UNIT 1 : INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

UNIT STRUCTURE

1.1 Learning Objectives
1.2 Introduction
1.3 Benefits of OOP
1.4 OOP Languages
    1.4.1 C++
    1.4.2 Smalltalk
    1.4.3 Java
1.5 Elements of Object Oriented Programming
    1.5.1 Objects
    1.5.2 Classes
1.6 Writing, Compiling and Executing C++ programs
1.7 Let Us Sum Up
1.8 Further Reading
1.9 Answers to Check Your Progress
1.10 Model Questions

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.

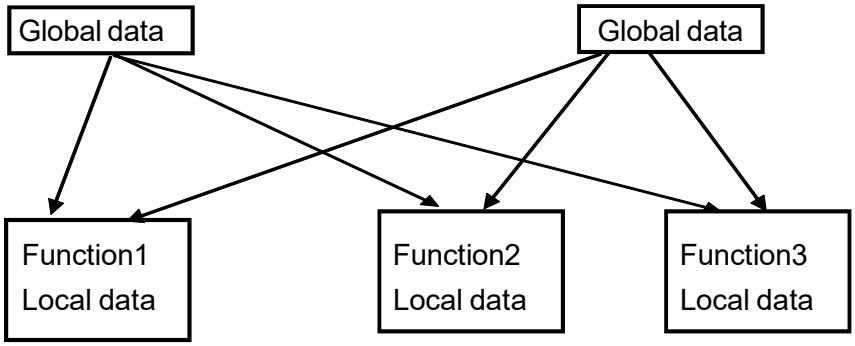


Fig. 1.1 : Relationship between data and functions

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.

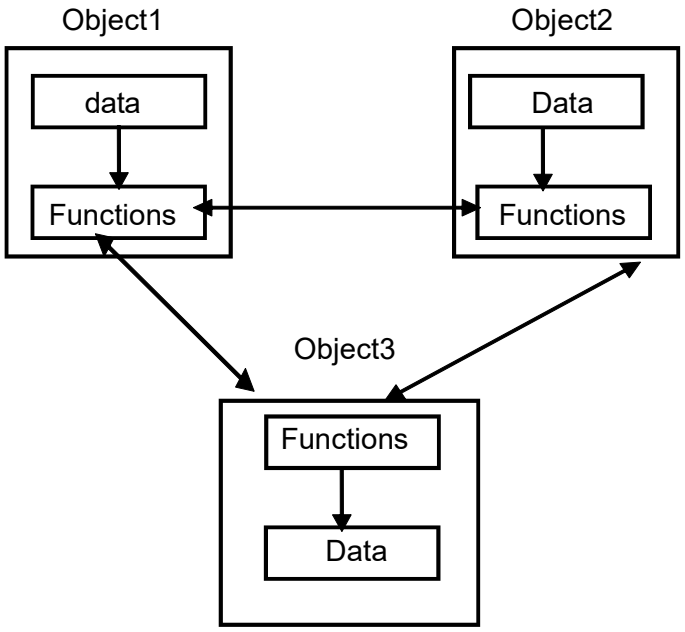


Fig. 1.2 : Relationship between data and function

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 WRITING, COMPILING AND EXECUTING 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
2.2 Introduction
2.3 Inheritance
    2.3.1 Forms of Inheritance
2.4 Defining the Derived Class
    2.4.1 Making Private Member Inheritable
2.5 Virtual Base Class
2.6 Encapsulation
2.7 Polymorphism
    2.7.1 Function Overloading
    2.7.2 Operator Overloading
2.8 Let Us Sum Up
2.9 Further Reading
2.10 Answers to Check Your Progress
2.11 Model Questions

2.1 LEARNING OBJECTIVES

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

- learn about the different features like abstraction, encapsulation, inheritance, polymorphism.
- learn the basics of 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 derivation :

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

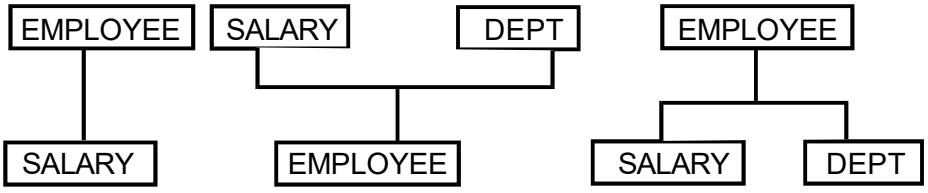
(i) Single Inheritance : A derived class with only one base class is called single inheritance.

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

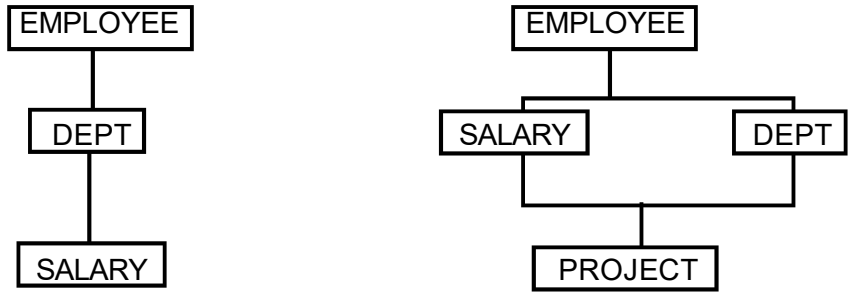
(iii) Hierarchical Inheritance : Several derived classes with only one base class is called hierarchical inheritance.

(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.

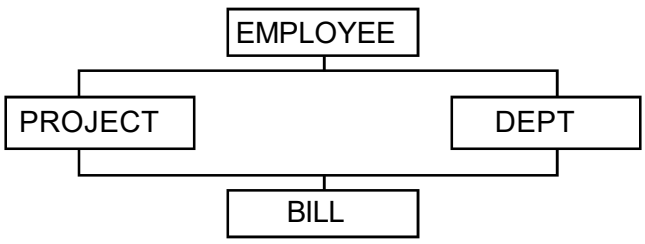


Fig 2.2 : Multiple base class

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

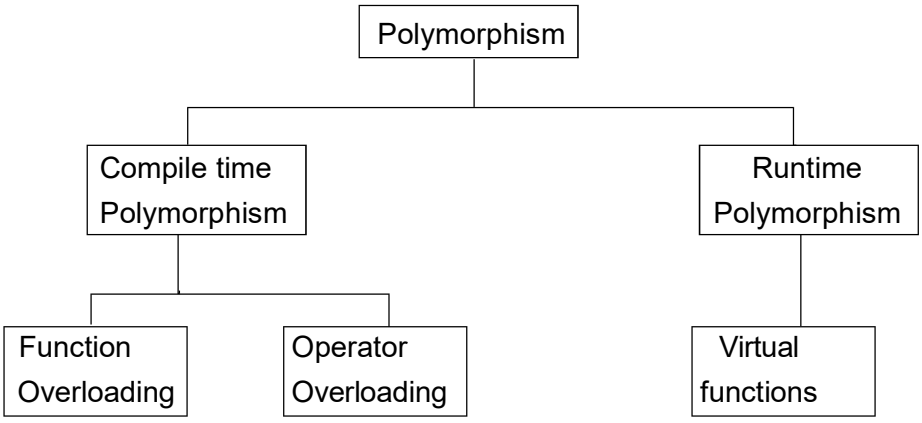


Fig. 2.3 : Forms of Polymorphism

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 : ++ --	Members access operators (., .*)
Arithmetic : + - / *	Scope resolution operator(::)
Relational : < > ==	Size operator(sizeof)
Shorthand :+= -= *= /=	Conditional operator(?:)
Stream :<< >>	

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.



2.8LET 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
3.2 Introduction
3.3 Token, Identifier and Keywords
3.4 Character Set and Symbols
3.5 Basic Data types in C++
3.6 Variables
3.7 Constants
3.8 Dynamic Initialization of Variables
3.9 Reference Variable
3.10 Streams in C++
3.11 Let Us Sum Up
3.12 Further Reading
3.13 Answers to Check Your Progress
3.14 Model Questions

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 3.2 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.*

3.9 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.

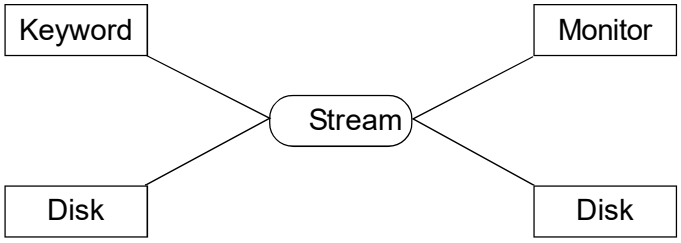


Fig. 3.1 : Streams with I/O devices

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.

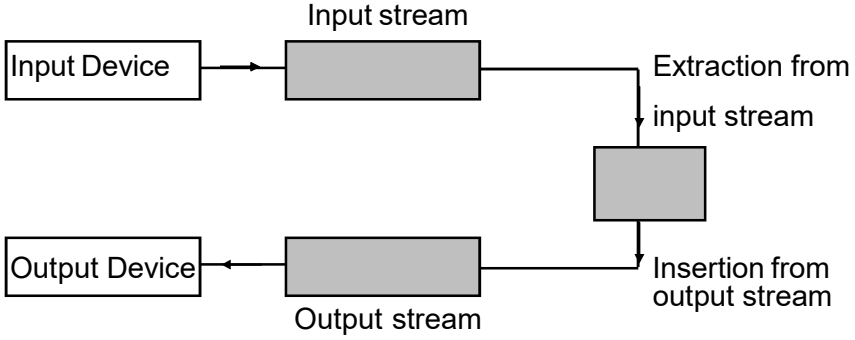


Fig. 3.2 : Input and output streams

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 :

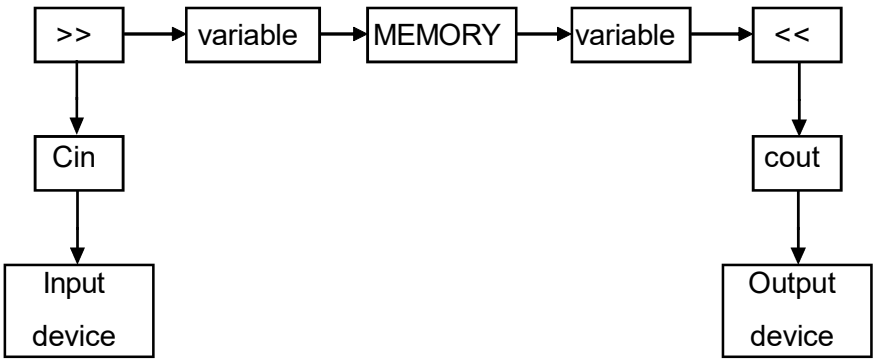


Fig. 3.3 : Working of *cin* and *cout* statement



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



3.10 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.



3.11 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



3.12 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.



3.13 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
4.2 Introduction
4.3 Operators
4.4 Types of Operators
- 4.4.1 Arithmetic Operator
- 4.4.2 Relational and Logical Operator
- 4.4.3 Assignment Operator
- 4.4.4 Increment and Decrement Operator
- 4.4.5 Bitwise Operator
- 4.4.6 Conditional Operator
- 4.4.7 Comma Operator
- 4.4.8 *sizeof* Operator
- 4.4.9 Scope Resolution Operator
- 4.4.10 Insertion and Extraction Operator
- 4.4.11 Address and Indirection Operator
- 4.4.12 Memory Management Operator

4.5 Precedence and Associativity
4.6 Manipulators
4.7 Let Us Sum Up
4.8 Further Reading
4.9 Answers To Check Your Progress
4.10 Model Questions

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*. For example, total = a + b ;
Here, a and b are operands and + is an operator.

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*/

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



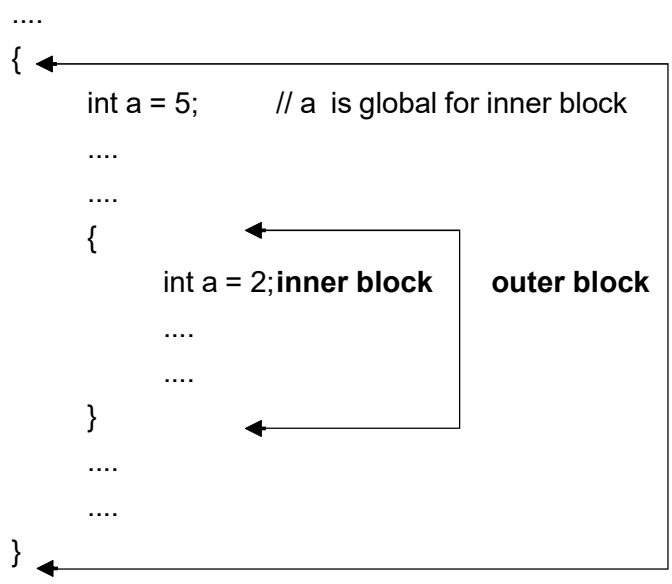
EXERCISE-1

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:



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";
```

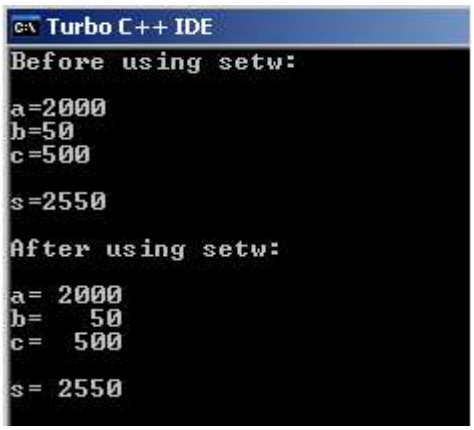
The manipulator ***setw(n)*** sets the field width to ***n*** for displaying the value of a variable. The use of these two manipulators are demonstrated in the following program where sum of three values are displayed in two different format.

// ***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

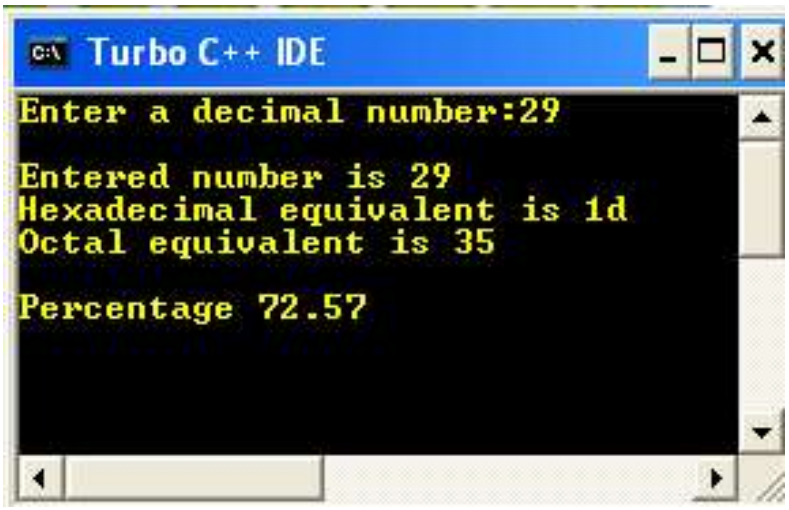
Manipulators	Function
setw(int n)	Sets the field width to *n.*
setprecision(int d)	Sets the floating point precision to *d.*
setfill(char f)	The fill character is set to character. Stored in variable *f.*
setiosflags(long f)	Sets the format flag *f* .
resetiosflags(long f)	Clears the flag indicated by *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
5.2 Introduction
5.3 Decision Making Statements
  5.3.1 *if* statements
  5.3.2 *if-else* statements
  5.3.3 *switch* statements
5.4 Loops
  5.4.1 *while* loop
  5.4.2 *do-while* loop
  5.4.3 *for* loop
5.5 Unconditional Branching Statements
  5.5.1 *break* statement
  5.5.2 *continue* statement
  5.5.3 *goto* statement
5.6 Let Us Sum Up
5.7 Further Reading
5.8 Answers to Check Your Progress
5.9 Model Questions

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:

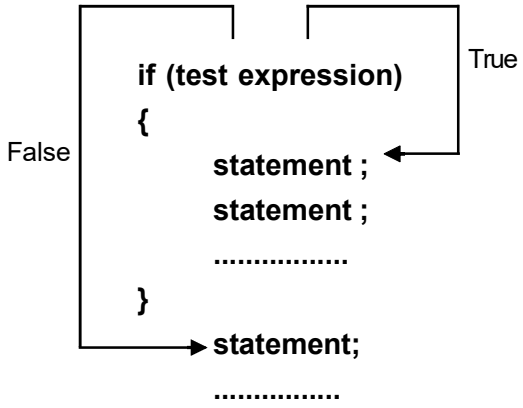
- ***if*** statement
- ***if-else*** statement
- ***switch*** statement

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.

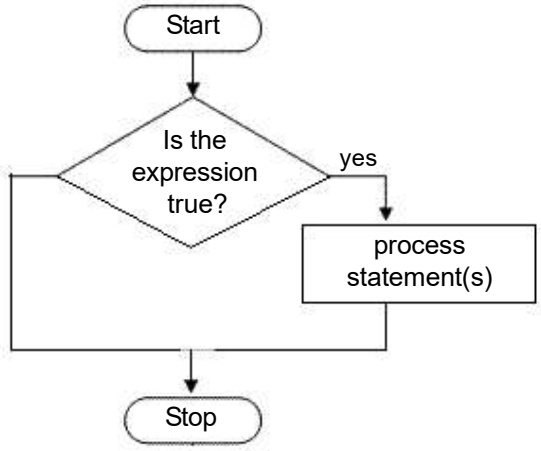


Fig. 5.1 : Flow chart of *if* statement

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:

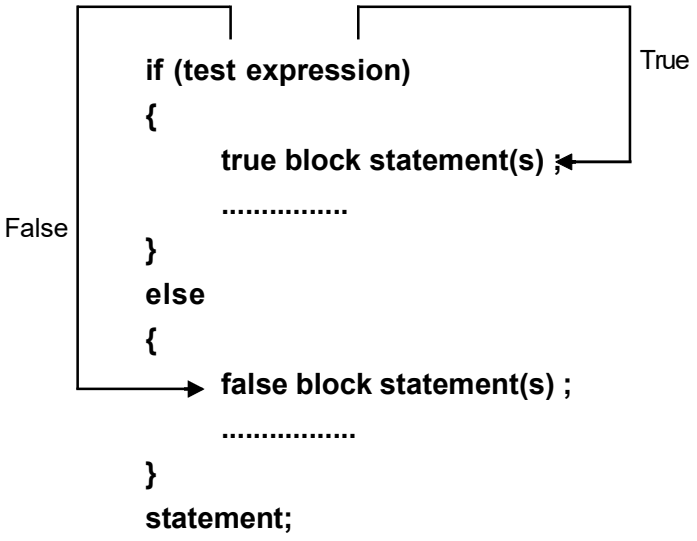
/***Program 5.2*:** Demonstration of *if* statement (more than one statements inside if statement)*/

```
#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 :

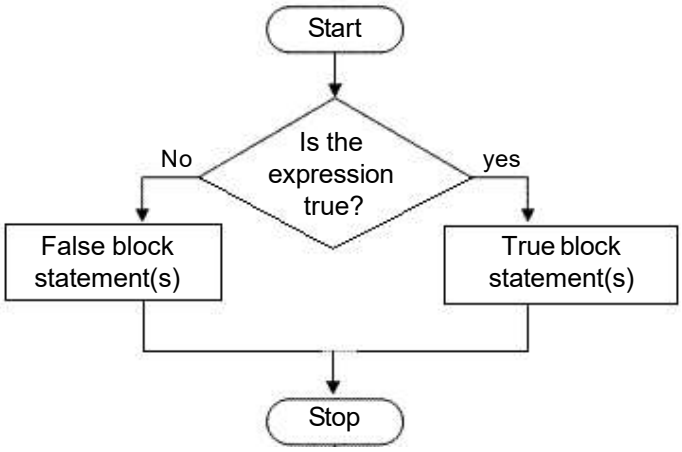


Fig. 5.2 : Flow chart of *if-else* statement

/****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

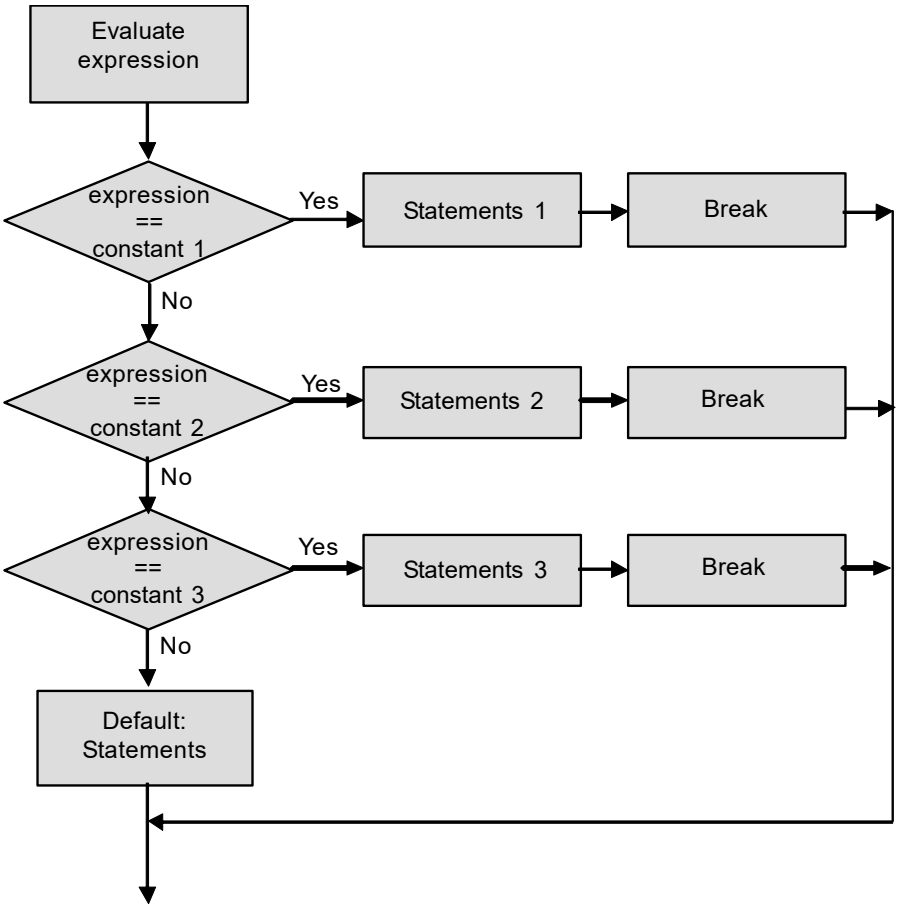


Fig. 5.3 : Flow chart of *switch* statement

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

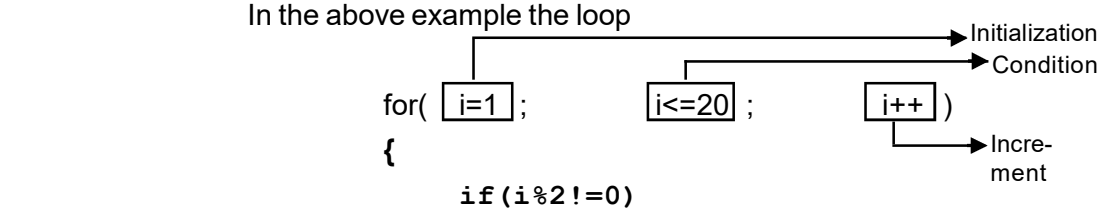
```
#include<iostream.h>
#include<conio.h>
int main()
{
   int i;
   cout<<"The odd numbers between 1 to 20
are:"<<endl;
   for(i=1;i<=20;i++)
   {
        if(i%2!=0)
            cout<<i<<"\t";
        }
        getch();
        return 0;
   }
```

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 avove 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* stetement

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
6.2 Introduction
6.3 Arrays
6.4 Pointers
6.5 Structures
6.6 Unions
6.7 Let Us Sum Up
6.8 Further Reading
6.9 Answers to Check Your Progress
6.10 Model Questions

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 discuss the concept 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

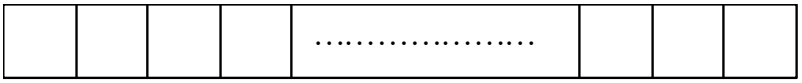
Storage Class datatype arrayname [expression];

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.



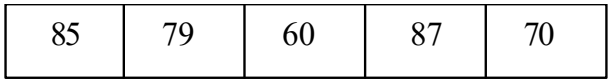
x[0] x[1] x[2] x[3] …… x[12] x[13] x[14]

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 :



marks [0] marks [1] marks [2] marks [3] marks [4]

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: :

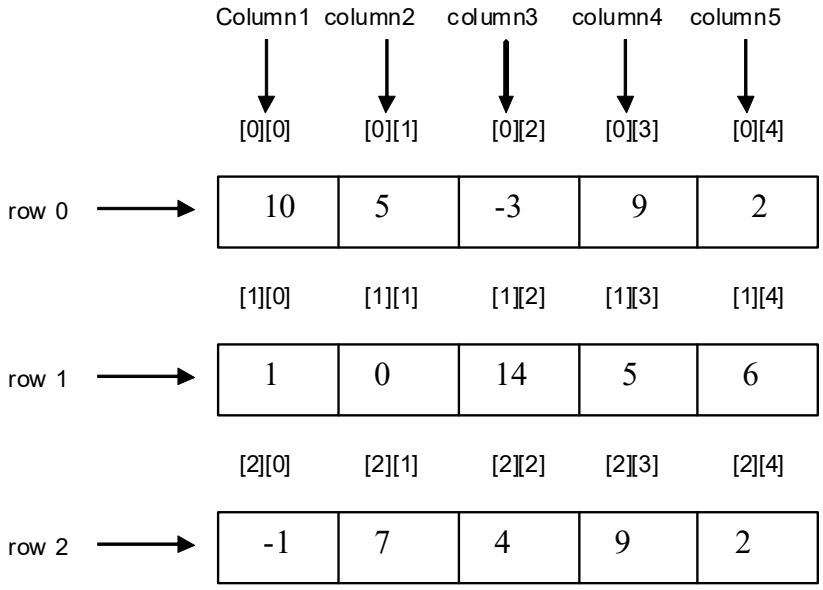


Fig. 6.3 : Matrix array after initialization

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.

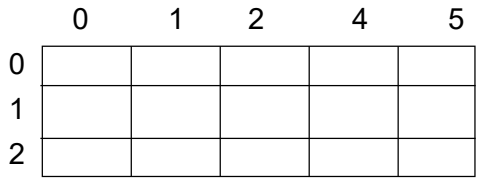


Fig. 6.5 : An array of strings representation

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 = &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

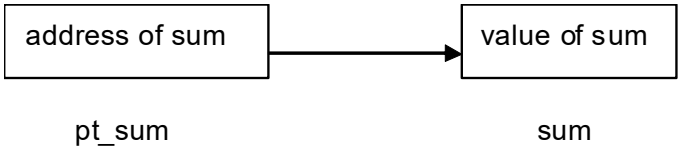


Fig. 6.6 : Relationship between pt_sum and sum

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.

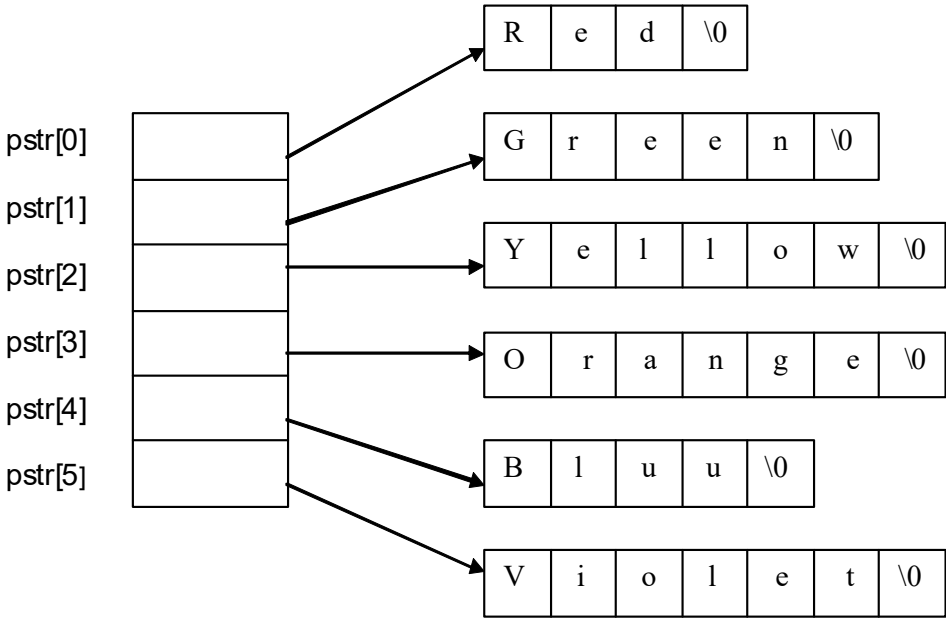


Figure 6.7 : Representation of array of pointers *char *pstr[ ]*

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 :

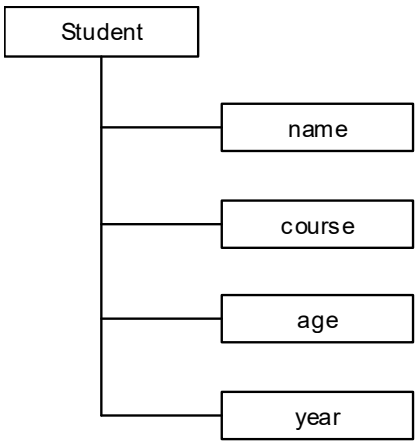


Figure 6.8 : A structure named Student

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.

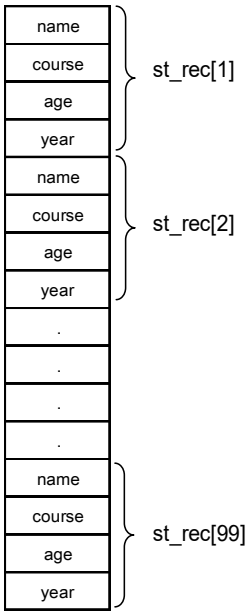


Figure 6.9 : Array of structure representation

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 Objectives
7.2 Introduction
7.3 Library Function
7.4 I/O Functions
7.5 Unformatted I/O Functions
7.6 User Defined Function
7.7 Key Terms Related to Function
7.8 Rules for Writing Function
7.9 Syntax for Function Declaration
7.10 Category of Function
7.11 Mathematical Library Function
7.12 Inline Function
7.13 Function Overloading
7.14 Default Argument
7.15 Macros or Macro Function
7.16 Let Us Sum Up
7.17 Further Reading
7.18 Answers to Check Your Progress
7.19 Model Questions

7.1 LEARNING OBJECTIVE

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

- 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

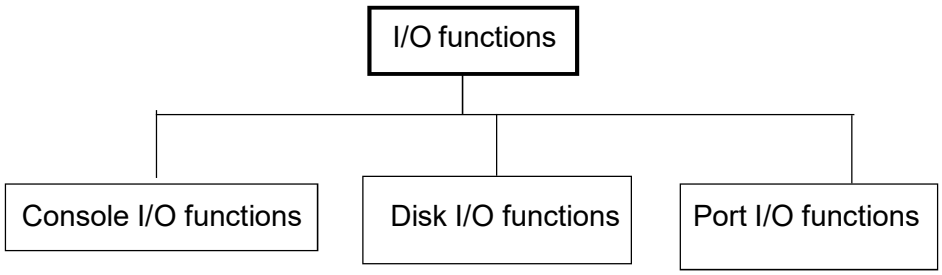
7.4 I/O FUNCTIONS

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();
}
```

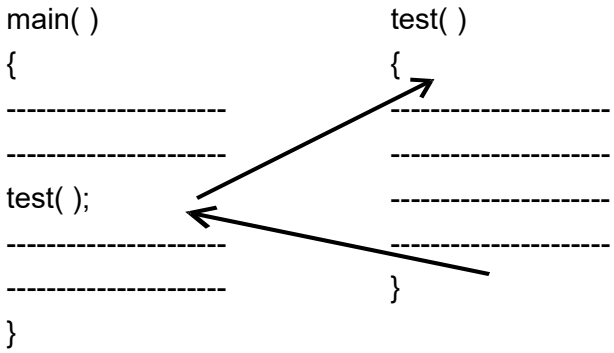
Output

```
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 CATEGORY 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

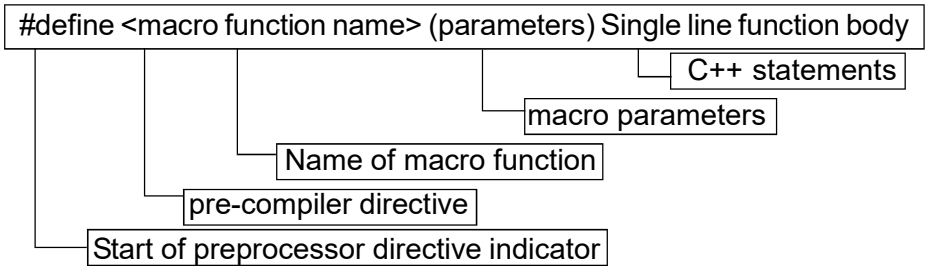
Valid macro declaration	Invalid macro declaration

`#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.