the inline mechanism reduces overhead relating to accessing the member function. It provides better efficiency and allows quick execution of functions. An inline member function is treated like a **macro** i.e. any call to this function in a program is replaced by the function itself. This is known as *inline expansion*. Inline functions are also called open subroutines because their code is replaced at the place of function call in the caller function. The normal functions are known as closed subroutines because when such functions are called, the control passes to the function. By default, all member functions defined inside the class are inline function.

We can modify the *Program 8.1* by defining the member functions as inline function as shown below:

```
    // Program 8.3
#include<iostream.h>
#include<string.h>
#include<conio.h>
class employee
{
    private:
      char name [30];
      float age;
    public :
      void insert_data (char *name1, float age1);
      void show_data ( );
```

```
} ;
inline void employee::insert_data(char *name1, float age)
{
     strcpy (name, name1);
     age = age1;
}
inline void employee : : show_data ( )
{
       cout <<"Name :"<<name <<endl;
       cout <<"Age:" <<age <<endl;
}
void main()
{
       employee e1, e2;
       clrscr( );
       e1.insert_data ("Hemanga", 30);
       e2.insert_data ("Prakash",  32);
       cout<< "Employee Details .." << endl;
       e1.show_data ( );
       e2.show _data ( );
       getch ( );
}
```

OUTPUT :

Employee Details–

Name : Hemanga
Age : 30
Name : Prakash
Age : 32

8.8 ARRAY OF OBJECTS

We know that arrays hold data of similar type. Arrays can be of any data type including user defined data type, created by using *struct, class* etc. We can create an array of variables by using class data type. Such an

array of variables of class data type is also known as an array of objects which handle a group of objects.

Let us consider the following class definition :

```
class employee
{
        private :
        char name [30]
        float age ;
        Public :
        void insert_data (char* name1, float age1);
        void show_data ( );
};
```

Here, the identifier '*employee*' is a user defined data type and can be used to create objects that relate to different categories of employees. For example, the following definition will creates an array of objects of '*employee*' class–

```
employee consultant [30]; // array of cosultant
employee clerk [15];      // array of clerk
employee lecturer [20];   // array of lecturer
```

The array consultant contain 30 objects (consultant), namely, consultant [0], consultant [1],..... consultant [29], of type employee class. Similarly, clerk array contains 15 objects and lecturer array contains 20 objects.

We know that as array elements occupy continuous memory locations like the same way as array of objects occupy contiguous memory locations as shown in the fig. 8.4

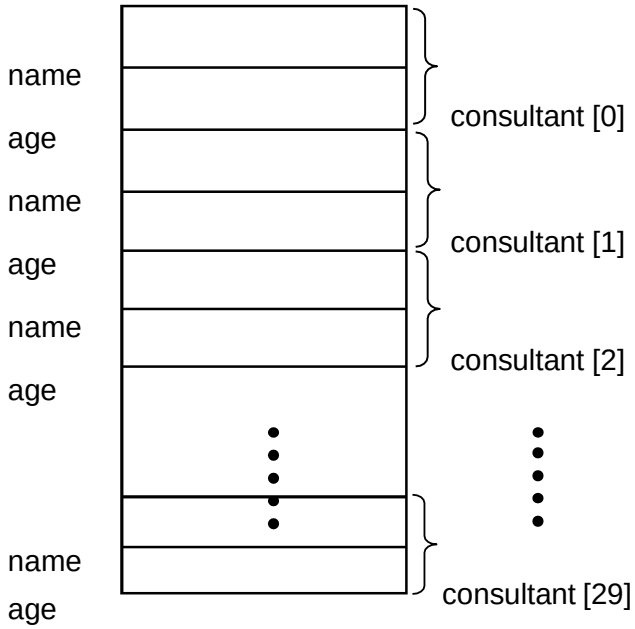


Fig.8.4 Stoage of data items in '*consultant'* array of objects.

By using index an individual element of an array of objects can be referred i.e. ***consultant [15], consultant [9] .....etc***. By using the dot operator [**.**] we can access any member of an object. For example

consultant [30] . show_data ( )

will display the data of 30th consultant.

We can rewrite the Program 8.1 by using array of objects as shown below:

***//* Program 8.4**

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class employee
{
    private:
    char name [30];
    float age;
    public:
    void insert_data (char *name1, float age)
```

```
        {
              strcpy (name, name1);
              age = age1;
          }
          void show_data ( )
          {
              cout<<"Name:"<<name<<endl;
              cout<<"Age:"<<age<<endl;
          }
};
void main ( )
{
    int i, age, count;
    char name [30], tag;
    employee consultant [30];
    clrscr ( );
    count = 0;
    for (i=0; i<30; i++)
    {
      cout<<"Enter Data For Employee (Y/N):";
      cin >> tag;
      if (tag = = 'y' || tag = = 'Y')
      {
          cout <<"\n Enter Name of Employee:";
          cin >> name;
          cout << "Age:";
          cin >> age;
          consultant[i].insert_data(name, age);
          count ++;
        }
      else
          break;
    }
    cout <<"\n\n Employee Details ......\n" ;
```

```
    for (i=0; i < count; i ++)
    consultant[i].show_data();
            getch ( );
    }
```

OUTPUT :

```
Enter Data for Employee (Y/N) : y
Enter Name of Employee        : Prakash
Age                           : 30

Enter Data for Employee (Y/N) : y
Enter Name of Employee        : Hemanga
Age : 28
Enter Data for Employee (Y/N) : n

Employee Details...
Name                          : Prakash
Age                           : 30
Name                          : Hemanga
Age                           : 28
```

8.9 OBJECTS AS FUNCTION ARGUMENTS

Objects can be passed as an argument to a function. There are three ways of passing objects as function arguments:

- a copy of the entire object is passed to the function, which is also called ***pass-by-value***
- only the address of the object is sent implicitly to the function, which is also called – ***pass by-reference***.
- the address of the object is sent explicitly to the function, which is also called – ***pass-by-pointer***.

8.9.1 Pass-by-value

In this technique, a copy of the object is passed to the called function (callee) from the calling function (caller). Since a copy of the object is passed so any changes made to the object inside the called function do

not affect the object used to call the function

The following program demonstrates the use of objects as function arguments in pass-by-value mechanism.

//Program 8.5

```
#include<iostream.h>
#include<conio.h>
class age
{
    private:
    int birthyr ;
    int presentyr ;
    int year ;
    public:
    void getdata ( );
    void period (age);
};
void age : : getdata ( )
{
    cout<<" \ n Year of Birth:";
    cin >> birthyr ;
    cout << "Current year:" ;
    cin >> presentyr ;
}
void age : : period (age x1)
{
    year = x1 . presentyr - x1 . birthyr ;
    cout << "Your Present Age :" <<year<<"Years" ;
}
void main ( ){
    clrscr ( );
    age a1 ;
    a1 . getdata ( );
    a1 . period (a1) ;
    getch( );
```

```
}
```

OUTPUT: Year of Birth : 1990
Current Year : 2002
Your Present Age : 19 years

In the above program, the class age has three data member. The function *getdata ( )* reads integers through keyboard. The function *period ( )* calculates the difference between the two integers. In function main ( ), a1 is an object to the class age. The object a1 calls the function *getdata ( )*. The same object a1 is passed to the function *period ( )*, which calculates the difference between the two integers. Thus, an object can be passed to a function.

8.9.2 Pass-by-Reference

In this technique, only the address of the object is sent to the function. When an address of the object is passed, the address acts as reference pointer to the actual object in the calling funciton. Therefore, any change made to the objects inside the called function will reflect in the actual object in the calling funciton. We can modify the program 8.5 by using the pass by reference mechanism

```
// Program 8.6
#include<iostream.h>
#include<conio.h>
class age
  {
      private:
          int birthyr ;
          int presentyr ;
          int year ;
      public:
          void getdata ( );
          void period (age);
```

```
      };
void age : : getdata ( )
{
     cout<<" Year of Birth:";
     cin >> birthyr ;
     cout << "Current year:" ;
     cin >> presentyr ;
}
void age : : period (age & x1)
{
     x1. year = x1 . presentyr - x1 . birthyr ;
     cout << "Your Present Age :" <<year<<"Years" ;
}
void main ( )
{
     clrscr ( );
     age a1 ;
     a1 . getdata ( );
     a1 . period (a1) ;
     getch ( );
}
```

RUN :

Year of Birth	:	1990
Current Year	:	2009
Your Present Age	:	19 years

8.9.3 Pass-by-Pointer

In this mechanism also, the address of the object is passed explicitly to the called function from the calling function. The program 8.6 is modified by using the mechanism pass-by-pointer as follows:

//Program 8.7

```
#include<iostream.h>
#include<conio.h>
class age
{
       private:
            int birthyr ;
            int presentyr ;
            int year ;
       public:
            void getdata ( );
            void period (age * );
};
void age : : getdata ( )
{
       cout<<" \n Year of Birth:";
       cin >> birthyr ;
       cout << "current year:" ;
       cin >> presentyr ;
}
void age : : period (age * x1)
{
     year = x1→presentyr - x1→birthyr ;
     cout << "Your Present Age :" <<year<<"Years" ;
}
void main ( )
{
     clrscr ( );
     age a1 ;
     a1 . getdata ( );
     a1 . period (&a1) ;
     getch ( );
}
```

OUTPUT:
```
Year of Birth      : 1990
Current Year       : 2009
Your Present Age   : 19 years
```

In *Program 8.6* and *Program 8.7* we need to keep an eye on the symbols '.', '→', '*', '&', and the statements(bold lines) where we have appropriately used them.

8.10 FRIEND FUNCITON AND FRIEND CLASS

We have already discussed the fact that the private members of a class cannot be accessible from the outside of the class. Only the member functions of that class have permission for accessing the private members. This policy enforces the encaptulation and data hiding techniques.

Let us think of a situation where a user need a funciton to operate on objects of two different classes. It means that the function will be allowed to access the private data of both the classes. In C++, this situation is over come by using the concept of friend function. It permits a friend function to access the private members of different classes.

The declaration of a friend function must be prefixed by the keyword "**friend**". In the following class is shown a declaration of a friend function.

```
class test
{
        private:
                ---------
                ---------
        public :
                ---------
                ---------
                friend void sum( ) ;
};
```

The function can be defined anywhere in the program similar to any normal C++ function. The function definition does not use either the keyword friend or the scope operator **': :'**. The functions that are declared with the keyword 'friend' are called friend functions. A friend function can be a friend to a multiple classes. The friend function have the following properties :

- There is no scope restriction for the friend function; hence they can be called directly without using objects.
- Unlike member functions of class, friend function cannot access the members directly. On the other hand, it uses object and dot operator to access the private and public member variables of the class.
- Use of friend function is rarely done, because it violates the rule of encapsulation and data hiding.
- The function can be declared in public or private sections without changing its meaning.

The following program demonstrates the use of friend function:

// Program 8.8

```
#include<iostream.h>
#include<conio.h>
class first ;     /*forward declaration like function Prototype*/
class second
{
     int x ;
     public :
     void get value ( )
     {
       cout << "\n Enter a number :" ;
       cin >> x ;
     }
     friend void sum(second, first);//declaration of friend
dunction
} ;
class first
{
```

```
            int y ;
            public :
            void getvalue ( )
            {
                  cout << "\n Enter a number:" ;
                  cin >> y ;
            }
            friend void sum (second, first) ;
} ;
void sum (second m, first n)
{
     cout << "\n Sum of two numbers :" << n.y + m.x
}
void main( )
{
     clrscr ( );
     first a ;
     second b ;
     a.get value ( ) ;
     b.get value ( ) ;
     sum(b,a);//funciton is called like a general function in C++
}
```

OUTPUT:

```
Enter a number        :  9
Enter a number        :  12
 Sum of two numbers :  21
```

In the above program each of the two classes 'first' and 'second' has a member function named getvalue ( ) and one private data member. Notice that, the function sum ( ) is declared as friend function in both the class. Hence, this function has the ability to access the members of both the classes. Using sum ( ) function, addition of integers is calculated and displayed.

It is possible to declare all the member functions of a class as the friend functions of another class. When all the functions need to access another class in such a situation we can declare an entire class as ***friend class***. Always remember that friendship is not exchangeable. Its meaning is that

- declaring *class A* to be a friend of *class B* does not mean that *class B* is also a friend of *class A*. The declaration of a friend class is as follows:

```
class second
  {
       -------
       -------
       friend class first;
  };   /* all member functions of class first are friends to
          class second */
```

The following program demonstrates the use of friend class :

//Program 8.9

```
#include<iostream.h>
#include<conio.h>
class smallvalue;
class value
{
     int a;
     int b;
     public:
     value (int i, int j)   // declaration of constructor with
arguments
     {
       a = i;
       b = j;
     }
 friend class smallvalue;
};
class smallvalue
{
     public:
     int minimum(value x)
     {
       return x.a < x.b ? x.a : x.b;
     }
```

```
  };
void main ( )
{
        value x (15, 25);
        clrscr ( ) ;
        smallvalue y;
        cout << y. minimum(x);
        getch ( );
}
```

In the above program we have used the constructor with arguments. The concept of constructor is illustrated in unit 9 'Constructors and Destructors'.

CHECK YOUR PROGRESS

2. Fill in the blanks of the following :
   (i) Member functions defined within the class definition are implicity____.
   (ii) When only the address of the object is sent explicity, it is called _____
   (iii) _____function can access the private data members of a class.
3. State whether the following statements are true or false:
   (a) To reference an object using a pointer to object, uses the<> operator.
   (b) In the prototype void sum (int &) arguments are passed by reference.
   (c) If class A is a friend class of class B then a member function of class B can access the data members of class A.

8.11STATIC DATA MEMBER AND MEMBER FUNCTON

After studying public and private members, let us study about the static members of a class. Recall what we have learnt from C Programming:

(i) A variable can be declared as static inside a function or outside main().

(ii) Static variables value do not disappear when function is no longer active; their last updated value always persists. That is, when the control comes back to the same function again the static variables have the same value as they leave at the last time.

in C++ also. However, C++ has objects. Hence, the meaning of static with respect to member variables of an object is different.

We have already gained the idea that each object has its separate set of data member variable in memory. The member functions are created only once and all object share the function. No separate copy of the function of each object is created in the memory like data member variables. Figure shows the accessing of member function by objects.

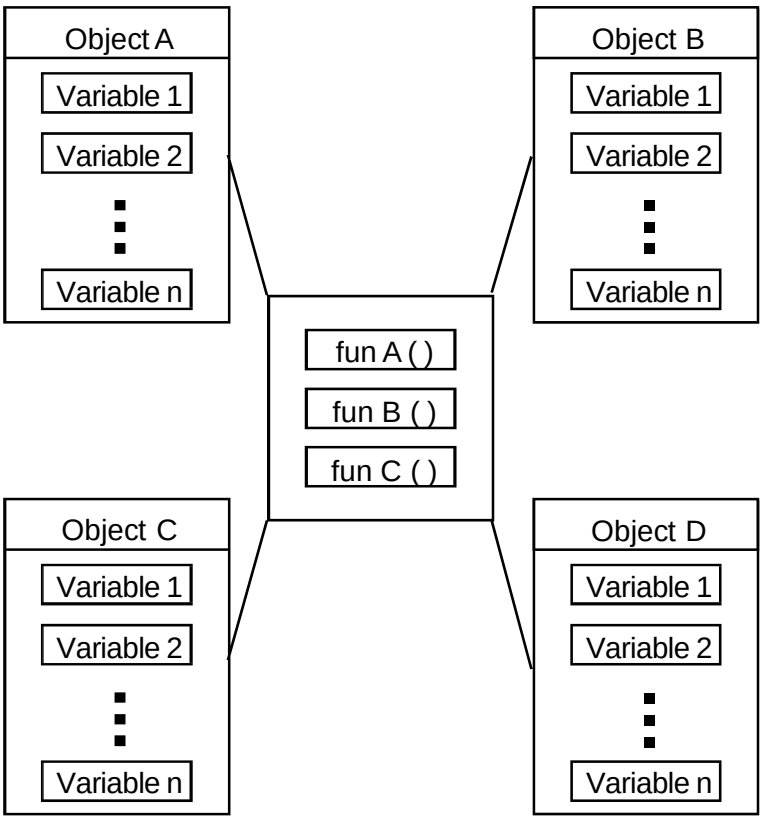


Fig. 8.5 Data members and member functions in memory

In C++, it is possible to create common member variables like function using the **static** keyword. Once a data member variable is declared as static, only one copy of the member is created for the whole class and all objects of the class will share that variable.

Always remember–

- A static variable preserves the value of a variable.
- When a variable is declared as static it is initialized to zero.
- A static data member or member function is only recognized inside the scope of the present class.
- A static variable can be a public or private.

The syntax for declaring static data member or member function within a class is shown below:

static <variable name> ;

static <function name> ;

When you declare a static data member within a class, you are not defining it i.e. you are not allocating storage for it. Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the static variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated.

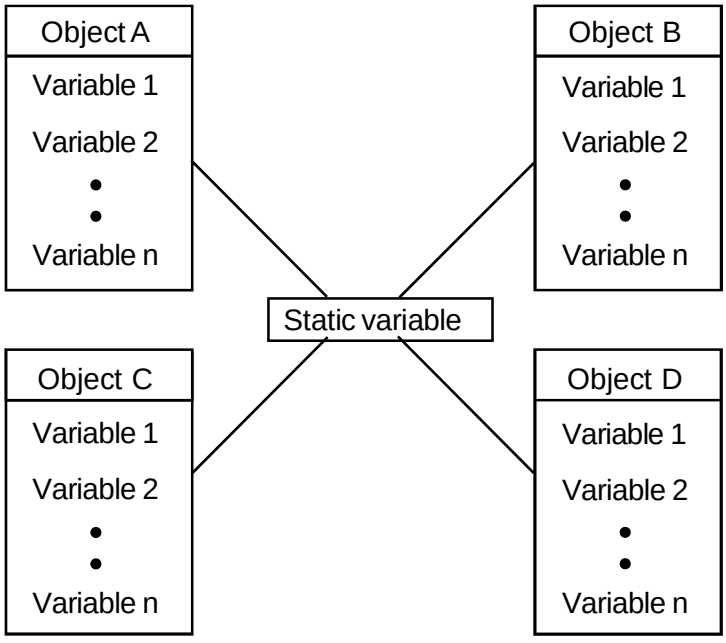


Fig. 8.6 Static member in memory

The declaration of static member is shown below :

```
class number
{
        static int C;
        public:
                ----------
                ----------
};
int number : : C = 0  // initializaiton of static member variable
```

The following program demonstrates the use of static data member in a class–

// Program 8.10

```
#include<stdio.h>
#include<conio.h>
class number
{
        static int C;
        public:
        void count ( )
        {
                C ++;
                cout << "\n C =" << C;
        }
};
int number : : C = 0;
void main( )
 {
        number a, b, c;
        clrscr ( ) ;
        a.count ( );
        b.count ( );
        c.count ( );
        getch ( );
 }
```

OUTPUT: c = 1
c = 2
c = 3

In the above program, the class number has one static data variable C. The *count( )* is a member funciton, increment value of static member variable C by 1 when called. The statement *int number : : C = 0* initiatize the static member with 0. It is possible to initialize the static data members with other values. In the function *main( )*, a, b and c are three objects of *class number*. Each object calls the funciton *count( )*. At each call to the function *count( )* the variable C gets incremented and the count statement displays the value of variable C. The objects a, b and c share the same copy of static data member C.

STATIC MEMBER FUNCTION

In C++, like member variables, funcitons can also be declared as static. When a function is defined as static, it can access only static member variable and functions of the same class. The non-static members are not available to these functions. The static member function declared in public section can be invoked using its class name without using its objects. The static keyword makes the funciton free from the individual object of the class and its scope is global in the class without creating any side effect for other part of the program.

The following points should be remembered while declaring static function:

a) Just one copy of the static member is created in the memory for the entire class. All objects of the class share the same copy of static member.

b) Static member functions can access only static data member, or funcitons.

c) Static member functions can be invoked using class name.

d) It is also possible to invoke static member funcitons using objects.

e) When one of the objects changes the value of data member variables, the effect is visible to all the objects of the class.

The following program demonstrates the use of the static member function in a class.

//Program 8.11

```
#include<iostream.h>
#include<conio.h>
class number
{
    private:
    static int X;
    public:
    static void count ( ) {X++; }
    static void display ( )
    {
          cout << "\n value of X =" << X;
    }
};
int number : : X = 0;
void main ( )
{
clrscr ( ) ;
number::display( );  //invokes display function
number::count( );     //invokes count function
number::count( );     //invokes display function
number::display( );  //invokes display function
getch ( );
}
```

OUTPUT: Value of X : 0
Value of X : 2

In the above program, the member variable X and functions *count ( ) & display ( )* of *class number* are static. The function *count ( )* when called, increases the value of static variable X. The function *display ( )* prints the current value of the variable X. The static functions can be called using class name and scope resolution operator as shown in the program–

number : : count ( );
number : : display ( );

8.12 LET US SUM UP

- Classes are the basic language construct in C++ for creating the user defined data types.
- A class contains member variable or data members and member functions.
- The members of a class are grouped into two sections, namely, private and public.
- Defining variables of a class data type is known as a class instantiation and such variables are called objects.
- Using the member accessed operator, dot(**.**), the class members can be access by the objects.
- The member function can be defined as a) private or public b) inside the class or outside the class.
- The scope resolution operator (**::**) is used, when a member function is defined outside the class body.
- Inline member function is treated like a macro, when a function is declared as inline, function body is inserted in place of function call during compilation.
- We can create an array of variables by using the class data type, then these variables are called array of objects, which occupies contiguous memory locations inmemory.
- There are three methods of passing objects to function, namely, pass-by-value, pass-by-reference, and pass-by-pointer.
- The function that are declared with the keyword ***friend*** are called friend function. A function can be a friend to multiple classes.
- static is the keyword used to preserve value of a variable. When a variable is declared as static, it is initialized to zero. A static function or data element is only recognized inside the scope of the present class.
- When a function is defined as static, it can access only static member variables and functions of the same class. The static member functions are called using its class name without using its objects.

8.13 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education

8.14 ANSWERS TO CHECK YOUR PROGRESS

1. a) i. b) iii. c) iv. d) i. e) ii
2. i) inline, ii) pass-by-pointer, iii) friend
3. a) False, b) True, c) False

8.15 MODEL QUESTIONS

1. What is a class ? How does it accomplish data hiding ?
2. What is an object ? How is an object created ?
3. How is a member function of a class defined or declared ?
4. Explain the use of *private* and *public* keywords. How are they different from each other ?
5. What is the significance of scope resolution operator **::** ?
6. When will you make a function inline and why ?
7. Explain the different methods of passing objects to functions.
8. What is a friend function and a friend class ? Explain with example.

****

UNIT 9 : CONSTRUCTORS AND DESTRUCTORS

UNIT STRUCTURE

9.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- define and use constructors in programming
- learn about default constructor
- learn to use parameterized contructors
- define and use copy constructor
- learn about destructors
- describe the way of declaring a destrcutor
- initialize object dynamically

9.2 INTRODUCTION

In the previous unit we have studied data members and member functions of class. We have seen so far a few examples of classes being implemented. In all cases, we defined a separate member function for reading input values for data members. With the use of object, member

functions are invoked and data members are initialized. These functions cannot be used to initialize the data members at the time of creation of object. C++ provides a pair of special member functions called *constructor* and *destructor*. Constructor enables an object to initialize itself when it is created and destructor destroys the object when it is no longer required.

In this unit we will discuss the constructor and destructor. Different types of contructor and their implementation will also be discussed in this unit.

9.3 CONSTRUCTORS

A constructor is a special member function in a class that is called when a new object of the class is created. It, therefore, provides the opportunity to initialize objects as they are created and to ensure that data members only contain valid values. ***It is a member function whose task is to initialize the object of its class*** and allocate the required resources such as memory. It is distinct from other members of the class because it has the same name as its class.

A constructor can be declared and defined with the following syntax:

```
class classname
{                          Private members
      ..... ;
      public :                    Constructor declaration
            classname( );
};                                Constructor definition
classname :: classname( )
{
            //Body of constructor
}
```

Constructors have some special constraints or rules that must be followed while defining them. These are as follows:

- Constructor is executed automatically whenever the class is instantiated i.e., object of the class is created.
- It always has the same name as the class in which it is defined
- It cannot have any return type, not even *void*.
- It is normally used to initialize the data members of a class.
- It is also used to allocate resources like memory to the dynamic data members of a class.
- It is normally declared in the public access within the class.

For example, let us declare a class with class name 'circle'. If we use constructor, then the constructor name must also be 'circle'. The program can be written as follows:

/***Program 9.1:*** Program to demostrate constructor while calculating the area of a circle*/

```
#include<iostream.h>
#include<conio.h>
#define PI 3.1415
class circle
{
    private:
    float radius;
    public:
    circle();           //constructor declaration
    void area()         // member function
    {
        cout<<"\nArea of the circle is " ;
        cout<<PI*radius*radius<<" sq.units\n";
    }
};
circle :: circle()   //constructor definition
{
    cout<<"\nEnter radius of the circle: ";
    cin>>radius;
}
```

```
int main()
{
    clrscr();
    circle c;//constructor invoked automatically
    c.area();
    getch();
    return 0;
}
```

The output of the above program will be :

Enter radius of the circle: 5

Area of the circle is 78.5375 sq. units

In the above program, the constructor **circle( )** takes the value of radius from the keyboard. Although the data member is private, the constructor could initialize them. The statement **circle c** declares a variable **c** as object of (type) class **circle**. It also calls the constructor implicitly.

The Default Constructors

The constructor which takes no arguments is called the ***default constructor***. The following code fragment shows the syntax of a default constructor.

```
class class_name
{
    private:
    data members;
    public:
    class_name( );        //default constructor
};
class_name : : class_name( )
{
    /* definition of constructor without any arguments and body */
}
```

If no constructor is defined for a class, then the compiler supplies the default constructor.

Instantiation of Object

Instantiating an object is what allows us to actually use objects in our program. We can write hundreds and hundreds of class declarations, but none of that code will be used until we create an instance of an object. A class declaration is merely a template for what an object should look like. When we instantiate an object, C++ follows the class declaration as if it were a blueprint for how to create an instance of that object.

CHECK YOUR PROGRESS

1. State whether the following statements are True (T) or False (F):
   (i) The constructor is not a member function.
   (ii) It is wrong to specify a return type for a constructor.
   (iii) It is possible to define a class which has no constructor at all.
   (iv) The name of a constructor need not be same as that of the class to which it belongs.
   (v) A class may have two default constructors.
2. Choose the appropriate option:
   (i) A function that is automatically called when an object is created is known as :
   (a) constructor (c) delete
   (b) destructor (d) free( ) function
   (ii) Constructor with no parameter is known as
   (a) stand-alone constructor (c) copy constructor
   (b) default constructor (d) none of these
   (iii) For a class namely *Shape*, the contructor will be like
   (a) void Shape() { .....} (b) shape() {.....}
   (c) Constructor Shape(){.....} (d) Shape() {.....}

3. Write a program in C++ that displays the factorial of a given number using a constructor member function.
4. Write a program in C++ to find the area of a rectangle of length and breadth 6.0 by using a class "rectangle". The program shouldalso contain a constructor along with other member function.

9.3.1 Parameterized Constructors

The constructor that can take arguments are called ***parameterized constructor.*** The arguments can be separated by commas and they can be specified within braces similar to the argument list in function.

When a constructor has been parameterized, the object declaration without parameter may not work. In case of parameterized constructor, we must provide the appropriate arguments to the constructor when an object is declared. This can be done in two ways:

- By calling the constructor ***explicitly***.
- By calling the constructor ***implicitly***.

The implicit call method is sometimes known as shorthand method as it is shorter and is easy to implement.

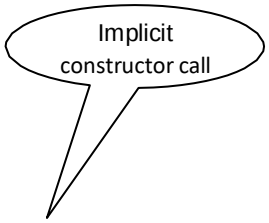
Let us consider the following example to demonstrate how parameterized constructor works.

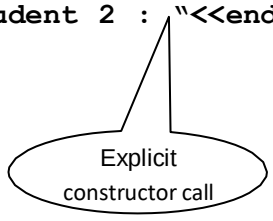
// **Program 9.2:** Program creating parameterized constructor

```
#include<iostream.h>
#include<conio.h>
class student
{
    private:
    int roll,age, marks;
    public:
    student(int  r,int  m,int  a);//parameterized
```

constructor

```
    void display( )
    {
            cout<<"\nRoll number :" <<roll <<endl;
            cout<<"Total marks : <<marks<<endl;
            cout<<"Age:"<<age<<endl;
        }
};  //end of class declaration
student::student(int r, int m, int a) //constructor definition
{
    roll = r;
    marks =m;
    age=a;
}
int main( )
{
    student manoj(5,430,16);   //object creation
    cout<<"\nData of student 1 : "<<endl;
    manoj.display();
    student rahul = student(6,380,15);//object creation
    cout<<"\n\nData of student 2 : "<<endl;
    rahul.display();
    getch();
    return 0;
}
```





The output of the above program will be like this:

```
Data of student 1 :
Roll number     : 5
Total Marks     : 430
Age             : 16
Data of student 2 :
Roll number     : 6
Total Marks     : 380
```

In the above program, the statement **student(int r, int m, int a,);** is a parameterized constructor with two arguments. In the main( ) function, we see that there are two objects, **manoj** and **rahul** of class type student. The object manoj is initialized with the values 5,430 and16 with the *implicit call* statement **student manoj(5, 430,16);**. The object rahul is initialized with the values 6,380 and 15 with the *explicit call* statement **student rahul = student(6, 380,15);** This statement creates a student object rahul and passes the values 6, 380 and 15 to it.

Constructors with Default Arguments

It is possible to have a constructor with arguments having default values. It means that if we have a parameterized constructor with n parameters, then we can invoke it with less than n parameters specified in the call.

It is useful when most of the objects to be created are likely to have the same value for some data members. We need not specify that in every invocation. For example, suppose we want to record the data of class x (ten) students. For this, we can consider the same class **student** as shown in *Program 8.2.* Most of the students are of age 16. Hence, their birth year is likely to be the same. Only a few of them may have a different year of birth. For this we can use the statement

student(int r, int m, int a=16);

where the third parameter is given the default value. When we write the following statement

student s(1,360);

it assigns the values **1** and **360** to the argument **r** and **m** respectively and **16** to the argument **a** by default. Again, if we write

student t(2,400,15);

it assigns **2**, **400** and **15** to **r**, **m** and **a** respectively. Thus, the actual arguments, when specified, overrides the default values.

EXERCISE

Q. Write a C++ program to create a class “EMPLOYEE” to initialize EMP_ID, DESIGNATION and BASIC_PAY using a constructor and display data for three employees.

9.3.2 Copy Constructors

Object of the same class cannot be passed as argument to constructor to that class by value method.

```
class student
{
     private:
          .......          This is not a valid
     public:               declaration
          student(student s);
};
```

But it is possible to pass an object of the same class as argument to a constructor by reference method. Such type of constructor, i.e., a constructor having parameter which is a reference to object of the same class is called a ***copy constructor***. Thus, the following declaration is a valid declaration.

```
     class student
     {
          private:
                              int roll, marks, age;
//data members
          public:
                              student(student &s);
//copy constructor
     };                          Reference to an object
                                 of class student
```

Copy constructor is also called one argument constructor as it takes only one argument. The main use of copy constructor is to initialize the objects while in creation, also used to copy an object. This constructor allows the programmer to create a new object from an existing one by initialization. Let us demostrate copy constructor with the following program :.

//***Program 9.3:*** Program to demonstrate copy constructor

```
#include<iostream.h>
#include<conio.h>
class student
{
    private:
        int roll, marks, age;
    public:
        student(int r,int m,int a)
// parameterized constructor
        {
            rol l = r;
            marks = m;
            age = a;
        }
        student(student &s) //copy constructor
        {
            roll = s.roll;
            marks = s.marks;
            age=s.age;
        }
        void display( )
        {
            cout<<"Roll number :" <<roll <<endl;
            cout<<"Total marks :"<<marks<<endl;
            cout<<"Age:"<<age<<endl;
        }
};
```

```
int main( )
{
     clrscr();
     student t(3,350,17);// or student k(t);
     student k = t;// invokes copy constructor
     cout<<"\nData of student t:"<<endl;
     t.display();
     cout<<"\n\nData of student k:"<<endl;
     k.display();
     getch();
     return 0;
}
```

The outout of the above program will be like this:

```
Data of student t :
 Roll number  :  3
  Total marks  :  350
          Age  :  17
Data of student k :
 Roll number  :  3
  Total marks  :  350
          Age  :  17
```

The statements

student t(3,350,17); //invokes parameterized constructor

student k = t; //invokes copy constructor

initialize one object **k** with another object **t**. The data members of **t** are copied member by member into **k**. When we see the output of the program we observe that the data of both the objects t and k are same.

9.4 OVERLOADING OF CONTRUCTORS

A class may contain multiple constructors i.e., a class can have more than one constructor with the same name. The constructors are then recognized depending on the arguments passed. It may differ in terms of arguments, or data types of their arguments, or both. This is called ***overloading of constructors*** or ***constructor overloading***.

Program 9.3 is also an example of constructor overloading as it has two constructors: one is parameterized and the other is a copy constructor. Let us take a suitable example to demostrate overloading of constructors.

//**Program 9.4:** Demonstration of constructor overloading

```
#include<iostream.h>
#include<math.h>
#include<conio.h>
class complex
{
    private:
    float real,imag;
    public:
    complex( )//constructor with no argument
    {
        real = imag = 0.0;
    }
    complex(float r, float i)
    //constructor with two arguments
    {
        real = r;
        imag = i;
    }
    complex(complex &c)   //copy constructor
    {
        real = c.real;
        imag = c.imag;
```

```
    }
    complex addition(complex d);
   /*member function returning object and taking
               object as argument */
    void display( )  //Display member function
    {
        cout<<real;
        if(imag < 0)
            cout<<"-i";
        else
            cout<<"+i";
        cout<<fabs(imag);/*fabs calcute the absolute
value
                                of a  floating point
number */
        }
    };
    complex complex::addition(complex d)
    {
        complex temp;   //temporary object of type
complex class
        temp.real=real+d.real; //real parts added
        temp.imag=imag+d.imag;    //imaginary parts
added
        return(temp);
    }
int main( )
{
    clrscr( );
    complex x1,x4; //invokes default constructor
    cout<<"\nThe complex numbers in a+ib form :\n\n";
    cout<<"First complex number: ";
    x1.display();
    complex x2(1.5,5.3);//invokes parameterized construc-
tor
```

```
    cout<<"\nSecond complex number: ";
    x2.display();
    complex x3(2.4,1.9);
    cout<<"\nThird complex number: ";
    x3.display();
    cout<<"\nAddition of 2nd and 3rd complex number: ";
    x4=x2.addition(x3);  //function call
    x4.display( );
    cout<<"\nThe result is copied to another object:
";
        complex x5(x4); //invokes copy constructor
        x5.display();
        getch();
        return 0;
    }
```

The output of the above program will be:

The complex numbers in a+ib form:

First complex number: 0+i0

Second complex number: 1.5+i5.3

Third complesx number: 2.4+i1.9

Addition of second and third complex number: 3.9+i7.2

The result is copied to another object: 3.9+i7.2

We have the following three constructors

```
complex( );
complex(float r, float i) ;
```

and `complex(complex &c);`

The above program indicates overloading of constructors. These constructors are invoked during the creation of an object depending on the number and types of arguments passed. The default constructor **complex( );** initializes the data members *real* and *imag* to 0.0. In main( ), the statement **complex x2 (1.5,5.3);** passes two parameters to the constructor explicitly with the help of the parameterized constructor **complex (float r, float i);** With the help of copy constructor **complex (complex &c);** data members of one object is copied member by member into another.

9.5 DESTRUCTORS

Like constructors, destructors are also special member functions used in C++ programming language. Destructors have the opposite function of a constructor. The main function of destructors is to free memory and to release resources. Destructors take the same name as that of the class name preceded by a ***tilde ~***. A destructor takes no arguments and has no return type.The general syntax of a destructor is as follows:

```
class classname
{
     private :            //Private members
          ........ ;
     public :
          ~classname( );        //Destructor declaration
};
       classname :: ~classname( )//Destructor definition
{
          //Body of destructor
}
```

In the above, the symbol tilde ~ represents a destructor which precedes the name of the class. Like the default constructor, the compiler always creates a default destructor if we donot create one. Similar to constructors, a destructor must be declared in the public section of a class. Destructor cannot be *overloaded* i.e., a class cannot have more than one destructor.

/* **Program 9.5 :** Program demonstrating destructor */

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class student
{
     private:
```

```
        int age;
        char name[25];
    public:
        student(int a, char n[25])
        {
            age=a;
            strcpy(name,n);
        }
        void show()
        {
            cout<<"\nThe name of the student is:
"<<name;
            cout<<"\nHis age is: "<<age;
        }
        ~student() //destructor defined
        {
            cout<<"\nObject Destroyed";
            }
    };
    int main()
    {
        student s1(21,"Rahul");
        student s2(23,"Dipankar");
        clrscr();
        s1.show();
        s2.show();
        return 0;
    }
```

To see the execution of destructor, we have to press Alt+F5 after compiling and running the program. The ouput will be like this:

The name of the student is : Rahul
His age is : 21

The name of the student is : Dipankar
His age is : 23

Object Destroyed
Object Destroyed

The following points should be kept in mind while defining and writing the syntax for the destructor

- A destructor must be declared with the same name as that of the class to which it belongs. But destructor name should be preceeded by a tilde (~).
- A destructor should be declared with no return type.
- A destructor must have public access in the class declaration.

9.6 DYNAMIC INITIALIZATION OF OBJECTS

We can dynamically initialize objects through constructors. Object's data members can be initialized dynamically at run time even after their creation.

//Program 9.6 Demonstration of dynamic initialization of object

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
class number
{
     public:
     int num;
     number(int n)
     {
          num=n;
     }
     int sum( )
     {
          num=num+5;
          return(num);
```

```
    }
};
int main( )
{
    number obj1(1);  // parameterized constructor invoked
    number obj2(2);
    clrscr();
    cout<<"\nValue of object 1 and object 2 are : ";
    cout<<obj1.num<<"\t"<<obj2.num;
    number obj3(obj1.sum( ));//dynamic initialization of
object
    cout<<"\nValue of object 1 after calling sum() :
"<<obj1.num;
    cout<<"\nValue of object 3 is :"<<obj3.num;
    getch();
    return 0;
}
```

The output of the above program will be :

Value of object 1 and object 2 are : 1 2

Value of object 1 after calling sum() : 6

Value of object 3 is : 6

The statements **number obj1(1);** and **number obj2(2);** will initialize obj1 and obj2 by 1 and 2 respectively by invoking the constructor number(int n). **obj1.sum( )** will return the value 6. This is passed as argument in the statement

```
number obj3(obj1.sum( ));
```

where obj3 is initialized dynamically with the value returned by **obj1.sum( )**.

CHECK YOUR PROGRESS

5. State whether the following statements are true(T) or false(F) :
   (i) A class may have more than one destructor.
   (ii) When an object is destroyed, destructor is automatically called.
   (iii) Presence of many destructor in a class is called destructor overloading.
   (iv) A destructor can have a return type.
   (v) A destructor can have arguments like destructor.
   (vi) We can pass arguments to a constructor.
   (vii) Destructors cannot take arguments.
   (viii) Destructors can be overloaded.
6. Choose the appropriate option:
   (i) Return type of a destructor is
   (a) int (b) tilde
   (c) void (d) nothing, destructor has no return value
   (ii) A class may have_____destructor
   (a) two (b) only one
   (c) many (d) none of these
7. Distinguish between the following two statements:
   time t1(14, 10, 30); //statement1
   time t1 = time(14, 10, 30); //statement2

9.7 LET US SUM UP

Based on the discussion so far, the key points to be kept in mind from this unit are:

- A constructor is a special member function for automatic initialiation of an object. Whenever an object is created, the constructor will be called.

- Constructor should have the same name as that of class name to which it belong.
- Constructor has no return type.
- The constructor without argument is called a default constructor.
- Constructors may be overloaded to provide different ways of initializing an object.
- We may have more than one constructor with the same name, provided each has a different signature or arguments list.
- A constructor can accept a reference to its own class as a parameter. Such a constructor having a reference to an instance of its own class as an argument is known as copy constructor.
- C++ also provides another member function called destructor that releases memory by destroying objects when they are no longer required.
- A destructor never takes any argument nor does it return any value.

9.8 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education

9.9 ANSWERS TO CHECK YOUR PROGRESS

1. (i) False (ii) True (iii) True (iv) False (v) False
2. (i) (a) constructor (ii) (b) default constructor (iii) (d) Shape() {.....}
3. // Program for finding factorial of a number

```
#include<iostream.h>
#include<conio.h>
class factorial
{
```

```
        private:
            long n;
        public:
        factorial();
    };
    factorial::factorial()
    {
        cout<<"\nEnter the number to find factorial:
";
        cin>>n;
        long fact =1;
        while(n>1)
        {
            fact =fact *n;
            n=n-1;
        }
        cout<<"\n\nThe factorial is : "<<fact;
    }
    int main()
    {
        factorial f;
        getch();
        return 0;
    }
```

4. /*Program for finding the area of a rectangle of length 10.0 and breadth 6.0*/

```
#include<iostream.h>
#include<conio.h>
class rectangle
{
    private:
        float length,breadth;
    public:
        rectangle( )
        {
```

```
            length = 10.0;
            breadth=6.0;
        }
        float area( )
        {
            return( length*breadth);
        }
};
void main( )
{
    rectangle r;
    clrscr();
    cout<<"\n The area of the rectangle is:
"<<r.area()
        <<" square units";
    getch( );
    return 0;
}
```

5. (i) False (ii)True (iii) False (iv) False
   (v) False (vi) False (vii) True (viii) False
6. (i) (d) nothing, destructor has no return value,
   (ii) (b) only one
7. The first statement creates the object *t1* by calling the *time* constructor implicitly. On the other hand, the second statement creates the object *t1* by calling the *time* constructor explicitly.

9.10 MODEL QUESTIONS

1. What are constructor and destructors? Expalin how they differ from normal member functions.
2. What are the characteristics of a constructor?
3. Write short notes on (i) Default Constructor
   (ii) Copy Constructor

4. Differentiate between constructor and destructors.
5. Give the rules governing the declaration of a constructor and destructor.
6. What is a parameterized constructor ?
7. Write a C++ program to show overloading of constructors?
8. Define a class ‘complex_no’ which has two data members, one for representing the real part and the other for complex part. Define a constructor to initialize the object.
9. Define suitable constructor(s) for the STRING class defined below;

```
STRING
{
        int length;
        char s[50];
    public:
        //define suitable constructors
};
```

****

UNIT 10: OPERATOR OVERLOADING

UNIT STRUCTURE

10.1 LEARNING OBJECTIVES

After going through this unit, you will able to :

- describe the fundamental concept of overloading
- describe the use of the keyword *operator*
- illustrate the overloading of unary and binary operators
- describe manipulation of strings using operators

10.2 INTRODUCTION

So far, we have discussed the concept of class and objects and how to allocate required resource such as memory and initialize the objects of classes using constructors and how to deallocate the memories using destructors. C++ offers an another important feature namely *operator overloading*, through which operators like +,-,<=,>= etc. can be used with user defined data types, with some additional meaning.

In this unit, we will concentrates on the dicussion of overloading of operators (unary and binary) as well as the string manipulations using operators.

10.3 BASIC CONCEPT OF OVERLOADING

We know that operators (+,-,<=,>= etc.) are used to perform operation with the constants and variables. Without the use of the operators a programmer cannot write or built an expression. We have already used these operators with the basic data types such as *int* or *float* etc. The operator +(plus) can be used to perform addition of two variables but we cannot apply the + operator for addition of two objects. If we want to add two objects using the + operator then the compiler will show an error message. To avoid this error message you must have to make the compiler aware about the addition process of two objects. To perform operation with objects you need to redefine the definition of various operators. For example, for addition of objects X and Y, we need to define operator +(plus). Re-defining an operator does not change its original meaning. It can be used for both variables of built-in data types as well as objects of user defined data types.

Operator overloading in C++, permits to provide additional meaning to the operators such as +, *, >=, –, = etc., when they are applied to user defined data types. Hence, the operator overloading is one of the most valuable concepts introduced by C++ language. It is a type of polymorphism. We will discuss polymorphism in a later unit. C++ allows the following list of operators for overloading.

Table 10. 1:C++ Overlodable Operators

Operator Category	Operators
Arithmetic	+, -, *, /, %
Bit-Wise	&, |, ~, ^
Logical	&&, ||, !
Relational	<, >, ==, !=, <=, >=
Assignment	=
Arithmetic assignment	+=, -=, *=, /=, %=, &=, |=, ^=
Shift	>>, <<, >>=, <<=
Unary	++, --
Subscripting	[ ]
Function call	( )
Dereferencing	->
Unary sign prefix	+, -
Allocate and free	new, delete

10.4 operator KEYWORD

The keyword *operator* helps in overloading of the C++ operators. The general format of operator overloading is shown below :

```
ReturnType opeator OperatorSymbol ([arg1], [arg2])
{
     // body of the function
}
```

Here, the keyword **operator** indicates that the *OperatorSymbol* is the name of the operator to be overloaded. The operator overloaded in a class is known as *overloaded operator function.*

The following statements shows the use of the *operator* keywords.

```
class Index
{
     // class data and member function
     Index operator ++( )
     {
            index temp;
```

```
                value = value+1;
                temp.value = value;
                return temp;
            }
    };
```

Here, return type of the operator function is the name of a class within which it is declared. It can be defined as follows :

```
    class Index
      {
          // class data and member function
             Index operator ++( );
    };

Index Index :: operator ++( )
{
    index  temp;
    value = value+1;
    temp.value = value;
    return temp;
}
```

The operator function should be either a member function or a friend function. When the operator function is declared as member function and takes no argument, it is known as *unary operator overloading* and when it takes one argument it is known as *binary operator overloading*.

10.5 OVERLOADING UNARY OPERATORS

When an operator function takes no argument, it is called as unary operator overloading. You are already familiar with the operators ++, --, and -, *which have only single operands* are called unary operators. The unary operators ++ and -- can be used as **prefix** or **suffix** with the functions. The following program demonstrates the overloading of unary '--' operators.

// Program 10.1

```
#include<iostream.h>
#include<conio.h>
class unary
{
     int x, y, z;
     public :
     unary (int i, int j, int k) // parameterized construc-
tor
     {
          x=i; y=j; z=k;
     }
     void display();//displays contents of member variables
     void operator --();//overloads the unary operator --
};
void unary :: operator --()
{
     --x; --y; --z;// values of variables will decrease by 1
}
void unary :: display()
{
       cout<<"X="<<x<<"\n";
       cout<<"Y="<<y<<"\n";
       cout<<"Z="<<z<<"\n";
}
void main()
{
     clrscr();
     unary A(31, 41, 51);
     cout<<"\n Before Decrement of A :\n";
     A.display();
     --A;// calls the function operator --()
     cout<<"\n After Decrement of A :\n";
     A.display();
     getch();
}
```

OUTPUT : Before Decrement of A : X=31
Y=41
Z=51
After Decrement of A : X=30
Y=40
Z=50

10.6 OPERATOR RETURN TYPE

In the above example, we have declared the operator function of type *void* i.e. it will not return any value. But it is possible to return value and assign it to other object of same type. The return value of the operator is always of the class type, it means that class name will be in the place of the return type specification because we are applying the operator overloading properties only for the objects. Always remember that an operator cannot be overloaded for basic data types, so the return value of operator function will be of class type. The following program demonstrates the operator return types :

```
// Program 10.2
#include<iostream.h>
#include<conio.h>
class unary
{
     int x;
     public :
     unary () { x=0; }
     int getx() // returns the current value of variable x
     { return x; }
     unary operator ++();
};
unary unary :: operator ++()
{
     unary temp;
```

```
     x=x+1;
       temp.x=x;
       return temp;
}
void main()
{
     clrscr();
     unary A1,A2;
     cout<<"\n A1="<<A1.getx();
     cout<<"\n A2="<<A2.getx();
     A1=A2++;//first increment the value of A2 and assigns it to
A1
     cout<<"\n A1="<<A1.getx();
     cout<<"\n A2="<<A2.getx();
     A1++; // object A1 is incresed
     cout<<"\n A1="<<A1.getx();
     cout<<"\n A2="<<A2.getx();
     getch();
}
```

RUN : A1 = 0
A2 = 0
A1 = 1
A2 = 1
A1 = 2
A2 = 1

10.7 OVERLOADING BINARY OPERATORS

Binary operators are overloaded by using member functions and friend functions. The difference is in the number of arguments used by the function. In the case of binary operator overloading, when the function is a member function then the number of arguments used by the operator member function is one. When the function defined for the binary operator overloading is a friend function, then it uses two arguments. Here, we will dis-

cuss the overloading of binary operator when the operator function is a member function.

Binary operator overloading, as in unary operator overloading, is performed using a keyword *operator*. The following program demonstrates the overloading of binary operators.

// Program 10.3

```
#include <iostream.h>
#include<conio.h>
class Binary
{
    private:
      int x;
      int y;
    public:
      Binary()   //Constructor
      { x=0; y=0; }
    void getvalue() //Member Function for Inputting values
    {
      cout <<"\n Enter value for x:";
      cin >> x;
      cout << "\n Enter value for y:";
      cin>> y;
    }
    void displayvalue( )//Member Function for Outputting Values
    {
      cout<<"\n\nThe resultant value : \n";
      cout <<" x =" << x <<"; y ="<<y;
    }
    Binary operator +(Binary);//Binary is class name
};
Binary Binary :: operator +(Binary e2)
//Binary operator overloading for + operator defined
{
    Binary temp;
    temp.x = x+ e2.x;
```

```
    temp.y = y+e2.y;
    return (temp);
}
void main( )
{
    Binary e1,e2,e3;  //Objects e1, e2, e3 created
    clrscr();
    cout<<"\nEnter value for Object e1:";
    e1.getvalue( );
    cout<<"\nEnter value for Object e2:";
    e2.getvalue( );
    e3= e1+ e2;  //Binary Overloaded operator used
    e3.displayvalue( );
    getch();
}
```

OUTPUT : Enter value for Object e1 :
Enter value for x : 10
Enter value for y : 20
Enter value for Object e2 :
Enter value for x : 30
Enter value for y : 40
The Resultant Value : x = 40; y = 60

In the above example, the class Binary has created three objects e1, e2, e3. The values are entered for objects e1 and e2. The binary operator overloading for the operator '+' is declared as a member function inside the class Binary. The definition is performed outside the class Binary by using the scope resolution operator and the keyword *operator*.

The important aspect is that the following two statements are equivalent.

```
e3= e1 + e2; // e3= e1.operator +(e2)
```

The binary overloaded operator '+' is used. When the compiler encounters such expressions, it examines the argument type of the operator. In this statement, the argument on the left side of the operator '+', e1, is the object

of the class Binary in which the binary overloaded operator '+' is a member function. The right side of the operator '+' is e2. This is passed as an argument to the operator '+' i.e. the expression means **e3 = e1.operator +(e2).** The operator returns a value (binary object temp in this case), which can be assigned to another object (e3 in this case).

Since the object e2 is passed as argument to the operator'+' inside the function defined for binary operator overloading, the values are accessed as e2.x and e2.y. This is added with e1.x and e1.y, which are accessed directly as x and y.

Always remember that, in the overloading of binary operators, the left-hand operand is used to invoke the operator function and the right-hand operand is passed as an argument to the operator function.

10.8 STRINGS AND OPERATOR OVERLOADING

We are already familiar with the *strcat()* function which is used for concatenation of strings. Consider the following two strings

```
char str1[50]    = "Bachelor of Computer";
char str2[20]    = "Application";
```

The string str1 and str2 are combined, and the result is stored in str1 by invoking the function *strcat()* as follows :

```
strcat(str1, str2);
```

The same operation can be done by defining a string class and overloading the + operator. The following program demonstrates the concatenation of two string using the overloading concept.

// Program 10.4

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

class String
{
```

```
      private:
      char str[100];
      public:
      String()    //Constructor
      { strcpy(str," "); }
      String(char *msg)   //Constructor
      { strcpy(str, msg); }
      void display() //Member Function for Display strings
      {
           cout <<str;
      }
      String operator +(String s);
    };
    String String :: operator +(String s)
    //Binary operator overloading for + operator defined
    {
          String temp = str;
          strcat(temp.str, s.str);
          return temp;
    }
void main( )
{
    clrscr();
    String str1 = "Bachelor of Computer";
    String str2 = "Application";
    String str3;
    str3= str1+str2;
    cout<<"\n str1 =";
    str1.display();
    cout<<"\n str2 =";
    str2.display();
    cout<<"\n The String after str3=str1+str2 \n \n";
    str3.display();
    getch();
    }
```

In this program, the concatenation is performed by creating a temporary string object *temp* and initializing it with the first string. The second string is added to first string in the object temp using the *strcat()* and finally the resultant temporary string object temp is returned. Here, in the program, the length of str1+str2 should not exceed the array size 100 (i.e. char str[100]).

Thus, we have seen that, in C++ programming language, operator overloading adds new functionality to its existing operators. The programmer must add proper comments concerning the new functionality of the overloaded operator. The program will be efficient and readable only if operator overloading is used only when necessary. The operators that cannot be overloaded are - **.**, **?:**, **sizeof**, **::**, **.*** , #, ##.

10.9 TYPE CONVERSION

We cannot convert between user-defined data types(classes) just as we can convert between basic types. This is beacuse the compiler does not know anything about the user-defined type.

Now, let us look into how C++ handles conversions for its **built-in types (int, float, char, double etc.)**. When you make a statement assigning a value of one standard type to a variable of another standard type, C++ automatically will convert the value to the same type as the receiving variable, provided the two types are compatible.

For example, the following statements all generate numeric type conversions:

```
long count = 8;  // int value 8 converted to type long
double time = 11;// int value 11 converted to type double
int side = 3.33;  // double value 3.33 converted to type int 3
```

These assignments work because C++ recognizes that the diverse numeric types all represent the same basic thing, a number, and because C++ incorporates built-in rules for making the conversions. However, you can lose some precision in these conversions. For example, assigning

3.33 to the int variable results in only getting the value 3, losing the 0.33 part.

The C++ language does not automatically convert types that are not compatible.

For example, the statement

```
int * p = 10; // type clash
```

fails because the left-hand side is a pointer-type, whereas the right-hand side is a number. And even though a computer may represent an address internally with an integer, integers and pointers conceptually are quite different. For example, you wouldn't square a pointer. However, when automatic conversions fail, you may use a type cast:

```
int * p = (int *) 10;  // ok, p and (int *) 10 both pointers
```

This sets a pointer to the address 10 by type casting 10 to type pointer-to-int (that is, type int *).

Now, let us look into how C++ handles conversions from basic type to user-defined types vice-versa.

Basic type to user defined type :

This type of conversion can be easily carried out. It is automatically done by the compiler with the help of in-built routines or by type casting. In this type the left hand operand of = sign is always class type or user defined type and the right hand side operand is always basic type. The following program explains this type of conversion.

```
//Program 10.5
#include <iostream.h>
#include<conio.h>
class Test
{
  private:
```

```
        int x;
        float y;
        public:
        Test()    //Constructor
        { x=0; y=0; }
        Test(float z)  //Constructor with one argument
        { x=2; y=z; }
        void display()  // Function for displaying values
        {
             cout <<"\n x =" << x <<" y ="<<y;
             cout <<"\n x =" << x <<" y ="<<y;
        }
};
void main( )
{
        Test a;
        clrscr();
        a=9;
        a.display();
        a=9.5;
        a.display();
        getch();
}
```

OUTPUT : x=2 y=9
x=2 y=9
x=2 y=9.5
x=2 y=9.5

In the above program, the class Test has two data member of type integer and float. It also has two constructors one with no arguments and the second with one argument. In *main()* function, **a** is an object of class Test. When a is created the constructor with no argument is called and data memberms are initialize to zero. When a is initialized to 9 the constructor with float argument i.e. ***Test(float z)*** is invoked. The integer value is converted to float type and assigned to data member y. Again, when a is

assigned to 9.5, same process repeated. Thus, the conversion from basic to class type is carried out.

User defined type to basic type :

As we know, the compiler does not have any prior information about user defined data type using class, so in this type of conversion it needs to inform the compiler how to perform conversion from class to basic type. For this purpose, a conversion function should be defined in the class in the form of the operator function. The operator function is defined as an overloaded basic data type which takes no arguments. The syntax of such a conversion function is shown below-

```
operator Basic type( )
    {
        // steps for converting
    }
```

In the above syntax, you have noticed that, the conversion function has no return type. While declaring the operator function the following condition should always remember :

i) the operator function should not have any argument.
ii) it has no any return type.
iii) it should be a class member.

The following program demonstrates this conversion mechanism :

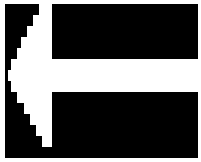
// Program 10.6

```
#include <iostream.h>
#include<conio.h>
class Time
{
    private:
    int hour;
      int minute;
      public:
```

```
        Time(int a)
        {
             hour=a/60;
             minute=a%60;
        }
        operator int()
        {
             int a;
             a=hour*60+minute;
             return a;
        }
};
void main( )
{
     clrscr();
     Time t1(500);
     int i=t1;  // operator int() is invoked
     cout<<"\n"<<"The value of i:"<<i;
     getch();
}
```

OUTPUT : The value of i : 500

In the above program, the statement ***int i=t1***, invokes the operator function which finally converts a time object to corresponding magnitude (of type int).



CHECK YOUR PROGRESS

1. Choose the correct answer from the following :
   a) Operator overloading is
      i) making C++ operators work with objects
      ii) giving C++ operators more than they can handle
      iii) giving new meaning to existing C++ operators
      iv) making new C++ operators

b) To convert from a user-defined class to basic type, you would use
   i) a built-in conversion function
   ii) a one argument constructor
   iii) an overloaded = operator
   iv) a conversion function that is a member of the class

c) To convert from a basic type to user-defined class, you would use
   i) a built-in conversion function
   ii) a one argument constructor
   iii) an overloaded = operator
   iv) a conversion function that is a member of the class

d) _____operator must have one class object.
   i) + ii) new
   iii) all iv) none of these

e) Binary overload operators are passed _____ arguments.
   i) one ii) two
   iii) no iv) none of the above

2. Fill in the blanks :
   i) The statement x=y will cause _____ if the objects are of different classes.
   ii)ii) _____ is making operators to woork with user defined data types.
   iii) Single argument constructor is usually defined in the _____ class.
   iv) _____ function must not have a return type.
   v) _____ are operators that act on only one operand.

10.10 LET US SUM UP

- Operator overloading is one of the important concepts in C++ which allows to provide additional meaning to operators +, -, >=, <= etc. when they are applied to user defined data types.

- Overloaded operators are redefined within a class using the keyword ***operator*** followed by an operator symbol. When an operator is overloaded, the produced symbol is called the operator function name.
- Overloading of operator cannnot change the basic meaning of an operator. When an operator is overloaded, its prooperties like systax, precedence and associativity remain constant.
- Operators ++, --, and -, *which have only single operands* are called unary operators. The unary operators ++ and -- can be used as **prefix** or **suffix** with the functions.
- The binary operators require two operand. Binary operators are overloaded by using member functions and friend functions.
- The operators which cannot be overloaded are - **.**, **?:**, **sizeof**, **::**, **.*** , #, ##.
- The concept of operator overloading can also be applied to data conversion. C++ offers automatic conversion of primitive data types.
- Actually there are three possibilities of data conversion :
  a) Basic type to user defined type(class type)
  b) User defined type(class type) to basic type
  c) Class type to another class type (we have not discussed here)

10.11 FURTHER READINGS

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education

10.12 ANSWERS TO CHECK YOUR PROGRESS

1. a) iii, b) iv, c) iii,
   d) ii, e) i.

2. i) Compiler error,
   ii) operator overloading,
   iii) destination,
   iv) casting operator,
   v) unary operator

10.13 MODEL QUESTIONS

1. What is operator overloading? Give the advantage of operator overloading.
2. What is operator function? Describe operator function with syntax and examples.
3. What is the difference between overloading of binary operators and of unary operators?
4. Explain the conversion from basic type to user defined type(class type) with examples.
5. Explain the conversion from user defined type(class type) to basic type with examples.
6. Write a program to overload the -- operator.
7. Write a program to overload the binary operator + in order to perform addition of complex numbers.
8. Write a program to overload the relational operator (>, <, ==) in order to perform the comparision of two strings.

****

UNIT 11 : INHERITANCE

UNIT STRUCTURE

11.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- describe the concept of inheritance in C++
- create new classes by reusing the members and properties of existing classes
- describe the advantages and disadvantages of inheritance in pro gramming
- describe how to use base class access specifier *public*, *private* and *protected*
- describe about the use of virtual base class and abstract class

11.2 INTRODUCTION

In this unit, we shall discuss one important and useful feature of Object - Oriented Programming (OOP) which is called ***inheritance***. C++ supports the concept of inheritance. We have already discussed the concept of classes and objects in our previous unit. The knowledge of class and objects is a prerequisite for this unit.

With the help of inheritance we can reuse (or inherit) the property of a previously written class in a new class. There are different types of inheritance which will be discussed in this unit. The concept of abstract and virtual base class will also be covered in this unit.

11.3 INHERITANCE

In Biology, *inheritance* is a term which represents the transformation of the hereditary characters from parents or ancestors to their descendent. In the context of Object-Oriented Programming, the meaning is almost same. The process of creating a new class from existing classes is called ***inheritance***. The newly created class is called ***derived class*** and the existing class is called the ***base class***. The derived class inherits some or all of the characteristics of the base class. Derived class may also possess some other characteristics which are not in the base class.

For example, let us consider two classes namely, “employee ” and “manager”. Whatever information is present in “employee” class, the same will be present in “manager’ also. Apart from that, there will be some extra information in “manager” class due to other responsibilities assinged to managers. Due to the facility of inheritance in C++, it is enough only to indicate those pieces of information which are specific to manager in its class. In addition, the “manager” class will inherit the information of “employee” class.

Before discussing the different types of Inheritance and their implementation in C++, we will first denote the advantages and disadvantages of inheritance.

Advantages of Inheritance

- ***Reusability***

  Reusability is an important feature of Object Oriented Programming. We can reuse code in many situations with the help of inheritance. The base class is defined and once it is compiled, it need not be rewritten. Using the concept of inheritance the programmer can create as many derived classes from the base class as needed. New features can also be added to each derived class when required.

- ***Reliability and Cost***

  Reusability would not only save time but also increase reliability and decrease maintenance cost.

- ***Saves Time and Effort***

  The reuse of class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

Disadvantages of Inheritance

- Inappropriate use of inheritance makes a program more complicated.
- In the class hierarchy various data elements remain unused, the memory allocated to them is not utilized.

11.3.1 Defining a Derived Class

A derived class is defined by specifying its relationship with the base class in addition to its own details. The general form of deriving a new class from an existing class is as follows:

```
class DerivedClassName : access specifier BaseClassName
  {
        .....
        .....   //members of derived class
        .....
  };
```

The derived class name and the base class name are separated by a colon "**:**". We can derive classes using any of the three base class access specifiers: ***public***, ***private*** or ***protected***. If we do not specify the access specifier, then by default it will be *private.* Generally, it is convenient to specify the access specifier while deriving a class. Access specifiers are sometime called *visibility mode*. It determines the access control of the base class members inside the derived class. In the previous unit, we have already learnt about *private* and *public* access specifier while declaring classes. ***Protected*** specifier has a significant role in inheritance.

11.3.2 Accessing Base Class Members

There may be three types of base class derivation:

- Public derivation
- Private derivation
- Protected derivation

- **Public derivation**

When the base class is inherited by using ***public access specifier*** then all public members of the base class become public members of the derived class, and all protected members of the base class become protected members of the derived class. The private members of the base class remain private and are not accessible by members of derived class. Let us examine this with the following example:

// **Program 11.1:** Base class with public access specifier

```
#include<iostream.h>
#include<conio.h>
class base
{
  private:
       int num1,num2;
```

```
  public:
      void input(int n1, int n2 )
                  {
                        num1=n1;
                        num2=n2;
                  }
      void display( )
      {
           cout<<"Number 1 is: "<<num1<<endl;
           cout<<"Number 2 is: "<<num2;
      }
  };  //end of base class

class derived : public base
    {
        private:
            int num3;
        public:
            void enter(int n3)
            {
                num3=n3;
            }
            void show()
            {
                cout<<"\nNumber 3 is: "<<num3;
            }
    }; //end of derived class
    int main()
    {
    derived d;     // d is an object of derived class
    clrscr();
    d.enter(15); //enter() function is called by object d
    d.input(5,10);      /* accessing base class member. input()
is a public member of base class */
    d.display(); /* accessing base class member display() is a
```

```
public member of base class */
        d.show();
        getch();
        return 0;
    }
```

OUTPUT: Number 1 is : 5
Number 2 is : 10
Number 3 is : 15

Here, ***d*** is a derived class object. With the statement ***d.input(5,15);*** we have made a call to the function *input()* of base class by the derived class object. Similarly, we have called *display( )* member function of base class.

- **Private derivation**

When the base class is inherited by using ***private access specifier***, all the public and protected members of the base class become private members of the derived class. Therefore, public members of base class can only be accessed by the member functions of the derived class and they are not accessible to the objects of the derived class.

For example, if we use the statement

```
class derived : private base
```

instead of `class derived : public base`

in the above program, it will give two error messages while compiling:

Error: base::input(int,int) is not accessible

Error: base::display() is not accessible

As the base class is privately inherited, *input()* and *display()* become private to the derived class althogh they were public in the base class. So other functions like *main( )* cannot access them. Statement like **d.input(5,10);** and **d.display( );** will be invalid in that case.

Protected Members and Inheritance:

Protected members provide greater flexibility in case of inheritance. By using ***protected*** instead of private declaration, we can create class members that are private to their class but that can still be inherited and accessed by derived class. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it.

When a ***protected*** member is inherited in ***public*** mode, it becomes ***protected*** in the derived class too and therefore is accessible by the member functions of the derived class. It also becomes ready for further inheritance. If a base class is inherited as ***private***, then the ***protected*** member of base class becomes ***private*** in the derived class. Although it is available to the member function of the derived class, it is not available for further inheritance since ***private*** members cannot be inherited.

/***Program 11.2:*** Program showing protected members inherited in public mode */

```
#include<iostream.h>
#include<conio.h>
class base
{
    protected:
int num1,num2;     /*private to base, protected to derived and accessible by derived class member function*/
    public:
    void input(int n1, int n2 )
    {
        num1=n1;
        num2=n2;
    }
    void display( )
    {
```

```
        cout<<"Number 1 is: "<<num1<<endl;
        cout<<"Number 2 is: "<<num2;
    }
};  //end of base class

class derived : public base//base class is publicly inherited
{
    private:
    int s;
    public:
    void add( )
    {
    s=num1+num2 ;      /* derived class accessing base class protected member num1, num2 */
    }
    void show( )
    {
        cout<<"\nSummation is : "<<s;
    }
};  //end of derived class
int main()
{
    derived d;     // d is an object of derived class
    clrscr();
d.input(10,20); /* accessing base class member. input() is a function of base class */
d.display(); /* accessing base class member. display() is a function of base class */
    d.add();
    d.show();
    getch();
    return 0;
}
```

OUTPUT: Number 1 is : 10
Number 2 is : 20
Summation is : 30

The derived class member function *void add()* can access *num1* and *num2* of the base class because *num1* and *num2* are declared as *protected* and the base class access specifier is *public*.

- **Protected derivation**

When the base class is inherited by using ***protected access specifier***, then all protected and public members of base class become protected members of the derived class. Let us consider the following example:

//***Program 11.3*** : Base class derived as protected

```
#include<iostream.h>
#include<conio.h>
class base
{
    protected:
        int num1,num2;
    public:
        void input(int n1, int n2 )
        {
            num1=n1;
            num2=n2;
        }
        void display( )
        {
            cout<<"Number 1 is: "<<num1<<endl;
            cout<<"Number 2 is: "<<num2;
        }
};  //end of base class
class derived : protected base
{
    private:
```

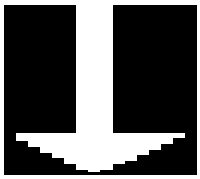
```
        int s;
    public:
        void add( )
        {
        input(30,60);      /*member function add() of derived
class can access input() as it is inherited as protected */
        s=num1+num2; // num1,num2 are inherited as
                              // protected, so add() can access
            }
            void showall( )
            {
            display();    //display() is inherited as protected
            cout<<"\nSummation is : "<<s;
            }
    };    //end of derived class
    int main()
    {
        derived d;
        clrscr();
    //d.input(10,20);  /* invalid. input() is inherited as protected member
of derived. main() can't access it */
    d.add(); /* accessing base class member display() is a function of
base class */
    d.showall(); // public member of derived
    // d.display();     /* invalid. display() can be accessible by derive
class member function only */
        getch();
        return 0;
    }
```

OUTPUT:

```
Number 1 is   :   30
Number  2  is  :  60
Summation is  :   90
```

In the program we can see that *input()*, *display()*, *num1*, *num2* of base class are inherited as protected to derive class. The member functions *add()*, *showall()* of derived class can use them; as protected

members are accessible by derive class members. But main() is not a member function and it cannot access *input()* and *display()*. Although *num1*, *num2* are protected to derived class, but they behave as private to base class.



CHECK YOUR PROGRESS

1. Answer the following by selecting the appropriate option:

   (i) By using protected, one can create class members that
   (a) cannot be inherited and accessed by a derived class
   (b) can be accessed by a derived class
   (c) can be public
   (d) none of these

   (ii) Class members are by default _____
   (a) protected (b) public
   (c) private (d) none of these

   (iii) When base class access specifier is protected, then public members of base class can be accessible by
   (a) member function of derived class
   (b) main() function
   (c) objects of derived class
   (d) none of these

   (iv) Private data members of base class can be inherited by declaring them as
   (a) private (b) public
   (c) protected (d) none of these

   (v) If we donot specify the visibility mode in base class derivation then by default it will be
   (a) protected (b) private
   (c) public (d) none of these

(vi) Private data members can be accessed

(a) from derive class

(b) only from the base class itself

(c) both from the base class and from its derived class

(d) None of these

11.4 TYPES OF INHERITANCE

A program can use one or more base classes to derive a single class. It is also possible that one derived class can be used as base class for another class. Depending on the number of base classes and levels of derivation inheritance is classified into the following forms:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance
- Multipath Inheritance

11.4.1 Single Inheritance

The programs discussed so far in this unit are examples of single inheritance. In ***single inheritance*** the derived class has only one base class and the derived class is not used as base class. The pictorial representation of single inheritance is given in Fig. 11.1

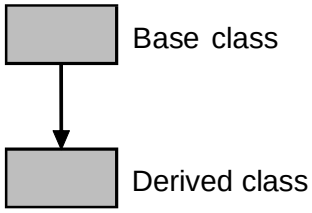


Fig. 11.1: Single Inheritance

The arrow directed from base class towards the derived class indicates that the features of base class are inherited to the derived class. In the following program, we have derived "***employee***" class from "***person***" class. Data member "***name***" and "***age***" are common to both of the two classes. "***name***" and "***age***" are declared as ***protected*** so that derive class can inherit "***name***" and "***age***" from the base class "***person***". Base class is inherited in public mode. Employee may have some other data like designation, salary. So the other data members of "***employee***" class are "***desig***" and "***salary***".

/* **Program 11.4**: Single inheritance with protected data member and public inheritance of base class */

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class person //base class "person"
{
    protected:
    char name[30];     //protected to derived class 'employee'
    float age;
    public:
    void enter(char *nm, float a)/* base class member function */
    {
        strcpy(name,nm);
        age=a;
    }
    void display()     //base class member function
    {
        cout<<"Name: "<<name<<endl;
        cout<<"Age: "<<age<<endl;
    }
};
class employee : public person     */base class "person" is publicly
```

```
                                    inherited */
{                                   */by derived class "employee" */
    private:
    float salary;
    char desig[20];
    public:
    void enter_data(char *n,char *d,float ag,float
s)
    {
    strcpy(name,n);   //"name" of base class can be accessible
                                //by derived class member function
            strcpy(desig,d);
            salary=s;
            age=ag;  //age can be accessible by
                     //enter_data() of derived class
            }
            void display_all()     //derived class member
function
            {
                 display();    //can be used here as publicly
inherited
                 cout<<"Designation:
"<<desig<<endl;
                 cout<<"Salary: "<<salary<<endl;
            }
};
int main()
{
employee e1,e2;  //e1,e2 are objects of derived class "employee"
    person p;     // p is an object of base class "Person"
    clrscr();
    e1.enter_data("Raktim","Clerk",32,5000);
    cout<<"Employee Details .... "<<endl;
```

```
    e1.display_all();
e2.enter("Vaskar",41);      /*erived class object e1 accessing public member of base enter() */
e2.display();      /*derived class object e2 accessing public member of base display() */
    cout<<endl<<"Person Details .... "<<endl;
    p.enter("Pragyan",24);
    p.display();
    getch();
    return 0;
}
```

Here, the derived class "***employee***" uses ***name*** and ***age*** of base class "***person***" with the help of derived class member function ***enter_data()***.

Two different classes may have member functions with the same name as well as the same set of arguments. But in case of inheritance, an ambiguous situation arises when base class and derived classes contain member functions with the same name. In main(), if we call member function of that particular name of base class with derived class object, then it will always make a call to the derived class member function of that name. This is because, the function in the derived class *overrides* the inherited function. However, we can invoke the function defined in base class by using the ***scope resolution operator (::)*** to specify the class. For example, let us consider the following program.

/***Program 11.5**: When base and derived class has member functions with same name*/

```
#include<iostream.h>
#include<conio.h>
class B
{
```

```
  protected:
  int p;
  public:
  void enter()
  {
       cout<<"\nEnter an integer:";
       cin>>p;
  }
  void show()
  {
       cout<<"\n\nThe number in Base Class is:
"<<p;
  }
};
class D : public B
{
  private:
  int q,r;
  public:
       void enter() //overrides enter() of "B"
       {
            B::enter();
            cout<<"\nEnter an integer:";
            cin>>q;
       }
       void show()
       {
            r=p*q;
            cout<<"\nEntered numbers in Base and
                Derived class are:"<<p<<"\t"<<q;
            cout<<"\n\nThe product is :"<<r;
       }
};
int main()
{
  D d;          //d is an object of class derived class "D"
```

```
    clrscr();
    d.enter();          //invokes enter() of "D"
    d.show();           //invokes show() of "D"
    d.B::show();  //invokes show() of "B"
    getch();
    return 0;
}
```

In the program, the function name *show()* is same in both base “B” and derived class “D”. To call show() of base class “B”, we have used the statement **d.B::show();** If we use simply **d.show();** then it will invoke show() of derived class “D”.

When a derive class implements a function that has the same name as well as the same set of arguments as the function in the base class, it is called ***function overriding***. When such a function is called through a object of derived class, then the derived class function would be invoked. However, that function in base class would remain hidden.

But there are certain situations where function overriding plays an important role.

11.4.2 Multiple Inheritance

When one class is derived from two or more base classes then it is called ***multiple inheritance***. This type of inheritance allows us to combine the properties of more than one existing classes in a new class. Fig. 11.2 depicts multiple inheritance

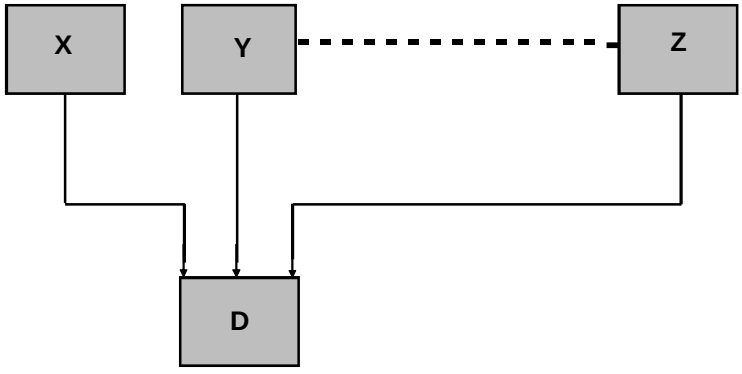


Fig.11.2: Multiple inheritance

We have to specify the base classes one by one separated by commas with their access specifiers. The general form of deriving a derived class from numbers of base class is as follows:

```
class D : public X, public Y, public Z
{
     ......  //body of the derived class
};
```

where X,Y, Z are base classes and D is the derived class. There may be numbers of base classes in multiple inheritance which is indicated by the dotted line in the figure.

For demonstration of multiple inheritance let us consider the following program. There are three base classes and one derived class. The derived class CHARACTER has one private member “n” and two public member functions “enter()” and “show()”. The function “enter()” is used to read a number, a vowel, a consonent and a symbol from the keyboard and the “show()” function is used to display the contents on the screen. The class members of all the three base classes are publicly derived.

// **Program 11.6** : Example of Multiple inheritance

```
#include<iostream.h>
#include<conio.h>
class V  //base class
{
    protected:
         char v;
};
class C  //base class
{
    protected:
         char c;
};
class S  //base class
```

```
{
    protected:
        char s;
};
class CHARACTER : public V, public C, public S
{
    private:
        int n;
    public:
        void enter() //derived class member function
        {
            cout<<"\nEnter a vowel:";
        cin>>v;//accessing protected member v of class
"VOWEL"
            cout<<"\nEnter a consonent:";
        cin>>c;  //accessing c of "CONSONENT" class
        cout<<"\nEnter a symbol:";
        cin>>s  //accessing s of "SYMBOL" class
        cout<<"\nEnter a number:";
        cin>>n;  //accessing n of "NUMBER" class
    }
    void show()
    {   cout<<"\nThe entered characters are
:\n\n";
        cout<<"\nVowel: "<<v;
        cout<<"\nConsonent: "<<c;
        cout<<"\nSymbol: "<<s;
        cout<<"\nNumber: "<<n;
    }
};
int main()
{
    CHARACTER o; //o is an object of derived class
"character"
    clrscr();
```

```
    o.enter();
    o.show();
    getch();
    return 0;
}
```

One suitable example of the implementation of multiple inheritance is shown in the program below:

/***Program 11.7**: Program showing multiple inheritance with two base class (practical, theory) and one derived class (result) /*

```
#include<iostream.h>
#include<conio.h>
class practical  //base class "practical"
{
    protected:
    float p1_marks, p2_marks,total;
    public:
    void practical_marks()
    {
    cout<<"Enter marks of practical paper 1 and
paper 2: ";
        cin>>p1_marks>> p2_marks;
    }
    float add()  //returns the total of practical
    {
        total=p1_marks+p2_marks;
        return total;
    }
    void display_practical()
    {
        cout<<endl<<"Total      Practical
marks:"<<total;
    }
};    //end of "practical" class
```

```
class theory // base class "theory"
{
    protected:
    float phy, chem, math, total_marks;
    public:
    void theory_marks(){
cout<<"Enter marks of Physics, Chem and Maths:";
    cin>>phy>>chem>>math;
}
float sum()
{
    total_marks=phy+chem+math;
    return total_marks;
}
void display_theory()
{
    cout<<endl<<"Total                Theory
marks:"<<total_marks;
}
};  //end of base class "theory"
class result : public practical,public theory
{
    protected:
    int roll;
    float grand_total,t,p;
    public:
    void enter() {
cout<<"ENTER STUDENT INFORMATION ..... "<<endl;
    cout<<"Enter Roll no.:";
    cin>>roll;
theory_marks();    //inherited publicly from base class "theory"
practical_marks();     //inherited from base class "practical"
}
void theory_practical()
{
    t=sum()
```

```
    p=add();
    grand_total=t+p;
cout<<"\nThe total marks of the student is:"
<<grand_total;
}
};  //end of derived class "result"
int main()
{
    result s1;   //object of derived class
    clrscr();
    s1.enter();
    s1.theory_practical();// accessing derived class mem-
ber
s1.display_theory(); //s1 accessing "display_theory()" of
"theory"
    s1.display_practical();
    getch();
    return 0;
}
```

When we execute the program entering marks for practical and theory papers for a particular student, then it will display the result as follows:

ENTER STUDENT INFORMATION.....
Enter Roll no.: 1
Enter marks of Physics, Chemistry and Mathematics: 65 72 81
Enter marks of practical paper 1 and paper 2 : 25 26
The total marks of the student is : 269
Total Theory marks : 218
Total Practical marks : 51

The above program consists of three classes: two base classes ("***practical***" and "***theory***") and one derived class ("***result***"). The member function "***enter()***" of derive class inherits member functions "***practical_mark()***" and "***theory_ marks()***" of base class "***practical***"

and "***theory***" respectively. Similarly, member function "***theory_practical()***" uses "***sum()***" and "***add()***" of base class to calculate the "*grand_total*" marks of student. Thus, in the derived class we need not have to write functions for entering practical and theory marks. We just inherit them from the base classes.

EXERCISE

Q. Suppose a class D is derived from class B. B has two public member functions *getdata()* and *showdata()* and D has two public functions *readdata()* and *displayall()*. Define the classes such that both function *getdata()* and *showdata()* should be accessible in the *main()* function.

Hierarchical Inheritance

Derivation of more than one classes from a single base class is termed as ***hierachical inheritance***. This is a very common form of inheritance in practice. The rules for defining such classes are the same as in single inheritance. The pictorial representation of hierarchical inheritance is shown in Fig. 11.3

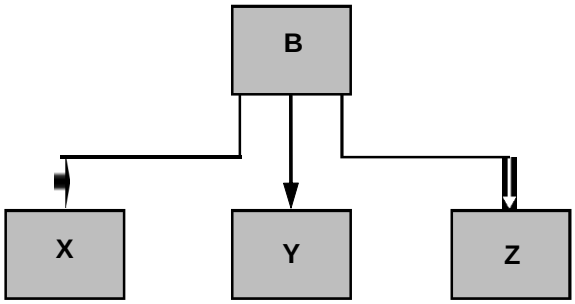


Figure11 3: Hierarchical Inheritance

For demonstration of hierarchical inheritence let us consider a program with one base class ("student") and three derived classes ("arts", "science" and "commerce").

//**Program 11.8**: Demonstration of hierarchical inheritance

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

```
#include<conio.h>
class student    // base class "student"
{
    protected:
    char fname[20],lname[20];
    int age,roll;
    public:
    void student_info()
    {
        cout<<"Enter the first name and last name:
";
        cin>>fname>>lname;
        cout<<"\nEnter the Roll no.and Age: ";
        cin>>roll>>age ;
    }
    void display()
    {
        cout<<"\nRoll Number = "<<roll;
        cout<<"\nFirst        Name        =
"<<fname<<"\t"<<lname;
        cout <<"\nAge = " << age;
    }
};
class arts : public student   //derived class arts
{
    private:
    char asub1[20], asub2[20], asub3[20] ;
    public:
    void enter_arts()
    {
        student_info(); //base class member function
        cout <<"\n Enter the subject1 of the arts
student:";
    cin >> asub1 ;
    cout<<"\nEnter the subject2 of the arts stu-
dent:";
```

```
    cin >> asub2 ;
    cout<<"\nEnter the subject3 of the arts stu-
dent:";
    cin >> asub3 ;
}
void display_arts()
{
    display();//base class member function
    cout<<"\n\t Subject1 of the arts student="<<
asub1;
    cout<<"\n\t Subject2 of the arts student="<<
asub2;
    cout<<"\n\t Subject3 of the arts student="<<
asub3;
}
};
class commerce : public student    //derived class
"commerce"
{
    private:
    char csub1[20], csub2[20], csub3[20] ;
    public:
    void enter_com(void)
    {
    student_info();  //base class member function
cout<<"\tEnter the subject1 of the commerce student:";
    cin>> csub1;
cout<<"\tEnter the subject2 of the commerce student:";
    cin>>csub2 ;
cout<<"\tEnter the subject3 of the commerce student:";
    cin>> csub3 ;
    }
    void display_com()
    {
    display();   //base class member function
cout<<"\nSubject1 of the commerce student="
```

```
<<csub1;
cout<<"\nSubject2 of the commerce student="
<<csub2;
cout<<"\nSubject3 of the commerce student="<< csub3
    }
    };
class science : public student  //derived class "science"
{
    private:
    char ssub1[20], ssub2[20], ssub3[20] ;
    public:
    void enter_sc(void)
    {
    student_info();  //base class member function
cout<<"\nEnter the subject1 of science student:";
    cin>>ssub1;
cout<<"\nEnter the subject2 of science student:";
    cin>>ssub2 ;
cout<<"\nEnter subject3 of the science student:";
    cin>>ssub3 ;
    }
    void display_sc()
    {
    display();   //base class member function
cout<<"\nSubject1 of the science student="<< ssub1
cout<<"\nSubject2 of the science student="<< ssub2;
cout<<"\nSubject3 of the science student="<< ssub3;
    }
};
int main()
{
    arts a ; //a is an object of derived class "arts"
    clrscr();
    cout << "\n Entering details of the arts
student\n";
```

```
    a.enter_arts( );
    cout <<"\nDisplaying the details of arts student\n";
    a.display_arts( );
    science s;   //s is an object of derived class "science"
    cout <<"\n\n Entering details of science student\n"
    s.enter_sc( );
    cout<<"\n Display details of the science student\n";
    s.display_sc( );
    commerce c ;    //c is an object of derived class "commerce"
    cout<<"\n\nEnter details of commerce student\n";
    c.enter_com( );
    cout << "\n Display details of commerce student\n";
    c.display_com() ;
    getch();
    return 0;
}
```

11.4.4 Multilevel Inheritance

C++ also provides the facility of ***multilevel inheritance***, according to which the derived class can also be derived by an another class, which in turn can further be inherited by another and so on. For instance, a class X serves as a base class for class Y which in turn serves as base class for another class Z. The class Y which forms the link between the classes X and Y is known as the ***intermediate base class***. Further, Z can also be used as a base class for another new class. The following figure depicts multilevel inheritance.

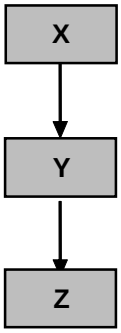


Figure11.4: Multilevel Inheritance

11.4.5 Hybrid Inheritance

It is possible to derive a class involving more than one type of inheritance. ***Hybrid inheritance*** is that type of inheritance where several forms of inheritance are used to derive a class. There could be situations where we need to apply two or more types of inheritance to design a particular program.

For example, let us assume that we are to design a program which will select players for a particular compitetion. For this purpose we could consider four classes PLAYER, GAME, RESULT and PHYSIQUE. PLAYER class contains the player details including name, address, location etc. GAME class can be derived from PLAYER class. Again, if weightage for physical test should be added before finalizing the result, then we can inherit that from PHYSIQUE. RESULT class is derived from two base classes GAME and PHYSIQUE. The following diagram gives us the inhertance relationship between various classes.

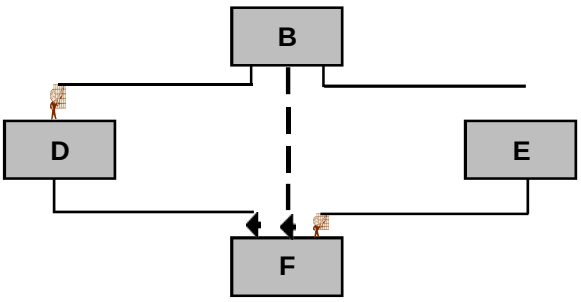


Fig.11.5: Hybrid Inheritance

In the diagram, RESULT has two base classes,GAME and PHYSIQUE. GAME is not only a base class but also a derived class. Here we can see that two types of inheritance *multiple* and *multilevel* are combined to create the RESULT class.

11.4.6 Multipath Inheritance

The inheritance where a class is derived from two or more classes, which are in turn derived from the same base class is known as ***multipath inheritance***. There may be many types of inheritance such as multiple, multilevel, hierarchical etc. in multipath inheritance. Certain difficulties may arise in this type of inheritance. Suppose we have two derived classes D and E that have a common base class B, and we have another class F that inherits from D and E.

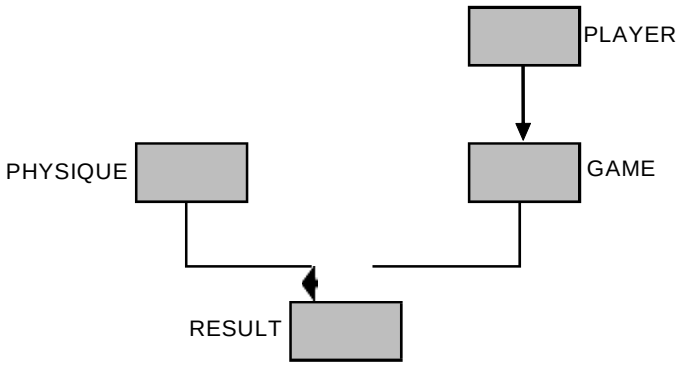


Fig.11.6 : Multipath Inheritance

In the above diagram, we can observe three types of inheritances, i.e., multiple, multilevel and hierarchical. For better illustration let us consider the following program.

// **Program 11.9:** Demosntration of multipath inheritance

```
#include<iostream.h>
#include<conio.h>
class B
{
    protected:
    int b;
```

```
};
class D : public B    //B is publicly inherited by D
{
    protected:
    int d;
};
class E : public B    //B is publicly inherited by E
{
    protected:
    int e;
};
class F : public D, public E//D,E are publicly inherited
by F
{
    protected:
    int f;
    public:
    void enter_number()
    {
        cout<<"Enter some integer values for
b,d,e,f:";
        cin>>b>>d>>e>>f;
    }
    void display()
    {
    cout<<"\nEntered numbers are:\n";
    cout<<"\nb= "<<b<<"\nd= "<<d<<"\ne= "<<e<<"\nf=
"<<f;
    }
};
int main()
{
    F obj;        //instantiaton of class F
    clrscr();
    obj.enter_number();   //enter_number() of F is called
```

```
    obj.display();
    getch();
    return 0;
}
```

Now, if we instantiate class F and call the functions ***enter_number()*** and ***display()***, then the compiler shows the following types error messages:

Error M.cpp 28: Member is ambiguous:'B ::b' and 'B::b'

Error M.cpp 33: Member is ambiguous:'B ::b' and 'B::b'

This is due to the duplication of members of class B in F. The member **b** of class B is inherited twice to class F: one through class D and another through class E. This leads to ambiguity. To avoid such type of situation, ***virtual base class*** is introduced.

11.5 VIRTUAL BASE CLASSES

C++ provides the concept of ***virtual base class*** to overcome the ambiguity occurring due to *multipath inheritance*. While discussing multipath inheritance, we have faced a situation which may lead to duplication of inherited members in the derived class F (Fig.11.6). This can be avoided by making the common base class (i.e.,B) as *virtual base class* while derivation. We can declare the base class B as ***virtual*** to ensure that D and E share the same data member B.

This is shown in the following program which is the modification of the previous program 11.9.

/***Program 11.10:** Virtual base class and removal of ambiguity occuerd in multipath inheritance */

```
#include<iostream.h>
#include<conio.h>
class B
```

```
{
    protected:
    int b;
};
class D:public virtual B
//B is publicly inherited by D and made virtual
{
    protected:
    int d;
};
class E:public virtual B
//B is publicly inherited by E and made virtual
{
    protected:
    int e;
};
class F : public D, public E
//D,E are publicly inherited by F
{
    protected:
    int f;
    public:
    void enter_number()
    {
        cout<<"Enter some integer values for b,d,e,f:
";
        cin>>b>>d>>e>>f;
    }
    void display()
    {
        cout<<"\nEntered numbers are:\n";
        cout<<"\nb= "<<b<<"\nd= "<<d<<"\ne=
"<<e<<"\nf= "<<f;
```

```
    }
};
int main()
{
    F obj; //instantiaton of class F
    clrscr();
    obj.enter_number();
    obj.display();
    getch();
    return 0;
}
```

Here we have used the keyword virtual in front of the base class specifiers to indicate that only one subobject of type B, shared by class D and class E, exists.. When a class is made a ***virtual base class***, C++ takes the necessary action that only one copy of that class is inherited, regardless of how many paths exist between the virtual base class and a derived class.

11.6 ABSTRACT CLASSES

The objects created often are the instances of a derived class but not of the base class. The base class just becomes a structure or foundaion with the help of which other classes are built and hence such classes are called ***abstract class*** or ***abstract base class***. In other words, when a class is not used for creating objects then it is called *abstract class*. In the *Program 10.10*, B is an abstract class since it was not used for creating any object.

LET US KNOW

Inheritance and Constructors, Destructors

Although constructors are suitable for initializing objects , we have not used them in any program in this unit for the sake of simplicity. But if we use contructors in program, then we must follow certain definite rules

while inheriting derive classes. If the base class contains no argument constructor then the derived class does not require a constructor. If any base class contains parameterized constructor, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors. In case of inheritance, normally derived classes are used to declare objects. Hence it is necessary to define constructor in the derived class. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in derived class is executed. Destructors are executed in the reverse order of constructor execution.

CHECK YOUR PROGRESS

2. Select whether the following statements are True (T) or False (F):
   (i) A class can serve as base class for many derived classes.
   (ii) When one class is derived from another derived class then that is called *multiple inheritance*.
   (iii) When one class is derived from more than one base class then that is called *multiple inheritence*.
   (iv) When more than one form of inheritance is used in designing a class then that type is called *hybrid inheritance*.
3. Answer the following by selecting the appropriate option:
   (i) In multilevel inheritance, the middle class acts as
   (a) only derived class (b) only base class
   (c) base class as well as derived class
   (d) none of the above
   (ii) A class is declared “virtual” when
   (a) more than one class is derived
   (b) two or more classes have common base class
   (c) there are more than one base classes

(d) none of the above

(iii) When a class is not used for creating objects, it is called

(a) abstract class (b) virtual base class

(c) derived class (d) none of these

(iv) *Intermediate base class* is present in case of

(a) single inheritance (b) multiple inheritance

(c) multilevel inheritance (d) hierarchical inheritance

11.7 LET US SUM UP

- ***Inheritance*** is one of the most useful and essential characteristics of object-oriented programming language. If we have developed a class and we want a new class that is almost similar, but slightly different, the principles of inheritance come handy. The existing class is known as ***base*** class and the newly formed class is known as ***derived*** class.The derived class can have some other characteristics which are not in base class.
- Private members of a class cannot be inherited either in public mode or in private mode.
- When a public member inherited in public, protected and private mode, then in derived class it remains with the same access specifiers as in base class i.e., public, protected and private respectively.
- A protected member inherited in public mode becomes protected, whereas inherited in private mode becomes private in derived class.
- The protected and public data
- In ***single inheritance***, one new class is derived from a single base class.
- When a class is derived using the properties of several base classes, then it is called ***multiple inheritance.***

- The process of deriving a class from another derived class is called ***multilevel inheritance***.
- More than one class can be derived from only one base class i.e., characteristics of one class can be inherited by more than one class. This is called ***hierarchical inheritance***.
- When different types of inheritance are applied in a single program then it is termed as ***hybrid inheritance***.
- When a class is derived from two or more classes, which are derived from the same base class, such type of inheritance is known as ***multipath inheritance***.
- We can make a class ***virtual*** if it is a base class that has been used by more than one derived class as their base class. When classes are declared as *virtual*, the compiler takes necessary caution to avoid the duplication of the member variables.
- When a class is not used for creating objects, it is called an ***abstract class***.

11.8 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education

11.9 ANSWERS TO CHECK YOUR PROGRESS

1. (i) (b) can be accessed by a derived class
   (ii) (c) private
   (iii) (a) member function of derived class
   (iv) (c) protected

(v) (b) private

(vi) (b) only from the base class itself

2. (i) True (ii) False
   (iii) True (iv) True
3. (i) (c) base class as well as derived class
   (ii) (b) two or more classes have common base class
   (iii) (a) abstract class
   (iv) (c) multilevel inheritance

11.10 MODEL QUESTIONS

1. What does inheritance mean in C++?
2. What are the types of inheritance? Explain any three of them with examples.
3. What are the different types of visibility modes of base class?
4. Write a program to derive a class from multiple base classes.
5. When do we make a class virtual?
6. What are abstract base classes?
7. Explain multipath inheritance.
8. Write a C++ program involving appropriate type of inheritance which will inherit two classes *triangle* and *rectangle* from *polygon* class. Use member functions for entering appropriate parameters like *width*,*height* etc. and to calculate the area of triangle and rectangle.
9. Consider a case of University having the disciplines of Engineering, Management, Science, Arts and Commerce. There are many colleges in the University. Assuming a college can run a course partaining to only one discipline, draw the class diagram.To which type of inheritance does this structure belong?

****

UNIT 12 : VIRTUAL FUNCTIONS AND POLYMORPHISM

UNIT STRUCTURE

12.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- describe polymorphism and its types
- define the rules for virtual function
- describe how to use virtual function to achieve run-time polymorphism
- describe and implement pure virtual function

12.2 INTRODUCTION

In the previous unit, we have studied the concept of inheritance and its importance in Object-Oriented Programming language like C++.

In this unit, we will discuss one useful feature of Object-Oriented Programming, ***polymorphism***. It is the ability of objects to take different forms. The ability to display variable behavior depending on the situation is a great facility in any programming language. In the earlier units, we have seen operator overloading and function overloading. Those are also one kind of polymorphism. C++ supports a mechanism known as ***virtual***

function to achieve run-time polymorphism. The necessity and usefulness of virtual functions in programming will also be covered in this unit.

12.3 POLYMORPHISM

The word *polymorphism* is a combination of two Greek words, *poly* and *morphism*. “Poly” means “many” and “morphism” means “form”. The functionality of this feature accords with its name.

12.3.1 Types of Polymorphism in C++

Polymorphism is supported by C++, at both compile-time and at run-time. Hence, there are two types of polymorphism:

- ***Compile-time Polymorphism***
- ***Run-time Polymorphism***

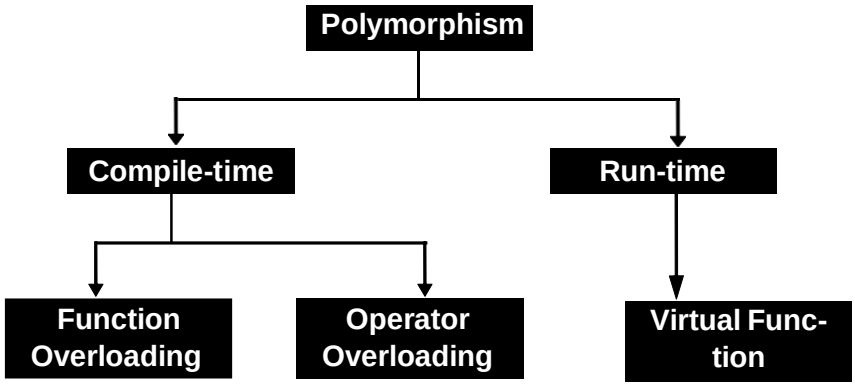


Fig. 12.1: Polymorphism in C++

Operator overloading is achieved by allowing operators to operate on the user defined data type with the same manner as that of built-in data types. For example, plus “+” operator produces different actions in case of integers, complex numbers or strings. With the help of ***function overloading***, we can write different functions by using the same *function name* but with *different argument lists*. The functon would perform different operations depending on the argument list in the functon call. The overloaded member functions are

selected for invoking by matching the number of arguments and type of arguments. This information is known to the compiler at the compile time itself and, therefore, the selection of the appropriate function is made at the compile time only.

In both cases, the compiler is aware of the complete information regarding the type and number of operands. Hence, it is possible for the compiler to select a suitable function at compile time. This is known as ***compile-time polymorphism***. It is also termed as ***static binding*** or ***early binding***.

Let us consider a program where the function name and argument list are same in both the base and derived class.

//**Program 12.1:**

```
#include<iostream.h>
#include<conio.h>
class B  //base class
{
     protected:
          int n;
     public:
          void enter()
          {
               cout<<"Enter a number in base class:\n";
               cin>>n;
          }
          void display()
          {
               cout<<"\nThe number in base class is: "<<n;
          }
};   //end of base class declaration
class D:public B   //derived class D
{
     private:
          int num;
     public:
```

```
            void input()
            {
        cout<<"\nEnter a number in derived class:";
        cin>>num;
    }
    void display( )
    {
        cout<<"\nThe number in derived class: "<<n;
    }
};
int main()
{
    D d;
    clrscr();
    d.enter();   //will call the enter() of base class
    d.display( );//display() of derived class will be invoked,
    getch();
    return 0;
}
```

Output of the above program will be like this:

Enter a number in base class : 6

The number in derived class : 6

But our intention is to display is :

The number in base class : 6

It has been observed that prototype of **display()** is same in both base and derived class and we cannot term it as *function overloading*. Thus *static binding* does not apply in this case. We have already used statement like **d.B::show();** in such situation (*program 10.5* of unit *Inheritance);* i.e., we used the scope resolution operator (**::**) to specify the class while invoking the functions with the derived class objects. But it would be nice if the appropriate member function could be selected while the program is running. With the help of inheritance and virtual functions, C++ determines which version of that function to call. This determination is made at run-time and is known as ***run-time polymorphism***. Here, the function is linked with a particular class much later after the compilation and thus it is also

known as ***late binding*** or ***dynamic binding***. In the following section, we will discuss how to implement virtual function to achieve run-time polymorphism.

12.4 VIRTUAL FUNCTIONS

The concept of virtual functions is different from function overloading. A ***virtual function*** is a member function that is declared within a base class and redefined by a derived class. The whole function body can be replaced with a new set of implementation in the derived class. To make a function virtual, the ***virtual*** keyword must precede the function declaration in the base class. The redefinition of the function in any derived class does not require a second use of the *virtual* keyword. The difference between a non virtual member function and a virtual member function is that the non virtual member functions are resolved at compile time whereas the virtual member functions are resolved during run-time.

The concept of *pointers to object* is prior to knowing before implementing virtual function. We have already studied the concept of pointers in earlier units. At this point, we shall discuss how class members are accessible with the help of pointers.

Pointers to Objects

A pointer can point to a class object. This is called ***object pointer***. Object pointers are useful in creating objects at run time and public members of class can be accessible by object pointers. For example, we can create pointers pointing to classes, as follows:

```
polygon *optr;
```

i.e., class name followed byan asterik (*)and then the variable name. Thus, in the above declaration, ****optr*** is a pointer to an object of class ***polygon***. To refer directly to a member of an object pointed by a pointer we can use arrow operator (**->**). Here is a program for the illustration of object pointers:

//**Program 12.2**: Demonstration of pointer to object

```
#include<iostream.h>
#include<conio.h>
class polygon
{
     protected:
     int width, height;
     public:
     void set_values (int w, int h)
     {
          width=w;
          height=h;
     }
     void display()
     {
          cout<<"Width :"<<width<<endl<<"Height : <<height;
     }
};
int main ()
{
     polygon  p;         // p is an object of type polygon
     polygon *optr = &p;// creation and initiazation of
                              // object pointer
     optr->set_values  (8,6);//object pointer accessing member
     optr->display( );//function "set_values()" and
                         //"display( )"
     getch();       // with arrow operator.
     return 0;
}
```

With the statement polygon *optr = &p;

we have created object pointer ***optr*** of type ***polygon*** and initialized it with the address of ***p*** object. We can also create the objects using pointers and ***new*** operator as follows:

polygon *optr = **new** polygon;

This statement allocates enough memory for the data members in the object of the particular class and assigns the address of the memory space to ***optr***.

Pointer to base and derived class objects

Pointers can also be used to point base or derived class object. Pointers to object of base class is a type-compatible with a pointer to object of

derived class. If we create a base class pointer, then that pointer can point to object of base as well as object of derived class.

For example, let us consider the following program :

//**Program 12.3**:

```
#include<iostream.h>
#include<conio.h>
class polygon
{
    protected:
        int width, height;
    public:
        void set_values(int w, int h)
        {
            width=w;
            height=h;
        }
};
class rectangle: public polygon //derived class rectangle
{
    public:
        int area()
        {
            return (width*height);
        }
};
class triangle: public polygon  //derived class triangle
{
    public:
        int area()
        {
            return (width*height / 2);
        }
};
int main ()
{
```

```
    rectangle r; // derived class object r
    triangle t;  // derived class object t
    clrscr();
    polygon *p1 = &r;//base class pointer pointing derived class object r
    polygon *p2 = &t;   // p2 pointing to object t of triangle class
    p1->set_values(5,6);
    p2->set_values(5,6);
    cout<<"\nArea of the rectangle is :"<<r.area()
<<endl;
    cout<<"\nArea of the triangle is :" <<t.area()
<<endl;
        getch();
        return 0;
    }
```

The output of the programm will be like this:

Area of the rectangle is : 30

Area of the triangle is : 15

In function main, we create two pointers ***p1*** and ***p2*** that point to objects of class ***polygon***. Then we assign references to ***r*** and ***t*** to these pointers. Both are valid assignment operations as because both are objects of classes derived from ***polygon***. The only limitation in using ****p1*** and ****p2*** instead of ***r*** and ***t*** is that both ****p1*** and ****p2*** are object pointers of type ***polygon*** and therefore we can only use these pointers to refer to the members that ***rectangle*** and ***triangle*** inherit from ***polygon***.

The use of pointer to objects of base class with the objects of its derived class does not allow access even to public members of a derived class. That is, it allows access only to those members inherited from the base class but not to the members which are defined to the derived class. For that reason when we call the ***area()*** members at the end of the program we have had to use directly the objects ***r*** and ***t*** instead of the pointers ****p1*** and ****p2***.

In order to use ***area()*** with the pointers to base class ***polygon***, this member should also have been declared in the class ***polygon***, and not only in its derived classes. But the problem is that, ***rectangle*** and ***triangle*** implement

different versions of ***area()***. Therefore, we cannot implement it in the base class ***polygon***. In such situations, ***virtual functions*** are necessary.

A pointer to a derived class object may be assigned to a base class pointer, and a ***virtual function*** called through the pointer. If the function is virtual and occurs both in the base class and in derived classes, then the right function will be picked up based on what the base class pointer really points at.

//**Program 12.3**: Program demonstrating the use of virtual function

```
#include<iostream.h>
#include<conio.h>
class polygon
{
    protected:
        int width, height;
    public:
        void set_values(int w, int h)
        {
            width=w;
            height=h;
        }
        virtual int area()    //virtual function
        {
            return (0);
        }
};
class rectangle: public polygon
{
    public:
        int area()    //virtual function redefined
        {
            return (width*height);
        }
};
class triangle: public polygon
```

```
{
    public:
        int area()    //virtual function redefined
        {
            return (width * height / 2);
        }
};
int main()
{
    rectangle r; //r is an object of derived class rectangle
    triangle t; //t is an object of derived class triangle
    polygon p;   //p is an object of base class polygon
    clrscr();
    polygon *p1=&r;  //pointer to a derived class object r
    polygon *p2=&t;
    polygon *p3=&p;
    p1->set_values(5,6);
    p2->set_values (5,6);
    p3->set_values (5,6);
    cout<<"Area of rectangle is:"<<p1-
>area()<<endl;
    cout<<"Area of triangle is: "<<p2-
>area()<<endl;
    cout<<"Area in polygon is:"<<p3->area()<<endl;
    getch();
    return 0;
}
```

In the above program, the three classes ***polygon***, ***rectangle*** and ***triangle*** have one common member function: ***area()***. The member function ***area()*** has been declared as ***virtual*** in the base class and it is later redefined in each derived class. The output of the program willl be like this:

Area of rectangle is : 30

Area of triangle is : 15

Area in polygon : 0

If we remove the ***virtual*** keyword from the declaration of ***area()*** within *polygon* and run the program, the result will be 0 for the three polygons instead of 30, 15 and 0. That is because instead of calling the corresponding ***area()*** function for each object (*rectangle::area()*, *triangle::area()* and *polygon::area()*, respectively), ***polygon::area()*** will be called in all cases since the calls are via a pointer of type ***polygon***. A class that declares or inherits a virtual function is called a ***polymorphic class***.

When functions are declared as virtual, the compiler adds a data member secretly to the class. This data member is referred to as a ***virtual pointer (VPTR)***. A table called ***virtual table (VTBL)*** contains pointers to all the functions that have been declared as virtual in a class, or any other classes that are inherited. Whenever a call to a virtual function is made in the C++ program, the compiler generates code to treat VPTR as the starting address of an array of pointers to functions. The function call code simply indexes into this array and calls the function located at the indexed addresses. The binding of function call always requires this dynamic indexing activities which always happens at run-time. If a call to a virtual function is made, while treating the object in question, as a member of its base class, the correct derived class function will be called. Thus, dynamic binding is achieved with the help of virtual functions.

There are some definite rules for writing virtual function. These rules are:

- The virtual functions must be members of some class.
- Object pointers should be used to access virtual function.
- A virtual function in a base class must be defined even though it may not be used.
- The prototype of the function which we declare as *virtual* in the base class must be same with all its derived class versions.
- A base pointer can point to any type of the derived object. But we cannot use a pointer to a derived class to access an object of the base class.
- Constructors cannot be virtual but destructors can be virtual.

CHECK YOUR PROGRESS

1. Choose the appropriate option for the correct answer:
   (i) Run-time polymorphim can be accomplished with the help of
   (a) operator overloading (b) function overloading
   (c) virtual function (d) friend function
   (ii) Static binding is associated with
   (a) compile-time polymorphism
   (b) run-time polymorphism
   (c) virtual function (d) none of these
   (iii) Pionter to object of base class can point
   (a) base class object (b) derived class object
   (c) both (a) and (b) (d) none of these
   (iv) Virtual functions can be accessible by
   (a) scope resolution operator
   (b) object pointer
   (c) object (d) none of these
   (v) The ability to take many forms is called .....
   (a) encapsulation (b) polymorphism
   (c) inheritance (d) none of these
2. State which of the following statements are True (T) or False F) :
   (i) The prototype of the function which we declare as *virtual* in the base class must be different with all its derived class versions.
   (ii) Run-time polymorphism can be achieved only when a virtual function is accessed through a pointer to the base class.
   (iii) Functions and operator overloading are examples of compile-time polymorphism.

12.5 PURE VIRTUAL FUNCTIONS

Generally, we declare a virtual function inside a base class and redefine it in the derived classes. In many situations, there can be no meaningful definition of a virtual function within a base class. Most of the times, the idea behind declaring a function virtual (in the base class), is to stop its execution.Then the question arises why should we define virtual functions? This leads to the idea of pure virtual functions.

For example, in the previous program *12.3*, we have defined a virtual function *area()* within the base class *polygon* . We have also created objects of *polygon* class and made a call to its own *area()* function with object pointer. As the function has minimal functionality, we could leave that *area()* member function without any definition in the base class. This can be done by appending =0 (equal to zero) to the function declaration as follows:

```
virtual int area( ) = 0;
```

Such functions are called pure virtual functions. The general form of declaring a pure virtual function is:

```
virtual return_type function_name(parameter_list) = 0;
```

A ***pure virtual function*** is a virtual function that has no definition within the base class. It only serves as a placeholder. In such cases, the compiler requires each derived class to either define the function or redeclare it as pure virtual function. A class containing pure virtual functions cannot be used to declare objects of its own. Such classes are known as ***abstract base class***. As stated earlier, when a class is not used for creating objects then it is called *abstract class or abstract base class*, similarly, a class containing pure virtual functions cannot be used for creating objects. A class that cannot instantiate objects is not useless. We can create pointers to it and take advantage of all its polymorphic abilities. Let us examine the working of pure virtual functions with an example:

//**Program 12.4**: Demonstration of pure virtual function

```
#include<iostream.h>
#include<conio.h>
class polygon
{
    protected:
        int width, height;
    public:
        void set_values(int w, int h){
            width=w;
            height=h;
        }
        virtual int area() = 0;//pure virtual function
};
class rectangle: public polygon
{
    public:
        int area()
        {
            return (width*height);
        }
};
class triangle: public polygon{
        public:
            int area()
            {
                return (width * height / 2);
            }
};
int main()
{
    rectangle r; //r is an object of derived class rectangle
    triangle t; //t is an object of derived class triangle
    clrscr();
```

```
    polygon *p1=&r; //p1 points to object r
    polygon *p2=&t; //p2 points to object t
    p1->set_values(5,6);
    p2->set_values (5,6);
    cout<<"Area of rectangle is:"<<p1->area()<<endl;
    cout<<"Area of the triangle is:"<<p2-
>area()<<endl;
    getch();
    return 0;
}
```

The output will be like this:

Area of the rectangle is : 30

Area of the rectangle is : 15

We can observe that, here we refer to objects of different but related classes using a unique type of pointer (polygon *p1,*p2). In the *main()* function, if we try to create object of *polygon* class with statement like **polygon p;** then the compiler will give error message of the following type:

Error: Cannot create instance of abstract class ‘polygon’.

We should remember that when a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile time error will occur.

Virtual function and dynamic allocation of objects

Virtual member function can also be implemented with dynamically allocated objects. Let us demonstrate the same example with objects that are dynamicalled allocated.

/***Program 12.5**: Demonstration of pure virtual function and dynamically allocated object */

```
#include<iostream.h>
#include<conio.h>
class polygon
{
```

```
    protected:
        int width, height;
    public:
        void set_values(int w, int h)
        {
            width=w;
            height=h;
        }
        virtual int area()=0;   //pure virtual function
};
class rectangle: public polygon
{
    public:
        int area()
        {
            return (width*height);
        }
};
    class triangle: public polygon
{
    public:
        int area()
        {
            return (width * height / 2);
        }
};
int main()
{
    polygon *p1=new rectangle;
    polygon *p2=new triangle;
    clrscr();
    p1->set_values(5,6);
    p2->set_values (5,6);
    cout<<"Area of rectangle is:"<<p1->area()<<endl;
    cout<<"Area of triangle is:"<<p2->area()<<endl;
    delete p1;
```

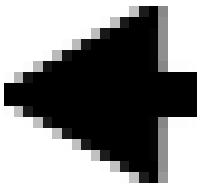
```
    delete p2;
    getch();
    return 0;
}
```

In the main() function, we have used the following statements:

polygon * p1= new rectangle;

polygon * p2= new triangle;

Here the pointer *p1* and *p2* are declared being of type pointer to *polygon* but the objects dynamically allocated have been declared having the derived class type directly.



CHECK YOUR PROGRESS

3. Choose the appropriate option for the corect answer:
   (i) Dynamic binding is done using the keyword
   (a) static (b) dynamic
   (c) virtual (d) abstract
   (ii) Virtual function helps us in achieving
   (a) run-time polymorphism
   (b) compile-time polymorphism
   (c) both (a) and (b) (d) none of these
   (iii) A base class which is not used for object creation is called
   (a) abstract class (b) derived class
   (c) virtual class (d) none of these
   (iv) The function in the statement *virtual show()=0;* is a
   (a) virtual function (b) pure member function
   (c) friend function (d) pure virtual function
   (v) A ..... pointer can point to ..... object
   (a) derived class, base class
   (b) void, NULL (c) base class, derived class
   (d) none of these
4. State which of the following statements are True(T) or False(F):
   (i) Class containg pure virtual function can instantiate objects of its own.

(ii) Pointers to objects of a base class type are compatible with the pointer to objects of a derived class.

(iii) A virtual function is a member function that expects to be overridden in a derived class.

12.6 LET US SUM UP

- Polymorphism is the ability to use an operator or function in different ways. *Poly*, referring to many, signifies the many uses of these operators and functions. C++ supports polymorphism both at run-time and at compile-time.
- The use of overloaded funcAE1 `tions is an example of compile-time poly- morphism. Run-time polymorphism can be achieved through the use of pointer to base class and *virtual functions.*
- Object pointers are useful in creating objects at run-time. It can be used to access the public members of an object along with an arrow operator.
- A base class pointer may address an object of its own class or an object of any class derived from the base class.
- A pure virtual function is a virtual function declared in a base class that has no definition.
- A class containing pure virtual functions cannot be used to declare any objects of its own. Such classes are called abstract class or abstract base class.

12.7 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education

12.8 ANSWERS TO CHECK YOUR PROGRESS

1. (i) (c) virtual function (ii) (a) compile-time polymorphism
   (iii) (c) both (a) and (b) (iv) (b) object pointer
   (v) (b) polymorphism
2. (i) False (ii) True (iii) True
3. (i) (c)virtual (ii) (a) run-time polymorphism
   (iii) (a)abstract class (iv) (d)pure virtual function
   (v) (c)base class, derived class
4. (i) False (ii) True (iii) True

12.9 MODEL QUESTIONS

1. What is polymorphism? What are the different types of polymprphism in C++?
2. How is polymorphism achieved
   (i) at compile time (ii) at run-time
3. What is a virtual function?
4. Describe the rules for declaring virtual functions.
5. How can C++ achieve dynamic binding?
6. What are pointer to base and derived classes?
7. Write a C++ program to demostrate the use of abstract classes.
8. Find the error in the following declaration:

```
class Base
{
    public:
    virtual void display()=0;
};
void main()
{
    Base b;
}
```

9. What are virtual and pure virtual functions? Use this concept to calculate the area of a square and a rectangle.

****

UNIT 13 : FILE HANDLING

UNIT STRUCTURE

13.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- describe file classes in C++
- open and close a file
- learn about different file modes
- define file pointers and them in programming use
- perform functions for input/output operations
- use some additional file handling features

INTRODUCTION

In the previous units we have come across many useful features of the C++ language like operators, arrays, pointers, functions etc, which provide the basic programming platform for programmers. Also, we were introduced to some new Object Oriented features and applications of the C++ language like classes, operator overloading, inheritance, polymorphism etc. In this unit, we will deal with a very important feature of this language which deals with handling files and manipulating them.

Files are a means to store data in a storage device. When we have to deal with handling enormous volumes of data, we use several external storage devices like floppy disks, hard disks etc. In the same way, we can write programs to perform these file manipulation tasks by using C++ file handling features. C++ file handling provides a mechanism to store the output of a program in a file and read from a file on the disk. The file operations of C++ are very much similar to the console oriented input and output operations where a file stream acts as the interface between the program and the file.

13.2 FILE CLASSES

File processing in C++ is very similar to ordinary interactive input and output because the same kind of stream objects are used. Input from a file is managed by an `ifstream` object the same way that input from a keyboard is managed by the `istream` object `cin`. Similarly, output to a file is managed by an `ofstream` object the same way that output to the monitor or printer is managed by the `ostream` object `cout`. The only difference is that `ifstream` and `ofstream` objects have to be declared explicitly and initialized with the external name of the file which they manage. In other words, as we have been using `<iostream>` header file which provide functions `cin` and `cout` to take input from console and write output to a console respectively, we introduce one more header file `<fstream>` which provides data types or classes (ifstream, ofstream, fstream) to read from a file and write to a file.

Table 13.1 : Stream classes

Data type	Description
ofstream	This data type represents the output file stream and is used to create files and to write information to files.
ifstream	This data type represents the input file stream and is used to read information from files.
fstream	This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means that it can create files, write information to files, and read information from files.

These classes, designed to manage the disk files, are declared in **fstream** and therefore we have to **`#include`** the **`<fstream>`** header file that defines these classes in any program that uses files. These classes, designed to manage the disk files, are declared in **fstream** and therefore we have to **`#include`** the **`<fstream>`** header file that defines these classes in any program that uses files.

13.3 OPENING AND CLOSING A FILE

In C++, a file is opened by linking it to a stream. There are three types of streams: **input**, **output** and **input/output**. To open an input stream we must declare the stream to be of class **ifstream**. To open an output stream, it must be declared as class **ofstream**. A stream that will be performing both input and output operations must be declared as class **fstream**. For example, the following fragment creates one input stream, one output stream and one stream that is capable of both input and output.

```
ifstream in;

ofstream out;

fstream both;
```

Once a stream has been created, the next step is to associate a file with it, and thereafter the file is available (opened) for processing.

Opening of files can be achieved in the following two ways:

1. Using the constructor function of the stream class.
2. Using the function **open()**.

The first method is preferred when a single file is used with a stream. However, for managing multiple files with the same stream, the second method is preferred. Let us discuss each of these methods one by one.

Opening a File Using Constructors

We know that a constructor of a class initializes its object when it (the object) is created. Simillarly, constructors of the stream classes (ifstream, ofstream, or fstream) are used to initialize file stream objects with the filenames passed to them, as given below:

To open a file named myfile as an input file (i.e., data will be needed from it and no other operation like writing or modifying would take place on the file), we shall create a file stream object of input type i.e., ifstream type like:

```
ifstream fin("myfile", ios::in);
```

The above statement creates an object, **fin**, of input file stream. After creating the ifstream object **fin**, the file **myfile** is opened and attached to the input stream, **fin**.

To read from this file, this stream object will be used using the operator (“>>”) as,

13.4 OPENING AND CLOSING A FILE

In C++, a file is opened by linking it to a stream. There are three types of streams: **input**, **output** and **input/output**. To open an input stream we must declare the stream to be of class **ifstream**. To open an output stream, it must be declared as class **ofstream**. A stream that will be performing both input and output operations must be declared as class **fstream**. For example, the following fragment creates one input stream, one output stream and one stream that is capable of both input and output.

ifstream in;

ofstream out;

fstream both;

Once a stream has been created, the next step is to associate a file with it, and thereafter the file is available (opened) for processing.

Opening of files can be achieved in the following two ways:

1. Using the constructor function of the stream class.
2. Using the function **open()**.

The first method is preferred when a single file is used with a stream. However, for managing multiple files with the same stream, the second method is preferred. Let us discuss each of these methods one by one.

Opening a File Using Constructors

We know that a constructor of a class initializes its object when it (the object) is created. Simillarly, constructors of the stream classes (ifstream, ofstream, or fstream) are used to initialize file stream objects with the filenames passed to them, as given below:

To open a file named myfile as an input file (i.e., data will be needed from it and no other operation like writing or modifying would take place on the file), we shall create a file stream object of input type i.e., ifstream type like:

```
ifstream fin("myfile", ios::in);
```

The above statement creates an object, **fin**, of input file stream. After creating the ifstream object **fin**, the file **myfile** is opened and attached to the input stream, **fin**.

To read from this file, this stream object will be used using the operator **(">>") as,**

```
char ch;fin>>ch; // read a character from the
filefloat amt;fin>>amt;// read a floating-point
number form the file
```

Similarly, opening an output file (on which there is no operation except writing) can be accomplish by –

1. Creating **ofstream** object to manage the output stream.
2. Associating that object with a particular file.

```
ofstream fout("secret" ios::out);//create ofstream...
// ..object named fout
```

This would create an output stream object named **fout** and attach the file **secret** with it.

Now, to write something to it, we use the << operator like,

```
int code=2193;fout<<code<<"xyz";
```

The connections with a file are closed automatically when the input and the output stream objects expire i.e., when they go out of scope. We may close a connection with a file explicitly by using the close() method:

```
fin.close(); // close input connection to
```

```
filefout.close();// close output connection to file
```

Closing such a connection does not eliminate the stream; it just disconnects it from the file. For example, after the above statements, the streams **fin** and **fout** still exist along with the buffers they manage. We may later reconnect the stream to the same file or to another file, if needed. Closing a file flushes the buffer which means that the data remaining in the buffer (input or output stream) is moved out of it. For example, when an input file's connection is closed, the data is moved from the input buffer to the program and when an output file's connection is closed, the data is moved from the output buffer to the disk file.

Opening Files Using Open() Function

There may be situations requiring a program to open more than one file. The strategy for opening multiple files depends upon how they will be used. If the situation requires simultaneous processing of two files, then we need to create a separate stream for each file. However, if the situation demands sequential processing of files (i.e., processing them one by one), then we can open a single stream and associate it with each file in turn. To use this approach, we declare a stream object without initializing it, then use a second statement to associate the stream with a file. For example,

```
ifstream fin;//create an input stream
fin.open("Master.dat", ios::in);
//associate fin with Master.dat:
// process Master.dat
fin.close();//terminate association with Master.dat
fin.open("Tran.dat", ios::in);
//associate fin with Tran.dat:
process Tran.dat
fin.close();
// terminate association
```

The Concept of File Modes

The **filemode** describes how a file is to be used: to read from it, to write to it, to append it, and so on. When we associate a stream with a file, either by initializing a file stream object with a file name or by using the **open()** method, we can provide a second argument specifying the file mode, as

mentioned below:

```
stream_object.open("filename", (filemode));
```

The second method argument of **open()**, the filemode, is of type int, and we may choose one from several constants defined in the **ios class**.

List of File Modes in C++

Following table lists the filemodes available in C++ with their meanings:

Table 14.2 : Different file modes

Constant	Meaning	Stream Type
ios :: in	Opens file for reading, i.e., in input mode.	ifstream
ios :: out	Opens file for writing, i.e., in output mode.	ofstream
ios :: ate	This also opens the file in ios::trunc mode, by default.This means an existing file is truncated when opened, i.e., its previous contents are discarded.	ofstream
ifstream	This seeks to end-of-file upon opening of the file. I/O operations can still occur anywhere within the file.	
ios :: app	This causes all output to that file to be appended to the end. This value can be used only with files capable of output	ofstream
ios :: trunc	This value causes the contents of a pre-existing file by the same name to be destroyed and truncates the file to zero length.	ofstream
ios :: nocreate	This cause the open() function to fail if the file does not already exist. It will not create a new file with that name.	ofstream
ios :: noreplace	This causes the open() function to fail if the file already exists.This is used when you want to create a new file and at the same time.	ofstream

	ios :: binary	This causes a file to be opened in binary mode. By default, files are opened in text mode. When a file is opened in text mode, various character translations may take place, such as the conversion of carriage-return into newlines.However, no such character translations occur in file opened in binary mode.	ofstream ifstream

The **fstream** class does not provide a mode by default and, therefore, one must specify the mode explicitly when using an object of the **fstream** class.

Both **ios::ate** and **ios::app** place us at the end of the file just opened. The difference between the two is that the **ios::app** mode allows us to add data to the end of the file only, while the **ios::ate** mode lets us write data anywhere in the file.

We may combine two or more filemode constants using the C++ **bitwise OR** operator (symbol |). For example, the following statement:

```
ofstream fout;fout.open("Master", ios::app |
ios::nocreate);
```

will open a file in the append mode if the file exists and will abandon the file opening operation if the file does not exist.

To open a binary file, we need to specify ios :: binary along with the file mode, e.g.,

```
fout.open("Master", ios::app | ios::binary);
```

or,

```
fout.open("Main", ios::out | ios::nocreate |
ios::binary);
```

Closing a File in C++

As already mentioned, a file is closed by disconnecting it with the stream it is associated with. The **close()** function accomplishes this task and it takes the following general form:

```
stream_object.close();
```

For example, if a file Master is connected with an **ofstream** object **fout**, its connections with the stream **fout** can be terminated by the following statement:

```
fout.close();
```

C++ Opening and Closing a File Example

Here is an example for the complete understanding on:

- how to open a file in C++ ?
- how to close a file in C++ ?

Program 13.1: Open a file to store/retrieve information to/from it, and close that file after storing/retrieving the information to/from it.

```
#include<conio.h>
#include<string.h>
#include<stdio.h>
#include<fstream.h>
#include<stdlib.h>
void main()
{
        ofstream fout;
        ifstream  fin;
        char fname[20];
        char rec[80], ch;
        clrscr();
        cout<<"Enter file name: ";
        cin.get(fname, 20);
        fout.open(fname, ios::out);
        if(!fout)
        {
            cout<<"Error in opening the file "<<fname;
            getch();
            exit(1);
        }
        cin.get(ch);
        cout<<"\nEnter a line to store in the file:\n";
        cin.get(rec, 80);
        fout<<rec<<"\n";
        cout<<"\nThe line is stored successfully.";
        cout<<"\nPress any key to see...\n";
        getch();
        fout.close();
        fin.open(fname, ios::in);
        if(!fin)
        {
        cout<<"Error in opening the file "<<fname;
        cout<<"\nPress any key to exit...";
        getch();
        exit(2);
}
```

```
cin.get(ch);
fin.get(rec, 80);
cout<<"\nThe file contains:\n";
cout<<rec;
cout<<"\n\nPress any key to exit...\n";
fin.close();
getch();
}
```

CHECK YOUR PROGRESS

1. Fill in the blanks

(a) Output to a file is managed by an _____ object.

(b) In C++, a file is opened by linking it to a _____.

(c) A stream that will be performing both _____ and _____ operations must be declared as class _____.

(d) The _____ with a file are closed automatically when the input and the output _____ objects expire.

(e) The _____ operator is used to combine two or more filemode constants.

FILE POINTERS AND THEIR MANIPULATION

The C++ input and output system manages two integer values associated with a file.

These are:

- **get pointer** – specifies the location in a file where the next read operation will occur.
- **put pointer** – specifies the location in a file where the next write operation will occur. In other words, these pointers indicate the current positions for read and write operations, respectively. Each time an input or an output operation takes place, the pointers are automatically advanced sequentially.

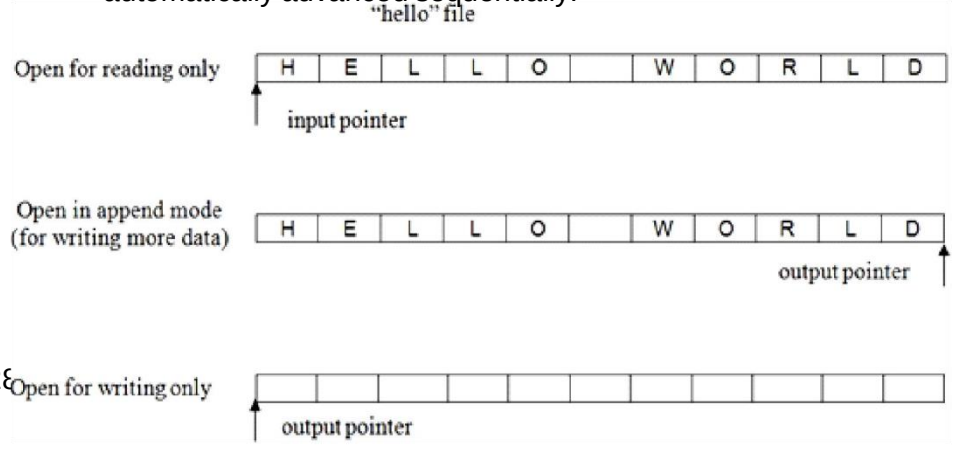


Fig 13.1: Action on file pointers while opening a file

Functions for manipulation of file pointers

The read operation from a file involves the **get** pointer. It points to a specific location in the file and the reading starts from that location. Then, the **get** pointer keeps moving forward which lets us read the entire file. Similarly, we can start writing to a location where **put** pointer is currently pointing. The **get** and **put** are known as file position pointers and these pointers can be manipulated or repositioned to allow random access of the file. The functions which manipulate file pointers are shown in Table 13.1:

Table 13.3 : File pointer

Function	Description
seekg()	Moves the **get** pointer to a specific location in the file
seekp()	Moves the **put** pointer to a specific location in the file
tellg()	Returns the current position of the **get** pointer
tellp()	Returns the current position of the **put** pointer

seekg()

Sets the position of the get pointer. The **get** pointer determines the next location to be read in the source associated to the stream. The function **`seekg(n, ref_pos)`** takes two arguments:

1)) **`n`** denotes the number of bytes to move and **`ref_pos`** denotes the reference position relative to which the pointer moves.

2)) **`ref_pos`** can take one of the three constants:

- **`ios:: beg`** moves the get pointer `n` bytes from the beginning of the file,
- **`ios:: end`** moves the get pointer `n` bytes from the end of the file
- **`ios:: cur`** moves the get pointer `n` bytes from the current position. If we don't specify the second argument, then **`ios:: beg`** is the default reference position.

Program 13.2: Read a file into memory

```
#include<fstream.h>
#include<iostream.h>
int main (int argc, char** argv)
{
     fstream myFile("test.txt", ios::in | ios::out |
        ios::trunc);
     myFile << "Hello World";
     myFile.seekg(6, ios::beg);
     char buffer[6];
     myFile.read(buffer, 5);
     buffer[5] = 0;
     cout << buffer << endl;
     myFile.close();
}
```

In the above example, we open a new file for input/output discarding any current content in the file. Adding the characters “Hello World” to the file, we seek to read 6 characters from the beginning of the file. Then we read the next 5 characters from the file into a buffer and end the buffer with a null terminating character. Finally, we output the contents read from the file and close it.

seekp()

The behaviour of `seekp(n, ref_pos)` is same as that of `seekg()`.The `seekp` method changes the location of a stream object’s file pointer for output (put or write.) In most cases, `seekp` also changes the location of a stream object’s file pointer for input (get or read).

Program 13.3: Read a file into memory

```
#include <fstream.h>
int main()
{
 long  pos;
 ofstream outfile;
 outfile.open("test.txt");
 outfile.write("This is an apple",16);
 pos=outfile.tellp();
 outfile.seekp(pos-7);
 outfile.write(" sam",4);
 outfile.close();
 return 0;
}
```

In this example, `seekp` is used to move the **put pointer** back to a position 7 characters before the end of the first output operation.

The final content of the file shall be:

```
This is a sample
```

tellg()

The tellg() function is used with input streams, and returns the current **get** position of the pointer in the stream.

Syntax: pos_type tellg();

It has no parameters and return a value of the member type pos_type, which is an integer data type representing the current position of the get stream pointer.

tellp()

Returns the absolute position of the **put** pointer. The put pointer determines the location in the output sequence where the next output operation is going to take place.

Syntax: pos_type tellp();

The tellp() function is used with output streams, and returns the current **put** position of the pointer in the stream. It has no parameters and return a value of the member type pos_type, which is an integer data type representing the current position of the put stream pointer.

Program 13.4: To demonstrate example of tellg() and tellp() function

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
    fstream file;
    //open file sample.txt in and Write mode
    file.open("sample.txt",ios::out);
    if(!file)
    {
        cout<<"Error in creating file!!!";
        return 0;
    }
    //write A to Z
file<<"ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    //print the position
   cout<<"Current position is: "<<file.tellp()<<endl;
    file.close();
    //again open file in read mode
```

```
    file.open("sample.txt",ios::in);
    if(!file)
    {
        cout<<"Error in opening file!!!";
        return 0;
    }
    cout<<"After opening file position is:"
     <<file.tellg()<<endl;
    //read characters until end of file is not found
    char ch;
    while(!file.eof())
    {
           cout<<"At position: "<<file.tellg();
           //current position
           file>>ch;   //read character from file
           cout<<" Character \""<<ch<<"\""<<endl;
    }
    //close the file
    file.close();
    return 0;
}
```

The tellg() and tellp() functions can be used to find out the current position of the get and put file pointers respectively in a file.

FUNCTIONS FOR INPUT AND OUTPUT OPERATIONS

We have seen the use of **cin** and **cout** which are the I/O operators that give us formatting control over the input and output, but these are not character I/O functions. We have also seen the file stream classes that support a number of member functions for performing the input and output operations on files. Some functions like **put()** and **get()** are designed for handling a single character at a time, and some like **write()** and **read()** are designed for writing and reading blocks of binary data.

The functions, put() and get(), which allow reading/writing character by character are called **character I/O functions.**

put()

put() function sends one character at a time to the output stream, where an output stream can be a standard output stream object or user-defined output stream object

Program 13.5:

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<fstream.h>
void main()
{
      clrscr();
      char string[50];
      cout<<"\n Enter a string to write in a file ";
      gets(string);
      fstream FILE;
      FILE.open("MYTEXT.TXT",ios::app);
      for(int i=0;string[i]!='\0';i++)
      {
            FILE.put(string[i]);
      }
      FILE.close();
      getch();
}
```

In the above program **put()** function is used with user-defined output stream object "FILE" which represents a disk file "MYTEXT.TXT".

get()

The **get()** inputs a single character from the standard input device (by default it is keyboard). ***Syntax***: device.get(char_variable); The device can be any standard input device. If we want to get input from a keyboard then we should use **cin** as the device. Because, most of the computers consider the keyboard as the standard input device, **stdin.**

```
char ch;

cin.get(ch);
```

The **get()** function is a buffered input function. When we type in, data does not go into our program unless we hit the Enter key. **get()** function receives one character, including white space, at a time from the input stream. An input stream can be a standard input stream object or user defined stream object of the istream class.

Program 13.6:

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<fstream.h>
#include<ctype.h>
void main()
```

```
{
      ifstream infile;
      infile.open("PARA.TXT");
      char ch;
      int count=0;
      while(infile)
      {
            infile.get(ch);
            if(isdigit(ch))
            count++;
      }
      infile.close();
      cout<<"\n Total digits = "<<count;
}
```

Another way to read and write blocks of binary data is to use C++'s **read()** and **write()** functions. Their prototypes are:

istream &read(char **buf*, streamsize *num*);

ostream &write(const char **buf*, streamsize *num*);

The **read()** function reads ***num*** characters from the invoking stream and puts them in the buffer pointed to by ***buf***. The **write()** function writes ***num*** characters to the invoking stream from the buffer pointed to by ***buf***

Program 13.7: Writing data using write()

```
#include<fstream.h>
#include<conio.h>
class Student
{
       int roll;
       char name[25];
       float marks;
       void getdata()
       {
              cout<<"\n\nEnter Roll : ";
              cin>>roll;
              cout<<"\nEnter Name : ";
              cin>>name;
              cout<<"\nEnter Marks : ";
              cin>>marks;
       }
       public:
       void AddRecord()
       {
              fstream f;
              Student Stu;
```

```
 f.open("Student.dat",ios::app|ios::binary);
              Stu.getdata();
              f.write((char *) &Stu, sizeof(Stu));
              f.close();
       }
};
void main()
{
       Student S;
       char ch='n';
       do
       {
              S.AddRecord();
              cout<<"\nAny more data(y/n): ";
              ch = getche();
       }
       while(ch=='y'  || ch=='Y');
       cout<<"\nData written successfully...";
}
```

Program 13.8: Reading data using read()

```
#include<fstream.h>
#include<conio.h>
class Student
{
  int roll;
  char name[25];
  float marks;
  void putdata()
  {
cout<<"\n\t"<<roll<<"\t"<<name<<"\t"<<marks;    }
public:
void Display()
{
fstream f;
StudentStu;
f.open("Student.dat",ios::in|ios::binary);
cout<<"\n\tRoll\tName\tMarks\n";
while((f.read((char*)&Stu,sizeof(Stu)))!=NULL)
Stu.putdata();
f.close();
}
};
void main()
{ Student S;
  S.Display();
}
```

EXCEPTION HANDLING

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.
Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch:** A program catches an exception with an exception handler at the place in a program where we want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It is followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
   // protected code
}catch(Exception-Name  e1)
{
   // catch block
}catch(Exception-Name  e2)
{
   // catch block
}catch(Exception-Name  eN)
{
   // catch block
}
```

We may list down multiple **catch** statements to catch different type of exceptions in case our **try** block raises more than one exception in different situations.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using throw

statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.
Following is an example of throwing an exception when dividing by zero condition occurs

```
double division(int a, int b)
{
   if(b==0)
   {
      throw "Division by zero condition!";
   }
   return(a/b);
}
```

Catching Exceptions

The **catch** block following the **try** block catches any exception. We can specify what type of exception we want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {
   // protected code
}catch(ExceptionName e) {
   // code to handle ExceptionName exception
}
```

The above code will catch an exception of ExceptionName type. If we want to specify a catch block handling any type of exception that is thrown in a try block, we must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows

```
try {
   // protected code
}catch(...) {
   // code to handle any exception
}
```

Program 13.9:

```
#include <iostream>
using namespace std;
int main()
{
cout<<"Start/n";
try {                    // start a try block
cout<<"Inside Try block\n";
throw 100;   //Throw an error
```

```
cout<<"This will not execute";
}
catch(int i)              //catch an error
{
cout<<"Caught an Exception. Value is"<<i;
}
return 0;
}
```

Because we are raising an exception of type **const char***, so while catching this exception, we have to use const char* in catch block.

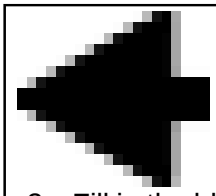
Detecting EOF

The physical contents of a file may not be precisely known. C++ provides a special function, **eof()**, that returns nonzero (TRUE) when there are no more data to be read from an input file stream, and zero (FALSE) otherwise. Returns true if the eofbit ***error state flag*** is set for the stream. This flag is set by all standard input operations when the End-of-File is reached in the sequence associated with the stream. Note that the value returned by this function depends on the last operation performed on the stream (and not on the next). Operations that attempt to read at the ***End-of-File*** fail, and thus both the **eofbit** and the **failbit** end up set. This function can be used to check whether the failure is due to reaching the *End-of-File* or to some other reason.

Program 13.10

```
#include <iostream>
using namespace std;
int main()
{
  int a, b;
  cout<<"Enter Values of a and b \n";
  cin>>a;
  cin>>b
  int x=a-b;
  try
  {
   if(x!=0)
     {
      cout<<"Result(a/x)="<<a/x<<"\n";
     }
   else
     {
      throw(x);
     }
  }
```

```
  catch(int i)
   {
    cout<<"Exception caught: x="<<x<<"\n";
   }
 cout<<"END";
return 0;
}
```



CHECK YOUR PROGRESS

2. Fill in the blanks
   (a) The get and put are known as file_____pointers.
   (b) The tellg() function is used with _____streams, and returns the current_____position of the pointer in the stream.
   (c) _____function sends one character at a time to the output stream.
   (d) The get() function is a_____input function.
   (e) _____provide a way to transfer control from one part of a program to another.
   (f) _____returns nonzero when there are no more data to be read from an input file stream.
3. State TRUE or FALSE
   (a) get pointer specifies the location in a file where the next write operation will occur
   (b) seekg() sets the position of the get pointer.
   (c) The tellp() function is used with output streams, and returns the current put position of the pointer in the stream
   (d) The get() inputs a single character from the standard input device
   (e) Exceptions can be thrown anywhere within a code block using try statements
   (f) The catch block following the throw block catches any exception

LET US SUM UP

- C++ file handling provides a mechanism to store the output of a program in a file.
- ofstream, ifstream and fstream classes are designed to manage disk files.
- There are three types of streams: input, output and input/output.
- Constructors of the stream classes are used to initialize file stream objects.
- A file is closed by disconnecting it with the stream it is associated with.

- The C++ input and output system manages two integer values associated with a file.
- The seekp() method changes the location of a stream object's file pointer for output.
- The functions, put() and get(), allow reading/writing character by character.
- The read() and write() functions read and write blocks of binary data.

13.9 FURTHER READING

- Balagurusamy, E. (2011), *Object-oriented programming with C++*, 6e. Tata McGraw-Hill Education
- Venugopal, K.R. (2013), Rajkumar, *Mastering C++*. Tata McGraw-Hill Education
- Ravichandan D. (2002), *Programming with C++*, 2e. Tata McGraw-Hill Education

ANSWERS TO CHECK YOUR PROGRESS

1

(a) ofstream.
(b) stream.
(c) input, output, fstream.
(d) connections, stream.
(e) bitwise OR.

2.

(a) position.
(b) Input, get.
(c) put().
(d) buffered.
(e) Exceptions.
(f) eof().

3.

(a) False
(b) True
(c) True
(d) True
(e) False
(f) False

MODEL QUESTIONS

1. What are Files classes in C++? Illustrate their use with examples.
2. Describe opening a file in C++ with an example.
3. Name and describe the different file modes in C++.
4. What are file pointers in C++? How are they used?
5. Illustrate the usage of seekg(), seekp(), tellg() and tellp() functions with the help of examples.
6. Differentiate between the put() and get() functions.
7. Why are read() and write() functions used.
8. What is exception handling in C++. Illustrate its use with an example.
9. How do we throw and catch an exception in C++?
10. Why is it necessary to detect EOF in C++ programs? Illustrate.

****