MSc IT (S3) 02
Exam. Code:M3OOP

Object-Oriented Programming through C++

SEMESTER-I

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY

BLOCK- 1



KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

Subject Experts

Prof. Anjana Kakati Mahanta, Gauhati University

Prof. (Retd.) Pranhari Talukdar, Gauhati University

Dr. Jyotiprokash Goswami, Assam Engineering College

Prof. Shikhar Kumar Sarma, Gauhati University

Programme Co-ordinator

Dr. Ridip Dev Choudhury, KKHSOU

Course Co-ordinators :

Dr. Ridip Dev Choudhury, KKHSOU

Dr. Tapashi Kashyap Das, KKHSOU

Dr. Sruti Sruba Bharali, KKHSOU

SLM Preparation Team

UNITS	CONTRIBUTORS
1	**Ms. Jonalee Barman Kakati**, NERIM
2	**Mr. Dwipen Laskar**, Gauhati University
3 , 5	**Ms. Sangeeta Borkakoty**, USTM
4	**Dr. Tapashi Kashyap Das**, KKHSOU
6	**Dr. Rita Chakraborty**, IDOL, Gauhati University
7	**Dr. Amitabha Nath**, NEHU

Editorial Team

Content : **Dr. Swapnanil Gogoi**, IDOL , Gauhati University

Structue, Format & Graphics: **Dr. Tapashi Kashyap Das**, KKHSOU

January, 2022 **ISBN: .....**

This Self Learning Material (SLM) of the Krishna Kanta Handiqui State Open University is made available under a Creative Commons Attribution-Non Commercial-Share Alike 4.0 License (international): http://creativecommons.org/licenses/by-nc-sa/4.0/

Printed and published by Registrar on behalf of the Krishna Kanta Handiqui State Open University

Headquarters: Patgaon, Rani Gate, Guwahati-781017

City Centre: NH 37, Resham Nagar, Khanapara, Guwahati-781022; Web:www.kkhsou.in

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY

Object-Oriented Programming through C++

Block-1

DETAILED SYLLABUS

		Pages
UNIT 1 :	**Introduction to Object-Oriented Programming [5 marks]** Basic concept of OOP, Comparison of Procedural Programming and OOP, Benefits of OOP, C++ compilation, Abstraction, Encapsulation, Inheritance, Polymorphism, Difference between C and C++.	**7-29**
UNIT 2 :	**Elements of C++ Language [5 marks]** Tokens and identifiers: Character set and symbols, Keywords, C++ identifiers. Variables and constants: Integers & characters, Constants and symbolic constants, Dynamic initialization of variables, Reference variables, Basic data types in C++, Streams in C++, scope resolution operator.	**30-74**
UNIT 3 :	**Operators and expressions [5 marks]** Operators, Types of Operators in C++, Precedence and Associativity, Manipulators, Enumerated data type, storage classes.	**75-95**
UNIT 4 :	**Decision and Control Structures [5 marks]** If statement, if-else statement, switch statements, Loop: while, do-while, for; jump statements : break, continue, goto.	**96-129**
UNIT 5 :	**Array and Structure [6 marks]** Arrays, structure, unions, Runtime memory management: new and delete operator.	**130-151**

UNIT 6 : Pointer [6 marks] **152-188**

Introduction, Address operator, pointer variable, pointer definition, pointers and parameter passing, void pointer, pointer arithmetic, Runtime memory management: new and delete operator, Pointers to pointer, Passing address of a pointer, Array of Pointers, Pointers to functions.

UNIT 7 : Functions [6 marks] **189-222**

main() function, components of function : prototype, function call, definition, parameter; passing arguments; types of function, inline function, function overloading.

COURSE INTRODUCTION

The course on ***Object-Oriented Programming through C++*** is an approach to programming, as it offers a powerful way to cope with the real-world problems. This Self-Learning Material (SLM) is for all those who want to learn and expand their C++ Object-Oriented programming ability. This course include theory as well as practical components. In order to bridge the gap between theory and practical, each concept is explained supported with various programs and examples.

The course is divided into two blocks. First block consist of seven units which explain the fundamentals concepts of C++ programming. Learners will be able to learn the basic difference of C and C++ programming with their syntax. The second block consist of eight units concentrating the Object-Oriented features of C++ programming language.

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

BLOCK INTRODUCTION

The first block is organised as follows:

The **first unit** is an introductory unit of Object-Oriented Programming. It describes the basic features of Object-Oriented Programming.

The **second unit** explains various elements of the language that are useful to write a program. It introduces C++ tokens, identifiers, constants, variables and data types etc.

The **third unit** discusses various operators and their use in writing expressions in C++.

The **forth unit** discusses various decision making statements such as *if()*, *switch_case* and jumping statements such as *break, continue* etc. The looping statements such as *while()*, *do_while()* and *for()* are described together with examples in C++ programming language.

The **fifth unit** is about arrays, structures and union. Examples with one-dimentional and two-dimentional arrays are covered in this unit.

The **sixth unit** is on pointers in C++. Various concepts associated with pointers and runtime memory management are covered in this unit.

The **seventh unit** is on functions in C++. How functions can be declared, defined and called are discussed in this unit. The C++ function supports inline mechnism, function overloading, default arguments. All these concepts are explained with examples.

UNIT 1: INTRODUCTION TO OBJECT-ORIENTED

UNIT STRUCTURE

1.1 Learning Objectives

1.2 Introduction

1.3 Basic Concepts of Object-Oriented Programming

1.4 Procedural Programming

1.5 Comparison of Procedural Programming and Object-Oriented Programming

1.6 Benefits of Object-Oriented Programming

1.7 C++ Compilation

1.8 Class

1.9 Objects

1.10 Abstraction

1.11 Encapsulation

1.12 Inheritance

1.13 Polymorphism

1.14 Difference between C and C++

1.15 Let Us Sum Up

1.16 Further Reading

1.17 Answers to Check Your Progress

1.18 Model Questions

1.1 LEARNING OBJECTIVES

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

- learn the basic concepts of Object-Oriented Programming (OOP)
- compare OOP with Procedural Programming
- describe the various benefits provided by OOP
- learn the programming paradigm of OOP
- learn about C++ compilation
- learn the differences between C and C++

1.2 INTRODUCTION

This is an introductory unit of Object-Oriented Programming. In Object-Oriented Programming (OOP), everything we see around us is considered as an object. An object is an identifiable entity with some attributes and behavior. Objects can be tangible like a car, mobile, book, etc, or intangible things like a bank account, email, mp3 file, etc. For example, mobile has attribute color, weight, size, etc. and behavior is dialing a call, ending a call, etc.

Alan Kay developed the concept of Object-Oriented programming. Alan Kay, Adele Goldberg, Dan Ingalls, and others developed the first Smalltalk programming language, which follows the concept of Object-Oriented Programming System.

The problem-solving aspect in OOP focuses on objects. The entities are first identified in the problem. Attributes of an object are represented by data and its behavior is represented by functions or operations that can be performed by the object. In this problem-solving approach, we start at the bottom by identifying the objects and work upwards towards the solution of the problem. Thus, it is termed as Bottom-up approach. OOP is a programming paradigm in which data and functions are wrapped into a single unit.

The major need for developing such languages was to manage the ever-increasing size and complexity of programs. Java, Python, C++, Lisp, and Perl are all examples of popular object-oriented programming languages. The idea behind OOP is to make the language more applicable, easier to use with a wider range of problems.

1.3 BASIC CONCEPTS OF OBJECT-ORIENTED PROGRAMMING

The programming paradigm is the way of writing computer programs. In the Object-Oriented Programming (OOP) paradigm, a program is built from objects. Objects contain both data and functions. OOP is a programming paradigm to design a program using classes and objects. The OOP has the following characteristics:

- In this paradigm, the whole program is created on the concept of objects.
- The objects may communicate with each other through functions.
- OOP mainly focuses on data rather than functionality.
- The programs are divided into what are known as objects.
- It follows the Bottom-up approach.
- The data and functions are wrapped into a single unit.
- Classes and objects are the two main aspects of object-oriented programming.
- It introduces concepts like data abstraction, inheritance, polymorphism, andoverloading of functions and operators overloading.
- In this paradigm, data is hidden and cannot be accessed by an external function.
- It has the concept of friend functions and virtual functions.

The followings are the main concepts of Object-Oriented Programming:

a) Object
b) Class
c) Encapsulation
d) Abstraction
e) Inheritance
f) Polymorphism

These concepts would be discussed in some detail in later sections.

OOP treats data as a critical element in the program and does not allow it to flow freely around the system. It ties data more closely to the function that operates on it and protects it from accidental modification from outside function. OOP allows the decomposition of a problem into several entities called objects and then builds data and functions around these objects. The organization of data and function in OOP are shown in the figure 1.1.

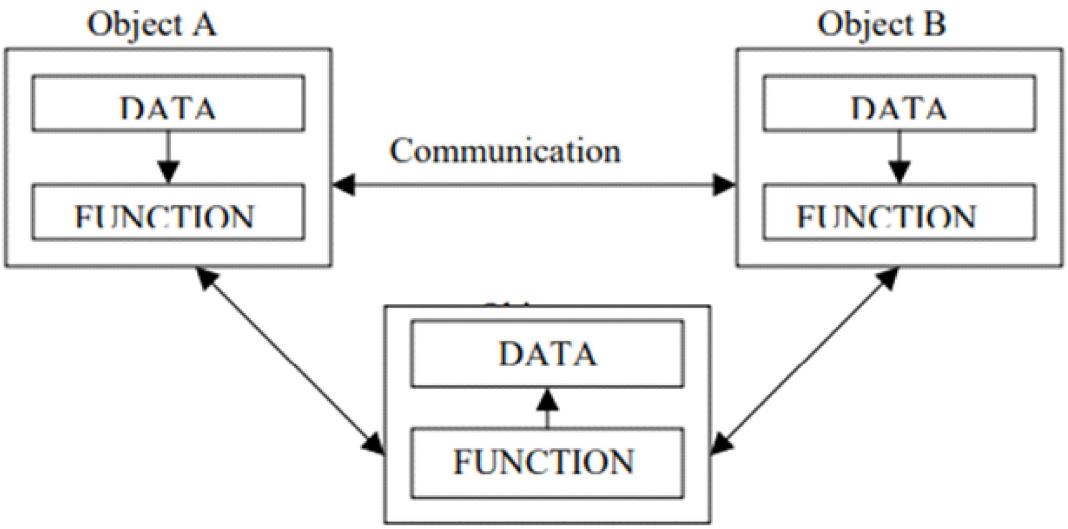


Figure 1.1: Organization of Data and Functions in OOP

The data of an object can be accessed only by the function associated with that object. However, the function of one object can access the function of other objects.

1.4 PROCEDURAL PROGRAMMING

Procedural Programming involves writing down a sequence of instructions to convey the computer what it should do step-by-step to solve the problem at hand. In Procedural Programming, a complex program is divided into smaller programs that are called procedures. The procedures are also referred to as subprograms, subroutines, methods, or functions. Each procedure or subroutine consists of a set of instructions to do a particular task. Procedural programming follows the Top-Down approach. In this approach, a main program is available where the subprograms are called as required to solve a particular problem. Procedural programming focuses on the process rather than data.

Some examples of the procedural programming languages are FORTRAN, ALGOL, COBOL, PL/I, and BASIC. The procedural Programming paradigm is one of the first paradigms to emerge in the computational world.A typical structure for procedural programming is shown in the figure 1.2.

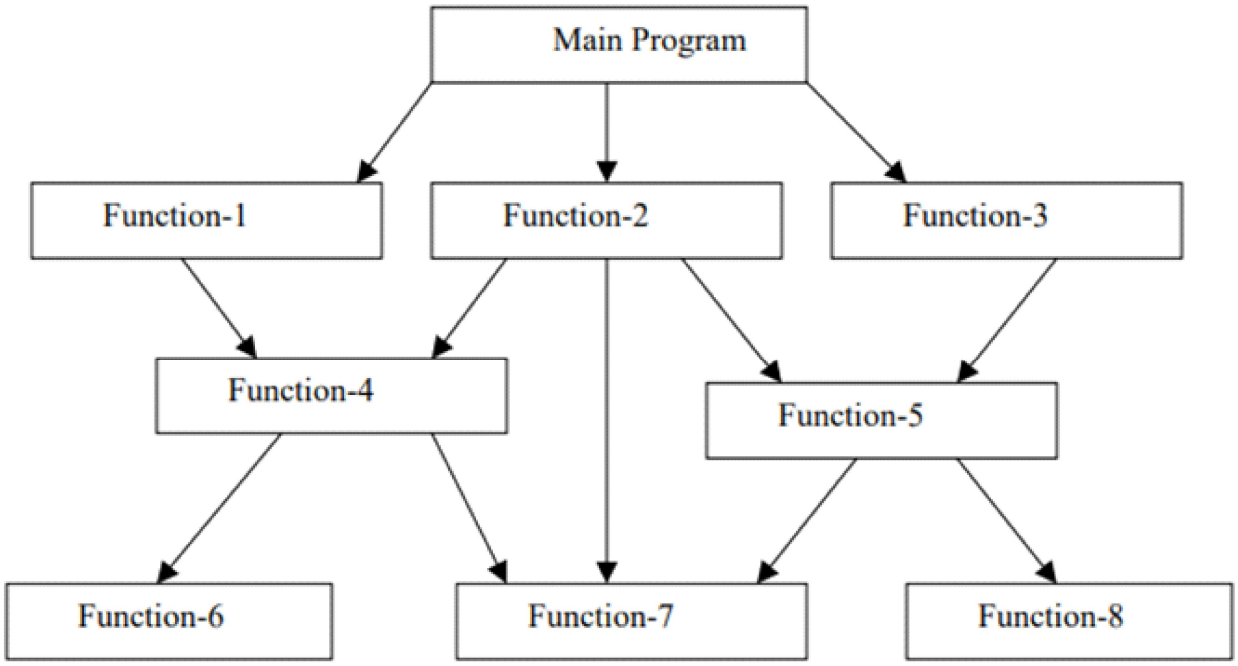


Figure 1.2: Typical Structure of Procedural Oriented Programs

In a multi-function program, data items can be placed as global so that they may be accessed by all the functions in the program. Each function may have its own local data. In a large program, it is very difficult to identify what data is used by which function. A drawback with the procedural approach is that the real-world problems cannot be modeled appropriately. This is because functions are action-oriented and do not correspond to the element of the problem.

1.5 COMPARISON OF PROCEDURAL PROGRAMMING AND OBJECT-ORIENTED PROGRAMMING

Procedural Programming focuses on the steps to be performed to complete a task. The tasks are broken down into sub-programs, variables, and data structures. At any point in time, these procedures can be called within the program execution.

Object-Oriented Programming (OOP), however, focuses on the interaction between the objects. Objects contain data in the form of attributes and code in the form of methods. In OOP, the programs are designed by using the concept of objects that represents the real world. In fact, the

major motivating factor in the invention of OOP was to remove some of the flaws encountered in the procedural approach.

The comparison of procedural programming and OOPispresented in the following tabular form.

Basis for comparison	**Procedural Programming**	**Object-Oriented Programming**
Basis	Procedural programming is an approach that relies on procedures, and operates on data, to solve a problem.	OOP is an approach where the concept of objects is used for problem-solving.
Division	Program is divided into units called functions.	Program is divided into units called objects.
Approach	Procedural Programming is a Top-Down Approach.	OOP follows a Bottom-up approach.
Data security	In Procedural Programming, data security is at risk as the data may freely move from function to function in the program.	In object-oriented programming, data security is given main concern. It provides data hiding.
Inheritance	Concept of Inheritance is not available in Procedural programming.	Concept of Inheritance is supported by OOP. Code reusability is achieved in OOP by Inheritance.
Access mode	There is no access specifier in procedural programming.	OOP has access specifiers like private, public, protected, etc.
Emphasis	In procedural programming, the primary focus is given to functions.	In OOP, data is given more importance than functions.
Friend function	Concept of friend function is not available.	Classes or functions can become a friend of another class with the keyword "friend".
Polymorphism	Concept of polymorphism is not available.	Polymorphism can be achieved in OOP by overloading functions and operators.

1.6 BENEFITS OF OBJECT-ORIENTED PROGRAMMING

The Object-oriented programming paradigm offers greater programmer productivity, a better quality of software, and lesser maintenance cost. There are multiple benefits of OOP. Some of the advantages are presented as follows:

- Data security is the first and one of the main advantages of OOP. The data and functions are combined together in the form of class. The concept of data hiding helps the programmer to build secure programs, so that data of one class will be secure from the other parts of the program.
- Concept of Inheritance may reduce redundant code and allows reusability of the code.
- Programs can be built from the standard working modules that communicate with one another. This leads to saving the development time and to increase high productivity.
- Software complexity can be easily managed.
- Object oriented systems can be easily upgraded from small to large systems.
- It is possible to have multiple instances of object to co-exist without any interference.

1.7 C++ COMPILATION

C++ is a high-level language and it follows the OOP paradigm. The source code of a C++ program needs to be translated into machine language (object code) with the help of a compiler. The extension of C++ source files is generally .cpp.

In this section, we would learn how C++ compilation works to understand how C++ programs are compiled and executed. Compiling C++ source code into machine-readable code consists of the following four processes:

- Preprocessing the source code.

- Compiling the source code.
- Assembling the compiled file.
- Linking the object code file to create an executable file.

Let us start with a simple C++ program to understand the C++ compilation process. Let us create a file named **Hello.cpp**

// Program 1.1

```
#include<iostream.h>
int main ()
{
      // This is a single line comment
      /* This is a multi-line
         comment */
      cout << "Hello;
      return 0;
}
```

In the above C++ program, iostream.h is a header file. Header files contain code that describes the available functionalities. These functionalities can be referred to and used in a program depending upon the requirement. The most common extensions for header files are .hpp, and .h.

To create an executable file from the header and the source files, the compiler preprocesses the directives (preceded by a **#** sign) that are available in them. The directive available in the above **Hello.cpp** program, is **#include**. This directive is preprocessed by the compiler before actual compilation and replaced it with the content of the **iostream** header, which describe standard functionality for reading and writing from streams.

Next, the compiler works through the preprocessed code by translating each line into the appropriate machine language instruction. This will also uncover any syntax errors that are present in your source code and will throw an error to the command line. If no errors are present, the compiler creates an object file in machine language.

To have a final executable program, another utility program known as the linker is used to combine the object files with the library functions necessary to run the code. Thus finally, the compiler links all the object files into a single executable program.

Compilation is a process that ensures that a program is syntactically correct, but it does not perform any checks regarding its logical correctness. This means that a program that compiles correctly might still produce undesired results.

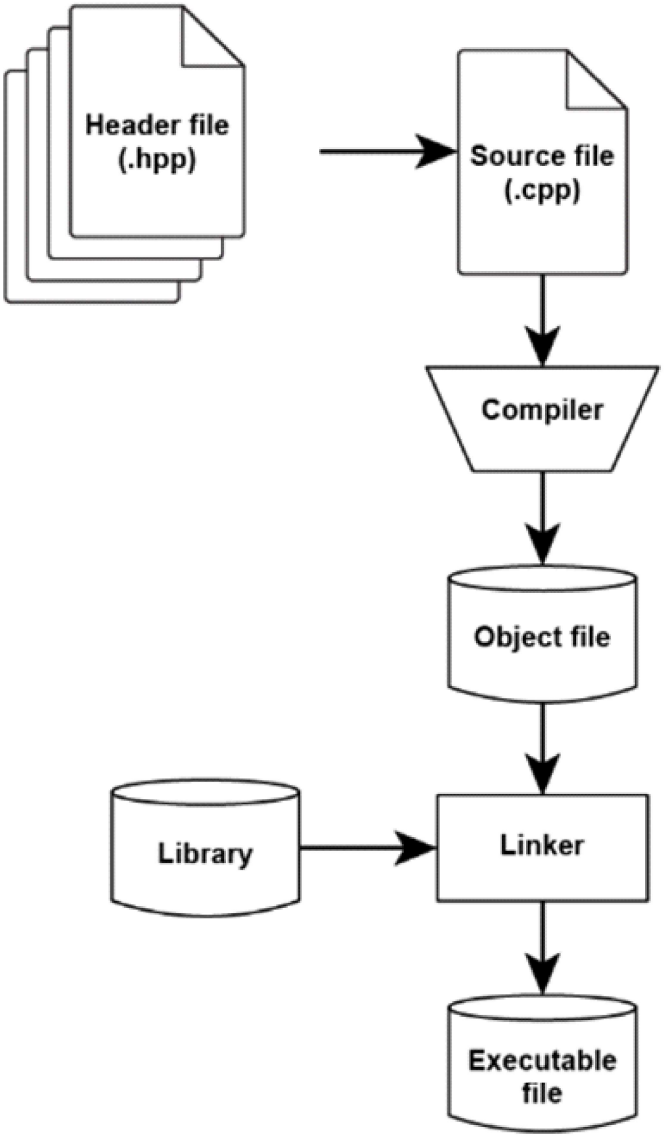


Figure 1.3: Compilation and linking processes for a C++ file



CHECK YOUR PROGRESS

Q 1: Fill In the blanks

i. Object-oriented programming (OOP) paradigm focuses on _____.

ii. The attributes of an object are represented by_____ and its behavior is represented by _____.

iii. The approach followed by OOP is a _____ approach.

iv. Object-Oriented programming was developed by _____.

v. Procedural Oriented Programming follows a _____.

vi. Procedural programming focuses on the _____ rather than _____.

vii. The access specifiers in OOPs are _____, _____ and _____.

viii. In OOP, _____ is given more importance than functions.

ix. Inheritance eliminates redundant code and allows _____ of the code.

x. C++ source files generally have the _____ extension.

1.8 CLASS

Class is the building block of Object-Oriented Programming (OOP). A class is a user-defined data type that contains data members and member functions. Data members and member functions of a class can be accessed and used by creating an instance of that class.

Class is an extension of structure that is used in C language. In structure, different types of data elements can be combined as a single entity. On the other hand, in class, different types of data elements, as well as member functions, can be combined. The class is used to implement encapsulation, data abstraction, and data hiding.

The access to the data and functions available in a class is governed by the three access specifiers — private, protected, or public. By default, access to members of a C++ class is private. The private members are not accessible outside the class; they can be accessed only through the member functions of the class. The public members are accessible outside the class.

In C++, a class is defined using the 'class' keyword. This should be followed by the class name. The class body is then added between curly braces { }. The general structure of a class is presented as follows.

```
class class-name
{
   Access specifier:    //can be private, public or protected
        Data members    //variables to be used
   Access specifier:
        Member functions   //methods to access data members
};                 //class definition ends with a semi-colon
```

For example, let us create a class 'Person' having the data members as name and age. That is, name and age are the variables to be declared in the class. The access specifier of the data members is going to be declared as public.

```
class Person
{
     public:
           string name;
          int age;
};
```

1.9 OBJECT

Object is an instance of a class. So, we can state that a class is a collection of similar types of objects. They may represent a person, a place, a bank account, a table of data, or any item that a program has to handle. Each object contains data and code to manipulate that data. Objects are the basic run-time entities in an object-oriented system.

Memory spaces are allocated for objects and each object is associated with a memory address. On execution of a program, the objects may interact with each other by sending messages to one another. They can interact without having to know details of each other's data or code.

To access the data members and the member functions of a class, objects must be created. Syntax to declare an object is:

Class Name Object Name;

For example: For the class Person (declared in the above section), the object can be created as:

```
Person obj1;
```

Here, obj1 is the name of the object of the class Person.

The data members and member functions of a class can be accessed using the dot ('.') operator with the object of that class. Let us consider the following program in C++ to understand object declaration and access of class members using dot operator with an object of a class.

// Program 1.2

```
#include <iostream.h>
class Abc
 {
     public:                    // Access specifier
          int age;             // int variable
          string name;        // string variable
};
void main()
{
  Abc Obj;   // Object declaration of class 'Abc'
  // Access attributes and set values
  Obj.age = 7;
Obj.name = "Priya";
  // Print attribute values
  cout <<"Age :"<<Obj.age << "\n";
  cout <<"Name :" <<Obj.name;
}
```

The output of the above program is presented as follows:

Age : 7
Name : Priya

1.10 ABSTRACTION

Data abstraction provides the foundation for OOP. Abstraction is the concept of OOP that “shows” only essential attributes and “hides” unnecessary information. The main purpose of abstraction is to hide unnecessary details from the users. Thus, data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation. It helps in reducing programming complexity and efforts.

Abstraction can be explained by taking into account a real-life example. Car is an example of abstraction. The car is started by turning the key or pressing the start button. It is not necessary for us to know how the engine is getting started, what machinery components the car has. The inner mechanism of the car and its complex logic are completely hidden from the user.

Abstraction in C++ can be implemented using class. Class helps us to group data members and member functions using available access specifiers. A class can decide which data member will be visible to the outside world and which is not.

Access specifiers can be used to enforce restrictions on class members. Members declared as public in a class, can be accessed from anywhere in the program. Members declared as private in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of code outside the class.

1.11 ENCAPSULATION

Encapsulation is a process of combining data members and functions in a single unit called class. This is to prevent access to the data directly. Access to the data is provided only through the functions of the class. These functions are considered as member functions in C++. In other words, encapsulation is wrapping the data together and the functions

that manipulate them. It leads to data hiding or abstraction. Encapsulation is the most basic concept of OOP.

A real-life example of encapsulation can be shown as in the diagram below:

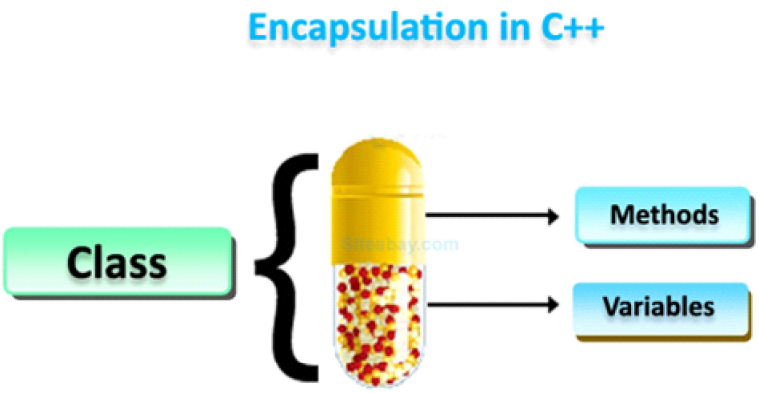


Figure 1.4: A Real-life Example of Encapsulation

Inside the capsule, all medicine is encapsulated together. Similarly, in a mobile phone, the keypad and the screen are encapsulated together.

Encapsulation can be used to hide the values of data objects inside a class so as to prevent unauthorized access to them.

In C++, encapsulation can be implemented using class and access specifiers i.e., private, protected, and public. An example is given below:

// Program 1.3

```
#include <iostream.h>
class encapsule
{
private:
          int x;
public:
          void accept (int a)
          {
               x=a;
          }
void print( )
{
```

```
cout<<" Value is:" << x;
}
};

void main()
{

      encapsule  obj;
      obj.accept(3);
      obj.print( );

}
```

Output:

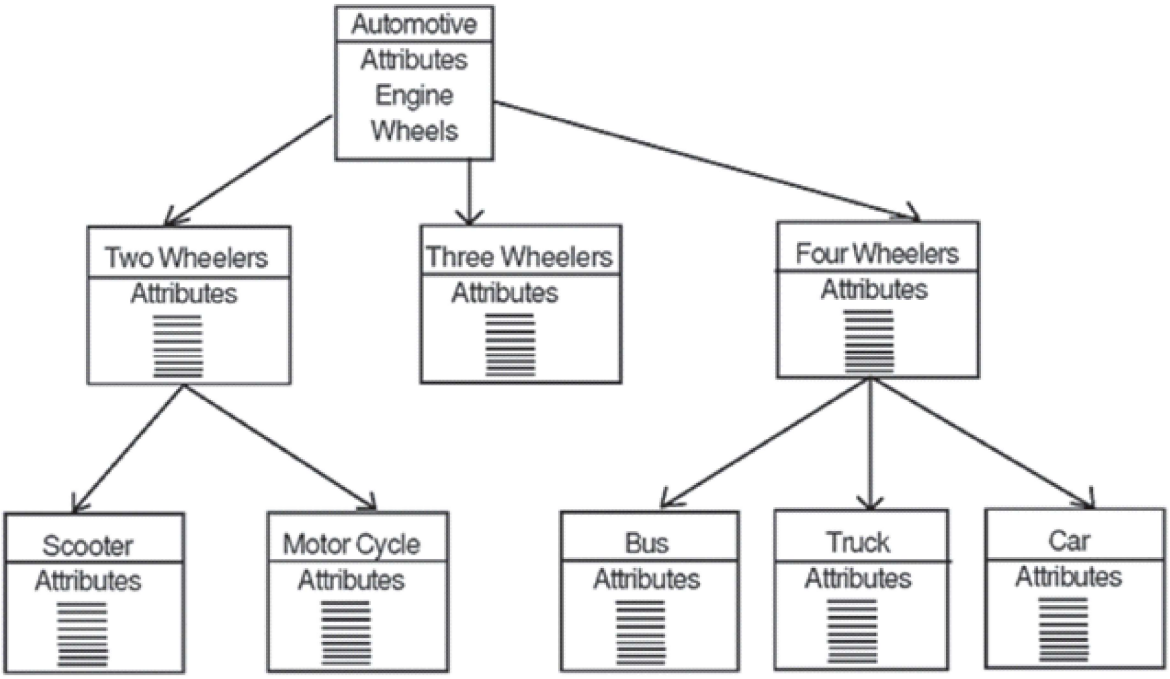
Value is: 3

1.12 INHERITANCE

Inheritance is a process by which object of one class inherits the properties of objects of another class. In OOP, the concept of inheritance provides the idea of reusability, that is, when a new class is derived it includes features of the existing class, and also the new class may have some additional features. It is always important to reuse something that is already exists rather than creating the same thing repeatedly. This is an important concept of OOP.

A class can inherit some or all of the properties of another class. The class from which the properties are inherited is called the *base class* or parent class or super class and the class which inherits the properties is called the *derived class* or child class.

An example of Motorcycle, as shown in the figure 1.5, can be used to explain the concept of inheritance. Motorcycle is a class in itself. It is a derived class of the base class 'Two Wheelers'. The 'Two Wheelers' class is again a derived class of the base class 'Automotive'. Thus, in simple words, a motorcycle is a two wheeler automotive.



Types of Inheritance:

In general, inheritance is classified into five different categories and they are:

- Single inheritance
- Multiple inheritance
- Multilevel inheritance
- Hierarchical inheritance
- Hybrid inheritance

These types of inheritance are discussed in later unit of this course.

1.13 POLYMORPHISM

Polymorphism means the ability to provide more than one form. Polymorphism is one of the crucial features of OOP. The word polymorphism is made up of two Greek words: “poly” and “morphism”. “Poly” means many and “morphism” means forms. So, polymorphism means many forms. In polymorphism, an operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation.

As an example, let us consider the addition operation. When the addition is carried out on numbers, it results in the summation of the numbers. If the same addition operation is performed on strings, it results in the concatenation of the strings. This can be explained in the figure given below:

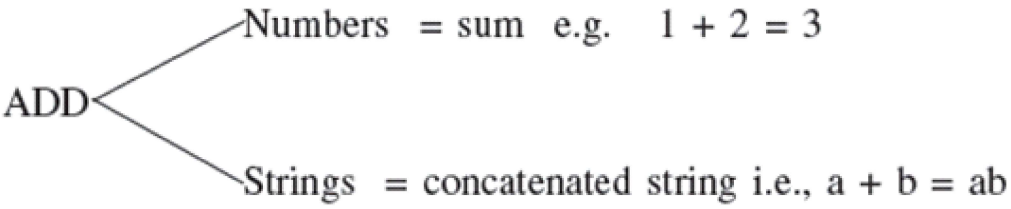


Figure1.6: Addition operation performed on different types of data

A real-life example of polymorphism can be given as — a person at the same time can have different characteristics. He can be a father, a husband, an employee at the same time. So the same person may possess different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of OOP.

In C++, polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism

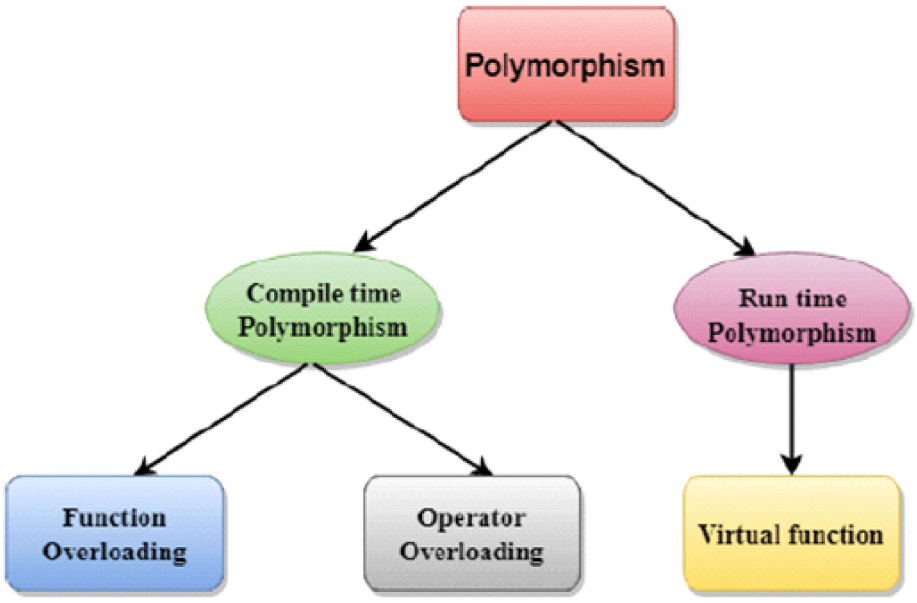


Figure 1.7: Classification of Polymorphism in C++

1.14 DIFFERENCES BETWEEN C AND C++

Basis of Distinction	C	C++
Programming Paradigm	It is a Procedural Oriented Programming language.	It is an Object-Oriented Programming language.
Focus	Focuses on method or procedures instead of data.	Focuses on data instead of method or procedure.
Approach	C language follows Top-Down programming approach.	C++ follows Bottom-Up programming approach.
File extension	In general, C program has file extension as .c	In general, C++ program has file extension as .cpp
Standard I/O operations	In C, scanf() and printf() are used for the standard input and output operations.	In C++, cin» and cout« are used for standard input and output operations.
Data security	In C language, the data is not secured.	Data security can be provided in C++ by using Encapsulation.
Encapsulation	Does not support encapsulation. As Data and functions are separate and free entities.	Supports encapsulation. Data and functions are encapsulated together as an single entity.
Information hiding	C does not support information hiding. In this language, data are free entities and can be changed outside code.	Data can be hidden by using Encapsulation.
Memory management	C provide malloc() and calloc() functions for dynamic memory allocation.	C++ provides an operator for dynamic memory allocation that is 'new'.
Inheritance	Inheritance is not supported in C	Inheritance is one of the main features of C++ language.
Polymorphism	In C, Polymorphism is not possible.	The concept of polymorphism is available in C++.
Friend functions	Concept of Friend function is not supported by C.	Concept of Friend function is supported by C++.
Exception Handling	C does not support Exception Handling.	C++ supports Exception handling.



CHECK YOUR PROGRESS

Q 2: Fill In the blanks

i. A class is a user defined data type which contains _____ members and member _____

ii. The default, access to members of a C++ class is _____.

iii. The private members are not accessible outside the _____ .

iv. An instance of a class is called an _____.

v. Objects take up _____ in the memory and have an associated _____.

vi. Abstraction is a concept of object-oriented programming that _____ only essential attributes and _____ unnecessary information.

vii. The wrapping up of data together with the functions that manipulate them is called _____ in OOPs.

viii. The process by which object of one class inherit the properties of objects of another class is called _____.

ix. The class from which the properties are inherited is called the _____ and the class which inherits the properties is called the _____.

x. An operation may exhibit different behavior in different instances and this is the concept of _____.



1.15 LET US SUM UP

- In the object-oriented programming (OOP) paradigm, a program is built from objects.
- The objects may communicate with each other through functions.

- OOP treats data as a critical element in the program and does not allow it to flow freely around the system.
- The data of an object can be accessed only by the function associated with that object.
- A programming paradigm where a complex program is solved by dividing it into smaller programs, called procedures, is called Procedural Programming.
- Procedural programming focuses on the process rather than data, as in case of object-oriented programming.
- Object Oriented Programming focuses on the interaction between the objects.
- In Object-Oriented Programming, the programs are designed using the concept of objects that represents the real world entities.
- Object Oriented Programming follows bottom-up approach but Procedural Oriented Programming follows top-down approach.
- In OOP, data security is given main concern. It provides data hiding.
- OOP has access specifiers like private, public, protected, etc.
- Polymorphism can be achieved in OOP by overloading functions and operators.
- The data and functions are combined together in the form of class.
- Inheritance eliminates redundant code and allows reusability of the code.
- The source code of C++ needs to be translated into machine code(object code) with the help of a compiler.
- C++ source files generally have the .cpp extension.
- Compilation of a program is a process that ensures that a program is syntactically correct, but it does not perform any checks regarding its logical correctness.
- Class is the building block of object-oriented programming. A class is a user defined data type that contains data members and member functions.

- The access to the data and functions in a class is governed by the three access specifiers — private, protected, or public. By default, access to members of a C++ class is private.
- Object is an instance of a class. Members of a class can be accessed by using objects of that class with dot operator.
- Objects take up space in the memory and have an associated address. On execution of a program, the objects interact with each other by sending messages to one another.
- Abstraction is the concept of object-oriented programming that "shows" only essential attributes and "hides" unnecessary information.
- Abstraction in C++ can be implemented using classes. Class helps us to group data members and member functions using available access specifiers.
- Encapsulation is a process of combining data members and functions in a single unit called class. This is to prevent access to the data directly. Access to the data is provided only through the functions of the class.
- To access a data item in an object, a member function in the object has to be called.
- Inheritance is a process by which object of one class inherits the properties of objects of another class. In OOP, the concept of inheritance provides the idea of reusability and extensibility.
- The class from which the properties are inherited is called the *base class* or parent class or super class and the class which inherits the properties is called the *derived class* or child class.
- Polymorphism means the ability to take more than one form. In polymorphism, an operation may exhibit different behavior in different instances.
- In C++ polymorphism is mainly divided into two types: compile time polymorphism and runtime polymorphism.



1.16 FURTHER READING

1. Schildt, H. (2003). *C++: The complete reference*. New York: McGraw-Hill.
2. Venugopal, K. R., & Rajkumar, T. (2004). Ravishankar: Mastering C++.
3. Balagurusamy, E. (2001). *Object-Oriented Programming with C++, 7e*. McGraw-Hill Education.
4. Jana, D. (2014). *C++ and Object-Oriented Programming Paradigm*. PHI Learning Pvt. Ltd.
5. SAMANTA, D. (2006). *Object-oriented Programming with C++ and Java*. PHI Learning Pvt. Ltd.



1.17 ANSWERS TO THE CHECK YOUR PROGRESS

Ans.to.Q.No.1

i. data ii. data, functions iii. Bottom-Up
iv. Alan Kay v.Top-Down Approach vi. process, data
vii. private, public, protected viii. data
ix. reusability x. .cpp

Ans.to.Q.No. 2

i. data, function ii. private iii. class iv. object
v. space, address vi. shows, hides vii. Encapsulation
viii. Inheritance ix. base class, derived class
x. Polymorphism



1.18 MODEL QUESTIONS

Q1. What are class and object in C++?

Q2. What are the differences between Procedural Programming and Object-Orientedprogramming?

Q3. Discuss the benefits of OOPs.

Q4. What do you mean by abstraction in C++?

Q5. List out the differences between C and C++.

Q6. Explain important properties of OOP.

*** ***** ***

UNIT 2 : ELEMENTS OF C++ LANGUAGE

UNIT STRUCTURE

2.1 Learning Objectives

2.2 Introduction

2.3 Character Sets and Symbols

2.4 Tokens

- 2.4.1 Keywords
- 2.4.2 Identifiers
- 2.4.3 Constants
- 2.4.4 Strings
- 2.4.5 Operators

2.5 Basic Data Types

- 2.5.1 Character Data Type
- 2.5.2 Integer Data Type
- 2.5.3 Float Data Type
- 2.5.4 Double Data Type
- 2.5.5 Void Data Type

2.6 User Defined Data Types

2.7 Derived Data Types

2.8 Declaration of Variables

2.9 Initialization of Variables

2.10 Dynamic Initialization of Variables

2.11 Reference Variables

2.12 Streams in C++

- 2.12.1 Unformatted Console I/O Operations
- 2.12.2 Formatted Console I/O Operations

2.13 Scope Resolution Operator

2.14 Let Us Sum Up

2.15 Further Reading

2.16 Answer to Check Your Progress

2.17 Model Questions

2.1 LEARNING OBJECTIVES

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

- learn about different types of tokens
- differentiate between keywords, constants and identifiers
- learn the naming rule of defining identifier
- learn about basic data types available in C++
- learn how to declare and initialize variables statically and dynamically
- understand the concept of reference variable.
- learn about different streams that are available in C++
- understand the concept of formatted and unformatted console I/O operations
- learn about scope resolution operator and its uses

2.2 INTRODUCTION

We are already familiar with basic concepts of OOP such as class, object, abstraction, encapsulation, inheritance, etc. We are also acquainted with the benefits of OOP, the difference between C and C++ language and how to compile C++ programs.

In this unit, we will learn about character set and symbols, various types of tokens such as keyword, identifiers and constants, etc., initialization of variables, basic data types, stream classes and scope resolution operator. We will also learn about different formatted and unformatted console I/O operations.In the next unit, we will explore various operators and expressionsused in C++.

2.3 CHARACTER SET AND SYMBOLS

We know that any language like the English language has some alphabets and symbols that are used to form meaningful words, sentences and paragraphs. Similarly, C++ has also its own character sets. A character set is a set of all valid characters and symbols that are recognized by a C++ program. These characters are the fundamental units of the C++

language. The C++ character sets are categorized into the following types presented as follows:

1	Letters	Alphabets from A to Z and a to z
2	Digits	0 to 9
3	Special Symbols	+ – * / ^ \ ( ) [ ] { } = != < > <= >= . ‘ ” $, ; : % ! & ? _ # @
4	White spaces	Blank space, Horizontal tab (→), Carriage return(↲), New line, Form feed

2.4 TOKENS

A token is the smallest individual unit in a C++program that is meaningful to the compiler. Various tokens available in the C++ language are classified into the following five classes:

- Keywords
- Identifiers
- Constants
- Special Characters
- Strings
- Operators

2.4.1 Keywords

A keyword is a predefined word in C++ Library that has a special meaning to the Compiler. Keywords cannot be used as an identifier.Keywords are also known as reserved words and are typed in lowercase in C++ language. The complete set of C++ keywords is given in Table 2.1. Many keywords as listed in Table 2.1 are common to both C and C++.

Table 2.1: C++ Keywords

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

2.4.2 Identifiers

Identifier is the name used to identify variables, functions, arrays, classes, objects, etc. that are declared by the programmer. They are the basic requirements of any programming language. Each programming language has its own rules to assign these identifiers. C++ language follows the following general rules for declaring its identifiers:

- Identifiers can contain letters, digits and underscores only.
- Identifiers cannot start with a digit.
- Identifiers are case sensitive i.e., uppercase and lowercase letters are distinct.
- Keywords cannot be used as identifiers in a C++ program.
- Blank spaces cannot be included in an identifier.

For example:*_Assam29*, *assam*, *a_28*, *name29* are examples of valididentifiers. Whereas *9assam*, *_ass$,abc-3* are examples of invalid identifiers

2.4.3 Constants

The constants can be referred to as fixed values that the program cannot change or alter during the execution of a program. The constants are also known as Literals. Like C, in C++ a constant can be of any data type (Integers, Characters, Floating points and Strings).C++ supports two types of constants that are literal and symbolic.

- **Literal constant**: A literal constant is a value typed directly into the program wherever it is needed. For example,let us consider the following program:

  Program 2.1:

```
#include <iostream.h>
int main()
{
     int size=5;
     cout<<"size="<<size;
     return 0;
}
```

Here, the value 5 is assigned to the variable *size.* The value 5 is the literal constant in this statement because we cannot assign a new value to 5 during execution of the program.

- **Symbolic constant**: Here, the *const* keyword is explicitly used to declare a variable as constant. Example of declaring different symbolic constants are shown in the following program.

  Program 2.2:

```
#include <iostream.h>
int main()
{
     const int a=5;
     const float b=2.78;
     const int c=017;
     const int d=0x17;
```

```
    const char e='A';
      const char f[112]="C++ PROGRAM";
      cout<<"\nConstant integer,a="<<a;
      cout<<"\nConstant float,b="<<b;
      cout<<"\nConstant octal,c="<<c;
      cout<<"\nConstant hexadecimal,d="<<d;
      cout<<"\nConstant character,e="<<e;
      cout<<"\nConstant string,f="<<f;
return 0;
}
```

Output:

Constant integer,a=5

Constant float,b=2.78

Constant octal,c=15

Constant hexadecimal,d=23

Constant character,e=A

Constant string,f=C++ PROGRAM

Symbolic constant can also be defined using enumeration. In C++, the keyword, 'enum' is used to declare an enumeration. For example,

enum color {black, white, red};

Here, color is the enumerated data type. 'black', 'white' and 'red' are the enumeration constants available in 'color'. By default, the value of the first enumeration constant is 0, the second enumeration constant has the value 1, and the third has the value 2, and so on. So, in case of the above example, black=0, white=1 and red=2. However, we can assign our required value to these constants. The following program code shows the defining symbolic constants using enumeration.

Program 2.3:

```
#include<iostream.h>
int main()
{
```

```
    enum color{black,white,red};
    enum age{Raju=20,Hari=11,Gita=19};
    cout<<"\n Value of black="<<black;
    cout<<"\n Value of white="<<white;
    cout<<"\n Value of red="<<red;
    cout<<"\n Age of Raju="<<Raju;
    cout<<"\n Age of Hari="<<Hari;
    cout<<"\n Age of Gita="<<Gita;
return 0;
}
```

Output:

Value of black=0

Value of white=1

Value of red=2

Age of Raju=20

Age of Hari=11

Age of Gita=19

2.4.4 Strings

Strings are the collection of characters always enclosed in double-quotes. It is an array of characters ended with the character,'NULL' ('\0'). The character, 'NULL' indicates the end of the string. Some examples of strings are presented as follows:

```
char name[6] ="Assam"  ;
char  name[6]={'A','s','s','a','m','\0'};
char name[ ] ="Assam";
```

2.4.5 Operators

Operators are special symbols used for specific purposes such as mathematical or logical manipulations. C++ supports all types of operators that are available in C such as *Arithmetical operators*, *Relational operators*, *Assignment operators*, *Conditional operators*, *Comma operator, Unary operators, Conditional*

Operators. C++ also introduces some new operators such as insertion operators (<<), extraction operators (>>), scope resolution operator(::), pointer-to-member declaration (::*), memory allocation operator (new), memory release operator (delete), pointer-to-member operator (->*) and, line feed operator (endl). More details about different C++ operators will be discussed in the next unit.

2.5 BASIC DATA TYPES

The type of data that a variable can store is defined by the data type of that variable e.g., integer, float, etc. In C++, data types are classified as *user-defined*, *built-in* and *derived* data type. The hierarchy of data type is shown in figure 2.1.

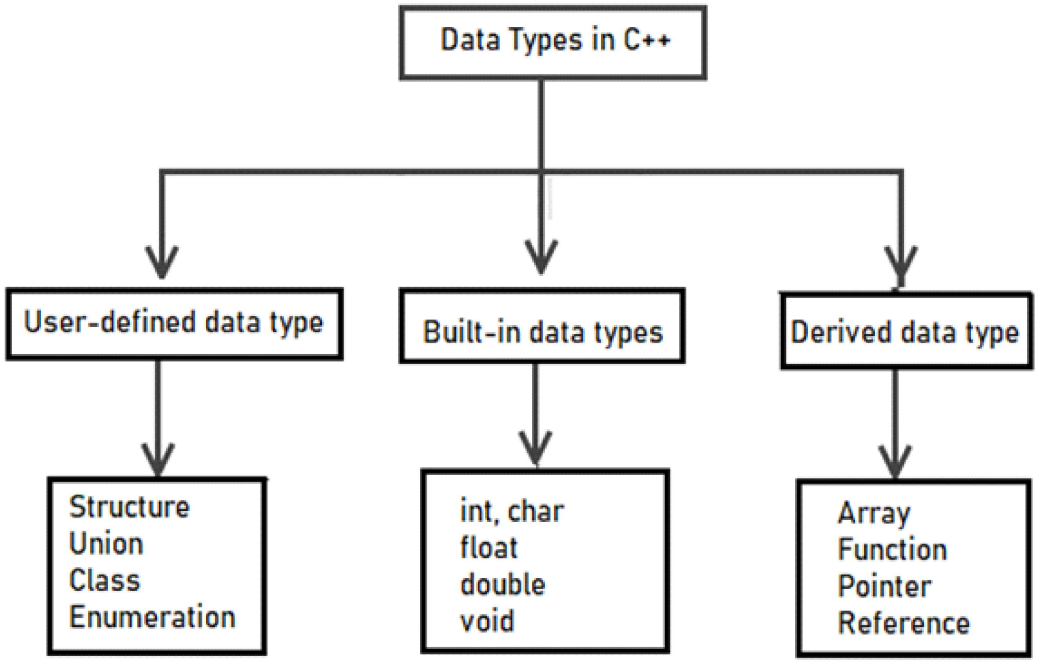


Fig.2.1. Hierarchy of data types in C++

Built-in data types are also known as basic or fundamental data types. Built-in data types available in C++ are *int (integer), char (Character), float (Floating Point), double (Double floating point)* and *void (*valueless). In C++, there are several modifiers that can be used to modify the meaning of the predefined built-in data types to serve the needs of various situations.

The modifiers expand the basic data types to hold much larger values than the predefined values. There are four datatype modifiers in C++, such as *long, short, signed* and *unsigned*. The modifiers *long* and *short* modify the maximum and minimum values that a variable of a data type canstore. The modifier *Signed* includes both positive and negative numbers whereas *unsigned* includes only positive values.

2.5.1 Character Data Type

The character data type is used to store character values. These characters include any valid character in C++. In C++, character data type is represented by the keyword '*char'.*A variable of characterdata type occupies 1 byte of memory space. By default, the range of values that a character variable can store isfrom -128 to 127. Two more variants of character data type are also supported by C++. These are*unsigned char* and *signed char.*Both of them occupy 1 byte of memory. The range of *unsigned char* is 0 to 255 whereas the range of signed character is same as default *char* data type. The following program code shows the use of *char* data type.

Program 2.4:

```
#include <iostream.h>
int main()
{
    char a='A';
    signed char b='B';
    unsigned char c='C';
    cout<<"\nChar a="<<a<<"\t\tsize="<<sizeof(a);
    cout<<"\nSigned Char b="<<b<<"\t\tsize="<<sizeof(b);
    cout<<"\nUnsigned Char c="<<c<<"\tsize="<<sizeof(c);
    return 0;
}
```

Output:

Char a=A size=1

Signed Char b=B size=1

Unsigned Char c=C size=1

2.5.2 Integer Data Type

The integer data type is used to store integer numbers. This data type is represented by the keyword '*int'.* A variable of integer data type occupies 2 or 4 bytes of memory space and ranges from -32,768 to 32,767 or -2147483648 to 2147483647. There are eight more variants of integer data types. They are *unsigned, signed, short, unsigned short, signed sort, long, signed long* and *unsigned long* integers. The memory size and range of different type of integer data types are shown in table 2.2.

Table 2.2: Size and range of different integer data types in C++ (32-bit compiler)

Type	Memory size(in bytes)	Typical range
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4 bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	8bytes	-2,147,483,648 to 2,147,483,647
signed long int	8bytes	same as long int
unsigned long int	8bytes	0 to 4,294,967,295

The following program code shows use of '*int'* data type.

Program 2.5:

```
#include <iostream.h>
int main()
{
    int a=1;
    signed int b=2;
    unsigned int c=3;
    short int d=1;
    signed short int e=2;
    unsigned short int f=3;
    long int g=1;
    signed long int h=2;
    unsigned long int m=3;
    cout<<"\nint a="<<a<<"\t\t\tsize="<<sizeof(a);
    cout<<"\nSigned int b="<<b<<"\t\tsize="<<sizeof(b);
    cout<<"\nUnsigned int c="<<c<<"\tsize="<<sizeof(c);
    cout<<"\nShort int d="<<d<<"\t\tsize="<<sizeof(d);
    cout<<"\nSigned Short int e="<<e<<"\tsize="<<sizeof(e);
    cout<<"\nUnsigned short int f = " << f << " \t size = "
    <<sizeof(f);
    cout<<"\nLong int a="<<g<<"\t\tsize="<<sizeof(g);
    cout<<"\nSigned Long int h="<<h<<"\tsize="<<sizeof(h);
    cout<<"\nUnsigned Long int m="<<m<<"\tsize="<<sizeof(m);
    return 0;
}
```

Output:

```
int a=1                 size=4
Signed int b=2          size=4
Unsigned int c=3        size=4
Short int d=1           size=2
Signed Short int e=2    size=2
Unsigned short int f=3  size=2
Long int a=1            size=8
Signed Long int h=2     size=8
Unsigned Long int m=3   size=8
```

2.5.3 Float Data Type

The float data type is used to store real numbers i.e. numbers that have decimal places. This data type is represented by the keyword *float.* A variable of float data type occupies 4 bytes of memory space and ranges from 3.4E-38 to 3.4E+38. The following program code shows the use of *float* data type.

Program 2.6:

```
#include<iostream.h>
int main()
{
    float f=2.5696;
    cout<<"\nfloat  f="<<f<<"\n  Size  of  f="<<sizeof(f)<<"
bytes";
    return 0;
}
```

Output:

float a=2.5696

Size of a=4 bytes

2.5.4 Double Data Type

The double data type is also used to store real numbers but with higher precisions. This data type is represented by the keyword *double.* A variable of double data type occupies 8 bytes of memory space and ranges from 1.7E-308 to 1.7E+308. There is one more double data type called *long double*. A variable of long double type occupies 16 bytes of memory. The following program code shows the use of double and long double data type.

Program 2.7:

```
#include <iostream.h>
int main()
{
```

```
    double d=2.5696;
    long double dd=567.567;
     cout<<"\nDouble d="<<d<<"\t\tSize of
     d="<<sizeof(d)<<" bytes";
     cout<<"\nLong double d="<<dd<<"\tSize of
     dd="<<sizeof(dd)<<" bytes";
    return 0;
}
```

Output:

Double d=2.5696	Size of d=8 bytes
Long double d=567.567	Size of dd=16 bytes

2.5.5 Void Data Type

The void data type is used to represent variable that contains no value or nothing i.e., empty value. This data type is represented by the keyword *void.* This data type can be used as return type of a function that does not return any value. It can also be used to represent empty argument list to a function. E.g. *void function_name(void).*

2.6 USER-DEFINED DATA TYPES

User-defined data types are required to be defined by the users before their use. In C++, there are four types of user-defined data types available: *class, structure, union* and *enumeration.*

- **Class data type**: A *class* data type includes *data (data member)* and *functions*(*member function*) that can be used by the class variables known as class *object*. Class data type is the basis of the *object*-oriented programming and it should be declared by using the keyword *class.* The access of the content of the class is governed by three access specifiers that are *private, public* and *protected*. By default the access to the class members is *private.* The syntax to define a *class* data type is presented as follows.

```
class <class_name>
{
     Access specifier:
          Data members;
          Member functions () {};
};
```

The following program shows the use of *class* data type.

Program 2.8

```
#include <iostream.h>
class salary
{
        float basic;
        float hra;
        float gross;
    public:
        void input_salary(float a, float b);
        void show_salary()
        {
            gross=basic + hra;
            cout<<"\nBasic salary="<<basic;
            cout<<"\nHRA="<<hra;
            cout<<"\nGross salary="<<gross;
        }
};
void salary::input_salary(float a, float b)
{
    basic=a;
    hra=b;
}

int main()
{
    salary x;
    x.input_salary(200.50,505.75);
    x.show_salary();
    return 0;
}
```

Output:

Basic salary=200.5

HRA=505.75

Gross salary=706.25

- **Structure data type**: A *structure* data type creates a single data type by grouping of possibly different data types. Similar to *class* data type it can also include *data (data member)* and *functions* (*member function*) which can be used by the variable of this data type i.e., instances of structure data type. Structure data type is declared by using the keyword *struct.* The syntax to define structure data type is presented as follows.

```
struct  struct_name
{
        // struct members
        Data type data_member1;
        Data type data_member2;
        ----------------------
        ----------------------
        ----------------------
        Data type data_memberN;
};
```

The following program shows the use of *struct* data type.

Program 2.9:

```
#include <iostream.h>
#include<string.h>
struct Passenger
{
     int TicketNo;
     int BusNo;
     char P_Name[20];
     int age;
};
```

```
int main()
{
      struct Passenger p;
      p.TicketNo = 101;
      p.BusNo = 21;
      strcpy(p.P_Name,"Ramesh  Das");
      p.age=56;
      cout << "Ticket No: " << p.TicketNo << endl;
      cout << "Bus No: " << p.BusNo<<endl;
      cout << "Name: " << p.P_Name << endl;
      cout << "Age: " << p.age;
      return 0;
}
```

Output:

Ticket No: 101

Bus No: 21

Name: Ramesh Das

Age: 56

- **Union data type**: A *union* data type is similar to structure and also creates a single data type by grouping of possibly different data types. But, unlike structure, it allocates same memory location for its members. Union data type is declared by using the keyword *union*. The syntax to define *union* data type is presented as follows.

```
unionunion_name
{
        // union members
       Data type data_member1;
       Data type data_member2;
       ----------------------
       Data type data_memberN;
};
```

The following program shows the use of *union*data type.

Program 2.10:

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

union Passenger
{
      int TicketNo;
      int BusNo;
      char P_Name[20];
};
int main()
{
      union Passenger p1;
      cout<<"Passenger-1:\n";
      p1.TicketNo = 101;
      cout << "TicketNo : " << p1.TicketNo << endl;
      p1.BusNo = 21;
      cout << "Bus number : " << p1.BusNo << endl;
      strcpy(p1.P_Name,"Ramesh Das");
      cout << "Name : " << p1.P_Name << endl;
      return 0;
}
```

Output:

Passenger-1:

TicketNo : 101

Bus number : 21

Name : Ramesh Das

- **Enumeration data type:**We have already discussed about enumeration data type in section 2.4.3. It is a user-defined data type that allows to assign integer constants to its members. By default, the first member is assigned value 0, second as 1 and so on. But,

users can change the default value and assign their own integer constant to the members of an enumeration data type. An enumeration data type is declared by using the keyword *enum*. Syntax to define enumeration *data* type is presented as follows:

```
enum e_name{member-1,member-2,.......,member-n};
```

Where, *-member-1*, *member-2*,....., *member-n* are enumeration constants. The following program shows the use of *enum* data type.

Program2.11:

```
#include <iostream.h>
enum Months_1 { January, February, March, April};
enum Months_2 { May=5, June=9, July, August};
int main()
{
    Months_1 m1,m11;
    m1=January;
    m11=March;    ;
    cout <<"January ="<<m1<<endl;
    cout <<"February ="<<m11<<endl;
    Months_2 m2,m21;
    m2=May;
    m21=August;
    cout <<"March ="<<m2<<endl;
    cout <<"April ="<<m21<<endl;
    return 0;
}
```

Output:

January =0

February =2

March =5

April =11

2.7 DERIVED DATA TYPES

The data types that are derived from the built-in/basic data types in C++ are called derived data types. There are *four* types of derived data types supported in C++. They are *arrays, functions, pointers* and *references*. In this section, we will discuss about *arrays, functions* and *pointers*. The reference data type will be discussed in section 2.11.

- **Arrays:** An array is a linear data structure and stores a collection of similar data items stored at contiguous memory locations. The idea of array is to represent many instances in one variable. The syntax for declaring an array data structure is:

```
Data_type arrayName[size_of_array];
```

Where *data_type* is the built-in data type and *arrayName* is the name of the array data type having size *size_of_array*. For example:

```
int roll_no[20];
char name[20];
```

The following program shows the implementation of *array* data type. In Unit-5, more details about this data type will be discussed.

Program 2.12:

```
#include <iostream.h>
int main()
{
    int Num[5]={1,12,37,34,5};
    char Str[30]="This is an array in C++";
    cout<<"\n1st Number="<<Num[0];
    cout<<"\n2nd Number="<<Num[1];
    cout<<"\n3rd Number="<<Num[2];
    cout<<"\n4th Number="<<Num[3];
    cout<<"\n5th Number="<<Num[4];
    cout<<"\nInput String="<<Str;
    return 0;
}
```

Output:

1st Number=1

2nd Number=12

3rd Number=37

4th Number=34

5th Number=5

Input String=This is an array in C++

- **Functions:** A function is a block of code that performs a specific task when it is called. It is a user defined data type and allows users to reuse the code it contains. Users can pass data items to functions that are known as parameters. There are various types of parameter passing techniques. To use the function, a statement is used called *function call* statement. More details of functions are discussed in Unit-7. The syntax for declaring a *function* is presented as follows.

```
Return_type  funtionName[parameterList];
```

Where, *Return_type* is the return type of the function `funtion Name` having parameters `parameter List`.

The following program shows the implementation of *function* data type.

Program 2.13:

```
#include<iostream.h>
float simple_interest(float p, float i, int y)
// function definition
{
     float si=0;
     si= (p*i*y)/100;
     return si;
}
int main()
{
     float p_amt;
     float R_int;
```

```
    int Yr;
    float result;
    cout<<"\nEnter the Principal amount=";
    cin>>p_amt;
    cout<<"\nEnter the Rate of Interest=";
    cin>>R_int;
    cout<<"\nEnter the number of year=";
    cin>>Yr;
   result = simple_interest(p_amt, R_int,Yr);
   // Calling the function
   cout <<"Simple interest:="<< result;
   return 0;
}
```

Output:

Enter the Principal amount=30000
Enter the Rate of Interest=16.72
Enter the number of year=15
Simple interest:=75240

- **Pointers:** A pointer is a derived data type that is used to store the address of another variable. More details about this data type are discussed in Unit-6. The syntax for declaring a pointer is presented as follows.

```
Data_type *PointerName;
```

For example:

```
int *ptr;
int b=10;
ptr=&b;
```

The above statement means that ptris a pointer variable and it stores the address of an integer variable *b* having value 10. The following program shows the implementation of *pointer* variable.

Program 2.14:

```
#include <iostream.h>
 int main()
{
      int *a;
      int b=10;
      a=&b; //address of variable b is assigned to a
      cout<<"\nb="<<b;
      cout<<"\nAddress of b="<<&b;//address of variable b
      cout<<"\na="<<a;
      cout <<"\n Value pointed by a="<<*a;
      cout<<"\nAddress of a="<<&a;
      return 0;
      }
```

Output:

b=10

Address of b=0x7ffd10d2f934

a=0x7ffd10d2f934

Value pointed by a=10

Address of a=0x7ffd10d2f938



CHECK YOUR PROGRESS

Q 1: Choose the correct option:

i) The value 132.54 can be represented using which of the following data type?

(a) double (b) int

(c) void (d) char

ii) The data type *short int* consists of all integers in the range of

(a) (-128 to +128) (b) (-32768 to +32767)

(c) (-6536 to +6536) (d) (0 to 255)

iii) The information that is stored in a variable is called :

(a) String (b) Literal

(c) Value (d) Operator

iv) In C++, …… refer to fixed values that do not change during the execution of a program.

(a) Identifiers (b) Constants

(c) Strings (d) Operators

v) Which of the following data types is used to specify the return type of a function when it is not returning any value?

(a) int (b) float

(c) double (d) void.

vi) Which of the following is NOT the user-defined data type in C++?

(a) Structure (b) Pointer

(c) Union (d) Class

Q 2: Fill in the blanks:

(i) The constants are also called as_____.

(ii) _____data types are structure, union, class and enumeration.

(iii) _____are explicitly reserved words and cannot be used as names for the program variables or other user-defined program elements.

(iv) _____ refers to the names of variables, functions, arrays, classes etc. created by the programmer.

2.8 DECLARATION OF VARIABLE

The variables in C++ need to be declared before their first use in the program. Unlike C, a variable in C++ can be declared anywhere in the *scope* which means it can be declared right at the place of its first use. The scope of a variable means the part of the program where the variable is accessible. The flexibility of variable declaration makes the program easier to understand. The Syntax for variable declaration in C++ is as given below:

```
<data type><variable name >;
```

e.g., int a;

Declarations of multiple variables are also allowed in the same statement in C++. Syntax for declaring multiple variables is:

```
<data type><variable-1>, <variable-2>,....., <variable-n>;
```

e.g., float `a, b, c;`

The following example shows the variable declaration in C++.

Program 2.15:

```
#include <iostream.h>
int main()
{
    int a;
    float b,c;
    a=6;
    b=7.78;
    c=17.98;
     cout<<"\na="<<a<<"\nb="<<b<<"\nc="<<c;
    float d;
    d=a+b+c;
    cout<<"\nSum of a,b,c="<<d;
    return 0;
}
```

Output:

a=6

b=7.78

c=17.98

sum of a,b,c=31.76

2.9 INITIALIZATION OF VARIABLES

When the variables are declared, they store an undetermined or garbage value until they are assigned a value for their first use. The assignment of a value to a variable at the time of its declaration is known as the initialization of the variable. The syntax for initialization of a variable is:

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

For example: `int a=2;`

The variable can also be initialized after declaration. For example:

```
int a;
a=5;
```

C++ allows initializing multiple variables at the time of their declarations. The syntax for initialization of a variable is:

```
<datatype><variable1>=<value1>,.......<variable-n>=<value-n>;
```

For example: `int a=2, b=7;`

There are two ways of variable initialization available in C++. These are *static initialization* and *dynamic initialization.* In case of static initialization, variables are assigned values before compilation as shown in the above examples. On the other hand, in dynamic initialization, a variable is assigned a value at run time. In this case, the value of the variable can be changed every time when the program is executed. The following example shows the static initialization of variable in C++.

Program 2.16:

```
#include <iostream.h>
int main()
{
    int a=6;
    float b=7.78,c;
    c=17.98;
     cout<<"\na="<<a<<"\nb="<<b<<"\nc="<<c;
    return 0;
}
```

Output:

a=6

b=7.78

c=17.98

2.10 DYNAMIC INITIALIZATION OF VARIABLES

C++ allows initialization of variables at run time i.e., variables are assigned values during the program in execution. It is termed as dynamic initialization. The value of this variable can be altered every time the program executes. The following example shows the dynamic initialization of variable. Dynamic initialization is extensively used in OOP (Object Oriented Programming) because of its flexibility to assign values to the variables at the time of program execution. The following example shows the dynamic initialization of variable.

Program 2.17

```
#include <iostream.h>
int main()
{
    int a;
    float b,c;
    cout<<"\nEnter the value of a=";
    cin>>a;
    cout<<"\nEnter the value of b=";
    cin>>b;
    c=a+b;
    cout<<"\nYou have enteed:\n";
    cout<<"\na="<<a<<"\nb="<<b;
    cout<<"\nSum of "<<a<<" and "<<b<<" is="<<c;
    return 0;
}
```

Output:

Enter the value of a=8
Enter the value of b=9.67
You have entered:
a=8
b=9.67
Sum of 8 and 9.67 is=17.67



EXERCISE

1. Modify program 2.17 to include dynamic initialization of *character* variable and display it.

2.11 REFERENCE VARIABLES

In C++, a new type of variable is introduced, known as *reference variable.* A reference variable allows us to create an *alias* i.e. alternative name for an already defined variable.

The general syntax for creating a reference variable is as follows:

<Data-type>&<Referace_Name> = <Variable_Name>;

When a variable is defined as reference to an already declared variable then bothrefer to the same memory location. So,if we change the value to any of them then it will affect both.Let us consider example 2.18. Here, a variable,'**mult'**is declared as a reference to an already defined variable,'**product'**. So, we can use both '**mult'**and '**product'**interchangeably in the program. The variable,'**product'** is initialized as 20. Hence, variable,'**mult'**will also be initialized to the value of the variable,'**product'**i.e. 20. Now, the value of referencing variable is changed as 78. Hence, value of the variable, **'product'** is also changed to 78.

Program 2.18

```
#include <iostream.h>
int main()
{
     int product=20;
     int &mult=product;
     cout<<"\nValue of product="<<product;
     cout<<"\nValue of mult="<<mult;
    mult=78;
     cout<<"\nValue of product="<<product;
     cout<<"\nValue of mult="<<mult;
     return 0;
}
```

Output:

Value of product=20

Value of mult=20

Value of product=78

Value of mult=78

2.12 STREAMS IN C++

A stream in C++ is a sequence of bytes or characters that are transferred between the program thread and I/O. It is a general name given to the flow of data. There are two types of streams: *input stream* and *output stream.*The input stream is the source stream that takes input from any type of input device (e.g. keyboard) or any other storage device (e.g. files, program etc.) and provides data to the program. The output stream is the destination stream that extracts data from the program and provides output to any type of output device (e.g. monitor, printer, etc.) or any other storage device. A pictorial view of data streams is shown in figure 2.2.

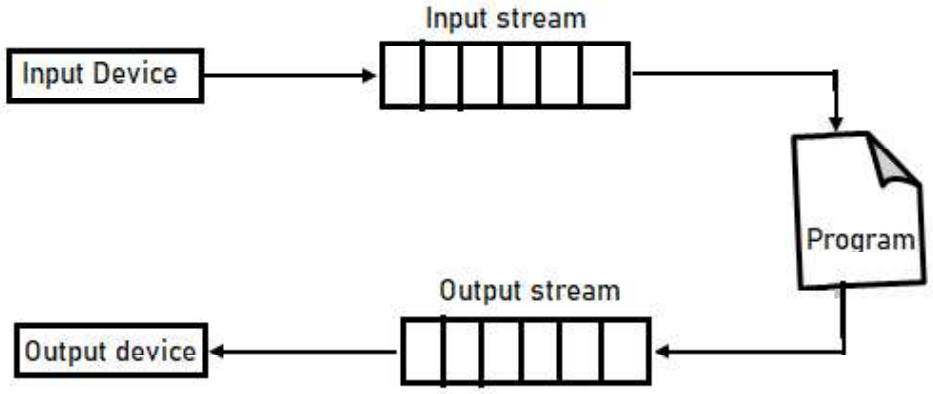


Fig. 2.2 Data streams in C++

In C++ I/O system, various streams are defined as a class hierarchy known as stream classes to perform operations on both console and disk files. All these classes are included and declared in the header file *iostream.h*. The header file *iostream* communicates with the console unit and it is mandatory to be included in all C++ programs. The hierarchy of the stream classes are shown in figure 2.3.

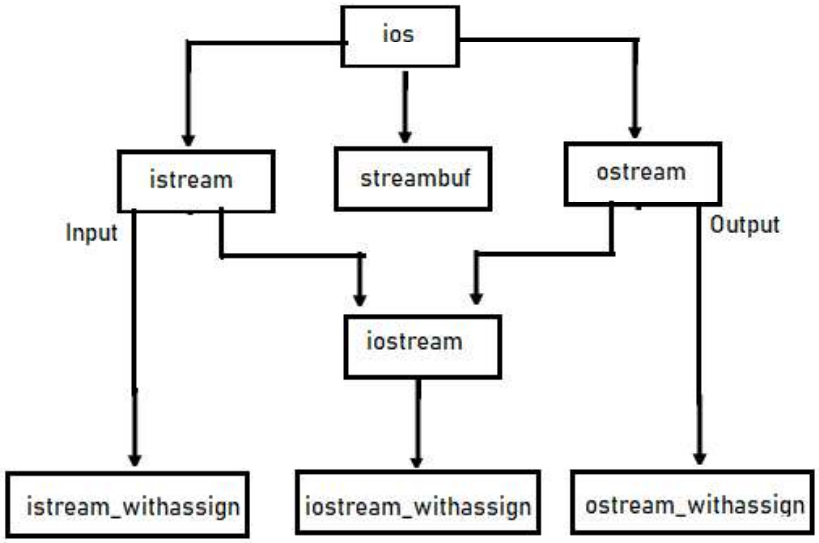


Fig. 2.3 Stream Class hierarchy for console I/O operations in C++

The **ios** is the base class for **istream** (input stream) and **ostream** (output stream) as shown in the figure Fig. 2.3. The class **ios** is declared as a virtual base class to avoid the duplicity of data and member functions of **ios** class to be inherited in **istream** and **ostream**. Both *istream* and *ostream* are base classes for **iostream**(input/output stream). '**cin'**and '**cout'** are objects of *iostream* class. The object *cin* uses the extraction operator (**>>**) to perform data input operation while *cout* uses the insertion operator (**<<**) to perform data output operation. The operator **>>** overloaded in *istream* class and operator **<<** overloaded in *ostream* class. Example 2.19 shows the use of *cin* and *cout* objects.

Program 2.19:

```
#include <iostream.h>
int main()
{
     int productNo;
     cout<<"\nEnter the productNo=";
     cin>>productNo;
     cout<<"\nEntered Product No is="<<productNo;
     return 0;
}
```

Output:

Enter the productNo=56

Entered Product No is=56

The **streambuf** class provides an interface to physical devices through buffers. The **istream_withassign class**, **ostream_withassign** class and iostream**_withassign class** add assignment operators respectively to istream class, ostream class and iostream class.The I/O console operations in C++ are classified as *formatted* and *unformatted* I/O operations.

2.12.1 Unformatted Console I/O Operations

Unformatted console I/O operations are performed at the console and the resulting data is left unformatted i.e., data remains in raw and original input form. Two overloaded operators:insertion operator (**<<**) and extraction operator (**>>**) are mostly used for unformatted I/O operations. There are many unformatted console I/O functions available in C++. Some of the most important functions for this type of I/O operations are: put(), get(), getline() and write() functions.

- **Extraction operator (>>):**Extraction operator is overloaded in *istream* class and used to read data from the keyboard. The operator **>>**is used to read character by character and assigned the input data to the indicated memory locations. The general format for reading input data by using **>>** is:

  ```
  cin>>variable-1>>variable-2>>.....>> variable-n;
  ```

  where, *variable-1, variable-2,....., variable-n* are valid C++ variables that are already declared.

  e.g.,
  ```
  int  a, b, c;
  cin>> a >>b >>c;
  ```

- **Insertion operator (<<):**Insertion operator is overloaded in *ostream* class and used to output data to the displaying screen. The general format for output data by using **<<** is:

  ```
  cout<<variable-1<< variable-2<<.....<< variable-n;
  ```

  where, *variable-1, variable-2,....., variable-n* are valid C++ variables or constants of any basic data that are already declared.

e.g., `int a, b, c;`

`cout << a <<b <<c;`

- **get()** function**:** get() is defined in *istream* class and used to read a *single* character from the user at the console, *and returns it*. There are two types of **get()** functions: **get(char *)** and **get(void)**.
  - In **get(char *)**, the input is assigned to its argument, e.g. **cin.get(c)**. Here, *c* is the character variable already declared. The input from keyboard is assigned to the variable *c.* The following example shows the use of **get(char*).**

    Program 2.20

```
#include <iostream.h>
int main()
{
    char p;
    cout<<"\nEnter the character, p=";
    cin.get(p);
    cout<<"\nEntered character is="<<p;
    return 0;
}
```

    Output:

    Enter the character, p=A

    Entered character is=A

  - The **get(void)** function return the input character, e.g. **c=cin.get()**. Here, the input from keyboard is assigned to the character *c*. The following example shows the use of **get(void).**

    Program 2.21:

```
#include <iostream.h>
int main()
{
    char p;
```

```
        cout<<"\nEnter the chacter, p=";
        p=cin.get();
        cout<<"\nEntered character is=";
        cout.put(p);
        return 0;
    }
```

Output:

Enter the character, p=A

Entered character is=A

- **put()** function**:**put() is defined in *ostream* class and used to output a *single* character on the output device. For examples, **cout.put(c),** where *c* is a character variable already declared. Example 2.21 shows the use of **put()** function.
- **getline()** function**:**getline() is defined in *istream* class and used to read a whole line of text that ends with a newline character or that reached its maximum limit of characters. This function can be invoked by the *cin* object. For examples, **cin.getline(line_of_text, size_of_the_line),** where *line_of_text* is a input sequence of characters and *size_of_the_line* is the length of the input sequence i.e. no of characters.
- **write()** function**:**write() is defined in *ostream* class and used to output a whole line of text. It can be invoked by the object **cout**. For examples, **cout.write(line_of_text, size_of_the_line),** where *line_of_text* is a input sequence of characters and *size_of_the_line* is the number of characters to be written to the output device.

Program 2.22 shows the use of **getline()** and **write()** functions.

Program 2.22:

```
    #include <iostream.h>
int main()
{
    char p[20];
```

```
    cout<<"\nEnter a line of text=";
    cin.getline(p,20);
    cout<<"\n";
    cout.write(p,20);
    cout<<"\n";
    cout.write(p,15);
    cout<<"\n";
    cout.write(p,5);
    return 0;
}
```

Output:

Enter a line of text=Niraj Chopra Won Gold Medal in Olympic
Niraj Chopra Won Go
Niraj Chopra Wo
Niraj



EXERCISE

2. Modify Program 2.15 read a line of text "Unformatted Console I/O operations" and display it using *write ()*.

2.12.2 Formatted Console I/O Operations

Formatted console I/O operations are those input/output operations that are performed at the console to display the data in formatted form. C++ provides many predefined member functions in ***ios class***, *manipulators* to display formatted data. Manipulators are special functions to modify the input/output stream. To use these manipulators we have to include the header file **iomanip.h** in our program. C++ also allows user to design user-defined manipulators.

ios Class Member Functions :

There are various **ios** member functions available in C++ to format data. They are **width()**, **precision()**, **fill()**, **setf()** and **unsetf()**.

- **width ()** function: width() specifies the width of a field that is required to display the output data at the console. The syntax for using this function is: **cout.width(w),** where *w* is the field width of the displayed output. For example, consider the following statements:

```
cout.width(9);
cout<<"INDIA";
```

Above statements will show following output:

				I	N	D	I	A

- **precision ()** function**:** precision() specifies the number of digits to be displayed at the console after the decimal place of a floating-point value. The syntax for using this function is: **cout.precision(d),** where *d* is the number of digits after the decimal place of the number.
- **fill()** function**:**fill() fills the unused white spaces of the output data with a user-specified character. The syntax for using this function is: **cout.fill(c),** where *c* is the character that will be used to fill up the unused white spaces. For example, consider the following statements:

```
cout.width(9);
cout.fill('X')
cout<<"INDIA";
```

X	X	X	X	I	N	D	I	A

The above statements will show the following output:

- **setf()** function**:** setf() allows to set formatting flags in the output stream. The syntax for using this function is: **cout.setf (argument-1,argument-2),** where *argument-1* denotes the formatting *flag* and *argument-2* denotes the group to which the formatting flag belongs. There are three groups of formatting flags such as **ios::adjustfield**, **ios::floatfield** and **ios:basefiled**. In **ios::adjustfield,** three formatting flags are defined which are used to justify the outputs as *right justified*,

left justified and *internal justified.* They are: **ios::right**, **ios::left** and **ios:internal**. In **ios::floatfield,** two formatting flags are defined which are used to format the floating-point number at the console in *scientific* and *fixed point* notations. They are: **ios::scientific** and **ios:fixed**. In **ios::basefield,** three formatting flags are defined which are used to format the base of the output data at the console in *decimal*, *octal* and *hexadecimal* notations. They are: **ios::dec**, **ios::oct** and **ios:hex**.

Now let us consider the following C++ statements.

```
cout.fill('X');
cout.width(15);
cout.precision(5);
cout.setf(ios::internal, ios::adjustfield);
cout.setf(ios::scientific, ios::floatfield);
cout<<-32.345617;
```

The output of the above statements is presented as follows.

–	**X**	**X**	**X**	**3**	**.**	**2**	**3**	**4**	**5**	**6**	**e**	**+**	**0**	**1**

- **unsetf()** function**:**unsetf() allows to reset the formatting flags in the output stream. The syntax for using this function is: **cout.unsetf(flag)**, where *flag* denotes the *ios*flag that will be unset i.e. removed from the output stream. For example, the C++ statement: **cout.unsetf(ios::internal)**will remove the *internal flag* from the *cout* object.

The following example 2.23 shows the use of all the above *ios* member functions:

Program 2.23:

```
#include <iostream.h>
int main()
{
     cout<<"\nUse of width() function:\n";
     cout<<"Good\n";
```

```
    cout.width(15);
       cout<< "Good";
       cout<<"\nUse of precision function:\n";
       float a=2.34578;
       cout.precision(2);
       cout<<"a="<<a<<endl;
       cout.precision(3);
       cout<<"a="<<a;
       cout<<"\nUse of fill() function:\n";
       cout.width(15);
       cout.fill('*');
       cout<<567;
       cout<<"\nUse of setf() function:\n";
       cout.fill('X');
       cout.width(15);
       cout.precision(5);
       cout.setf(ios::internal, ios::adjustfield);
       cout.setf(ios::scientific, ios::floatfield);
       cout<<-32.345617;
       cout<<"\nUse of unsetf() function:\n";
       cout.unsetf(ios::internal);
       cout<<"\n"<<-32.345617;
       return 0;
}
```

Output:

Use of width() function:

Good

Good

Use of precision function:

a=2.35

a=2.346

Use of fill() function:

**********567

Use of setf() function:

-XXX3.23456e+01

Use of unsetf() function:

-3.23456e+01

Output Using Manipulators :

In C++, variousfunctions known as manipulators are also available to format data.They provide the same functionalities as the member functions of *ios* class. But, it only modifies the I/O stream using insertion (<<) and extraction (>>) operators. Some of the most commonly used manipulators are: **setw()**, **setprecision()**, **setfill()**, **setbase(), setiosflags()**,**resetiosflags()**and **endl**.

- **setw ()** function**:**setw() is equivalent to *width()* function. It also sets the field width of the displayed output data at the console. The syntax to use this function is presented as follows.

  setw(w);

  In the above syntax**,***w* is the field width of the displayed output.
- **setprecision ()** function**:**setprecision() is equivalent to *precision()* function. It also sets the precision value of the displayed floating point output data at the console. The syntax to use this function is presented as follows.

  setprecision(d);

  In the above syntax**,***d* is the number of digits after the decimal place of the number.
- **setfill()** function**:**setfill() is equivalent to *fill()* function and used to fill the unused white spaces in the output at the console. The syntax to use this function is presented as follows.

  setfill(c);

  In the above syntax**,***c* is the character that will be used to fill up the unused white spaces.
- **setbase() function:** setbase() is used toformat the base of the output data.The syntax to use this function is presented as follows.

setbase(int base);

In the above syntax, base is an integer value that corresponds to a particular base as mentioned below.

Value of base	Base Used for the input/output
0	Base 10
8	Octal
10	Decimal
16	Hexadecimal

- **setiosflags()** function**:**setiosflags() is equivalent to *setf()* function. It is used to set formatting flags in the output stream. The syntax to use this function is presented as follows.

  setiosflags(flag);

  In the above syntax**,***flag* is *ios formatting flag.*
- **resetiosflags()** function**:**resetiosflags() is equivalent to *unsetf()* function and it reset the formatted flag from the output stream at the console. The syntax to use this function is presented as follows.

  resetiosflags(flag);

  In the above syntax**,***flag* is *ios formatting flag.*
- **endl** function**:**endl is used to insert a new line character (carriage or a line feed character) to the output. It is associated to an output buffer and known as *end of line*. It is similar to '\n' character in C++. The syntax to use this function is presented as follows.

  endl

  As an example to use endl, let consider the following statement.

  ```
  cout<<"KKH State Open University"<<endl;
  ```

  This statement will insert a new line character after the string "KKH State Open University".

The following example shows the use of all the above *manipulator* functions:

Program 2.24:

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

int main()
{
     cout<<"\nUse of setw() function:\n";
     cout<<"Good\n";
     cout<<setw(15)<<"Good";
     cout<<"\nUse of setprecision() function:\n";
     float a=2.34578;
      cout<<setprecision(2)<<"a="<<a<<endl;
      cout<<setprecision(3)<<"a="<<a;
     cout<<"\nUse of setfill() function:\n";
cout<<setw(15)<<setfill('*')<<567;
     cout<<"\n Use of setbase() function:\n";
cout<<"\nThe Octal equivalent of a is="<<setbase(8)<<a;
     cout<<"\nUse of setiosflags() function:\n";
   cout<<setfill('X')<<setw(15)<<setprecision(5);
cout<<setiosflags(ios::internal);
      cout<<setiosflags(ios::adjustfield);
     cout<<-32.345617;
    cout<<"\nUse of resetiosflags() function:\n";
      cout<<resetiosflags(ios::internal);
     cout<<"\n"<<-32.345617;
     return 0;
}
```

Output:

Use of setw() function:

Good

Good

Use of setprecision() function:

a=2.35

a=2.346

Use of setfill() function:

**********567

Use of setbase() function:

The Octal equivalent of a is=2.346

Use of setiosflags() function:

-XXXXXXXX32.34562

Use of resetiosflags() function:

-32.34562

User-Defined Manipulators :

C++ allows users to define user-defined manipulators for controlling the input and output as their requirement. The general syntax for designing a user-defined manipulator is presented as follows.

```
ostream & manipulator_name (ostream & obj_name)
{
      ........................
      return obj_name;
}
```

Where *manipulator_name* is the user-defined manipulator name and ***ostream & obj_name*** is the manipulator called with user-defined object name ***obj_name***. The following example shows a user-defined manipulator called *show_bases* that show input decimal number in hexadecimal base.

Program 2.25:

```
#include<iostream.h>
#include<iomanip.h>
ostream & show_bases(ostream & output)
{
output<<setbase(8);
cout << "\nThe Octal equivalent is= ";
```

```
return output;
}
int main()
{
int num = 70;
cout<<"\nGiven number is="<<num;
cout<<show_bases << num;
      return 0;
}
```

Output:

Given number is=70

The Octal equivalent is= 106

2.13 SCOPE RESOLUTION OPERATOR

The scope of a variable is limited inside the block where it is declared and is local to that block. Two different declarations of variables with the same names in different blocks have different scopes and memory locations. When a global variable is declared and there is also a variable with the same name as the global variable declared in a function then both variables have different scope and memory locations. In C, it is not possible to access the global variable inthat function. But, in C++, we can access the global variable in that function by using the scope resolution operator (::). The syntax for using the scope resolution operator is **::variable_name**. The following program illustrates the use of scope resolution operator.

Program 2.26

```
#include<iostream.h>
int num=20;
int main()
{
      int num = 70;
      cout<<"\nThe local variable, num="<<num;
```

```
    cout <<"\nThe global variable, num=" << ::num;
    return 0;
}
```

Output:

The local variable, num=70

The global variable, num=20

In C++, scope resolution operator is also used to:

- Definemember functions outside a class.
- Access static variable of a class.
- Access a class nested inside another class



CHECK YOUR PROGRESS

Q 3: Declare a reference variable *sum* which referencesan integer variable *total* having value 20.

Q 4: State TRUE/FALSE

i. For using the manipulators in C++, iomanip.h header file should be included in a program.

ii. cin is an output defined in ostream class

iii. cout uses << (insertion) operator.

iv. The *ios* class contains *setw()* member function

Q 5: Fill in the blanks:

i. :: operator is called _____ operator.

ii. The *width(),precision()* member functions are included in _____class.

iii. >> operator is overloaded in the _____ class.

iv. In C++, _____provides an alias (alternative name) for a previously defined variable



2.14 LET US SUM UP

- In C++ various types of **tokens** are defined which include keywords, identifiers, constants, strings and operators.

- C++ provides a special type of variable called **reference variable** to assign an alternative name to previously defined variables.
- A constant variable is a variable whose value cannot be modified or altered during the run of the program. It is defined by using **const** modifier.
- C++ supports different data types namely **built-in**, **derived** and **user-defined** data types.
- Variable in C++ can be initialized either in run time or before compilation. Run time initialization is known as **dynamic initialization** and initialization performed before compilation is known as **static initialization.**
- Variable in C++ can be initialized anywhere in the program.
- C++ has two **streams** called *input stream* and *output streams* to get input from any input devices and to provide output at console.
- Various functions, **manipulators** are provided in C++ to control the formatted and unformatted I/O at the console.
- A special operator called **scope resolution operator** in C++ helps to uncover the scope of a variable.



2.15 FURTHER READING

1. Balagurusamy, E. (2001). *Object-Oriented Programming with C++, 7e*. McGraw-Hill Education.
2. Thareja, R.(2014)*Object-Oriented Programming with C++*, 5e, Oxford Publication.



2.16 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1:

i) (a)	ii) (b)	iii) (c)
iv) (b)	v) (d)	vi) (b)

Ans to Q No 2:

(i) Literals (ii) Use-defined

(iii) Keywords (iv) Identifiers

Ans to Q No 3:

```
int total=20;
    int &sum=total;
```

Ans to Q No 4:

(i) TRUE (ii) FALSE (iii) TRUE (iv) FALSE

Ans to Q No 5:

(i) Scope Resolution (ii) ios

(iii) istream (iv) reference variable



2.17 MODEL QUESTIONS

Q1. What is a token? Explain different types of token in C++ with example.

Q2. What is a variable? Enumerate the rules of naming variables in C++.

Q3. Name the different types of constants in C++.

Q4. What is a data type? State different types of data supported in C++. Give example of each. Differentiate between *derived* data type and *user-defined* data type.

Q5. What is *void* data type? State the importance of it.

Q6. Describe, with examples, the uses of enumeration data types.How a *union* data type is different from *structure* data type?

Q7. What is a class? Name the different access specifiers used in class to access its members.

Q8. What is dynamic initialization of variable? How it is different from static initialization?

Q9. What is a reference variable? Give example.

Q10. When there are a Global variable and Local variable with the same name, how will you access the global variable?

Q11. What is stream in C++? What are the Extraction and Insertion operators in C++? Explain with examples.

Q12. Illustrate about different formatted and unformatted console I/O operations in C++.

Q13. Write down the uses of scope resolution operator.Write a C++ program to show the use of scope resolution operator.

Q14. Write a C++ program to show the uses *setw()* and *width()* function.

*** ***** ***

UNIT 3: OPERATORS AND STORAGE CLASSES

UNIT STRUCTURE

3.1 Learning Objectives

3.2 Introduction

3.3 Types of C++ Operators

- 3.3.1 Arithmetic Operators
- 3.3.2 Assignment Operators
- 3.3.3 Relational Operators
- 3.3.4 Logical Operators
- 3.3.5 Bitwise Operators
- 3.3.6 Other C++ Operators

3.4 Precedence and Associativity

3.5 Storage Classes

3.6 Let Us Sum Up

3.7 Further Reading

3.8 Answers to Check Your Progress

3.9 Model Questions

3.1 LEARNING OBJECTIVES

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

- learn about different C++ operators and use them in programming
- learn about Precedence and Associativity of C++ operators
- learn to use manipulators
- learn about enumerated types and storage classes

3.2 INTRODUCTION

In the previous unit, we have learnt various elements of C++ programming language. We have learnt various basic as well as user-defined data types in C++. We have also learnt how to declare variables, various I/O operations etc.

In this unit,we will learn about the different types of operators in C++ with the help of examples. In programming, an operator is a symbol that performs operations on variables and values. For example, + is an operator used for addition, while - is an operator used for subtraction.

3.3 TYPES OF C++ OPERATORS

Operators in C++ can be classified into six types that are presented as follows.

- Arithmetic Operators
- Assignment Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Other Operators

3.3.1 ARITHMETIC OPERATORS

Arithmetic operators are used for performing arithmetic operations on variables and data.

For example let us consider the following statement,

a + b;

Here, the + operator is used to add values stored in two variables,a and b.

Arithmetic operators available in C++ are presented in the following table.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (Remainder after division)

Table 3.1 Arithmetic Operators and their Operations

Program3.1: C++ program to show the use of Arithmetic Operators

```
#include <iostream.h>
int main()
      {
      int a, b;
      a = 7;
      b = 2;
      // printing the sum of a and b
      cout << "a + b = " << (a + b) << endl;
      // printing the difference of a and b
      cout << "a - b = " << (a - b) << endl;
      // printing the product of a and b
      cout << "a * b = " << (a * b) << endl;
      // printing the division of a by b
      cout << "a / b = " << (a / b) << endl;
      // printing the modulo of a by b
      cout << "a % b = " << (a % b) << endl;
      return 0;
      }
```

Output

a + b = 9

a - b = 5

a * b = 14

a / b = 3

a % b = 1

Note:

1. As we can see from the above example, if an integer is divided by another integer, we will get the quotient. However, if either divisor or dividend is a floating-point number, we will get the result in decimals.

2. The modulo operator % computes the remainder. So, if a = 9 is divided by b = 4, then a % b = 1. Again if a = 9 is divided by b = 3 then a % b = 0. The % operator can only be used with integers.

Increment and Decrement Operators:

C++ also provides two more arithmetic operators that are increment (++)operator and decrement (– –) operator.Both these operators are unary operators. The increment operator increases the value of the operand by 1 and the decrement operator decreases the value of its operand by 1.Both increment and decrement operators can be used as prefix or postfix in an expression. If they are used as a prefix then depending upon the operator, the value of the operand is incremented or decremented by 1 before they are used in the expression. On the other hand if they are used as a postfix then their current value is first used in the expression. After that, depending upon the operator, the value of the operand will be incremented or decremented.

Program3.2: C++ program to show the use of Increment and Decrement Operators.

```
#include <iostream.h>
int main()
{
      int a = 10, b = 100, result_a, result_b;
            cout<< "\n a="<<a;
      cout<< "\nUse of increment operator as a prefix:";
      result_a = ++a;
      cout<< "\na = " << a <<endl;
      cout<< "\nresult_a = " <<result_a<<endl;
      cout<< "\nUse of increment operator as a postfix:"
      result_a = a++;
      cout<< "\na = " << a <<endl;
      cout<< "\nresult_a = " <<result_a<<endl;
```

```
        cout<<  "\nb="<<b;
        cout<< "\nUse of decrement operator as a prefix:";
        result_b = –b;
        cout<< "\nb = " << b <<endl;
        cout<< "\nresult_b = " <<result_b<<endl;
        cout<< "\nUse of decrement operator as a postfix:";
        result_b = b--;
        cout<< "\nb = " << b <<endl;
        cout<< "\nresult_b = " <<result_b<<endl;
        return 0;
}
```

Output:

a=10

Use of increment operator as a prefix:

a=11

result_a = 11

Use of increment operator as a postfix:

a=12

result_a = 11

b= 100

Use of decrement operator as a prefix:

b =99

result_b = 99

Use of decrement operator as a postfix:

b =98

result_b = 99

3.3.2 ASSIGNMENT OPERATORS

In C++, assignment operators are used to assign values to variables.

For example,

To assign 5 to a, the following statement can be used.

a = 5;

Some of the commonly used compound assignment operators are presented in the following table.

Operator	Example	Equivalent to
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

Table 3.2: Assignment Operators

Program3.3: C++ program to show the use of Assignment Operators

```
#include <iostream.h>
int main()
      {
      int a, b;
      a = 2;          // 2 is assigned to a
      b = 7;         // 7 is assigned to b
      cout<< "a = " << a << endl;
      cout << "b = " << b << endl;
      cout<< "\nAfter a += b:" <<endl;
      // assigning the sum of a and b to a
      a += b;  // a = a +b
      cout << "a = " << a << endl;
      return 0;
}
```

Output

a = 2

b = 7

After a += b:

a = 9

3.3.3 RELATIONAL OPERATORS

A relational operator is used to check the relationship between two operands.

For example:

To check whether a is greater than b or not, the following statement can be used.

a > b;

Here, > is a relational operator. It checks if a is greater than b or not. If the relation is true, it returns 1 whereas if the relation is false then it returns 0.

Different relational operators available in C++ are presented with their meaning in the following table.

Operator	Meaning	Example	Output
==	Is Equal To	3 == 5	False
!=	Not Equal To	3 != 5	True
>	Greater Than	3 > 5	False
<	Less Than	3 < 5	True
>=	Greater Than or Equal To	3 >= 5	False
<=	Less Than or Equal To	3 <= 5	True

Table 3.3: Relational Operators with Their Meanings

Program 3.4: C++ program to show the use of Relational Operator

```
#include <iostream.h>
int main()
{
      int a, b;
      a = 3;
      b = 5;
      int result;
```

```
    result = (a == b);    // false
    cout << "3 == 5 is " << result << endl;

    result = (a != b);  // true
    cout << "3 != 5 is " << result << endl;

    result = a > b;    // false
    cout << "3 > 5 is " << result << endl;

    result = a < b;    // true
    cout << "3 < 5 is " << result << endl;

    result = a >= b;  // false
    cout << "3 >= 5 is " << result << endl;

    result = a <= b;  // true
    cout << "3 <= 5 is " << result << endl;

    return 0;
}
```

Output

3 == 5 is 0

3 != 5 is 1

3 > 5 is 0

3 < 5 is 1

3 >= 5 is 0

3 <= 5 is 1

3.3.4 LOGICAL OPERATORS

Logical operators are used for checking one or more logical conditions in an expression. If acondition is true, it returns **1** whereas if it is false then it returns **0**.

Operator	Example	Meaning
&&	expression1 && expression2	Logical **AND**. True only if both expression1 and expression2 are True.
||	expression1 || expression2	Logical **OR**. True if at least one of the operands is true.
!	!expression	Logical **NOT**. True only if the operand is false.

Table 3.4: Logical Operators with Their Meanings

Program3.5: C++ program to show the use of Logical Operators

```
#include <iostream.h>
int main()
{
      int result;
      result = (3 != 5) && (3 < 5);      // true
      cout << "(3 != 5) && (3 < 5) is " << result << endl;

      result = (3 == 5) && (3 < 5);     // false
      cout << "(3 == 5) && (3 < 5) is " << result << endl;

      result = (3 == 5) && (3 > 5);     // false
      cout << "(3 == 5) && (3 > 5) is " << result << endl;
      result = (3 != 5) || (3 < 5);     // true
      cout << "(3 != 5) || (3 < 5) is " << result << endl;

      result = (3 != 5) || (3 > 5);     // true
      cout << "(3 != 5) || (3 > 5) is " << result << endl;

      result = (3 == 5) || (3 > 5);     // false
      cout << "(3 == 5) || (3 > 5) is " << result << endl;

      result = !(5 == 2);     // true
      cout << "!(5 == 2) is " << result << endl;
```

```
        result = !(5 == 5);     // false
        cout << "!(5 == 5) is " << result << endl;

        return 0;
}
```

Output

(3 != 5) && (3 < 5) is 1

(3 == 5) && (3 < 5) is 0

(3 == 5) && (3 > 5) is 0

(3 != 5) || (3 < 5) is 1

(3 != 5) || (3 > 5) is 1

(3 == 5) || (3 > 5) is 0

!(5 == 2) is 1

!(5 == 5) is 0

Explanation of the output:

- (3 != 5) && (3 < 5) evaluates to **1** because both operands (3 != 5) and (3 < 5) are 1 (**true**).
- (3 == 5) && (3 < 5) evaluates to **0** because the operand (3 == 5) is 0 (**false**).
- (3 == 5) && (3 > 5) evaluates to **0** because both operands (3 == 5) and (3 > 5) are 0 (**false**).
- (3 != 5) || (3 < 5) evaluates to **1** because both operands (3 != 5) and (3 < 5) are 1 (**true**).
- (3 != 5) || (3 > 5) evaluates to **1** because the operand (3 != 5) is 1 (**true**).
- (3 == 5) || (3 > 5) evaluates to **0** because both operands (3 == 5) and (3 > 5) are 0 (**false**).
- !(5 == 2) evaluates to **1** because the operand (5 == 2) is 0 (**false**).
- !(5 == 5) evaluates to **0** because the operand (5 == 5) is 1 (**true**).

3.3.5 BITWISE OPERATORS

In C++, bitwise operators are used for performing operations on individual bits. These operatorscan only be used alongside char and int data types.

Operator	Description	Result
&	Binary AND	Returns 1 if and only if both the operands are 1. Otherwise, it returns 0.
|	Binary OR	Returns 1 if at least one of the operands is 1. Otherwise, it returns 0.
^	Binary XOR	Returns 1 if and only if one of the operands is 1. However, if both the operands are 0, or if both are 1, then the result is 0.
~	Binary One's Complement	Changes binary digits 1 to 0 and 0 to 1.
<<	Binary Shift Left	Shifts all bits towards the left by a certain number of specified bits.
>>	Binary Shift Right	Shifts all bits towards the right by a certain number of specified bits.

Table 3.5: Bitwise Operators with Their Descriptions

3.3.6 OTHER C++ OPERATORS

A list of some other common operators available in C++ is presented in the following table.

Operator	Description	Example
sizeof	Returns the size of data type or variable.	*sizeof(int); // 4*
?:	Termed as conditional operator and the syntax is: A ? B : C ; Here A, B and C are three expressions. If A is True then the value of B will be the result. On the other hand if A is False then the value of C will be the result.	*string result = (5 > 0) ? "even" : "odd";* *// "even"*
&	Represents memory address of the operand.	*# // address of num*
.	Used to access members of struct variables or objects of a class.	*s1.marks = 92;// s1 is an object*
->	Used with pointers to access members of struct variables or objects of a class.	*ptr->marks = 92;* *// ptr is a pointer to an object*
<<	Used to display the output on standard output device.	*cout << 5;*
>>	Used to get the input value from standard input device.	*cin>>num;// num is a variable*
new	Used for dynamic memory allocation.	*char *ch;* *ch = new char[50];*
delete	Used for memory de-allocation.	*delete ch; // ch is a pointer variable*

Table 3.6: Other Operators with Their Descriptions



CHECK YOUR PROGRESS

Q 1: ++ increases the value of the operand by _____.

Q 2: The % operator can only be used with integers. True/False.

Q 3: In C++, _____ operators are used to assign values to variables.

Q 4: What is the meaning of <= operator?

Q 5: What is the C++ operator for Logical NOT?

3.4 PRECEDENCE AND ASSOCIATIVITY

Operator precedence determines the grouping of terms in an expression. The associativity of an operator is a property that determines how operators of the same precedence are grouped in the absence of parentheses. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example, let us consider the following expression

```
x = 7 + 3 * 2;
```

Here, x is assigned 13, not 20 because operator * has higher precedence than +. Similarly operator + has higher precedence than =. So, in the above statement, at first, 3*2 will be estimated and then 7 is added toits result.After that the final result will be assigned to x.

The following table shows different C++ operators according to their precedence. Operator with the highest precedence appears at the top of the table and those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] ->:: .	Left to right
Unary	+ - ! ~ ++ -- (type)*&sizeof new delete	Right to left
Member Access	.* ->*	Left to right
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<<>>	Left to right
Relational	<<= >>=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Left to right
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Table 3.7: Operators According to Their Precedence with Their Categories and Associativity

3.5 STORAGE CLASSES

Storage classes are used to describe the scope and life-time of variables and functions within a C++ Program. In C++, five types of storage classes are available discussed as follows.

1. **auto**

   The **auto** storage class is the default storage class for all local variables. For example, let us consider the following statements.

```
{
      int mount;
      auto int month;
}
```

In the above statements, by default the storage class of 'mount' will be auto. So, the storage class for both the variables,'mount' and 'month' will be auto. The auto storage classcan only be used within functions, i.e., local variables.The scope and life-time of an auto variable is limited within the function or block where it is declared.

2. Register

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that a register variable has a maximum size equal to the register size.A register variable does not have a memory location.

Example to declare a register variable:

```
{
      register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it might be stored in a register depending on hardware and implementation restrictions.

3. static

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time when it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C++, when static is used on a class data member then it causes only one copy of that member to be shared by all objects of its class.

Program 3.6: C++ program to show the use of static storage class

```
#include <iostream.h>

// Function declaration
void func(void);
static int count = 10; /* Global variable */
int main()
{
while(count—)
{
func();
      }

      return 0;
}

// Function definition
void func( void )
{
static int i = 5; // local static variable
i++;
cout<< "i is " << i ;
cout<< " and count is " << count <<endl;
}
```

Output:

i is 6 and count is 9

i is 7 and count is 8

i is 8 and count is 7

i is 9 and count is 6

i is 10 and count is 5

i is 11 and count is 4

i is 12 and count is 3

i is 13 and count is 2

i is 14 and count is 1

i is 15 and count is 0

4. extern

The **extern** storage class is used to give a reference of a global variable that is visible to all the program files. When you use 'extern', the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.When you have multiple files and you define a global variable or function in one file which will be used in other files also, then extern will be used to give reference to the defined variable or function. So, the extern modifier is most commonly used when two or more files are sharing the same global variables or functions.Let us consider the following example to understand about the extern storage class.

Program 3.7:C++ program files to show the use of extern storage class.

First File: First.cpp

```
#include <iostream.h>
extern void Function_Extern();

int main()
{
      Function_Extern();
      return 0;
}
```

Second File: Second.cpp

```
#include <iostream.h>
int temp = 10;
void Function_Extern()
{
      cout << temp;
}
```

Here, 'extern' keyword is being used to declare the function, 'Function_Extern()' in the first file, 'First.cpp' as the function with extern storage class. The function, 'Function_Extern()' is defined in the second file, 'Second.cpp'. When these two files are compiled and executed then we will get the output as: 10.

5. mutable

The **mutable** specifier applies only to data members of a class.If the object of a class is declared as const then only mutable data members of that object can be changed in runtime.A mutable member can be modified by a const member function of an object. Let us consider the following example to show the use of mutable specifier.

Program 3.8: C++ program to show the use of mutable specifier

```
#include <iostream.h>
class A
{
public:
mutable int var1,var2,var3;

A()
{
      var1 = 50;
      var2 = 60;
            }
     void display()const
     {

                  cout<<"\nvar1= "<<var1;
                  cout<<"\nvar2= "<<var2;
                  cout<<"\nvar3= "<<var3;
 }
};
int main()
{
```

```
const A a;  // 'a' is a constant type object
a.var3 = a.var1 + a.var2; /* Return error if var1, var2 and
var3 are not declared with mutable specifier.*/
      a.display(); /* Return error if display() is not
constant type.*/
      return 0;
}
```

Output:

var1= 50

var2= 60

var3= 110

In the above program, 'a' is a constant type object of class 'A' and 'display()' is a constant member function of class 'A'. So, if we don't declare the variables, var1, var2 and var3 with mutable specifier then the program will return error as 'a' is a constant object.



CHECK YOUR PROGRESS

Q 6: What is the associativity of additive operators?

Q 7: The ___ storage class is the default storage class for all local variables.

Q 8: 'New' is a unary operator. (True/False).

Q 9: & is ____operator.Q10. The register storage class is used to define local variables that should be stored in a register instead of _____.



3.6 LET US SUM UP

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

- Operators are symbols that perform operations on variables and values.

- Operators in C++ can be classified into 6 types: Arithmetic Operators, Assignment Operators, Relational Operators, Logical Operators, Bitwise Operators, and Other Operators.
- Arithmetic operators are used for performing arithmetic operations on variables and data.
- Assignment operators are used for assigning values to variables and constants.
- A relational operator is used to check the relationship between two operands.
- Logical operators are used for checking whether an expression is true or false.
- Bitwise operators are used for performing operations on individual bits.
- Operator precedence determines the grouping of terms in an expression.
- The associativity of an operator is a property that determines how operators of the same precedence are grouped in the absence of parentheses.
- A storage class defines the scope (visibility) and a lifetime of variables and/or functions within a C++ Program.



3.7 FURTHER READING

1. Schildt, H. (2017). *C++: The complete reference*. New York: McGraw-Hill.
2. Balagurusamy, E. (2020). *Object-Oriented Programming with C++, 7e*. McGraw-Hill Education.
3. Yashavant, K. (2020). *Let us C*. BPB Publications.



3.8 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1: 1

Ans to Q No 2: True

Ans to Q No 3: assignment

Ans to Q No 4: Less Than or Equal To

Ans to Q No 5: !

Ans to Q No 6: Left to right

Ans to Q No 7: auto

Ans to Q No 8: True

Ans to Q No 9: Bitwise AND

Ans to Q No 10: RAM



3.9 MODEL QUESTIONS

Q1. What are operators? What are the different kinds of operators in C++?

Q2. Describe Increment and Decrement Operators with examples.

Q3. What are the different kinds of Assignment Operators?

Q4. Write a C++ program illustrating the use of different Relational Operators.

Q5. What is the difference between Precedence and Associativity? Give an example.

Q6. What are the different types of storage classes in C++?

*** ***** ***

UNIT 4: DECISION AND CONTROL STRUCTURES

UNIT STRUCTURE

4.1 Learning Objectives

4.2 Introduction

4.3 Decision Making Statements

4.3.1 The *if* statement

4.3.2 The *if-else* statement

4.3.3 The *switch-case* statement

4.4 Iterative Statements

4.4.1 *while* loop

4.4.2 *do-while* loop

4.4.3 *for* loop

4.5 Unconditional Branching Statements

4.5.1 *break* statement

4.5.2 *continue* statement

4.5.3 *goto* statement

4.6 Let Us Sum Up

4.7 Further Reading

4.8 Answers to Check Your Progress

4.9 Model Questions

4.1 LEARNING OBJECTIVES

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

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

4.2 INTRODUCTION

In the previous unit, we have discussed different types of operators and their use in programming. We have also learnt the precedence and associativity of operators.

In programming, sometimes it may happen that the programmer requires to take some decisions based on some conditions to solve a particular problem. A program is not limited to a linear sequence of instructions. C++ provides us decision and control structures for performing the same operation repetitively, for altering the flow of execution of statements, bifurcation, etc. Decision and control structures are used to specify what has to be done by our program when and under which circumstances.

This unit is essential for those who are new to C++ and have no prior knowledge of programming languages like C. In this unit, we will discuss how to use decision-making statements such as *if*, *if-else,* and *switch*, and jumping statements such as *break*, *continue,* etc. We will also discuss the various iterative statements which are used to repeat a set of actions until a specified condition is met.

4.3 DECISION MAKING STATEMENTS

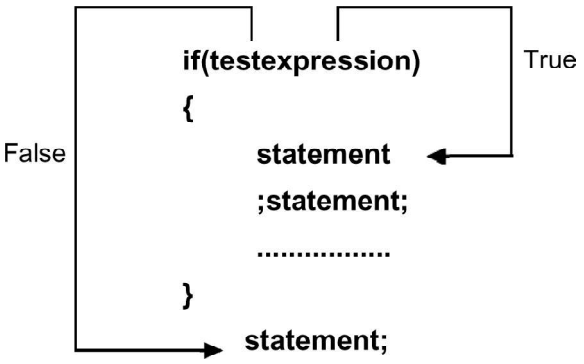
Decision-making is an important feature in any programming language. Statements like ***if***,***if-else,*** and ***switch-case*** are the decision-making statements provided by C++language.

4.3.1 The *if* statement

The ***if*** statement is the most simple decision-making statement. It contains an expression that is evaluated. If the expression evaluates *true* value then all the statements inside the braces of *if* block are executed. Otherwise, if the expression evaluates *false*, then the statement outside the *if* block is executed. The value 1 or any non-zero value is considered as *true* and 0 as *false.* The syntax of *if* statement is given here:



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.



If there is only one statement to be executed, then it can be written without the curly braces. If we do not provide the curly braces '{' and '}' after if (condition) then by default *if* statement will consider the first statement immediately below the *if* condition to be inside its block. In the following lines of code, statement 1 will be considered by default inside the *if* statement block.

```
if(condition)
statement1;
statement2;
```

A flow chart of *if* statement is provided in figure 4.1.

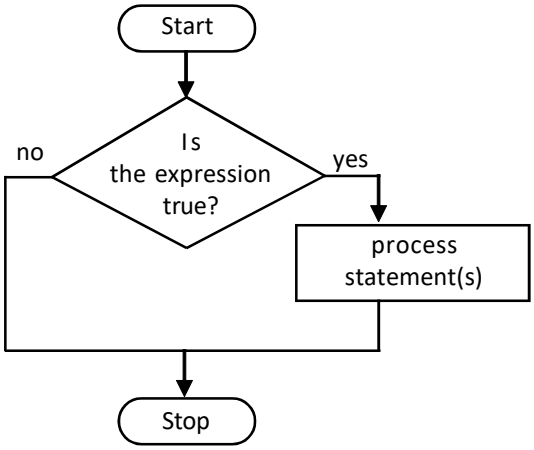


Fig.4.1:Flow chart of *if* statement

In the flowchart, it is shown that if the expression evaluates *true*, then execution of statements in the statement block will be executed and if it is *false*, then the control will come out of the *if* statement block. Concept of *if statement* will be more clear if we discuss a few programs where decision-making is required.

For example, let us consider a program where the programmer desires to display "Pass" if the marks obtained by a student is greater than or equal to 250. Thus, this can be performed as follows:

/***Program 4.1:***

Program to display "Pass" if the marks obtained by a student is greater than or equal to 250*/

```
#include<iostream.h>
int main()
      {
      floatmarks;
      cout<<"\nEnter the marks obtained by a student:";
      cin>>marks;
      if(marks>=250)
      cout<<"Pass";
      return 0;
}
```

Here in the program, there is only one statement to execute if the condition ***if (marks>=250)*** returns *true* value. Therefore, the braces "{" and "}" are not used. They are optional in such cases. If someone uses the braces, then also the program will work.

Again, if the programmer wants to display the percentage of the student if he/she is passed, then we can modify ***Program 4.1*** with the following lines of code. Here braces are used for the *if* block as follows:

/***Program 4.2:*** Demonstration of *if* statement (more than one statements available inside if statement)*/

```
#include<iostream.h>
int main()
{
  float totalmarks=500,marks,per;
  cout<<"\nEnter the marks obtained by the student:";
  cin>>marks;
  if(marks>=250)
  {
```

```
            cout<<"Pass";
            per=(marks/totalmarks)*100;
          cout<<"\nPercentage of the student is "<<per<<"%";
    }
    return 0;
}
```

/***Program 4.3:**

Program to enter age of a person and display message whether the person is eligible for voting or not*/

```
#include<iostream.h>

int main()

{

    int age;

    cout<<"\nEnter the age of the person in years:";

    cin>>age;

    if(age>=18)

        cout<<"\nYou are eligible for voting. ";

    return 0;

}
```

4.3.2 The *if- else* statement

The ***if-else*** statement is an extension of the simple ***if*** statement. It performs some action even when the test expression fails. The *else* key word is used when the expression is not *true*.

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

```
if (test expression)
{
      trueblockstatement(s);
      ..............
```

```
}
else
{
      falseblockstatement(s);
      ..................
}
statement;
```

The flowchart of *the if-else* statement is shown infigure 4.2:

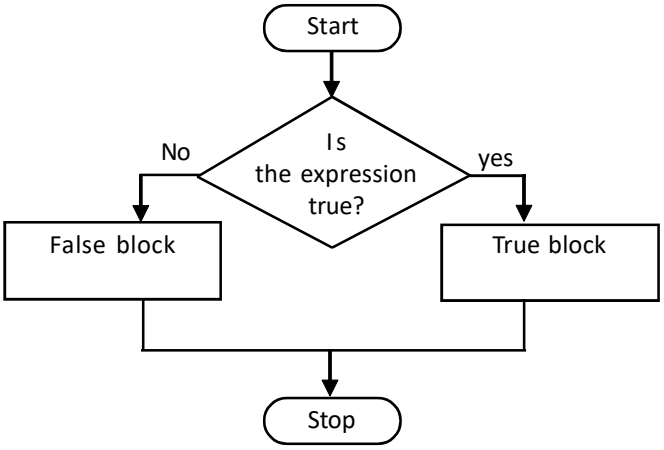


Fig.4.2: Flow chart of *if-else* statement

/***Program 4.4:*** Program to display "Pass" or "Fail" along with student's percentage. If marks obtained by the student are greater than or equal to 250 then the result will be Pass other wise Fail. There are 5 subjects and the total mark is 500. */

```
#include<iostream.h>
int main()
{
     float  totalmarks=500,marks,per;
     cout<<"\nEnter the marks obtained by the students:";
     cin>>marks;
     if(marks>=250)
     {
          cout<<"\nPass";
          per=(marks/totalmarks)*100;
          cout<<"\nPercentage  is:"<<per<<"%";
     }
```

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

Nested if: Sometimes our program may require checking multiple conditions and different actions to be performed under each condition. For such a situation **nested if** statement can be used. When we need to use an **if** statement inside another **if** statement, then it is known as **nested if** statement.

In the case of **nested if** statement, an **else** statement always refers to the nearest **if** statement that is written in the same block and not already associated with an **else**.

The **nested if** statement can be of the following forms:

a)

```
if(condition1)
{
        if(condition2)
        statement1;
        else
            statement2;
 }
else
 {
      if(condition3)
      {
          statement3;
          statement4;
       }
       else
          statement5;
}
```

b)

```
if(condition1)
{
     statement1;
     statement2;
     ........
 }
 else
 {
      if(condition2)
           statement3;
      else
           statement4;
     if(condition3)
          ........
 }
```

c)

```
if(condition1)
{
     if(condition2)
      statement1;
      else
           statement2;
 }
 else
 {
      statement3;
       .........
 }
```

In the following program, use of nested if is shown.

/***Program 4.5**: Program to in put a character and print whether it is a letter, digit or any other character (use of nestedif statement)/*

```
#include<iostream.h>
int main()
{
 char ch;
 cout<<"Enter the character:";
 cin>>ch;
 if(((ch>='A')&&(ch<='Z')) ||(ch>='a'&&(ch<='z')))
     cout<<"It is a letter"<<endl;
 else
 {
      if(ch>='0'&& ch<='9')
         cout<<"It is a digit"<<endl;
      else
         cout<<"It is some other character";
 }
 return 0;
 }
```

if-else-if Ladder

Multi-way decisions need to be taken when there are multiple conditions available and different actions to be performed under each condition. In C++, **if-else-if ladder** statement helps the user to select among multiple options.

In the case of **if-else-if ladder,** the conditions/expressions are evaluated from the top to the bottom. As soon as the true condition is found, the statement associated with it is executed and the rest of the ladder is by-passed. If none of the conditions is true, then **else** is executed. The syntax is presented as follows.

```
if (condition1)
        statement;
else if (condition2)
        statement;
```

....

....

else

statement;

Because of the appearance, this construct is referred to as **if-else-if ladder** statement. The following program shows the use of **if-else-if ladder**.

/***Program 4.6*: Program to display division and percentage of a student ifhe/she has passed the examination. There are 5 papers and each paper carries 100 marks. If marks obtained by the student are greater than orequal 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 are less than 150 then the result should be Fail.*/

```
#include<iostream.h>
int main()
{
    float totalmarks=500,marks,per;
    cout<<"\nEnter the marks obtained by the student:";
    cin>>marks;
    if(marks>=300)
    {
        cout<<"\nPass. Student has got First Division";
        per=(marks/totalmarks)*100;
        cout<<"\nPercentage is: "<<per<<"%";
    }
    else if(marks>=225)
    {
      cout<<"\nPass. Student has got Second Division";
    per=(marks/totalmarks)*100; cout<<"\nPercentage is: "<<per<<"%";
    }
    else if(marks>=150)
    {
        cout<<"\nPass. Student has got Third
        Division";per=(marks/totalmarks)*100;
        cout<<"\nPercentage is: "<<per<<"%";
    }
    else
        cout<<"\nFail";
        return 0;
    }
```

/***Program4.7**:

The program will display tariff charges as per data pack in GB. Data pack required will be taken asuser input.*/

```
#include<iostream.h>

int main()
{
    int pack;
    cout<<"\Enter data pack required in GB";
    cin>>pack;
    if(pack==1)
       cout<<"Rs.11/-"<<endl;
    else if(pack==2)
        cout<<"Rs.21/-"<<endl;
    else if(pack==6)
        cout<<"Rs.51/-"<<endl;
    else if(pack==12)
        cout<<"Rs.101/-"<<endl;
    else if(pack==30)
        cout<<"Rs.151/-"<<endl;
    else
     cout<<pack<<" GB data pack not available fornow"<<endl;

     return 0;
```

4.3.3 The *switch-case* statement

Another way for multi-way decision making other than **if-else-if** statements is possible with the use of **switch case** statement. The **switch** statement is suitable when many conditions are being tested for. Switch statement successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed. The general form of a ***switch*** statement is:

switch (*expression*)

{

case *constant 1*:

```
        statement 1 sequence
        break;
    case constant 2:
        statement 2 sequence
        break;
    case constant 3:
        statement 3 sequence
        break;
    .....
    .....
    default:
        statement sequence
}
```

The ***expression*** value must be always integral. The expression to be tested comes after the ***switch*** statement and it can be avariable, 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 *constant 1.* If it is, it executes *statement 1 sequence* until it finds the ***break*** statement. When it finds this *break* statement the control jumps to the end of the ***switch*** block.

If the *expression* was not equal to *constant 1* it will be checked against *constant 2*. If it is equal to this, it will execute *statement 2 sequence* until a *break* keyword is found, and then will jump to the end of the ***switch*** block. Finally, if the value of *expression* didnot match any of the previously specified *constants*, the program will execute the *statements* included after the ***default*** statement, if it exists. If the ***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 don't include break

statement, all the statements for the *cases* following the one selected will be executed. For example, if *constant 1* 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 constant 2*, *case constant 3*, etc., and even *default* will be executed. The flow chart of switch statement is given as in figure 4.3.

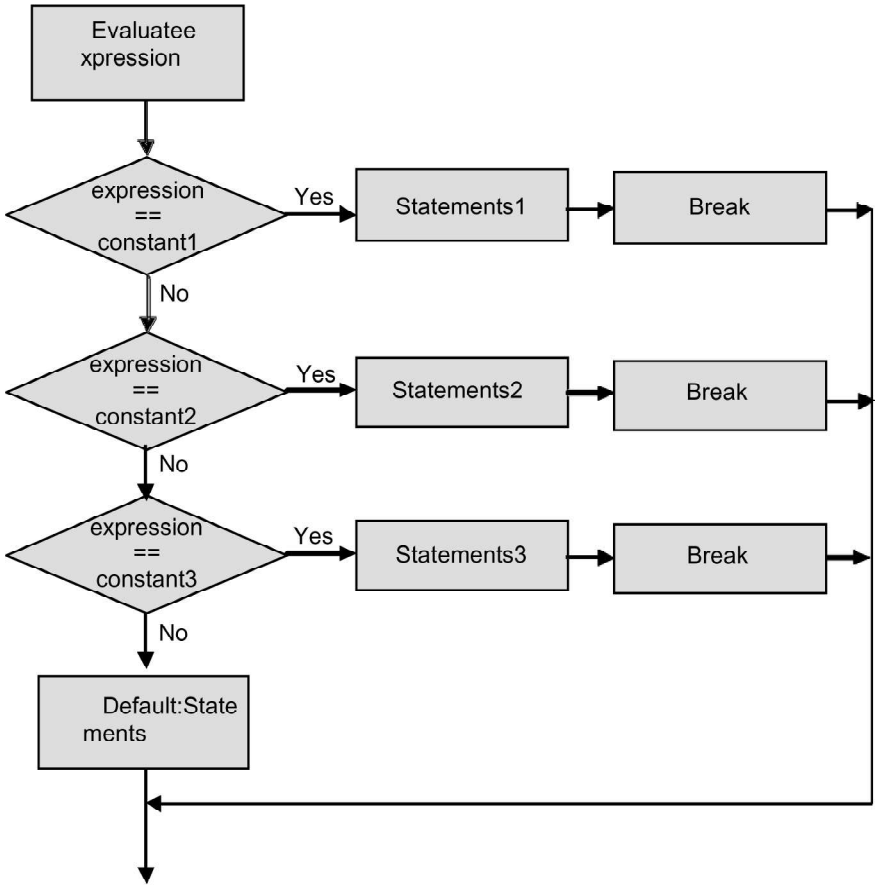


Fig.4.3: Flow chart of *switch* statement

It is important to know that no two ***case constants*** in the same ***switch*** can have identical values.

Nested switch statement:

We can write one switch statement inside another switch statement i.e., switch statement can be of nested type. Even if the case constants of the inner and outer switch contain common values, no conflict arises. This means a switch statement enclosed by an outer switch may have case constants that are the same.

Let us consider an example to calculate the area of circle/ rectangle/triangle as per user choice. If 1 is pressed in the keyboard, then an area of circle should appear. Similarly, for 2 and 3, areas of rectangle and triangle should appear. Other parameters like radius, length, breadth, baseetc. should be entered through the keyboard.

/***Program 4.8***: Calculating the area of circle, rectangle, triangle based on user choice*/

```
#include<iostream.h>
int main()
{
    float  area,radius,length,breadth,base,height;
    int ch;

    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)
    {
      case 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:
        cout<<"\nEnter the length & breadth :";
        cin>>length>>breadth;
       area=length*breadth;
                                //area=length*bre
      adth
        cout<<"\nRectangle area is "<<area<<" sq.units";
```

```
    break;
case 3:
 cout<<"\nEnter triangle base and height:";
 cin>>base>>height;
 area=(base*height)/2.0;//area=(base*height)/2.0
 cout<<"\nTriangle area is "<<area<<" sq.units";
       default:
            cout<<"\nYour choice is not valid";
            break;
     }
     return 0;
 }
```

If the choice is 2, then the output will be like this:

```
1.Area of Circle::
2.Area of Rectangle::
3.Area of Triangle::

Enter your choice(1,2,or 3):2

Enter the length & breadth :6 4

Rectangle area is 24 sq.units_
```



CHECK YOUR PROGRESS

Q 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* construct 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.*

Q 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 with in a *switch* statement.

(iv) In the 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.

4.1 ITERATIVE STATEMENTS

In programming, we often come across some situations where we are to perform some tasks repeatedly. C++ and all other modern programming languages provide ***iterative statements,*** also called ***loops***, which allow a set of instructions to be executed repeatedly until a certain condition ismet. The iterative statement allows us to repeat lines of code as many times as we need instead of typing the same code again and again. 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 in C++ are exactly the same a sin C language.

4.4.1 *while* loop

The **while** loop is an entry-controlled loop as the condition is tested at the beginning. If the condition is true, then only the statements inside the body of the loop will be executed. The syntax of while loop is as follows:

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

After the execution of the body, the test condition is again evaluated and if it is true, the body of the loop 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 continue swith 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 with in the block some method to for cethe condition to become false at some point, otherwise, the loop will continue looping forever.

In **Program 4.9***,* the user is prompted to insert a starting number forthe countdown. After that, the while loop begins, based on the value entered by the user and on fulfilling the condition (a>0), the block that follows the condition gets executed. This is continued till the condition (a>0) is true. The output statement displays the value of a on the screen and decreases a by 1.

```
cout<<a<<"";
```

/***Program 4.9***:* Program to display numbers up to a specified limit using whileloop*/

```
#include<iostream.h>
int main()
{
      int a;
      cout<<"Enter a number to start with:";
      cin>>a;
      while(a>0)
      {
            cout<<a<<" ";
            - - a;
      }
      cout<<"\n The numbers are displayed."<<"\n";
      return 0;
}
```

The output of the program will be as follows if the number starts with 9.

```
Enter a number to start with :9
9  8  7  6  5  4  3  2  1
The numbers are displayed.
```

In the following program (***Program4.10***), while loop is used to display all the odd numbers between1 to100.

//**Program 4.10*:*** Program to display odd numbers from1 to100

```
#include<iostream.h>
int main()
{
      int i=1;
      //while loop from1 to100
      while(i<=100)
      {
      cout<<i<<" ";
      i=i+2;
      }
    return 0;
}
```

Let us take another small example of a program that displays "Assam" four(4) times. Here we have written the statement **i++;** within the

body of while loop, that increases the value of the variable 'i' by one that is being evaluated in the condition. This will eventually make the condition (i<=4) become false after a certain number of loop iterations.

//**Program 4.11***:* Demonstration of *while* loop # include<iostream.h>

```
int main()
{
int i=1;
while(i<=4)
{
      i++;
      cout<<"Assam"<<endl;
      }
      return 0;
}
```

The output will be:

Assam

Assam

Assam

Assam

While loop is used for displaying the multiplication table of a number entered through the keyboard in the following **Program 4.12**.

/****Program 4.12:*** Displaying multiplication table of a number using while loop*/

```
#include<iostream.h>
int main()
{
      int n,i=1;
      cout<<"\n Enter a number for multiplication
table:";
      cin>>n;
      while(i<=10)
      {
      cout<<n<<"*"<<i<<"="<<n*i<<endl;
```

```
        i++;
        }
        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

4.4.2 *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 atleast 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 4.12 for demonstration of *do-while* loop.

//Program 4.13: Displaying multiplication table using *do-while* loop.

```
#include<iostream.h>
int main()
{
int n,i=1;
      cout<<"\n Enter a number for multiplication table:";
      cin>>n;
      do   //beginning of do-while loop
      {
      cout<<n<<"*"<<i<<"="<<n*i<<endl;
      i++;
      } while (i<=10);      //end of do-while loop
      return 0;
}
```

//Program 4.14:/*Program to find the sum of positive numbers entered through the keyboard. If the user enters a negative number, the loop terminates; the negative number entered is not added to the sum*/

```
#include<iostream.h>
int main()
{
      int num= 0;
      int sum= 0;
      do
      {
            sum=sum+num;
            cout<<"Enter a number:";
            cin>>num;
      }
      while(num>=0);
      cout<<"\nThe sum is:"<<sum<<endl;
      return 0;
}
```

In the above program, the do-while loop will continue until a negative number is entered. The program will give the sum of all positive numbers entered by the user.

4.4.3 *for* loop

The ***for*** loop provides a more concise loop control structure. The for loop allows many variations. 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 set 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. Ifthe *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 a demonstration of *for* loop, let us take an example of displaying all odd numbers from 1 to 20.

//**Program 4.15*:*** Displaying odd numbers between 1 to 20

```
#include<iostream.h>
int main()
{
      int i;
      cout<< "The odd numbers from 1 to 20 are:" <<endl;
```

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

The output will be like this:

The odd numbers from 1 to 20 are:

1 3 5 7 9 11 13 15 17 19

The loop will execute 20 times in the program. 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. In our previous example, we can write the loop to include the declaration for the loop counter ***i*** as follows:

```
        //iis declared and in itialized inside for
        for(int  i=1;i<=20;i++)
        {
        if(i%2!=0)
        cout<<i<<"\t";
        }
```

We can also omitthe initialization part altogether. If we initialize *i* appropriately in the declaration, wecanwritetheloopas:

```
        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. Infact, both semicolons must be in place. This flexibility also applies to the contents of the increment/decrement part.

/***Program 4.16*** Combining two strings without using string library functions. (for loop without any body)*/

```
#include<iostream.h>
int main()
{
     char a[20],b[20],c[40];
     int  length1,length2,i,j;
     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;
     return 0;
}
```

Let us consider the following statements from the above program.

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

In these statements, we have used these micolon after the closing parentheses, to indicate that the loop body is empty. If we omit the semicolon, the statement immediately following the loop will be interpreted as the loop statement. In ***Program 4.16,*** these two for loops are simply used to determine the length of the first and these condstring.

The infinite loop:

When a test *condition* specified in a loop evaluates and returns true for

ever 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 4.17: Example of infinite loop; program will run indefinitely.

```
#include<iostream.h>
int main()
{
      inti=1;
      while(i)
      {
            cout<<i;
      i++;
      }
            return0;
}
```

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

4.5 UNCONDITIONAL BRANCHING STATEMENTS

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

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

4.5.1 *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 these quence of items has been completely iterated over. So, the *break* statement can be used to terminate a loop immediately. The break statement can be written as:

break;

With in 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* with in 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 with in the *switch* statement. The flowchart of break statement is shown in figure 4.4

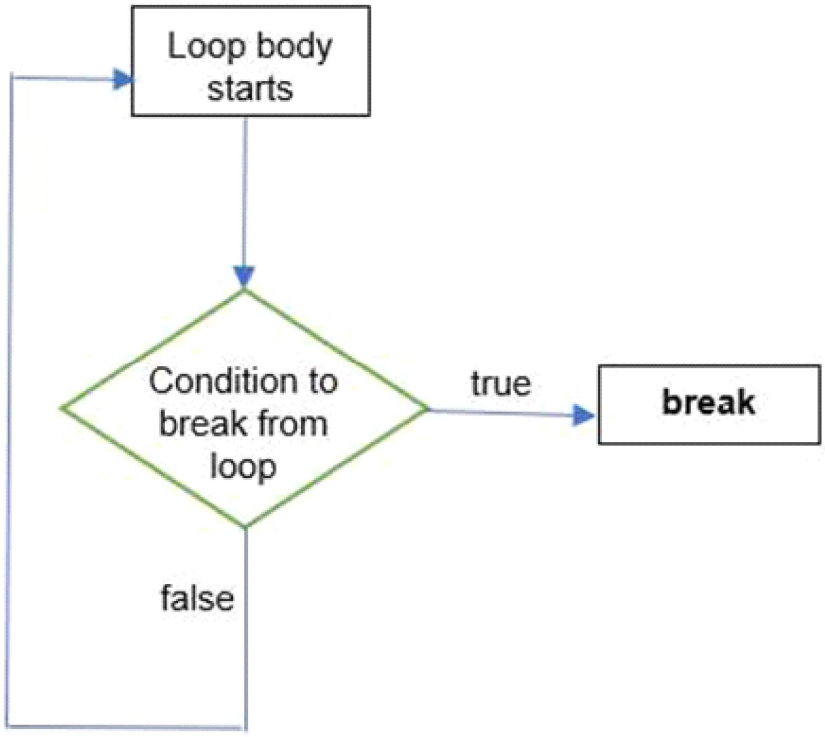


Fig.4.4:Flowchartof*break*statement

Useof*break* statementisshown in **Program 4.18**

//**Program 4.18** Displaying the summation of all entered numbers

```
#include<iostream.h>
int main( )
{
     int i,n,sum=0;
     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;
     return 0;
}
```

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

4.5.2 *Continue* statement

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

continue;

The flow chart of *continue* statement is shown in figure 4.5.

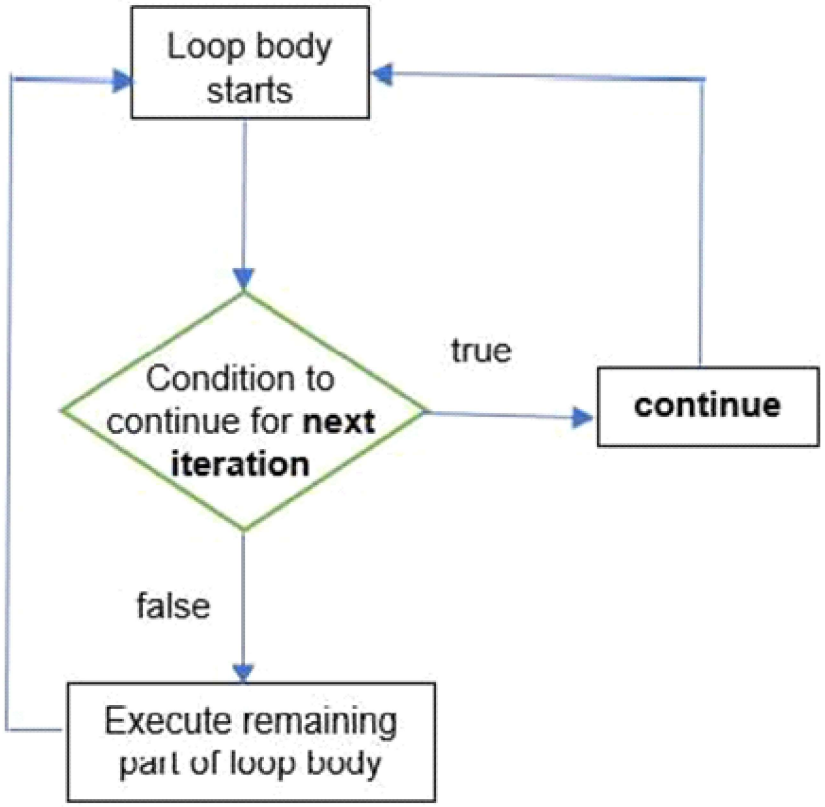


Fig.4.5: Flow chart of *continue* statement

Usually, *continue* statement is associated with an *if statement*. The *continue* statement in *for*, *while,* and *do-while* loop are presented 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 4.19:*** Program to find the summation of all even numbers from 1 to10*/

```
#include<iostream.h>
int main()
{
      intn,i,sum=0;
      cout<<"\n The even numbers from 1 to 10 are:";
      for  (i=1;i<=10;i++)
         {
      if(i%2!=0) //ifi%2!=0i.e.,i is an odd number
      continue;//it will skip the odd no. and incrementi
      cout<<"\t"<<i;
      sum=sum+i;
      }
      cout<<"\n The summation of all even numbers is:"<<sum;
      return0;
}
```

4.5.2 *goto* statement

With the help of *goto* statement, one can transfer the control to any where 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 demonstrate the *goto* statement with an example.

/***Program 4.20:*** Calculating the summation of first *n* natural numbers using *goto* statement.(value of *n* should been tered through keyboard)*/

```
#include<iostream.h>
int main()
{
    int limit, num,sum=0;
    cout<<"\nEnter the limit for addition of naturalnumbers\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 numberis :"<<sum;
    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 be have like a loop from 1 to 9 in the above program.

```
num=1;
target:sum=sum+num; //target used as label
if(num<limit)
{
      num++;
      goto target;     //goto 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. Occasional situations do arise, however, in which the goto statement canbe useful. In a difficult programming situation, it seems so easy to use a *goto* to take control where you want to.



CHECK YOUR PROGRESS

Q 1: Fill in the blanks:

i) _____ statement allows us to repeat lines of code as many times as we need.

ii) The braces "{"and"}" are needed only if the body of the loop contained _____ statements.

iii) Do-while loop tests the condition at the_____ of the loop after executing the body of the loop.

iv) With in nested loop, the_____ statement terminates only the *inner* loop that immediately encloses it.

v) When a test condition specified in a loop evaluates and returns true forever then statements with in the loop are executed_____.

vi) The_____statement causes the program to skip the rest of the statements of the loop in the current iteration and causes it to jump to the start of the following iteration.



4.6 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 statements 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 the 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* sequences which is difficult to maintain in a program.
- The **switch** statement is often used to process keyboard commands, such as menu selection.
- 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 statements associated with that constant are executed.
- A program may require that a group of instructions is executed repeatedly 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 execute sat least 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 doesnot terminate when a *continue* statement is encountered.
- The ***goto*** statement unconditionally transfers control to the statement labeled by the specified identifier.



4.7 FURTHER READING

1. Venugopal, K. R. Rajkumar and Ravishankar T. *Mastering C++,* Tata McGraw Hill Publishing Co. Ltd., New Delhi.
2. Kamthane, A. (2003). *Object-oriented Programming with ANSI and Turbo C++*. Pearson Education India.



4.8 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q.No.1:

(i) (c) Control falls through to the statement following the *if* statement

(ii) (c) The *switch* is easier to understand.

Ans. to Q.No.2:

(i) True (ii) True (iii) False (iv) True (v) True

Ans. to Q.No.3:

(i) Iterative (ii) twoormore (iii) bottom

(iv) break (v) infinitely (vi) continue



4.9 MODEL QUESTIONS

Q1. What is the purpose of *if-else* statements?

Q2. What are the differences between *break* and *continue* statements? Explain with examples.

Q3. What is the purpose of the *break* statement? Within which control statements can the *break* statement be included?

Q4. What is the purpose of *continue* statement? Compare it with the *break* statement?

Q5. What is the purpose of the *goto* statement? How are the associated target statements identified?

Q6. 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 doesnot match any of the case labels?

Q7. What is iterative statement? What are its types? Give their syntax with an example.

Q8. What is the purpose of *do-while* loop? What is the minimum number of times that a *do-while* loop can be executed?

Q9. What are the differences between *while* and do-while loop?

Q10. What is an infinite loop? Write a C++ program to show the use of infinite loop.

Q11. Write a C++ program to check whether a given number is prime or not?

Q12. Write a C++ program to determine the sum of all digits of an entered number.

Q13. Write a C program to enter two characters and display the larger character. Use nested if-else statements.

Q14. Write a C++ program to find out whether a year is a leap year or not?

Q15. How is the break statement useful in both switch and loops? Explain with an example.

Q16. Write a C++ program to find the largest of the three numbers using *if-else* statement. Numbers should be entered through the keyboard.

Q17. Write a C++ program using *switch* statement that will examine thevalue 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 *temperature* 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.

Q18. 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;
}
```

Q19. Write a C++ program that displays every integer between 1 to 50 divisible by m (a value less than 10). Also indicate whether the number divisible by miseven or odd.

Q20. Explain how the for loop may be varied. Also, discuss the implications of such variations.

*** ***** ***

UNIT 5: ARRAYS AND STRUCTURES

UNIT STRUCTURE

5.1 Learning Objectives

5.2 Introduction

5.3 Arrays in C++

5.3.1 Declaring and Initializing Arrays

5.3.2 Accessing Array Elements

5.4 Multi-Dimensional Arrays

5.4.1 Two-Dimensional Arrays

5.4.2 Initializing and Accessing Two-Dimensional Array Elements

5.5 C++ Structures

5.5.1 Array of Structures

5.6 Unions

5.7 Let Us Sum Up

5.8 Further Reading

5.9 Answer To Check Your Progress

5.10 Model Questions

5.1 LEARNING OBJECTIVES

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

- learn about Arrays in C++
- learn about Structures and Unions
- learn about runtime memory management in C++

5.2 INTRODUCTION

In the previous unit we have learnt different types of decision making, iterative and unconditional branching statements. In this unit we will learn to use arrays, structures and union in C++ programming language.

An array in C++ or in any programming language is a collection of similar data items stored at contiguous memory locations and elements

can be accessed randomly using indices with array name of that array. They can be used to store data of primitive data types such as int, float, double, char, etc. Structure and union are user defined heterogeneous data type which allows us to hold data items of different kinds.

5.3 ARRAYS IN C++

C++ provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type. Instead of declaring individual variables, such as number0, number1, ..., and number99, we can declare one array such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by using the array name and indices that are called as subscript values. In case of one-dimensional array, only one subscript value is associated with one array element.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

5.3.1 Declaring and Initializing Arrays

To declare an array in C++, the programmer specifies the type of the elements, array name and the number of elements that can be stored in that array. The syntax to declare a single-dimensional array is presented as follows.

type array Name [array Size];

Here, 'type' is any valid C++ data type and 'arrayName' is the name of the array.The'arraySize' must be an integer constant greater than zero.For example, the following statement can be used to declare a 10-element array,'balance', where data of type double will be stored.

double balance[10];

Now, we can initialize C++ array elements either one by one or using a single statement, presented as follows.

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

Or,

```
double balance[5];
balance[0] = 1000.0;
balance[1] = 2.0;
balance[2] = 3.4;
balance[3] = 17.0;
balance[4] = 50.0;
```

In case of the single statement initialization, the number of values between braces, { } cannot be larger than the number of elements that we declare for the array within the square brackets, [ ].Now, if we omit the size of the array in that array declaration then an array just big enough to hold the initialized values will be created. Therefore, if we write the following statement then the same array as in the previous example will be created.

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The index or subscript value of a one-dimensional array is started with 0 in C++.So, if the array size of an array is N then the subscript value of the last element will be N-1. The index of the first element is referred as base index and the memory address
of the first element is referred as the base address. Apictorial representation of the array that is initialized in the above example, presented as follows.

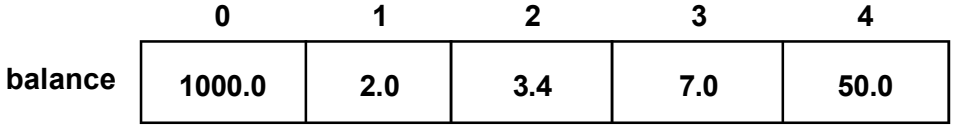


Figure 5.1: Pictorial Representation of a One-Dimensional Array

5.3.2 Accessing Array Elements

An array element can be accessed by placing the index or subscript value of the element within square brackets after the name of the array. For example –

double salary = balance[9];

In the above statement, the 10th element from the array, 'balance' is assigned to the variable, 'salary'.

Program 5.1: C++ program to show array declaration, assignment, and accessing array elements.

```
#include <iostream.h>
#include <iomanip.h>
int main ()
{
      int n[ 10 ]; // n is an array of 10 integers

            // initialize elements of array n to 0
      for ( int i = 0; i< 10; i++ )
      {
   n[ i ] = i + 100;//set element at location i to i + 100
      }
      cout<< "Element" <<setw( 13 ) << "Value" <<endl;

      // output each array element's value
      for ( int j = 0; j < 10; j++ )
      {
      cout<<setw( 7 )<< j <<setw( 13 ) << n[ j ] <<endl;
      }

      return 0;
}
```

The above program makes use of **setw()** function to format the output. When the above code is compiled and executed, it produces the following result “

Element	Value
0	100
1	101
2	102
3	103
4	104
5	105
6	106
7	107
8	108
9	109

Program 5.2: C++ program to find out the summation of all the numbers available in an integer array.

```
#include<iostream.h>
int main()
{
intAr[100], i, N, sum=0;
cout<<"\n Enter the value of N:";
cin>>N;
if(N>100)
cout<<"\n Wrong input";
else
{
cout<<"\n Enter Array elements:";
for(i = 0 ; i< N ; i++)
     {
     cout<<"\n Enter "<<i+1<<"element:";
     cin>>Ar[i];
     sum = sum + Ar[i];
```

```
   }
cout<<"\n The array elements are:\n";
for(i = 0 ; i< N ; i++)
     {
     cout<<"\t"<<Ar[i];
     }
cout<<"\n Summation of the array elements is="<<sum;
 }
return 0;
}
```

Output:

Enter the value of N: 5

Enter Array elements:

Enter 1 element: 8

Enter 2 element: 10

Enter 3 element: 1

Enter 4 element: 54

Enter 5 element: 2

Summation of the array elements is= 75

Program 5.3: C++ program to sort array elements using Bubble sort algorithm.

```
#include<iostream.h>
int main()
{
intAr[100], i, j, N, temp;

cout<<"\n Enter the value of N:";
cin>>N;
if(N>100)
cout<<"\n Wrong input";
else
  {
```

```
cout<<"\n Enter Array elements:";
for(i = 0 ; i< N ; i++)
 {
      cout<<"\n Enter "<<i+1<<"th element:";
      cin>>Ar[i];
}
cout<<"\n Before sorting, the array elements are:\n";
for(i = 0 ; i< N ; i++)
{
      cout<<"\t"<<Ar[i];
 }
for(i=0;i<N-1;i++)
{
      for(j=0;j<N-1-i;j++)
      {
      if(Ar[j]>Ar[j+1])
       {
            temp=Ar[j];
            Ar[j]=Ar[j+1];
            Ar[j+1]=temp;
}
      }
}
cout<<"\n After sorting, the array elements are:\n";
for(i = 0 ; i< N ; i++)
 {
      cout<<"\t"<<Ar[i];
 }
}
return 0;
}
```

Output:

Enter the value of N: 6

Enter Array elements:

Enter 1th element: 12

Enter 2th element: 3

Enter 3th element: 45

Enter 4th element: 6

Enter 5th element: 78

Enter 6th element: 9

Before sorting, the array elements are:

12 3 45 6 78 9

After sorting, the array elements are:

3 6 9 12 45 78

5.4 MULTI-DIMENSIONAL ARRAYS

Multi-dimensional arrays are also supported in C++. The general form of an N-dimensional array declaration is presented as follows.

type name [size1] [size2]...[sizeN];

For example, the following declaration creates a three dimensional integer array –

int three dim [5] [10] [4];

5.4.1 Two-Dimensional Arrays

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. The syntax to declare a two-dimensional array in C++ is presented as follows.

type array Name [X][Y];

Where 'type' can be any valid C++ data type and 'arrayName' will be a valid C++ identifier. X and Y are two positive integer constants. Now, the two-dimensional array, 'array Name' can be thought of as a table, which will have X number of rows and Y number of columns.

A pictorial representation of a 2-dimensional array, 'a', which contains three rows and four columns can be shown as below –

	Column 0	**Column 1**	**Column 2**	**Column 3**
Row 0	**a[0] [0]**	**a[0] [1]**	**a[0] [2]**	**a[0] [3]**
Row 1	**a[1] [0]**	**a[1] [1]**	**a[1] [2]**	**a[1] [3]**
Row 2	**a[2] [0]**	**a[2] [1]**	**a[2] [2]**	**a[2] [3]**

Figure 5.2: Pictorial representation of a 2-Dimensional Array

Thus, every element in array,'a' is identified by the form, a[i][j]. Here 'a' is the name of the array, and i and j are the subscript values that uniquely identify each element in 'a'.

5.4.2 Initializing and Accessing Two-Dimensional Array Elements

Two-dimensional arrays can be initialized in three ways as shown below:

Method i: int a[3][4] =

```
{
        {0, 1, 2, 3},/*  initializers for row indexed by 0 */
        {4, 5, 6, 7}, /*  initializers for row indexed by 1 */
        {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

Method ii: int a[3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};

Method iii: int a[ ][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};

In C++, to initialize a two-dimensional array, the second subscript value of the array must be declared. For example, the following initialization of a two dimensional array is not valid in C++.

int a[3][ ] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};

An element in a 2-dimensional array is accessed by using the array name and the subscripts, i.e., row index and column index of the array. For example –

int val = a[2][3];

In the above statement,the 4th element from the 3rd row of the array, a is assigned to the variable, val.

Program 5.4: C++ program to show initialization and accessing 2-dimensional array.

```
#include <iostream.h>
int main ()
{
      // an array with 5 rows and 2 columns.
      int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
      // output each array element's value
      for ( int i = 0; i< 5; i++ )
      for ( int j = 0; j < 2; j++ )
      {
      cout<< "a[" <<i<< "][" << j << "]: ";
      cout<< a[i][j]<<endl;
      }
      return 0;
}
```

When the above code is compiled and executed, it produces the following result –

a[0][0]: 0

a[0][1]: 0

a[1][0]: 1

a[1][1]: 2

a[2][0]: 2

a[2][1]: 4

a[3][0]: 3

a[3][1]: 6

a[4][0]: 4

a[4][1]: 8

Program 5.5: C++ program to input two matrices using two 2-dimensional arrays and find out the summation of those matrices.

```
#include<iostream.h>
int main()
{
int Matrix1[20][20], Matrix2[20][20], MatrixSum[20][20];
 // Declaration of 2-dimensional arrays
int i, j, row, column;
cout<<"\n Enter the number of rows:";
cin>>row;
cout<<"\n Enter the number of columns:";
cin>>column;
if(row > 20 || column > 20)
      cout<<"\n Wrong input";
else
{
      cout<<"\n Input 1st Matrix:";
      for(i=0 ; i<row; i++)
      {
      for(j=0; j<column; j++)
        {
      cout<<"\n Enter ("<<i<<","<<j<<")th element:";
      cin>>Matrix1[i][j];
        }
```

```
        }
        cout<<"\n Input 2nd Matrix:";
        for(i=0 ; i<row; i++)
        {
        for(j=0; j<column; j++)
          {
        cout<<"\n Enter ("<<i<<","<<j<<")th element:";
        cin>>Matrix2[i][j];
          }
        }
        for(i=0 ; i<row; i++)
        {
for(j=0; j<column; j++)
          {
        MatrixSum[i][j] = Matrix1[i][j] + Matrix2[i][j];
// Summation of Matrix1 data with Matrix2 data

          }
        }
        cout<<"\n 1st Matrix is:\n";
        for(i=0 ; i<row; i++)
        {
        for(j=0; j<column; j++)
          {
        cout<<"\t"<<Matrix1[i][j];
          }
        cout<<"\n";
        }
        cout<<"\n 2nd Matrix is:\n";
        for(i=0 ; i<row; i++)
        {
        for(j=0; j<column; j++)
          {
```

```
        cout<<"\t"<<Matrix2[i][j];
          }
        cout<<"\n";
        }

        cout<<"\n Result of Matrix summation is:\n";
        for(i=0 ; i<row; i++)
        {
        for(j=0; j<column; j++)
          {
        cout<<"\t"<<MatrixSum[i][j];
          }
        cout<<"\n";
        }
  }
return 0;
 }
```

The output of the above program is presented as follows.

Enter the number of rows:2

Enter the number of columns:2

Input 1st Matrix:

Enter (0,0)th element:1

Enter (0,1)th element:2

Enter (1,0)th element:3

Enter (1,1)th element:4

Input 2nd Matrix:

Enter (0,0)th element:4

Enter (0,1)th element:2

Enter (1,0)th element:1

Enter (1,1)th element:2

1st Matrix is:

1 2

3 4

2nd Matrix is:

4 2

1 2

Result of Matrix summation is:

5 4

4 6



CHECK YOUR PROGRESS

Q 1: An array is a collection of (similar/dissimilar) data items.

Q 2: All arrays consist of _____ memory locations.

Q 3: Array size must be greater than zero. True/False.

Q 4: C++ does not allow multi-dimensional arrays. True/False

5.5 C++ STRUCTURES

A structure is a user-defined data type in C++. Structure can be used to create a data type that can be used to group items of possibly different types into a single type.

The Syntax of Structure is presented as follows.

```
struct structure_name
    {
        // member declarations.
    }
```

In the above declaration, a structure is declared by preceding the keyword 'struct' followed by the identifier (structure name). Inside the curly braces, we can declare the member variables of different types. For example, let us consider the following declaration of a structure.

```
struct Student
{
        char name[20];
        int id;
        int age;
}
```

In the above declaration, 'Student' is a structure that contains three variables name, id, and age. When the structure is declared, no memory is allocated. Memory is allocated when the variable of a structure is created.

The syntax to declare a variable of a structure is presented as follows.
Structure_Name Structure_Variable;

Where Structure_Name is the name of a declared structure and Structure_Variable is the variable of that structure. For example, variable of the structure, 'Student' can be declared as follows-

```
struct Student s;
```

Here, s is a structure variable of type 'structStudent'. The structure, 'Student' contains one char variable and two integer variables. Therefore, the memory for one char variable is 1 byte and two integer variables will be 2*4 = 8 byte. The total memory occupied by the variable, s is 9 byte.

The member variable of a structure can be accessed by simply using the instance of the structure followed by the dot (.) operator and then the field of the structure.

For example, let us consider the following statement.

```
s.id = 4;
```

In the above statement, we are accessing the id field of the structure,'Student' by using the dot(.) operator and assigning the value 4 to the id field where s is the variable of type struct Student.

Program 5.6: C++ program to show the declaration of structure and accessing its fields.

```
#include <iostream.h>
struct Rectangle
{
      int width, height;
};
int main()
{
      struct Rectangle rec;
      rec.width=8;
      rec.height=5;
      cout<<"Area of Rectangle is:
      "<<(rec.width * rec.height)<<endl;
      return 0;
}
```

Output:

Area of Rectangle is: 40

5.5.1 Array of Structures

In C++, array of structures can also be declared and accessed.An array of structures is a collection of variables of the same structure stored in contiguous memory locations. The syntax to declare an array of structures is presented as follows.

structstr_Name strArr[Size];

Here,

str_Name - The name of a structure

strArr – The name of the array of structures

Size – It is a positive integer constant referring size of the array

The syntax to access an array of structures is presented as follows.

s = strArr[i].member1 ;

Here,

s – A variable

strArr – The Name of the array of structures

i – Subscript value of a structure available in strArr

member1 – A member of the structure. The data types of member1 and the variable, s are same.

Program 5.7: C++ program to show the declaration and accessing of array of structures.

```
#include<iostream.h>
#include<stdio.h>
struct Student {
          introllNo;
          char Name[200];
          int percentage;
};
int main()
{
struct Student St[100];
int i,N;

cout<<"\n The memory size of St is:"<<sizeof(St);
cout<<"\n Enter the number of students:";
cin>>N;
if(N>100)
cout<<"\n Wrong Input";
else
    {
     cout<<"\n Enter Student information:";
     for(i=0;i<N;i++)
     {
     cout<<"\n Enter "<<i+1<<"th student's record:\n";
     cout<<"\n Enter Roll number=";
```

```
        cin>>St[i].rollNo;
        cout<<"\n Enter Percentage:";
        cin>>St[i].percentage;
        cout<<"\n Enter Name:";
        gets(St[i].Name);
         }
        cout<<"\n Students Records:";
        for(i=0;i<N;i++)
        {
        cout<<"\n"<<i+1<<"th student's record:\n";
        cout<<"\n Roll number="<<St[i].rollNo;
        cout<<"\n Name="<<St[i].Name;
        cout<<"\n Percentage="<<St[i].percentage;
         }
       }
return 0;
 }
```

Output:

The memory size of St is: 20400

Enter the number of students: 2

Enter Student information:

Enter 1th student's record:

Enter Roll number: 1

Enter Percentage: 87

Enter Name:Pranab Das

Enter 2th student's record:

Enter Roll number: 2

Enter Percentage: 81

Enter Name:AnkurSarma

Students Records:

1th student's record:

Roll number= 1
Name=Pranab Das
Percentage= 87

2th student's record:
Roll number= 2
Name=AnkurSarma
Percentage= 81

5.6 UNIONS

A union is also a user-defined datatype. All the members of a union share same memory locations. Size of a union is decided by the size of the largest member of that union. Unions are similar to structures. But if we want to use same memory locations for two or more membersthen union can be used instead of structure.Union variables are created in the same manner as structure variables. The keyword "union" is used to define unions in C++ language.The syntax to declare a union is presented as follows.

```
union union_name
{
member definition;
};
```

Here,
union_name " Any name given to the union.
member definition " Set of member variables.

Program5.8:C++ program to show the declaration and accessing of union.

```
#include <iostream.h>
union Data
{
      chari;
      float f;
};
int main( )
{
```

```
union Data data1 ;
      data1.i = 'W';
cout<< "\n Character stored in variable i is:";
cout<<data1.i;
      data1.f = 12.45;
cout<< "\n Value stored in variable f is:";
cout<<data1.f;
cout<<"\n Memory size occupied by data1is:";
cout<<sizeof(data1);
return 0;
}
```

Output :

Character stored in variable i is: W

Value stored in variable f is: 12.45

Memory size occupied by data1 is : 4



CHECK YOUR PROGRESS

Q 5: A structure is a type of array. True/False

Q 6: Unions are declared using the _____ keyword.

Q 7: The members of union share same/different memory locations.



5.7 LET US SUM UP

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

- An array in C++ or in any programming language is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using indices of an array.
- Instead of declaring individual variables, we can declare one array to store multiple values.
- An array element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array.

- C++ allows multidimensional arrays.
- A two-dimensional array is a list of one-dimensional arrays.
- A structure creates a data type that can be used to group items of possibly different types into a single type.
- Array of structures can also be declared in C++. An array of structures is the collection of similar type of structures.
- A union is also a user-defined datatype. It is similar to structure but all the members of a union share same memory locations. Size of a union is decided by the size of largest member of that union.



5.8 FURTHER READING

1. Schildt, H. (2017). *Turbo C/C++: the complete reference*. McGraw-Hill Education
2. Balagurusamy, E. (2020). *Object-Oriented Programming with C++, 7e*. McGraw-Hill Education.
3. Yashavant, K. (2020). *Let us C*. BPB Publications.



5.9 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q.No.1: similar

Ans to Q.No.2: contiguous

Ans to Q.No.3: True

Ans to Q.No.4: False

Ans to Q.No.5: False

Ans to Q.No.6: union

Ans to Q.No.7: same



5.10 MODEL QUESTIONS

Q1. What is an array? How is an array declared and initialized in C++?

Q2. Why array is required in C++ program?

Q3. How are array elements accessed in C++? Explain with an example.

Q4. What are multi-dimensional arrays? How is a two-dimensional array declared in C++?

Q5. What is a Structure? Write a C++ program to demonstrate the use of structures.

Q6. What is a Union? Write a C++ program to demonstrate the use of unions.

Q7. Write down the differences between structure and union in C++.

*** ***** ***

UNIT 6: POINTERS

UNIT STRUCTURE

6.1 Learning Objectives

6.2 Introduction

6.3 Address Operator

6.4 Pointer Definition and Declaration

6.5 Parameter Passing by Pointer

6.6 Void Pointer

6.7 Pointer Arithmetic

6.8 Runtime Memory Management

6.9 Pointer to Pointer

6.10 Arrayof Pointers

6.11 Pointers to Functions

6.12 Let Us Sum Up

6.13 Further Reading

6.14 Answers to Check Your Progress

6.15 Model Questions

6.1 LEARNING OBJECTIVES

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

- understand the basics of the address operator
- learn about the pointer variable and its various types
- describe parameter passing by pointers
- learn about different pointer arithmetic operations
- get the concept of runtime memory management with new and delete operators
- learn the principles of double pointers
- understand the functioning of array of pointers
- learn how pointers operate on functions when passed as arguments

6.2 INTRODUCTION

So far, various concepts related to C++ programming are discussed. We have learnt the basic concepts of object-oriented programming. We are introduced to the elements of C++ like variables and constants, tokens and identifiers, basic data types, keywords, streams, etc. We also have gained ideas on operators, expressions, various decision control statements as well as on arrays and structures.

In this unit, we will learn about pointers. A pointer is a special variable that points to the address of another variable but the data types of both should be the same. In C++, pointers are used in dynamic memory allocations. They may be useful for representing variables, arrays, objects, etc with the requisite amount of memory allocated to them. Also, with these representations, conditions and decisions can be made. Throughout this chapter, we will learn various operations on pointers, their different types, and also how the pointer variables work on different elements to efficiently utilize the memory of a computer system.

6.3 ADDRESS OPERATOR

The operator through which the memory address of a variable is returned is termed as the address operator. The address operator is denoted by the symbol '&' (ampersand).

The syntax of using address operator can be learnt by considering the following statement.

```
Ptr = &Var ;
```

In the above statement, 'Ptr' is a pointer variable and 'Var' is a simple variable. So, the expression,'&Var', refers to the address of variable'Var'. The address is returned in terms of numerical value and it is assigned to 'Ptr'.

6.4 POINTER DEFINITION AND DECLARATION

We already have acquired a detailed discussion on pointer variables in C-programming language. In this unit, we will also try to gather some ideas on pointers; but in terms of C++ programming language.

Pointer can be defined as a variable that can hold the address of another variable of same data type. Each pointer returns a long hexadecimal number which represents a memory address.

The general form of declaration of pointer variables for any data type is presented as follows.

```
Data_type *variableName;
```

Where 'Data_type' represents a valid C++ data type and 'variableName' is the name of the pointer variable. The asterisk symbol (*) also known as the indirection operator must precede the variable name. The indirection operator is also known dereferencing operator. Dereferencing a pointer indicates that the value of a variable pointed by the pointer is accessed or manipulated. Some examples of pointer declarations are presented as follows.

```
int *var;
char *ch;
float *varF;
double *varD;
```

Here, var, ch, varF, varD are the pointer variables of data types integer, character, float, and double respectively. Let us consider the following statement.

```
int *var, x;
```

'var' is a pointer variable of type integer and 'x' another variable of the same type. We will now make 'var', to point the variable,'x'. The statement to serve this purpose is:

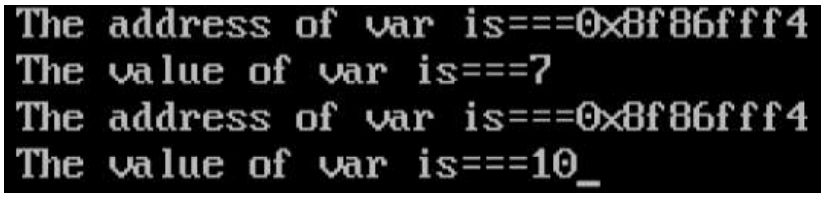
```
var =&x;
```

The address operator,& gives address of variable 'x'.

Program 6.1: C++ program to show the relationship between a pointer variable and address operator.

```
#include<iostream.h>
int main()
{
      int *ptr, var=7;
      clrscr();
      ptr=&var;
      *ptr=var;
      cout<<"\nThe address of var is==="<<ptr<<endl;
      cout<<"\nThe value of var is==="<<*ptr<<endl;
      *ptr=10;
      cout<<"\nThe address of var is==="<<ptr<<endl;
      cout<<"\nThe value of var is==="<<var
      return 0;
}
```

The output of the program is:



The program has an integer pointer ‘ptr’ and an integer variable ‘var’. ‘var’ is initialized to the value, 7.The address of ‘var’ is assigned to the pointer, ‘ptr’. ‘*ptr’ is interpreted as the value of ptr. So, ‘*ptr’ will refer the value stored in the variable, ‘var’,that is pointed by ‘ptr’.

When a pointer variable is declared, it is always good to assign a NULL value to it. Pointer of such kind is known as NULL pointer and contains value 0. The C++ statement for this declaration is presented as follows.

int *var=NULL;

This declaration interprets that the pointer ‘var’ points to nothing.



CHECK YOUR PROGRESS

Q 1: Fill in the blanks:

i. The address operator returns _____ of a variable.

ii. The address operator is denoted by the symbol _____.

iii. The address of variable, 'a' is defined by the expression _____.

iv. The indirection operator is denoted by _____.

v. The indirection operator is also known _____.

vi. A pointer is a variable which points to the memory _____ of a variable.

vii. A _____ pointer is assigned with value 0.

Q 2: State true/false

i. The statement *ptr=var assigns address of var to pointer ptr.

ii. Each pointer returns a memory address in terms of a binary number.

6.5 PARAMETER PASSING BY POINTER

Like C, C++ also has provided the facility of passing parameters or arguments to its functions. Parameters are passed into a function definition to perform some tasks. Accordingly, some sort of modifications may also take place into the values of the parameters. Parameters are of two kinds – **formal parameters** and **actual parameters**. Parameters that are used in the definition of a function are called formal parameters. Parameters that are used when the function is called are known as actual parameters. In other words, a caller function uses actual parameters. These actual parameters get replaced by the formal parameters in the function definition.

There are two methods of passing parameters to a function which are– (i) pass by value and (ii) pass by address.

As the name implies, pass by value method requires arguments to be passed only in terms of their values. However, one major drawback of this method is that once the formal arguments get modified, those changes never get reflected outside the called function. Therefore, the scope of the arguments falls only inside the function. Therefore, modifications in formal parameters are not reflected in actual parameters.

Program 6.2: C++ program to show parameter passing by value.

```
#include<iostream.h>
void swap(int x, int y)
{
      int t;
      t=x;
      x=y;
      y=t;
}
int main()
{
      int a,b,t;
      cout<<"\nEnter a:";
      cin>>a;
      cout<<"\nEnter b:";
      cin>>b;
      swap(a,b);    // Parameter passing by value
      cout<<"\nAfter swapping, a and b becomes:"<<endl;
      cout<<"\na:"<<a;
      cout<<"\nb:"<<b;
      return 0;
}
```

The program output is:

Enter a: 8

Enter b: 4

After swapping, a and b becomes:

a: 8

b: 4

In the above program, the actual arguments are a and b. In main(), a and b are passed by value as parameters to the caller function swap().In the definition of swap(), the value of a and b are copied to the formal parameters x and y respectively. The function definition uses a third variable t to interchange thevalues of the formal parameters. However, changes are limited to only within the scope of the function. Once the control returns to the main() function, the interchanged values are not visible. Therefore, it can be concluded that changes are not permanent if parameters are passed by value into a function.

The other method of passing parameters to a function is pass by address. In this method, addresses of the actual parameters are passed to a function and pointers are used as formal parameters to hold these addresses. In this mode of passing parameters, changes made to the formal parameters are also reflected in the actual parameters. So, the scope of such changes is throughout the whole program.

Program 6.3: C++ program to show parameter passing by address.

```
#include<iostream.h>
void swap(int * x, int *y)
{
      int t;
      t=*x;
      *x=*y;
      *y=t;
}
int main()
{
      int a,b,t;
      cout<<"\nEnter a:";
```

```
    cin>>a;
    cout<<"\nEnter b:";
    cin>>b;
    swap(&a,&b);    //Parameter passing by address
    cout<<"\nAfter swapping, a and b becomes:"<<endl;
    cout<<"\na:"<<a;
    cout<<"\nb:"<<b;
    return 0;
}
```

The output is :

Enter a: 8

Enter b: 4

After swapping, a and b becomes:

a: 4

b: 8

The execution of the above program results in changes in the values of actual parameters a and b. We are passing the addresses in the caller function. The called function has its formal parameters in terms of pointers. So the actual values referred by the addresses are accessed in the function and finally, we get interchanged values in the actual parameters.

6.6 VOID POINTER

As the name implies, a pointer that does not have any associated data type is called void pointer. It can hold addresses of any type and also can be type casted to any type. Since a void pointer do not have any type, we cannot directly dereference it as the compiler is not sure about the type of the variable pointed by it. We need it to be type casted explicitly and then dereference it.

Example 6.4: C++ program to show the use of void pointer.

```
#include<iostream.h>
int main()
{
```

```
        int x=7;
        char y='a';
        void *ptr=&x;
        cout<<"The address of x is=="<<ptr<<endl;
        ptr=&y;
        cout<<"The address of y is=="<<ptr<<endl;
        return 0;
}
```

The output of the program is:

```
The address of x is==0x8f81fff4
The address of y is==0x8f81fff3
```



CHECK YOUR PROGRESS

Q 3: State whether true or false:

i. Passing by value method requires arguments to be passed using dereferencing operator.

ii. Formal parameters are passed in terms of pointers in pass by address method.

iii. Void pointers can be type casted to any type.

6.7 POINTER ARITHMETIC

As already mentioned, a pointer holds an address in terms of numeric value. Therefore, it can be implied that arithmetic operations can be performed on pointers. The four basic arithmetic operations performed on pointers are:

1. Unary increment operation on pointer
2. Unary decrement operation on pointer
3. The addition between pointer and integer
4. Subtraction between pointer and integer

To understand the arithmetic operations on pointers, let us consider an integer pointer variable,ptr pointing to the address 1016.

Now, let us perform the increment operation ptr++. Assuming the size of integer variables as 32 bits, the next integer will be accessed at a location that is 4 bytes away from the current one. Therefore, the return address of ptr would be 1020. Now if ptr is a character pointer and it holds the address value 1016, the same increment operation will produce 1017 as the next address.

Program 6.5: C++ program to show the increment operation on pointer.

```
#include<iostream.h>
int main()
{
      int *ptr, x=30;
      char *ptr1, c='A';
      ptr=&x;
      cout<<"The value of ptr would be=="<<ptr<<endl;
      ptr++;
      cout<<"The value of ptr would be=="<<ptr<<endl;
      ptr1=&c;
      cout<<"The value of ptr1 would be=="<<ptr1<<endl;
      ptr1++;
      cout<<"The value of ptr1 would be=="<<ptr1;
      return 0;
}
```

The program output will be :

```
The value of ptr would be==0x8f85fff4
The value of ptr would be==0x8f85fff6
The value of ptr1 would be==A▲
The value of ptr1 would be==▲_
```

A very unusual behavior can be seen here while printing the addresses of a character pointer variable. Unlike the integer pointer which shows the address possessed by an integer pointer, a character pointer does not display the address. Rather, it prints the value of the character variable followed by some random characters.

The character pointer shows different behavior from that of other variables. They are mostly used to refer to the string literals as these literals are basically composed of an array of characters. Now, our example program contains the character variable c holds value 'A'. Once we attempt to print the address, it only prints some random character after 'A'. This is because astring is always expected to be terminated by a '\0' character. Therefore, random characters will be printed until the NULL character is encountered. So, in order to print the actual address referred to by the pointer, we have to type cast it to void *. The following example tries to print the address held by the character pointer.

Program 6.6: C++ program to display the address of a character variable held by a character pointer.

```
#include<iostream.h>
int main()
{
      char *ptr1, c = 'A';
      ptr1 = &c;
      cout<<"The value of ptr1 would be=="<<(void *)ptr1<<endl;
       ptr1++;
      cout<<"The value of ptr1 would be=="<<(void *)ptr1<<endl;
      return 0;
}
```

The program output will be :

```
The value of ptr1 would be==0x8f82fff5
The value of ptr1 would be==0x8f82fff6
```

A pointer variable can also be decremented with the same considerations applied to pointers employing increment operation. Now, in this case, the address value is decremented by the number of bytes depending upon the data type.

Program 6.7: C ++ program to show the decrement operation on pointer.

```
#include<iostream.h>
int main()
{
      int *ptr, x = 30;
      char *ptr1, c = 'A';
      ptr = &x;
      cout<<"The value of ptr would be=="<<ptr<<endl;
      ptr–;
      cout<<"The value of ptr would be=="<<ptr<<endl;
      ptr1 = &c;
      cout<<"The value of ptr1 would be=="<<(void *)ptr1<<endl;
      ptr1–;
      cout<<"The value of ptr1 would be=="<<(void *)ptr1;
      return 0;
}
```

The output :

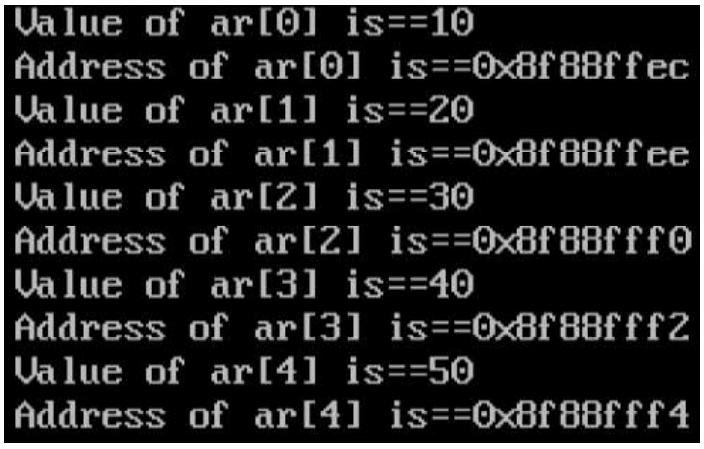
```
The value of ptr would be==0x8f86fff4
The value of ptr would be==0x8f86fff2
The value of ptr1 would be==0x8f86fff3
The value of ptr1 would be==0x8f86fff2
```

The best waypointer arithmetic can be demonstrated is with the help of an array. Initially, a pointer variable can be used torefer the address of the first element of an array. Then by incrementing the pointer, the subsequent elements of the array can be referred by it. Let us see the following program which shows how pointer can be used to manipulate the array elements.

Program 6.8: C++ program to show how pointer can be used to access array elements using increment operation.

```
#include<iostream.h>
int main()
{
      int ar[5]={10, 20, 30, 40, 50};
      int *ptr;
      ptr=ar;
      for(int  i=0;i<5;i++)
      {
      cout<<"Value of ar["<<i<<"] is=="<<*ptr<<endl;
      cout<<"Address of ar["<<i<<"] is=="<<ptr<<endl;
      ptr++;
      }
      return 0;
}
```

The program output is:



Similarly, the decrement operation can also be performed on pointers to access array elements.In case of arrays, depending on the number of bytes required by a pointer variable, the address value is decremented and the pointer is made to move in the backward direction. The following program will demonstrate it.

Program 6.9: C++ program to show how pointer can be used to access array elements using decrement operation.

```
#include<iostream.h>
const int SIZE=3 ;
int main()
{
      int ar[SIZE]={100, 200, 300};
      int *ptr, i;
      ptr=&ar[SIZE-1];
      for(i= SIZE-1; i>=0; i—)
      {
      cout<<"Value of ar["<<i<<"] is=="<<*ptr<<endl;
      cout<<"Address of ar["<<i<<"] is=="<<ptr<<endl;
      ptr—;
      }
       return 0;
}
```

The program produces the following output:

```
Value of ar[2] is==300
Address of ar[2] is==0x8f88fff4
Value of ar[1] is==200
Address of ar[1] is==0x8f88fff2
Value of ar[0] is==100
Address of ar[0] is==0x8f88fff0
```

It is required to mention at this point that the result of addition or subtraction between a pointer and an integer is always a pointer.

We can use the following statement to point the i^{th} element of an array ar[ ] by using the pointer, ptr.

ptr = &ar[i];

Now, if we perform the following operation,

ptr = ptr+4;

The above operation means that ptr will be moved forward by 4 positions. If we know the current position of the pointer ptr, we can add 4 to this so as to reach the new position. The current position would be given by the value of i. So, the expression, 'ptr + 4' can be interpreted as '&ar[i + 4]'. Similarly, the expression,'ptr + j' is equivalent to the expression, '&ar[i + j]'.

Now let us consider the following statement.

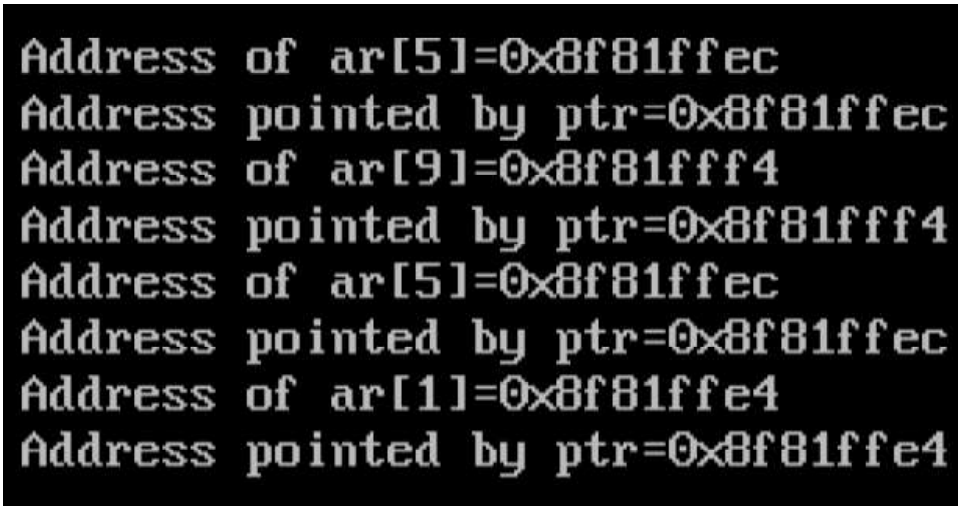
ptr = ptr – 4;

The above operation means that ptr will be moved backward by 4 positions. So, the expression, 'ptr – 4' can be interpreted as '&ar[i - 4]'.

Program 6.10: C++ program to show addition and subtraction operation on pointers.

```
#include<iostream.h>
const int SIZE = 10 ;
int main()
{
        int ar[SIZE] = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};
        int *ptr;
        ptr=&ar[5];
        cout<<"\n Address of ar[5] ="<<&ar[5];
        cout<<"\n Address pointed by ptr  ="<<ptr;
        ptr = ptr + 4;
        cout<<"\n Address of ar[9] ="<<&ar[9];
        cout<<"\n Address pointed by ptr ="<<ptr;
        ptr=&ar[5];
        cout<<"\n Address of ar[5] ="<<&ar[5];
        cout<<"\n Address pointed by ptr ="<<ptr;
        ptr = ptr - 4;
        cout<<"\n Address of ar[1] ="<<&ar[1];
        cout<<"\n Address pointed by ptr ="<<ptr;
         return 0;
}
```

The program produces the following output:



Pointers can also be used to access multidimensional arrays. Let us consider the following two-dimensional array declaration.

int ar[3][3]={{2, 4, 6}, {1, 3, 5}, {7, 8, 9}};

From the above statement, ar can be considered as a pointer which point to the base address of the two-dimensional array, ar[ ][ ]. So, ar is equivalent to &ar[0]. Now,*(*ar) can be used to access the first element of the array. Similarly, ar[i][j] is equivalent to the expression, *(*(ar + i) + j).In case of the two-dimensional array, ar, the correspondence among pointer annotation, array annotation, and array value is represented by the following table.

Pointer Annotation	**Array Annotation**	**Array Value**
*(*ar)	ar[0][0]	2
*(*ar + 1)	ar[0][1]	4
*(*ar + 2)	ar[0][2]	6
((ar + 1))	ar[1][0]	1
((ar + 1)+1)	ar[1][1]	3
((ar + 1)+2)	ar[1][2]	5
((ar + 2))	ar[2][0]	7
((ar + 2)+1)	ar[2][1]	8
((ar + 2)+2)	ar[2][2]	9

Table 6.1: Relationship between pointer annotation, array annotation, and array value



CHECK YOUR PROGRESS

Q 4: Fill in the blanks:

i. The result of addition or subtraction between a pointer and an integer is always a _____.

ii. While referring to an array, a pointer variable actually holds the address of the _____ element of the array.

iii. Any array element ar[i] [j] is equivalent to the representation _____.

Q 5: Can pointers operate on multidimensional arrays.?

6.8 RUNTIME MEMORY MANAGEMENT

C++ allows a facility to allocate memory to variables, arrays, or objects during runtime. This is known as runtime memory management. This provides an efficient memory management facility that not only allocates but also deallocates memory dynamically. There are two operators used to allocate and de-allocate memoryin C++. These are'new' and 'delete'.

The operator, 'new' allocates memory to a variable dynamically. The variable may be of any type or it may be an array or an object also. We can allocate that much memory to a variable as required by the variable's data type. The essence of new operator can be felt very well in the declaration of arrays. If we need to store five elements in an array then it is unnecessary to statically allot storage for ten or twenty elements. This extra space could be allotted to store other variables. Static allocation wastes space as the array size is predefined and the number of array elements may be smaller than the actual array size. On the other hand, in a particular situation, if the array needs to store 15 elements in an array of size 10, then also a problem arises. We cannot accommodate more elements in an array than the actual array size. Therefore, dynamic allocation is the best practice to reserve space for variables of any type. The general syntax to use 'new' operator is presented as follows.

```
pointer_variable = new data_type;
```

Here, pointer_variable is the name of a pointer variable. The operator,'new' initializes the memory required by the data_type. The data_type may be any basic data type in C++. The operator, 'new' returns the address of the memory location. The following block of C++ code allocates memory to an integer.

Program 6.11: C++ program to allocate memory dynamically.

```
#include<iostream.h>
int main()
{
      int *ptr;
      ptr = new int;  // Memory Allocation
      *ptr = 60;
      cout<<"The value pointed byptr is=="<<*ptr;
      return 0;
}
```

The program output will be :

```
The value pointed by ptr is==60
```

In the above program, the 'new' keyword assigns the number of bytes required by an integer to an integer pointer variable, 'ptr'. Then, integer 60 is assigned to the allocated memory locations pointed by 'ptr'. Similarly, we can assign the required memory to any type of pointer variable.

Program 6.12: C++ program to dynamically allocate memory to character and floating point pointer variables.

```
#include<iostream.h>
int main()
{
      char *varC;
      float *varF;
      varC = new char;
```

```
        varF = new float;
        *varC = 'R';
        *varF= 60.66;
        cout<<"The value pointed byvarC is=="<<*varC<<endl;
        cout<<"The value pointed byvarF is=="<<*varF;
        return 0;
}
```

And the program produces the following output :

```
The value pointed by varC is==R
The value pointed by varF is==60.66
```

Memory allocated dynamically to a variable can also be de-allocated. In C++, the operator, 'delete'is used to de-allocate the memory that is allocated dynamically by using the operator 'new'. If a variable is no longer needed by a program, the memory can be freed using 'delete' operator. The released memory is returned to the operating system. The general syntax for de-allocation of memory using 'delete' operator is presented as follows.

delete pointer_variable ;

Here, pointer_variable is the name of the pointer that points to a particular dynamically allocated memory space.

Program 6.13: C++ program to de-allocate memory using 'delete' operator.

```
#include<iostream.h>
int main()
{
        char *varC;
        float *varF;
        varC = new char;
        varF = new float;
        *varC = 'R';
        *varF= 60.66;
```

```
      cout<<"The value pointed by varC is=="<<*varC<<endl;
      cout<<"The value pointed byvarF is=="<<*varF;
      delete varC;    // Memory De-allocation
      delete varF;
      return 0;
}
```

Memory for an array can also be dynamically allocated by using 'new' operator. In this process, the address of the first array element will be returned by 'new' operator and it is assigned to a pointer variable.The general syntax to allocate memory for array is presented as follows.

Pointer_variable = new Data_type[Size];

Here, Pointer_variable is the pointer that point to the first array element. Data_type is any valid data type in C++ and Size is a positive integer constant that refers the array size.

The general syntax to de-allocate memory of dynamically allocated array is presented as follows.

delete[ ] Pointer_variable;

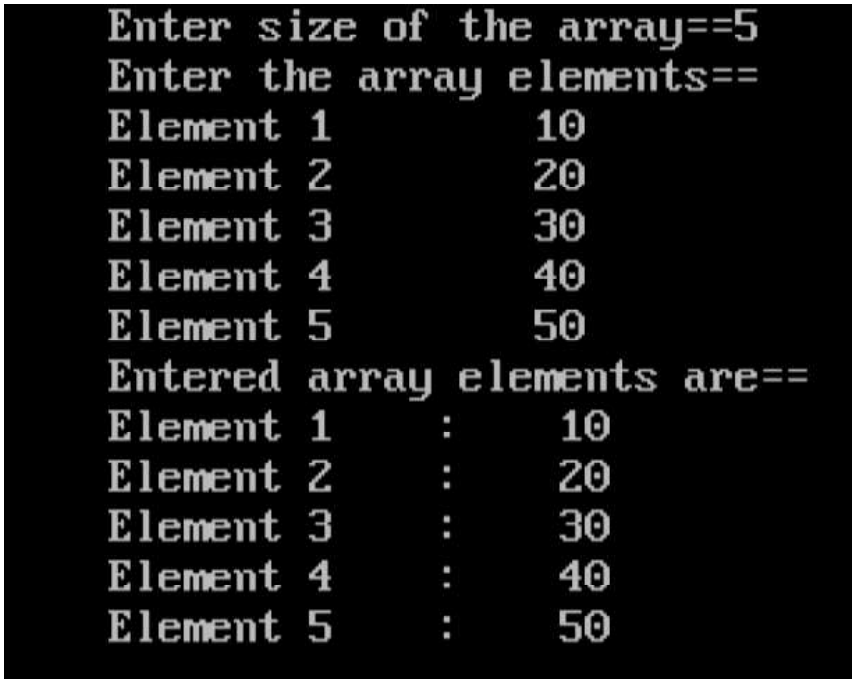
The following C++ program shows the use of'new' and 'delete' operators on array.

Program 6.14: C++ program to show the use of 'new' and 'delete' operators on array.

```
#include<iostream.h>
int main()
{
      int *arr, size;
      cout<<"Enter size of the array==";
      cin>>size;
      arr = new int[size];  // Memory Allocation for Array
      cout<<"Enter the array elements==\n";
      for (int i=0; i<size; i++)
```

```
{
      cout<<"Element "<<i+1<<"         ";
      cin>>*(arr+i);
   }
      cout<<"Entered array elements are==\n";
      for (int i=0; i<size; i++)
   {
      cout<<"Element "<<i+1<<"    :    "<<*(arr+i);
      cout<<endl;
   }
      delete[] arr;  // Memory De-allocation for Array
      return 0;
}
```

The output produced will be:



In the above program, once memory is allocated dynamically, array elements are scanned and printed with the help of pointer notation. The pointer traverses the whole array. Finally, when the array is no longer needed, 'delete' operator is used to de-allocate the memory. The '[]' following 'delete' operator designate that the de-allocation has occurred on an array.

Dynamic memory allocation can also be performed in case of objects. The general syntax to allocate memory dynamically for an object is presented as follows.

Class_Name *Obj_Ptr = new Class_Name( );

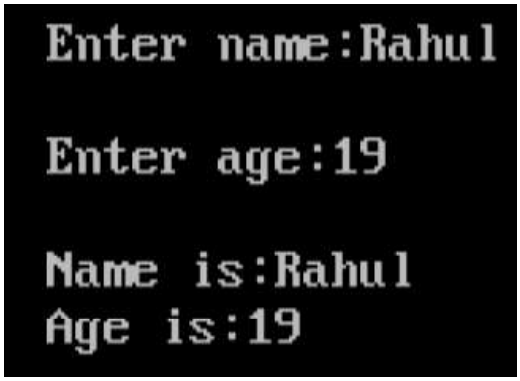
Here, 'Class_Name' is a name of a class and 'Obj_Ptr' is a pointer which point to the memory location allocated dynamically for an object of the class 'Class_Name' by using operator 'new'.

Program 6.15: C++ program to allocate memory dynamically for object of a class.

```
#include<iostream.h>
#define SIZE 20
class student
{
  private:
    char name[SIZE];
    int age;
public:
    void getdata()
    {
cout<<"\nEnter name:";
cin>>name;
cout<<"\nEnter age:";
cin>>age;
    }
    void display()
    {
cout<<"\nName is:"<<name;
cout<<"\nAge is:"<<age;
    }
};
int main()
{
```

```
    student *st = new student();    //Memory allocation
cout<<"\n\n";
st->getdata();
st->display();
    delete st;  // Memory De-allocation
    return 0;
}
```

The program output is :



In the above program, a class, 'student' has been declared with two private data members,'name' and 'age'. The public function getdata() enables access to these members. In the main() function, a pointer (st) to object of class, 'student' is declared and it is assigned with the memory address of the object that is created by allocating memory dynamically using operator, 'new'. The arrow operator (->) is used to access class members when pointers are used. Finally, memory de-allocation is performed using operator, 'delete'.



CHECK YOUR PROGRESS

Q 6: What is runtime memory management?

Q 7: What are the two operators to allocate and deallocate memory?

Q 8: Fill in the blanks:

i. If a variable is no longer required by a program, the memory can be freed using the _____operator.

ii. Memory is initialized using the _____operator.

6.9 POINTER TO POINTER

We have already learnt that pointers contain the address of a variable. A pointer to pointer is a concept where one pointer refers to the address of another pointer, which in turn points to a location where the actual value is stored. It means that the first pointer holds the address of the second pointer, which in turn holds the memory address of a variable where the actual value is stored.

A pointer to pointer variable is also referred as double pointer. The general syntax for declaring a double pointer is presented as follows.

Data_type **pointer_variable;

Here, 'Data_type' may be any valid data type and pointer_variable is the name of the pointer.

Program 6.16: C++ program to demonstrate the declaration and use of pointer to pointer.

```
#include<iostream.h>
int main()
{
      int var=50;
      int *ptr;
      int **ptrP;
      ptr=&var;
      ptrP=&ptr;
      cout<<"\n\n";
      cout<<"\nThe value of var is=="<<var;
      cout<<"\nThe address of var is=="<<&var;
      cout<<"\nThe address stored in ptr is=="<<ptr;
      cout<<"\nThe value of *ptr is=="<<*ptr;
      cout<<"\nThe address of ptr is=="<<&ptr;
      cout<<"\nThe address stored in ptrP is=="<<ptrP;
      cout<<"\nThe value of **ptrP is=="<<**ptrP;
      cout<<"\n";
      return 0;
}
```

The program produces the following output :

```
The value of var is==50
The address of var is==0x8f80fff4
The address stored in ptr is==0x8f80fff4
The value of *ptr is==50
The address of ptr is==0x8f80fff2
The address stored in ptrP is==0x8f80fff2
The value of **ptrP is==50
```

Let us try to understand the above program with the help of a diagram. It shows the memory addresses and contents of the three variables ‘var’, ‘ptr’ and ‘ptrP’. Here,‘ptrP’ is the double pointer which holds the address of ‘ptr’ and ‘ptr’ is a pointer which holds the address of the variable,‘var’. Value stored in ‘var’ is 50. So, when we try to print the values of var, *ptr and **ptrP then 50 is displayed in the standard output device.

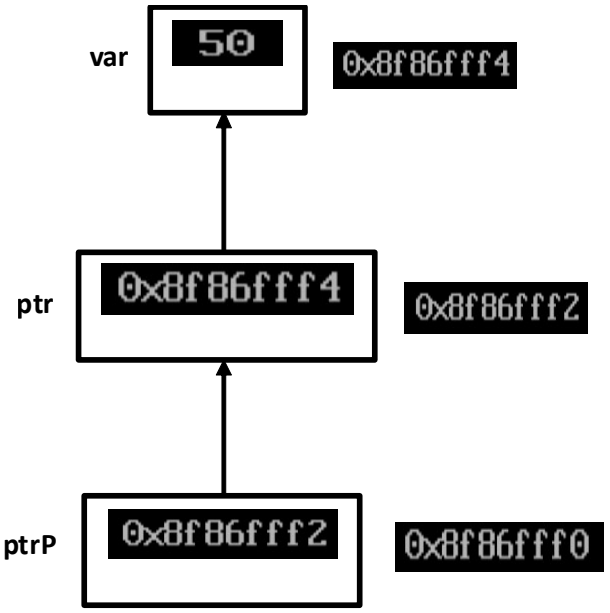


Figure 6.1: Diagrammatic representation of a pointer to pointer

6.10 ARRAY OF POINTERS

There is a close association between an array and a pointer. C++ allows the array name to be represented as a pointer. The name itself can refer to the address of an element. It can hold the address of the first element of an

array. Suppose, we have an integer array ar[] having size equal to 10 and an integer pointer *ptr. Now, if we write the assignment statement as follows.

ptr =ar;

This will make the pointer pointing to the array, ar[ ]. Pointer, 'ptr' will contain the address of the first element, i.e. ar[0]. Finally, if we print the values of *ptr and *ar, same value will be displayed, i.e. the value of ar[0].

An array of pointers is an array that contains pointer variables of same type. The general syntax for declaring an array of pointers is presented as follows.

data_type *array_name[size] ;

Here, 'data_type' is any valid data type in C++ and 'array_name' is the name of array of pointers. 'size' is the array size. For example, let us consider the following statement.

int *ptr[10] ;

The above declaration declares an array of 10 integer pointers.

However, pointer to an array and array of pointers are two entirely different concepts. As we have already discussed, a pointer to an array holds the address of the first element in the array. On the other hand, an array of pointers holds addresses of different variables of same type or pointers to different arrays of same type. The following program demonstrates how an array of pointers can hold pointers to different arrays.

Program 6.17: C++ program to show the use of array of pointers.

```
#include<iostream.h>
const int SIZE = 5;
int main()
{
       int ar1[SIZE] = {1,2,3,4,5};
       int ar2[SIZE] = {10,20,30,40,50};
       int ar3[SIZE] = {100,200,300,400,500};
       int ar4[SIZE] = {11,22,33,44,55};
```

```
int ar5[SIZE] = {111,222,333,444,555};
int* ptr[SIZE],i;
ptr[0] = ar1;
ptr[1] = ar2;
ptr[2] = ar3;
ptr[3] = ar4;
ptr[4] = ar5;
cout<<"\n Elements in array ar1[] are:\n";
for(i = 0; i<SIZE ; i++)
{
cout<<"\t";
cout<<*(ptr[0]+i); //Pointer in ptr[0] points to ar1[]
}
cout<<"\n Elements in array ar2[] are:\n";
for(i = 0; i<SIZE ; i++)
{
cout<<"\t";
cout<<*(ptr[1]+i); //Pointer in ptr[1] points to ar2[]
 }
cout<<"\n Elements in array ar3[] are:\n";
 for(i = 0; i<SIZE ; i++)
 {
cout<<"\t";
cout<<*(ptr[2]+i); //Pointer in ptr[2] points to ar3[]
 }
cout<<"\n Elements in array ar4[] are:\n";
 for(i = 0; i<SIZE ; i++)
 {
cout<<"\t";
cout<<*(ptr[3]+i); //Pointer in ptr[3] points to ar4[]
 }
cout<<"\n Elements in array ar5[] are:\n";
 for(i = 0; i<SIZE ; i++)
```

```
        {
       cout<<"\t";
       cout<<*(ptr[4]+i); //Pointer in ptr[4] points to ar5[]
        }
        return 0;
}
```

The program produces the following output :

```
Elements in array ar1[] are:
        1       2       3       4       5
Elements in array ar2[] are:
        10      20      30      40      50
Elements in array ar3[] are:
        100     200     300     400     500
Elements in array ar4[] are:
        11      22      33      44      55
Elements in array ar5[] are:
        111     222     333     444     555
```

Program 6.18: C++ program to use array of pointers for the representation ofstrings.

```
#include<iostream.h>
int main()
{
       char*name[5]= {"Anil","Mita","Akhil","Nitu","Tina"};
       for(int i=0 ; i<5 ; i++)
       {
       cout<<"\n";
       cout<<i+1<<"th string pointed by name["<<i<<"] is:";
       cout<< *(name + i)<<endl;
       }
       return 0;
}
```

The output is :

```
1th string pointed by name[0] is:Anil

2th string pointed by name[1] is:Mita

3th string pointed by name[2] is:Akhil

4th string pointed by name[3] is:Nitu

5th string pointed by name[4] is:Tina
```



EXERCISE

1. Write a C++ program to perform the selection sort algorithm using pointers.
2. Write a C++ program to perform the bubble sort algorithm using pointers.
3. Write a C++ program to sort the strings from an array of strings using pointers.



CHECK YOUR PROGRESS

Q 9: Fill in the blanks:

i. When a pointer indirectly refers to another pointer, it is known as _____.

ii. The statement pointer to pointer variable is declared using _____ symbol.

iii. If ptr is a pointer and ar[] is an array, the statement which makes the pointer point to first array element is _____.

iv. An array name can also be used as _____.

v. An array of pointers contains variables of _____ type.

6.11 POINTERS TO FUNCTIONS

C++ allows using pointers to call functions. Also, functions may be used to pass as arguments to other functions. The general syntax for declaring a function pointer is presented as follows.

Return_type *(function_pointer)(parameter1, parameter2,…);

In the above declaration, 'Return_type' is return type of the function which will be pointed by the pointer. Then,'function_pointer' is the name of the pointer. Finally, data types of the parameters are provided within parenthesis as required. Let us consider the following simple C++ program to understand this concept

Program 6.19:

```
#include<iostream.h>
int sum(int n)
{
      int res=0;
      for(int i=1; i<=n; i++)
      res=res+i;
      return res;
}
      int main()
      {
      int (*ptr)(int); // Declaration of pointer to function
      ptr = sum;    // pointer ptr points to function sum()
      int result = ptr (10);
      cout<<"The value after adding numbers is=="<<result;
         return 0;
      }
```

The output of the program is :

```
The value after adding numbers is==55
```

The above program declares function pointer ptr which points to the address of function 'sum()'. Now, we can call 'sum()' using ptr. In other words, the statement 'ptr(10);' calls the function 'sum(10)'.

Program 6.20: C++ program to use a function pointer for pointing a function.

```
#include<iostream.h>
void display(char * name, int age)
{
      cout<<"The entered name is="<<name<<endl;
      cout<<"The entered age is="<<age;
}
      int main()
      {
        char name[30];
        int age;
       void (*ptr)(char*, int); // Function pointer declaration
      ptr = display;
      cout<<"\nEnter the name:";
      cin>>name;
      cout<<"\nEnter age:";
      cin>>age;
      ptr(name, age);
      getch();
        return 0;
      }
```

The program output is :

```
Enter the name:Ajay

Enter age:17
The entered name is=Ajay
The entered age is=17
```

In the above program, the function,'display()' has two parameters- one is character pointer and another is an integer. The address of display() is stored in the function pointer,'ptr'. Statement,'ptr(name, age);' will call function,'display()'.

Some interesting facts need special mention here that normal pointers point to the data. But, function pointers never point to data, rather they point to the code. They hold the starting address of the executable code.

A simple assignment statement like 'ptr = display;' allows to assign address of the function, 'display()'to the function pointer, 'ptr'. It is always not mandatory to write an '&' symbol for getting the address of a function.

Function pointers do not facilitate allocation or de-allocation of memory, unlike normal pointers.

A function pointer can also be passed as argument or they can be returned from a function.Let us consider the following example to understand how function pointer can be passed as argument to a function.

Program 6.21: C++ program to show passing of a function pointer as argument to a function.

```
#include<iostream.h>
int add_three(int a, int b, int c)
{
        return a+b+c;
}
int fun(int (*fp)(int,int,int))//Function pointer as formal parameter
{
     int temp,a,b,c;
     cout<<"\n Enter a=";
     cin>>a;
     cout<<"\n Enter b=";
     cin>>b;
     cout<<"\n Enter c=";
     cin>>c;
```

```
        temp = fp(a,b,c);
        return temp;
}
int main()
{
      int result;
      cout<<"\n\n";
      result=fun(add_three); // Function pointer as parameter
      cout<<"\n The summation of three numbers is=="<<result;
      return 0;
}
```

The output of the program is as follows :

```
Enter a=12

Enter b=3

Enter c=1

The summation of three numbers is==16
```

In the above program, address of the function, 'add_three()' is passed as parameter to the function, 'fun()'. In the function,'fun()',function pointer, 'fp' is used as formal parameter to hold the address of the function, 'add_three()'.



CHECK YOUR PROGRESS

Q 10: State whether true or false.

i. C++ allows using pointers to call functions.

ii. Functions may not be used to pass as arguments to other functions.

iii. Function pointers hold the starting address of the executable code.



6.12 LET US SUM UP

- The address operator '&' returns the memory address of a variable. It is returned in terms of numerical value.
- Pointer variables are always declared in terms of the dereferencing/ indirection operator '*'. It is declared with the data type followed by the '*' with a variable name. The general syntax to declare a pointer variable is:

  Data_type *Pointer_variableName ;
- As the pointer variable holds the address of another variable, the data types of both the variables must be the same. The syntax to assign address of a variable to a pointer variable is:

  Pointer_variableName = &Variable_name ;

  The statement,'*Pointer_variableName'will return the value stored in the variable, 'Variable_name'.
- There are two ways of passing parameters to functions that are'by value' and 'by address'. Pass by value requires the parameters to be passed only with their values. This method does not reflect the changes in actual parameters once the formal parameters get modified. In case of pass by address, if formal parameters are modified then the changes are also reflected in actual parameters.
- A void pointer can hold the address of a variable of any data type.
- The four basic arithmetic operations performed on pointers are:
  - Unary increment operation on pointer
  - Unary decrement operation on pointer
  - The addition between pointer and integer
  - Subtraction between pointer and integer
- C++ has a special feature that allows memories to be allocated to any variables during runtime. At the same time, the allocated memory can also be released to free up space. This is known as dynamic memory management which initializes and releases the memory

space during program execution. It is one of the ways to ensure that storage space should be utilized efficiently and requisite space should be allocated. The two operators 'new' and 'delete' perform these tasks respectively.

- Pointers to pointer or double pointer point to the address of a pointer variable which in turn points to the location of another variable.
- An array of pointers is an array that stores pointers of same type.
- Function pointers allow pointers to call functions. It refers to the address of functions.



6.13 FURTHER READING

1. Venugopal, K. R., & Rajkumar, T. (1999). Ravishankar: *Mastering C++*.
2. Lafore, R. (1997). *Object-Oriented Programming in C++.* United Kingdom: Pearson Education.
3. Kanetkar, Y. P. (2010). *Let us C++*. BPB Publications.



6.14 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1:

i. memory address
ii. '&' (ampersand)
iii. &a
iv. *
v. dereferencing operator
vi. address
vii. NULL

Ans to Q No 2:

i. False ii. False

Ans to Q No 3:

i. False ii. True iii. True

Ans to Q No 4:

i. Pointer ii. First iii. *(*(ar + i) + j)

Ans to Q No 5: Yes, pointers can operate on multidimensional arrays.

Ans to Q No 6: C++ allows a facility to allocate the memory to variables, arrays or objects during runtime. This is known as runtime memory management.

Ans to Q No 7: 'new' and 'delete'.

Ans to Q No 8:

i. delete ii. new

Ans to Q No 9:

i. pointers to pointer
ii. **
iii. ptr = ar;
iv. pointer to the array
v. pointer

Ans to Q No 10:

i. True ii. False iii. True



6.15 MODEL QUESTIONS

Q1. Short answer type questions:

a. What are the address operators?
b. What are pointers? Discuss.
c. How the address operator and pointer variables are related.
d. What is indirection operator? Why do we use it?
e. Write a program to showcase the use of dereferencing operator.
f. Write a program to return the addresses of basic data type variables using pointers.
g. Discuss the NULL pointers in brief.
h. Define void pointer. How do we declare it?
i. Why void pointers are needed to be explicitly type casted?
j. Why are 'new' and 'delete' operators used?

k. How the memory allocation and de-allocation takes place in arrays? Discuss.

l. Explain how dynamic memory allocation takes place on objects of a class.

m. Write a C++ program to demonstrate the use of pointer to pointer.

n. What is array of pointers? Explain.

o. What is pointer to function? Briefly discuss.

Q2. Long answer type questions

a. Explain pointers in details by taking suitable examples.

b. Considering a suitable example, explain void pointer in detail.

c. Explain the four basic arithmetic operations performed on pointers. Also consider examples of each.

d. What are multidimensional arrays? Describe the correspondence among pointer annotation, array annotation and array value using a multidimensional array.

e. What is runtime memory management? How is it performed? Explain with example.

f. What do you understand by pointer to pointer? How do we declare it? Explain in detail with a diagram.

g. What are function pointers? Give a detailed discussion. Also consider examples.

*** ***** ***

UNIT 7: FUNCTIONS

UNIT STRUCTURE

7.1 Learning Objectives

7.2 Introduction

7.3 Introduction to Function

7.4 The main() Function

7.5 Components of a Function

- 7.5.1 Function Prototype or Declaration
- 7.5.2 Function Definition
- 7.5.3 Function Call
- 7.5.4 Function Parameteror Argument

7.6 Passing Arguments to a Function

- 7.6.1 Call by Value
- 7.6.2 Call by Pointer (*)
- 7.6.3 Call by Reference (&)

7.7 Types of Function

- 7.7.1 Built-in Function
- 7.7.2 User- Defined Function

7.8 Inline Function

7.9 Function Overloading

7.10 Let Us Sum Up

7.11 Further Reading

7.12 Answers to Check Your Progress

7.13 Model Questions

7.1 LEARNING OBJECTIVES

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

- learn about the importance of modular programming
- write modular programs using function concept
- learn about C++ main() function
- learn how functions are declared, defined, and accessed

- learn about different components of functions
- learn how arguments are passed to a function
- learn how functions return value
- learn built-in and user-defined functions
- learn about inline function
- explore polymorphism using function overloading

7.2 INTRODUCTION

We have already been acquainted with derived data types such as pointer in the previous unit. Pointer provides a useful mechanism to access the address of a variable or a function.Similar to the pointer, C++ provides another derived data type known as function.It is one of the main ingredients to achieve modular programming by dividing a problem into smaller sub-components.

In this unit, we will see functions in action. Here we will see how a function is declared, defined, and accessed in the C++ programming language. In the following unit, we will explore more into another component of modular programming known as class and object. The concept of class and object provides the main essence of object-oriented programming.

7.3 INTRODUCTION TO FUNCTION

Modular programing is the need of the hour because with increasing problem complexities various difficulties creep into the picture. The study suggests that the human brain has a capping on the number of distinctive thoughts it can process at a time. Researchers at the University of Oregon have concluded that the human brain can process only four things meaningfully at a time. This is a very important discovery particularly with respect to the problem-solving approach adopted by various programmers. We all know that with the increase in problem complexity, program complexity also increases simultaneously. The number of variables and other programming constructs required to implement an algorithm increases accordingly. In such a situation, it becomes difficult for a programmer to

write and manage a program efficiently because of the cognitive limitations explained earlier. Commonly this problem is addressed using the principle of abstraction and decomposition. Abstraction refers to the construction of a simpler version of a problem to be solved by ignoring the underlying details. Here, we only try to focus our attention on the specific aspects of the problem and ignore the rest. Similarly, the decomposition principle, also known as the divide and conquer principle advocates breaking of a problem into many smaller independent components. This way complexity of smaller components becomes easily manageable.It also facilitates modular programming by formulating and integrating solutions to each sub-problems. The concept of abstraction and decomposition can be understood using Figure 7.1.

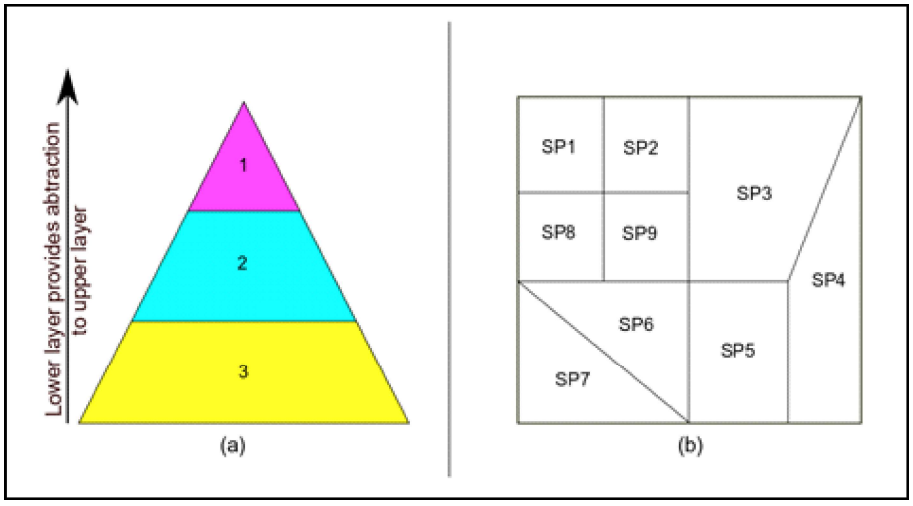


Figure 7.1. Conceptual diagram of (a) Abstraction, and (b) Decomposition

Figure 7.1 (a) represents the layered structure enabling lower layers to provide abstraction to the upper layer. Layer 1 can work independently without worrying about the working of Layer 3. Layer 3's complexity is been shielded by Layer 2.Hence, provides an abstract view to layer 1. This is called abstraction. Similarly, Figure 1(b) shows a typical case of decomposition whereby, a bigger problem is divided into smaller sub-problems by carefully following the divide and conquer principle. However, there are some important points to consider whenever we go for

decomposition. First, it is not always possible to break a problem into meaningful subproblems as shown in Figure 1(b). Second, there are no clear guidelines about up to what extent the decomposition should be carried out. A commonly adopted approach suggests the decomposition process needs to be carried out till the subproblems become meaningful, understandable, and easily manageable to the programmer concerned. Dividing beyond that point might not help rather may increase the overall cost.

In the programming domain, modularity is achieved through the use of functions. A program can be divided into smaller meaningful blocks known as functions to maximize modularity, code reuse, and reduce coding effort. Formally a function can be defined as the *collection of statements that together perform a specific task*. Some of the benefits of using functions are:

- **Simplicity**: Functions make the implementation process easy to comprehend, define and apply. It improves the readability of the code by dividing the code into smaller sub-blocks.
- **Control**: Control flow from one part of the program to the other becomes easier with the incorporation of functions into the program.
- **Reusability**: Code reusability is another aspect. A function defined once can be called any number of times. This essentially saves a programmer from writing the same set of instructions again and again.
- **Debugging**: Error localization and program debugging also becomes easier as the entire program is divided into smaller blocks.
- **Time-saving**: Induction and deletion of new modules become easier with functions. In the case of the modular programming approach, a new module can be developed separately and swiftly integrated with the existing modules. The same is also true with removing an existing module.

7.4 THE main() FUNCTION

The main() function in C++ is a designated global function that acts as a starting point in any program execution. This is one function we have been using even before we knew about functions. Perhaps this is the right time to understand main() in more detail. The function main() can appear in one of the following forms:

```
int main () {          ——                    (1)
        return 0;}
int main (int argc, char *argv [ ]) {    ——    (2)
        return 0;
}
```

Where,

- ***argc*** – Or, **ARGument Count** is a positive value that represents the total number of arguments passed to the program from the host environment.
- ***argv*** – Also known as **ARGument Vector** is an array of pointer variables that point to strings. This array contains the list of arguments required to be passed to the program from the execution environment. In C++, argv[0] always points to the name of the program and the rest of the array elements till argv[argc-1] points to the command line arguments.
- ***return****: The function m*ain() always returns an integer back to the native environment. Usually, *return 0* is used in writing C++ programs. The main() can have one or more *return* statements. Execution of any *return* statement would mean the execution of `std::exit`. This essentially informs the operating system that the program terminated successfully.

In a common scenario, we see the first form of main() function in use. In this case, we do not pass any argument to the main function. All the programs presented till this point used this version of main() only. However, it is important to know that C/C++ allows us to pass external inputs to the

main() as arguments. These are known as the command line arguments (CLA). CLAs are passed to the main() as function arguments. It provides an alternative to hardcoding these values inside the code. Below is a simple program to explain to you the use of command-line arguments.

Program 7.1: Program to use command line argument in C++.

```
#include <iostream.h>

int main(int argc, char* argv[])
{
    int count;    // Counter variable to control the loop

    // Simply printing the name of the executable program
    cout << "Program Name Is: " << argv[0] << endl;

    // Printing argument list based on the no. of args passed
    if(argc==1)
        cout << "No command line argument other than program name passed";
    if(argc>=2)
    {
        cout << "Number Of arguments Passed= " << argc << endl;
        cout << "---- Command line arguments are----" << endl;
        for(count=0; count<argc; count++)
        {
            cout << "argv[" << count << "]: " << argv[count] << endl;

        }
    }
    return 0;
}
```

Output: When the program is executed with no command-line argument is passed (**$ ./a.out**)

```
Program Name Is: ./a.out
No command line argument other than program name passed
```

Output: When the program is executed with "*Hello World*" as the input or command line argument (**$ ./a.out Hello World**)

```
Program Name Is: ./a.out
Number Of arguments Passed= 3
—— Command line arguments are——
argv[0]: ./a.out
argv[1]: Hello
argv[2]: World
```

In the above program, we can see that the program name is printed onto the console by querying argv[0]. argv[0] is the first element of the argument vector and always contains the name of the program in execution. After that, we check to see if the user has input any CLA to the program or not. This is done by checking if(argc>=2).Condition of greater than or equal to 2 is set because argc contains value 1 (i.e., program name) even when no arguments are passed.

Two cases of outputs are presented in the output window. The first case shows execution output when no CLA is passed, whereas, second output window shows the program output when string *Hello World* is input as the CLA. It is important to note that we are required to pass all the CLAs separated by a space. However, if the argument itself has a space in between, then we have to use double quotes (“ “) or single quotes (‘ ‘) to segregate the arguments. Let us observe one more example to add two integers using command line arguments.

Program 7.2:C++ program to add two number using command line arguments

```
#include <iostream.h>
#include<stdlib.h>
int main(int argc,char* argv[])
{
      cout << "Program Name Is: " << argv[0] << endl;
      //Checking if any CLA provided
      if(argc==1)
      cout << "User input as CLA not provided" << endl;
      if(argc>=2)
```

```
    {
        int val1 = atoi(argv[1]);   //conversion of char to int using atoi
        int val2 = atoi(argv[2]);
        int result = val1 + val2;

        cout << "The sum of input values is: " << result << endl;
    }
    return 0;
}
```

Output: When the program is executed using **$ ./a.out 10 20**

Program Name Is: ./a.out

The sum of input values is: 30

In this case, the main() gets 10 and 20 as command-line arguments. It then stores the values in *val1* and *val2* respectively. A customary check is done to see if the user has input the operands and then the final addition is done before printing the result onto the console.

7.5 COMPONENTS OF A FUNCTION

A C++ function has four important parts. Function declaration and definition are two mandatories in using a function whereas, prototyping and parameter usage are not a must but subjective to the demand of the situation. Understanding each of these concepts is important in function implementation.

- Function Prototype
- Function Call
- Function Definition
- Function Parameter

The following subsections explain each of these concepts.

7.5.1 Function Prototype or Declaration

Similar to declaring a variable, we can also declare a function. This is also referred to as function prototype. Syntax of declaring a C++ function is:

```
return_type function_name (args);      //Function prototype
```

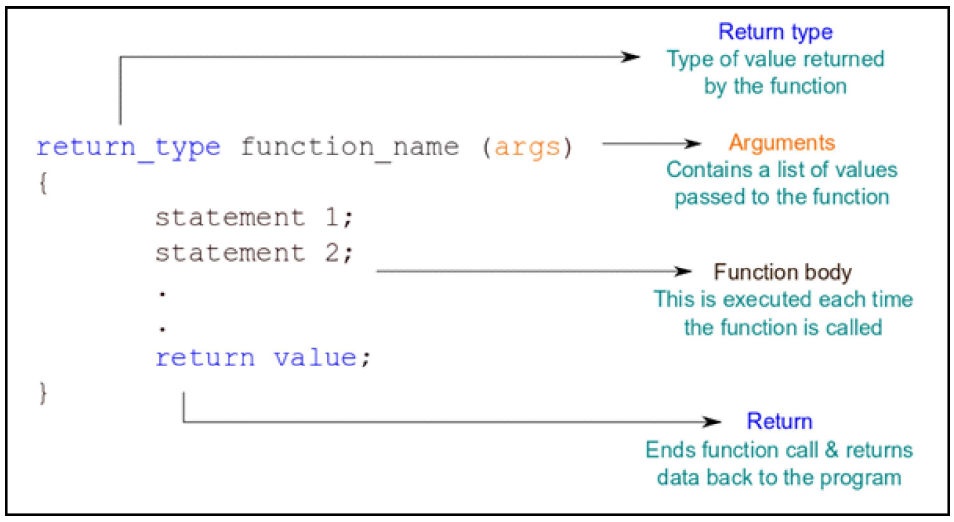
Much like any variable declaration, a C++ function declaration also has a name and a return_type. The *return_type*of a function defines the type of value a function passes back to the caller program. Whereas the identifier *function_name* is the name used in identifying a function distinctly. Additionally, a function can also have a list of zero or more arguments wrapped within a pair of parentheses or round brackets “( )”.This list paves a passage of external values to be input into the function. Hence, we can conceptualize a function declaration as a means of providing the following basic function information to the compiler.

- The name and type of the function and
- The number and type of arguments pass to the function.

Note: It is important to note that the function prototype needs to be placed at the beginning of the program. Usually, it is put just after any global constant declarations but must be ahead of any call made either from *main()* or in any other functions.

7.5.2 Function Definition

Function definition represents that section of the code where a function gets its body defined. Syntax of defining a function is:



The manner in which we define a function is very much similar to the way we declare it. However, in this case, we emphasize ondefining the body of the function. The body contains a set of instructions intended to accomplish a particular task. The entire body of the function is enclosed within a pair of curly braces “{}”.

Program 7.3: Program to add two numbers using function.

```
#include <iostream.h>
int add(int x, int y)   // Function to add two numbers
{
      int temp;
      temp = x + y;
      return temp; // Result of summation is returned
}
int main()
{
      int val1, val2;
      cout << "Enter the first number: ";
      cin >> val1;
      cout << "Enter the second number: ";
      cin >> val2;
      // Display the result of by calling add() function.
      cout << "The sum is: "<< add(val1, val2) << endl;
      return 0;
}
```

Program 7.3 provides a demonstration to add two integers using the function concept. The main() basically accepts inputs from the user using the *cin* statement. User inputs are stored in *val1* and *val2* for its onward transfer to the *add()* function. Finally, the last *cout* statement of *main()* is executed which in turn makes a call to the *add(val1, val2)* with *val1* and *val2* as the arguments. The *add()* calculates the sum and returns the output back to the main function. The output of this program is:

```
Enter the first number: 4
Enter the second number: 6
The sum is: 10
```

At this point, one might be thinking that what is the point of jumping this way when we can replace the *add()* function call with its definition. The body hardly consists of three lines of code. Well, you are correct up to a certain extent. We can do that but think what will happen if we call this *add()* repeatedly. In that case, these three lines of code will have to be replicated repeatedly. This unnecessary replication can be avoided by reusing these three lines of code. Here, *add()* is exactly serving that purpose. Let us have a look at some more programs to fully understand the usage of function concepts in C++.

Program 7.4:C++ Program to reverse a string using function concept

```
#include <iostream.h>
void rev(char []);  // Function prototype
// Here, main()  precedes the rev()
int main()
{
      char str[100];
      cout << "Enter the string: " ;
      cin >> str;
      rev(str);
      return 0;
}
// rev() definition
void rev(char str[])
{
      int i = 0, slen = 0;
      while(str[i]!='\0')
      {
```

```
        i++;
        slen++;
        }
        cout<<"After reversing the string is: ";
        for(i = slen - 1; i >= 0; i—)
        {
        cout << str[i]; //Since, string is an array of characters
        }
}
```

In this case, if you miss out on the inclusion of the function prototype, you will receive an error similar to the one shown below.

main.cpp:10:12: error: 'rev' was not declared in this scope rev(str);

Another important point you have to remember while executing this program is not to input anystring with spaces in between (e.g. "Hello World"). In such cases, only the characters appearing ahead of the first blank space character are considered and ignored the rest of the string. However, there are ways available to get around this problem which is discussed in the next program.

Program 7.5: C++ program to reverse a string containing spaces

```
#include <iostream.h>
#include<stdio.h>
char * rev(char [ ]);  // Function prototype
// Here, main() precedes the rev()
int main()
{
        char str[100];
        int i=0;
        cout << "Enter the string: ";
        while((str[i]=getchar())!='\n')
        i++;
        str[i]='\0';
```

```
        cout<<"After reversing the string is: ";
        cout<< rev(str);
        return 0;
}
// rev() definition
char * rev(char str[ ])
{
        int i=0, j=0, slen=0;
        char rev_str[100];
        while(str[slen]!='\0')
        {
        slen++;
        }
        for(i = slen-1; i >= 0; i—)
        {
        rev_str[j]=str[i]; //Since, string is an
        array of characters
        j++;
        }
        rev_str[j]='\0';
        return rev_str;
}
```

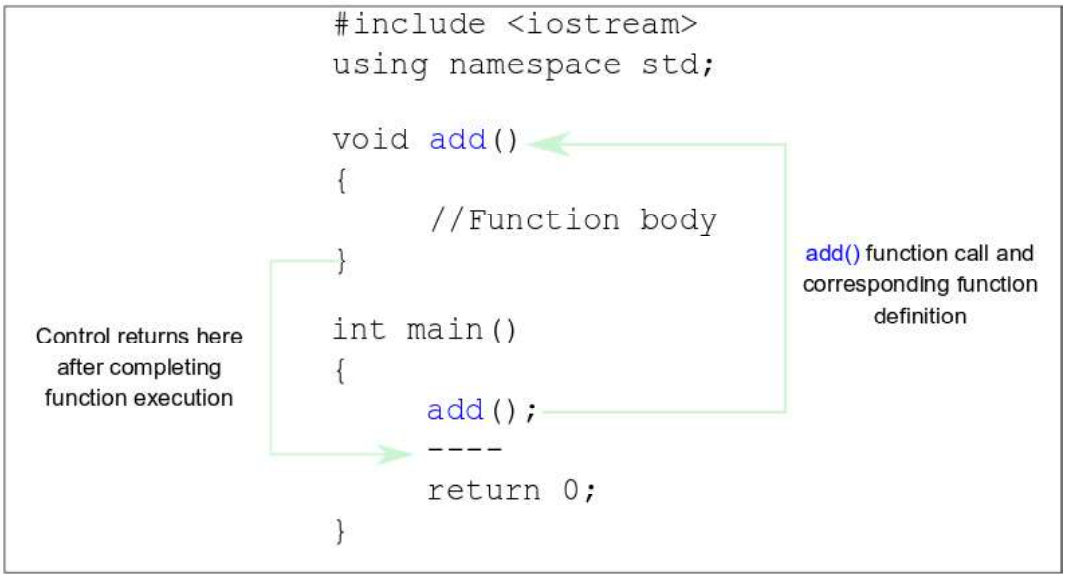
Output:

```
Enter the string: Hello World
After reversing the string is: dlroW olleH
```

In the above program, we can see how the input string gets reversed using the *rev()* function. The *main()* accepts the user input string and passes it to the *rev()*. The function then starts copying character by character into the output string, rev_str. As we can see that the *for loop* begins copying characters from the end of the string *str* and traverses back to the front while copying.

7.5.3 Function Call

We have seen how a function declaration and definition work. Now, to invoke a function we have to call that function. To call a function, all we need to do is use the function name with empty braces if no arguments are passed or use braces with comma-separated arguments. A sample case of a function call and linking it to its definition is shown below.



Here, *main() and add()* are the caller and callee functions respectively. When the ***add()*** function is called (i.e., *add();*) , program control is transferred to that function. Once the execution of the called function gets over, it returns the program control back to the caller program as shown above.

7.5.4 Function Parameter or Argument

A function call may or may not use any arguments. But when it does, the corresponding function definition must declare a set of variables that can accept the values of arguments. This set of variables is known as the formal arguments or formal parameters. Whereas arguments passed through a function call are known as the actual arguments.

In addition, C++ also provided a provision to use default arguments. Default arguments are optional parameters for which no arguments are required in the call. In such cases, if a function definition has one default argument, then it can be called with one less argument in the function call. An implementation of the default argument is presented below.

Program 7.6: C++ Program to demonstrate default argument

```
#include <iostream.h>
int area(float rad, float pi=3.14) //pi is default argument
{
      float circle_area;
      circle_area = pi * rad * rad;
      return circle_area;
}
int main ()
{
      float rad = 0;
      cout << "Enter the radius of the circle: ";
      cin >> rad;    cout << "Area of the circle:
      " << area(rad) << endl;
      return 0;
}
```

Output:

```
Enter the radius of the circle: 10
Area of the circle: 314
```

This is an exception to the fact that the number, type, and sequence of arguments should match between function call and definition.Ideally, they must match but in the presence of a default argument, a function call can provide one less argument.

7.6 PASSING ARGUMENTS TO A FUNCTION

Functions divide a bigger program into smaller sub-components. These sub-components or functionsare expected to communicate and work in tandem to accomplish the overall goal. In a C++ program, this communication happens through message passing. Functions pass messages by passing arguments or parameters from one function to another. There exists three different ways in which we can pass parameters to a function. These are:

1. Call by value or pass by value,
2. Call by pointer or pass by pointer, and
3. Call by reference or pass by reference.

C only supports call by value and call by pointer, whereas C++ supports all.

The nature of parameter transfer can either be unidirectional or bidirectional. Bidirectional communication would mean that the caller function passes parameters to the called function and receives a value in return. Unidirectional communication would mean the other way around and nothing is received in return. Let us have a look at different ways of parameter passing available with C++.

7.6.1 Call by Value

In call by value, a caller function passes the actual arguments and the called function copies those into formal arguments. In simple terms, arguments from the caller function are copied into duplicate variables in the callee function. Therefore, any changes made on the cloned variable do not reflect on the original variable. The concept of call by value can be better understood using the representation shown in Figure 7.2.

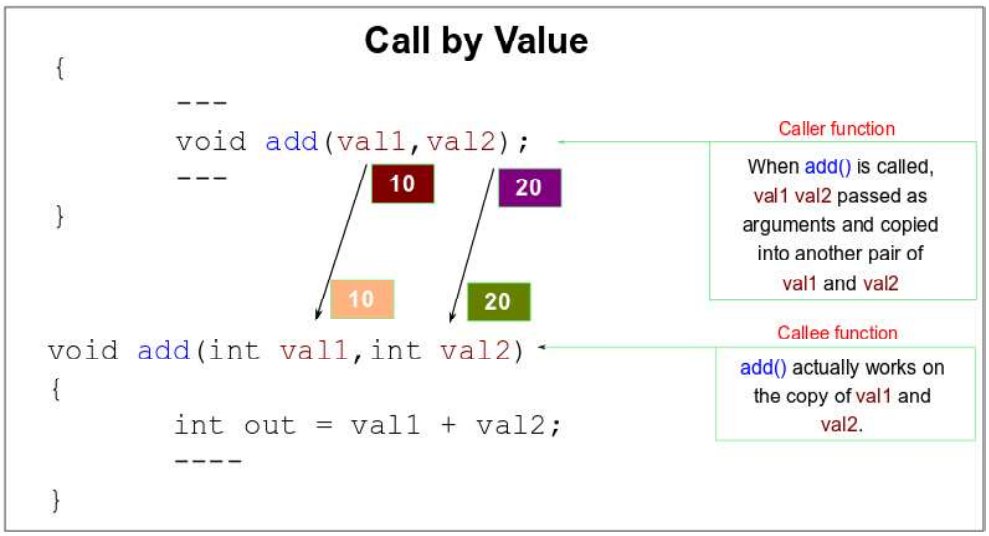


Figure 7.2 Parameter transfer using call by value.

As we can see from Figure 7.2, when a call to add(val1,val2) is made, the callee function declares two same type variables and copies the actual argument values into the corresponding formal arguments. The actual and formal arguments may have the same name (as in this case) since these variables belong to different blocks. An example program to understand how call by value works is given below.

Program 7.7:C++ program to swap two values using call by value

```
#include<iostream.h>
// function definition of swap().
void swap(int x, int y)
{
      int temp;     //Declare a temporary variable
      temp = x;    //temporarily save the value of x
      x = y;         //Copy value of y into x
      y = temp;     // Copy value of temp into y
}
int main ()
{
      int val1 = 5, val2 = 10;
```

```
        cout << "Before swap, value of val1= " << val1 << endl;
        cout << "Before swap, value of val2= " << val2 << endl;
        cout << "--------------------" << endl;
        //Calling the swap().
        swap(val1, val2);
        cout << "After swap, value of val1= " << val1 << endl;
        cout << "After swap, value of val2= " << val2 << endl;
        return 0;
}
```

Output:

```
Before swap, value of val1= 5
Before swap, value of val2= 10
-----------------------------
After swap, value of val1= 5
After swap, value of val2= 10
```

The above program tries to swap values of two variables using call by value approach. It has a function *swap()*to do the actual swapping. We can see how two times values of val1 and val2 are printed i.e., before and after swapping of values. This is done to check whether the swapping of formal arguments reflects on the actual arguments or not. From the output window, we can see that although the swap function did its job but still the value of val1 and val2 in the caller function remains unchanged.

This is in accordance with the discussion we had above. Therefore, the bottom line is, in a call by value the called function works on the copy of the actual variablesand hence, any modifications made on the copies do not reflect on the original variables.

7.6.2 Call by Pointer (*)

In call by pointer, the callee function works on the address of the actual argument. C++ allows us to use *value at address operator (*)* in copying the address of an argument into the formal parameter.

This is different from call by value where the callee function works on the copy of actual arguments. To understand the concept of call by pointer approach, we can see the diagrammatic representation given in Figure 7.3.

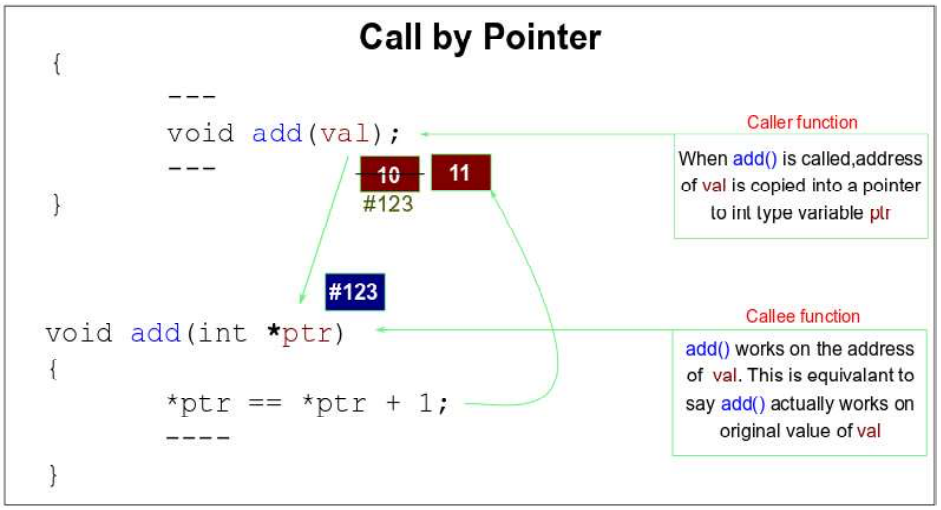


Figure 7.3 Parameter transfer using call by pointer.

For implementation purposes, we will modify the function call and definition part of the ongoing swapping problem.

Program 7.8: C++ program to swap two values using call by pointer

```
#include<iostream.h>
// function definition of swap().
void swap(int *x, int *y)
{
      int temp;   //Declare a temporary variable
      temp = *x; //Assign the value at address stored in x to temp
      *x = *y; //Assign value pointed by *y into *x
      *y = temp; //Assign value of temp into *y
}
int main ()
{
int val1 = 5, val2 = 10;
      cout << "Before swap, value of val1= " << val1 << endl;
```

```
        cout << "Before swap, value of val2= " << val2 << endl;
        cout << "--------------------" << endl;
        //Calling the swap().
        swap(&val1, &val2);
        cout << "After swap, value of val1= " << val1 << endl;
        cout << "After swap, value of val2= " << val2 << endl;
        return 0;
}
```

Output:

```
Before swap, value of val1= 5
Before swap, value of val2= 10
----------------------------
After swap, value of val1= 10
After swap, value of val2= 5
```

So, from the output window, we can see that the original value of val1 and val2 have changed after swapping. This is so because when the swap(&val1,&val2) is called, addresses of val1 and val2 are copied into the formal argument. Therefore, when swap() makes any changes on the formal parameters, the changes get reflected on the original variable.

7.6.3 Call by Reference (&)

In this method, the reference of an argument is copied into the formal parameter. This method is similar to the method of call by pointer. It is important that we do not confuse between both the methods as pointer is a variable that stores the address of another variable. Whereas, reference is a variable that refers to another variable. The only difference in both methods lies in how we declare the formal arguments. In the case of call by pointer we used *value at address operator* (*), but in case of call by reference we use the *address of operator* (&). Therefore, the previous program can be changed as shown below to demonstrate call by reference.

Program 7.9: C++ program to swap two values using call by reference

```
#include<iostream.h>

// function definition of swap().
void swap(int &x, int &y) {
   int temp;            //Declare a temporary variable
   temp = x;            //Assign value of x into temp
   x = y;               //Assign the value of y into x
   y = temp;            //Assign the value of temp into y
}

int main () {
   int val1 = 5, val2 = 10;

   cout << "Before swap, value of val1= " << val1 << endl;
   cout << "Before swap, value of val2= " << val2 << endl;
   cout << "-------------------------------" << endl;

   //Calling the swap().
   swap(val1, val2);

   cout << "After swap, value of val1= " << val1 << endl;
   cout << "After swap, value of val2= " << val2 << endl;

   return 0;
}
```

Output:

```
Before swap, value of val1= 5
Before swap, value of val2= 10
-----------------------------
After swap, value of val1= 10
After swap, value of val2= 5
```

We can understand the difference between call by pointer and call by reference by looking at the way swap() is defined. There is a

subtle difference in the manner in which pointer and reference operators are used. Although the output of both the programs are same but they are different in the way pointer variables are used in accessing the arguments passed.



CHECK YOUR PROGRESS

Q 1: Which of the following is not an advantage of function in C++?

(a) Simplicity

(b) Reusability

(c) Easier debugging

(d) None of the above

Q 2: Starting point of execution of a C++ program is _____.

Q 3: Function prototype can also be termed as function declaration (True/False).

Q 4: Parameters available in a function definition are called as _____.

Q 5: Which of the following is not a way to pass parameters to a function in C++?

(a) Pass by value

(b) Pass by data type

(c) Pass by reference

(d) Pass by pointer

7.7 TYPES OF FUNCTION

C++ provides broadly two types of functions:

- Built-in (library) Functions
- User-defined Functions

7.7.1 Built-in Function

A built-in function or a library function is made available by C++ as a part of the overall package. Here, library refers to a collection of sub-programs useful in developing other programs. We can use these functions without explicitly declaring or defining them. We can

directly call them whenever required. The only requirement is that linking of proper library function has to be done from the source program. An example program to explain the usage of built-in functions is given below.

Program 7.10: C++ program to demonstrate usage of built-in function

```
#include <iostream.h>
#include <math.h>        //Linking to the library

int main ()
{
    int result = pow (2, 5)     //Calling the built-in pow(x, y)
    cout <<"5 raised to the power of 2 = " << result;

    return 0;
}
```

Output:

```
5 raised to the power of 2 = 32
```

The above program uses *pow(x, y)* to compute the value of x raised to the power of y (i.e. x^y).*pow()* is a built-in function available with *math.h* header file.As we are using built-in or library function, we do not have to explicitly declare or define this pow(). Instead, we just have to include the proper header file to the program which is *math.h* in this case.

7.7.2 User-Defined Function

Unlike built-in functions, user-defined functions are declared and defined by the programmer. As these functions are explicitly constructed, hence these are not part of any default libraries available with C++. There are many example programs already presented in this unit explaining user-defined functions. Yet another program in light of the user-defined function is given below.

Program 7.11: C++ Program to convert Fahrenheit to Celsius

```
#include <iostream.h>

float f_to_c(float far)
{
    float cel;
    cel = (far - 32.0) * 5.0 / 9.0;      //Conversion formula

    return cel;
}

int main()
{
    float far;

    cout << "Input temperature in Fahrenheit: ";
    cin  >> far;
    cout << "Temperature (Deg. C) = " << f_to_c(far) << endl;

    return 0;
}
```

Output:

```
Input temperature in Fahrenheit: 98
Temperature (Deg. C) = 36.6667
```

In the above program, f_to_c(float) is the user-defined function.

7.8 INLINE FUNCTION

Functions provide plenty of benefits to the programmer.One such benefit is code reuse. You can write once and use it many times. However, there are certain issues involved with it. Every time a function call is executed, CPU has to perform a certain set of tasks, e.g.:

- CPU stores the address of the next instruction. This helps in resuming program execution after completing the function execution.
- It copies the function arguments onto the execution stack, and

- Transfers control to the called function.

This process of saving current execution status and transferring of control can become an overhead if the execution time of a function is less than the overall switching time. For a comparatively larger function (e.g., functions having loops, recursion), this overhead might not be significant in comparison to the overall switching cost. However, for smaller functions, the overhead may suppress the function execution cost. In such cases, it will be practical to replace the function call with its body so that the switching cost can be saved.

C++ provides a mechanism to achieve this through the use of inline function. An inline function is a function that is expanded in line whenever it is called. It means, when an inline function is called the body of the function replaces the function call itself. This substitution is done by the compiler and it happens during the compilation phase. Hence, CPU has no need to switch from one point to another resulting in reduced switching overhead. The syntax of defining an inline function is:

```
inline  return_type  function_name(args)
{
    // function body
}
```

An example of inline function implementation is shown below. The program demonstrates a simple case of incrementing an integer value using an inline function. The function is a very trivial one where a variable is passed and its value is incremented before returning it back to the called program. As we can see that the function in this case, has just one single task to do that is incrementing. But if we compare the overall switching cost and the cost of direct execution of function then the switching overhead is likely to become significant. Therefore, it is sensible to make the increment function as inline.

Program 7.12: C++ Program to increment a value using inline function

```
#include <iostream.h>
inline int increment(int x)
{
    x = x + 1;  // increment operation
    return x;
}
int main()
{
    int val;
    cout<<"\n Enter a integer value:";
    cin>> val;
    cout << "Original value of Val is: " << val << endl;
    val = increment(val);
    cout << "Val++: " << val << endl;
    return 0;
}
```

As a programmer, there are certain things we have to be aware of while working with inline functions. The most important thing is making a function inline is a request to the compiler and not a command. So, when a function is made inline, the compiler has the right to decide whether or not the call be replaced with its body. In general, a compiler may choose not to extend the code if:

- A function's body contains looping constructs (e.g., *for, while, do, do-while* loop).
- The function is a recursive function.
- The function contains *switch* or *goto* statements in its body.
- The function makes usage of static functions and variables.

Another example of inline function is given below. Here, we are trying to find the sum of first *n* natural numbers using an inline function. In this case, as mentioned earlier, the compiler may or may not choose to replace the function call with it body as looping is involved. We can understand this fact

by considering two values e.g., 2 and 250 as the user input. In the case of the first input i.e., 2, only two iterations would be sufficient to calculate the output. Whereas, in the second case 250 iterations would be required. That's a huge cost and we may assume that this cost may overrun the overall switching cost. Hence, in such a situation, it becomes uncertain whether or not the code expansion will take place. The decision solely lies on the compiler itself.

Program 7.13: C++ Program to add first n natural numbers using inline function

```
#include <iostream.h>
inline int sum(int x)
{
      int temp = 0;
      for(int i=x; i>=0; i—)
      {
      temp = temp + i; // incrementally adding numbers
      }
      return temp;.
}
      int main()
      {
      int num;
      cout << "Enter the number(n): ";
      cin >> num;
      int out = sum(num);
      cout << "The sum of n numbers is: " << out << endl;
      return 0;}
```

Output:

Enter the number(n): 2

The sum of n numbers is: 3

We have seen some of the advantages of using inline functions with examples. Let us also know about the disadvantages as well. One of the major disadvantages is that inline functions can nullify the benefits of using functions i.e., code reuse. If a function call is replaced with its body every time an inline function is called then the very concept of "*write once and use many times*" gets invalidated. Although the function body may be very small but when you replace ten times or more then the overall size of the program gets increased. Therefore, it is very important on the programmer's behalf that the use of inline functions be done carefully.

7.9 FUNCTION OVERLOADING

Overloading refers to loading additional definitions on top of the existing one. This may sound confusing but things are very simple in reality. It essentially means that the same function name can be used to define multiple functions to perform multiple tasks. Since the same function exhibits multiple forms, we can term it as *function overloading* or *function polymorphism*.

Function overloading enables us to bring implementation closer to the problem definition. For an instance, if we are to find the area of a given geometrical shape, ideally our function name should appear as *area()*. But having a single area() function might not be sufficient to calculate the area of different shapes (e.g. Triangle, Circle etc.). C++ allows us to use the same *area()* function in different forms to calculate the area of different shapes. For example:

```
float area(float r) {return(3.14 * r * r);}          //Area of Circle
float area(float b, float h) {return(0.5 * b * h);}  //Area of Triangle
```

As we can see from above that using function overloading we can define a family of *area()* functions meant to calculate the area of circle and triangle.Both the functions have the same name yet they are different as they perform a different task. Is this possible? This is possible because the linking of a function call to its definition is determined by checking the number,

order, and type of arguments used in the function call. In this case, a call to *area()* function with area(2.5) is different from area(2.5, 2.5). The first call will match with the signature of area(float r){} and hence it will lead to the calculation of the area of circle. Similarly, the second function call would match with the signature area(float b, float h){} and hence this will lead to the calculation of the area of a triangle. The program given below will help you to understand it better.

Program 7.14:C++ program to calculate area using function overloading

```
#include<iostream.h>
float area(float r)
{
return(3.14 * r * r);
}
float area(float b,float h)
{
return(0.5 * b * h);
}
int main()
{
      float b,h,r;
      cout<<"Enter the Radius of Circle: ";
      cin>>r;
      cout<<"Area of Circle= "<<area(r) << endl;
      cout<<"\nEnter the Base & Height of Triangle: ";
      cin>>b>>h;
      cout<<"Area of Triangle= "<< area(b,h) << endl;
      return 0;
}
```

Output:

Enter the Radius of Circle: 2.5

Area of Circle= 19.625

Enter the Base & Height of Triangle: 2.5 2.5
Area of Triangle= 3.125

As we can see from the above program that the first call to the *area()* effectively links with the calculation of area of a circle. This is because only one float type argument (i.e., 2.5) is passed in the function call. When this call is made compiler looks for the best-fit match among all available function definitions. In this case area(float r) provides the best match as the other version of *area()* expects two float type arguments. Similarly, the second call links with the area of triangle calculation. Since, there is only one area() function available that accepts two float type arguments and thus provides the best match against the area(b, h) function call.

Now imagine what will happen if we try to include another function definition into the same program. Let's say, we want to compute the area of a rectangle. In that case, we have to make room for a new*area()* definition as shown below.

```
float area(float l, float b) {return(l * b);}        //Area of Rectangle
```

The modified program will look like:

Program 7.15: Modified C++ program to calculate the area using function overloading

```
#include<iostream.h>
float area(float r){return(3.14 * r * r);}
float area(float b, float h){return(0.5 * b * h);}
float area(float l, float b){return (l * b);}
int main()
{
    float b,h,r,l, br;
    cout<<"Enter the Radius of Circle: ";
    cin>>r;
    cout<<"Area of Circle= "<< area(r) << endl;
    cout<<"\n Enter the Base & Height of Triangle: ";
    cin>>b>>h;
```

```
    cout<<"Area of Triangle= "<< area(b,h) << endl;
   cout<<"\n Enter the Length & Breadth of Rectangle: ";
    cin>>l>>br;
    cout<<"Area of Rectangle= "<< area(l,br) << endl;
    return 0;
}
```

Everything seems fine till this moment. But, on compilation, we will have some problems as shown below.

```
error: redefinition of 'float area(float, float)'
float area(float l,float b){return (l * b);}
     ^~~~
note: 'float area(float, float)' previously defined
here
float area(float b,float h){return(0.5 * b * h);}
     ^~~~
```

We can observe that the compilation process results in an error. This is because the new definition of *area()* is the same in signature with one of the existing *area()* function. Thus, when a call to area(l,br) is made, it becomes ambiguous so as to which version of the matching area(float, float) is to be executed. Hence, it is important to perform function overloading with caution. We should never overload unrelated functions restricted to only closely related tasks. Doing so may help to reduce the chances of any inadvertent errors. General rules of linking a function call to an appropriate definition are:

1. Firstly, the compiler looks for an exact match in terms of number, sequence, and type of arguments are used in the function call. If a unique match is found then that function is executed.

2. If an exact match is not found then an integration promotion is performed. E.g., Char, Unsigned char, and short are promoted to an int. Similarly, a float is promoted to double to find a match.

3. Still, there is no match then the compiler tries to find a match using implicit type conversion (e.g., int to long or double).

4. Even after this, if there are no match or match with multiple function definition is found then the compiler generates an error.



CHECK YOUR PROGRESS

Q 6: Which of the following is a user-defined function?

(a) pow()
(b) strlen()
(c) main()
(d) None of the above

Q 7: When an inline function is called the body of the function replaces the _____.

Q 8: A recursive function may not be worked as inline function.(True/ False)

Q 9: The program size of a C++ program will be decreased if the functions are declared as inline.(True /False)

Q 10: In _____, same function name can be used for multiple functions.



7.10 LET US SUM UP

The keypoints to keep in mind in this unit are:

- Modular programming is important in handling problem complexity.
- Abstraction and decomposition are two techniques used in implementing modular programming. Functions provide a way to bring these concepts into the programming domain.
- Functions can facilitate the use of procedural abstraction in the top-down design of programs.
- A function is a collection of statements that together perform a specific task.

- The main components of functions are function prototype, function call, function definition and function parameters.
- Functions can pass parameters, which may either be value parameters or reference parameters. A function can pass arguments in three different ways i.e., call by value, address and reference.
- C++ programs can include user-defined functions in addition to built-in functions.
- Inline functions are functions that are expanded in line. It can save from the overhead of context switching during program execution.
- We can define multiple functions with the same name. This is basically known as function overloading.
- Function overloading is a form of polymorphism.



7.11 FURTHER READING

1. Balagurusamy, E. (2011). *Object Oriented Programming with C++, 6e*. McGraw-Hill Education (India) Pvt Limited. https://books.google.co.in/books?id=TN9wQjjDwp0C
2. ISO/IEC. (2003). International Standard ISO/IEC 14882 Second Edition Programming languages - C++. *ISO/IEC*, 786. http://www.iso.org/iso/catalogue_detail.htm?csnumber=38110
3. Pardoe, J., & King, M. (1997). Object Oriented Programming Using C++. In *Object Oriented Programming Using C++*. https://doi.org/10.1007/978-1-349-14449-5
4. Plauger, P. J. (2002). The C/C++ programming language. *C/C++ Users Journal*.
5. Stroustrup, B. (2014). A Tour of C++. In *Climate Change 2013 - The Physical Science Basis*.
6. Van Weert, P., & Gregoire, M. (2016). C++ Standard Library Quick Reference. In *C++ Standard Library Quick Reference*. https://doi.org/10.1007/978-1-4842-1876-1



7.12 ANSWERS TO CHECK YOUR PROGRESS

Ans to Q No 1: (d) None of the above

Ans to Q No 2: main() function

Ans to Q No 3: True

Ans to Q No 4: Formal parameters

Ans to Q No 5: (b) Pass by data type

Ans to Q No 6: (c) main()

Ans to Q No 7: Function call

Ans to Q No 8: True

Ans to Q No 9: False

Ans to Q No 10: Function overloading



7.13 MODEL QUESTIONS

Q1. What is a function? Write a C++ program to explain call by value and call by reference concept.

Q2. Write down the advantages and disadvantages of functions in C++.

Q3. Write a C++ program to implement a function named *absVal()* that returns the absolute value of an integer passed to it by value. Do not use any special library functions.

Q4. What is an inline function? List out the conditions critical in deciding whether a function would be made inline or not.

Q5. What do you mean by function argument? Write a C++ program to swap two integers without using any temporary variable. The program should use a function for this purpose.

Q6. What is function overloading? Write a C++ program to overload a function *convert()* so that it can be used in converting different measurement scales such as meter-centimeter, kilogram-gram, etc.

*** ***** ***