

# Introduction to C++

- 1.1 Evolution of C++
- 1.2 ANSI Standard
- 1.3 Object-Oriented Technology
- 1.4 Disadvantage of Conventional Programming
- 1.5 Programming Paradigms
- 1.6 Preface to Object-Oriented Programming
- 1.7 Key Concepts of Object-Oriented Programming
- 1.8 Advantages of OOP
- 1.9 Object-Oriented Languages
- 1.10 Usage of OOP
- 1.11 Usage of C++

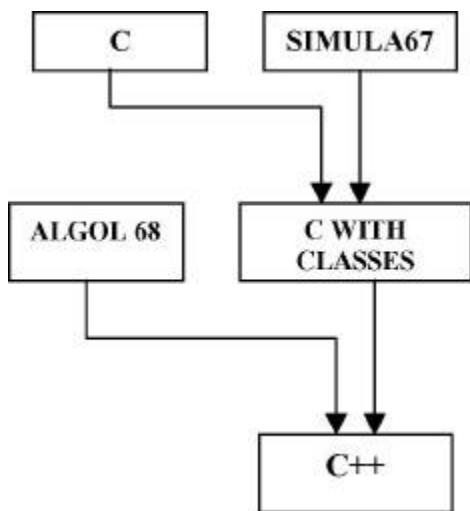
## 1.1 EVOLUTION OF C++

C++ is an object-oriented programming language and is considered to be an extension of C. Bjarne Stroustrup at AT&T Bell Laboratories, Murray Hill, New Jersey (USA) developed it in the early eighties of twentieth century. Stroustrup, a master of Simula67 and C, wanted to combine the features of both the languages into a more powerful language that could support object-oriented programming with features of C. The outcome was C++ as per Figure 1.1. Various ideas were derived from **SIMULA67** and **ALGOL68**. Stroustrup called the new language '**C with classes**'. However, in 1983, the name was changed to C++. The

thought of C++ came from the C increment operator `++`. Rick Mascitti coined the term C++ in 1983. Therefore, C++ is an extended version of C. C++ is a superset of C. All the concepts of C are applicable to C++ also. For developing complicated applications object-oriented languages are most convenient and easy. Hence, a programmer must be aware of the new concepts of object-oriented techniques.

## 1.2 ANSI STANDARD

**ANSI** stands for **American National Standards Institute**. This Institute was founded in **1918**. The main goal for establishing this Institute was to suggest, reform, recommend, and publish standards for data processing in the USA. This committee sets up standards for the computer industry. The recognized council working under the procedure of the American National Standard Institute (ANSI) has made an international standard for C++. The C++ standard is also referred to as ISO (International Standard Organization) standard. The process of standardization is gradual and the first draft of the planned ANSI standard was made on **25 January 1994**. This book will continue to refer to ANSI standard code, which is more commonly used. The ANSI standard is an attempt to ensure that C++ is portable.



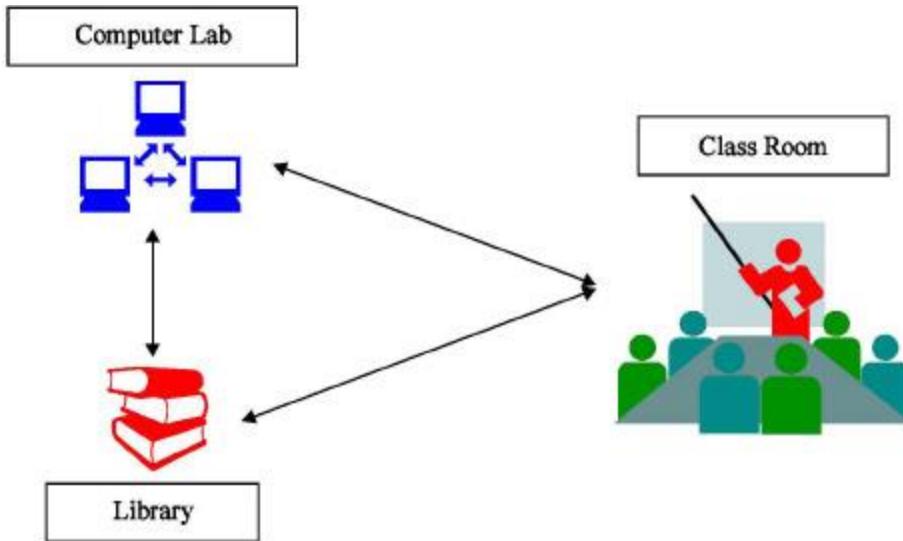
**Fig. 1.1** Evolution of C++

## 1.3 OBJECT-ORIENTED TECHNOLOGY

Nature is composed of various objects. Living beings can be categorized with different objects.

Let us consider an example of a teaching institute having two different working sections. They are teaching and non-teaching. Further sub-grouping of teaching and non-teaching can be made for the ease of management. The various departments of any organization can be thought as objects working with certain goals and objectives.

Usually, an institute has faculties for different departments. Also, laboratory staff for the assistance in conducting of practicals and site development section for beautification of the campus are essential. The Principal is a must for the overall management of the Institute. The accounts department is also required for handling monetary transactions, and salaries of the employees. The sports section is entrusted the responsibility of sports activities, the Registrar for administration and his staff for dealing with administrative-matters of the Institute and so on are required. Each department has an In-Charge who has clear-cut responsibilities. Every department has its own work as stated above. When the work is distributed to different departments as shown in Figure 1.2, it becomes easy to accomplish goals and objectives. The activities are carried out smoothly. The burden of one particular department has to be shared among personnel. The staff in the department are controlled properly and act according to the instructions laid down by the management. The faculty performs activities related to teaching. If higher authority needs to know the details regarding the theory, practical, seminar and project workload of individuals of the department then some responsible person from the department furnishes the same. This way some responsible person from the department accesses the data and provides the higher authority with requisite information. Only authorized persons have access to data base. As shown in Figure 1.2 an institute is divided into different departments such as library, classroom, computer laboratory, etc. Each department performs its own activities in association with other departments. This theory of objects can be extended to every walk of life and can be implemented with the help of software. In general, objects are in terms of entities.



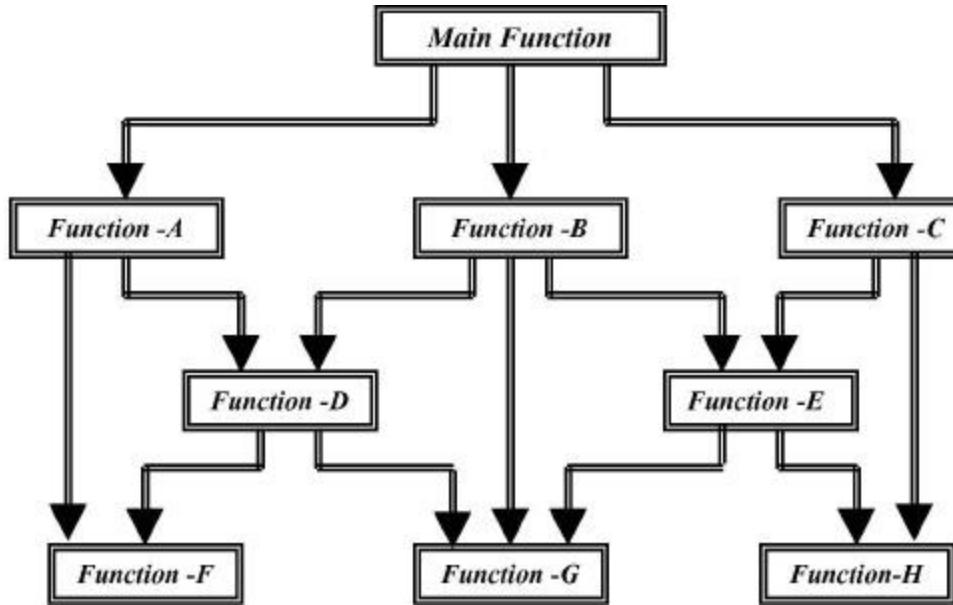
**Fig. 1.2** Relationship between different sections

#### 1.4 DISADVANTAGE OF CONVENTIONAL PROGRAMMING

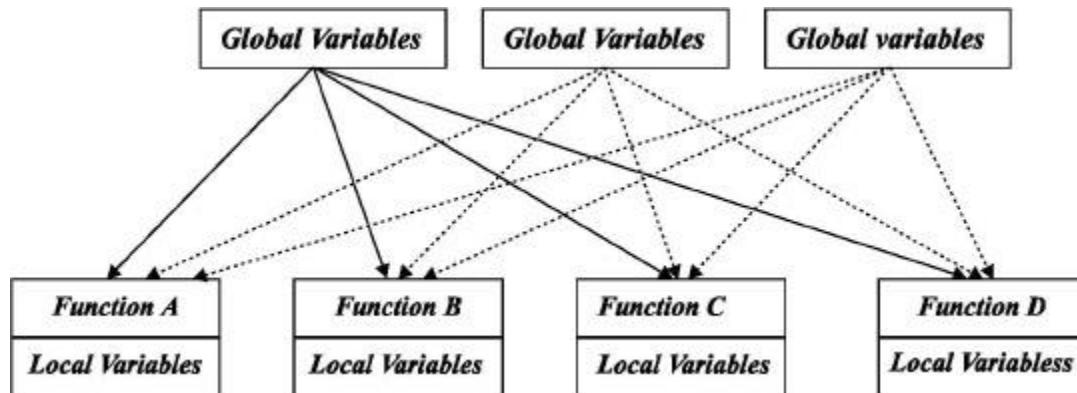
Traditional programming languages such as COBOL, FORTRAN etc. are commonly known as procedure oriented languages. The program written in these languages consists of a sequence of instructions that tells the compiler or interpreter to perform a given task. When program code is large then it becomes inconvenient to manage it. To overcome this problem, procedures or subroutines were adopted to make a program more understandable to the programmers. A program is divided into many functions. Each function can call another function as shown in Figure 1.3. Each function has its own task. If the program is too large the function also creates problems. In many programs, important data variables are declared as global. In case of programs containing several functions, every function can access the global data as simulated in Figure 1.4. Generally in a program of large size, it becomes difficult to understand the various data used in different functions. As a result, a program may have several logical errors.

Following are the drawbacks observed in monolithic, procedural, and structured programming languages:

- (1) Large size programs are divided into smaller programs known as functions. These functions can call one another. Hence, security is not provided.
- (2) Importance is not given to security of data but on doing things.
- (3) Data passes globally from one function to another.
- (4) Most functions have access to global data.



**Fig. 1.3** Flow of functions in non-OOP languages



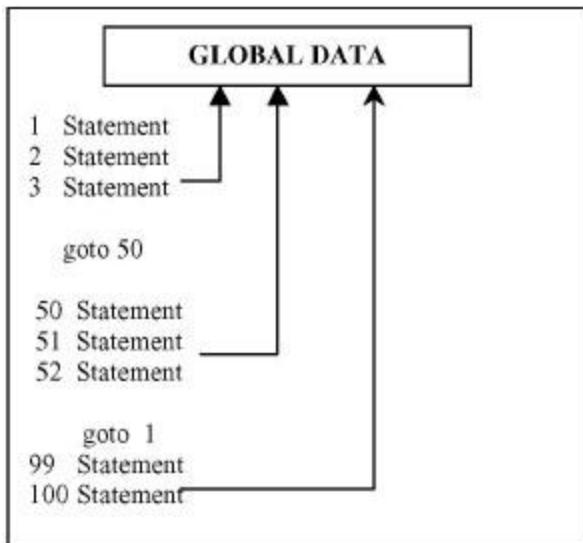
**Fig. 1.4** Sharing of data by functions in non-OOP languages

## 1.5 PROGRAMMING PARADIGMS

### (1) MONOLITHIC PROGRAMMING

(A) In monolithic programming languages such as BASIC and ASSEMBLY, the data variables declared are global and the statements are written in sequence.

(B) The program contains jump statements such as `goto`, that transfer control to any statement as specified in it. Figure 1.5 shows a program of monolithic type. The global data can be accessed from any portion of the program. Due to this reason the data is not fully protected.



**Fig. 1.5** Program in monolithic programming

(C) The concept of subprograms does not exist and hence useful for smaller programs.

## (2) PROCEDURAL PROGRAMMING

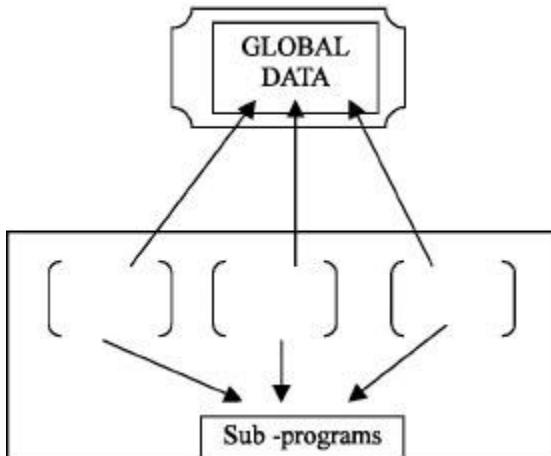
(A) In procedural programming languages such as FORTRAN and COBOL, programs are divided into a number of segments known as subprograms. Thus it focuses on functions apart from data. Figure 1.6 describes a program of procedural type. It shows different subprograms having access to the same global data. Here also, the data is not fully protected.

- (B) The control of program is transferred using unsafe `goto` statement.
- (C) Data is global and all the subprograms share the same data.
- (D) These languages are used for developing medium size applications.

## (3) STRUCTURED PROGRAMMING

(A) In structured programming languages such as PASCAL and C larger programs are developed. These programs are divided into multiple submodules and procedures.

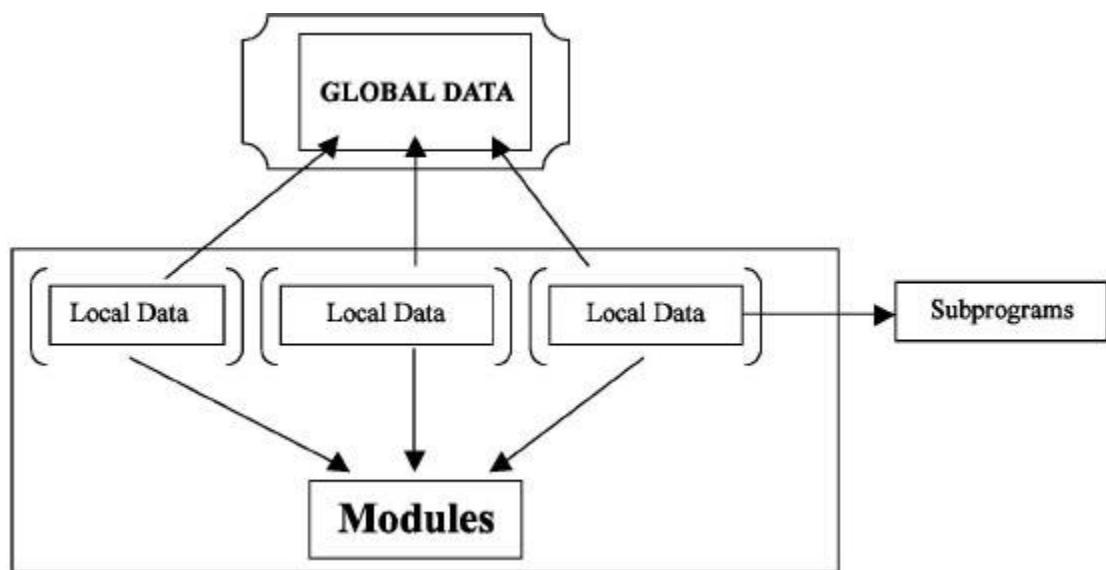
- (B) Each procedure has to perform different tasks.
- (C) Each module has its own set of local variables and program code as shown in Figure 1.7. Here, the different modules are shown accessing the global data.
- (D) User-defined data types are introduced.



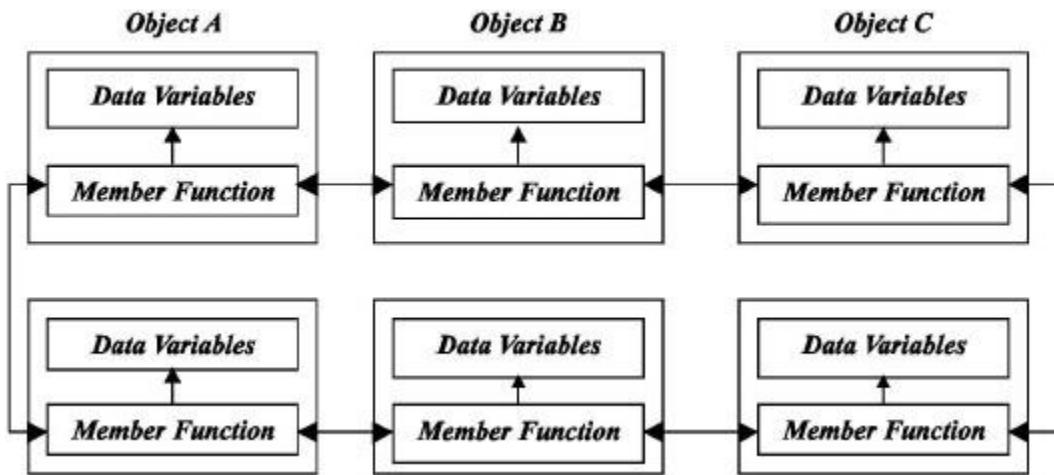
**Fig. 1.6** Program in procedural programming

## 1.6 PREFACE TO OBJECT-ORIENTED PROGRAMMING

The prime factor in the development of object-oriented programming approach is to remove some of the shortcomings associated with the procedure oriented programming. OOP has data as a critical component in the program development. It does not let the data flow freely around the systems. It ties data more firmly to the functions that operate on it and prevents it from accidental change due to external functions. OOP permits us to analyze a problem into a number of items known as objects and then assembles data and functions around these items as shown in Figure 1.8. The basic objective of OOP is to treat data and the program as individual objects. Following are the important characteristics of object-oriented programming:



**Fig. 1.7** Program in structured programming



**Fig. 1.8** Relation between data and member function in OOP

- (1) OOP pays more importance to data than to function.
- (2) Programs are divided into classes and their member functions.
- (3) New data items and functions can be comfortably added whenever essential.
- (4) Data is private and prevented from accessing external functions.
- (5) Objects can communicate with each other through functions.?

## OBJECT-ORIENTED PROGRAMMING

Object-oriented programming language is a feature that allows a mode of modularizing programs by forming separate memory area for data as well as functions that is used as object for making copies of modules as per requirement.

### 1.7 KEY CONCEPTS OF OBJECT-ORIENTED PROGRAMMING

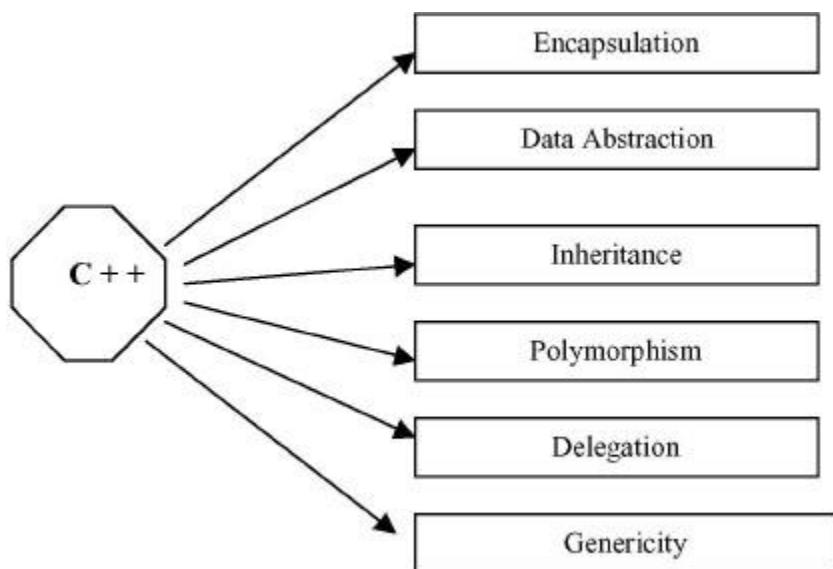
There are several fundamental or key concepts in object-oriented programming. Some of these are shown in Figure 1.9 and are discussed below:

#### (1) OBJECTS

##### OBJECT

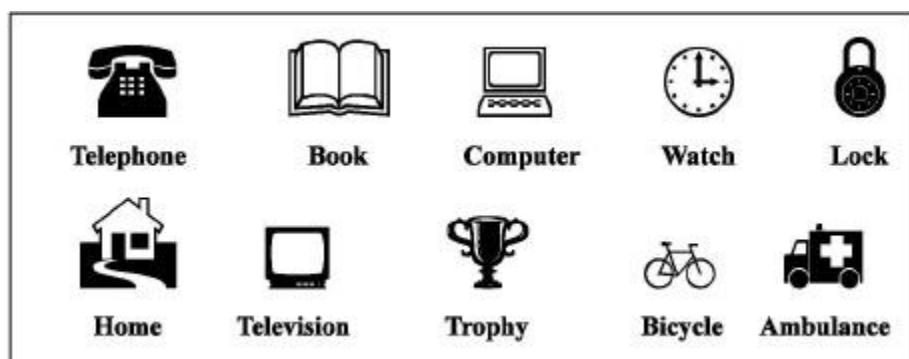
Objects are primary run-time entities in an object-oriented programming. Objects are primary run-time entities in an object-oriented programming. They may stand for a thing that has specific application for example, a spot, a person, any data item related to program, including user-defined data

types. Programming issues are analyzed in terms of object and the type of transmission between them. Figure 1.10 describes various objects. The selected program objects must be similar to actual world objects. Objects occupy space in memory. Every object has its own properties or features. An object is a specimen of a class. It can be singly recognized by its name. It declares the state shown by the data values of its characteristic at a specific time. The state of the object varies according to procedure used. It is known as the action of the object. The action of the object depends upon the member function defined within its class.



**Fig. 1.9** Features of object-oriented programming

Given below are some objects that we use in our daily life.



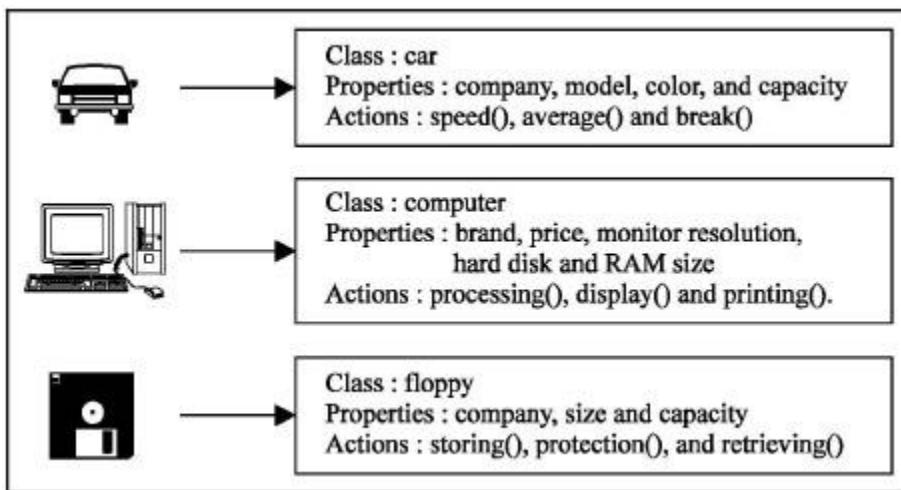
**Fig. 1.10** Commonly available objects

## (2) CLASSES

## CLASS

A class is grouping of objects having identical properties, common behavior, and shared relationship.

A class is the accomplishment of abstract data type. It defines the nature and methods that act on the data structure and abstract data type respectively. Specimens are also known as objects. In other words, a class is a grouping of objects having identical properties, common behaviour, and shared relationship. The entire group of data and code of an object can be built as a user-defined data type using class. Objects are nothing but variables of type class. Once a class has been declared, the programmer can create a number of objects associated with that class. The syntax used to create an object is similar to the syntax used to create an integer variable in C. A class is a model of the object. Every object has its own value for each of its member variables. However, it shares the property names or operations with other instances of the class. Thus, classes define the characteristics and actions of different objects.



**Fig. 1.11** Objects and their properties

In Figure 1.11 some common objects, their properties, and the actions they perform have been described. A car can be distinguished from another car on the basis of its properties, for example, company, model, color, capacity etc. and also on the basis of its actions for example, speed, average, break etc.

Figure 1.12 describes different classes and related data that come under them.

## (3) METHOD

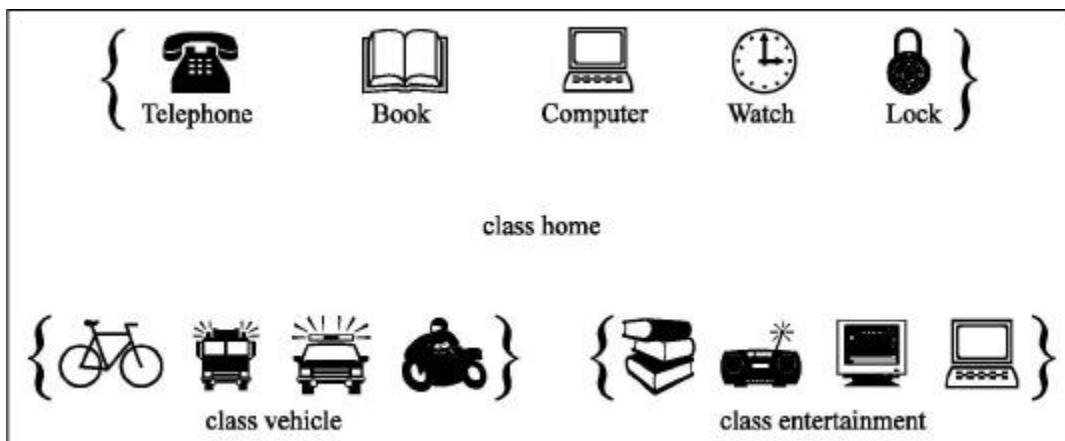
## METHOD

An operation required for an object or entity when coded in a class is called a method.

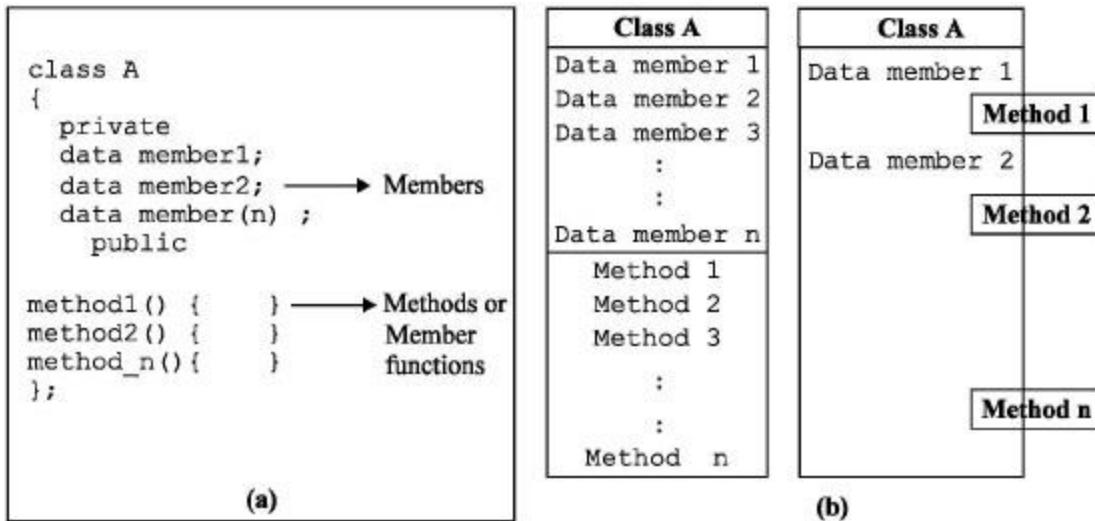
An operation required for an object or entity when coded in a class is called a method. The operations that are required for an object are to be defined in a class. All objects in a class perform certain common actions or operations. Each action needs an object that becomes a function of the class that defines it and is referred to as a method.

In Figure 1.13, the class, its associated data members, and functions are shown in different styles. This style is frequently used while writing a class in the program.

The class A contains private data members and public methods or member functions. Generally, the data members are declared private, and member functions are declared public. We can also define public data members and private member functions. The data member of any class uses its member functions or methods to perform operations.



**Fig. 1.12** Classes and their members



**Fig. 1.13** Representation of methods in different manners

#### (4) DATA ABSTRACTION

##### DATA ABSTRACTION

Abstraction directs to the procedure of representing essential features without including the background details.

Abstraction directs to the procedure of representing essential features without including the background details. Classes use the theory of abstraction and are defined as a list of abstract properties such as size, cost, height, and few functions to operate on these properties. Data abstraction is the procedure of identifying properties and methods related to a specific entity as applicable to the application.

A powerful method to achieve abstraction is by the manipulation of hierarchical classifications. It permits us to break the semantics of multiple systems into layers, by separating them into multiple controllable parts. For example, a computer as shown in Figure 1.14 is made of various parts such as cpu, keyboard, and so on. We think it as a single unit, but it has several sub units. These units when combined together perform the required task. By assembling subparts we can make a single system.



**(a)** Computer as a single unit

**(b)** Different components of a computer

**Fig. 1.14** Computer and its parts

Hierarchical abstraction of complicated systems can also be used in computer software. The data from conventional procedure oriented programs can be converted by abstraction mechanism into its partial objects. A series of operation steps may develop a set of messages between these objects. Each object shows its own attributes.

## (5) ENCAPSULATION

### ENCAPSULATION

The packing of data and functions into a single component is known as encapsulation.

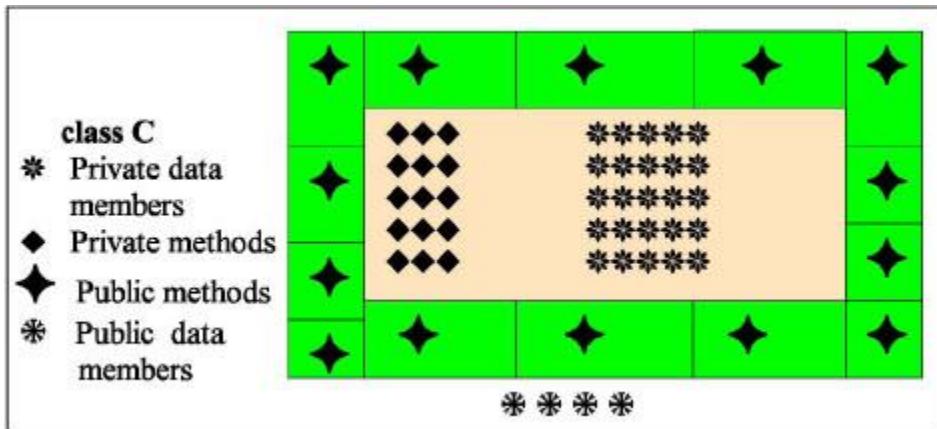
C++ supports the features of encapsulation using classes. The packing of data and functions into a single component is known as encapsulation. The data is not accessible by outside functions. Only those functions that are able to access the data are defined within the class. These functions prepare the interface between the object's data and the program. With encapsulation data hiding can be accomplished. Data hiding is an important feature. By data hiding an object can be used by the user without knowing how it works internally.

In C++, the fundamental of encapsulation is class. A class defines the structure of data and member functions. It is common to all its objects. Class is a logical structure whereas an object is a physical quantity. The goal of the class is to encapsulate complication. The class also has mechanism for hiding the data. Each member in the class may be private or public. Any non-member function cannot access the data of the class. The public section of the class must be carefully coded not to expose the inner details of the class. Figure 1.15 explains different sections of encapsulation.

## (6) INHERITANCE

### INHERITANCE

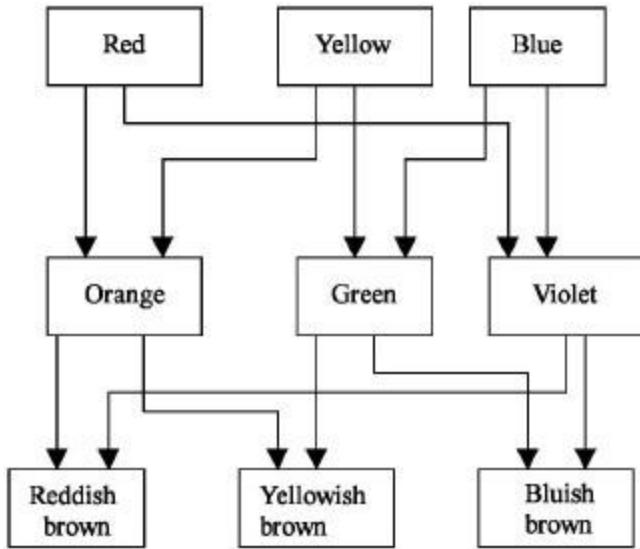
Inheritance is the method by which objects of one class get the properties of objects of another class.



**Fig. 1.15** Encapsulation: Private and Public sections

Inheritance is the method by which objects of one class get the properties of objects of another class. In object-oriented programming, inheritance provides the thought of reusability. The programmer can add new properties to the existing class without changing it. This can be achieved by deriving a new class from the existing one. The new class will possess features of both the classes. The actual power of inheritance is that it permits the programmer to reuse a class that is close to what he wants, and to tailor the class in such a manner that it does not bring any unwanted incidental result into the rest of the class. Thus, inheritance is the feature that permits the reuse of an existing class to make a new class. The Figure 1.16 shows an example of inheritance.

In Figure 1.16 red, yellow and blue are the main colours. Orange colour is created from the combination of red and yellow, green is created from yellow and blue, and violet is created from red and blue. Orange colour has attributes of both red and yellow, which produces a new effect. Thus, many combinations are possible.

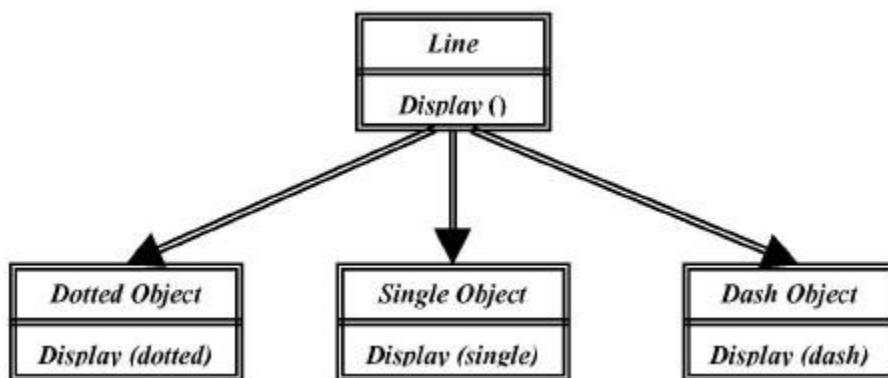


**Fig. 1.16** Inheritance

## (7) POLYMORPHISM

### POLYMORPHISM

Polymorphism allows the same function to act differently in different classes. Polymorphism makes possible the same functions to act differently on different classes as shown in Figure 1.17. It is an important feature of OOP concept and has the ability to take more than one form. Polymorphism accomplishes an important part in allowing objects of different classes to share the same external interface. It is possible to code a non-specific (generic) interface to a set of associated actions.



**Fig. 1.17** Polymorphism in OOP

## (8) DYNAMIC BINDING

### DYNAMIC BINDING

Binding means connecting one program to another program that is to be executed in reply to the call.

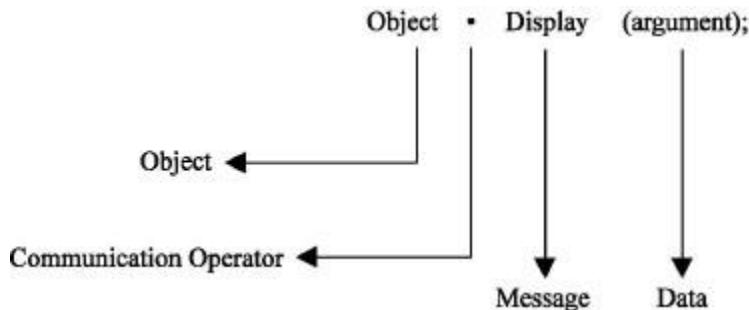
Binding means connecting one program to another program that is to be executed in reply to the call. Dynamic binding is also known as *late binding*. The code present in the specified program is unknown till it is executed. It is analogous to inheritance and polymorphism.

In Figure 1.17, polymorphism allows single object to invoke similar function from different classes. The program action is different in all the classes. During execution time, the code analogous to the object under present reference will be executed.

## (9) MESSAGE PASSING

Object-oriented programming includes objects which communicate with each other. Programming with these objects should be followed in steps shown below:

- (1) Declaring classes that define objects and their actions.
- (2) Declaring objects from classes.
- (3) Implementing relation between objects.



**Fig. 1.18** Message passing

Data is transferred from one object to another. A message for an object is the demand for implementation of the process. Message passing as shown in Figure 1.18 contains indicating the name of the object, function, and required data elements. Objects can be created, released, and interacted with each other. An object is workable, as long as it is active. In object-oriented programming, there is a panorama of independent objects that communicate with each other by swapping messages. Objects invoke member functions. They also negate if the calling object is not a member of the same class. Thus a message is a solicitation to an object to call one of its member functions. A message contains name of the member function and parameters of the function. Execution of member function is just a

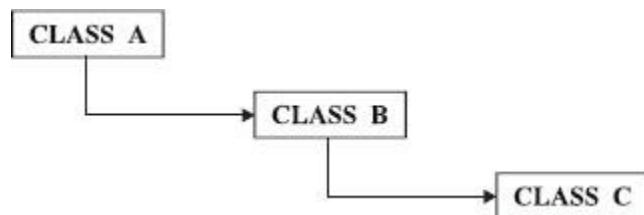
response generated due to receipt of a message. It is possible when the function and the object are of the same class.

## (10) REUSABILITY

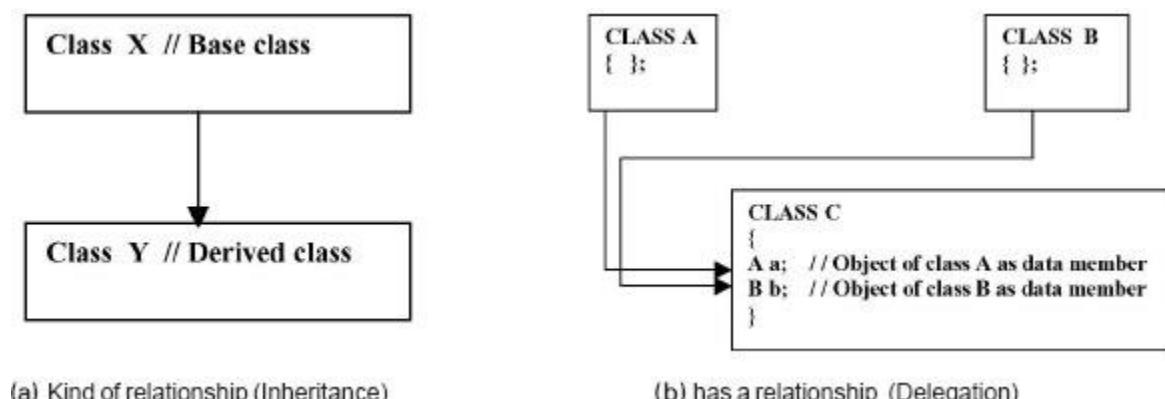
### REUSABILITY

Object-oriented technology allows reusability of the classes by extending them to other classes using inheritance.

Object-oriented technology allows reusability of classes by extending them to other classes using inheritance. Once a class is defined, the other programmer can also use it in their programs and add new features to the derived classes. Once verified the qualities of base classes need not be redefined. Thus, reusability saves time. In Figure 1.19 class A is reused and class B is created. Again class B is reused and class C is created.



**Fig. 1.19** Reusability



(a) Kind of relationship (Inheritance)

(b) has a relationship (Delegation)

**Fig. 1.20** Relationships between two classes

## (11) DELEGATION

In OOP, two classes can be joined either by - inheritance or delegation, which provide reusability of the class.

In inheritance, one class can be derived from other class and relationship between them is known as ***kind of relationship***. For example, if **class**

**Y** is derived from **class X**, then **class Y** is known as kind of **X**, as shown in Figure 1.20 (a).

The second type of relationship is **has a relationship**. When object of one class is used as data member in other class, such composition of objects is known as delegation. As shown in Figure 1.20 (b) class C has two data members that are objects of class A and B. Such a relationship between two classes is known as **has a relationship**.

## (12) GENERICITY

The software components of a program have more than one version depending on the data types of arguments. This feature allows declaration of variables without specifying exact data type. The compiler identifies the data type at run-time. The programmer can create a function that can be used for any type of data. The template feature in C++ allows generic programming.

## 1.8 ADVANTAGES OF OOP

Object-oriented programming provides many advantages to the programmer and the user. This technology solves many problems related to software development, provides improved quality, and low cost software.

OOP has the following advantages:

- (1) Object-oriented programs can be comfortably upgraded.
- (2) Using inheritance, redundant program codes can be eliminated and the use of previously defined classes may be continued.
- (3) The technology of data hiding facilitates the programmer to design and develop safe programs that do not disturb code in other parts of the program.
- (4) The encapsulation feature provided by OOP languages allows programmer to define the class with many functions and characteristics and only few functions are exposed to the user.
- (5) All object-oriented programming languages can create extended and reusable parts of programs.
- (6) Object-oriented programming enhances the thought process of a programmer leading to rapid development of new software in short span of time.

## 1.9 OBJECT-ORIENTED LANGUAGES

There are many languages which support object-oriented programming. Table 1.1 and 1.2 describe the OOP languages and their features.

Following object-oriented languages are widely accepted by the programmer:

- C++
- Smalltalk
- Charm ++
- Java

**Table 1.1** Properties of pure OOP and object based languages

	Pure Object-Oriented Languages				Object-Based Languages
<i>Properties</i>	<i>Java</i>	<i>Simula</i>	<i>Smalltalk</i>	<i>Eiffel</i>	<i>Ada</i>
Encapsulation	✓	✓	✓	✓	✓
Inheritance	✓	✓	✓	✓	No
Multiple Inheritance	✗	✗	✓	✓	No
Polymorphism	✓	✓	✓	✓	✓
Binding (Early and Late)	Both	Both	Late Binding	Early Binding	Early Binding
Genericity	✗	✗	✗	✓	✓
Class Libraries	✓	✓	✓	✓	Few
Garbage collection	✓	✓	✓	✓	✗
Persistence	✓	✗	Promised	Less	Same as 3GL
Concurrency	✓	✓	Less	Promised	Hard

**Table 1.2** Properties of extended traditional languages

	Extended Traditional Languages				
<i>Properties</i>	<i>Objective c</i>	<i>C++</i>	<i>Charm ++</i>	<i>Objective Pascal</i>	<i>Turbo Pascal</i>
Encapsulation	✓	✓	✓	✓	✓
Inheritance	✓	✓	✓	✓	✓
Multiple Inheritance	✓	✓	✓	---	---
Polymorphism	✓	✓	✓	✓	✓
Binding (Early and Late)	Both	Both	Both	Late	Early
Genericity	✗	✓	✓	✗	✗
Class Libraries	✓	✓	✓	✓	✓
Garbage collection	✓	✗	✗	✓	✓
Persistence	✗	✗	✗	✗	✗
Concurrency	Poor	Poor	✓	✗	✗

**SMALLTALK** Smalltalk is purely an object-oriented programming language. C++ makes few compromises to ensure quick performance and small code size. Smalltalk uses run-time binding. Smalltalk programs are considered to be faster than C++ and needs longer time to learn. Smalltalk programs are written using Smalltalk browser. Smalltalk uses dynamic objects and memory is allocated from free store. It also provides automatic garbage collection and memory is released when object is no longer in use.

**CHARM++** Charm ++ is also an object-oriented programming language. It is portative. The language provides features such as inheritance, strict type checking, overloading, and reusability. It is designed in order to work efficiently with different parallel systems, together with shared memory systems and networking.

**JAVA** Java was developed by Patrick Naughton, James Gosling, Chris Warth, Mike Sheridan, and Ed Frank at Sun Microsystems. Java is an object-oriented programming language and supports maximum OOP characteristics. Its statement structure is like C and C++, but it is easier than C++. Java excludes few confusing and unsafe features of C and C++, for example, pointers.

Java allows client/server programming and is used for Internet programming. The Java programs are downloaded by the client machines and executed on different types of hardware. This portability is achieved by translation of Java program to machine code using compiler and interpreter.

The Java compiler converts the source program to JVM (Java virtual machine). The JVM is a dummy CPU. The compiler Java program is known as byte code. The Java interpreter translates byte code into object code. Compiler and interpreter do the conversion of Java program to object code.

## 1.10 USAGE OF OOP

Object-oriented technology is changing the style of software engineers to think, analyze, plan, and implement the software. The software developed using OOP technology is more efficient and easy to update. OOP languages have standard class library. The users can reuse the standard class library in their program. Thus, it saves lot of time and coding work.

The most popular application of object-oriented programming is windows. There are several windowing software based on OOP technology. Following are the areas for which OOP is considered:

- (1) Object-Oriented DBMS
- (2) Office automation software
- (3) AI and expert systems
- (4) CAD/CAM software
- (5) Network programming
- (6) System software

## 1.11 USAGE OF C++

C++ is a flexible language. Lengthy programs can be easily controlled by the use of C++. It creates hierarchy -associated objects and libraries that can be useful to other programmers. C++ helps the programmer to write bug-free program, which are easy to maintain.

### SUMMARY

- (1) C++ is an object-oriented programming language invented by Bjarne Stroustrup at AT&T Bell Laboratories.
- (2) The recognized council working under the procedure of the American National Standard Institute (ANSI) has made an international standard for C++.
- (3) The disadvantages with conventional programming language is that the program written in these languages consists of a sequence of instructions that tells the compiler or interpreter to perform a given task. When program code is large it becomes inconvenient to manage it.
- (4) The prime factor in the development of object-oriented programming approach is to remove some of the shortcoming of the procedure oriented languages.
- (5) In monolithic programming languages such as BASIC and ASSEMBLY, the declared data variables are global and the statements are written in sequence.
- (6) In procedural programming languages such as FORTRAN and COBOL, programs are divided into a number of segments known as subprograms. Thus it also focuses on functions apart from data.
- (7) Large size programs are developed in structured programming such as PASCAL and C. Programs are divided in multiple submodules and procedures.
- (8) OOP acts with data as a critical component in the program development and does not let data to flow freely around the systems.
- (9) Object: Objects are primary run-time entities in an object-oriented programming. They may stand for a thing that has a specific application.
- (10) Class: A class is a grouping of objects having identical properties, common behavior, and shared relationship. The entire group of data and code of an object can be built as a user-defined data type using class.
- (11) Method: An operation required for an object or entity when coded in a class is called a method.
- (12) Data Abstraction: Abstraction directs to the procedure of representing essential features without including the background details.
- (13) Encapsulation: C++ supports the features of encapsulation using classes. The packing of data and functions into a single component is known as encapsulation.

(14) Inheritance: Inheritance is the method by which objects of one class get the properties of objects of another class. In object-oriented programming, inheritance provides the thought of reusability.

(15) Polymorphism: Polymorphism makes possible the same functions to act differently on different classes. It is an important feature of OOP concept.

(16) Object-oriented technology allows reusability of the classes by extending them to other classes using inheritance.

(17) C++, Smalltalk, Eiffel and Java are widely used OOP languages.

(18) Object-oriented technology is changing the style of software engineers to think, analyze, plan, and implement the software. The software developed using OOP technology is more efficient and easy to update.

(19) C++ is a flexible language. Lengthy programs can be easily controlled by C++.

## EXERCISES

### [A] Answer the following questions.

- (1) What is object-oriented programming?
- (2) Explain the key concepts of OOP.
- (3) What is ANSI standard?
- (4) What are the disadvantages of conventional programming languages?
- (5) Explain the evolution of C++.
- (6) List the names of popular OOP languages.
- (7) List the unique features of an OOP paradigm.
- (8) What is an object and a class?
- (9) Compare and contrast OOP languages with procedure oriented languages.
- (10) Mention the types of relationships between two classes.
- (11) What is structure oriented programming? Discuss its pros and cons.
- (12) How is global data shared in procedural programming?
- (13) Describe any two object-oriented programming languages.
- (14) What are the differences between Java and C++?
- (15) Why pointers are passed by in Java?
- (16) Mention the advantages of OOP languages.
- (17) What do you mean by message passing?
- (18) Distinguish between inheritance and delegation.
- (19) Can we define member functions in private section?
- (20) Can we declare data member in public section?

### [B] Answer the following by selecting the appropriate option.

- (1) C++ language was invented by  
(a) Bjarne Stroustrup

- (b) Dennis Ritche
  - (c) Ken Thompson
  - (d) none of the above
- (2) The languages COBOL and BASIC are commonly known as
- (a) procedure oriented languages
  - (b) object-oriented languages
  - (c) low level languages
  - (d) none of the above
- (3) The packing of data and functions into a single component is known as
- (a) encapsulation
  - (b) polymorphism
  - (c) abstraction
  - (d) none of the above
- (4) The method by which objects of one class get the properties of objects of another class is known as
- (a) inheritance
  - (b) encapsulation
  - (c) abstraction
  - (d) none of the above
- (5) The mechanism that allows same functions to act differently on different classes is known as
- (a) polymorphism
  - (b) encapsulation
  - (c) inheritance
  - (d) none of the above
- (6) Object-oriented programming is popular because
- (a) user can define user-defined data types
  - (b) programming statements are easy
  - (c) it is easy to find bugs and errors
  - (d) all of the above
- (7) The existing class can be reused by
- (a) inheritance
  - (b) polymorphism
  - (c) dynamic binding
  - (d) abstraction
- (8) Composition of objects in a class is known as
- (a) delegation
  - (b) inheritance
  - (c) polymorphism
  - (d) none of the above

# 2

CHAPTER

## Input and Output in C++

C  
H  
A  
P  
T  
E  
R  
O  
U

- [2.1 Introduction](#)
- [2.2 Streams in C++](#)
- [2.3 Pre-Defined Streams](#)
- [2.4 Buffering](#)
- [2.5 Stream Classes](#)
- [2.6 Formatted and Unformatted Data](#)

- [2.7 Unformatted Console I/O Operations](#)
- [2.8 Typecasting with cout Statement](#)
- [2.9 Member Functions of Istream Class](#)
- [2.10 Formatted Console I/O Operations](#)
- [2.11 BIT Fields](#)
- [2.12 Flags without BIT Field](#)
- [2.13 Manipulators](#)
- [2.14 User-Defined Manipulators](#)
- [2.15 Manipulator with one Parameter](#)
- [2.16 Manipulators with Multiple Parameters](#)
- [2.17 Custom Built I/O Objects](#)

## 2.1 INTRODUCTION

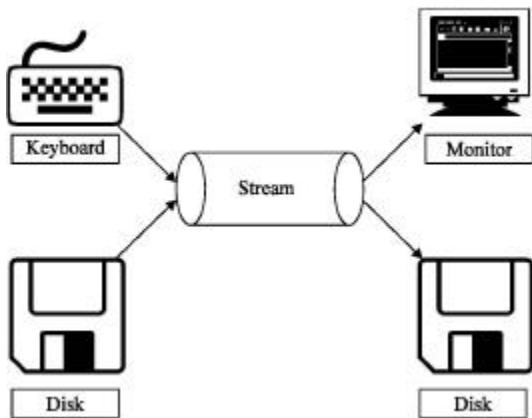
Applications generally involve reading a large amount of data from input devices and sending them to the output devices. Hence to control such operations every language provides a set of in-built functions. C++ supports all Input /Output functions of C. C++ also has library functions. A library is a set of .obj files, which is linked to the program and gives additional support to its functions. A programmer can use the library functions in the programs. The library is also called as *iostream* library. The difference between C and C++ I/O functions is that C functions do not support object-oriented platform whereas C++ supports it.

## 2.2 STREAMS IN C++

C++ supports a number of Input/Output (I/O) operations to read and write operations. These C++ I/O functions make it possible for the user to work with different types of devices such as keyboard, disk, tape drivers etc. The stream is an intermediary between I/O devices and the user. The standard C++ library contains the I/O stream functions. The I/O functions are part

of the standard library that provides portability to the language itself. A library is nothing but a set of .obj files. It is connected to the user's program.

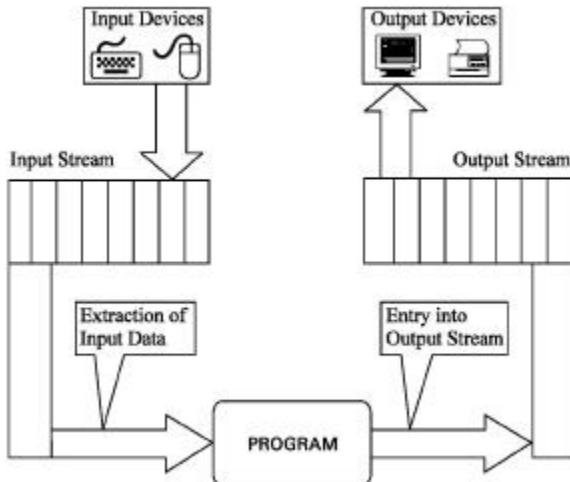
Stream is flow of data in bytes in sequence. If data is received from input devices in sequence then it is called as *source stream* and when the data is passed to output devices then it is called as *destination stream*. It is also referred as encapsulation through streams. The data is received from keyboard or disk and can be passed on to monitor or to the disk. Figure 2.1 describes the concept of stream with input and output devices.



**Fig.2.1** Streams and I/O devices

Data in source stream can be used as input data by the program. So, the source stream is also called as input stream. The destination stream that collects output data from the program is known as output stream. The mechanism of input and output stream is illustrated in Figure 2.2.

As mentioned earlier, the stream is an intermediately between I/O devices and the user. The input stream pulls the data from keyboard or storage devices such as hard disk, floppy disk etc. The data present in output stream is passed on to the output devices such as monitor or printer according to the user's choice.



**Fig.2.2** C++ Input and output streams

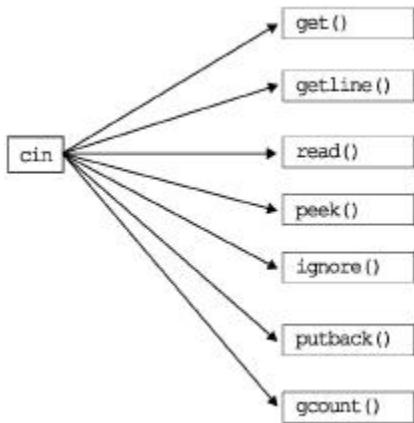
### 2.3 PRE-DEFINED STREAMS

C++ has a number of pre-defined streams. These pre-defined streams are also called as standard I/O objects. These streams are automatically activated when the program execution starts. Table 2.1 describes the pre-defined streams.

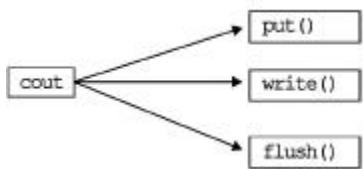
**Table 2.1** Predefined C++ stream or I/O global objects

cin	Standard input, usually keyboards, corresponding to <code>stdin</code> in C. It handles input devices usually from keyboard. The <u>Figure 2.3</u> indicates frequently used members with <code>cin</code> object
cout	Standard output, usually screen, corresponding to <code>stdout</code> in C. It passes data to such as monitor and printers. Thus, it controls output. The <u>Figure 2.4</u> indicates functions with <code>cout</code> object.
clog	A fully buffered version of <code>cerr</code> (no C equivalent). It controls error messages that from buffer to the standard error device.
cerr	Standard error output, usually screen, corresponding to <code>stderr</code> in C. It controls output data. It catches the errors and passes to standard error device monitor.

A simple program is illustrated using above I/O objects as follows.



**Fig.2.3** Frequently used unformatted input functions with `cin` object



**Fig.2.4** Frequently used unformatted output functions with `cout` object

## 2.1 Write a program to display message using pre-defined objects.

```

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

void main( )
{
    clrscr( );
    cout << "\nSTREAMS";
    cerr << "\nSTREAMS";
    clog << "\nSTREAMS";
}

```

### OUTPUT

STREAMS

STREAMS

STREAMS

**Explanation:** In this program the pre-defined object `cout`, `cerr`, and `clog` are used to display messages “STREAMS” on the screen.

## 2.4 BUFFERING

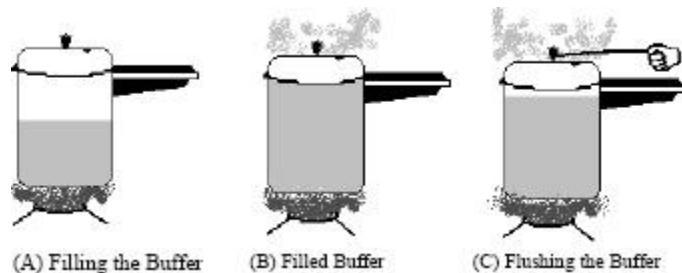
**Buffer** It is a block of memory used to hold data temporarily. It is always located between a peripheral device and faster computer. Buffers are used

to pass data between computer and devices. The buffer increases the performance of the computer by allowing read/write operation in larger chunks. For example, if the size of buffer is  $N$ , the buffer can hold  $N$  number of bytes of data.

Writing data to the disk is very costly. The read/write operation with disk takes a long time. While these operations are carried out the execution of the program will be slow for few seconds. If the program involves a lot of read and write operations, the program execution speed will be slower. To avoid this problem the stream provides buffer. Buffer holds the data temporarily. The input data is passed on to the stream buffer. The data is not written to the disk immediately till the buffer fills. When the buffer is completely filled the data is written on to the disk. This example is related with the example shown in [Figure 2.5](#).

In [Figure 2.5 \(a\)](#) the level of the steam in the pressure cooker is increasing. The steam is not full, hence the whistle will not blow, and the steam will not be released from the pressure cooker.

In [Figure 2.5 \(b\)](#) the pressure cooker is completely filled. The level of the steam reaches to the top. Due to high-pressure the whistle blows automatically and the steam is released from the pressure cooker.?



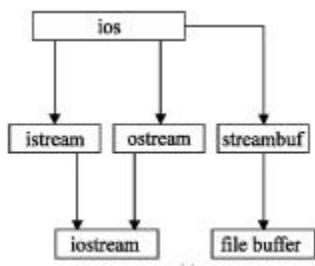
**Fig.2.5** Working of buffer

After the steam has been released the whistle falls down. Even when the cooker is not completely filled with steam, the steam can be released manually by pulling up the whistle. It is just like flushing the buffer even when the buffer is not completely filled.

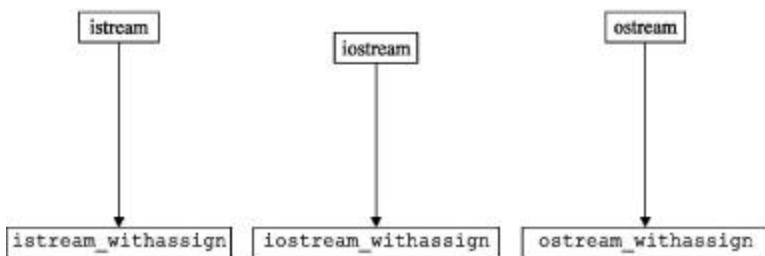
## 2.5 STREAM CLASSES

C++ streams are based on class and object theory. C++ has a number of stream classes that are used to work with console and file operations. These classes are known as stream classes. [Figure 2.6](#) indicates the stream classes. All these classes are declared in the header file `iostream.h`. The

file `iostream.h` must be included in the program if we are using functions of these classes.



**Fig. 2.6 (a)** Stream classes



**Fig. 2.6 (b)** Hierarchy of stream classes

As described in [Figure 2.6 \(a\)](#) the classes `istream` and `ostream` are derived classes of base class `ios`. The `ios` class contains member variable object `streambuf`. The `streambuf` places the buffer. The member function of `streambuf` class handles the buffer by providing the facilities to flush, clear and pour the buffer. The class `iostream` is derived from the classes `istream` and `ostream` by using multiple inheritance.

The `ios` class is a virtual class and this is to avoid ambiguity that frequently appears in multiple inheritance. The chapter on **Inheritance** describes multiple inheritance and virtual classes.

The `ios` class has an ability to handle formatted and unformatted I/O operations. The `istream` class gives support to both formatted and unformatted data. The `ostream` classes handle the formatting of output of data. The `iostream` contains functioning of both `istream` and `ostream` classes. The classes `istream_withasssign`, `ostream_withasssign`, and `iostream_withasssign` append appropriate assignment operators as shown in [Figure 2.5 \(b\)](#).

## 2.6 FORMATTED AND UNFORMATTED DATA

Formatting means representation of data with different settings as per the requirement of the user. The various settings that can be done are number format, field width, decimal points etc.

The data accepted or printed with default setting by the I/O function of the language is known as unformatted data. For example, when the `cin` statement is executed it asks for a number. The user enters a number in decimal. For entering decimal number or displaying the number in decimal using `cout` statement the user won't need to apply any external setting. By default the I/O function represents number in decimal format. Data handled in such a way is called as unformatted data.

If the user needs to accept or display data in hexa-decimal format, manipulators with I/O functions are used. The data obtained or represented with these manipulators are known as formatted data. For example, if the user wants to display the data in hexa-decimal format then the manipulator can be used as follows.

```
cout<<hex<<15;
```

The above statement converts decimal 15 to hexadecimal F.

## 2.7 UNFORMATTED CONSOLE I/O OPERATIONS

### INPUT AND OUTPUT STREAMS

The input stream uses `cin` object to read data and the output stream uses `cout` object to display data on the screen. The `cin` and `cout` are predefined streams for input and output of data. The data type is identified by these functions using operator overloading of the operators `<<` (**insertion operator**) and `>>` (**extraction operator**). The operator `<<` is overloaded in the `ostream class` and the operator `>>` is overloaded in `istream class`. Figure 2.7 shows the flow of input and output stream.

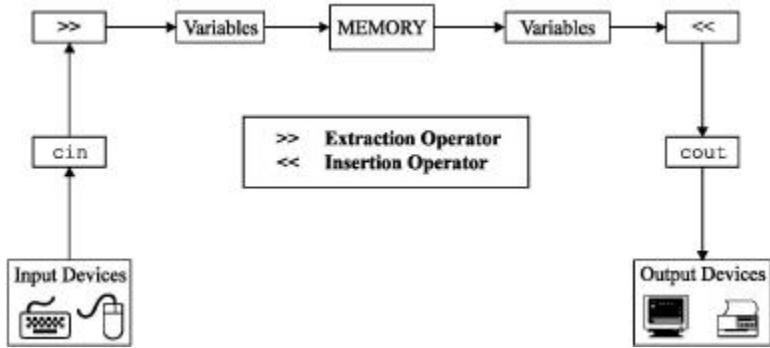
**INPUT STREAM** The input stream does read operation through keyboard. It uses `cin` as object. The `cin` statement uses `>>` (extraction operator) before variable name. The `cin` statement is used to read data through the input device. Its syntax and example are as follows.

Syntax:

```
cin >>variable
```

Example:

```
int v1;  
float v2;  
char v3;  
cin >> v1 >> v2 >> v3.... . >>vn;
```



**Fig.2.7** Working of `cin` and `cout` statements

Where `v1`, `v2`, and `v3` are variable names. The response of the user to this statement would be as per given below.

2 5.4 A // input data

If the user enters data in the manner 2 5.4A, the operator will assign 2 to `v1`, 5.4 to `v2` and a to `v3`. If the entered data is more than the variable, it remains in the input stream. While entering string, blank spaces are not allowed. More than one variable can be used in `cin` statement to input data. Such operations are known as *cascaded* input operations.

Example:

`cin>>v1>>v3`

where `v1` and `v2` are variables.

The operator `>>` accepts the data and assigns it to the memory location of the variables. Each variable requires `>>` operator. Both these statements should not be included in the bracket. The entered data is separated by space, tab, or enter. Like `scanf( )` statement `cin` does not require control strings like `%d` for integer, `%f` for float and so on.

*More examples*

```

int weight;
cin>>weight      // Reads integer value

float height;
cin>>height;       // Reads float value

double volume;
cin>>volume;      // Reads double value

char result[10];
cin>>result;      // Reads char string

```

**OUTPUT STREAM** The output stream manages output of the stream i.e., displays contents of variables on the screen. It uses `<<` insertion operator

before variable name. It uses the `cout` object to perform console write operation. The syntax and example of `cout` statements are as follows:

Syntax:

```
cout<<variable
```

Example

```
cout <<v1 <<v2 <<v3...<<vn;
```

where `v1`, `v2`, and `v3` are variables. The above statement displays the contents of these variables on the screen. The syntax rules are same as `cin`. Here, the `cout` statement uses the insertion operator `<<`.

The `cout` statement does not use the format specification like `%d`, `%f` as used in C, and so on. The `cout` statement allows us to use all ‘C’ escape sequences like ‘`\n`’, ‘`\t`’, and so on.

*More examples*

```
int weight;
cout<<weight;      // Displays integer value

float height;
cout<<height;      // Displays float value

double volume;
cout<<volume;      // Displays double value

char result[10];
cout<<result;      // Reads char string
```

The box given below illustrates comparative programs on `cout` statements.

PROGRAM 2.2	PROGRAM 2.3	PROGRAM 2.4
<pre># include &lt;iostream.h&gt; void main( ) { cout &lt;&lt;" C PLUS PLUS"; }</pre>	<pre># include &lt;iostream.h&gt; void main( ) { char *n="Hello"; cout &lt;&lt;n; }</pre>	<pre># include &lt;iostream.h&gt; void main( ) { int x=10; float f=3.14; cout &lt;&lt;x; cout &lt;&lt;"\t"; cout &lt;&lt;f; }</pre>

OUTPUT C PLUS PLUS	OUTPUT Hello	OUTPUT 10 3.14
In this program the <code>cout</code> statement displays the message “ <b>C PLUS PLUS</b> ” on the screen.	In this program the string is first assigned to character pointer <code>n</code> . The <code>cout</code> statement	In this program <code>integ</code> values are assigned to <code>x</code> and <code>f</code> . The <code>cout</code> statement displays the values of <code>x</code> and <code>f</code> separated by a tab.

	displays the contents of variable n.	of x and f. “\t” is escape sequence to insert tab.
--	--------------------------------------	--

## 2.5 Write a program to accept string through the keyboard and display it on the screen. Use cin and cout statements.

```
# include <iostream.h>
# include <conio.h>
main( )
{
    char name[15];
    clrscr( );
    cout <<"Enter Your Name :";
    cin >>name;
    cout <<"Your name is "<<name;
    return 0;
}
```

### OUTPUT

Enter Your Name :Amit

Your name is Amit

**Explanation:** In the above program, cout statement displays given string on the screen. It is similar to printf( ) statement. The cin statement reads data through the keyboard. It is similar to scanf( ) statement.

## 2.6 Write a program to read two integers and display them.

Use cin and cout statements.

```
# include <iostream.h>
# include <conio.h>
main( )
{
    int num,num1;
    clrscr( );
    cout <<"Enter Two numbers : ";
    cin >>num>>num1;
    cout <<"Entered Numbers are : ";
    cout <<num<<"\t"<<num1;
    return 0;
}
```

### OUTPUT:

Enter Two numbers : 8 9

Entered Numbers are : 8 9

**Explanation:** In the above program, the cin statement reads two integers in variables num and num1. The cout statement displays the read numbers. The escape sequence ‘\t’ is used to insert tab between two numbers.

## 2.7 Write a program to display data using cout statements.

```
# include <iostream.h>
# include <conio.h>
void main( )
{
    clrscr( );
    cout <<"======" ;
    cout<<endl;
    cout <<"Hello";      // prints hello
    cout<<"\n";
    cout <<123;        // prints 123
    cout <<endl<<3.145;   // prints 3.145
    cout <<endl<<"NUMBER : "<<"\t"<<452; // prints string and value
    cout <<"\n==== The end =====" ;
}
```

### OUTPUT

```
=====
```

Hello  
123  
3.145  
NUMBER : 452

```
==== The end =====
```

**Explanation:** Explanation of each of the cout statements is as follows.

- (a) cout <<"======" – Displays line on the screen.
- (b) cout<<endl - Breaks a line.
- (c) cout <<"Hello" – Display string “Hello” on the screen.
- (d) cout<<"\n" – Breaks a line.
- (e) cout <<123; – Displays integer 123.
- (f) cout <<endl<<3.145 – Breaks a line and display float number 3.145.
- (g) cout <<endl<<"NUMBER : "<<"\t"<<452 – Breaks a line, display string “number”, insert a tab and integer number 452.
- (h) cout <<"\n==== The end =====" – Display the line and string.

## 2.8 Write a program to display integer, float, char, and string using cout statement.

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

void main( )
{
    clrscr( );
    int x=5;
    float y=5.2;
```

```

    char z='z';
    char city[]="NANDED";
    cout <<"x = "<<x <<" y ="<<y <<" z = "<<z <<endl;
    cout <<"City = "<<city;
}

```

## **OUTPUT**

**X = 5 y = 5.2 z = z**

**City = NANDED**

**Explanation:** In the above program, the statement `cout <<"x = "<<x <<" y ="<<y <<" z = "<<z <<endl` displays values of  $x$ ,  $y$ , and  $z$ . The statement `cout <<"City = "<<city` displays the contents of character array `city`.

## **2.9 Write a program to input integer, float, char, and string using `cin` statement and display using `cout` statement.**

```

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

void main( )
{
    clrscr( );
    int x;
    float y;
    char z;
    char city[15];
    cout <<"\n Enter integer, float and char ";
    cin >>x>>y>>z;
    cout <<"\n Enter a string : ";
    cin >>city;
    cout <<"x = "<<x <<" y ="<<y <<" z = "<<z <<endl;
    cout <<"City = "<<city;
}

```

## **OUTPUT**

**Enter integer, float, and char 12 1.2 H**

**Enter a string : NAGPUR**

**X = 12 y =1.2 z = H**

**City = NAGPUR**

**Explanation:** Explanation of the program is as follows:

- (a) `cout <<"\n Enter integer, float and char "`—Prompts message “Enter integer, float and char”
- (b) `cin >>x>>y>>z`—Accepts integer, float and char and stores in  $x$ ,  $y$ , and  $z$ .
- (c) `cout <<"\n Enter a string: "`—Displays message “Enter a string : ”.

- (d) `cin >>city`—Reads string through the keyboard and stores in the array `city[15]`.  
(e) `cout <<"City = "<<city;` displays contents of the array `city`.

## 2.8 TYPECASTING WITH COUT STATEMENT

Typecasting refers to the conversion of data from one basic type to another by applying external use of data type keywords. The description of typecasting is explained in [Chapter 3 \(C++ declarations\)](#). Programs on typecasting are as follows.

### 2.10 Write a program to use different formats of typecasting and display the converted values.

```
# include <iostream.h>
# include <conio.h>
void main( )
{
    int a=66;
    float f=2.5;
    double d=85.22;
    char c='K';
    clrscr( );
    cout <<"\n int in char format : "<<(char)a;
    cout <<"\n float in int format : "<<(int)f;
    cout <<"\n double in char format : "<<(char)d;
    cout <<"\n char in int format : "<<(int)c;
}
```

#### OUTPUT

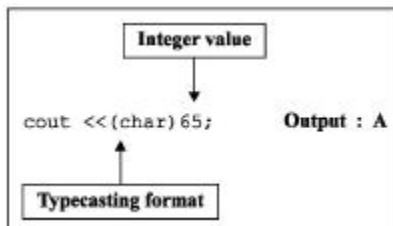
```
int in char format : B
float in int format : 2
double in char format : U
char in int format : 75
```

**Explanation:** In the above program, the variables of `int`, `float`, `double` and `char` type are declared and initialized. The variable `a` is initialized with 66, `f` with 2.5, `d` with 85.22 and `c` with character ‘K’. The first `cout` statement converts integer value to corresponding character according to ASCII character set and the character B is displayed. The second `cout` statement converts float value to integer. The value displayed is 2 and not 2.5. When type cast format (`int`) is used, the decimal portion of float value is removed and only integer part is considered. In the third statement the double value is converted to character. The number 85.22 are converted to integer and then character. The `char` data type is more or less similar to `int` data type except one difference that

the `char` data type ranges from 128 to 127, which requires one byte in the memory.

The last statement converts character to `int`. The value of ‘K’ is 75 when printed as an integer. The format (`int`) converts `char` to `int`.

In Figure 2.8, 65 is an integer and it is converted to character A by using typecasting format (`char`). Table 2.2 describes various typecasting formats and their output results.



**Fig.2.8** Typecasting integer

**Table 2.2** Typecasting formats

Typecasting formats	Outputs	Conversion
<code>cout &lt;&lt; (char) 65;</code>	A	int to char
<code>cout &lt;&lt; (int) 'A' ;</code>	65	char to int
<code>cout &lt;&lt; (int) 5.22;</code>	5	float to int
<code>cout &lt;&lt; (char) 78.33;</code>	N	float to char
<code>cout&lt;&lt;(double)123445338.33;</code>	1.234453e+08	float to double
<code>cout&lt;&lt; (unsigned) -1;</code>	65535	signed to uns

## 2.11 Write a program to display data using typecasting.

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

void main( )
{
    clrscr( );
    int x=77;
    float y=5.1252;
```

```

    char z='A';
    char city[15];
    cout <<" x = "<<(char)x<<endl;
    cout <<" y = "<<(int)y <<endl;
    cout <<" z = "<<(int)z;
}

```

## **OUTPUT**

**x = M**

**y = 5**

**z = 65**

**Explanation:** Consider the following statements:

`int x=77`—Declares integer variable `x` and initializes it with `77`.

`float y=5.1252`—Declares `float` variable `y` and initialize it with `5.1252`.

`char z = 'A';`—Declares character variable `z` and initializes it with character '`A`'.

To display the contents on the screen following statements are used:

`cout <<" x = "<<(char)x<<endl`—Variable `x` is an integer but value displayed will be '`M`' because the statement `(char)` converts integer to corresponding **ASCII** character.

`cout <<" y = "<<(int)y <<endl`—The variable `y` is a float but before printing the value, `5.1252` is converted to integer and the output will be `5`.

`cout <<" z = "<<(int)z`—The variable `z` is of character type. The character value is converted to integer and the output displayed will be `65`.

## **2.12 Write a program to display A to Z alphabets using ASCII values.**

```

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

    for (j=65;j<91;j++)
    cout <<(char)j<<" ";
    cout <<endl;

    for (j=65;j<91;j++)
    printf ("%c ",j);
    return 0;
}

```

## **OUTPUT**

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

**Explanation:** In the above program, A to Z alphabets are displayed using `cout( )` and `printf( )` statements. In `cout( )` statement, typecasting is done before printing. The integer is converted to corresponding `char` type symbol and displayed. In the `printf( )` statement, the control string `%c` performs this task. The quotation mark (" ") inserts space between two successive characters.

## 2.13 Write a program to display addresses of variables in hexadecimal and unsigned integer formats.

```
# include <iostream.h>
# include <conio.h>
void main( )
{
    clrscr( );
    int x=77;
    float y=5.1252;
    int z=78;

    cout <<" Address of x = "<<&x<<endl;
    cout <<" Address of y ="<<&y<<endl;
    cout <<" Address of z ="<<(unsigned)&z <<endl;
}
```

### OUTPUT

Address of x = 0x887ffff4

Address of y = 0x887ffff0

Address of z = 65518

**Explanation:** The `cout` statement displays address of variables in hexadecimal format. The typecasting syntax `(unsigned)` converts hexadecimal to unsigned integer (decimal). The output shows addresses in both hexadecimal and unsigned integer (decimal) formats.

The `&` (ampersand) operator is used to display address of the variable. The address operator is preceded by the variable name. The address is always represented in unsigned integer. The `cout` statement displays the address in hexadecimal format. To convert the hexa-decimal address to unsigned integer, type casting is used.

## 2.14 Write a program to display string using different syntaxes using operator \*, & with cout statement.

```
# include <iostream.h>
# include <conio.h>
void main( )
```

```

{
clrscr( );
char *name="C Plus Plus";
cout<<*&(name)<<"\n";
cout <<&(*name)<<"\n";
cout<<*&name<<"\n";
}

```

## **OUTPUT**

**C Plus Plus**

**C Plus Plus**

**C Plus Plus**

**Explanation:** In the first statement, the operator \* and & are used one after another and the variable name is inside parenthesis. In the second statement, operator & is outside and operator \* and variable name are inside the parenthesis. In the third statement there is no use of parenthesis. All the three statements display the output “C Plus Plus.”

## **DIFFERENCE IN USING C AND C++ I/O FUNCTIONS**

The `printf( )` and `scanf( )` of C language need format string. For example, to read and display an integer the `scanf( )` and `printf( )` statements can be written as follows.

```

int x;
scanf("%d", &x)
printf("%d", x);

```

In the above statements `%d` is used to tell the I/O functions to treat the data as integer.

If the integer `x` is changed to long integer the programmer needs to change every occurrence of `%d` in the program with `%ld`.

The C++ statement reads and displays the same data as follows.

```

int x;
cin>>x;
cout<<x;

```

Here, the `cin` and `cout` statements do not require any format string. If the type of `x` is changed to long integer, the user won't need to specify the type of data or any correction in the statement. The `cin` and `cout` statement identifies the data type. The format of `cin` and `cout` statements is same for all types of variables.

`get( )` and `put( )` functions  
`get( )` function

The single character input and output operations in C++ can be done with the help of `put()` and `get()` functions. The class `istream` and `ostream` provides the two member functions `put()` and `get()`. The `get()` is used to read a character and `put()` is used to display the character on the screen.

The `get()` function has two syntaxes.

- (a) `get(char*);`
- (b) `get(void);`

If syntax (a) is used, the `get()` function assigns the read data to its argument, whereas if syntax (b) is used, the `get()` function returns the data read. The data is assigned to the variable present at left-hand side of the assignment operator. These functions are members of I/O stream classes and can be called using object.

`put()` function

The `put()` function is used to display the string on the screen. It is a member of `ostream` class. The syntax of `put()` is given below.

- (a) `cout.put('A');`
- (b) `cout.put(x);`

The statement (a) displays the character ‘A’ on the screen and the statement (b) displays the contents of variable `x` on the screen. If an integer is used as an argument, its corresponding **ASCII** value is displayed. Few examples, are illustrated below.

## 2.15 Write a program to display the character on the screen using `put()` function.

```
# include <iostream.h>
# include <conio.h>
void main( )
{
    clrscr();
    cout.put('C');
    cout.put('+');
    cout.put('+');
}
```

**OUTPUT**

**C++**

**Explanation:** The `cout.put()` statement displays one character at a time on the screen. In this program three characters are displayed on the screen using `cout.put()` statement.

## 2.16 Write a program to use multiple `put()` statements with single `cout` object and display the characters.

```
# include <iostream.h>
# include <conio.h>
void main( )
{
    clrscr( );
    cout.put('C').put('+').put('+');
}
```

## OUTPUT

### C++

**Explanation:** In the above program, the single object `cout` is used followed by sequence of `put()` statement. The `put()` statements are separated by dot operators. In this way multiple statements can be combined.

## 2.17 Write a program to read character using `get()` and display it using `put()`.

```
# include <iostream.h>
# include <conio.h>
void main( )
{
    char ch;
    clrscr( );
    cout <<"\n Enter a character : ";
    cin.get(ch);
    cout <<"\n Entered character was : ";
    cout.put(ch);
}
```

## OUTPUT

Enter a character : C

Entered character was : C

**Explanation:** In the above program, character variable `ch` is declared. The first `cout` statement displays message “Enter a character:” on the screen. The `cin.get()` function activates input stream and character entered by the user is stored in the variable `ch`. The `cout.put()` statement displays the character on the screen.

## 2.18 Write a program to read characters using sequence of `get()` statements and display read characters using sequence of `put()` statement.

```
# include <iostream.h>
# include <conio.h>
void main( )
{
    char ch[3];
```

```

    clrscr( );
    cout <<"\n Enter characters : ";
    cin.get(ch[0]).get(ch[1]).get(ch[2]);
    cout <<"\n Characters Entered : ";
    cout.put(ch[0]).put(ch[1]).put(ch[2]);
}

```

## **OUTPUT**

**Enter characters : cpp**

**Characters Entered : cpp**

**Explanation:** In the above program, a character array ch [ 3 ] is declared. The sequence of get ( ) and put ( ) functions are used to read and display the characters. The get ( ) function reads characters and stores in array ch [ 3 ] . The put ( ) function displays the same on the screen.

## **2.19 Write a program to read and display the string. Use get ( ) and put ( ) functions.**

```

# include <iostream.h>
# include <conio.h>
# include <stdio.h>
main( )
{
    clrscr( );
    char j=0;
    char x[20];
    cout <<"\n Enter a string : ";

    while(x[j]!='\n')
        cin.get(x[j]);
    j=0;
    cout <<"\n Entered string : ";
    while(x[j]!='\n')
        cout.put(x[j]);
    return 0 ;
}

```

## **OUTPUT**

**Enter a string : Programming**

**Entered string : Programming**

**Explanation:** In the above program, the character array x [ ] is declared. The first while loop reads characters using cin.get ( ) function through the keyboard. When user presses enter key, the while loop terminates. The second while loop displays the characters read using function cout.put ( ).

**getline( ), read( ) and write( ) function**  
**getline( )**

The `getline( )` and `write( )` functions are useful in string input and output. The `getline( )` function reads the string including white space. The `cin ( )` function does not allow the string to enter with blank spaces. The input reading is terminated when user presses the enter key. The new line character is accepted but not saved and replaced with the null character. The object `cin` calls the function as follows.

```
cin.getline(variable, size);
```

where the variable name may be any character, array name and size is the size of the array.

`read( )` : The `read( )` function is also used to read text through the keyboard. The syntax of `read` is given below.

```
cin.read(variable, size);
```

where the variable name may be any character, array name and size is the size of the array.

When we use `read` statement it is necessary to enter character equal to the number of size specified. The `getline( )` statement terminates the accepting data when enter is pressed where as the `read( )` continues to read characters till the number of characters entered are equal to the size specified.

```
write( )
```

The `write( )` function is used to display the string on the screen. Its format is the same as `getline( )` but the function is exactly opposite. The syntax is given below.

```
cout.write (variable, size);
```

where variable name is a character type and size is the size of the character arrays. The `cout. write ( )` statement displays only specified number of characters given in the second argument, though actual sting may be more in length. If the size of array is larger than the actual string length then the size argument contains the actual size of the array. In this case the `getline( )` displays blank spaces for remaining unfilled elements.

The following program illustrates both the functions:

## 2.20 Write a program to read characters using `read( )` statement.

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

void main( )
{
    clrscr( );
    char name[20];
    cout<<"\n Enter a string : ";
    cin.read(name,20);
```

```
    cout<<endl;
    cout.write(name, 20);
}
```

## OUTPUT

Enter a string : abcdefg

Hijkl

Mnopq

Abcdefg

Hijkl

Mnopq

**Explanation:** In the above program, the input data and output data can be seen in three lines. This is because when we enter data, enter keys are pressed. The same effect is produced while displaying the data. The enter key is stored as '\n'.

## 2.21 Write a program to display string using cout.write( ) statements.

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

void main( )
{
    clrscr( );
    cout.write ("INDIA", 6);
    cout.write ("IS", 3);
    cout.write ("GREAT", 5);
}
```

## OUTPUT

INDIA IS GREAT

**Explanation:** In this program, the first statement displays "INDIA" followed by one blank space. This is because the argument value is greater by one than actual string length.

Similarly, the second statement displays "IS" followed by one blank space. The reason is same. The last statement displays "GREAT." Here, the argument value and string lengths are same and no blank spaces are displayed.

In case the argument value is lesser than the actual string length, the complete string will not be displayed. The number of characters of the string displayed will be up to the given argument.

## 2.22 Write a program to show the effect if less argument is given than actual string length in the cout.write( ) statement.

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

```
void main( )
{
    clrscr( );
    cout.write ("SUNDAY", 3);
}
```

## OUTPUT

SUN

**Explanation:** In the above program, the `cout.write()` will not display the complete string. The statement displays the characters according to the value of second argument. The value of second argument is three. Hence, instead of 6 characters only three characters are displayed and remaining characters are skipped.

### 2.23 Write a program to read string using `getline()` function and display it using `write()` statement.

```
# include <iostream.h>
# include <conio.h>
# include <string.h>
main( )
{
    clrscr( );
    char x[30];
    cout <<"\n Enter any string : ";
    cin.getline(x,30);
    cout <<"\n Entered string : " <<x;
    cout <<"\n Entered string : ";
    cout.write(x,30);
    cout <<"\n Entered string : ";
    cout.write(x,strlen(x));
    return 0 ;
}
```

## OUTPUT

Enter any string : C++ is advanced C

Entered string : C++ is advanced C

Entered string : C++ is advanced C h > &

Entered string : C++ is advanced C

**Explanation:** In the above program, the `x[]` is a character array. The `getline()` function reads string through the keyboard. The `getline()` function accepts the string including spaces. The `cout()` statement displays the string including white spaces. The first `write()` statement displays the string with garbage values because the string length is less than the actual size of the array. In the second `write()` statement, `strlen()` function calculates the length of the string and

length is used as an argument in the `write( )` statement. This statement displays the entered string without any garbage collection.

The `cin( )` statement cannot accept string including spaces. It accepts only single word. The `cout( )` statement displays the string read through `cin( )` and `getline( )` functions i.e., it can display the string with and without blank spaces. The `write( )` statement displays the string according to size specified. It displays the string with and without blank spaces. The `write( )` does not support any escape sequence.

## **2.24 Write a program to display the string using different arguments in `write( )` statement.**

```
# include <iostream.h>
# include <conio.h>
# include <string.h>
main( )
{
    clrscr( );
    char a,x[30];
    cout <<"\n Enter any string : ";
    cin.getline(x,30);
    cout <<"\n Entered string : \n";
    for(a=0;a<=strlen(x);a++)
    {
        cout <<"\n";
        cout.write(x,a);
    }
    return 0 ;
}
```

### **OUTPUT**

**Enter any string : C Plus Plus**

**Entered string :**

C  
C  
C P  
C Pl  
C Plu  
C Plus  
C Plus  
C Plus P  
C Plus Pl  
C Plus Plu  
C Plus Plus

**Explanation:** In the above program, a string is entered using `getline()` function. The `for` loop executes from 0 to the length of the string. The string length is calculated using `strlen()` function in the for loop. In each iteration value of loop, variable `a` is used as an argument in the `write` statement. The `write` statement displays the number of characters according to value of `a`.

## 2.25 Write a program to read two strings through the keyboard.

Concatenate the strings and display the resulting string.

```
# include <iostream.h>
# include <conio.h>
# include <string.h>
main( )
{
    clrscr( );
    char a[15], x[15] ;
    cout <<"\n Enter any string : ";
    cin.getline(a,15);
    cout <<"\n Enter any string : ";
    cin.getline (x,15);
    cout <<"\n The Concatenated strings : ";
    cout.write(a,strlen(a)) .write(" ", 1) .write(x,strlen(x));
    return 0 ;
}
```

### OUTPUT

Enter any string : Good

Enter any string : Morning

The Concatenated strings : Good Morning

**Explanation:** In the above program, two strings are entered in character arrays `x[]` and `a[]`. The successive `write()` statement in one line displays the string one after another. Thus, we can use multiple `write()` statements followed by single `cout` object.

## 2.9 MEMBER FUNCTIONS OF ISTREAM CLASS

The `istream` contains following functions that can be called using `cin` object.

`peek()`: It returns the succeeding characters without extraction.

**Example:** `cin.peek() == '#' ;`

where `cin` is an object and '#' is a symbol to be caught in the stream.

`ignore()`: The member function `ignore()` has two arguments, maximum number of characters to avoid, and the termination character.

**Example:** `cin.ignore(1, '#') ;`

The statement ignores 1 character till '#' character is found.

## 2.26 Write a program to demonstrate the use of peek( ) and ignore( ) functions.

```
# include <iostream.h>
# include <conio.h>
main( )
{
    char c;
    clrscr( );
    cout <<"enter text (press F6 to end:) ";
    while(cin.get(c))
    {
        cout<<c;
        while(cin.peek( )=='#')
        {
            cin.ignore(1,'#');
        }
    }
    return 0;
}
```

### OUTPUT

enter text (press F6 to end :) ABCDEFG###HIJK^Z

ABCDEFGHIJK

**Explanation:** In the above program, the `cin.get( )` function continuously reads characters through the keyboard till the user presses F6. The `cout` statement inside the loop displays the contents of variable `c` on the console. The `cin.peek( )` statement checks the variable `c`. The variable `c` containing '#' is ignored from the stream and not displayed on the screen.

`putback( )` The `putback( )` replaces the given character into the input stream

**Example:** `cin.putback('*');`

`cin` is an object and '\*' is the symbol. The `putback( )` function sends back the given symbol into input stream. It replaces the previous character with the new one specified in the `putback( )` function.

## 2.27 Write a program to demonstrate the use of putback( ) function.

```
# include <iostream.h>
# include <conio.h>
main( )
{
    char c;
    clrscr( );
    cout <<"enter text (press F6 to end : ";
```

```

    while (cin.get(c))
    {
        if (c=='s')
            cin.putback('S');
        cout.put(c);
    }
return 0;
}

```

## **OUTPUT**

**enter text (press F6 to end :)**

**c plus plus^Z**

**c plusS plusS**

**Explanation:** In the above program, the `cin.get()` function continuously reads characters through the keyboard till the user presses F6. The `cout` statement inside the loop displays the contents of variable `c` on the console. The `if` statement checks the contents of variable `c`. If variable `c` contains small letter ‘s’, the `putback()` statement sends capital ‘S’ in the stream. The small ‘s’ is replaced with capital ‘S’. The contents displayed will be with capital ‘S’.

`gcount()`: This function returns the number of unformatted characters extracted from input stream. The last statement should be `get()`, `getline()`, or `read()`.

## **2.28 Write a program to demonstrate the use of `gcount()` function.**

```

# include <iostream.h>
# include <conio.h>
void main( )
{
    char text[20];
    int len;
    clrscr( );
    cout <<"Enter text : ";
    cin.getline(text,20);
    len=cin.gcount( );
    cout<<"The Number of characters extracted are : "<<len;
}

```

## **OUTPUT**

**Enter text : Virtual**

**The Number of characters extracted are : 8**

**Explanation:** In the above program, the `getline()` function reads text through the keyboard. The `gcount()` function returns the number of

characters extracted from stream to variable `len`. It also counts the null character. The `cout` statement displays value of variable `len` on the screen.

**2.29 Write a program to perform the operation with `peek()` and `putback()`.**

```
# include <iostream.h>
# include <conio.h>
main( )
{
    char c;
    clrscr( );
    cout <<"enter text (press F6 to end :)" " ;
    while (cin.get(c))
    {
        if (c==' ')
            cin.putback ('.');
        else
            cout <<c;
        while (cin.peek()=='#') cin.ignore(1,'#');
    }
    return 0;
}
```

## OUTPUT

Enter text (press F6 to end:) One! Two! Three! Four!

One..Two..Three..Four.

**^Z**

**Explanation:** In the above program, the `cin.get()` function reads data through the keyboard using first `while` loop. The `if` condition checks blank spaces in the entered text. If space is found, the `putback()` statement replaces space `dot(.)`. The `putback()` sends the given character in the input stream. The `peek()` also checks the '#' symbol in the entered text. If it is found `ignore()` statement ignores the character and the character won't be displayed on the screen. The program is terminated when key **F6** or **ctrl + z** is pressed.

## 2.10 FORMATTED CONSOLE I/O OPERATIONS

C++ provides various formatted console I/O functions for formatting the output. They are of three types.

- (1) `ios` class function and flags
- (2) Manipulators
- (3) User-defined output functions

The `ios` grants operations common to both input and output. The classes derived from `ios` are (**`istream`, `ostream`, and `iostream`**) special I/O with high-level formatting operations: The `iostream` **class** is automatically loaded in the program by the compiler. Figure 2.6 describes hierarchy of stream classes.

- (a) `istream` performs formatted input.
- (b) `ostream` performs formatted output.
- (c) `iostream` performs formatted input and output.

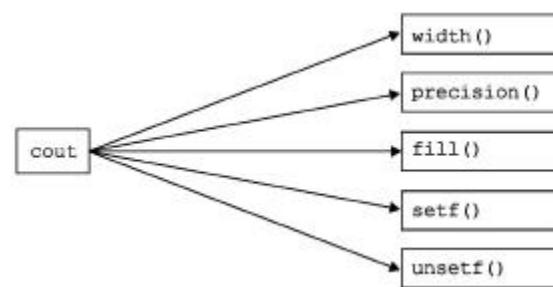
The `streambuf` allows an abstraction for connecting to a physical device. Classes derived from it work with files, memory, etc. The `ios` communicates to a `streambuf`. It keeps information on the state of the `streambuf` (good, bad, eof, etc.), and saves flags for use by `istream` and `ostream`.

- The `streambuf` class controls the buffer and its related member functions. It allows the ability to fill, flush, and empty the buffer.
- The `streambuf` is an object of `ios` class. The base class of input and output stream classes is `ios` class.
- The `istream` and `ostream` classes are derived classes of `ios` class and control input and output streams.
- The `iostream` class is derived class of `istream` and `ostream`. It provides input and output functions to control console operations.

Table 2.3 describes the functions of `ios` class in brief. Figure 2.9 indicates various formatted output functions supported by `cout` object.

**Table 2.3** `ios` class functions

Function	Working
<code>width( )</code>	To set the required field width. The output will be displayed in given width.
<code>precision( )</code>	To set number of decimal point for a float value.
<code>fill( )</code>	To set a character to fill in the blank space of a field.
<code>setf( )</code>	To set various flags for formatting output.
<code>unsetf( )</code>	To remove the flag setting.



**Fig.2.9** Formatted functions with `cout` object

(1) `ios::width` (member functions)

The `width( )` function can be declared in two ways.

**Declaration:**

- (a) `int width( );`
- (b) `int width(int);`
- (c) `ios::width :`

**Declaration:**

```
int width( );
```

If this function is declared as given above, it returns to the present width setting.

(b) `ios::width`

**Declaration:**

```
int width(int);
```

If this function is declared as given above, it sets the width as per the given integer, and returns the previous width setting. The setting should be reset for each input or output value if width other than the default is desired.

**2.30 Write a program to set column width and display the characters at specified position.**

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

void main( )
{
    clrscr();
    cout . width(5);
    cout <<"A";
    cout . width(15);
    cout <<"B";
}
```

**OUTPUT**

A        B

**Explanation:** The first `cout . width( )` statement sets the width 5. The `cout` statement sets the column width position at 5 and displays the character “A” at column 5. Similarly, the column width is set 15 and the character ‘B’ is displayed at the 15th column.

**2.31 Write a program to use both the formats of `width( )` function and display the result.**

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

```
    clrscr( );
    cout.width(50);
    int x=cout.width(1);
    cout <<x;
}
```

## OUTPUT

**50**

**Explanation:** In the above program, the `width( )` function call sets the width at column 50. The second `width( )` function call sets the width at column 1 and returns the previous setting i.e., 50. The integer x collects this value. The `cout( )` statement displays the number 50 at column 1.

### (2) `ios::precision`

This function can be declared in two ways.

#### **Declaration:**

- (a) `int precision(int);`
- (b) `int precision( );`
- (a) `int precision(int);`

#### **Declaration:**

```
int precision(int);
```

If the function is declared as given above, it sets the floating-point precision and returns the previous setting. The precision should be reset for every value being output if we want a precision result other than the default.

#### (b) `ios:: precision`

#### **Declaration:**

```
int precision( );
```

If the function is used as given above, it returns the current setting of floating-point precision.

### **2.32 Write a program to set precision to two and display the float number.**

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

void main( )
{
    clrscr( );
    cout.precision(2);
    cout <<3.1452;
}
```

## OUTPUT

**3.14**

**Explanation:** In the above program, the `cout.precision()` statement sets float point precision to 2. The `cout` statement displays 3.14 instead of 3.1452.

### 2.33 Write a program to set a number of precision points. Display the results of $22/7$ in different precision settings.

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

void main( )
{
    clrscr( );
    int x=0;
    float pi;
for (x=10;x>=1;x--)
{
    pi=(float)22/7;
    cout.precision(x);
    cout <<"\n" <<pi;
}
    x=cout.precision(1);
    cout <<"\n\n Previous Setting :" <<x;
}
```

#### OUTPUT

**3.1428570747**

**3.142857075**

**3.14285707**

**3.1428571**

**3.142857**

**3.14286**

**3.1429**

**3.143**

**3.14**

**3.1**

#### Previous Setting : 1

**Explanation:** In the above program, the `for` loop executes from 10 to 1 in reverse order. Each time in `for` loop, equation  $22/7$  is calculated and result is stored in the variable `pi`. The precision is set according to the value of variable `x`.

(3) `ios::fill`

This function can be declared in two ways.

#### Declaration:

(a) `char fill( );`

- (b) char fill(char);
- (a) ios::fill

**Declaration:**

```
char fill( );
```

If the function is used as above, it returns the current setting of fill character.

- (b)ios::fill

**Declaration:**

```
char fill(char);
```

If the function is used as above it resets the fill character and returns the previous setting.

### 2.34 Write a program to set fill blank spaces with different symbols.

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

void main( )
{
    clrscr( );
    cout.fill('/');
    cout.width(20);
    cout<<"WEL" << endl;
    cout.fill('-');
    cout.width(10);
    cout<<"DONE";
}
```

#### OUTPUT

```
//////////WEL
-----DONE
```

**Explanation:** This program is same as the last one. Here two different characters are used to fill the blank spaces in the specified width. The width is specified using `cout.width( )` statement. The output can be observed as shown below:

/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	W	E	L
-	-	-	-	-	-	D	O	N	E									

**Tip:** The character '\' (back slash) is not allowed in `cout.fill( )` statement to fill the blanks. This is because it is used with escape sequences.

### 2.35 Write a program to show the effect of `cout.fill( )`.

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

```
# include <conio.h>

void main( )
{
    clrscr( );
    cout<<"Begin : ";
    cout.width(15);
    cout <<"ABC";
    cout<<"\n";
    cout<<"Begin : ";
    cout.width(15);
    cout.fill('#');
    cout<<"ABC";
}
```

### OUTPUT :

**Begin :        ABC**  
**Begin : #####ABC**

**Explanation:** In the first output line, blanks are displayed because the fill character is not set. In the second statement the symbol ‘#’ is displayed because the fill character is set to ‘#’. The output can be observed as shown below:

Begin						A	B	C	
Begin	#	#	#	#	#	#	A	B	C

### 2.36 Write a program to fill the trailing digits of a number by '\*' symbol.

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

void main( )
{
    clrscr( );
    int x=0, j=0;
    float pi;
    cout.fill('*');
for (x=10;x>=1;x--)
{
    pi=(float)22/7;
    cout.width(j++);
    cout.precision(x);
    cout <<"\n"<<pi;
}
    cout.width(j++);
    cout.precision(x);
    cout <<"\n"<<pi;
    x=cout.fill( );
```

```

    cout <<"\n\n Fill setting :"<<(char)x;
}

OUTPUT
3.1428570747
3.142857075*
3.14285707**
3.1428571***
3.142857****
3.14286*****
31429*****
3.143*****
3.14*****
31*****
3.142857
Fill setting :*

```

**Explanation:** In the above program, the precision setting and display of floating point number using precision is the same as previous example. In the `for` loop each time `width( )` is set to the value of variable `j`. The value of `j` increases at each iteration. The floating precision number decreases from top to bottom and blank spaces remain in the specified field area by the statement `width( )`. The unused area is filled with '`*`'. The setting of fill character is done before the `for` loop statement. The fill character may be set to any character. The `fill( )` statement after `for` loop body displays the character used `for` filling.

(4) `ios::setf`

This function can be declared in two ways.

**Declaration:**

- (a) `long setf(long sb, long f );`
- (b) `long setf(long);`
- (a) `ios:: setf`

**Declaration:**

`long setf(long sb, long f);`

The bits according to that marked in variable `f` are removed in the data member `x_flags`, and then reset to those marked in variable `sb`. Using the constants in the formatting flag enumeration of class `ios` can specify value of variable `sb`.

(b) `ios:: setf`

**Declaration:**

`long setf(long);`

If the above declaration is used in the program, it sets the flags according to those marked in the given long. The flags are set in the data member `x_flags` of class `ios`. The use of constants in the formatting flags can specify the long enumeration of class `ios`. It returns the previous settings.

## 2.11 BIT FIELDS

The `ios` class contains the `setf( )` member function. The flag indicates the format design. The syntax of the `unsetf( )` function is used to clear the flags. The syntaxes of function are as follows.

Syntax: `cout.setf(v1, v2);`

Syntax: `cout.unsetf(v1);`

where, variable `v1` and `v2` are two flags. **Table 2.4** describes the flag and bit-field setting that can be used with this function. The `unsetf( )` accepts setting of `v1` and `v2` arguments.

**Table 2.4** Flags and Bits

Format	Flag (v1)	Bit-field (v2)
Left justification	<code>ios::left</code>	<code>ios::adjustleft</code>
Right justification	<code>ios::right</code>	<code>ios::adjustright</code>
Padding after sign and base	<code>ios::internal</code>	<code>ios::adjustinternal</code>
Scientific notation	<code>ios::scientific</code>	<code>ios::floatscientific</code>
Fixed point notation	<code>ios::fixed</code>	<code>ios::floatfixed</code>
Decimal base	<code>ios::dec</code>	<code>ios::base10</code>
Octal base	<code>ios::oct</code>	<code>ios::base8</code>
Hexadecimal base	<code>ios::hex</code>	<code>ios::base16</code>

(a) `ios:: adjustfield`

It is a data member and used with `setf( )` function to arrange padding to the left or right, or for internal fill. It can be declared as follows.

**Declaration:**

```
static const long adjustfield;
```

(b) `ios::floatfield`

It is a data member and used with `setf( )` function to set the float point notation to scientific or fixed. It is declared as follows.

**Declaration:**

```
static const long floatfield;
```

(c) `ios::basefield`

It is a data member and used with `setf( )` function to set the notation to a decimal octal, or hexadecimal base. It is declared as follows.

**Declaration:**

```
static const long basefield;
```

**2.37 Write a program to display the message left and right justified.**

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

void main( )

{
    clrscr( );
    cout.fill('=");
    cout.setf(ios::right, ios::adjustfield);
    cout.width(20);
    cout <<"Figure " << "\n";
    cout.setf(ios::left, ios::adjustfield);
    cout.width(20);
    cout <<"Figure " << "\n";
}
```

**OUTPUT**

=====Figure

Figure =====

**Explanation:** In the above program, the fill character is set to sign “=” . The `setf( )` is set to right justified. The column width is set to 20. The message “Figure” appears at the 20th column. Before the message the blank space is filled with the sign “=” . Again the justification property is set to left in `setf( )` function. The text appears left justified. The output can be observed as follows.

=	=	=	=	=	=	=	=	=	=	=	=	=	=	F	i	g	u	r	e
F	i	g	u	r	e	=	=	=	=	=	=	=	=	=	=	=	=	=	=

**2.38 Write a program to display the number in scientific format with sign.**

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

void main( )
{
    clrscr( );
    cout.fill('=");
    cout.setf(ios::internal, ios::adjustfield);
```

```

    cout.setf(ios::scientific, ios::floatfield);
    cout.width(15);
    cout <<- 3.121;
}

```

## **OUTPUT**

- ===== 3.121e+00

**Explanation:** In the above program, the fill character is set to sign “=”. The setf( ) properties are set to internal and scientific. The scientific properties display the number in scientific (e) format. The internal properties display the sign in spaces before blank. If the settings were removed the output would be as given below:

Effect of statement

```
cout.setf (ios::internal, ios::adjustfield);
```

-	=	=	=	=	=	3	.	1	2	1	e	+	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

The output without this statement would be

=	=	=	=	=	-	3	.	1	2	1	e	+	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

The property ios::fixed displays the float number without scientific format, though the number is big. If the floating-point number is more than 6, extras are ignored. This format displays floating point up to 6 only.

## **2.39 Write a program to convert decimal number to hexadecimal and octal format.**

```

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

void main( )
{
    clrscr( );
    cout.setf(ios::hex, ios::basefield);
    cout <<"\n Hexadecimal of 184 is : "<<184;
    cout.setf(ios::oct, ios::basefield);
    cout <<"\n Octal of 15 is      : "<<15;
    cout.setf(ios::dec, ios::basefield);
    cout <<"\n Decimal of 0xfe is   : "<<0xfe;
}

```

## **OUTPUT**

**Hexadecimal of 184 is : b8**

**Octal of 15 is : 17**

**Decimal of 0xfe is : 254**

**Explanation:** In the above program, the properties of setf( ) function are set to ios::hex, ios:: oct and ios::dec. After these settings,

the decimal number given in `cout( )` statement will convert to its hexadecimal and octal equivalent respectively. The hexadecimal number `0xfe` is converted to its decimal equivalent.

## 2.12 FLAGS WITHOUT BIT FIELD

The flags described in [Table 2.5](#) have no corresponding bit fields. The programs on these functions are illustrated below:

**Table 2.5** Flags without bit fields

Flag	Working
<code>ios :: showbase</code>	Use base indicator on output
<code>ios :: showpos</code>	Display + preceding positive number
<code>ios :: showpoint</code>	Show trailing decimal point and zeroes
<code>ios :: skipws</code>	Skip white space on input
<code>ios :: unitbuf</code>	Flush all streams after insertion
<code>ios :: uppercase</code>	Use capital characters for scientific and hexadecimal values
<code>ios :: stdio</code>	Adjust the stream with C standard Input and output
<code>ios :: boolalpha</code>	Converts bool values to text (“true” or “false”)

## 2.40 Write a program to use various setting given in the table.

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

void main( )
{
    clrscr( );
    cout.setf(ios::showpos);
    cout <<1254;

    cout.setf(ios::showpoint);
    cout <<"\n" <<1254.524;

    cout.setf(ios::hex,ios::basefield);
    cout <<"\n" <<184;
    cout.setf(ios::uppercase);
    cout <<"\n" <<184;
}
```

### OUTPUT

```
+1254
+1254.524000
0xb8
0XB8
```

**Explanation:** In the above program, the setting `ios::showpos` displays the + sign before the number **1254**. The setting `ios::showpoint` displays the trailing zeroes after the number **1254.524**. The setting `ios::hex` converts the decimal number to hexadecimal. It uses small letters. The setting `ios::uppercase` displays the hexadecimal number in uppercase.

## 2.13 MANIPULATORS

The output formats can be controlled using manipulators. The header file `iomanip.h` has a set of functions. Effect of these manipulators is the same as `ios` class member functions. Every `ios` member function has two formats. One is used for setting and second format is used to know the previous setting. But the manipulator does not return to the previous setting. The manipulator can be used with `cout( )` statement as given below.

```
cout << m1 << m2 << v1;
```

Here `m1` and `m2` are two manipulators and `v1` is any valid C++ variable.

**Table 2.6** describes the most useful manipulators. The manipulator `hex`, `dec`, `oct`, `ws`, `endl`, and `flush` are defined in `iostream.h`. The manipulator `setbase`, `width( )`, `fill( )` etc. that requires an argument are defined in `iomanip.h`.

**Table 2.6** Pre-defined manipulators

Manipulator	Function
<code>setw( int n)</code>	The field width is fixed to <code>n</code> .
<code>setbase</code>	Sets the base of the number system.
<code>setprecision(int p)</code>	The precision point is fixed to <code>p</code> .
<code>setfill( char f)</code>	The fill character is set to character stored in variable <code>f</code> .
<code>setiosflags(long l)</code>	Format flag is set to <code>1</code> .
<code>resetiosflags(long l)</code>	Removes the flags indicated by <code>1</code> .
<code>endl</code>	Splits a new line.
<code>skipws</code>	Omits white space on input.
<code>noskipws</code>	Does not omit white space on input.
<code>ends</code>	Adds null character to close an output string.
<code>flush</code>	Flushes the buffer stream.
<code>lock</code>	Locks the file associated with the file handle.
<code>ws</code>	Used to omit the leading white spaces present before the output.
<code>hex, oct , dec</code>	Displays the number in hexadecimal, octal, and decimal.

## **2.41 Write a program to display formatted output using manipulators.**

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

main( )
{
    clrscr( );
    cout <<setw(10) <<"Hello"<<endl;
    cout <<setw(15) <<setprecision(2) <<2.5555;
    cout <<setiosflags(ios::hex);
    cout <<endl<<"Hexadecimal of 84 is : "<<84;
return 0;
}
```

### **OUTPUT**

**Hello**

**2.56**

**Hexadecimal of 84 is : 54**

**Explanation:** In the above program, the manipulator `setw(10)` sets the field width to 10. The message “hello” is displayed at column 10.

The `endl` inserts a new line. In the second `cout( )` statement, the `setprecision(2)` manipulator sets the decimal point to 2. The number 2.5555 will be displayed as 2.56 at column 15. The manipulator `setiosflag` sets the hexadecimal setting for the display of number. The last `cout( )` statement displays the equivalent hexadecimal of 84 i.e., 54.

## **2.42 Write a program to display the given decimal number in hexadecimal and octal format.**

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

void main( )
{
    clrscr( );
    int x=84;
    cout <<"\n Hexa-decimal Number : "<<hex<<x;
    cout <<"\n Octal Number : "<<oct<<x;
}
```

### **OUTPUT**

**Hexadecimal Number : 54**

**Octal Number : 124**

**Explanation:** In the above program, the integer variable x is declared and initialized with 84. The first cout statement displays the decimal number to its equivalent hexadecimal number. The manipulator hex converts decimal number to its hexadecimal equivalent. The second cout statement converts decimal number to its equivalent octal number.

#### **2.43 Write a program to read number in hexadecimal format using cin statement. Display the number in decimal format.**

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

void main( )
{
    clrscr( );
    int x;
    cout <<"\n Enter Hexa-decimal Number : ";
    cin >>hex>>x;
    cout <<"\n Decimal Number : "<<dec<<x;
}
```

#### **OUTPUT**

**Enter Hexadecimal Number : 31**

**Decimal Number : 25**

**Explanation:** In the above program, the cin statement reads a number in hex format. The cout statement displays its equivalent decimal number with the use of dec manipulator.

#### **2.44 Write a program to demonstrate the use of endl manipulator.**

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

void main( )
{
    clrscr( );
    cout <<" Demo of endl ";
    endl(cout);
    cout <<" It splits a line" ;
}
```

#### **OUTPUT**

**Demo of endl**

**It splits a line**

**Explanation:** In the above program, endl manipulator is used to split the line. The two strings are displayed on two separate lines.

#### **2.45 Write a program to demonstrate the use of flush( ) statement.**

```

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

void main( )
{
    char text[20];
    clrscr( );

    cout <<"Enter text : ";
    cin.getline(text,20);
    cout<<"Text Entered : "<<text;
    flush(cout);
}

```

## OUTPUT

**Enter text : Buffer**

**Text Entered : Buffer**

**Explanation:** In the above program, the statement `flush (cout)` flushes the buffer. This statement can be used as `cout<<flush`.

## 2.14 USER-DEFINED MANIPULATORS

The programmer can also define his/her own manipulator according to the requirements of the program. The syntax for defining manipulator is given below:

```

ostream & m_name( ostream & o )
{
    statement1;
    statement2;
    return o;
}

```

The `m_name` is the name of the manipulator.

**2.46 Write a program to create manipulator equivalent to '\t'. Use it in the program and format the output.**

```

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

ostream & tab (ostream & o)
{
    o <<"\t";
    return o;
}

void main( )
{
    clrscr( );
}

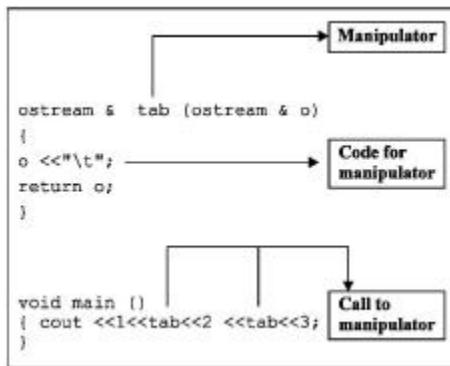
```

```
    cout <<1<<tab<<2 <<tab<<3;  
}
```

## OUTPUT

1 2 3

**Explanation:** Figure 2.10 illustrates working of the program.



**Fig.2.10** Working of tab manipulator

In the above program, tab named manipulator is defined. The definition of tab manipulator contains the escape sequence '\t'. Whenever we call the tab manipulator, the '\t' is executed and we get the effect of tab.

### 2.47 Write a program to display message using manipulator.

```
# include <iostream.h>  
# include <iomanip.h>  
# include <conio.h>  
  
ostream & N (ostream & o)  
{  
    o <<"Negative Number";  
    return o;  
}  
  
ostream & P (ostream & o)  
{  
    o <<"Positive Number";  
    return o;  
}  
  
void main( )  
{  
    int x;  
    clrscr( );  
    cout <<"\n Enter a Number : ";  
    cin >>x;
```

```

    if (x<0)
        cout <<x <<" is " <<N;
    else
        cout <<x <<" is " <<P;
}

```

## **OUTPUT**

**Enter a Number : -15**

**-15 is Negative Number**

**Explanation:** In the above program, two manipulators N and P are created. When called, N displays the message “Negative number” and P displays the message “Positive Number.”

## **2.15 MANIPULATOR WITH ONE PARAMETER**

The prototype declared in the iomanip.h as described earlier allows us to define individual set of macros. The manipulator can be created without the use of int or long arguments.

### **2.48 Write a program with one parameter to display the string in the middle of the line.**

```

#include <iostream.h>
#include <iomanip.h>
#include <string.h>
#include <conio.h>

ostream& fc (ostream& ost, int iw)
{
    for (int k=0;k<((75-iw)/2);k++)
        ost <<" ";
    return (ost);
}

OMANIP (int) middle (int iw)
{
    return OMANIP (int) (fc,iw);
}
int main( )
{
    clrscr( );
    char *m=" * Well come *";
    cout <<middle (strlen(m)) <<m;
    return 0;
}

```

## **OUTPUT**

**\*\* WEL COME \*\***

**Explanation:** In the above program, `middle` is a user-defined parameterized manipulator. It receives a value `strlen(m)` i.e., string length. The header file `IOMANIP.H` defines a macro, `OMANIP(int)`. It is expanded to `class_OMANIP_int`. Due to this the definition consists of a constructor and an overloaded ostream insertion operator. When `middle()` function is added into the stream, it invokes the constructor which generates and returns an `OMANIP_int` object. The `fc()` function is called by the object constructor.

## 2.16 MANIPULATORS WITH MULTIPLE PARAMETERS

In this type of manipulator, multiple arguments are passed on to the manipulator. The following program explains it.

### 2.49 Write a program to create manipulator with two arguments.

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

struct w_p
{
    int w;
    int p;
};

IOMANIPdeclare (w_p);

static ostream& ff(ostream& os, w_p w_p)
{
    os.width (w_p.w);
    os.precision(w_p.p);
    os.setf(ios::fixed);
    return os;
}

OMANIP (w_p) column (int w, int p)
{
    w_p w_p;
    w_p.w=w;
    w_p.p=p;
    return OMANIP (w_p) (ff,w_p);
}
int main ( )
{
    clrscr( );
    double n,sq,sqr;
```

```

cout <<"number\t" <<"square\t" <<"\tsquare root\n";
cout <<"=====\n";
n=1.0;
for (int j=1;j<16;j++)
{
    sq=n*n;
    sqr=sqrt(n);
    cout.fill('0');
    cout <<column(3,0)<<n <<"\t";
    cout <<column(7,1) <<sq <<"\t\t";
    cout <<column(7,6) <<sqr <<endl;
    n=n+1.0;
}
return 0;
}

```

## OUTPUT

<b>number</b>	<b>square</b>	<b>square root</b>
=====	=====	=====
<b>001</b>	<b>0000001</b>	<b>0000001</b>
<b>002</b>	<b>0000004</b>	<b>1.414214</b>
<b>003</b>	<b>0000009</b>	<b>1.732051</b>
<b>004</b>	<b>0000016</b>	<b>0000002</b>
<b>005</b>	<b>0000025</b>	<b>2.236068</b>
<b>006</b>	<b>0000036</b>	<b>2.44949</b>
<b>007</b>	<b>0000049</b>	<b>2.645751</b>
<b>008</b>	<b>0000064</b>	<b>2.828427</b>
<b>009</b>	<b>0000081</b>	<b>0000003</b>
<b>010</b>	<b>0000100</b>	<b>3.162278</b>
<b>011</b>	<b>0000121</b>	<b>3.316625</b>
<b>012</b>	<b>0000144</b>	<b>3.464102</b>

<b>013</b>	<b>0000169</b>	<b>3.605551</b>
<b>014</b>	<b>0000196</b>	<b>3.741657</b>
<b>015</b>	<b>0000225</b>	<b>3.872983</b>

**Explanation:** The above program is same as the previous one. The only difference is that here two parameters are used. The user-defined manipulator is assigned two integer values. The first argument decides the number of spaces and the second argument decides number of decimal places. After initializing the `w_p` structure, constructor is called. The constructor creates and returns a `_OMANIP` object.

## 2.17 CUSTOM BUILT I/O OBJECTS

### (1) Creating Output Object

We have studied the objects `cin` and `cout` and their uses. It is also possible to declare objects equivalent to these objects. The object declaration is same as variable declaration. In object declaration, instead of data type, the class name is written followed by object names.

For example,

```
ostream print(1);
```

In the above example, `print` is an object of class `ostream`. The `ostream` handles the output functions. The parenthesis with value one calls the constructor. The details about objects and constructors are illustrated in successive chapters of this book.

The object `print` can be used in place of `cout`. It can also call the member functions such as `put( )`, `write( )` etc. The following programs illustrate this concept.

### 2.50 Write a program to declare object of ostream class and display the text on the screen.

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

void main( )
{
    clrscr( );
    char text[10] = "Hello";
    ostream print(1);
    print << text;
}
```

## OUTPUT

Hello

**Explanation:** In the above program, the character array text is initialized with string “Hello”. The print is an object of class ostream. The print uses the operator << and displays the contents of array text on the screen.

### 2.51 Write a program to call member function of ostream class using custom object.

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

void main( )
{
    clrscr( );
    ostream print(1);
    print.put('C').put('P').put('P');
}
```

## OUTPUT

CPP

**Explanation:** In the above program, print is an object of class ostream. The ostream invokes the member function put( ) and displays the given character on the screen.

## (2) CREATING INPUT OBJECT

The object equivalent to cin can be declared as follows.

```
istream_withassign in;
```

Here, the `istream_withassign` is a class, which provides assignment operator. The object in is an object of the `istream_withassign` class.  
`in=cin;`

The operator associated with `cin` is assigned to object in. Now the object in can be used instead of `cin`. The following program illustrates this point.

### 2.52 Write a program to create object equivalent to `cin` and perform input operation.

```
# include <iostream.h>
# include <conio.h>
void main( )
{
    clrscr( );
    char text[20];
    istream_withassign in;
    in=cin;
    cout<<"\n Enter text : ";
```

```
    in.getline(text,20);
    cout <<"\n The text entered : ";
    cout<<text;
}
```

## OUTPUT

Enter text : BE HAPPY

The text entered : BE HAPPY

**Explanation:** In the above program, `in` is an object of class `istream_withassign`. The operator associated with `cin` is assigned to object `in`. The `in` object invokes the function `getline()` and reads text. The read text is displayed by the `cout` statement.

## SUMMARY

- (1) The input output function of C++ works with different physical devices. It also acts as interface between the user and the device.
- (2) A stream is a series of bytes that acts as a source and destination for data. The source stream is called **input stream** and the destination stream is called **output stream**.
- (3) The `cin`, `cout`, `cerr` and `clog` are predefined streams.
- (4) The header file `iostream.h` must be include when we use `cin` and `cout` functions.
- (5) The **istream** and **ostream** are derived classes from `ios` base class. Figure 2.6 displays all the derived classes.
- (6) The formatting of output can be effectively done with member functions of **ios** class. The member function `width()`, `precision()`, `fill()` and `setf()` allows user to design and display the output in formatted form.
- (7) Table 2.3 describes the list of functions without bit fields. These functions are also used for formatting the output.
- (8) The `putback()` replaces the given character into the input stream. The member function `ignore`, ignores a number of given characters till it finds termination character.
- (9) Manipulators also help the user in formatting of output. The programmer can also create his/ her own manipulators.
- (10) The header file `iomanip.h` contains pre-defined manipulators. Table 2.6 describes these manipulators.

## EXERCISES

[A] Answer the following questions.

- (1) List the names of ***pre-defined streams*** with their '**C**' equivalents?
- (2) What are formatted and unformatted input/output functions?
- (3) Distinguish between
  - (a) `cin( )` and `scanf( )`
  - (b) `cout( )` and `printf( )`
  - (c) `ios::fixed` and `cout.precision( )`
- (4) What are the uses of `put( )` and `get( )` functions?
- (5) What is the use of `getline( )` function? Which two arguments does it require?
- (6) Describe bit fields required in `setf( )` function.
- (7) List the flags without bit fields with their working.
- (8) What is the role of `iostream.h` and `iomanip.h` header files?
- (9) Write the statement for concatenation of two strings using `cout.write( )` statement.
- (10) Describe the procedure for designing manipulator.
- (11) What is the function of `peek( )` and `ignore( )` functions?
- (12) What are single and multiple parameter manipulators?
- (13) In which format the `cout` statement display the address of variable? How can it be converted to unsigned?
- (14) In which situation the `putback( )` function is useful?
- (15) What is the use of `ignore( )` function?
- (16) What do you mean by formatted and unformatted data?
- (17) Explain the procedure for creating custom input/output objects.

**[B] Answer the following by selecting the appropriate option.**

- (1) The `cin` and `cout` functions require the header file to include
  - (a) `iostream.h`
  - (b) `stdio.h`
  - (c) `iomanip.h`
  - (d) none of the above
- (2) The `set.precision( )` is used to set
  - (a) decimal places
  - (b) number of digits
  - (c) field width
  - (d) none of the above
- (3) To fill unused section of the field, the character is set by the function
  - (a) `fill( )`
  - (b) `width( )`
  - (c) `precision( )`
  - (d) none of the above

(4) The manipulator `<<endl` is equivalent to

- (a) '\n'
- (b) '\t'
- (c) '\b'
- (d) none of the above

(5) This function accepts the string with blank spaces

- (a) `getline( )`
- (b) `cin`
- (c) `scanf ( )`
- (d) none of the above

(6) The streams is a

- (a) flow of data
- (b) flow of integers
- (c) flow of statements
- (d) none of the above

(7) The statement `cin<<hex` reads the data in

- (a) hexadecimal format
- (b) octal format
- (c) binary format
- (d) decimal format

(8) The `gcount( )` function counts the

- (a) extracted character
- (b) inserted character
- (c) both (a) and (b)
- (d) none of the above

(9) The buffer is used to move data between

- (a) input/output devices and computer
- (b) input and output devices
- (c) input devices to storage devices
- (d) none of the above

(10) The buffer is a

- (a) block of memory
- (b) part of ram
- (c) part of hard disk
- (d) none of the above

### **[C] Attempt the following programs.**

(1) Write a program to receive 5 float numbers. Display the number with 6 decimal places.

(2) Write a program to display the hexadecimal, octal equivalent of 85, 25 152, 251, and 458 numbers.

(3) Write a program to display decimal equivalent of hexadecimal numbers 0x52, 0x98, 0x101, 0x524, and 0x421.

(4) Write a program to read ten records of student with the information name, age, and date of birth and arrange the output in the following format. Fill the unused field with dot.

Sr.no.	Name	Date of birth
01.....	Ajay.....	01/01/1978

(5) Write a program to accept string using `get()` function. Display the string using `write()` and `put()` function. Mention the difference between `write()` and `put()` functions.

(6) Write a program to design following manipulators:

Manipulator Name	Function ( Escape sequence)
<<bkp (Backspace)	'\b' or '\r'
<<newl	'\n'
<<bell	'\a'
<<plus	+'(Displays + sign)
<<dollar	\$ (Displays \$ sign)

(7) Write a program to receive item code, quantity, price and calculate the amount. Display the data in the following format.

**Note:** (a) Sr.No. and Item code is left justified.

(b) The price and amount is right justified.

(c) Three-digit precision for amount field.

Sr.No.	Item code	Quantity	Price	Amount
01	0101	10	55.15	551.501

(8) Write a program to enter text up to 100 characters through the keyboard. Ignore symbols and replace them with \*. Display dot in-between two words.

(9) Given  $x = 5$ ,  $y = 8$ ,  $z = 12$ . State the value of the variable on the left side of the following statements:

(a)  $a = x/y*z$

(b)  $b = y/z*x$

(c)  $c = z*y/z$

(d)  $d = x*z/y$

(e)  $d = (x/y)*z$

(10) If  $p = 0.5$ ,  $q = -2.0$ ,  $r = 7.3$ ,  $s = 10.4$ ,  $m = 2$ ,  $n = -3$ , find the values that will be stored as the result of the following expressions:

(a)  $x = 5.0 * p + q - (r + s * 3.0)$

(b)  $y = 3.0 * p + q + s * 2$

(c)  $z = p * s + q * r - s * q / 5.0$

(d)  $a = \sqrt{4.0 - r + 9.0 * q}$

(11) Solve the following algebraic expressions:

(a)  $k = 4.5 \log_{10} x + 2xy + x^5$

(b)  $j = (bx + x)/(bx - b)$

(c)  $z = (x)_2 + (x + 1)_2 + (x + 2)_2$

(d)  $u = a*b/(c + d*g + k) + e$

(12) Write a program to generate following outputs.

-	#	#	#	7	8	4	.	5	0
*	4	5	-	4	8	\$	\$	\$	2
7	8	-	4	9		7	9	.	4
-	%	%	5	8	9	.	4	8	0

(13) Write a program to generate following output.

```
C  
C L  
C L A  
C L A S  
C L A S S  
C P L U S P L U S  
C P L U S P L  
C P L U S  
C P L  
C
```

(14) Write a program to calculate simple interest and total amount. Input Principal amount, period, and rate of interest.

(15) Write a program to set width and display the integer number.

(16) Write a program to demonstrate use of showpos and showpoint flags.

(17) Write a program to display square and cube of number 1.5 to 16.5 in table format.

(18) Write a program to display octal and hexadecimal equivalent of decimal number ranging from 100 to 200.

### [D] Find the output of the following programs.

(1)

```
cout.setf(ios::left, ios::adjustfield);  
cout.fill('*');  
cout.precision(2);  
cout.width(7);  
cout<<345.54;  
cout.width(8);  
cout<<78;
```

(2)

```
cout.setf(ios::internal, ios::adjustfield);  
cout.fill('$');  
cout.precision(4);  
cout.width(11);  
cout<<-543.453;
```

```
(3)
cout.setf(ios::showpos);
cout.setf(ios::showpoint);
cout.setf(ios::internal, ios::adjustfield);
cout.precision(4);
cout.width(12);
cout<<2342.34;
(4)
cout.precision(2);
cout<<4.57<<endl;
cout<<7.1453<<endl;
cout<<4.5132<<endl;
cout<<8.004<<endl;
(5)
cout.fill('$');
cout.precision(4);
cout.setf(ios::internal, ios:: adjustfield);
cout.setf(ios::scientific, ios:: floatfield);
cout.width(13);
cout<<-78.47854;
(6)
cout fillC ('= ');
cout setf(ios:: internal, ios::adjustfield);
cout setf(ios:: scientific, ios::floatfield);
cout unsetf(ios ::scientific);
cout width(15);
cout <<- 3.121;
(7)
cout.width (7);
cout<<123<56;
(8)
# include <iostream.h>
# include <conio.h>
void main ( )
{
clrscr( );
int i;
char name[]="OBJECT";
for (i=6;i>=0;i--)
{
cout.write(name,i);
endl(cout);
}
}
(9)
# include <iostream.h>
# include <conio.h>
void main ( )
```

```
{  
clrscr( );  
char name[20];  
cout<<"\n Enter a string : "  
cin.getline(name, 0);  
cout<<name;  
}
```

# 3

## CHAPTER

# C++ Declarations

- [3.1 Introduction](#)
- [3.2 Parts of C++ Program](#)
- [3.3 Types of Tokens](#)
- [3.4 Keywords](#)
- [3.5 Identifiers](#)

- [3.6 Dynamic Initialization](#)
- [3.7 Data Types in C++](#)
- [3.8 Basic Data Type](#)
- [3.9 Derived Data Type](#)
- [3.10 User-Defined Data Type](#)
- [3.11 The void Data Type](#)
- [3.12 Type Modifiers](#)
- [3.13 Wrapping Around](#)
- [3.14 Typecasting](#)
- [3.15 Constants](#)
- [3.16 Constant Pointers](#)
- [3.17 Operators in C and C++](#)
- [3.18 Precedence of Operators in C++](#)
- [3.19 Referencing \(&\) and Dereferencing \(\\*\) Operators](#)
- [3.20 Scope Access Operator](#)
- [3.21 Memory Management Operators](#)
- [3.22 Comma Operator](#)

—• 3.23 Comma in Place of Curly Braces

### 3.1 INTRODUCTION

Before writing any program it is necessary to know the rules of syntaxes of the language. This enables a programmer to write error-free programs. The user makes many mistakes if he/she does not know the rules of the language even though the logic is clear. Thus the user fails to write programs due to poor knowledge of basic concepts.

Some basic concepts of C++ language are illustrated in this chapter to help the programmer develop the programs. As object-oriented programming is a new technology, the programmer will find a notable amount of new concepts to learn and master in it. This chapter explores syntaxes, data types, keywords etc.

Most of the C++ programmers are former C programmers. Some rules and regulations of C programs are valid in C++ also. Infact C++ is upward compatible to C. Hence the programs with C can be executed with C++. In C++, various improvements are done such as variable declaration is allowed anywhere in the program, new operators such as reference, scope access operator etc. are introduced and so on. All these concepts are illustrated ahead with figures and examples.

### 3.2 PARTS OF C++ PROGRAM

A Program consists of different sections as shown in Fig. 3.1. Some of them are optional and can be excluded from the program if not required. For example, every C and C++ program starts with function main( ), and if the user writes a program without main( ), it won't be executed. Hence, it is compulsory to write the main( ) function. In C, the structure allows to combine only variables of different data types. In addition, C++ allows combining variable and functions in structures and classes. The structure and classes are same and used for the same purpose. Functions can be defined inside or outside the class.

<b>Include Files</b>
<b>Class Declaration or Definition</b>
<b>Class Function Definitions</b>
<b>main() function</b>

**Fig.3.1** Parts of C++ program

#### (1) INCLUDE FILES SECTION

Like C, C++ program also depends upon some header files for function definition. Each header file has an extension.h. The file should be included using # include directive as per the format given below

**Example:** # include <iostream.h> or # include "iostream.h ."

In this example `<iostream.h>` file is included. All the definitions and prototypes of function defined in this file are available in the current program. This header file also gets compiled with the original program. The header files are included at the top of the program so that they can be accessible from any part of the program. The user can also include them inside the program, but it is a good practice to include header files at the top of the program.

## (2) CLASS DECLARATION OR DEFINITION

Declaration of a class is done in this section. It is also possible to declare class after the function `main()`, but it is better to declare it before `main()` function. A class contains data variables, function prototypes or function definitions. The class definition is always terminated by a semi-colon. Function prototype declaration reports to the compiler its name, return type and required argument list of the function. The function definition tells the compiler how the function works.

*Example:* `float sum(float, float);`

In the above example, the `sum()` function performs addition of two float numbers and returns float value.

## (3) CLASS FUNCTION DEFINITIONS

This part contains definition of functions. The function definition can be done outside or inside the class. The functions defined inside the class are implemented as inline functions. The inline mechanism is illustrated in [Chapter 5, “Functions in C++”](#). When a function is large, it is defined outside the class. In this case, prototype of a function is declared inside the class. [Chapter 6, “Classes and Objects”](#) illustrates these concepts.

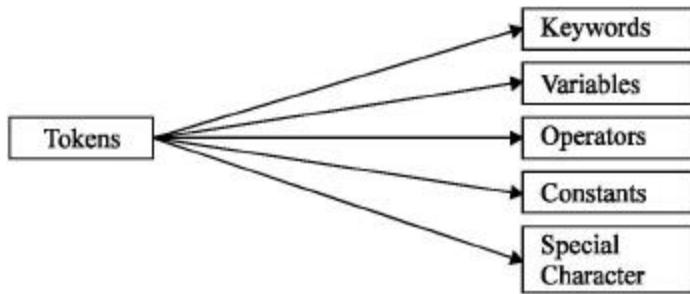
## (4) THE `MAIN()` FUNCTION

Like C, C++ programs always start execution from `main()` function. Thus execution of every program in C++ starts with `main()` function.

### 3.3 TYPES OF TOKENS

The C++ programs contain various components. The compiler identifies them as tokens. Tokens are classified in the following types. [Figure 3.2](#) also indicates types of tokens.

- (a) Keywords
- (b) Variables
- (c) Constants
- (d) Special character
- (e) Operators



**Fig.3.2** Types of tokens

The keywords are reserved set of words with fixed meanings. The variables are used to hold data temporarily. The operators are used to perform different operations such as arithmetic, logical etc. The values such as 1, 5, 2.5 etc. are known as constants. The operators such as #, and ~ are known as special characters. The # is used for preprocessor directive, ? is a conditional operator and ~ is used for bitwise operation. The detailed descriptions of these tokens are described in sections ahead.

### 3.4 KEYWORDS

The C++ keywords are reserved words by the compiler and have fixed meanings. All C keywords are valid in C++. There are 63 keywords in C++. In addition, Turbo C++ permits 14 extra keywords as described in [Table 3.4](#). The programmer may not apply them in the programs for defining variable names. However, few C++ compilers permit to declare variable names that exactly match to the keywords.

The common keywords between C and C++ are listed in [Table 3.1](#). [Table 3.2](#) describes the additional keywords of C++. These keywords are used with classes, templates, and exception handling etc. [Table 3.3](#) contains keywords added by the ANSI committee. [Table 3.4](#) contains additional keywords provided by Turbo C++ compiler.

### 3.5 IDENTIFIERS

Identifiers are names of variables, functions, and arrays etc. They are user-defined names, consisting of a sequence of letters and digits, with a letter as a first character. Lower case letters are preferred. However, the upper case letters are also permitted. The ( \_ ) underscore symbol can be used as an identifier. In general, underscore is used to link two words in long identifiers. ANSI C++ restricts no bound on length of variables.

**Table 3.1** C and C++ common keywords

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

**Table 3.2** Additional C++ keywords

asm	private
catch	protected
class	public
delete	template
friend	this
inline	throw
new	try
operator	virtual

**Table 3.3** Keywords added by ANSI committee

Keywords added by ANSI committee				
bool mutable typename	const_cast namespace Using	dynamic_cast reinterpret_cast wchar_t	explicit static_cast export	true false typeid

**Table 3.4** Additional Keywords in Turbo C++

Additional Keywords in Turbo C++				
cdecl _fastcall pascal	_cs huge _saveregs	_ds interrupt _seg	_es _loadds _ss	far near

- (1) The identifier name must begin with a character and should not start with a digit. There should not be any space between the characters in the variable but underscore is allowed.
- (2) The identifier name should not be a C++ keyword.
- (3) The identifier name may be a combination of upper and lower characters. For example, the variables suM, sum and Sum are not the same.

## **(1) VARIABLE DECLARATION AND INITIALIZATION**

**Variable** A variable is used to store values. Every variable has memory location. The memory locations are used to store the values of the variables. The variables can be of any type. It may be integer, char, float etc. The variable can hold single value at a time of its type. The values of variable varies i.e., the integer variable may contain values 2,6,33 and so on. The programmer can change contents of the variable. Thus, the identifier that holds varying values is called variables.

**Variable Declaration** In C, all the variables must be declared in the declaration part. Hence, every time if one needs to declare a variable, the programmer must go back to the beginning of the program.

C++ permits declaration of variables anywhere in the program. This makes the programmer more comfortable to declare the variables and need not go to the beginning of the program. The declaration of variable consists of name of data type and variable list as follows:

**Syntax:**

- (a) Data-type v;
- (b) Data-type v1, v2...vn;

The programmer should declare data type explicitly before variable name. The data type tells the compiler about the type of data that is allowed to store in the variable. The identifier v is a variable. The statement (a) declares only one variable. The programmer can also declare more than one variable in a single statement as per statement (b).

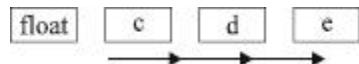
**Example:**

- (a) int a;
- (b) float c, d, e;

In example (a), a is a variable of integer type. Example (b) declares variable c, d, and e of float type.

The statement (b) declares three variables in a single statement. The variable c is created first followed by d and e. The creation of variable takes place from left to right as shown in [Figure 3.3](#). Each variable has a unique memory location.

In [Figure 3.3](#) float is a data type and c,d and e are variables. The variable c is created first, followed by d and e. The direction of creation is from left to right. The extreme left variable is created first and the extreme right variable is created in the end.



**Fig.3.3** Variable declarations

### 3.1 Write a program to read two integers through the keyboard. Declare the variables in C++ style.

```
# include <iostream.h>
# include <conio.h>
main ( )
{
    clrscr( );
    cout <<"Enter Two numbers : ";
    int num;
    cin >>num;
```

```

    int num1;
    cin > >num1;
    cout <<"Entered Numbers are : ";
    cout <<num<<"\t"<<num1;
    return 0;
}

```

### **OUTPUT:**

**Enter Two numbers : 8 9**

**Entered Numbers are : 8 9**

**Explanation:** In the above program, the variables `num` and `num1` are declared inside the program and not at the beginning of the program. After declaration they are used with `cin` statement. The `cin` statement reads two integers and stores them in the variables `num` and `num1`. The `cout` statement displays entered values on the screen.

**Initialization** When a variable is declared, appropriate number of bytes are reserved for it in the random access memory. The bytes are filled with garbage value if the user does not assign them a value. If the user performs operations without initializing a variable, the result will be unexpected. Hence, before using the variable it is essential to initialize them. Assigning a value to the variable is called as *initialization*. When a value is assigned to the variable, the garbage value is removed and replaced with the given value. The declaration of variable is done only once but initialization of variable can be done for any number of times. When a new value is assigned to a variable, its previous value is replaced with a new one.

### **Example:**

```
int x;
```

Here, integer variable `x` is declared and not initialized. When the variables are not initialized, they are neither initialized to zero by the compiler nor they remain empty. The variables contain garbage values. The user cannot predict the garbage value since it is system dependent. The garbage values can be different on different systems.

If you try to print the value of `x`, the statement `printf ("%d", x)` will display the garbage value (-29281 on my system). The `cout <<x` will display the value 0. If you use both these statements one after another, the value displayed will be 0.

### **3.2 Write a program to display value of uninitialized variable using cout statement.**

```

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

void main( )
{
    clrscr( );
    int x;
    cout<<x<<endl;
    cout<<x+5<<endl;
    cout<<5+(++x);
}

```

### **OUTPUT**

```
0
5
6
```

**Explanation:** In the above program, the first `cout` statement displays the value of `x` to be zero. The second statement displays the value 5. The third statement displays the value 6. In the third statement, `x` is incremented first and then added to 5. This program is compiled under compact memory model. If this program is compiled and executed under any other memory models, the result will be garbage.

All the operations performed in the above program are directly put in the `cout` statement. If the above program is executed with the `printf( )` statement, the program will display the garbage values.

The following program displays garbage value of the variable:

### 3.3 Write a program to display garbage value of a variable.

```
# include <iostream.h>
# include <conio.h>
# include <stdio.h>
void main ( )
{
    int x,y;
    clrscr( );
    y=5+(++x);
    cout<<"Value of Y : "<<y;?
}
```

#### OUTPUT

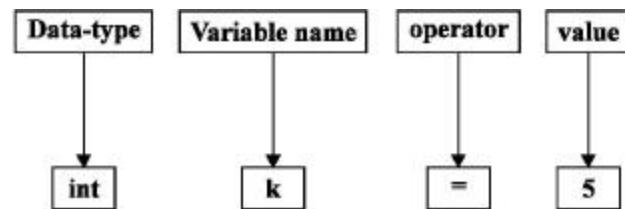
**Value of Y : 1134**

**Explanation:** In the above program, 5 is added to variable `x` and assigned to `y`. One may expect that the value of `y` will be 6. Here, the value of `y` displayed is a garbage value. From the above program it is clear that if the variable is not initialized, it will be assigned a garbage value.

In the last program, the statement `y = 5 + (++x);` uses variable `x` and `y`. The variable `x` and `y` are not initialized. The variable `y` has no effect on the operation because it is at the left side and the result of expression of right side is assigned to it. The variable `x` holds garbage value. Integer 5 is added to garbage value of `x` and stored in the variable `y`. Hence the value of `y` displayed is 1134.

Initialization of variable can be done at the place where they are declared or anywhere in the program before their use. The variable can be initialized using assignment operator or input statement such as `cin`, `get( )` etc.

Figure 3.4 indicates the syntax and example of initialization of variable.



**Fig.3.4** Initialization of variables

#### Syntax:

`data-type variable_name = value;`

#### Example:

```
int k=5;
```

In the above example, variable *k* of integer type is declared. The value 5 is stored in it.

```
int a,b,c,d;  
a=b=c=d=5;
```

In the above example, all the variables *a*, *b*, *c* and *d* are initialized to 5. An example is illustrated below.

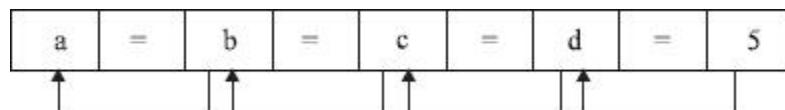
### 3.4 Write a program to initialize more than one variable at a time.

```
# include <conio.h>  
# include <iostream.h>  
void main( )  
{  
    int a,b,c,d;  
    a=b=c=d=5;  
    clrscr( );  
    cout <<"\n a=<<a <<" b=<<b <<" c=<<c <<" d=<<d;  
}
```

#### OUTPUT

**a=5 b=5 c=5 d=5**

**Explanation:** In the above program, variables *a*, *b*, *c* and *d* are declared. In the next statement the entire four variables are initialized with 5. The value 5 is first assigned to variable *d* then the value of *d* is assigned to *c*, *c* is assigned to *b* and finally *b* is assigned to *a*. Figure 3.5 simulates this assignment.



**Fig.3.5** Initialization of variables

### 3.5 Write a program to prove that a variable can be initialized for more than one time.

```
#include<iostream.h>  
#include<conio.h>  
void main( )  
{  
    clrscr( );  
    int x=10;  
    cout <<"\n x= "<<x;  
    x=30;  
    cout<<"\n x= "<<x;  
}
```

#### OUTPUT

**x= 10  
x= 30**

**Explanation:** In the above program, integer variable *x* is declared and initialized with 10. The `cout` statement displays the value of *x* on the screen. Again the variable *x* is initialized with 30. The previous value 10 is replaced with 30. The second `cout` statement displays the value of *x* equal to 30.

### 3.6 DYNAMIC INITIALIZATION

The declaration and initialization of variable in a single statement at any place in the program is called as dynamic initialization. The dynamic initialization is always accomplished at run-time i.e., when program execution is in action. Dynamic means process carried out at run-time, for example, dynamic initialization, dynamic memory allocation etc. The C++ compiler allows declaration and initialization of variables at any place in the program. In C, initialization of variables can be done at any place but the variable must be declared at the beginning of the program as illustrated in the following program:

#### 3.6 Write a program in C to demonstrate declaration and initialization of a variable.

```
# include <conio.h>
# include <stdio.h>
void main( )
{
    int r;
    float area;
    clrscr( );
    printf ("\n Enter radius : ");
    scanf ("%d",&r);
    area=3.14*r*r;
    printf ("\n Area =%g",area);
}
```

#### OUTPUT

Enter radius : 4

Area =50.24

**Explanation:** The above program is executed with C compiler. In this program, the variables area and r are declared at the beginning because in C, declaration is compulsorily done at the beginning. The multiplication of 3.14 and variable r is assigned to variable area. This assignment is done inside the program. The above program demonstrates that in C, variable declaration is done at the beginning and initialization can be done at any place in the program.

#### 3.7 Write a program in C++ to demonstrate dynamic initialization.

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

#### OUTPUT

Enter radius : 3

Area =28.26

**Explanation:** In program 3.7, the variable r and area are declared inside the program. The declaration and initialization of a variable area is done in single statement inside the program. Consider the following statement:

```
float area=3.14*r*r;
```

In the above statement float variable area is declared and product of  $3.14 * r * r$  is assigned to area. The assignment is carried out at run-time. Such type of declaration and initialization of a variable is called as dynamic initialization.

### 3.8 Write a program to calculate the length of string. Use run-time declaration and initialization of variables.

```
# include <iostream.h>
# include <string.h>
# include <conio.h>
main( )
{
    clrscr();
    char name[15];
    cout <<"Enter Your Name :";
    cin >>name;
    int len=strlen(name);
    cout <<"The length of the string is :"<<len;
    return 0;
}
```

#### OUTPUT:

Enter Your Name : Santosh

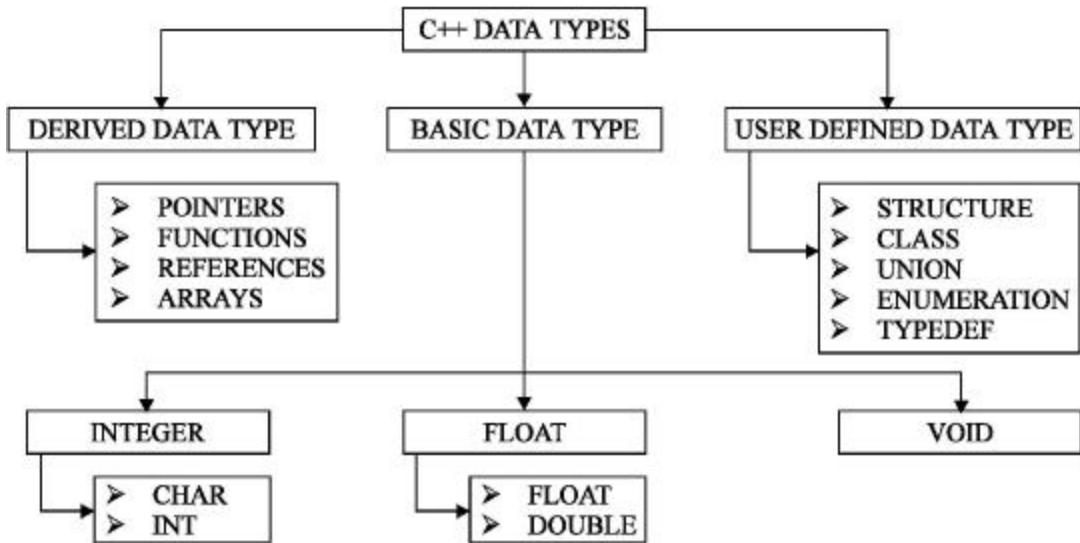
The length of the string is :7

**Explanation:** In the above program, the `cin` statement reads the string through the keyboard. The `strlen()` function is used to determine the length of the given string. The `strlen()` function calculates the length of the string and returns the length to variable `len`. The variable `len` is declared and value returned by `strlen()` is assigned to variable `len`. To avoid separate statements for declarations and initializations, both statements are combined in one statement. Declaration and initialization has been carried out in one statement `int len=strlen(name)`.

## 3.7 DATA TYPES IN C++

Data is a collection of characters, digits, symbols etc. It is used to represent information. The data are classified in various types. [Figure 3.6](#) indicates all data types. C++ data types can be classified in the following categories:

- (a) Basic data type
- (b) Derived type
- (c) User-defined type
- (d) Void data type



**Fig.3.6** C++ data types

### 3.8 BASIC DATA TYPE

The basic data types supported by C++ are described with their size in bytes and ranges in [Table 3.5](#). [Figure 3.7](#) shows data types and their sizes.

**Table 3.5** Basic data types supported by C++ with size and range

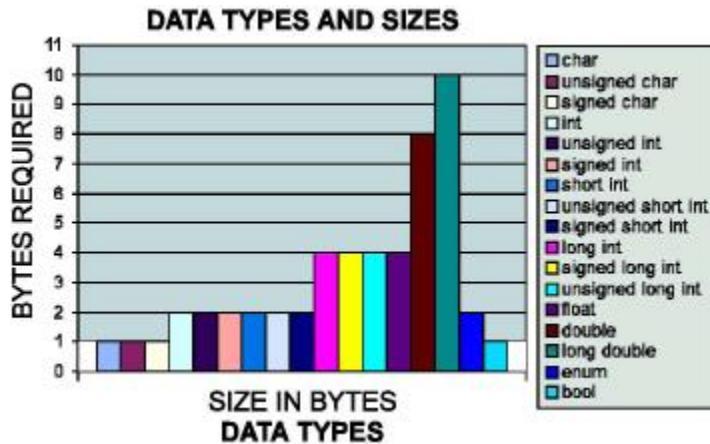
DATA TYPE	SIZE IN BYTES	RANGE
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767

unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
signed long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E +4932
enum	2	-32768 to 32767
bool	1	true / false

### 3.9 DERIVED DATA TYPE

The derived data types are of following types:

- (1) POINTERS
- (2) FUNCTIONS
- (3) ARRAYS
- (4) REFERENCES



**Fig.3.7** Data types and their sizes

## (1) POINTERS

A pointer is a memory variable that stores a memory address. Pointer can have any name that is legal for other variable and it is declared in the same fashion like other variable but it is always denoted by '\*' operator.

```
int *x;
float *f;
char *y;
```

In the first statement 'x' is an integer pointer and it tells to the compiler that it holds the address of any integer variable. In the same way f is a float pointer that stores the address of any float variable and 'y' is a character pointer that stores the address of any character variable.

### 3.9 Write a program to use pointers.

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

void main( )
{
    clrscr( );
    int x=2,*p;
    cout <<"\n Address of x = "<<(unsigned)&x;
    p=&x;
    cout <<"\n Value of x ="<<*p;
}
```

### OUTPUT

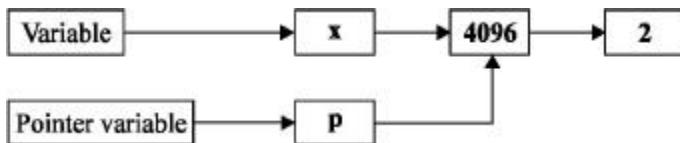
**Address of x = 4096**

**Value of x=2**

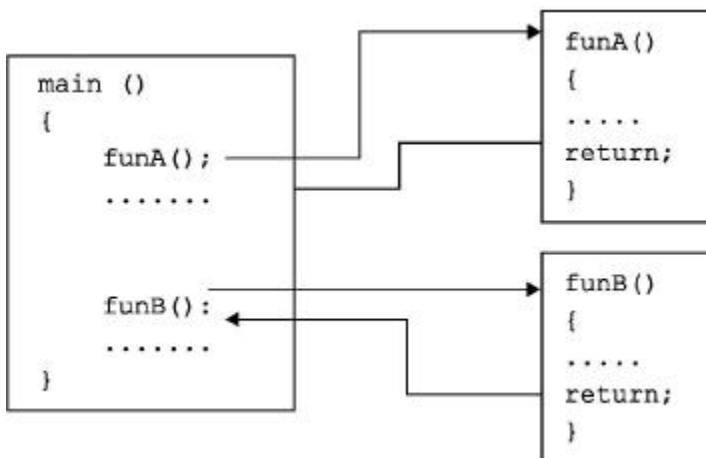
**Explanation:** In program 3.9, x is an integer variable and \*p is an integer pointer. The first cout statement displays the address of variable x. The address of x is assigned to pointer p. The pointer variables are always used to store address of another variable. The second statement displays the value of x using pointer p. [Figure 3.8](#) explains pointers.

## (2) FUNCTIONS

A function is a self-contained block or a sub-program of one or more statements that perform a special task when called. The C++ functions are more civilized than C. It is possible to use the same name with multiple definitions called as function overloading. [Figure 3.9](#) illustrates the function. In [Figure 3.9](#) funA( ) and funB( ) are two user-defined functions. These are invoked from the body of main. After execution of the funA( ) and funB( ) the program control returns back to the calling function main( ).



**Fig.3.8** Pointers



**Fig.3.9** Functionsx

A simple program on function is described below.

### 3.10 Write a program to demonstrate user-defined functions.

```

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

void main( )
{
    clrscr( );
    void show(void);
    show( );
}

void show( )
{
    cout <<"\n In function show( )";
}
  
```

#### OUTPUT

In function show( )

**Explanation:** In the above program, the function `show( )` is defined. The function body contains only one `cout` statement. When the function is executed, a message is displayed. For more details of functions refer to [chapter 5 “Functions in C++”](#).

### (3) ARRAYS

Array is a collection of elements of similar data types in which each element is located in separate memory location. For example,

```
int b[4];
```

The above statement declares an array `b [ ]` which can hold four integer values. The following program illustrates use of array.

#### 3.11 Write a program to declare, initialize an array. Display the elements of an array with their addresses.

```
# include <conio.h>
# include <iostream.h>
void main( )
{
clrscr( );
int b[4]={2,4,3,7};

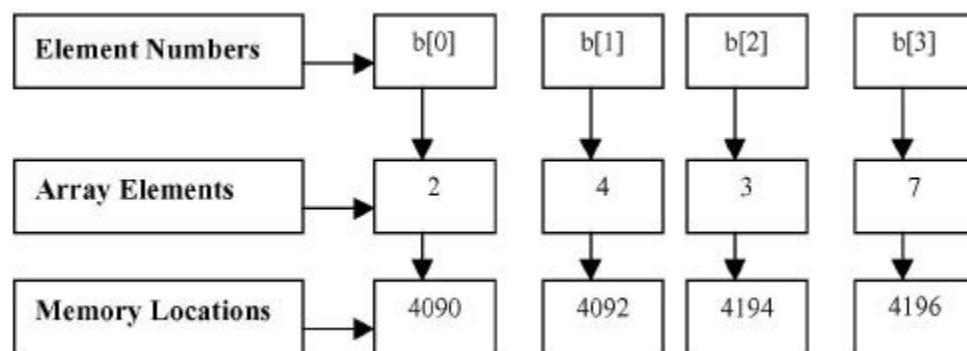
cout <<"b[0] Value = "<<b[0]<<" Address : "<<(unsigned)&b[0]<<endl;
cout <<"b[1] Value = "<<b[1]<<" Address : "<<(unsigned)&b[1]<<endl;
cout <<"b[2] Value = "<<b[2]<<" Address : "<<(unsigned)&b[2]<<endl;
cout <<"b[3] Value = "<<b[3]<<" Address : "<<(unsigned)&b[3]<<endl;
}
```

#### OUTPUT

```
b[0] Value = 2 Address : 4090
b[1] Value = 4 Address : 4092
b[2] Value = 3 Address : 4094
b[3] Value = 7 Address : 4096
```

**Explanation:** In the above program, an array `b [ 4 ]` is declared and initialized.

The `cout` statement displays the array elements with their addresses. The `b [ 0 ]` refers to first element; `b [ 1 ]` refers to second element and so on. [Figure 3.10](#) shows element numbers, elements and addresses.



**Fig. 3.10** Array elements in memory

### (4) REFERENCES

C++ reference types, declared with & operator are nearly identical but not exactly same to pointer types. They declare aliases for object variables and allow the programmer to use variable by reference. This reference type is illustrated in section 3.20.

### 3.10 USER-DEFINED DATA TYPE

User-defined data types are of the following types:

- (1) Structures and classes
- (2) Union
- (3) Enumerated data type

#### (1) STRUCTURE AND CLASSES

Keyword `struct`

The `struct` is a keyword and used to combine variables of different data types into a single record.

**Syntax:**

```
struct[< struct name >
{
<data-type> <variable-name1, variable-name, 2 > ;
<data-type> <variable-name3, variable-name, 4>

} <structure variable declarations>;
```

`struct name` : An optional tag name that defines the structure type.

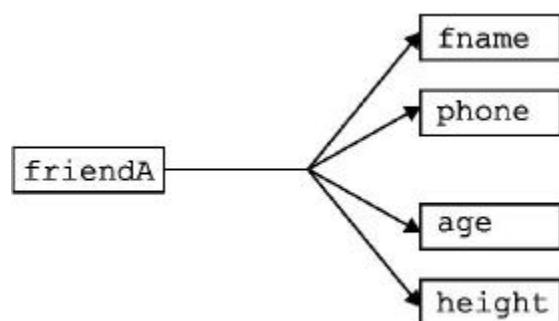
`structure variables` : These are member variables and hold data.

Though `struct name` and `structure variables` are noncompulsory, one of them should be present. Member variables in the structure are declared by naming a `<data-type>`, followed by one or more `<variable-name>` separated by commas. A semicolon can separate variables of different data types.

**Example:**

```
struct my_friend
{
    char fname [80], phone[80];
    int age, height;
} friendA ;
```

The structure `my_friend` defines a variable containing two strings (`fname` and `phone`) and two integers (`age` and `height`) as shown in Fig. 3.11. To access elements in a structure, we use a record selector (`.`) called as dot operator. For example,  
`strcpy(friendA.fname, "Sachin");`



**Fig.3.11** Structure and its elements

### 3.12 Write a program to declare struct object, initialize it and display the contents.

```
#include<iostream.h>
#include<conio.h>
void main( )
{
    clrscr( );

    struct my_friend
    {
        char *fname;
        int phone, age, height;
    } A ;
        A.fname="Bharat";
        A.phone=26251;
        A.age=22;
        A.height=4.5;
    cout <<"\n Contents of object A";
    cout <<"\nName : "<<A.fname;
    cout <<"\nPhone : "<<A.phone;
    cout <<"\nAge : "<<A.age;
    cout <<"\nHeight : "<<A.height;
}
```

#### OUTPUT

##### Contents of object A

Name : Bharat

Phone : 26251

Age : 22

Height : 4

**Explanation:** In the above program, `struct my_friend` is defined with four member variables. Identifier `A` is an object of `struct my_friend`. The initialization of data members of struct is done using dot operator with object `A`. The `cout` statements display contents of object `A`.

**Keyword class**

The `class` is a new keyword introduced in C++. Its use is same as `struct` keyword.

Example on classes are explained in chapter 6, Classes and Objects. To declare a class following syntax is used:

**Syntax:**

```
<classkeyword> <class- name> [<:baseclasslist>]
{<member variable list>}
```

**Class keyword:** It is one of the keywords `class`, `struct`, or `union`.

**Class-name:** It can be any unique name inside its scope.

**Baseclasslist:** If the class is derived class, then it follows the list of the base class (es). It is optional.

**Member variable list:** Defines the data member variables and member functions.

**Example:**

```
class circle
{
    int radius;           // data member
    int area (void);     // member function
```

```
};
```

## (2) Union

A union is same as compared to a struct, the only difference is that it lets the user to declare variables that share same memory space.

### **Syntax:**

```
union [<union name>]
{
<data-type> <variable names>;
} [<union variables name>];
```

### **Example:**

```
union charorint
{
    char c;
    int i;
} number;
```

C++ will allocate sufficient storage in **union variable number** to hold the big element in the union. The union member variable number.c and number.i uses the same memory location in memory. In this way, writing into one will replace the other. Member variables of a union are accessed in the same way as a struct.

**Anonymous Unions** An anonymous union does not contain tag name. Elements of such union can be accessed without using tag name. Consider the following example:

```
union
{
    int k;
    float j;
};
```

Both the member variables of union have the same memory location. They can be accessed as per the following:

```
k=20;
j =2.2;
```

The declaration should not declare a variable of the above union type. Following program illustrates the use of anonymous union:

### **3.13 Write a program to declare anonymous union and access its elements.**

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

main( )
{
    clrscr( );
    union
    {
        int k;
        float f;
    };
    f=3.1;
    k=2;

    cout <<"\n k = "<<k;
    printf ("\n f = %.1f",f);
    return 0;
}
```

## OUTPUT

k = 2

f = 3.1

**Explanation:** In program 3.13, **anonymous union** is declared. The union has two data member variables, *k* as integer variable and *f* as a float variable. The union has no tag name; hence it is called as **anonymous union**. The member variable of such a union can be accessed directly like normal variable. Both the variables hold the same memory location.

## (3) Enumerated Data Type

The `enum` is a keyword. It is used for declaring enumeration data types. The programmer can declare new data type and define the variables of these data types that can hold. For example, the user can define the material as new data type. Its variable may be solid, liquid or gas. Thus three values are restricted for this new data type. These enumeration data types are useful in `switch( ) case` statement.

The syntax for enumerated data type is given below. It uses a keyword `enum`.

```
enum logical { false,true};  
enum logical {true=2, false=4};  
enum components{solid,liquid,gas};
```

This statement declares a user-defined data type. The keyword `enum` is followed by the tag name `logical`. The enumerators are the identifiers `false` and `true`. Their values are constant unsigned integers and start from 0. The identifier `false` refers to 0 and `true` to 1. The identifiers are not to be enclosed with quotation marks. Please also note that integer constants are also not permitted. We can also start the constants as given in the second statement. In the second statement `true` refers to 2 and `false` refers to 4. In the third statement, the `components` is the user-defined data type and the variables attached to it are `solid`, `liquid` and `gas`.

The ANSI C++ and Turbo C++ allow us to declare variables of `enum` type.

```
logical =N // N is of the type logical  
logical F=false // valid  
logical TT=1 // invalid in c++  
logical TT=(logical) 1 // valid  
int k=true; // valid
```

We can also define `enum` without tag name. Consider the following example:

```
enum{yes, no};
```

Here, `yes` is 0 and `no` is 1 and can be used as `int answer=yes`.

## 3.14 Write a program to declare `enum` data type and display their values.

```
# include <iostream.h>  
#include <conio.h>  
void main( )  
{  
    clrscr( );  
    enum logical {false,true};  
    cout <<"true : "<<true <<" false : "<<false;  
}
```

## OUTPUT

true : 1 false : 0

**Explanation:** In program 3.14, enum data type logical is declared with two values i.e. false and true. The false contains value 0 and true contains the value 1. The cout statement displays the contents of true and false. True means 1 and false 0.

### 3.11 THE void DATA TYPE

The void type is added in ANSI C. It is also known as Empty data type. It can be used in two ways.

(a) When specified as a function return type, void means that the function does not return a value.

```
void message(char *name)
{
printf("Hello, %s.", name);
}
```

Here, message( ) is a void function. The keyword void is preceded by the function name. This function when executed displays only message and does not return any value to the calling function.

(b) The void keyword is also used as argument for function. When found in a function heading, void means that the function does not take any arguments.

```
int fun(void)
{
return 1;
}
```

Here, the function fun( ) does not require any argument. It returns an integer value.

(c) When specified as a function return type and in function heading i.e., the function neither returns a value nor requires any argument.

```
void fun(void);
```

The above function neither returns a value nor requires any argument.

### 3.12 TYPE MODIFIERS

The keywords signed, unsigned, short, and long are type modifiers. A type modifier changes the meaning of the base data type to produce a new data type. Each of these type modifiers is applicable to the base type int. The modifiers signed and unsigned are also applicable to the base type char. In addition, long can be applied to double data type. When the base type is absent from a declaration, int is supposed.

**Examples:**

```
long          l;      // int is implied
unsigned char c;
signed int     s;      // signed is default
unsigned long int u;  // int OK, not necessary
```

### 3.15 Write a program to declare variable with type modifiers, initialize them and display their contents.

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

void main( )
{
```

```

clrscr( );
short t=1;
long k=54111;
unsigned u=10;
signed j=-10;

cout <<"\n t="<<t;
cout <<"\n k="<<k;
cout <<"\n u="<<u;
cout <<"\n j="<<j;
}

```

### OUTPUT

**t=1**  
**k=54111**  
**u=10**  
**j=-10**

**Explanation:** In the above program, variables *t*, *k*, *u*, and *j* are declared. When the variable is declared with the keyword `short` or `signed`, it is of type short-signed integer.

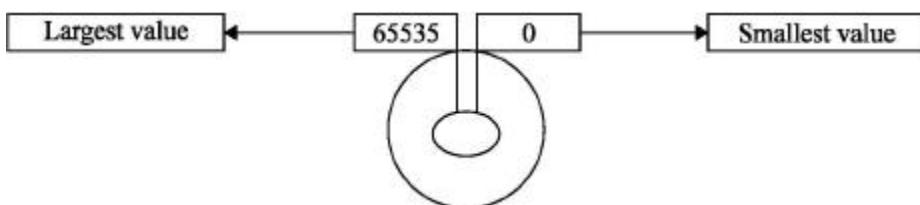
The `long` type of variable can store large integer. The `unsigned` type of variable can store only positive values upto 65535.

### 3.13 WRAPPING AROUND

When a variable contains a value, which goes over the range provided by compiler, the compiler won't flag any error in such cases and wrapping around takes places. For example, the range of unsigned integer is 0 to 65535. Negative values and values greater than 65535 are not permitted. If the variable of this type is assigned a value that is not in the above range, wrapping takes place as shown in [Fig. 3.12](#).

```
Unsigned int x=65536;
```

In the above statement, the unsigned variable *x* is assigned a value 65536 which is greater by one than the range. In this case, difference between assigned value and total numbers that comes under range are considered. For example, the total number that comes under this range is 65536 (0 to 65535). The value assigned is 65536. The difference is zero (65536-65536). The *x* contains 0. The same rule is applied to all other basic data type.



**Fig.3.12** Value wrapping around

### 3.16 Write a program to demonstrate wrapping around with unsigned integer.

```

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

void main( )
{

```

```

    clrscr();
    unsigned int x=65536;
    cout <<" x ="<<x;
}

```

## **OUTPUT**

**x=0**

**Explanation:** In the above program, unsigned integer variable x is declared and initialized with 65536. The value assigned is beyond the range. Hence, wrapping around takes place. The value of x displayed is zero.

### **3.17 Write a program to demonstrate wrapping around with unsigned integer variable.**

```

# include <conio.h>
# include <iostream.h>
void main ( )
{
    clrscr();
    unsigned int x=0;
    --x;
    cout <<"x ="<<x;
    x+=2;
    cout<<"\nx = "<<x;
}

```

## **OUTPUT**

**x=65535**

**x=1**

**Explanation:** In the above program, unsigned integer variable x contains value 0. The variable x is decremented. As unsigned integer never takes negative values, wrapping around takes place and x holds nearest positive value i.e., 65535. The variable is again incremented by two. In this case the value of x exceeds the limit and wrapping takes place. The value of x displayed is 1.

## **3.14 TYPECASTING**

C/C++ supports data types such as integer, character, floating, double precision, floating point etc. Some of them are specified as `int`, `short int`, and `long int`. Typecasting is essential when the values of variables are to be converted from one type to another type. C++ allows implicit as well as explicit type conversion.

### **(1) EXPLICIT TYPECASTING**

Sometimes errors may be encountered in the program while using implicit typecasting. It is preferred to use explicit type conversion process. The desired type can be achieved by typecasting a value of particular type. The explicit type conversion is done using `typecast` operator. The compiler is instructed to do type conversion using `typecast` operator. The following are the syntaxes of typecasting in C and C++.

```
(data-type name) expression // C style syntax
data-type name (expression) // C++ style syntax
```

#### **Examples:**

```
x=(float)5/2;    // C style
x=5/float(2);    // C++ style
```

In the C style example, the target data type name is enclosed in parenthesis whereas in C++ style example, the argument is enclosed in parenthesis.

In the example `(float) 5/2`, the expression `5/2` is declared as `float` type. Hence, the result of expression turns to `float` type. Consider the given example.

```
float x;  
x=(int) 5/2.0;
```

The above expression returns the value 2.5, even if an `int` typecast operator is used. The typecast operator attempts to make the expression `int` type, but the expression contains `float` type operand. Hence it is considered as `float` type expression and returns the result of `float` type.

```
x= int (5/2.0);
```

In the above example, the expression is enclosed in the parenthesis. The expression is solved first and the result obtained is used for typecasting. The expression `5/2.0` returns 2.5 and the typecast operator `int` converts it to integer.

### 3.18 Write a program to explain new style of typecasting in C++.

```
#include <iostream.h>  
#include<constream.h>  
  
void main( )  
{  
    clrscr( );  
    cout<<(int)45.4;           // C style  
    cout<<"\n"<<int(17.78); // C++ style  
}
```

#### OUTPUT

45

17

**Explanation:** In the above program, the `float` values are converted to `integer` type and then displayed. The compiler using the typecast operator `int` explicitly does the type conversion task.

The above example converts higher type to lower type data. The `float` type is converted to `integer` type. It is also possible to convert in reverse order i.e., `integer` to `float`. The example is illustrated below.

### 3.19 Write a program to convert float type data to integer. Use the syntaxes of both C and C++.

```
#include <iostream.h>  
#include<constream.h>  
void main ( )  
{  
    clrscr( );  
    int a,b,c;  
    float x,y;  
    a=19;  
    b=10;  
    c=3;  
    x=(float) a/c; // C style  
    cout <<"\n x="<<x;  
    y=float(b)/c; // C++ style  
    cout <<"\n y="<<y;  
    getch( );
```

```
}
```

**OUTPUT:**

**x=6.333333**  
**y=3.333333**

**Explanation:** In the above program, the integer values are converted to `float` type and the results are displayed as the output. The compiler using the `typecast` operator `float` explicitly does the type conversion task.

### 3.20 Write a program to show difference of typecasting between C and C++.

```
#include <iostream.h>
#include<constream.h>
void main( )
{
    clrscr( );
    float x,y;
    x=(float) 5/2; // C style
    cout <<"\n x="<<x;
    y=float(9)/2; // C++ style
    cout <<"\n y="<<y;
}
```

**OUTPUT**

**x=2.5**  
**y=4.5**

**Explanation:** The above example uses typecasting methods of C and C++. The `x` and `y` are two `float` variables. The expression `5/2` is evaluated and before storing in variable `x`, the result obtained is converted to `float`. In C style, the target data type name is enclosed in parenthesis. In C++ style, one of the argument of expression is enclosed in parenthesis preceded by target data type name. The expression `y=float (9)/2;` can be written in the following ways and produces the same result:

```
y=float (9)/2;
y=9/float(2);
```

Here, the `float` keyword is a typecast operator. The typecast operator instructs the compiler explicitly to make type conversion.

## (2) IMPLICIT TYPE CONVERSION

The type conversion is carried out when the expression contains different types of data items. When the compiler carries such type conversion itself by using inbuilt data types in routines then it is called implicit type conversion. The variable of lower data (small range) type when converted to higher type (large range) is known as promotion. When the variable of higher type is converted to lower type, it is called as demotion. [Table 3.6](#) describes various rules that a compiler follows.

**Table 3.6** Implicit type conversion rules

Argument1	Argument 2	Output
char	int	int
int	float	float
int	long	long
double	float	double
int	double	double

long int	double unsigned	double unsigned
-------------	--------------------	--------------------

3.21 Write a program to demonstrate implicit type conversion.

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

void main( )
{
    clrscr( );
    int j=2.54;
    int k='A';
    char c=87.5;
    cout <<" j= "<<j;
    cout<<endl<<" k= "<<k;
    cout<<endl<<" c= "<<c;
}
```

#### OUTPUT

j= 2  
k= 65  
c= W

**Explanation:** The above program works as described in [Table 3.7](#).

**Table 3.7** Examples of implicit type conversion

Expression	Output
int j=2.54	A float value is assigned to integer variable. Here, 2 is assigned to j and fractional part will get vanished.
int k='A';	A char type value is assigned to integer variable. Here, ASCII value of ?' (65) is assigned to k. Here, char to int conversion takes place.
char c=87.5	A float value is assigned to char variable. Here, fractional part is ignored and W ASCII equivalent of 87 is assigned to variable c.

## 3.15 CONSTANTS

The constants in C++ are applicable to the values which do not change during execution of a program. C++ supports various types of constants including integers, characters, floating, and string constants. C++ has two types of constants; literal and symbolic.

### (1) LITERAL CONSTANT

A literal constant is directly assigned to a variable. Consider the following example:

```
int x=5 ;
```

where `x` is a variable of type `int` and `5` is a literal constant. We cannot use `5` to store another integer value and its value cannot be altered. The literal constant doesn't hold memory location. C++ supports literal constants as listed in [Table 3.8](#) with examples.

**Table 3.8** Literal constants

Example	Constant type
542	Integer constant
35.254	Floating point constant
0x54	Hexadecimal integer constant
0171	Octal integer constant
'C'	Character constant
"C plus plus"	String constant
L "xy"	Wide-character constant

## (2) SYMBOLIC CONSTANT

A symbolic constant is defined in the same way as variable. However, after initialization of constants the assigned value cannot be altered.

The constant can be defined in the following three ways:

- (1) # define
- (2) The `const` keyword
- (3) The `enum` keyword

(1) The `# define` preprocessor directive can be used for defining constants.

```
# define price 152
```

In the above example, `price` symbolic constant contains `152` and here it is not mentioned whether the type is `int`, `float` or `char`. Every time when the preprocessor finds the word `price`, substitutes it with `152`.

(2) We can also define a constant using `const` keyword.

```
const int price=152;
```

The above declaration defines a constant of type `int`. Here, data type is strictly maintained while doing operation. The value of constant cannot be changed at run-time.

### 3.22 Write a program to define `const` variable using `const` keyword.

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

```
#include<constream.h>

void main( )
{
    clrscr( );
    const Sunday=0;
    const Monday=1;
    int c;

    cout <<"\n Enter day (0 or 1) :";
    cin>>c;
    if (c==Sunday && c!=Monday)
        cout<<"\n Holiday";
    else
        cout<<"\n Working day";
}
```

## **OUTPUT**

**Enter day (0 or 1) :0**

### **Holiday**

**Explanation:** In the above program, Sunday and Monday are constants initialized with 0 and 1. The data type name is not mentioned. In such case `int` type is considered. The user enters the choice either 0 or 1 and it is stored in the variable `c`. If condition statement compares the value of variable `c` with constants Sunday and Monday, one of the messages as given in the programs is displayed depending upon entered value.

(3) Constants can be defined using enumeration as given below:

### **Example:**

```
enum {a, b,c};
```

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

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

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

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

## **3.16 CONSTANT POINTERS**

C++ allows us to create constant pointer and also pointer to constant. Consider the following example:

### **(1) CONSTANT POINTER**

It is not possible to modify the address of the constant pointer.

```
char * const str="Constant";
```

In the above example, it is not possible to modify the address of the pointer `str`. Thus, the operations like the following will generate error:

```
// str="san"; // cannot modify a constant object
// ++str; // cannot modify a constant object
```

The following program won't work and will generate error message "cannot modify a constant object".

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

void main( )
{
```

```

clrscr( );

    char * const str="Constant";
    str="new constant";
    cout <<str;
}

```

## (2) POINTER TO CONSTANT

If pointer is declared to constant, we can change the value by using actual variable that is not possible using pointer.

```
int const *pm=&k;
```

In the above example, pm is declared as pointer to constant. The operations given below are possible and invalid:

```

// k=5; // possible
// *pm=5           // cannot modify the constant object
// pm++;          // possible
// ++ *pm;         // cannot modify a constant object
// *pm=5;          // cannot modify a constant object

```

**3.23 Write a program to declare constant pointer. Modify the contents of the pointer.**

```

# include <iostream.h>
# include <constream.h>

void main( )
{
    clrscr( );
    int k=10;
    int const *pm=&k;
    k=40;
// *pm=20; // invalid operation
    cout<<*pm;
}

```

### OUTPUT

**40**

**Explanation:** In the above program, variable k is an integer pointer. The variable \*pm is declared as constant integer pointer. The value of k cannot be changed through constant pointer because constants are fixed entity. Hence, operation such as \*pm=5, \*pm++ are invalid operations. These operations attempt to change the contents of constant which is never possible. Changing the value of variable k can change the contents. The variable k is not constant but its pointer is constant. Hence, the constant pointer can be involved to modify the value.

## (3) POINTER AND VARIABLE BOTH CONSTANTS

```
const char * const p="ABC";
```

In the above example, both the pointer and variable are constants. Hence it is not possible to change the value and address of the pointer.

### 3.17 OPERATORS IN C AND C++

Operator is an instruction to the compiler or interpreter, specified by a single or double symbol to perform certain operations with constants. Different types of operators are shown in [Fig. 3.13](#).

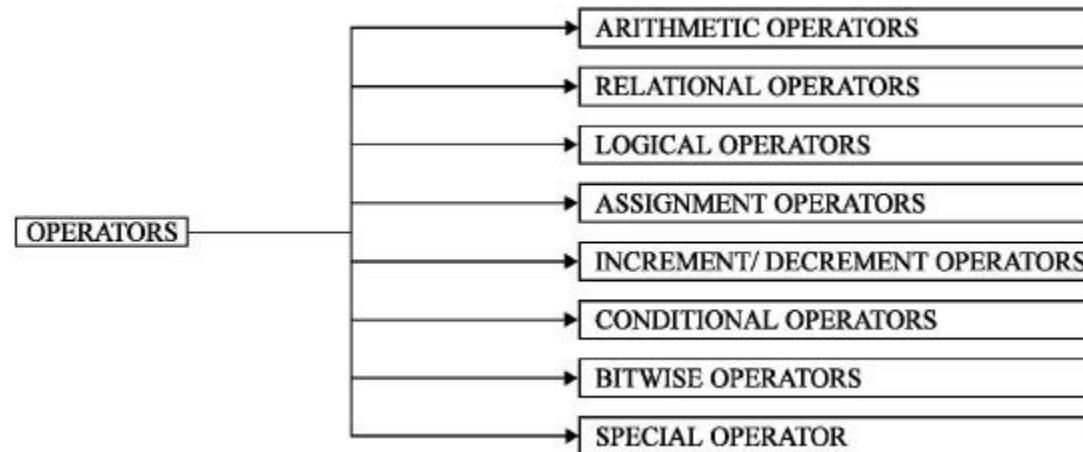
**Example:**

5+10

Here, 5 and 10 are constants. The symbol ‘+’ is an operator that indicates the operation to be performed and performs addition of numbers and is a single operator.

```
int x=5;  
++x;
```

In the above example the operator ++ is an increment operator. This operator adds one to the value of operand. The different types of operators with their symbols are described in Table 3.9.



**Fig. 3.13** Types of operators

**Table 3.9** Types of operators

Types of Operators	Symbolic Representation
Arithmetic Operators	+, -, *, / and %
Relational Operators	>, <, ==, !=, <= and !=
Logical Operators	&&,    and !
Increment and Decrement Operators	++ and --
Assignment Operators	=
Bitwise Operators	&/ !, *, ~ and ~
Special Operators	,
Conditional Operators	? ;

C++ supports all the operators of C. C++ also introduces few new operators as listed in Table 3.10.

**Table 3.10** Operators in C++

Operator	Description
<<	Insertion operator
>>	Extraction operator
::	Scope access (or resolution) operator
::*	Pointer to member decelerator
->*	Deference pointers to pointers to class members
. *	Deference pointers to class members
delete	Memory release operator
New	Memory allocation operator

The scope access (or resolution) operator :: (two semicolons) allows you to access a global (or file duration) name even if a local hides redecoration of that name. The . \* and ->\* operators are deference pointers to class members and pointers to pointers to class members. The insertion and extraction operator are used with cout and cin objects to perform I/O operations. The new and delete operators are used to allocate and de-allocate memory.

### 3.18 PRECEDENCE OF OPERATORS IN C++

C++ operators are classified into 16 groups as shown in [Table 3.11](#). The #1 group has the first (highest) precedence, group #2 (Unary operators) takes second precedence, and so on to the Comma operator, which has lowest precedence.

The operators within each category have equal precedence.

The Unary (group #2), Conditional (group #14), and Assignment (group #15) operators associate right-to-left; all other operators associate left-to-right.

**Table 3.11** Precedence of operators in C++

# Group	Operator	Operation
1. Top	( )	Function call
	[]	Array subscript
	->	C++ indirect component selector

	::	C++ scope access/resolution
	.	C++ direct component selector
2. Unary	!	Logical negation (NOT)
	~	Bitwise (1's) complement
	+	Unary plus
	-	Unary minus
	++	Pre-increment or post-increment
	--	Pre-decrement or post-decrement
	&	Address
	*	Indirection
	size of	Returns size of operand, in bytes
	new	Dynamically allocates C++ storage
	delete	Dynamically de-allocates C++ storage
3. Multiplicative	*	Multiply
	/	Divide
	%	Remainder (modulus)
4. Member Access	.*	C++ deference

	<code>-&gt;*</code>	C++ deference
5. Additive	<code>+</code>	Binary plus
	<code>-</code>	Binary minus
6. Shift	<code>&lt;&lt;</code>	Shift left
	<code>&gt;&gt;</code>	Shift right
7. Relational	<code>&lt;</code>	Less than
	<code>&lt;=</code>	Less than or equal to
	<code>&gt;</code>	Greater than
	<code>&gt;=</code>	Greater than or equal to
8. Equality	<code>==</code>	Equal to
	<code>!=</code>	Not equal to
9. Bitwise	<code>&amp;</code>	Bitwise AND
	<code>^</code>	Bitwise XOR
	<code> </code>	Bitwise OR
10. Logical	<code>&amp;&amp;</code>	Logical AND
	<code>  </code>	Logical OR
11. Conditional	<code>?:</code>	( <code>a ? x : y</code> means “if <code>a</code> then <code>x</code> , else <code>y</code> ”)

12. Assignment	=	Simple assignment
	*=	Assign product
	/=	Assign quotient
	%=	Assign remainder (modulus)
	+=	Assign sum
	- =	Assign difference
	&=	Assign bitwise AND
	^=	Assign bitwise XOR
	=	Assign bitwise OR
	<<=	Assign left shift
	>>=	Assign right shift
13. Comma	,	Evaluate

Most of the operators in this table can be overloaded except the following:

- . C++ direct component selector
- .\* C++ deference
- :: C++ scope access/resolution
- ??: Conditional

### PRECEDENCE OF \* AND [ ] OPERATORS

In C++, the statements `* x[4]` and `(* x) [4]` are not the same because the `*` operator has lower precedence than the `[ ]` operator. Consider the following examples:

(a) `int *arr[5];`

The above statement declares an array of five pointers and the following operation is invalid because the array name is itself an address and it is a constant. Hence, it cannot be changed.

`arr++;`      or    `++arr;`

(b) int (\* arr) [5]

The above declaration declares a pointer to an array of five elements. Hence the operations such as `arr++` and `++arr` are not supported. The following program explains both these points:

### 3.24 Write a program to declare a pointer to array and display the elements.

```
# include <iostream.h>
# include <conio.h>
void main( )
{
    void display(int [] [3],int);
    clrscr( );

    int a[3][3]={ {11,22,33},
                  {44,55,66},
                  {77,88,99}
                };
    display (a,3);
}

void display (int x[] [3], int k)
{
    int (*d) [3];
    d=x;

    for (int g=0;g<k;g++)
    {
        for (int h=0;h<3;h++)
        cout <<d[g] [h] <<"\t";

        cout<<"\n";
    }
}
```

#### OUTPUT

```
11 22 33
44 55 66
77 88 99
```

**Explanation:** In the above program, an integer array `a [ 3 ] [ 3 ]` is declared and initialized. The base address of array and number of rows are passed to function `display()`. In function `display()`, `d` is a pointer. The base address received by the variable `x` is assigned to pointer `d`. Using nested for loops, the elements of array are displayed.

## 3.19 REFERENCING (&) AND DEREFERENCING (\*) OPERATORS

The `&` and `*` operators are used for referencing and dereferencing. The `&` symbol is also used in C++ to define reference types, and as a bitwise AND operator. We can also use the asterisk as an operator to deference a pointer and as the multiplication operator.

### REFERENCING OPERATOR (&)

The referencing operator is used to define referencing variable. A reference variable prepares an alternative (alias) name for previously defined variable. The syntax of the referencing operator is given below.

**Syntax:**

Data-type & reference variable name = variable name

**Example:**

```
int qty=10;  
int & qt=qty;
```

Here, qty is already declared and initialized. The second statement defines an alternative variable name i.e., qt to variable qty. If both printed variables display the same value, any change made in one of the variable causes change in both the variables.

```
qt=qt*2;
```

Now, contents of qt and qty will be 20.

Note that the token & is not an address operator. The declaration int & indicates reference to data type int.

## PRINCIPLES FOR DECLARING REFERENCE VARIABLE

- (1) A reference variable should be initialized.
- (2) Once a reference variable is declared, it should not refer to any other variable. Once the actual variable and reference variable are connected they are tied jointly congenitally.
- (3) The reference variable can be created referring to pointer variable. The declaration would be as given below:

```
char * h="C++";  
char *&q=h;
```

- (4) A variable can contain various references. Modifying the value of one of them results in a change in all others.

- (5) Array of references is not allowed.

3.25 Write a program to declare reference variable to another variable. Display the assigned value using both the variables.

```
# include <iostream.h>  
# include <conio.h>  
# include <stdio.h>  
  
main( )  
{  
    clrscr( );  
    int qty=10;  
    int & qt=qty;  
  
    printf ("qty Location qt Location");  
    printf ("\n==== ===== == =====");  
    printf ("\n%d %u %d %u", qt,&qt,qty,&qty);  
    qt++;  
    printf ("\n%d %u %d %u", qt,&qt,qty,&qty);  
    qty--;  
    printf ("\n%d %u %d %u", qt,&qt,qty,&qty);  
    return 0;  
}
```

**OUTPUT:**

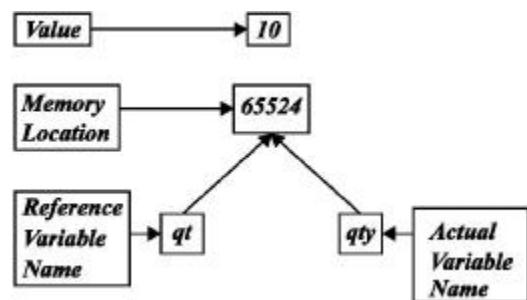
qty	Location	qt	Location
10	65524	10	65524
11	65524	11	65524
10	65524	10	65524

**Explanation:** In the above program, the variable `qty` is declared as integer variable and initialized with 10. The variable `qt` is declared as reference variable for variable `qty`. We can use variable `qt` to access the value of `qty`. Any change made in one of the variable changes the contents of both the variables. The contents of both variables and their address are always same. The variable `qt` and `qty` are modified using increment and decrement operator. But the contents and address printed on both variables are same. Figure 3.14 illustrates this.

### DEREFERENCING OPERATOR (\*)

The asterisk (\*) is a variable expression used to declare a pointer to a given type. In the example `int *x;` Where, `x` is a pointer of integer type, if the operand is a “pointer to function,” the result is a function designator. If the operand is a pointer to an object, the result is a lvalue, indicating that object. In the following conditions, the result of indirection is undefined.

- (1) The expression is a null pointer.
- (2) The expression is the address of an automatic variable and execution and it is out of scope.



**Fig. 3.14** Reference variable

### DIFFERENCE BETWEEN & AND \* OPERATOR

The & is a reference operator. It displays address of the variable in the RAM. To display the address of the variable it should be preceded by the variable name.

**Example:**

```
int b=10;
printf ("%u", &b);
```

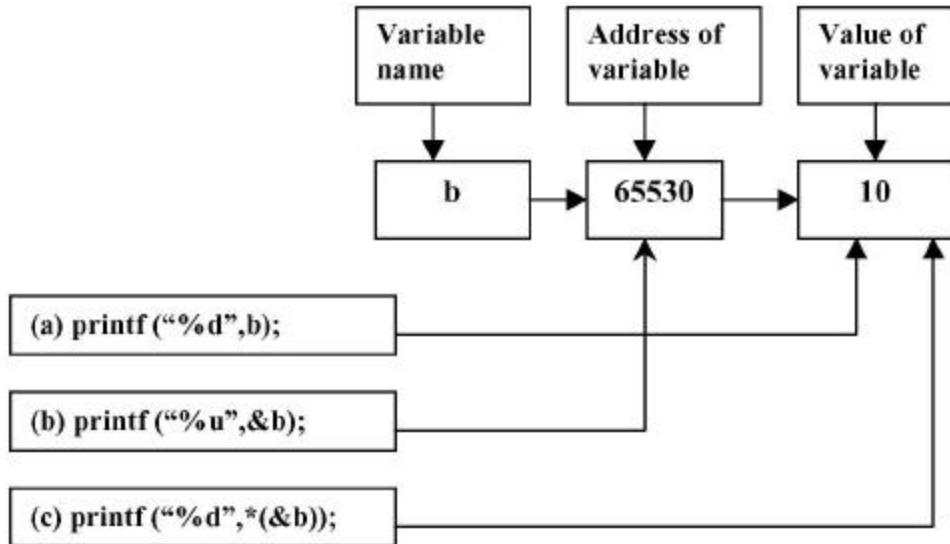
The above statement displays the address of the integer variable `b`.

The operator \* is used to display the value stored at the address of the variable.

**Example:**

```
int b=10;
printf (" %d", *(&b));
```

The above statement displays the value of value stored at address of `b` i.e., value of `b`.



**Fig.3.15** Difference between & and \* operator

As shown in [Figure 3.15](#), the statement (a) displays the contents of variable **b**. The statement **b** displays the address of the variable **b** and the statement (c) displays the value of variable **b**.

### 3.20 SCOPE ACCESS OPERATOR

Like C, the variables declared in C++ programs are totally different from other languages. We can use the same variable names in the C++ program in separate blocks. The declaration of the same variable refers to different memory locations. When we declare a variable it is available only to specific part or block of the program. The remaining block or other function cannot access the variable. The area or block of the C++ program from where the variable can be accessed is known as the scope of variables.

The scope access (or resolution) operator :: (two semicolons) allows a programmer to access a global (or file duration) name even if it is hidden by a local re-declaration of that name.

### 3.26 Write a program to use scope access operator. Display the various values of the same variable declared at different scope levels.

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

int a=10;

main( )
{
    clrscr( );
    int a=20;
    cout <<"::a=" <<>::a;
    cout <<" a=" <<a;
    return 0;
}
```

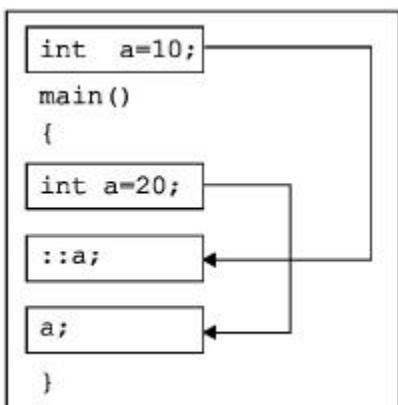
## OUTPUT:

::a=10 a=20

**Explanation:** In the above program, the integer variable a is declared before main( ) and initialized with 10. It is a global variable. In the function main( ) re-declaration of a is done and this time the variable a is initialized with 20. The first cout statement displays the global value of variable a i.e., 10. In this statement ::scope access operator is used to access the global value of the variable a. The second cout statement displays the local value of the variable a i.e., 20. [Figure 3.16](#) explains use of scope access operator.

## 3.21 MEMORY MANAGEMENT OPERATORS

In C language, we have studied the function malloc( ), calloc( ) and realloc( ) to allocate memory dynamically at run time in the program. The free( ) function is used to release the resources allocated by these functions. C++ allows us to use these functions. In addition, C++ provides operators which help us to allocate and release the memory in an easy way than these functions. These new operators are new and delete. The new operator creates an object and delete destroys the object. These operators are easy in writing as compared to malloc( ) and calloc( ). The syntax for new and delete are illustrated with suitable programs.



**Fig. 3.16** Scope access operator

Following are the advantages of new operator over the function malloc( ).

- (1) The new operator itself calculates the size of the object without the use of sizeof( ) operator.
- (2) It returns the pointer type. The programmers need not take care of its typecasting.
- (3) The new operator allocates memory and initializes the object at once.
- (4) The new and delete operators are very easy in syntax. They can be overloaded.

### The new operator

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

Here, pointer memory variable is a pointer to the data type. The new operator allocates memory of specified type and returns back the starting address to the pointer memory variable. Here, the element size is optional and used when we want to allocate memory

space for user defined data types such as arrays, classes and structures. If the new operator fails to allocate the memory it returns NULL, which can be used to detect failure or success of new operator.

- (a) `pv= new int;`
- (b) `int *pv=new int (50);`
- (c) `*p= new int [3]`

In example (a) `pv` is a pointer variable of integer type. After allocation `pv` contains the starting address. In example (b), 50 is assigned to pointer variable `pv`. In example (c) memory for 3 integers i.e., 6 bytes are assigned to pointer variable `p`. Examples of new operator with arrays are given below:

- (a) `pv= new int [5] [2]; // valid`
- (b) `pv= new int [8] [k] [2] // invalid`
- (c) `pv= new int [ ] [ 2 ] [ 2 ] // invalid`

### The delete operator

The delete operator frees the memory allocated by the new operator. This operator is used when the memory allocated is no longer usable in the program. Follow the following syntax to use the delete operator:

#### Syntax:

- (a) `delete <pointer memory variable>`
- (b) `delete [element size] <pointer memory variable>`

#### Example:

- (a) `delete p;`
- (b) `delete [5 ]p or delete [ ]p;`

In example (a) the delete operator releases the memory allocated to pointer `p`. The example (b) is advantageous when we want to free the dynamically allocated memory of array. The new C++ compilers do not require element size.

### 3.27 Write a program to allocate memory using new operator.

```
# include <iostream.h>
# include <conio.h>
void main( )
{
    clrscr( );
    int *p= new int[3],k;           // Memory allocation for 3 integers

for (k=0;k<3;k++)
{
    cout <<"\nEnter a Number : ";
    cin >>p;

    p++; // Pointing to next location
}

p-=3;           // Back to starting location

    cout <<"\n Entered numbers with their address are :\n";
    for (k=0;k<3;k++)
    {
        cout <<"\n\t" <<*p <<"\t" <<(unsigned)p; // type casting
        p++;
    }
}
```

```

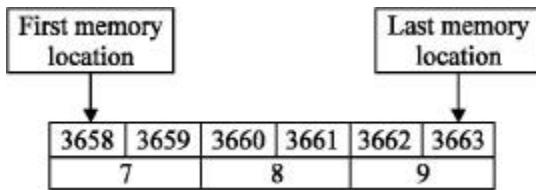
    p = 3;
    delete p;
}

OUTPUT
Enter a Number : 7
Enter a Number : 9
Enter a Number : 8

Entered numbers with their address are :
7 3658
9 3660
8 3662

```

**Explanation:** In the above program, `p` is an integer pointer variable. The `new` operator allocates memory required for three integers i.e., 6 bytes to pointer `p`. The first `for` loop reads integer through the keyboard and stores the number at memory location pointed by `p` as shown in [Figure 3.17](#). Each time pointer `p` is incremented it shows next location of its type. The second `for` loop displays the number by applying the same logic. Before that, 3 again sets the pointer to the starting location by decrementing. The `delete` operator releases the memory allocated by the `new` operator.



**Fig. 3.17** Memory map of integers

In the output of the program only starting memory location numbers are displayed. It is shown in [Fig. 3.17](#).

`sizeof( )`

The `sizeof( )` operator is used to return size occupied in bytes in memory by the variable. The `sizeof( )` operator in C++ displays different values as compared to C. Program 3.28 illustrates this:?

### 3.28 Write a program to display number of bytes occupied by `char` data type.

```

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

void main( )
{
    clrscr();
    cout << sizeof('a');
}

```

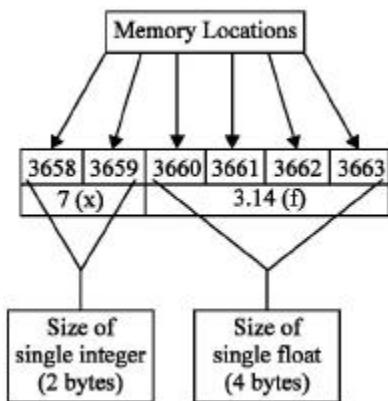
#### **OUTPUT**

**1**

**Explanation:** In the above program, character constant uses `sizeof( )` operator. The size determined is 1 byte. The same program C will display the size 2 bytes because C and

C++ reacts differently with data types. In C, a character is considered as integer. Hence size displayed is 2 whereas in C++ it is considered as a character.

The size is the space occupied in memory in bytes by the variable. It depends upon data type of variable. **Figure 3.18** describes the space occupied in the memory by the variable integer and float.



**Fig.3.18** Size of int and float data

**Example:**

```
int x=5;  
float f=3.14;
```

The integer variable occupies two bytes and float variable occupies four bytes in memory.

## COMMENTS

In C++, a symbol // (double slash) is used as a comment symbol and there is no termination symbol. Comment begins with // symbol. A comment can be inserted anywhere in the line and whatever follows till the end of line is ignored. The C comments symbol, /\* and \*/ are also valid in C++.

**Examples:**

```
// This is a C++ comment style.  
/* The C comment style is also valid in C++. */
```

## 3.22 COMMA OPERATOR

In C, every statement of the program is terminated by semi-colon (;). It is also applicable in C++ and in addition, C++ allows us to terminate a statement using comma operator after satisfying the following rules:

- (1) The variable declaration statements should be terminated by semi-colon.
- (2) The statements followed by declaration statements like clrscr( ), cin, cout can be terminated by comma operator.
- (3) C++ permits declaration of variables at any place in program but such declaration is not allowed in between the statements terminated by comma operator. The initialization of previously declared variables can be done.
- (4) The last statement of the program must be terminated by semi-colon.

Consider the following programs:

### 3.29 Write a program to use comma operator in place of semi-colon.

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

void main( )
{
    clrscr( ),
    cout<<"\nUse of comma operator",
    cout<<endl;
}
```

#### OUTPUT

##### Use of comma operator

**Explanation:** In function `main()`, comma operator terminates the first two statements and the last statement is terminated by semi-colon.

All the above points are noticed in the Turbo C++ compiler. The reader is advised to follow his/her own observations.

### 3.30 Write a program to declare an integer, initialize it and display it. Terminate the statements using comma operator.

```
# include <iostream.h>
# include <conio.h>
void main( )
{
    int x;
    clrscr( ),
    x=10,
    cout<<"\n x = "<<x,
    cout<<endl;
}
```

#### OUTPUT

**x = 10**

**Explanation:** In the above program, the first and last statements are terminated by semi-colon. Comma operator terminates the statements in between these two statements.

## 3.23 COMMA IN PLACE OF CURLY BRACES

The {} (curly braces) are used to define the body of a function and scope of the control statements. The opening curly brace {{}} indicates starting of the scope and {{}} closing curly brace indicates the end of the scope. It is also possible to use comma operator in condition and loop statement in place of {} to indicate the scope of the statement. The use of comma operator is not allowed in definition of function. The following declaration is invalid:

```
main( )
```

```
,
```

```
,
```

Following program explains the use of comma operator with conditional and loop statements.

### 3.31 Write a program to use comma operator in if..else structure as scope indicator.

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

```
# include <conio.h>

void main( )
{
    int x;
    clrscr( ),
    x=10;
if (x==10)      // if block
    cout <<x<<endl,
    cout<<x+1<<endl,
    cout<<x+2<<endl,
    cout <<"end of if block"; // end of if block terminated by semi-colon
else
    cout<<"False",           // else block
    cout<<"\nEnd";           // end of else block terminated by semi-colon
}
```

## **OUTPUT**

**10  
11  
12**

### **end of if block**

**Explanation:** In program 3.31, integer variable x is declared and initialized with 10. The if statement checks the value of x and executes respective blocks. The statements of if and else blocks are terminated by comma. The last statements of if block and else block are terminated by semi-colon to indicate the end of scopes. In this program the if block is executed.

## **3.32 Write a program to use comma operator in for loop to indicate the scope.**

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

void main( )
{
    int x;
    clrscr( ),
    x=5;

    for (;x>0;x--)
        cout<<"\nx = "<<x,           // {
        cout <<" In loop ",          //
        cout <<" In loop";         // }

    cout<<" Out of loop";       // executed after end of for loop
}
```

## **OUTPUT**

**x = 5 In loop In loop  
x = 4 In loop In loop  
x = 3 In loop In loop  
x = 2 In loop In loop  
x = 1 In loop In loop Out of loop**

**Explanation:** In the above program, the statements terminated by comma are included in the scope of for loop. The first two statements are terminated by comma and the last statement terminated by semicolon i.e., end of scope of for loop.

## SUMMARY

- (1) C++ programs consist of objects, classes, functions, variables, and other statements.
- (2) The C++ keywords are reserved words by the compiler. All C language keywords are valid in C++ and few additional keywords are added by ANSI committee.
- (3) Identifiers are names of variables, functions, and arrays. They are user-defined names, consisting of sequence of letters and digits, with a letter as a first character.
- (4) The constants in C++ are applicable to those values, which do not change during execution of a program. The two types of constants are literal and symbolic.
- (5) C++ supports all data types of C.
- (6) The keywords `signed`, `unsigned`, `short`, and `long` are type modifiers. A type modifier changes the meaning of the base data type to produce a new data type.
- (7) The `void` type is added in ANSI C. It is also known as Empty data type.
- (8) The `enum` is a keyword. It is used for declaring enumeration data types. C++ allows us to declare variable of `enum` data type and `enum` without tag name.
- (9) Variables are used to store constants i.e., information. A variable is a sequence of memory locations, which are used to store the assigned constant.
- (10) C++ permits declaration of variables anywhere in the program.
- (11) The initialization of variable at run-time is called as dynamic initialization.
- (12) C++ supports all the operators of C. The list of newly introduced operator is described in Table 3.4.
- (13) The referencing operator is used to define referencing variable. A reference variable prepares an alternative (alias) name for previously defined variable.
- (14) The asterisk (\*) in a variable expression is used to declare a pointer to a given type.
- (15) The scope access (or resolution) operator :: (two semicolons) allows a programmer to access a global (or file duration) name even if it is hidden by a local re-declaration of that name.
- (16) The `new` operator creates an object and `delete` destroys the object. These operators are easy in writing as compared to `malloc()` and `calloc()`.
- (17) C++ reference types, declared with & operator, are nearly identical but not exactly same to pointer types. They declare aliases for objects variables and allow the programmer to pass arguments by reference to functions.
- (18) In C++ a symbol // (double slash) is used as a comment symbol and there is no termination symbol.
- (19) C++ supports all the decision making and loop control statements of C.

## EXERCISES

### [A] Answer the following questions.

- (1) Describe different parts of C++ programs.
- (2) List the new keywords in C++ with their functions.
- (3) What are identifiers, variables, and constants?
- (4) Which are the two types of constants? Describe them with suitable examples.
- (5) Describe the statements for creating constants. Explain with examples.
- (6) What is the use of the keyword `void`? In how many ways can it be used with function?
- (7) What is the difference between variable declaration in C and C++?
- (8) What is dynamic initialization? Is it possible in C?
- (9) Describe the use of scope access operator (::) and reference operator (&).

- (10) What are the advantages of `new` operator over `malloc ( )` function?
- (11) What are type modifiers? Why are they essential in C++?
- (12) Describe types of derived data type.
- (13) Describe the following terms
  - (a) Precedence of operators in C++
  - (b) Type modifiers
  - (c) Constant pointers
- (14) Explain typecasting. What are explicit and implicit type conversions?
- (15) Explain wrapping around of value.
- (16) Explain use of comma operator.

**[B] Answer the following by selecting the appropriate option.**

(1) In C++, the symbol used for writing comments is

- (a) //
- (b) ///\* \*/
- (c) \*/\*/
- (d) none of the above

(2) The :: is known as

- (a) scope access operator
- (b) double colons
- (c) both (a) and (b)
- (d) none of the above

(3) The `delete` operator is used

- (a) to delete object
- (b) to delete file
- (c) both (a) and (b)
- (d) none of the above

(4) The `new` and `delete` are

- (a) operators
- (b) keywords
- (c) both (a) and (b)
- (d) none of the above

(5) The `new` operator

- (a) allocates memory
- (b) releases memory
- (c) both (a) and (b)
- (d) none of the above

(6) What will be the output of the following program?

```
# include <iostream.h>

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

- (a) 2
- (b) 1

(c) 4

(d) none of the above

(7) What will be the value of c after execution of the following program?

```
# include <iostream.h>
void main( )
{
    int *p, c=0;
    p=new int[4];
    for (int x=0;x<2;x++)
        c=c+sizeof((p+x));
}
```

(a) c=4

(b) c=8

(c) c=2

(d) c=0

(8) The union declared without tag name is called as

(a) anonymous union

(b) nameless union

(c) unknown union

(d) void union

(9) What will be the output of the following program?

```
# include <iostream.h>
void main( )
{
{
    for ( int i=0;i<5;i++)
    { cout <<i; }
    cout <<" i ="<<i;
}
```

(a) 01234 i=5

(b) undefined symbol i

(c) 012345

(d) none of the above

### [C] Attempt the following programs.

(1) Write a program to allocate memory using new operator for 10 integers. Read and display the integers.

(2) Write a program to evaluate following series:

(a)  $x=x^2+x^3+\dots+x^n$

(b)  $y=2^x+2*(x^3-10)$

(c)  $z=x-y$  (Use (a) and (b))

(d) Display square root of z.

(3) Write a program to display A to Z characters using while loop.

(4) Write a program to draw a square box. Use for loop.

(5) Write a program to declare and initialize a variable. Create a reference variable. Display the value of actual variable using reference variable.

(6) Given the Vander Wall's constants x and y for a gas. Calculate the critical temperature, pressure, and volume using the following formulas:

$$ct = 8x/27Rb$$

$$p=x/27b^2$$

v=3b

**R= 0.0821 dm<sup>3</sup>atm/mol/k**

Read values of x and y and calculate and print the values of ct,p, and v.

(7) The sum of the square of the first n natural numbers is calculated by the formula  $sum=n(n+1)*(2n+1)/5$ . Read value of n through the keyboard and calculate the sum of square of first n natural numbers.

(8) Write a program to calculate

(a) Area of circle ( $area=3.12*r^2$ )

(b) Circumference of the circle ( $c=2*3.12*r$ )

(c) Volume of the cylinder ( $v=3.12*r^2*h$ )

(d) Surface area of the closed cylinder ( $s=2*3.12*r*h+2*3.12*r^2$ )

(e) Volume of sphere ( $v=4/3*3.12*r^3$ )

(9) A company gives following rates of commission for the monthly sales of the product:

Below Rs. 15000/-	No commission
15001-20000/-	5%
20001-30000/-	10%
Above 30000/-	12%

Write a program to read the sales and display the commission.

(10) Write a program to display the sum of odd numbers between 1 to 150.

(11) A worker takes a job for 31 days. His pay for the first day is Rs. 20. His pay for the second day is Rs. 40. Each day's pay is twice what he gets in the previous day. What will be the total pay for 31 days?

(12) Write a program to find the range of the given numeric data.

(Range smallest number-largest number)

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

4

CHAPTER

# Control Structures

C  
H  
A  
P  
T  
E  
R  
  
O  
U  
T  
L  
I  
N  
E

- [4.1 Introduction](#)
- [4.2 Decision-Making Statements](#)
- [4.3 The if-else Statement](#)
- [4.4 The Nested if-else Statement](#)
- [4.5 The jump Statement](#)
- [4.6 The goto Statement](#)
- [4.7 The break Statement](#)
- [4.8 The continue Statement](#)
- [4.9 The switch case Statement](#)
- [4.10 The Nested switch\( \) case Statement](#)
- [4.11 Loops in C/C++](#)
- [4.12 The for Loop](#)
- [4.13 Nested for Loops](#)
- [4.14 The while Loop](#)
- [4.15 The do-while Loop](#)

## 4.1 INTRODUCTION

This chapter deals with the basics of control structures of C++ language. Those who are already familiar with C, may skip this chapter. Those who are new to C++, should read this chapter thoroughly. As discussed earlier, C++ is a superset of C. The control structures of C and C++ are same. In this chapter, the concepts of structures are illustrated in detail for the new users.

A Program is nothing but a set of statements written in sequential order, one after the other. These statements are executed one after the other. Sometimes it may happen that the programmer requires to alter the flow of execution, or to perform the same operation for fixed iterations or whenever the condition does not satisfy. In such a situation the programmer uses the control structure. There are various control structures supported by C++. Programs which use such (one or three) control structures are said to be structured programs.

The C++ control structure covers two sets of statements. The set that performs certain operations repetitively is called as loop statements and the set that makes decision is known as decision-making statements.

## 4.2 DECISION-MAKING STATEMENTS

Following are decision-making statements.

- (1) The `if` statement
- (2) The `switch ( ) case` statement.

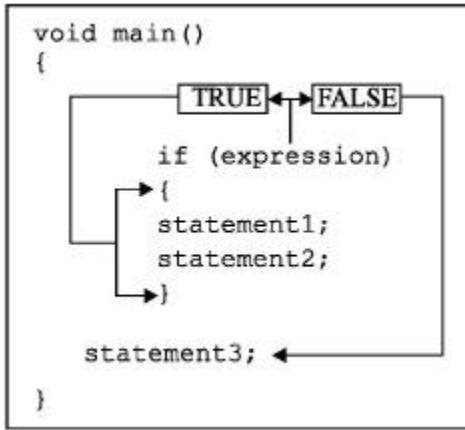
**Simple if statement** The syntax of simple `if` statement is described below

**Syntax** for the simplest `if` statement:

```
if (expression) /* no semi-colon */  
    Statement;
```

The `if` statement contains an expression. The expression is evaluated. If the expression is true it returns 1 otherwise 0. The value 1 or any non-zero value is considered as true and 0 as false. In C++, the values (1) true and (0) false are known as `bool` type data.

The `bool` type data occupies one byte in memory. If the given expression in the `if ( )` statement is true, the following statement or block of statements too are executed, otherwise the statement that appears immediately after `if` block (true block) is executed as given in [Figure 4.1](#).



**Fig. 4.1** The simple if statement

As shown in [Figure 4.1](#), the expression is always evaluated to true or false. When the expression is true, the statement in if block is executed. When the expression is false the if block is skipped and the statement (state-ment3) after if block is executed.

The expression given in the if statement may not be always true or false. For example, `if(1)` or `if(0)`. When such statement is encountered, the compiler will display a warning message "condition is always true" or "condition is always false". In place of expression we can also use function that returns 0 or 1 return values. The following programs illustrate the points discussed above.

#### 4.1 Write a program to enter age and display message whether the user is eligible for voting or not.

```

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

void main( )
{
    clrscr( );
    int age;
    cout<<"Enter Your Age :";
    cin>>age;
    if (age>=18)
    {
        cout <<" You are eligible for voting.";
    }
}

```

#### OUTPUT

Enter Your Age : 23

You are eligible for voting.

**Explanation:** In the above program, integer variable `age` is declared. The user enters his/her age. The value is tested with if statement. If the age is greater than or equal to 18 the statement followed by if statement is executed. The message displayed will be "You are eligible for voting." If the condition is false nothing is displayed.

#### 4.2 Write a program to use library function with if statement instead of expression.

```

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

void main( )
{
    static char nm[]="Hello";
    clrscr( );

    if (strlen(nm))
    { cout <<" The string is not empty."; }

}

```

## OUTPUT

**The string is not empty.**

**Explanation:** In the above program, the character array nm[ ] is initialized with the string "Hello". The strlen( ) function is used in the if statement. The strlen( ) function calculates the length of the string and returns it. If the string is empty it returns (0) false otherwise non-zero value (string length). The non-zero value is considered as true. The strlen( ) function returns non-zero value (true). The if statement displays the message "The string is not empty." Here, instead of expression, a library function is used.

## 4.3 THE IF-ELSE STATEMENT

The simple if statement executes statement only if the condition is true otherwise it follows the next statement. The else keyword is used when the expression is not true. The else keyword is optional. The syntax of if..else statement is given below and Figure 4.2 describes the working of if..else statement.

The format of if..else statement is as follows:

```

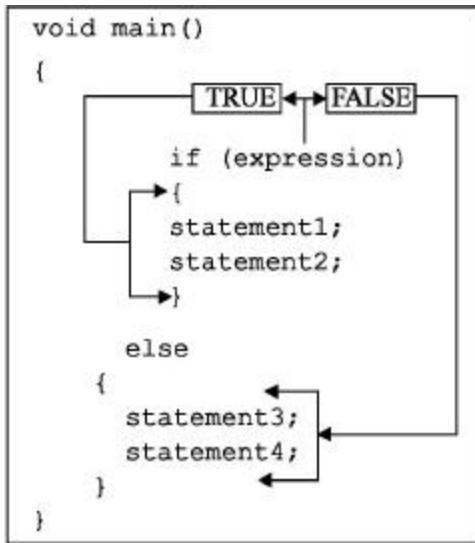
if(the condition is true)
execute the Statement1;
else
execute the Statement2;
OR
Syntax of if - else statement can be given as follows:
if ( expression is true)

{
statement 1;      →      if block
statement 2;
}

else

{
statement 3;      →      else block
statement 4;
}

```



**Fig.4.2** The if-else statement

As shown in [Figure 4.2](#), both if and else block contain statements. When the expression is true if block is executed otherwise else block is executed.

#### 4.3 Write a program to enter age and display message whether the user is eligible for voting or not. Use if-else statement.

```

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

void main( )
{
clrscr( );

int age;
cout<<"Enter Your Age : ";
cin>>age;

if (age>=18)
{
    cout <<" You are eligible for voting." ;
}
else
{
    cout <<" You are noneligible for voting"<<endl;
    cout<< " Wait for "<<18-age<<" year(s).";
}
}

```

#### OUTPUT:

Enter Your Age : 17  
 You are noneligible for voting  
 Wait for 1 year(s).

**Explanation:** In the above program, the user enters his/her age. The integer variable age is used to store the value. The if statement checks the value of age. If the age is greater than or equal to 18, the if block of statement is executed, otherwise the else block of statement is executed. Thus, the else statement extends if statement.

#### 4.4 Write a program with simple if statement. If entered score is less than 50 display one message, otherwise another message.

```
#include<iostream.h>
#include<conio.h>
int main( )
{
int v;
clrscr( );
cout<<"Enter a number : ";
cin>>v;
if(v>=50)
cout<<"\nCongrats !! You scored a half / more than half century ";
else
if(v<50)
cout<<"\nCome on !! You can score a half century ";
return 0;
}
```

#### OUTPUT:

```
Enter a number : 49
Come on !! You too can score a half century
```

```
Enter a number : 56
Congrats !! You scored a half / more than half century
```

**Explanation:** The program first prompts the user to enter a number 'v'. If it is greater than 50 or equal to 50, it prompts a particular message, otherwise for less than 50, other message is displayed. If found true, a particular message is prompted, otherwise another.

#### 4.4 THE NESTED IF-ELSE STATEMENT

In this kind of statements, number of logical conditions are checked for executing various statements. Here, if any logical condition is true, the compiler executes the block followed by if condition, otherwise it skips and executes else block.

In if..else statement, else block is executed by default after failure of if condition. In order to execute the else block depending upon certain conditions we can add repetitively if statements in else block. This kind of nesting will be unlimited.

**Syntax** of if-else...if statement can be given as follows:

```
if ( condition)
{
    statement 1;           -> if  block
    statement 2;
}

else    if (condition)
{
    statement 3;           -> else block
    statement4;
}
```

```

else
{
statement5;
statement6;
}

```

From the above block, following rules can be described for applying nested if..else..if statements:

1. Nested if..else can be chained with one another.
2. If the condition is false, control passes to else block where condition is again checked with the if statement. This process continues till there is no if statement in the last else block.
3. If one of the if statement satisfies the condition, other nested if..else will not be executed.

Following programs illustrates the working of nested if-else statements.

#### **4.5 Write a program to enter two characters and display the larger character. Use nested if-else statements.**

```

#include <iostream.h>
#include <constream.h>

void main( )
{
    char j,k;
    clrscr( );
    cout<<"\n Enter a character :";
    j=getche( );
    cout <<"\n Enter another character : ";
    k=getche( );
    cout<<endl;

    if (j>k)
    {
        cout <<j <<" is larger than "<<k;
    }
    else if(k>j)
    {
        cout <<k <<" is larger than "<<j;
    }
    else
    {
        cout <<"Both the characters are same";
    }
}

```

#### **OUTPUT**

```

Enter a character : S
Enter another character : A
S is larger than A

```

**Explanation:** In the above program, two characters are entered and stored in the character variable j and k. The first if statement checks whether j is greater than k. Here,

comparison is done considering ASCII values. If the condition is true, a message will be displayed and program terminates. In case the condition is false the `else` block is executed. In the `else` block another `if` statement is present. The `if` statement inside the `else` block checks whether `k` is greater than `j` or not. If the condition is true `if` block is executed, otherwise `else` block is executed.

#### 4.6 Write a program to explain the concept of nested `if-else` statements.

```
#include<conio.h>
#include<iostream.h>
int main( )
{
int score;
clrscr( );
cout<<"\nEnter Sachin's score :";
cin>>score;
if(score>=50)
{
    if(score>=100)
        cout<<"Sachin scored a century and more runs";
    else
    {
        cout<<"\nSachin scored more than half century ";
        cout<<"\nCross your fingers and pray he completes century ";
    }
}
else
{
    if(score==0)
        cout<<"Oh my God ";
    if(score>0)
        cout<<"Not in form today ";
}
return 0;
}
```

#### OUTPUT:

Enter Sachin's score : 0

Oh my God !

Enter Sachin's score : 45

Not in form today

Enter Sachin's score : 60

Sachin scored more than half century

Cross your fingers and pray he completes century

Enter Sachin's score : 116

Sachin scored a century and more runs

**Explanation:** From above it can be seen that if score was greater than 50 and greater than 100 then a particular message is prompted. If score was greater than 50 but less than 100, then another set of messages is prompted. If, however, the score was less than 50 and equal to 0 then a particular message is prompted. If score was less than 50 but not 0 then another message is displayed.

#### 4.7 Write a program to execute `if` statement without any expression.

```

# include <iostream.h>
# include <constream.h>

void main( )
{
    int j;
    clrscr( );
    cout <<"\n Enter a value : ";
    cin>>j;

    if(j)
        cout<<"Wel Come";
    else
        cout<<"Good Bye";
}

```

## OUTPUT

Enter a value : 0

Good Bye

**Explanation:** In the above program, an integer variable *j* is declared. The user is asked to enter an integer value. The entered value is stored in the variable *j*. The *if* statement checks the value of *j*. As explained at the beginning of this chapter, 0 is considered as false and any non-zero value is true. Thus, when user enters 0, the *else* block is executed and when a non-zero value is entered, *if* block is executed.

## THE IF-ELSE-IF LADDER STATEMENT

A common programming construct is the if-else-if ladder, sometimes called the *if-else-if* staircase because of it's appearance. It's general form is

```

if ( expression) statement block1;
    else statement block2;
        if(expression) statement block3;
            else statement block4;
                if(expression) statement block5;
                .
                .
                .
                .
                .
                else statement blockn;

```

The conditions are evaluated from the top. As soon as a true condition is met, the associated statement block gets executed and rest of the ladder is bypassed. If none of the conditions are met then the final *else* block gets executed .If this '*else*' is not present and none of the '*if*' evaluates to true then entire ladder is bypassed.

Although the indentation of the preceding '*if-else-if*' ladder is technically correct, it can lead to overly deep indentation. Imagine 256 (maximum allowed) such stairs and each indented: Enough to confuse!

This reason made it vital to use the form as shown below:

```

if ( expression)
    statement block1;      → if block

```

```

else    if (expression)
    statement block2;      → else block
else    if (expression)
    statement block3;      → else block
.
.
.
else
    statement block5;

```

The following program will clear your understanding.

#### **4.8 Write a program to simulate tariff charges for reaching different destinations by bus.**

```

#include<conio.h>
#include<iostream.h>
int main( )
{
int cost,ch;
clrscr( );
cout<<"\n.....NANDED BUS STATION.....\n";
cout<<".....Menu.....";
cout<<"\nBombay..1";
cout<<"\nNagpur..2";
cout<<"\nPune..3";
cout<<"\nAmravati..4";
cout<<"\nAurangabad..5";
cout<<"\n\nEnter your destination :";
cin>>ch;
if(ch==1)
    cost=100;
else if(ch==2)
    cost=70;
else if(ch==3)
    cost=50;
else if(ch==4)
    cost=60;
else if(ch==3)
    cost=40;
else
    cost=0;
if(cost!=0)
{
    cout<<"\n\nThe ticket cost is : Rs "<<cost;
    cout<<"\nPay the amount to get booked";
}
else
    cout<<"Sorry there's no bus to desired destination";
cin.get( );
return 0;
}

```

#### **OUTPUT:**

.....NANDED BUS STATION.....

.....Menu.....

Bombay..1

Nagpur..2

Pune..3

Amravati..4

**Aurangabad..5**

**Enter your destination : 4**

**The ticket cost is : Rs 60**

**Pay the amount to get booked**

**Explanation:** The program above simulates a bus station. We use an 'if-else-if' ladder to find out the cost of ticket to a particular station, if there was a bus to that station. There wasn't a bus to that station having cost=0.Finally if cost is non-zero it's printed. The user is prompted to enter the choice. Depending upon the choice, the fare of destination station is displayed. In the above example, choice 4 is given. The result displayed is

"The ticket cost is : Rs 60

Pay the amount to get booked".

## **4.5 THE JUMP STATEMENT**

C/C++ has four statements that perform an unconditional control transfer. These are `return( )`, `goto`, `break` and `continue`. Of these, `return( )` is used only in functions. The `goto` and `return( )` may be used anywhere in the program but `continue` and `break` statements may be used only in conjunction with a loop statement. In 'switch case' 'break' is used most frequently.

## **4.6 THE GOTO STATEMENT**

This statement does not require any condition. This statement passes control anywhere in the program without least care for any condition. The general format for this statement is shown below:

```
goto label;
```

```
—
```

```
—
```

```
label:
```

where, `label` is any valid `label` either before or after `goto`.The label must start with any character and can be constructed with rules used for forming identifiers. Avoid using `goto` statement.

### **4.9 Write a program to demonstrate the use of goto statement.**

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

void main( )
{
    int x;
    clrscr( );
    cout <<"Enter a Number :";
    cin>>x;
    if (x%2==0)
        goto even;
    else
        goto odd;
```

```
even :cout<<x<<" is Even Number.";
return;
odd: cout<<x<<" is Odd Number.";
}
```

## OUTPUT

Enter a Number: 5

5 is Odd Number.

**Explanation:** In the above program, a number is entered. The number is checked for even or odd with modulus division operator. When the number is even, the `goto` statement transfers the control to the label `even`. Similarly, when the number is odd, the `goto` statement transfers the control to the label `odd` and respective messages will be displayed.

## 4.7 THE BREAK STATEMENT

The `break` statement allows the programmer to terminate the loop. The `break` skips from the loop or the block in which it is defined. The control then automatically passes on to the first statement after the loop or the block. The `break` statement can be associated with all the conditional statements (especially `switch( )` case). We can also use `break` statements in the nested loops. If we use `break` statement in the innermost loop, then the control of program is terminated from that loop only and resumes at the next statement following that loop. The widest use of this statement is in `switch` case where it is used to avoid flow of control from one case to other.

## 4.8 THE CONTINUE STATEMENT

The `continue` statement works quite similar to the `break` statement. Instead of forcing the control to end of loop (as it is in case of `break`), `continue` causes the control to pass on to the beginning of the block/loop. In case of `for` loop, the `continue` case causes the condition testing and incrementation steps to be executed (while rest of the statements following `continue` are neglected). For `while` and `do-while`, `continue` causes control to pass on to conditional tests. It is useful in programming situation when you want particular iterations to occur only up to some extent or you want to neglect some part of your code. The programs on `break` and `continue` can be performed by the programmer.

## 4.9 THE SWITCH CASE STATEMENT

The `switch` statement is a multi-way branch statement and an alternative to `if-else-if` ladder in many situations. This statement requires only one argument, which is then checked with number of case options. The `switch` statement evaluates the expression and then looks for its value among the case constants. If the value is matched with a case constant then that case constant is executed until a `break` statement is found or end of `switch` block is reached. If not then simply `default` (if present) is executed (if `default` isn't present then simply control flows out of the `switch` block).

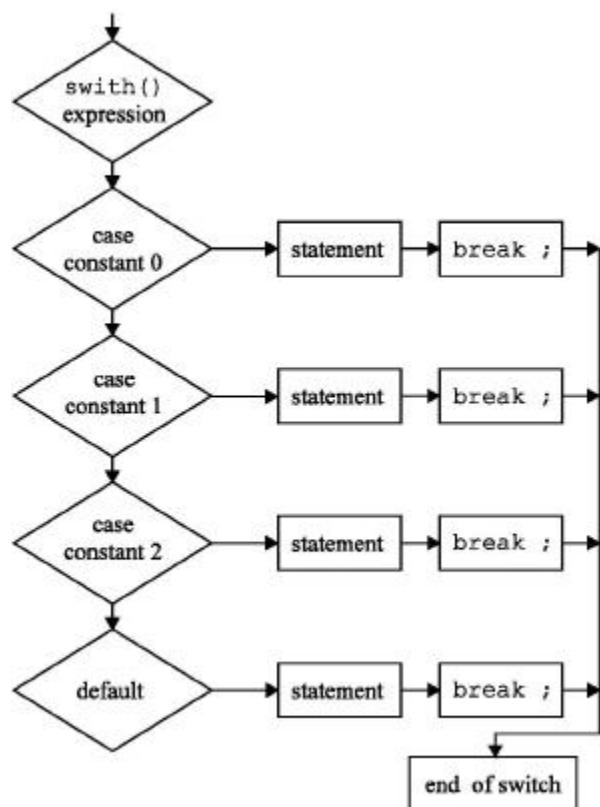
The `default` is normally present at the bottom of `switch` case structure. But we can also define `default` statement anywhere in the `switch` structure. The `default` block must not be empty. Every case statement terminates with : (colon). The `break` statement is used to

stop the execution of succeeding cases and pass the control to the switch end of block. The syntax of the `switch( )` case statement is shown below. [Figure 4.3](#) simulates the working of `switch( ) case statement`.

```
switch(variable or expression)
{
    case constant A :
        statement;
        break;

    case constant B :
        statement;
        break;

    default :
        statement ;
}
```



**Fig. 4.3** The switch case statement

**4.10 Write a program to display different lines according to users choice.**  
**Use `switch( ) case statement`.**

```
#include<stdlib.h>
#include<constream.h>
#include<iostream.h>
void main( )
{
int c;
```

```

clrscr( );

cout<<"\n LINE FORMAT MENU ";
cout<<"\n1] *****";
cout<<"\n2] ======";
cout<<"\n3] ~~~~~~~~";
cout<<"\n4] _____";
cout<<"\nEnter your choice :";

cin>>c;

switch(c)
{
    case 1:
        cout <<"\n*****";
        break;

    case 2:
        cout <<"\n=====";
        break;

    case 3:
        cout <<"\n~~~~~~~";
        break;

    case 4:
        cout <<"\n_____";
        break;

    default:
        cout <<"\n.....";
    }
}

```

## **OUTPUT**

### **LINE FORMAT MENU**

```

1] *****
2] =====
3] ~~~~~~~~
4] __

```

**Enter your choice :2**

```

=====

```

**Explanation:** In the above program, a menu is displayed on the screen with different type of lines. The user enters a number as given in the menu. The `switch( )` statement checks the value of variable `c`. All case statements are tested and one that satisfies, is executed. If the user enters a value other than that listed in the menu, `default` statement is executed.

**4.11 Write a program to enter a month number of year 2002 and display the number of days present in that month.**

```

#include <iostream.h>
#include<constream.h>
void main( )
{
clrscr( );

int month,days;
cout<<"Enter a month of year 2002 :";
cin>>month;

switch(month)
{
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        days=31;
        break;
    case 2:
        days=28;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        days=30;
        break;
}
cout<<"\nNumber of days in this month are "<<days;
}

```

## **OUTPUT**

**Enter a month of year 2002 :3**

**Number of days in this month are 31**

**Explanation:** The above program is meant to calculate number of days in a month (given by user) of year 2002. We know that month numbers 1, 3, 5, 7, 8, 10 and 12 have 31 days each. So they can be put together in switch( ) case. But since there isn't such an option we have used the technique shown above. Similar is the case with month numbers 4, 6, 9 and 11 having 30 days each.

## **4.10 THE NESTED SWITCH( ) CASE STATEMENT**

C/C++ supports the nesting of switch( ) case. The inner switch can be part of an outer switch. The inner and the outer switch case constants may be the same. No conflict arises even if they are same. The example below demonstrates this concept.

## **4.12 Write a program to demonstrate nested switch( ) case statement.**

```

#include<iostream.h>
#include<constream.h>
void main( )
{

```

```

int x;
clrscr( );

cout<<"\nEnter a number :";
cin>>x;
switch(x)
{
    case 0:
        cout<<"\nThe number is 0 ";
        break;

    default:
        int y;
        y=x%2;
        switch(y)
        {
            case 0:
                cout<<"\nThe number is even ";
                break;
            case 1:
                cout<<"\nThe number is odd ";
                break;
        }
}
}

```

## **OUTPUT**

**Enter a number :5**

**The number is odd**

**Explanation:** The above program identifies whether input number was zero, even or odd. The first `switch( )` case finds out whether the number is zero or non-zero. If a non-zero value is entered, a default statement of first `switch` case statement is executed which executes another nested `switch( )` case. The nested `switch( )` case statement determines whether the number is even or odd and displays respective messages.

## **4.11 LOOPS IN C/C++**

C/C++ provides loop structures for performing some tasks which are repetitive in nature. The C/C++ language supports three types of loop control structures. Their syntax is described in Table 4.1.

The `for` loop comprises of three actions. The three actions are placed in the `for` statement itself. The three actions initialize counter, test condition, and re-evaluation parameters are included in one statement. The expressions are separated by semi-colons (`;`). This helps the programmer to visualize the parameters easily. The `for` statement is equivalent to the `while` and `do-while` statements. The only difference between `for` and `while` is that the latter checks the logical condition and then executes the body of the loop, whereas the `for` statement test is always performed at the beginning of the loop. The body of the loop may not be executed at all times if the condition fails at the beginning. The `do-while` loop executes the body of the loop at least once regardless of the logical condition.

**Table 4.1 Loops in C++**

<b>for</b>	<b>while</b>	<b>do-while</b>
<pre>for (expression -1;       expression-2;       expression-3) statement ;</pre>	<pre>expression -1; while (expression -2) { statement; expression -3; }</pre>	<pre>expression -1; do { statement; expression-3; } while (expression-2);</pre>

## 4.12 THE FOR LOOP

The **for** loop allows execution of a set of instructions until a condition is met. Condition may be predefined or open-ended. Although all programming languages provide **for** loops, still the power and flexibility provided by C/C++ is worth mentioning. The general syntax for the **for** loop is given below:

### Syntax of for loop

```
for (initialization; condition; increment/decrement)
Statement block;
```

Though many variations of **for** loop are allowed, the simplest form is shown above.

The *initialization* is an assignment statement that is used to set the loop control variable(s). The condition is a relational expression that determines the number of the iterations desired or the condition to exit the loop. The *increment* or the re-evaluation parameter decides how to change the state of the variable(s) (quite often increase or decrease so as to approach the limit). These three sections must be separated by semi-colons. The body of the loop may consist of a block of statements (which have to be enclosed in braces) or a single statement (enclosure within braces is not compulsory but advised).

Following programs illustrate **for** loop:

4.13	4.14	4.15
<pre># include &lt;iostream.h&gt; # include &lt;constream.h&gt;  void main( ) { clrscr( ); int j;  for (j=1;j&lt;11;j++) cout&lt;&lt;" "&lt;&lt;j; }</pre>	<pre># include &lt;iostream.h&gt; # include &lt;constream.h&gt;  void main( ) { clrscr( ); int j=0;  for (;j&lt;11;j++) cout&lt;&lt;" "&lt;&lt;j; }</pre>	<pre># include &lt;iostream.h&gt; # include &lt;constream.h&gt;  void main( ) { clrscr( ); int j=0;  for (;j&lt;11;) cout&lt;&lt;" "&lt;&lt;j; ++j; }</pre>
<b>OUTPUT</b> <b>1 2 3 4 5 6 7 8 9 10</b>	<b>OUTPUT</b> <b>1 2 3 4 5 6 7 8 9 10</b>	<b>OUTPUT</b> <b>1 2 3 4 5 6 7 8 9 10</b>
Explanation: In this program initialization,	Explanation: In this program initialization is done before <b>for</b> statement.	Explanation: In this program initialization is done before <b>for</b> statement. Increment is

condition, and increment is done in single parenthesis.	In <code>for</code> statement only condition and increment is done.	done inside the loop. The <code>for</code> loop contains only condition.
---	---	--

#### 4.16 Write a program to display all leap years from 1900 to 2002.

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

void main( )
{
    int i=1900;
    clrscr( );
    cout<<"\nProgram to print all the leap years from 1900 to 2002
\n\n";

for(;i++<=2002;)
{
    if(i%4==0&&i%100!=0)
        cout<<i<<" ";
    else if(i%100==0&&i%400==0)
        cout<<i<<" ";
}
}
```

**Explanation:** One must be aware that only those years that are

- (a) divisible by 4 and not 100,
- (b) divisible by 100 and also 400,

alone are qualified to be called as leap years. The above program checks for these conditions for all years from 1900 to 2002 and then prints the leap years. Stress is on `for` loop construct and use of `if` and `else-if` in the body.

#### 4.13 NESTED FOR LOOPS

We can also nest `for` loops to gain more advantage in some situations. The nesting level is not restricted at all. In the body of a `for` loop, any number of sub `for` loop(s) may exist.

#### 4.17 Write a program to demonstrate nested `for` loops.

```
# include <iostream.h>
# include <constream.h>>
void main( )
{
int a,b,c;
clrscr( );
for (a=1;a<=2;a++) /* outer loop */
{
for (b=1;b<=2;b++) /* middle loop */
{
    for (c=1;c<=2;c++) /* inner loop */
        cout <<"\n a=<<a <<" + b="<<b <<" + c="<<c <<" :"<<a+b+c;
        cout <<"\n Inner Loop Over.";
}

cout <<"\n Middle Loop Over.";
}
```

```

cout<<"\n Outer Loop Over.";
}

```

**Explanation:** The above program is executed in the sequence shown below. The total number of iterations are equal to  $2*2*2=8$ . The final output provides 8 results. Table 4.2 shows the working of this program.

#### 4.14 THE WHILE LOOP

Another kind of loop structure in C/C++ is the while loop. It's format is given below.

**Syntax:**

```

while (test condition)
{
    body of the loop
}

```

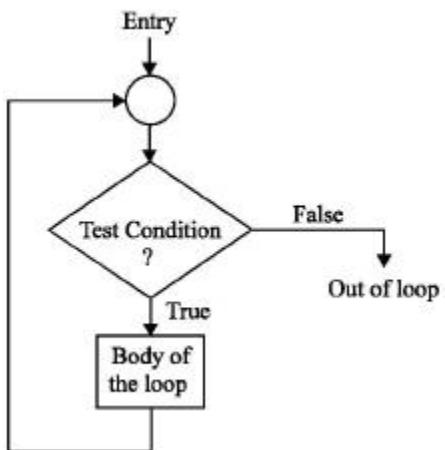
The test condition may be any expression. The loop statements will be executed till the condition is true i.e., the test condition is evaluated and if the condition is true, then the body of the loop is executed. When the condition becomes false the execution will be out of the loop.

The execution of the loop is shown in Figure 4.4.

The block of the loop may contain single statement or a number of statements. The same block can be repeated. The braces are needed only if the body of the loop contains more than one statement. However, it is a good practice to use braces even if the body of the loop contains only one statement.

**Table 4.2** Working of program 4.17

Values of loop variables			Output
Outer	Middle	Inner	
(1) a=1	b=1	c=1	a+b+c=3
2) a=1	b=1	c=2	a+b+c=4
<b>Inner Loop Over</b>			
(3) a=1	b=2	c=1	a+b+c=4
(4) a=1	b=2	c=2	a+b+c=5
<b>Inner Loop Over</b>			
<b>Middle Loop Over</b>			
(5) a=2	b=1	c=1	a+b+c=4
(6) a=2	b=1	c=2	a+b+c=5
<b>Inner Loop Over</b>			
(7) a=2	b=2	c=1	a+b+c=5
(8) a=2	b=2	c=2	a+b+c=6
<b>Inner Loop Over</b>			
<b>Middle Loop Over</b>			
<b>Outer Loop Over.</b>			



**Steps of while loops are as follows:**

- (1) The test condition is evaluated and if it is true, the body of the loop is executed.
- 2) On execution of the body, test condition is repetitively checked and if it is true the body is executed.
- 3) The process of execution of the body will be continued till the test condition becomes false.
- 4) The control is transferred out of the loop.

**Fig. 4.4** The while loop

**4.18 Write a program to add 10 consecutive numbers starting from 1. Use the while loop.**

```

#include <iostream.h>
#include <constream.h>

void main( )
{
    int a=1,sum=0;
    clrscr( );

while(a<=10)
{
    cout <<" "<<a;
    sum=sum+a;
    a++;
}
cout <<"\n Sum of 10 numbers : "<<sum;
}

```

**OUTPUT :**

**1 2 3 4 5 6 7 8 9 10**

**Sum of 10 numbers : 55**

**Explanation:** In the above program, integer variable a is initialized to 1 and variable sum to 0. The while loop checks the condition for  $a \leq 10$ . The variable a is added to variable sum and each time a is incremented by 1. In each while loop a is incremented and added to sum. When the value of a reaches to 10, the condition given in while loop becomes false. At last the loop is terminated. The sum of the number is displayed.

**4.19 Write a program to calculate the sum of individual digits of an entered number.**

```

#include <iostream.h>
#include <constream.h>
void main( )
{

```

```

int num,t;
clrscr( );
cout<<"\n enter a number : ";
cin>>num;
t=num;
int sum=0;
while(num)           //condition is true until num!=0
{
sum=sum+num%10;
num=num/10;
}
cout<<"\n Sum of the individual digits of the number "<<t<<" is =
"<<sum;
}

OUTPUT

```

**Enter a number : 234**

**Sum of the individual digits of the number 234 is = 9**

**Explanation:** The logic behind the program is to extract each time the LSD(lower significant digit) and divide the number by 10 so as to shift it by one place. To extract the LSD we use the fact that  $\text{num} \% 10$  = the remainder on dividing num by 10.

#### **4.20 Write a program to check whether the entered number is palindrome or not.**

```

# include <iostream.h>
# include <constream.h>
void main( )
{
int num;
clrscr( );
cout<<"\n Enter the number : ";
cin>>num;
int b=0,a=num;
while(a)
{
b=b*10+a%10;
a=a/10;
}
if(num==b)
cout<<"\n The given number is palindrome ";
else
cout<<"\n The given number is not palindrome ";
}

OUTPUT

```

**Enter the number: 121**

**The given number is palindrome**

**Explanation:** Palindrome is a number that is equal to it's reverse, for example, 51715. The program above first simply reverses the number and stores it in b. Then it compares b with original number to tell you whether or not the number is a palindrome. See that in each interaction the LSD of a is being made MSD of b. a is being shifted left and b is shifted right by one digit each in every iteration. Iterations are done until a exhausts or becomes equal to 0.

#### **4.15 THE DO-WHILE LOOP**

The format of do-while loop in C/C++ is given below.

```
do
{
statement/s;
}
while (condition);
```

while (condition); The difference between the while and do-while loop is the place where the condition is to be tested. In the while loop, the condition is tested following the while statement and then the body gets executed. Where as in do-while, the condition is checked at the end of the loop. The do-while loop will execute at least once even if the condition is false initially. The do-while loop executes until the condition becomes false.

#### **4.21 Write a program using do-while loop to print numbers and their cubes up to 10.**

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

void main( )
{
int y,x=1;
clrscr( );

cout <<"\n\tNumbers and their cubes \n";

do
{   y=pow(x,3);
    cout <<"\t" <<x <<"\t\t" <<y <<"\n";
    x++;
}   while (x<=10);
}
```

#### **OUTPUT**

#### **Numbers and their cubes**

**1**

**1**

**2**

**8**

**3**

**27**

**4**

**64**

**5**

**125**

**6**

**216**

**7**

**343**

**8**

**512**

**9**

**729**

**10**

**1000**

**Explanation:** Here, the mathematical function `pow (x, 3)` is used. Its meaning is to calculate the third power of x. With this function we get the value of  $y=x^3$ . For use of the `pow ( )` function we have to include `math.h` header file.

### SUMMARY

- (1) This chapter teaches the basics of control structures of C++ language.
- (2) The C++ control structure covers two sets of statements. The set that performs certain operations repetitively is called as loop statements and the set that makes decision is known as decision-making statements.
- (3) The simple `if` statement executes statement only if the condition is true, otherwise, it follows the next statement. The `else` keyword is used when the expression is not true. The `else` keyword is optional.
- (4) C/C++ has four statements that perform an unconditional control transfer. These are `return ( )`, `goto`, `break`, and `continue`. Of these, `return ( )` is used only in functions. `goto` and `return ( )` may be used anywhere in the program but `continue` and `break` statements may be used only in conjunction with a loop statement. `break` is used most frequently in `switch case`.
- (5) The `for` loop comprises of three actions. The three actions are placed in the `for` statement itself. The three actions ***initialize counter, test condition, and re-evaluation parameters*** are included in one statement. The `for` statement is equivalent to the `while` and `do-while` statement. The only difference between `for` and `while` is that the latter checks the logical condition and then executes the body of the loop, whereas the `for` statement test is always performed at the beginning of the loop. The body of the loop may not be executed at all times if the condition.

### EXERCISES

#### [A] Answer the following questions.

- (1) Explain the need of control structures in C++.
- (2) What are the differences between `break` and `continue` statements?
- (3) Why `goto` statement is not commonly used?
- (4) Explain the working of `if-else` statement.
- (5) Explain the working of `switch ( ) case` statement.

- (6) Explain the role of `break` statement in `switch( ) case`.
- (7) What are the differences between `while` and `do-while` loop statements?
- (8) What is an infinite loop?
- (9) Explain nested if's.
- (10) Explain nested `switch( ) case` statement.
- (11) Explain the use of the keyword `default`.

**[B] Answer the following by selecting the appropriate option.**

- (1) Which of the following loop statement uses two keywords?
  - (a) `do-while` loop
  - (b) `for` loop
  - (c) `while` loop
  - (d) none of the above
- (2) The statement which requires at least one statement followed by it is
  - (a) `default`
  - (b) `continue`
  - (c) `break`
  - (d) `else`
- (3) The loop statement terminated by a semi-colon is
  - (a) `do-while` loop
  - (b) `for` loop
  - (c) `while` loop
  - (d) none of the above
- (4) Every expression always returns
  - (a) 0 or 1
  - (b) 1 or 2
  - (c) -1 or 0
  - (d) none of the above
- (5) The meaning of `if(1)` is
  - (a) always true
  - (b) always false
  - (c) both (a) and (b)
  - (d) none of the above
- (6) The curly braces are not present; the scope of loop statement is
  - (a) one statement
  - (b) two statements
  - (c) four statements
  - (d) none of the above
- (7) In `nested` loop
  - (a) the inner most loop is completed first
  - (b) the outer most loop is completed first
  - (c) both (a) and (b)
  - (d) none of the above

**[C] Attempt the following programs.**

- (1) Write a program to display numbers from 10 to 1 using `for` loop.
- (2) Write a program to calculate the factorial of a given number.
- (3) Write a program to display only even numbers in between 1 to 150.
- (4) Write a program to solve the series  $x=1/2!+1/4!+1/n!$ .
- (5) Write a program to calculate the sum of numbers between 1 to N numbers. The user enters the value of N.
- (6) Write a program to use `break` and `continue` statements.
- (7) Write a program to display alphabets A to Z using `while` loop.
- (8) Write a program to use `break` statement and terminate the loop.
- (9) Write a program to demonstrate the use of `continue` statement.

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

[Skip to content](#)



- 
- 
- 
- 
- 

## Table of Contents for Object-Oriented Programming with ANSI and Turbo C++

- Search in book...
- Toggle Font Controls

- 
- 
- 
- 

PREV [Previous Chapter](#)

[3. C++ Declarations](#)

NEXT [Next Chapter](#)

[5. Functions in C++](#)

4

CHAPTER

# Control Structures

- [4.1 Introduction](#)
- [4.2 Decision-Making Statements](#)
- [4.3 The if-else Statement](#)
- [4.4 The Nested if-else Statement](#)
- [4.5 The jump Statement](#)
- [4.6 The goto Statement](#)
- [4.7 The break Statement](#)
- [4.8 The continue Statement](#)
- [4.9 The switch case Statement](#)
- [4.10 The Nested switch\( \) case Statement](#)
- [4.11 Loops in C/C++](#)
- [4.12 The for Loop](#)
- [4.13 Nested for Loops](#)
- [4.14 The while Loop](#)
- [4.15 The do-while Loop](#)

## 4.1 INTRODUCTION

This chapter deals with the basics of control structures of C++ language. Those who are already familiar with C, may skip this chapter. Those who are new to C++, should read this chapter thoroughly. As discussed earlier, C++ is a superset of C. The control structures of C

and C++ are same. In this chapter, the concepts of structures are illustrated in detail for the new users.

A Program is nothing but a set of statements written in sequential order, one after the other. These statements are executed one after the other. Sometimes it may happen that the programmer requires to alter the flow of execution, or to perform the same operation for fixed iterations or whenever the condition does not satisfy. In such a situation the programmer uses the control structure. There are various control structures supported by C++. Programs which use such (one or three) control structures are said to be structured programs.

The C++ control structure covers two sets of statements. The set that performs certain operations repetitively is called as loop statements and the set that makes decision is known as decision-making statements.

## 4.2 DECISION-MAKING STATEMENTS

Following are decision-making statements.

- (1) The if statement
- (2) The switch ( ) case statement.

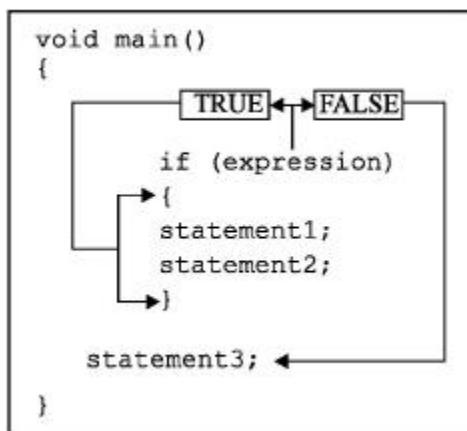
**Simple if statement** The syntax of simple if statement is described below

**Syntax** for the simplest if statement:

```
if (expression) /* no semi-colon */  
    Statement;
```

The if statement contains an expression. The expression is evaluated. If the expression is true it returns 1 otherwise 0. The value 1 or any non-zero value is considered as true and 0 as false. In C++, the values (1) true and (0) false are known as bool type data.

The bool type data occupies one byte in memory. If the given expression in the if ( ) statement is true, the following statement or block of statements too are executed, otherwise the statement that appears immediately after if block (true block) is executed as given in [Figure 4.1](#).



**Fig. 4.1** The simple if statement

As shown in [Figure 4.1](#), the expression is always evaluated to true or false. When the expression is true, the statement in `if` block is executed. When the expression is false the `if` block is skipped and the statement (statement3) after `if` block is executed. The expression given in the if statement may not be always true or false. For example, `if(1)` or `if(0)`. When such statement is encountered, the compiler will display a warning message “condition is always true” or “condition is always false”. In place of expression we can also use function that returns 0 or 1 return values. The following programs illustrate the points discussed above.

#### **4.1 Write a program to enter age and display message whether the user is eligible for voting or not.**

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

void main( )
{
    clrscr( );
    int age;
    cout<<"Enter Your Age :";
    cin>>age;
    if (age>=18)
    {
        cout <<" You are eligible for voting.";
    }
}
```

#### **OUTPUT**

**Enter Your Age : 23**

**You are eligible for voting.**

**Explanation:** In the above program, integer variable `age` is declared. The user enters his/her age. The value is tested with `if` statement. If the age is greater than or equal to 18 the statement followed by `if` statement is executed. The message displayed will be “You are eligible for voting.” If the condition is false nothing is displayed.

#### **4.2 Write a program to use library function with `if` statement instead of expression.**

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

void main( )
{
    static char nm[]="Hello";
    clrscr( );

    if (strlen(nm))
    { cout <<" The string is not empty.";  }

}
```

#### **OUTPUT**

**The string is not empty.**

**Explanation:** In the above program, the character array nm[ ] is initialized with the string "Hello". The `strlen()` function is used in the `if` statement. The `strlen()` function calculates the length of the string and returns it. If the string is empty it returns (0) false otherwise non-zero value (string length). The non-zero value is considered as true. The `strlen()` function returns non-zero value (true). The `if` statement displays the message "The string is not empty." Here, instead of expression, a library function is used.

### 4.3 THE IF-ELSE STATEMENT

The simple `if` statement executes statement only if the condition is true otherwise it follows the next statement. The `else` keyword is used when the expression is not true. The `else` keyword is optional. The syntax of `if..else` statement is given below and Figure 4.2 describes the working of `if..else` statement.

The format of `if..else` statement is as follows:

```
if(the condition is true)
execute the Statement1;
else
execute the Statement2;
```

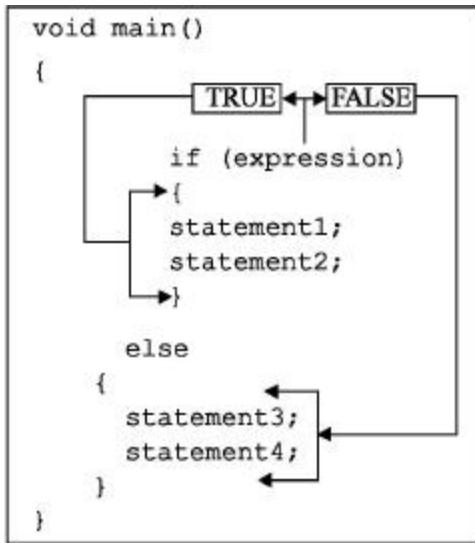
**OR**

**Syntax** of `if - else` statement can be given as follows:  
`if ( expression is true)`

```
{  
statement 1;      →      if block  
statement 2;  
}
```

`else`

```
{  
statement 3;      →      else block  
statement 4;  
}
```



**Fig.4.2** The if-else statement

As shown in [Figure 4.2](#), both `if` and `else` block contain statements. When the expression is true `if` block is executed otherwise `else` block is executed.

#### 4.3 Write a program to enter age and display message whether the user is eligible for voting or not. Use if-else statement.

```

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

void main( )
{
clrscr( );

int age;
cout<<"Enter Your Age : ";
cin>>age;

if (age>=18)
{
    cout <<" You are eligible for voting." ;
}
else
{
    cout <<" You are noneligible for voting" <<endl;
    cout << " Wait for "<<18-age<<" year(s).";
}
  
```

#### OUTPUT:

Enter Your Age : 17  
 You are noneligible for voting  
 Wait for 1 year(s).

**Explanation:** In the above program, the user enters his/her age. The integer variable age is used to store the value. The if statement checks the value of age. If the age is greater than or equal to 18, the if block of statement is executed, otherwise the else block of statement is executed. Thus, the else statement extends if statement.

#### 4.4 Write a program with simple if statement. If entered score is less than 50 display one message, otherwise another message.

```
#include<iostream.h>
#include<conio.h>
int main( )
{
int v;
clrscr( );
cout<<"Enter a number : ";
cin>>v;
if(v>=50)
cout<<"\nCongrats !! You scored a half / more than half century ";
else
if(v<50)
cout<<"\nCome on !! You can score a half century ";
return 0;
}
```

#### OUTPUT:

```
Enter a number : 49
Come on !! You too can score a half century
```

```
Enter a number : 56
Congrats !! You scored a half / more than half century
```

**Explanation:** The program first prompts the user to enter a number 'v'. If it is greater than 50 or equal to 50, it prompts a particular message, otherwise for less than 50, other message is displayed. If found true, a particular message is prompted, otherwise another.

#### 4.4 THE NESTED IF-ELSE STATEMENT

In this kind of statements, number of logical conditions are checked for executing various statements. Here, if any logical condition is true, the compiler executes the block followed by if condition, otherwise it skips and executes else block.

In if..else statement, else block is executed by default after failure of if condition. In order to execute the else block depending upon certain conditions we can add repetitively if statements in else block. This kind of nesting will be unlimited.

**Syntax** of if-else...if statement can be given as follows:

```
if ( condition)
{
    statement 1;           -> if  block
    statement 2;
}

else    if (condition)
{
    statement 3;           -> else block
    statement4;
}
```

```

else
{
statement5;
statement6;
}

```

From the above block, following rules can be described for applying nested if..else..if statements:

1. Nested if..else can be chained with one another.
2. If the condition is false, control passes to else block where condition is again checked with the if statement. This process continues till there is no if statement in the last else block.
3. If one of the if statement satisfies the condition, other nested if..else will not be executed.

Following programs illustrates the working of nested if-else statements.

#### **4.5 Write a program to enter two characters and display the larger character. Use nested if-else statements.**

```

#include <iostream.h>
#include <constream.h>

void main( )
{
    char j,k;
    clrscr( );
    cout<<"\n Enter a character :";
    j=getche( );
    cout <<"\n Enter another character : ";
    k=getche( );
    cout<<endl;

    if (j>k)
    {
        cout <<j <<" is larger than "<<k;
    }
    else if(k>j)
    {
        cout <<k <<" is larger than "<<j;
    }
    else
    {
        cout <<"Both the characters are same";
    }
}

```

#### **OUTPUT**

```

Enter a character : S
Enter another character : A
S is larger than A

```

**Explanation:** In the above program, two characters are entered and stored in the character variable j and k. The first if statement checks whether j is greater than k. Here,

comparison is done considering ASCII values. If the condition is true, a message will be displayed and program terminates. In case the condition is false the `else` block is executed. In the `else` block another `if` statement is present. The `if` statement inside the `else` block checks whether `k` is greater than `j` or not. If the condition is true `if` block is executed, otherwise `else` block is executed.

#### 4.6 Write a program to explain the concept of nested `if-else` statements.

```
#include<conio.h>
#include<iostream.h>
int main( )
{
int score;
clrscr( );
cout<<"\nEnter Sachin's score :";
cin>>score;
if(score>=50)
{
    if(score>=100)
        cout<<"Sachin scored a century and more runs";
    else
    {
        cout<<"\nSachin scored more than half century ";
        cout<<"\nCross your fingers and pray he completes century ";
    }
}
else
{
    if(score==0)
        cout<<"Oh my God ";
    if(score>0)
        cout<<"Not in form today ";
}
return 0;
}
```

#### OUTPUT:

Enter Sachin's score : 0

Oh my God !

Enter Sachin's score : 45

Not in form today

Enter Sachin's score : 60

Sachin scored more than half century

Cross your fingers and pray he completes century

Enter Sachin's score : 116

Sachin scored a century and more runs

**Explanation:** From above it can be seen that if score was greater than 50 and greater than 100 then a particular message is prompted. If score was greater than 50 but less than 100, then another set of messages is prompted. If, however, the score was less than 50 and equal to 0 then a particular message is prompted. If score was less than 50 but not 0 then another message is displayed.

#### 4.7 Write a program to execute `if` statement without any expression.

```

# include <iostream.h>
# include <constream.h>

void main( )
{
    int j;
    clrscr( );
    cout <<"\n Enter a value : ";
    cin>>j;

    if(j)
        cout<<"Wel Come";
    else
        cout<<"Good Bye";
}

```

## OUTPUT

**Enter a value : 0**

**Good Bye**

**Explanation:** In the above program, an integer variable *j* is declared. The user is asked to enter an integer value. The entered value is stored in the variable *j*. The *if* statement checks the value of *j*. As explained at the beginning of this chapter, 0 is considered as false and any non-zero value is true. Thus, when user enters 0, the *else* block is executed and when a non-zero value is entered, *if* block is executed.

## THE IF-ELSE-IF LADDER STATEMENT

A common programming construct is the if-else-if ladder, sometimes called the if-else-if staircase because of it's appearance. It's general form is

```

if ( expression) statement block1;
    else statement block2;
        if(expression) statement block3;
            else statement block4;
                if(expression) statement block5;
                    .
                    .
                    .
                    .
                    .
                    else statement blockn;

```

The conditions are evaluated from the top. As soon as a true condition is met, the associated statement block gets executed and rest of the ladder is bypassed. If none of the conditions are met then the final *else* block gets executed .If this '*else*' is not present and none of the '*if*' evaluates to true then entire ladder is bypassed.

Although the indentation of the preceding 'if-else-if' ladder is technically correct, it can lead to overly deep indentation. Imagine 256 (maximum allowed) such stairs and each indented: Enough to confuse!

This reason made it vital to use the form as shown below:

```

if ( expression)
    statement block1;      → if block

```

```

else    if (expression)
    statement block2;      → else block
else    if (expression)
    statement block3;      → else block
.
.
.
else
    statement block5;

```

The following program will clear your understanding.

#### **4.8 Write a program to simulate tariff charges for reaching different destinations by bus.**

```

#include<conio.h>
#include<iostream.h>
int main( )
{
int cost,ch;
clrscr( );
cout<<"\n.....NANDED BUS STATION.....\n";
cout<<".....Menu.....";
cout<<"\nBombay..1";
cout<<"\nNagpur..2";
cout<<"\nPune..3";
cout<<"\nAmravati..4";
cout<<"\nAurangabad..5";
cout<<"\n\nEnter your destination :";
cin>>ch;
if(ch==1)
    cost=100;
else if(ch==2)
    cost=70;
else if(ch==3)
    cost=50;
else if(ch==4)
    cost=60;
else if(ch==3)
    cost=40;
else
    cost=0;
if(cost!=0)
{
    cout<<"\n\nThe ticket cost is : Rs "<<cost;
    cout<<"\nPay the amount to get booked";
}
else
    cout<<"Sorry there's no bus to desired destination";
cin.get( );
return 0;
}

```

#### **OUTPUT:**

.....NANDED BUS STATION.....

.....Menu.....

Bombay..1

Nagpur..2

Pune..3

Amravati..4

**Aurangabad..5**

**Enter your destination : 4**

**The ticket cost is : Rs 60**

**Pay the amount to get booked**

**Explanation:** The program above simulates a bus station. We use an 'if-else-if' ladder to find out the cost of ticket to a particular station, if there was a bus to that station. There wasn't a bus to that station having cost=0.Finally if cost is non-zero it's printed. The user is prompted to enter the choice. Depending upon the choice, the fare of destination station is displayed. In the above example, choice 4 is given. The result displayed is

"The ticket cost is : Rs 60

Pay the amount to get booked".

## **4.5 THE JUMP STATEMENT**

C/C++ has four statements that perform an unconditional control transfer. These are `return( )`, `goto`, `break` and `continue`. Of these, `return( )` is used only in functions. The `goto` and `return( )` may be used anywhere in the program but `continue` and `break` statements may be used only in conjunction with a loop statement. In 'switch case' '`break`' is used most frequently.

## **4.6 THE GOTO STATEMENT**

This statement does not require any condition. This statement passes control anywhere in the program without least care for any condition. The general format for this statement is shown below:

```
goto label;
```

```
—
```

```
—
```

```
label:
```

where, `label` is any valid `label` either before or after `goto`.The label must start with any character and can be constructed with rules used for forming identifiers. Avoid using `goto` statement.

## **4.9 Write a program to demonstrate the use of goto statement.**

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

void main( )
{
    int x;
    clrscr( );
    cout <<"Enter a Number :";
    cin>>x;
    if (x%2==0)
        goto even;
    else
        goto odd;
```

```
even :cout<<x<<" is Even Number.";
return;
odd: cout<<x<<" is Odd Number.";
}
```

## OUTPUT

Enter a Number: 5

5 is Odd Number.

**Explanation:** In the above program, a number is entered. The number is checked for even or odd with modulus division operator. When the number is even, the `goto` statement transfers the control to the label `even`. Similarly, when the number is odd, the `goto` statement transfers the control to the label `odd` and respective messages will be displayed.

## 4.7 THE BREAK STATEMENT

The `break` statement allows the programmer to terminate the loop. The `break` skips from the loop or the block in which it is defined. The control then automatically passes on to the first statement after the loop or the block. The `break` statement can be associated with all the conditional statements (especially `switch( )` case). We can also use `break` statements in the nested loops. If we use `break` statement in the innermost loop, then the control of program is terminated from that loop only and resumes at the next statement following that loop. The widest use of this statement is in `switch` case where it is used to avoid flow of control from one case to other.

## 4.8 THE CONTINUE STATEMENT

The `continue` statement works quite similar to the `break` statement. Instead of forcing the control to end of loop (as it is in case of `break`), `continue` causes the control to pass on to the beginning of the block/loop. In case of `for` loop, the `continue` case causes the condition testing and incrementation steps to be executed (while rest of the statements following `continue` are neglected). For `while` and `do-while`, `continue` causes control to pass on to conditional tests. It is useful in programming situation when you want particular iterations to occur only up to some extent or you want to neglect some part of your code. The programs on `break` and `continue` can be performed by the programmer.

## 4.9 THE SWITCH CASE STATEMENT

The `switch` statement is a multi-way branch statement and an alternative to `if-else-if` ladder in many situations. This statement requires only one argument, which is then checked with number of case options. The `switch` statement evaluates the expression and then looks for its value among the case constants. If the value is matched with a case constant then that case constant is executed until a `break` statement is found or end of `switch` block is reached. If not then simply `default` (if present) is executed (if `default` isn't present then simply control flows out of the `switch` block).

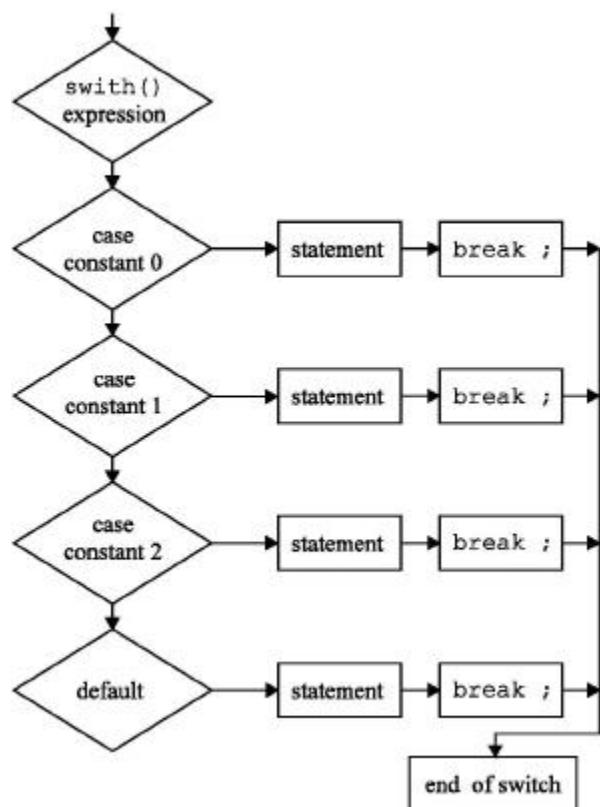
The `default` is normally present at the bottom of `switch` case structure. But we can also define `default` statement anywhere in the `switch` structure. The `default` block must not be empty. Every case statement terminates with : (colon). The `break` statement is used to

stop the execution of succeeding cases and pass the control to the switch end of block. The syntax of the `switch( )` case statement is shown below. [Figure 4.3](#) simulates the working of `switch( ) case statement`.

```
switch(variable or expression)
{
    case constant A :
        statement;
        break;

    case constant B :
        statement;
        break;

    default :
        statement ;
}
```



**Fig. 4.3** The switch case statement

**4.10 Write a program to display different lines according to users choice.  
Use `switch( ) case statement`.**

```
#include<stdlib.h>
#include<constream.h>
#include<iostream.h>
void main( )
{
int c;
```

```

clrscr( );

cout<<"\n LINE FORMAT MENU ";
cout<<"\n1] *****";
cout<<"\n2] ======";
cout<<"\n3] ~~~~~~~~";
cout<<"\n4] _____";
cout<<"\nEnter your choice :";

cin>>c;

switch(c)
{
    case 1:
    cout <<"\n*****";
    break;

    case 2:
    cout <<"\n=====";
    break;

    case 3:
    cout <<"\n~~~~~~~";
    break;

    case 4:
    cout <<"\n_____";
    break;

    default:
    cout <<"\n.....";
}
}

```

## **OUTPUT**

### **LINE FORMAT MENU**

```

1] *****
2] =====
3] ~~~~~~~~
4] __

```

**Enter your choice :2**

```

=====

```

**Explanation:** In the above program, a menu is displayed on the screen with different type of lines. The user enters a number as given in the menu. The `switch( )` statement checks the value of variable `c`. All case statements are tested and one that satisfies, is executed. If the user enters a value other than that listed in the menu, `default` statement is executed.

**4.11 Write a program to enter a month number of year 2002 and display the number of days present in that month.**

```

#include <iostream.h>
#include<constream.h>
void main( )
{
clrscr( );

int month,days;
cout<<"Enter a month of year 2002 :";
cin>>month;

switch(month)
{
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        days=31;
        break;
    case 2:
        days=28;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        days=30;
        break;
}
cout<<"\nNumber of days in this month are "<<days;
}

```

## **OUTPUT**

**Enter a month of year 2002 :3**

**Number of days in this month are 31**

**Explanation:** The above program is meant to calculate number of days in a month (given by user) of year 2002. We know that month numbers 1, 3, 5, 7, 8, 10 and 12 have 31 days each. So they can be put together in `switch( ) case`. But since there isn't such an option we have used the technique shown above. Similar is the case with month numbers 4, 6, 9 and 11 having 30 days each.

## **4.10 THE NESTED SWITCH( ) CASE STATEMENT**

C/C++ supports the nesting of `switch( ) case`. The inner `switch` can be part of an outer `switch`. The inner and the outer `switch case constants` may be the same. No conflict arises even if they are same. The example below demonstrates this concept.

## **4.12 Write a program to demonstrate nested switch( ) case statement.**

```

#include<iostream.h>
#include<constream.h>
void main( )
{

```

```

int x;
clrscr( );

cout<<"\nEnter a number :";
cin>>x;
switch(x)
{
    case 0:
        cout<<"\nThe number is 0 ";
        break;

    default:
        int y;
        y=x%2;
        switch(y)
        {
            case 0:
                cout<<"\nThe number is even ";
                break;
            case 1:
                cout<<"\nThe number is odd ";
                break;
        }
}
}

```

## **OUTPUT**

**Enter a number :5**

**The number is odd**

**Explanation:** The above program identifies whether input number was zero, even or odd. The first `switch( )` case finds out whether the number is zero or non-zero. If a non-zero value is entered, a default statement of first `switch` case statement is executed which executes another nested `switch( )` case. The nested `switch( )` case statement determines whether the number is even or odd and displays respective messages.

## **4.11 LOOPS IN C/C++**

C/C++ provides loop structures for performing some tasks which are repetitive in nature. The C/C++ language supports three types of loop control structures. Their syntax is described in Table 4.1.

The `for` loop comprises of three actions. The three actions are placed in the `for` statement itself. The three actions initialize counter, test condition, and re-evaluation parameters are included in one statement. The expressions are separated by semi-colons (`;`). This helps the programmer to visualize the parameters easily. The `for` statement is equivalent to the `while` and `do-while` statements. The only difference between `for` and `while` is that the latter checks the logical condition and then executes the body of the loop, whereas the `for` statement test is always performed at the beginning of the loop. The body of the loop may not be executed at all times if the condition fails at the beginning. The `do-while` loop executes the body of the loop at least once regardless of the logical condition.

**Table 4.1 Loops in C++**

<b>for</b>	<b>while</b>	<b>do-while</b>
<pre>for (expression -1;       expression-2;       expression-3) statement ;</pre>	<pre>expression -1; while (expression -2) { statement; expression -3; }</pre>	<pre>expression -1; do { statement; expression-3; } while (expression-2);</pre>

## 4.12 THE FOR LOOP

The **for** loop allows execution of a set of instructions until a condition is met. Condition may be predefined or open-ended. Although all programming languages provide **for** loops, still the power and flexibility provided by C/C++ is worth mentioning. The general syntax for the **for** loop is given below:

### Syntax of for loop

```
for (initialization; condition; increment/decrement)
Statement block;
```

Though many variations of **for** loop are allowed, the simplest form is shown above.

The *initialization* is an assignment statement that is used to set the loop control variable(s). The condition is a relational expression that determines the number of the iterations desired or the condition to exit the loop. The *increment* or the re-evaluation parameter decides how to change the state of the variable(s) (quite often increase or decrease so as to approach the limit). These three sections must be separated by semi-colons. The body of the loop may consist of a block of statements (which have to be enclosed in braces) or a single statement (enclosure within braces is not compulsory but advised).

Following programs illustrate **for** loop:

4.13	4.14	4.15
<pre># include &lt;iostream.h&gt; # include &lt;constream.h&gt;  void main( ) { clrscr( ); int j;  for (j=1;j&lt;11;j++) cout&lt;&lt;" "&lt;&lt;j; }</pre>	<pre># include &lt;iostream.h&gt; # include &lt;constream.h&gt;  void main( ) { clrscr( ); int j=0;  for (;j&lt;11;j++) cout&lt;&lt;" "&lt;&lt;j; }</pre>	<pre># include &lt;iostream.h&gt; # include &lt;constream.h&gt;  void main( ) { clrscr( ); int j=0;  for (;j&lt;11;) cout&lt;&lt;" "&lt;&lt;j; ++j; }</pre>
<b>OUTPUT</b> <b>1 2 3 4 5 6 7 8 9 10</b>	<b>OUTPUT</b> <b>1 2 3 4 5 6 7 8 9 10</b>	<b>OUTPUT</b> <b>1 2 3 4 5 6 7 8 9 10</b>
Explanation: In this program initialization,	Explanation: In this program initialization is done before <b>for</b> statement.	Explanation: In this program initialization is done before <b>for</b> statement. Increment is

condition, and increment is done in single parenthesis.	In <code>for</code> statement only condition and increment is done.	done inside the loop. The <code>for</code> loop contains only condition.
---	---	--

#### 4.16 Write a program to display all leap years from 1900 to 2002.

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

void main( )
{
    int i=1900;
    clrscr( );
    cout<<"\nProgram to print all the leap years from 1900 to 2002
\n\n";

for(;i++<=2002;)
{
    if(i%4==0&&i%100!=0)
        cout<<i<<" ";
    else if(i%100==0&&i%400==0)
        cout<<i<<" ";
}
}
```

**Explanation:** One must be aware that only those years that are

- (a) divisible by 4 and not 100,
- (b) divisible by 100 and also 400,

alone are qualified to be called as leap years. The above program checks for these conditions for all years from 1900 to 2002 and then prints the leap years. Stress is on `for` loop construct and use of `if` and `else-if` in the body.

#### 4.13 NESTED FOR LOOPS

We can also nest `for` loops to gain more advantage in some situations. The nesting level is not restricted at all. In the body of a `for` loop, any number of sub `for` loop(s) may exist.

#### 4.17 Write a program to demonstrate nested `for` loops.

```
# include <iostream.h>
# include <constream.h>>
void main( )
{
int a,b,c;
clrscr( );
for (a=1;a<=2;a++) /* outer loop */
{
for (b=1;b<=2;b++) /* middle loop */
{
    for (c=1;c<=2;c++) /* inner loop */
        cout <<"\n a=<<a <<" + b="<<b <<" + c="<<c <<" :"<<a+b+c;
        cout <<"\n Inner Loop Over.";
}

cout <<"\n Middle Loop Over.";
}
```

```

cout<<"\n Outer Loop Over.";
}

```

**Explanation:** The above program is executed in the sequence shown below. The total number of iterations are equal to  $2*2*2=8$ . The final output provides 8 results. Table 4.2 shows the working of this program.

#### 4.14 THE WHILE LOOP

Another kind of loop structure in C/C++ is the while loop. It's format is given below.

**Syntax:**

```

while (test condition)
{
    body of the loop
}

```

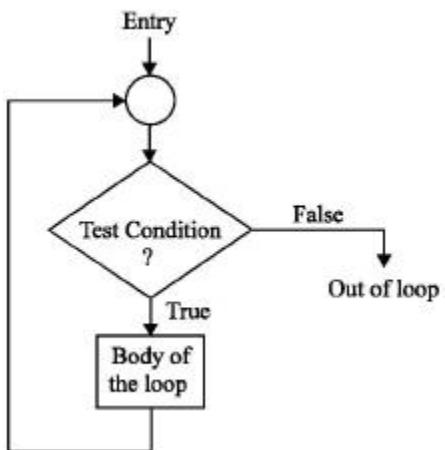
The test condition may be any expression. The loop statements will be executed till the condition is true i.e., the test condition is evaluated and if the condition is true, then the body of the loop is executed. When the condition becomes false the execution will be out of the loop.

The execution of the loop is shown in Figure 4.4.

The block of the loop may contain single statement or a number of statements. The same block can be repeated. The braces are needed only if the body of the loop contains more than one statement. However, it is a good practice to use braces even if the body of the loop contains only one statement.

**Table 4.2** Working of program 4.17

Values of loop variables			Output
Outer	Middle	Inner	
(1) a=1	b=1	c=1	a+b+c=3
2) a=1	b=1	c=2	a+b+c=4
<b>Inner Loop Over</b>			
(3) a=1	b=2	c=1	a+b+c=4
(4) a=1	b=2	c=2	a+b+c=5
<b>Inner Loop Over</b>			
<b>Middle Loop Over</b>			
(5) a=2	b=1	c=1	a+b+c=4
(6) a=2	b=1	c=2	a+b+c=5
<b>Inner Loop Over</b>			
(7) a=2	b=2	c=1	a+b+c=5
(8) a=2	b=2	c=2	a+b+c=6
<b>Inner Loop Over</b>			
<b>Middle Loop Over</b>			
<b>Outer Loop Over.</b>			



**Steps of while loops are as follows:**

- (1) The test condition is evaluated and if it is true, the body of the loop is executed.
- 2) On execution of the body, test condition is repetitively checked and if it is true the body is executed.
- 3) The process of execution of the body will be continued till the test condition becomes false.
- 4) The control is transferred out of the loop.

**Fig. 4.4** The while loop

**4.18 Write a program to add 10 consecutive numbers starting from 1. Use the while loop.**

```

#include <iostream.h>
#include <constream.h>

void main( )
{
    int a=1,sum=0;
    clrscr( );

    while(a<=10)
    {
        cout <<" "<<a;
        sum=sum+a;
        a++;
    }
    cout <<"\n Sum of 10 numbers : "<<sum;
}

```

**OUTPUT :**

**1 2 3 4 5 6 7 8 9 10**

**Sum of 10 numbers : 55**

**Explanation:** In the above program, integer variable a is initialized to 1 and variable sum to 0. The while loop checks the condition for  $a \leq 10$ . The variable a is added to variable sum and each time a is incremented by 1. In each while loop a is incremented and added to sum. When the value of a reaches to 10, the condition given in while loop becomes false. At last the loop is terminated. The sum of the number is displayed.

**4.19 Write a program to calculate the sum of individual digits of an entered number.**

```

#include <iostream.h>
#include <constream.h>
void main( )
{

```

```

int num,t;
clrscr( );
cout<<"\n enter a number : ";
cin>>num;
t=num;
int sum=0;
while(num)           //condition is true until num!=0
{
sum=sum+num%10;
num=num/10;
}
cout<<"\n Sum of the individual digits of the number "<<t<<" is =
"<<sum;
}

OUTPUT

```

**Enter a number : 234**

**Sum of the individual digits of the number 234 is = 9**

**Explanation:** The logic behind the program is to extract each time the LSD(lower significant digit) and divide the number by 10 so as to shift it by one place. To extract the LSD we use the fact that  $\text{num} \% 10$  = the remainder on dividing num by 10.

#### **4.20 Write a program to check whether the entered number is palindrome or not.**

```

# include <iostream.h>
# include <constream.h>
void main( )
{
int num;
clrscr( );
cout<<"\n Enter the number : ";
cin>>num;
int b=0,a=num;
while(a)
{
b=b*10+a%10;
a=a/10;
}
if(num==b)
cout<<"\n The given number is palindrome ";
else
cout<<"\n The given number is not palindrome ";
}

OUTPUT

```

**Enter the number: 121**

**The given number is palindrome**

**Explanation:** Palindrome is a number that is equal to it's reverse, for example, 51715. The program above first simply reverses the number and stores it in b. Then it compares b with original number to tell you whether or not the number is a palindrome. See that in each interaction the LSD of a is being made MSD of b. a is being shifted left and b is shifted right by one digit each in every iteration. Iterations are done until a exhausts or becomes equal to 0.

#### **4.15 THE DO-WHILE LOOP**

The format of do-while loop in C/C++ is given below.

```
do
{
statement/s;
}
while (condition);
```

while (condition); The difference between the while and do-while loop is the place where the condition is to be tested. In the while loop, the condition is tested following the while statement and then the body gets executed. Where as in do-while, the condition is checked at the end of the loop. The do-while loop will execute at least once even if the condition is false initially. The do-while loop executes until the condition becomes false.

#### **4.21 Write a program using do-while loop to print numbers and their cubes up to 10.**

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

void main( )
{
int y,x=1;
clrscr( );

cout <<"\n\tNumbers and their cubes \n";

do
{   y=pow(x,3);
    cout <<"\t" <<x <<"\t\t" <<y <<"\n";
    x++;
}   while (x<=10);
}
```

#### **OUTPUT**

#### **Numbers and their cubes**

**1**

**1**

**2**

**8**

**3**

**27**

**4**

**64**

**5**

**125**

**6**

**216**

**7**

**343**

**8**

**512**

**9**

**729**

**10**

**1000**

**Explanation:** Here, the mathematical function `pow (x, 3)` is used. Its meaning is to calculate the third power of x. With this function we get the value of  $y=x^3$ . For use of the `pow ( )` function we have to include `math.h` header file.

### SUMMARY

- (1) This chapter teaches the basics of control structures of C++ language.
- (2) The C++ control structure covers two sets of statements. The set that performs certain operations repetitively is called as loop statements and the set that makes decision is known as decision-making statements.
- (3) The simple `if` statement executes statement only if the condition is true, otherwise, it follows the next statement. The `else` keyword is used when the expression is not true. The `else` keyword is optional.
- (4) C/C++ has four statements that perform an unconditional control transfer. These are `return ( )`, `goto`, `break`, and `continue`. Of these, `return ( )` is used only in functions. `goto` and `return ( )` may be used anywhere in the program but `continue` and `break` statements may be used only in conjunction with a loop statement. `break` is used most frequently in `switch case`.
- (5) The `for` loop comprises of three actions. The three actions are placed in the `for` statement itself. The three actions ***initialize counter, test condition, and re-evaluation parameters*** are included in one statement. The `for` statement is equivalent to the `while` and `do-while` statement. The only difference between `for` and `while` is that the latter checks the logical condition and then executes the body of the loop, whereas the `for` statement test is always performed at the beginning of the loop. The body of the loop may not be executed at all times if the condition.

### EXERCISES

#### [A] Answer the following questions.

- (1) Explain the need of control structures in C++.
- (2) What are the differences between `break` and `continue` statements?
- (3) Why `goto` statement is not commonly used?
- (4) Explain the working of `if-else` statement.
- (5) Explain the working of `switch ( ) case` statement.

- (6) Explain the role of `break` statement in `switch( ) case`.
- (7) What are the differences between `while` and `do-while` loop statements?
- (8) What is an infinite loop?
- (9) Explain nested if's.
- (10) Explain nested `switch( ) case` statement.
- (11) Explain the use of the keyword `default`.

**[B] Answer the following by selecting the appropriate option.**

- (1) Which of the following loop statement uses two keywords?
  - (a) `do-while` loop
  - (b) `for` loop
  - (c) `while` loop
  - (d) none of the above
- (2) The statement which requires at least one statement followed by it is
  - (a) `default`
  - (b) `continue`
  - (c) `break`
  - (d) `else`
- (3) The loop statement terminated by a semi-colon is
  - (a) `do-while` loop
  - (b) `for` loop
  - (c) `while` loop
  - (d) none of the above
- (4) Every expression always returns
  - (a) 0 or 1
  - (b) 1 or 2
  - (c) -1 or 0
  - (d) none of the above
- (5) The meaning of `if(1)` is
  - (a) always true
  - (b) always false
  - (c) both (a) and (b)
  - (d) none of the above
- (6) The curly braces are not present; the scope of loop statement is
  - (a) one statement
  - (b) two statements
  - (c) four statements
  - (d) none of the above
- (7) In `nested` loop
  - (a) the inner most loop is completed first
  - (b) the outer most loop is completed first
  - (c) both (a) and (b)
  - (d) none of the above

**[C] Attempt the following programs.**

- (1) Write a program to display numbers from 10 to 1 using `for` loop.
- (2) Write a program to calculate the factorial of a given number.
- (3) Write a program to display only even numbers in between 1 to 150.
- (4) Write a program to solve the series  $x=1/2!+1/4!+1/n!$ .
- (5) Write a program to calculate the sum of numbers between 1 to N numbers. The user enters the value of N.
- (6) Write a program to use `break` and `continue` statements.
- (7) Write a program to display alphabets A to Z using `while` loop.
- (8) Write a program to use `break` statement and terminate the loop.
- (9) Write a program to demonstrate the use of `continue` statement.

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

- [Settings](#)

# 4

## CHAPTER

# Control Structures

C H A P T E R  O U T L I	<ul style="list-style-type: none"><li>—• <a href="#">4.1 Introduction</a></li><li>—• <a href="#">4.2 Decision-Making Statements</a></li><li>—• <a href="#">4.3 The if-else Statement</a></li><li>—• <a href="#">4.4 The Nested if-else Statement</a></li><li>—• <a href="#">4.5 The jump Statement</a></li><li>—• <a href="#">4.6 The goto Statement</a></li></ul>
--	--

- [4.7 The break Statement](#)
- [4.8 The continue Statement](#)
- [4.9 The switch case Statement](#)
- [4.10 The Nested switch\( \)case Statement](#)
- [4.11 Loops in C/C++](#)
- [4.12 The for Loop](#)
- [4.13 Nested for Loops](#)
- [4.14 The while Loop](#)
- [4.15 The do-while Loop](#)

## 4.1 INTRODUCTION

This chapter deals with the basics of control structures of C++ language. Those who are already familiar with C, may skip this chapter. Those who are new to C++, should read this chapter thoroughly. As discussed earlier, C++ is a superset of C. The control structures of C and C++ are same. In this chapter, the concepts of structures are illustrated in detail for the new users.

A Program is nothing but a set of statements written in sequential order, one after the other. These statements are executed one after the other. Sometimes it may happen that the programmer requires to alter the flow of execution, or to perform the same operation for fixed iterations or whenever the condition does not satisfy. In such a situation the programmer uses the control structure. There are various control structures supported by C++. Programs which use such (one or three) control structures are said to be structured programs.

The C++ control structure covers two sets of statements. The set that performs certain operations repetitively is called as loop statements and the set that makes decision is known as decision-making statements.

## 4.2 DECISION-MAKING STATEMENTS

Following are decision-making statements.

- (1) The if statement
- (2) The switch ( ) case statement.

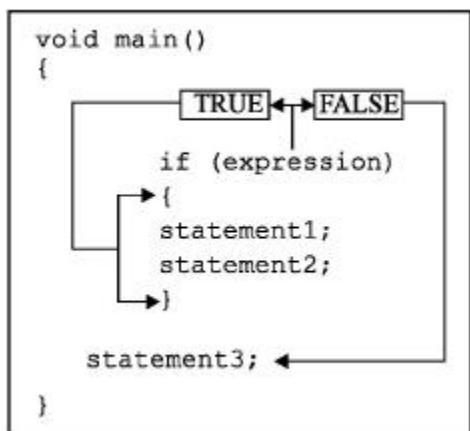
**Simple if statement** The syntax of simple if statement is described below

**Syntax** for the simplest if statement:

```
if (expression) /* no semi-colon */
    Statement;
```

The if statement contains an expression. The expression is evaluated. If the expression is true it returns 1 otherwise 0. The value 1 or any non-zero value is considered as true and 0 as false. In C++, the values (1) true and (0) false are known as bool type data.

The bool type data occupies one byte in memory. If the given expression in the if ( ) statement is true, the following statement or block of statements too are executed, otherwise the statement that appears immediately after if block (true block) is executed as given in [Figure 4.1](#).



**Fig. 4.1** The simple if statement

As shown in [Figure 4.1](#), the expression is always evaluated to true or false. When the expression is true, the statement in if block is executed. When the expression is false the if block is skipped and the statement (state-ment3) after if block is executed.

The expression given in the if statement may not be always true or false. For example, if(1) or if(0). When such statement is encountered, the compiler will display a warning message "condition is always true" or "condition is always false". In place of expression we can also use function that returns 0 or 1 return values. The following programs illustrate the points discussed above.

#### 4.1 Write a program to enter age and display message whether the user is eligible for voting or not.

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

void main( )
    clrscr( );
    int age;
```

```

cout<<"Enter Your Age :";
cin>>age;
if (age>=18)
{
    cout <<" You are eligible for voting.";
}

```

## **OUTPUT**

**Enter Your Age : 23**

**You are eligible for voting.**

**Explanation:** In the above program, integer variable age is declared. The user enters his/her age. The value is tested with if statement. If the age is greater than or equal to 18 the statement followed by if statement is executed. The message displayed will be "You are eligible for voting." If the condition is false nothing is displayed.

## **4.2 Write a program to use library function with if statement instead of expression.**

```

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

void main( )
{
    static char nm[]="Hello";
    clrscr( );

    if (strlen(nm))
    { cout <<" The string is not empty."; }

}

```

## **OUTPUT**

**The string is not empty.**

**Explanation:** In the above program, the character array nm[] is initialized with the string "Hello". The strlen( ) function is used in the if statement. The strlen( ) function calculates the length of the string and returns it. If the string is empty it returns (0) false otherwise non-zero value (string length). The non-zero value is considered as true.

The strlen( ) function returns non-zero value (true). The if statement displays the message "The string is not empty." Here, instead of expression, a library function is used.

## **4.3 THE IF-ELSE STATEMENT**

The simple if statement executes statement only if the condition is true otherwise it follows the next statement. The else keyword is used when the expression is not true. The else keyword is optional. The syntax of if..else statement is given below and Figure 4.2 describes the working of if..else statement.

The format of if..else statement is as follows:

```

if(the condition is true)
execute the Statement1;
else

```

```

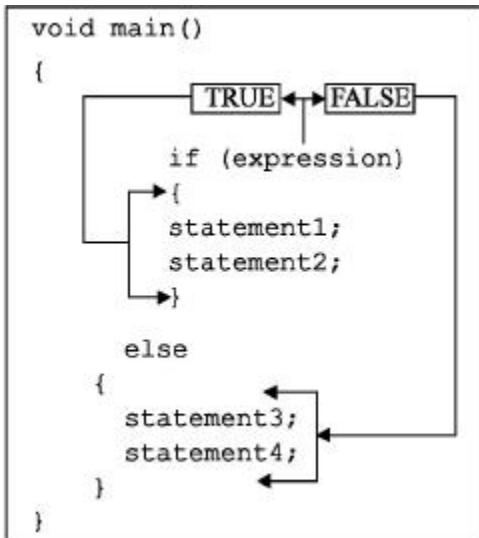
execute the Statement2;
OR
Syntax of if - else statement can be given as follows:
if ( expression is true)

{
    statement 1;      →      if block
    statement 2;
}

else

{
    statement 3;      →      else block
    statement 4;
}

```



**Fig.4.2** The if-else statement

As shown in [Figure 4.2](#), both `if` and `else` block contain statements. When the expression is true `if` block is executed otherwise `else` block is executed.

### 4.3 Write a program to enter age and display message whether the user is eligible for voting or not. Use if-else statement.

```

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

void main( )
{
    clrscr( );

    int age;

```

```

cout<<"Enter Your Age : ";
cin>>age;

if (age>=18)
{
    cout <<" You are eligible for voting.";
}
else
{
    cout <<" You are noneligible for voting"<<endl;
    cout<< " Wait for "<<18-age<<" year(s).";
}
}

```

**OUTPUT:**

**Enter Your Age : 17**  
**You are noneligible for voting**  
**Wait for 1 year(s).**

**Explanation:** In the above program, the user enters his/her age. The integer variable age is used to store the value. The if statement checks the value of age. If the age is greater than or equal to 18, the if block of statement is executed, otherwise the else block of statement is executed. Thus, the else statement extends if statement.

**4.4 Write a program with simple if statement. If entered score is less than 50 display one message, otherwise another message.**

```

#include<iostream.h>
#include<conio.h>
int main( )
{
int v;
clrscr( );
cout<<"Enter a number : ";
cin>>v;
if(v>=50)
cout<<"\nCongrats !! You scored a half / more than half century ";
else
if(v<50)
cout<<"\nCome on !! You can score a half century ";
return 0;
}

```

**OUTPUT:**

**Enter a number : 49**  
**Come on !! You too can score a half century**

**Enter a number : 56**  
**Congrats !! You scored a half / more than half century**

**Explanation:** The program first prompts the user to enter a number 'v'. If it is greater than 50 or equal to 50, it prompts a particular message, otherwise for less than 50, other message is displayed. If found true, a particular message is prompted, otherwise another.

#### **4.4 THE NESTED IF-ELSE STATEMENT**

In this kind of statements, number of logical conditions are checked for executing various statements. Here, if any logical condition is true, the compiler executes the block followed by `if` condition, otherwise it skips and executes `else` block.

In `if..else` statement, `else` block is executed by default after failure of `if` condition. In order to execute the `else` block depending upon certain conditions we can add repetitively `if` statements in `else` block. This kind of nesting will be unlimited.

**Syntax** of `if-else...if` statement can be given as follows:

```
if ( condition )
{
    statement 1;           -> if  block
    statement 2;
}

else if (condition)
{
    statement 3;           -> else block
    statement4;
}

else
{
    statement5;
    statement6;
}
```

From the above block, following rules can be described for applying nested `if..else..if` statements:

1. Nested `if..else` can be chained with one another.
2. If the condition is false, control passes to `else` block where condition is again checked with the `if` statement. This process continues till there is no `if` statement in the last `else` block.
3. If one of the `if` statement satisfies the condition, other nested `if..else` will not be executed.

Following programs illustrates the working of nested `if-else` statements.

#### **4.5 Write a program to enter two characters and display the larger character. Use nested if-else statements.**

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

void main( )
{
    char j,k;
    clrscr( );
    cout<<"\n Enter a character :";
    j=getche( );
    cout <<"\n Enter another character : ";
    k=getche( );
    cout<<endl;

    if (j>k)
```

```

    {
        cout <<j <<" is larger than "<<k;
    }
    else if(k>j)
    {
        cout <<k <<" is larger than "<<j;
    }
    else
    {
        cout <<"Both the characters are same";
    }
}

```

## OUTPUT

**Enter a character : S  
Enter another character : A  
S is larger than A**

**Explanation:** In the above program, two characters are entered and stored in the character variable j and k. The first if statement checks whether j is greater than k. Here, comparison is done considering ASCII values. If the condition is true, a message will be displayed and program terminates. In case the condition is false the else block is executed. In the else block another if statement is present. The if statement inside the else block checks whether k is greater than j or not. If the condition is true if block is executed, otherwise else block is executed.

## 4.6 Write a program to explain the concept of nested if-else statements.

```

#include<conio.h>
#include<iostream.h>
int main( )
{
int score;
clrscr( );
cout<<"\nEnter Sachin's score :";
cin>>score;
if(score>=50)
{
    if(score>=100)
        cout<<"Sachin scored a century and more runs";
    else
    {
        cout<<"\nSachin scored more than half century ";
        cout<<"\nCross your fingers and pray he completes century ";
    }
}
else
{
    if(score==0)
        cout<<"Oh my God ";
    if(score>0)
        cout<<"Not in form today ";
}
return 0;
}

```

**OUTPUT:**

Enter Sachin's score : 0

Oh my God !

Enter Sachin's score : 45

Not in form today

Enter Sachin's score : 60

Sachin scored more than half century

Cross your fingers and pray he completes century

Enter Sachin's score : 116

Sachin scored a century and more runs

**Explanation:** From above it can be seen that if score was greater than 50 and greater than 100 then a particular message is prompted. If score was greater than 50 but less than 100, then another set of messages is prompted. If, however, the score was less than 50 and equal to 0 then a particular message is prompted. If score was less than 50 but not 0 then another message is displayed.

**4.7 Write a program to execute if statement without any expression.**

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

void main( )
{
    int j;
    clrscr( );
    cout <<"\n Enter a value : ";
    cin>>j;

    if(j)
        cout<<"Wel Come";
    else
        cout<<"Good Bye";
}
```

**OUTPUT**

Enter a value : 0

Good Bye

**Explanation:** In the above program, an integer variable *j* is declared. The user is asked to enter an integer value. The entered value is stored in the variable *j*. The *if* statement checks the value of *j*. As explained at the beginning of this chapter, 0 is considered as false and any non-zero value is true. Thus, when user enters 0, the *else* block is executed and when a non-zero value is entered, *if* block is executed.

**THE IF-ELSE-IF LADDER STATEMENT**

A common programming construct is the if-else-if ladder, sometimes called the if-else-if staircase because of its appearance. Its general form is

```

if ( expression) statement block1;
else statement block2;
if(expression) statement block3;
else statement block4;
if(expression) statement block5;
.
.
.
.
.
else statement blockn;

```

The conditions are evaluated from the top. As soon as a true condition is met, the associated statement block gets executed and rest of the ladder is bypassed. If none of the conditions are met then the final `else` block gets executed .If this '`else`' is not present and none of the '`if`' evaluates to true then entire ladder is bypassed.

Although the indentation of the preceding '`if-else-if`' ladder is technically correct, it can lead to overly deep indentation. Imagine 256 (maximum allowed) such stairs and each indented: Enough to confuse!

This reason made it vital to use the form as shown below:

```

If ( expression)
    statement block1;      → if block
else    if (expression)
    statement block2;      → else block
else    if (expression)
    statement block3;      → else block
.
.
.
else
    statement block5;

```

The following program will clear your understanding.

#### **4.8 Write a program to simulate tariff charges for reaching different destinations by bus.**

```

#include<conio.h>
#include<iostream.h>
int main( )
{
int cost,ch;
clrscr( );
cout<<"\n.....NANDED BUS STATION.....\n";
cout<<".....Menu.....";
cout<<"\nBombay..1";
cout<<"\nNagpur..2";
cout<<"\nPune..3";
cout<<"\nAmravati..4";
cout<<"\nAurangabad..5";
cout<<"\nEnter your destination :";
cin>>ch;
if(ch==1)
    cost=100;
else if(ch==2)
    cost=70;
else if(ch==3)
    cost=50;
else if(ch==4)

```

```

    cost=60;
else if(ch==3)
    cost=40;
else
    cost=0;
if(cost!=0)
{
    cout<<"\n\nThe ticket cost is : Rs "<<cost;
    cout<<"\nPay the amount to get booked";
}
else
    cout<<"Sorry there's no bus to desired destination";
cin.get( );
return 0;
}

```

#### **OUTPUT:**

.....NANGED BUS STATION.....

.....Menu.....

Bombay..1

Nagpur..2

Pune..3

Amravati..4

Aurangabad..5

Enter your destination : 4

The ticket cost is : Rs 60

Pay the amount to get booked

**Explanation:** The program above simulates a bus station. We use an 'if-else-if' ladder to find out the cost of ticket to a particular station, if there was a bus to that station. There wasn't a bus to that station having cost=0. Finally if cost is non-zero it's printed. The user is prompted to enter the choice. Depending upon the choice, the fare of destination station is displayed. In the above example, choice 4 is given. The result displayed is

"The ticket cost is : Rs 60

Pay the amount to get booked".

## **4.5 THE JUMP STATEMENT**

C/C++ has four statements that perform an unconditional control transfer. These are `return( )`, `goto`, `break` and `continue`. Of these, `return()` is used only in functions. The `goto` and `return( )` may be used anywhere in the program but `continue` and `break` statements may be used only in conjunction with a loop statement. In 'switch case' 'break' is used most frequently.

## **4.6 THE GOTO STATEMENT**

This statement does not require any condition. This statement passes control anywhere in the program without least care for any condition. The general format for this statement is shown below:

`goto label;`

—

—

label:

where, label is any valid label either before or after goto. The label must start with any character and can be constructed with rules used for forming identifiers. Avoid using goto statement.

#### 4.9 Write a program to demonstrate the use of goto statement.

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

void main( )
{
    int x;
    clrscr( );
    cout <<"Enter a Number :";
    cin>>x;
    if (x%2==0)
        goto even;
    else
        goto odd;

even :cout<<x<<" is Even Number.";
return;
odd: cout<<x<<" is Odd Number.";
}
```

#### OUTPUT

Enter a Number: 5

5 is Odd Number.

**Explanation:** In the above program, a number is entered. The number is checked for even or odd with modulus division operator. When the number is even, the goto statement transfers the control to the label even. Similarly, when the number is odd, the goto statement transfers the control to the label odd and respective messages will be displayed.

#### 4.7 THE BREAK STATEMENT

The break statement allows the programmer to terminate the loop. The break skips from the loop or the block in which it is defined. The control then automatically passes on to the first statement after the loop or the block. The break statement can be associated with all the conditional statements (especially switch( ) case). We can also use break statements in the nested loops. If we use break statement in the innermost loop, then the control of program is terminated from that loop only and resumes at the next statement following that loop. The widest use of this statement is in switch case where it is used to avoid flow of control from one case to other.

#### 4.8 THE CONTINUE STATEMENT

The continue statement works quite similar to the break statement. Instead of forcing the control to end of loop (as it is in case of break), continue causes the control to pass on to the beginning of the block/loop. In case of for loop, the continue case causes the

condition testing and incrementation steps to be executed (while rest of the statements following `continue` are neglected). For `while` and `do-while`, `continue` causes control to pass on to conditional tests. It is useful in programming situation when you want particular iterations to occur only up to some extent or you want to neglect some part of your code. The programs on `break` and `continue` can be performed by the programmer.

#### 4.9 THE SWITCH CASE STATEMENT

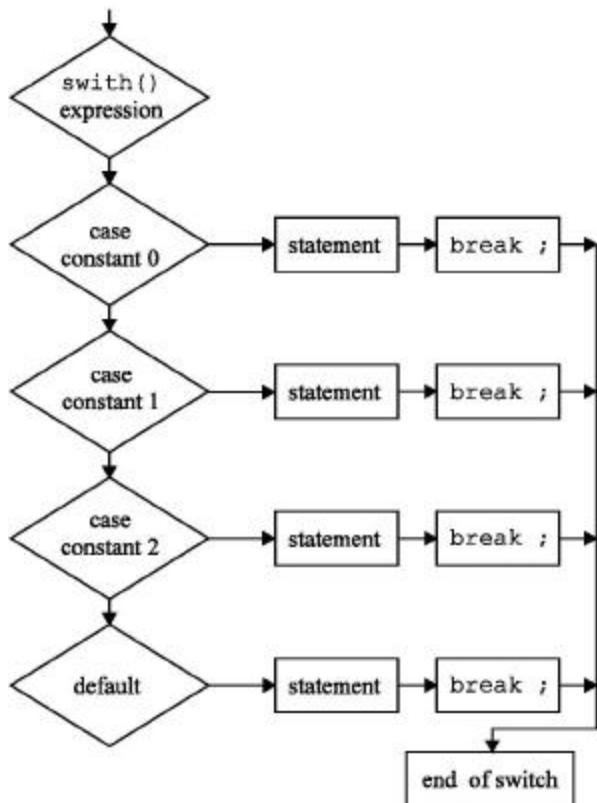
The `switch` statement is a multi-way branch statement and an alternative to `if-else-if` ladder in many situations. This statement requires only one argument, which is then checked with number of case options. The `switch` statement evaluates the expression and then looks for its value among the case constants. If the value is matched with a case constant then that case constant is executed until a `break` statement is found or end of `switch` block is reached. If not then simply `default` (if present) is executed (if `default` isn't present then simply control flows out of the `switch` block).

The `default` is normally present at the bottom of `switch` case structure. But we can also define `default` statement anywhere in the `switch` structure. The `default` block must not be empty. Every case statement terminates with `:` (colon). The `break` statement is used to stop the execution of succeeding cases and pass the control to the `switch` end of block. The syntax of the `switch( )` case statement is shown below. [Figure 4.3](#) simulates the working of `switch( )` case statement.

```
switch(variable or expression)
{
    case constant A :
        statement;
        break;

    case constant B :
        statement;
        break;

    default :
        statement ;
}
```



**Fig. 4.3** The switch case statement

#### 4.10 Write a program to display different lines according to users choice.

Use **switch( ) case statement**.

```

#include<stdlib.h>
#include<constream.h>
#include<iostream.h>
void main( )
{
int c;
clrscr( );

cout<<"\n LINE FORMAT MENU ";
cout<<"\n1] *****";
cout<<"\n2] ======";
cout<<"\n3] ~~~~~~";
cout<<"\n4] _____";
cout<<"\nEnter your choice :";

cin>>c;

switch(c)
{
    case 1:
        cout <<"\n*****";
    }
}

```

```

        break;

    case 2:
    cout <<"\n=====";
    break;

    case 3:
    cout <<"\n~~~~~";
    break;

    case 4:
    cout <<"\n_____";
    break;

default:
cout <<"\n.....";
}
}

```

## **OUTPUT**

### **LINE FORMAT MENU**

- 1] \*\*\*\*\*
- 2] =====
- 3] ~~~~~
- 4] \_\_\_\_

**Enter your choice :2**

**Explanation:** In the above program, a menu is displayed on the screen with different type of lines. The user enters a number as given in the menu. The switch( ) statement checks the value of variable c. All case statements are tested and one that satisfies, is executed. If the user enters a value other than that listed in the menu, default statement is executed.

### **4.11 Write a program to enter a month number of year 2002 and display the number of days present in that month.**

```

# include <iostream.h>
#include<constream.h>
void main( )
{
clrscr( );

int month,days;
cout<<"Enter a month of year 2002 :";
cin>>month;

switch(month)
{
    case 1:
    case 3:
    case 5:
    case 7:

```

```

    case 8:
    case 10:
    case 12:
    days=31;
    break;
    case 2:
    days=28;
    break;
    case 4:
    case 6:
    case 9:
    case 11:
    days=30;
    break;
}
cout<<"\nNumber of days in this month are "<<days;
}

```

## **OUTPUT**

**Enter a month of year 2002 :3**

**Number of days in this month are 31**

**Explanation:** The above program is meant to calculate number of days in a month (given by user) of year 2002. We know that month numbers 1, 3, 5, 7, 8, 10 and 12 have 31 days each. So they can be put together in `switch( ) case`. But since there isn't such an option we have used the technique shown above. Similar is the case with month numbers 4, 6, 9 and 11 having 30 days each.

## **4.10 THE NESTED SWITCH( ) CASE STATEMENT**

C/C++ supports the nesting of `switch( ) case`. The inner `switch` can be part of an outer `switch`. The inner and the outer `switch case` constants may be the same. No conflict arises even if they are same. The example below demonstrates this concept.

### **4.12 Write a program to demonstrate nested switch( ) case statement.**

```

#include<iostream.h>
#include<constream.h>
void main( )
{
    int x;
    clrscr( );

    cout<<"\nEnter a number :";
    cin>>x;
switch(x)
{
    case 0:
    cout<<"\nThe number is 0 ";
    break;

    default:
    int y;
    y=x%2;
    switch(y)
    {

```

```

        case 0:
        cout<<"\nThe number is even ";
        break;
        case 1:
        cout<<"\nThe number is odd ";
        break;
    }
}

```

## OUTPUT

**Enter a number :5**

**The number is odd**

**Explanation:** The above program identifies whether input number was zero, even or odd. The first `switch( )` case finds out whether the number is zero or non-zero. If a non-zero value is entered, a default statement of first `switch case` statement is executed which executes another nested `switch( )` case. The nested `switch( )` case statement determines whether the number is even or odd and displays respective messages.

## 4.11 LOOPS IN C/C++

C/C++ provides loop structures for performing some tasks which are repetitive in nature. The C/C++ language supports three types of loop control structures. Their syntax is described in Table 4.1.

The `for` loop comprises of three actions. The three actions are placed in the `for` statement itself. The three actions initialize counter, test condition, and re-evaluation parameters are included in one statement. The expressions are separated by semi-colons (`:`). This helps the programmer to visualize the parameters easily. The `for` statement is equivalent to the `while` and `do-while` statements. The only difference between `for` and `while` is that the latter checks the logical condition and then executes the body of the loop, whereas the `for` statement test is always performed at the beginning of the loop. The body of the loop may not be executed at all times if the condition fails at the beginning. The `do-while` loop executes the body of the loop at least once regardless of the logical condition.

**Table 4.1 Loops in C++**

<b>for</b>	<b>while</b>	<b>do-while</b>
<pre> for (expression -1;      expression-2;      expression-3) statement ; </pre>	<pre> expression -1; while (expression -2) { statement; expression -3; } </pre>	<pre> expression -1; do { statement; expression-3; } while (expression-2); </pre>

## 4.12 THE FOR LOOP

The `for` loop allows execution of a set of instructions until a condition is met. Condition may be predefined or open-ended. Although all programming languages provide `for` loops,

still the power and flexibility provided by C/C++ is worth mentioning. The general syntax for the `for` loop is given below:

#### **Syntax of for loop**

```
for (initialization; condition; increment/decrement)
Statement block;
```

Though many variations of `for` loop are allowed, the simplest form is shown above.

The *initialization* is an assignment statement that is used to set the loop control variable(s). The condition is a relational expression that determines the number of the iterations desired or the condition to exit the loop. The *increment* or the re-evaluation parameter decides how to change the state of the variable(s) (quite often increase or decrease so as to approach the limit). These three sections must be separated by semi-colons. The body of the loop may consist of a block of statements (which have to be enclosed in braces) or a single statement (enclosure within braces is not compulsory but advised).

Following programs illustrate `for` loop:

4.13	4.14	4.15
<pre># include &lt;iostream.h&gt; # include &lt;constream.h&gt;  void main( ) { clrscr( ); int j;  for (j=1;j&lt;11;j++) cout&lt;&lt;" "&lt;&lt;j; }</pre>	<pre># include &lt;iostream.h&gt; # include &lt;constream.h&gt;  void main( ) { clrscr( ); int j=0;  for (;j&lt;11;j++) cout&lt;&lt;" "&lt;&lt;j; }</pre>	<pre># include &lt;iostream.h&gt; # include &lt;constream.h&gt;  void main( ) { clrscr( ); int j=0;  for (;j&lt;11;j++) cout&lt;&lt;" "&lt;&lt;j; }</pre>
<b>OUTPUT</b> <b>1 2 3 4 5 6 7 8 9 10</b>	<b>OUTPUT</b> <b>1 2 3 4 5 6 7 8 9 10</b>	<b>OUTPUT</b> <b>1 2 3 4 5 6 7 8 9 10</b>
Explanation: In this program initialization, condition, and increment is done in single parenthesis.	Explanation: In this program initialization is done before <code>for</code> statement. In <code>for</code> statement only condition and increment is done.	Explanation: In this program initialization is done before <code>for</code> statement. Increment is done inside the loop. The <code>for</code> loop contains only condition.

#### **4.16 Write a program to display all leap years from 1900 to 2002.**

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

void main( )
{
    int i=1900;
    clrscr( );
    cout<<"\nProgram to print all the leap years from 1900 to 2002
\n\n";
```

```

for(;i++<=2002;
{
    if(i%4==0&&i%100!=0)
        cout<<i<<" ";
    else if(i%100==0&&i%400==0)
        cout<<i<<" ";
}
}

```

**Explanation:** One must be aware that only those years that are  
 (a) divisible by 4 and not 100,  
 (b) divisible by 100 and also 400,  
 alone are qualified to be called as leap years. The above program checks for these conditions for all years from 1900 to 2002 and then prints the leap years. Stress is on `for` loop construct and use of `if` and `else-if` in the body.

#### 4.13 NESTED FOR LOOPS

We can also nest `for` loops to gain more advantage in some situations. The nesting level is not restricted at all. In the body of a `for` loop, any number of sub `for` loop(s) may exist.

#### 4.17 Write a program to demonstrate nested for loops.

```

# include <iostream.h>
# include <constream.h>>
void main( )
{
int a,b,c;
clrscr( );
for (a=1;a<=2;a++) /* outer loop */
{
for (b=1;b<=2;b++) /* middle loop */
{
    for (c=1;c<=2;c++) /* inner loop */
        cout <<"\n a=" <<a <<" + b=" <<b <<" + c=" <<c <<" :" <<a+b+c;
        cout <<"\n Inner Loop Over.";
}

cout <<"\n Middle Loop Over.";
}
cout <<"\n Outer Loop Over.";
}

```

**Explanation:** The above program is executed in the sequence shown below. The total number of iterations are equal to  $2 \times 2 \times 2 = 8$ . The final output provides 8 results. Table 4.2 shows the working of this program.

#### 4.14 THE WHILE LOOP

Another kind of loop structure in C/C++ is the `while` loop. It's format is given below.

##### Syntax:

```

while (test condition)
{
    body of the loop
}

```

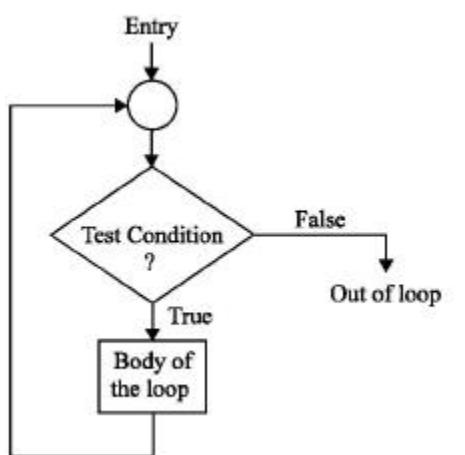
The test condition may be any expression. The loop statements will be executed till the condition is true i.e., the test condition is evaluated and if the condition is true, then the body of the loop is executed. When the condition becomes false the execution will be out of the loop.

The execution of the loop is shown in [Figure 4.4](#).

The block of the loop may contain single statement or a number of statements. The same block can be repeated. The braces are needed only if the body of the loop contains more than one statement. However, it is a good practice to use braces even if the body of the loop contains only one statement.

**Table 4.2** Working of program 4.17

Values of loop variables			Output
Outer	Middle	Inner	
(1) a=1	b=1	c=1	a+b+c=3
2) a=1	b=1	c=2	a+b+c=4
<b>Inner Loop Over</b>			
(3) a=1	b=2	c=1	a+b+c=4
(4) a=1	b=2	c=2	a+b+c=5
<b>Inner Loop Over</b>			
<b>Middle Loop Over</b>			
(5) a=2	b=1	c=1	a+b+c=4
(6) a=2	b=1	c=2	a+b+c=5
<b>Inner Loop Over</b>			
(7) a=2	b=2	c=1	a+b+c=5
(8) a=2	b=2	c=2	a+b+c=6
<b>Inner Loop Over</b>			
<b>Middle Loop Over</b>			
<b>Outer Loop Over.</b>			



**Steps of while loops are as follows:**

- (1) The test condition is evaluated and if it is true, the body of the loop is executed.
- (2) On execution of the body, test condition is repetitively checked and if it is true the body is executed.
- (3) The process of execution of the body will be continued till the test condition becomes false.
- (4) The control is transferred out of the loop.

**Fig. 4.4** The while loop

**4.18 Write a program to add 10 consecutive numbers starting from 1. Use the while loop.**

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

void main( )
{
    int a=1,sum=0;
    clrscr( );

    while(a<=10)
    {
        cout <<" " <<a;
        sum=sum+a;
        a++;
    }
    cout <<"\n Sum of 10 numbers : "<<sum;
}
```

**OUTPUT :**

**1 2 3 4 5 6 7 8 9 10**

**Sum of 10 numbers : 55**

**Explanation:** In the above program, integer variable `a` is initialized to 1 and variable `sum` to 0. The `while` loop checks the condition for `a<=10`. The variable `a` is added to variable `sum` and each time `a` is incremented by 1. In each `while` loop `a` is incremented and added to `sum`. When the value of `a` reaches to 10, the condition given in `while` loop becomes false. At last the loop is terminated. The sum of the number is displayed.

**4.19 Write a program to calculate the sum of individual digits of an entered number.**

```
# include <iostream.h>
# include <constream.h>
void main( )
{
    int num,t;
    clrscr( );
    cout<<"\n enter a number : ";
    cin>>num;
    t=num;
    int sum=0;
    while(num)           //condition is true until num!=0
    {
        sum=sum+num%10;
        num=num/10;
    }
    cout<<"\n Sum of the individual digits of the number "<<t<<" is =
"<<sum;
}
```

**OUTPUT**

**Enter a number : 234**

**Sum of the individual digits of the number 234 is = 9**

**Explanation:** The logic behind the program is to extract each time the LSD(lower significant digit) and divide the number by 10 so as to shift it by one place. To extract the LSD we use the fact that num % 10 = the remainder on dividing num by 10.

#### 4.20 Write a program to check whether the entered number is palindrome or not.

```
# include <iostream.h>
# include <constream.h>
void main( )
{
int num;
clrscr( );
cout<<"\n Enter the number : ";
cin>>num;
int b=0,a=num;
while(a)
{
b=b*10+a%10;
a=a/10;
}
if(num==b)
cout<<"\n The given number is palindrome ";
else
cout<<"\n The given number is not palindrome ";
}
```

#### OUTPUT

Enter the number: 121

The given number is palindrome

**Explanation:** Palindrome is a number that is equal to its reverse, for example, 51715. The program above first simply reverses the number and stores it in b. Then it compares b with original number to tell you whether or not the number is a palindrome. See that in each iteration the LSD of a is being made MSD of b. a is being shifted left and b is shifted right by one digit each in every iteration. Iterations are done until a exhausts or becomes equal to 0.

#### 4.15 THE DO-WHILE LOOP

The format of do-while loop in C/C++ is given below.

```
do
{
statement/s;
}
while (condition);
```

while (condition); The difference between the while and do-while loop is the place where the condition is to be tested. In the while loop, the condition is tested following the while statement and then the body gets executed. Whereas in do-while, the condition is checked at the end of the loop. The do-while loop will execute at least once even if the condition is false initially. The do-while loop executes until the condition becomes false.

#### 4.21 Write a program using do-while loop to print numbers and their cubes up to 10.

```

# include <iostream.h>
# include <constream.h>
# include <math.h>

void main( )
{
int y,x=1;
clrscr( );

cout <<"\n\tNumbers and their cubes \n";

do
{   y=pow(x,3);
   cout <<"\t"<<x<<"\t\t"<<y<<"\n";
   x++;
}  while (x<=10);
}

```

## **OUTPUT**

### **Numbers and their cubes**

<b>1</b>	<b>1</b>
----------	----------

<b>2</b>	<b>8</b>
----------	----------

<b>3</b>	<b>27</b>
----------	-----------

<b>4</b>	<b>64</b>
----------	-----------

<b>5</b>	<b>125</b>
----------	------------

<b>6</b>	<b>216</b>
----------	------------

<b>7</b>	<b>343</b>
----------	------------

<b>8</b>	<b>512</b>
----------	------------

<b>9</b>	<b>729</b>
----------	------------

<b>10</b>	<b>1000</b>
-----------	-------------

**Explanation:** Here, the mathematical function `pow (x, 3)` is used. Its meaning is to calculate the third power of x. With this function we get the value of  $y=x^3$ . For use of the `pow( )` function we have to include `math.h` header file.

### SUMMARY

- (1) This chapter teaches the basics of control structures of C++ language.
- (2) The C++ control structure covers two sets of statements. The set that performs certain operations repetitively is called as loop statements and the set that makes decision is known as decision-making statements.
- (3) The simple `if` statement executes statement only if the condition is true, otherwise, it follows the next statement. The `else` keyword is used when the expression is not true. The `else` keyword is optional.
- (4) C/C++ has four statements that perform an unconditional control transfer. These are `return( )`, `goto`, `break`, and `continue`. Of these, `return( )` is used only in functions. `goto` and `return( )` may be used anywhere in the program but `continue` and `break` statements may be used only in conjunction with a loop statement. `break` is used most frequently in `switch case`.
- (5) The `for` loop comprises of three actions. The three actions are placed in the `for` statement itself. The three actions ***initialize counter, test condition, and re-evaluation parameters*** are included in one statement. The `for` statement is equivalent to the `while` and `do-while` statement. The only difference between `for` and `while` is that the latter checks the logical condition and then executes the body of the loop, whereas the `for` statement test is always performed at the beginning of the loop. The body of the loop may not be executed at all times if the condition.

### EXERCISES

**[A] Answer the following questions.**

- (1) Explain the need of control structures in C++.
- (2) What are the differences between `break` and `continue` statements?
- (3) Why `goto` statement is not commonly used?
- (4) Explain the working of `if-else` statement.
- (5) Explain the working of `switch( ) case` statement.
- (6) Explain the role of `break` statement in `switch( ) case`.
- (7) What are the differences between `while` and `do-while` loop statements?
- (8) What is an infinite loop?
- (9) Explain nested `if's`.
- (10) Explain nested `switch( ) case` statement.
- (11) Explain the use of the keyword `default`.

**[B] Answer the following by selecting the appropriate option.**

- (1) Which of the following loop statement uses two keywords?
  - (a) `do-while` loop
  - (b) `for` loop
  - (c) `while` loop
  - (d) none of the above
- (2) The statement which requires at least one statement followed by it is

- (a) default
- (b) continue
- (c) break
- (d) else

(3) The loop statement terminated by a semi-colon is

- (a) do-while loop
- (b) for loop
- (c) while loop
- (d) none of the above

(4) Every expression always returns

- (a) 0 or 1
- (b) 1 or 2
- (c) -1 or 0
- (d) none of the above

(5) The meaning of if (1) is

- (a) always true
- (b) always false
- (c) both (a) and (b)
- (d) none of the above

(6) The curly braces are not present; the scope of loop statement is

- (a) one statement
- (b) two statements
- (c) four statements
- (d) none of the above

(7) In nested loop

- (a) the inner most loop is completed first
- (b) the outer most loop is completed first
- (c) both (a) and (b)
- (d) none of the above

### **[C] Attempt the following programs.**

- (1) Write a program to display numbers from 10 to 1 using for loop.
- (2) Write a program to calculate the factorial of a given number.
- (3) Write a program to display only even numbers in between 1 to 150.
- (4) Write a program to solve the series  $x=1/2!+1/4!+1/n!$ .
- (5) Write a program to calculate the sum of numbers between 1 to N numbers. The user enters the value of N.
- (6) Write a program to use break and continue statements.
- (7) Write a program to display alphabets A to Z using while loop.
- (8) Write a program to use break statement and terminate the loop.
- (9) Write a program to demonstrate the use of continue statement.

# 4

## CHAPTER

### Control Structures

C  
H  
A  
P  
T  
E  
R  
  
O  
U  
T  
L  
I  
N  
E

- [4.1 Introduction](#)
- [4.2 Decision-Making Statements](#)
- [4.3 The if-else Statement](#)
- [4.4 The Nested if-else Statement](#)
- [4.5 The jump Statement](#)
- [4.6 The goto Statement](#)
- [4.7 The break Statement](#)
- [4.8 The continue Statement](#)
- [4.9 The switch case Statement](#)
- [4.10 The Nested switch\( \) case Statement](#)
- [4.11 Loops in C/C++](#)

- [4.12 The for Loop](#)
- [4.13 Nested for Loops](#)
- [4.14 The while Loop](#)
- [4.15 The do-while Loop](#)

## 4.1 INTRODUCTION

This chapter deals with the basics of control structures of C++ language. Those who are already familiar with C, may skip this chapter. Those who are new to C++, should read this chapter thoroughly. As discussed earlier, C++ is a superset of C. The control structures of C and C++ are same. In this chapter, the concepts of structures are illustrated in detail for the new users.

A Program is nothing but a set of statements written in sequential order, one after the other. These statements are executed one after the other. Sometimes it may happen that the programmer requires to alter the flow of execution, or to perform the same operation for fixed iterations or whenever the condition does not satisfy. In such a situation the programmer uses the control structure. There are various control structures supported by C++. Programs which use such (one or three) control structures are said to be structured programs.

The C++ control structure covers two sets of statements. The set that performs certain operations repetitively is called as loop statements and the set that makes decision is known as decision-making statements.

## 4.2 DECISION-MAKING STATEMENTS

Following are decision-making statements.

- (1) The `if` statement
- (2) The `switch ( ) case` statement.

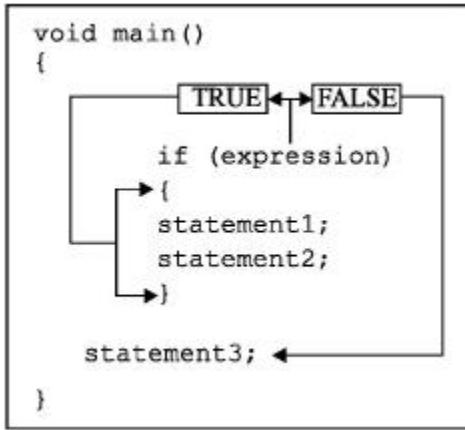
**Simple if statement** The syntax of simple `if` statement is described below

**Syntax** for the simplest `if` statement:

```
if (expression) /* no semi-colon */
    Statement;
```

The `if` statement contains an expression. The expression is evaluated. If the expression is true it returns 1 otherwise 0. The value 1 or any non-zero value is considered as true and 0 as false. In C++, the values (1) true and (0) false are known as `bool` type data.

The `bool` type data occupies one byte in memory. If the given expression in the `if ( )` statement is true, the following statement or block of statements too are executed, otherwise the statement that appears immediately after `if` block (true block) is executed as given in [Figure 4.1](#).



**Fig. 4.1** The simple if statement

As shown in [Figure 4.1](#), the expression is always evaluated to true or false. When the expression is true, the statement in if block is executed. When the expression is false the if block is skipped and the statement (state-ment3) after if block is executed.

The expression given in the if statement may not be always true or false. For example, `if(1)` or `if(0)`. When such statement is encountered, the compiler will display a warning message "condition is always true" or "condition is always false". In place of expression we can also use function that returns 0 or 1 return values. The following programs illustrate the points discussed above.

#### 4.1 Write a program to enter age and display message whether the user is eligible for voting or not.

```

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

void main( )
{
    clrscr( );
    int age;
    cout<<"Enter Your Age :";
    cin>>age;
    if (age>=18)
    {
        cout <<" You are eligible for voting.";
    }
}

```

#### OUTPUT

Enter Your Age : 23

You are eligible for voting.

**Explanation:** In the above program, integer variable `age` is declared. The user enters his/her age. The value is tested with if statement. If the age is greater than or equal to 18 the statement followed by if statement is executed. The message displayed will be "You are eligible for voting." If the condition is false nothing is displayed.

#### 4.2 Write a program to use library function with if statement instead of expression.

```

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

void main( )
{
    static char nm[]="Hello";
    clrscr( );

    if (strlen(nm))
    { cout <<" The string is not empty."; }

}

```

## OUTPUT

**The string is not empty.**

**Explanation:** In the above program, the character array nm[ ] is initialized with the string "Hello". The strlen( ) function is used in the if statement. The strlen( ) function calculates the length of the string and returns it. If the string is empty it returns (0) false otherwise non-zero value (string length). The non-zero value is considered as true. The strlen( ) function returns non-zero value (true). The if statement displays the message "The string is not empty." Here, instead of expression, a library function is used.

## 4.3 THE IF-ELSE STATEMENT

The simple if statement executes statement only if the condition is true otherwise it follows the next statement. The else keyword is used when the expression is not true. The else keyword is optional. The syntax of if..else statement is given below and Figure 4.2 describes the working of if..else statement.

The format of if..else statement is as follows:

```

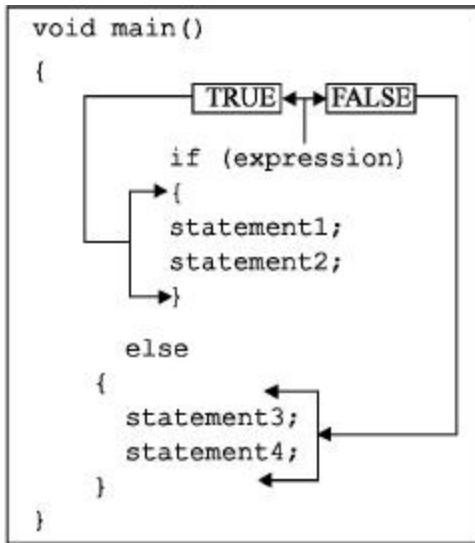
if(the condition is true)
execute the Statement1;
else
execute the Statement2;
OR
Syntax of if - else statement can be given as follows:
if ( expression is true)

{
statement 1;      →      if block
statement 2;
}

else

{
statement 3;      →      else block
statement 4;
}

```



**Fig.4.2** The if-else statement

As shown in [Figure 4.2](#), both if and else block contain statements. When the expression is true if block is executed otherwise else block is executed.

#### 4.3 Write a program to enter age and display message whether the user is eligible for voting or not. Use if-else statement.

```

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

void main( )
{
clrscr( );

int age;
cout<<"Enter Your Age : ";
cin>>age;

if (age>=18)
{
    cout <<" You are eligible for voting." ;
}
else
{
    cout <<" You are noneligible for voting" <<endl;
    cout << " Wait for "<<18-age<<" year(s).";
}
}

```

#### OUTPUT:

Enter Your Age : 17  
**You are noneligible for voting**  
**Wait for 1 year(s).**

**Explanation:** In the above program, the user enters his/her age. The integer variable age is used to store the value. The if statement checks the value of age. If the age is greater than or equal to 18, the if block of statement is executed, otherwise the else block of statement is executed. Thus, the else statement extends if statement.

#### 4.4 Write a program with simple if statement. If entered score is less than 50 display one message, otherwise another message.

```
#include<iostream.h>
#include<conio.h>
int main( )
{
int v;
clrscr( );
cout<<"Enter a number : ";
cin>>v;
if(v>=50)
cout<<"\nCongrats !! You scored a half / more than half century ";
else
if(v<50)
cout<<"\nCome on !! You can score a half century ";
return 0;
}
```

#### OUTPUT:

```
Enter a number : 49
Come on !! You too can score a half century
```

```
Enter a number : 56
Congrats !! You scored a half / more than half century
```

**Explanation:** The program first prompts the user to enter a number 'v'. If it is greater than 50 or equal to 50, it prompts a particular message, otherwise for less than 50, other message is displayed. If found true, a particular message is prompted, otherwise another.

#### 4.4 THE NESTED IF-ELSE STATEMENT

In this kind of statements, number of logical conditions are checked for executing various statements. Here, if any logical condition is true, the compiler executes the block followed by if condition, otherwise it skips and executes else block.

In if..else statement, else block is executed by default after failure of if condition. In order to execute the else block depending upon certain conditions we can add repetitively if statements in else block. This kind of nesting will be unlimited.

**Syntax** of if-else...if statement can be given as follows:

```
if ( condition)
{
    statement 1;           -> if  block
    statement 2;
}

else    if (condition)
{
    statement 3;           -> else block
    statement4;
}
```

```

else
{
statement5;
statement6;
}

```

From the above block, following rules can be described for applying nested if..else..if statements:

1. Nested if..else can be chained with one another.
2. If the condition is false, control passes to else block where condition is again checked with the if statement. This process continues till there is no if statement in the last else block.
3. If one of the if statement satisfies the condition, other nested if..else will not be executed.

Following programs illustrates the working of nested if-else statements.

#### **4.5 Write a program to enter two characters and display the larger character. Use nested if-else statements.**

```

#include <iostream.h>
#include <constream.h>

void main( )
{
    char j,k;
    clrscr( );
    cout<<"\n Enter a character :";
    j=getche( );
    cout <<"\n Enter another character : ";
    k=getche( );
    cout<<endl;

    if (j>k)
    {
        cout <<j <<" is larger than "<<k;
    }
    else if(k>j)
    {
        cout <<k <<" is larger than "<<j;
    }
    else
    {
        cout <<"Both the characters are same";
    }
}

```

#### **OUTPUT**

```

Enter a character : S
Enter another character : A
S is larger than A

```

**Explanation:** In the above program, two characters are entered and stored in the character variable j and k. The first if statement checks whether j is greater than k. Here,

comparison is done considering ASCII values. If the condition is true, a message will be displayed and program terminates. In case the condition is false the `else` block is executed. In the `else` block another `if` statement is present. The `if` statement inside the `else` block checks whether `k` is greater than `j` or not. If the condition is true `if` block is executed, otherwise `else` block is executed.

#### 4.6 Write a program to explain the concept of nested `if-else` statements.

```
#include<conio.h>
#include<iostream.h>
int main( )
{
int score;
clrscr( );
cout<<"\nEnter Sachin's score :";
cin>>score;
if(score>=50)
{
    if(score>=100)
        cout<<"Sachin scored a century and more runs";
    else
    {
        cout<<"\nSachin scored more than half century ";
        cout<<"\nCross your fingers and pray he completes century ";
    }
}
else
{
    if(score==0)
        cout<<"Oh my God ";
    if(score>0)
        cout<<"Not in form today ";
}
return 0;
}
```

#### OUTPUT:

Enter Sachin's score : 0

Oh my God !

Enter Sachin's score : 45

Not in form today

Enter Sachin's score : 60

Sachin scored more than half century

Cross your fingers and pray he completes century

Enter Sachin's score : 116

Sachin scored a century and more runs

**Explanation:** From above it can be seen that if score was greater than 50 and greater than 100 then a particular message is prompted. If score was greater than 50 but less than 100, then another set of messages is prompted. If, however, the score was less than 50 and equal to 0 then a particular message is prompted. If score was less than 50 but not 0 then another message is displayed.

#### 4.7 Write a program to execute `if` statement without any expression.

```

# include <iostream.h>
# include <constream.h>

void main( )
{
    int j;
    clrscr( );
    cout <<"\n Enter a value : ";
    cin>>j;

    if(j)
        cout<<"Wel Come";
    else
        cout<<"Good Bye";
}

```

## OUTPUT

Enter a value : 0

Good Bye

**Explanation:** In the above program, an integer variable *j* is declared. The user is asked to enter an integer value. The entered value is stored in the variable *j*. The *if* statement checks the value of *j*. As explained at the beginning of this chapter, 0 is considered as false and any non-zero value is true. Thus, when user enters 0, the *else* block is executed and when a non-zero value is entered, *if* block is executed.

## THE IF-ELSE-IF LADDER STATEMENT

A common programming construct is the if-else-if ladder, sometimes called the if-else-if staircase because of it's appearance. It's general form is

```

if ( expression) statement block1;
    else statement block2;
        if(expression) statement block3;
            else statement block4;
                if(expression) statement block5;
                .
                .
                .
                .
                .
                .
                else statement blockn;

```

The conditions are evaluated from the top. As soon as a true condition is met, the associated statement block gets executed and rest of the ladder is bypassed. If none of the conditions are met then the final *else* block gets executed .If this 'else' is not present and none of the 'if' evaluates to true then entire ladder is bypassed.

Although the indentation of the preceding 'if-else-if' ladder is technically correct, it can lead to overly deep indentation. Imagine 256 (maximum allowed) such stairs and each indented: Enough to confuse!

This reason made it vital to use the form as shown below:

```

if ( expression)
    statement block1;      → if block

```

```

else    if (expression)
    statement block2;      → else block
else    if (expression)
    statement block3;      → else block
.
.
.
else
    statement block5;

```

The following program will clear your understanding.

#### **4.8 Write a program to simulate tariff charges for reaching different destinations by bus.**

```

#include<conio.h>
#include<iostream.h>
int main( )
{
int cost,ch;
clrscr( );
cout<<"\n.....NANDED BUS STATION.....\n";
cout<<".....Menu.....";
cout<<"\nBombay..1";
cout<<"\nNagpur..2";
cout<<"\nPune..3";
cout<<"\nAmravati..4";
cout<<"\nAurangabad..5";
cout<<"\n\nEnter your destination :";
cin>>ch;
if(ch==1)
    cost=100;
else if(ch==2)
    cost=70;
else if(ch==3)
    cost=50;
else if(ch==4)
    cost=60;
else if(ch==3)
    cost=40;
else
    cost=0;
if(cost!=0)
{
    cout<<"\n\nThe ticket cost is : Rs "<<cost;
    cout<<"\nPay the amount to get booked";
}
else
    cout<<"Sorry there's no bus to desired destination";
cin.get( );
return 0;
}

```

#### **OUTPUT:**

.....NANDED BUS STATION.....

.....Menu.....

Bombay..1

Nagpur..2

Pune..3

Amravati..4

**Aurangabad..5**

**Enter your destination : 4**

**The ticket cost is : Rs 60**

**Pay the amount to get booked**

**Explanation:** The program above simulates a bus station. We use an 'if-else-if' ladder to find out the cost of ticket to a particular station, if there was a bus to that station. There wasn't a bus to that station having cost=0.Finally if cost is non-zero it's printed. The user is prompted to enter the choice. Depending upon the choice, the fare of destination station is displayed. In the above example, choice 4 is given. The result displayed is

"The ticket cost is : Rs 60

Pay the amount to get booked".

## **4.5 THE JUMP STATEMENT**

C/C++ has four statements that perform an unconditional control transfer. These are `return( )`, `goto`, `break` and `continue`. Of these, `return( )` is used only in functions. The `goto` and `return( )` may be used anywhere in the program but `continue` and `break` statements may be used only in conjunction with a loop statement. In 'switch case' 'break' is used most frequently.

## **4.6 THE GOTO STATEMENT**

This statement does not require any condition. This statement passes control anywhere in the program without least care for any condition. The general format for this statement is shown below:

```
goto label;
```

```
—
```

```
—
```

```
label:
```

where, `label` is any valid `label` either before or after `goto`.The label must start with any character and can be constructed with rules used for forming identifiers. Avoid using `goto` statement.

## **4.9 Write a program to demonstrate the use of goto statement.**

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

void main( )
{
    int x;
    clrscr( );
    cout <<"Enter a Number :";
    cin>>x;
    if (x%2==0)
        goto even;
    else
        goto odd;
```

```
even :cout<<x<<" is Even Number.";
return;
odd: cout<<x<<" is Odd Number.";
}
```

## OUTPUT

Enter a Number: 5

5 is Odd Number.

**Explanation:** In the above program, a number is entered. The number is checked for even or odd with modulus division operator. When the number is even, the `goto` statement transfers the control to the label `even`. Similarly, when the number is odd, the `goto` statement transfers the control to the label `odd` and respective messages will be displayed.

## 4.7 THE BREAK STATEMENT

The `break` statement allows the programmer to terminate the loop. The `break` skips from the loop or the block in which it is defined. The control then automatically passes on to the first statement after the loop or the block. The `break` statement can be associated with all the conditional statements (especially `switch( )` case). We can also use `break` statements in the nested loops. If we use `break` statement in the innermost loop, then the control of program is terminated from that loop only and resumes at the next statement following that loop. The widest use of this statement is in `switch` case where it is used to avoid flow of control from one case to other.

## 4.8 THE CONTINUE STATEMENT

The `continue` statement works quite similar to the `break` statement. Instead of forcing the control to end of loop (as it is in case of `break`), `continue` causes the control to pass on to the beginning of the block/loop. In case of `for` loop, the `continue` case causes the condition testing and incrementation steps to be executed (while rest of the statements following `continue` are neglected). For `while` and `do-while`, `continue` causes control to pass on to conditional tests. It is useful in programming situation when you want particular iterations to occur only up to some extent or you want to neglect some part of your code. The programs on `break` and `continue` can be performed by the programmer.

## 4.9 THE SWITCH CASE STATEMENT

The `switch` statement is a multi-way branch statement and an alternative to `if-else-if` ladder in many situations. This statement requires only one argument, which is then checked with number of case options. The `switch` statement evaluates the expression and then looks for its value among the case constants. If the value is matched with a case constant then that case constant is executed until a `break` statement is found or end of `switch` block is reached. If not then simply `default` (if present) is executed (if `default` isn't present then simply control flows out of the `switch` block).

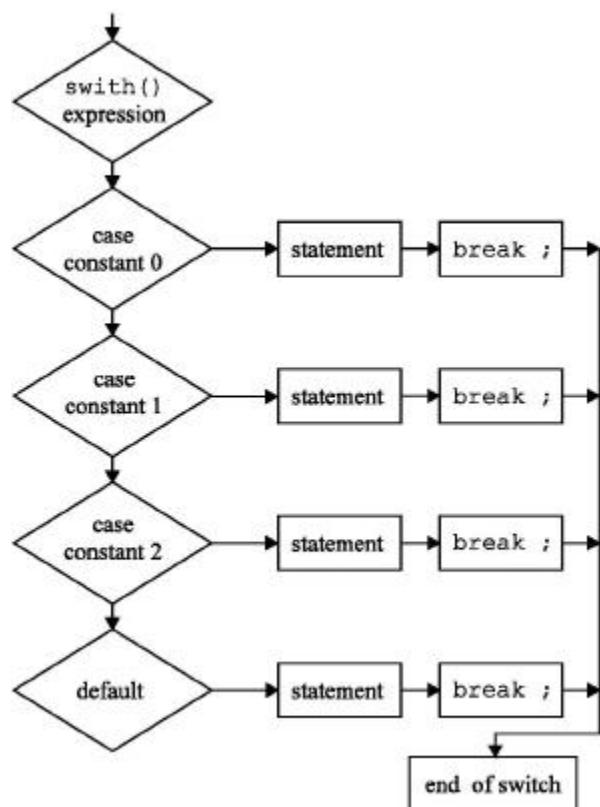
The `default` is normally present at the bottom of `switch` case structure. But we can also define `default` statement anywhere in the `switch` structure. The `default` block must not be empty. Every case statement terminates with : (colon). The `break` statement is used to

stop the execution of succeeding cases and pass the control to the switch end of block. The syntax of the `switch( )` case statement is shown below. [Figure 4.3](#) simulates the working of `switch( ) case statement`.

```
switch(variable or expression)
{
    case constant A :
        statement;
        break;

    case constant B :
        statement;
        break;

    default :
        statement ;
}
```



**Fig. 4.3** The switch case statement

**4.10 Write a program to display different lines according to users choice.  
Use `switch( ) case statement`.**

```
#include<stdlib.h>
#include<constream.h>
#include<iostream.h>
void main( )
{
int c;
```

```

clrscr( );

cout<<"\n LINE FORMAT MENU ";
cout<<"\n1] *****";
cout<<"\n2] ======";
cout<<"\n3] ~~~~~~~~";
cout<<"\n4] _____";
cout<<"\nEnter your choice :";

cin>>c;

switch(c)
{
    case 1:
        cout <<"\n*****";
        break;

    case 2:
        cout <<"\n=====";
        break;

    case 3:
        cout <<"\n~~~~~~~";
        break;

    case 4:
        cout <<"\n_____";
        break;

    default:
        cout <<"\n.....";
    }
}

```

## **OUTPUT**

### **LINE FORMAT MENU**

```

1] *****
2] =====
3] ~~~~~~~~
4] __

```

**Enter your choice :2**

```

=====

```

**Explanation:** In the above program, a menu is displayed on the screen with different type of lines. The user enters a number as given in the menu. The `switch( )` statement checks the value of variable `c`. All case statements are tested and one that satisfies, is executed. If the user enters a value other than that listed in the menu, `default` statement is executed.

**4.11 Write a program to enter a month number of year 2002 and display the number of days present in that month.**

```

#include <iostream.h>
#include<constream.h>
void main( )
{
clrscr( );

int month,days;
cout<<"Enter a month of year 2002 :";
cin>>month;

switch(month)
{
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        days=31;
        break;
    case 2:
        days=28;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        days=30;
        break;
}
cout<<"\nNumber of days in this month are "<<days;
}

```

## **OUTPUT**

**Enter a month of year 2002 :3**

**Number of days in this month are 31**

**Explanation:** The above program is meant to calculate number of days in a month (given by user) of year 2002. We know that month numbers 1, 3, 5, 7, 8, 10 and 12 have 31 days each. So they can be put together in `switch( ) case`. But since there isn't such an option we have used the technique shown above. Similar is the case with month numbers 4, 6, 9 and 11 having 30 days each.

## **4.10 THE NESTED SWITCH( ) CASE STATEMENT**

C/C++ supports the nesting of `switch( ) case`. The inner `switch` can be part of an outer `switch`. The inner and the outer `switch case constants` may be the same. No conflict arises even if they are same. The example below demonstrates this concept.

## **4.12 Write a program to demonstrate nested switch( ) case statement.**

```

#include<iostream.h>
#include<constream.h>
void main( )
{

```

```

int x;
clrscr( );

cout<<"\nEnter a number :";
cin>>x;
switch(x)
{
    case 0:
        cout<<"\nThe number is 0 ";
        break;

    default:
        int y;
        y=x%2;
        switch(y)
        {
            case 0:
                cout<<"\nThe number is even ";
                break;
            case 1:
                cout<<"\nThe number is odd ";
                break;
        }
}
}

```

## **OUTPUT**

**Enter a number :5**

**The number is odd**

**Explanation:** The above program identifies whether input number was zero, even or odd. The first `switch( )` case finds out whether the number is zero or non-zero. If a non-zero value is entered, a default statement of first `switch` case statement is executed which executes another nested `switch( )` case. The nested `switch( )` case statement determines whether the number is even or odd and displays respective messages.

## **4.11 LOOPS IN C/C++**

C/C++ provides loop structures for performing some tasks which are repetitive in nature. The C/C++ language supports three types of loop control structures. Their syntax is described in Table 4.1.

The `for` loop comprises of three actions. The three actions are placed in the `for` statement itself. The three actions initialize counter, test condition, and re-evaluation parameters are included in one statement. The expressions are separated by semi-colons (`;`). This helps the programmer to visualize the parameters easily. The `for` statement is equivalent to the `while` and `do-while` statements. The only difference between `for` and `while` is that the latter checks the logical condition and then executes the body of the loop, whereas the `for` statement test is always performed at the beginning of the loop. The body of the loop may not be executed at all times if the condition fails at the beginning. The `do-while` loop executes the body of the loop at least once regardless of the logical condition.

**Table 4.1 Loops in C++**

<b>for</b>	<b>while</b>	<b>do-while</b>
<pre>for (expression -1;       expression-2;       expression-3) statement ;</pre>	<pre>expression -1; while (expression -2) { statement; expression -3; }</pre>	<pre>expression -1; do { statement; expression-3; } while (expression-2);</pre>

## 4.12 THE FOR LOOP

The **for** loop allows execution of a set of instructions until a condition is met. Condition may be predefined or open-ended. Although all programming languages provide **for** loops, still the power and flexibility provided by C/C++ is worth mentioning. The general syntax for the **for** loop is given below:

### Syntax of for loop

```
for (initialization; condition; increment/decrement)
Statement block;
```

Though many variations of **for** loop are allowed, the simplest form is shown above.

The *initialization* is an assignment statement that is used to set the loop control variable(s). The condition is a relational expression that determines the number of the iterations desired or the condition to exit the loop. The *increment* or the re-evaluation parameter decides how to change the state of the variable(s) (quite often increase or decrease so as to approach the limit). These three sections must be separated by semi-colons. The body of the loop may consist of a block of statements (which have to be enclosed in braces) or a single statement (enclosure within braces is not compulsory but advised).

Following programs illustrate **for** loop:

4.13	4.14	4.15
<pre># include &lt;iostream.h&gt; # include &lt;constream.h&gt;  void main( ) { clrscr( ); int j;  for (j=1;j&lt;11;j++) cout&lt;&lt;" "&lt;&lt;j; }</pre>	<pre># include &lt;iostream.h&gt; # include &lt;constream.h&gt;  void main( ) { clrscr( ); int j=0;  for (;j&lt;11;j++) cout&lt;&lt;" "&lt;&lt;j; }</pre>	<pre># include &lt;iostream.h&gt; # include &lt;constream.h&gt;  void main( ) { clrscr( ); int j=0;  for (;j&lt;11;) cout&lt;&lt;" "&lt;&lt;j; ++j; }</pre>
<b>OUTPUT</b> <b>1 2 3 4 5 6 7 8 9 10</b>	<b>OUTPUT</b> <b>1 2 3 4 5 6 7 8 9 10</b>	<b>OUTPUT</b> <b>1 2 3 4 5 6 7 8 9 10</b>
Explanation: In this program initialization,	Explanation: In this program initialization is done before <b>for</b> statement.	Explanation: In this program initialization is done before <b>for</b> statement. Increment is

condition, and increment is done in single parenthesis.	In <code>for</code> statement only condition and increment is done.	done inside the loop. The <code>for</code> loop contains only condition.
---	---	--

#### 4.16 Write a program to display all leap years from 1900 to 2002.

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

void main( )
{
    int i=1900;
    clrscr( );
    cout<<"\nProgram to print all the leap years from 1900 to 2002
\n\n";

for(;i++<=2002;)
{
    if(i%4==0&&i%100!=0)
        cout<<i<<" ";
    else if(i%100==0&&i%400==0)
        cout<<i<<" ";
}
}
```

**Explanation:** One must be aware that only those years that are

- (a) divisible by 4 and not 100,
- (b) divisible by 100 and also 400,

alone are qualified to be called as leap years. The above program checks for these conditions for all years from 1900 to 2002 and then prints the leap years. Stress is on `for` loop construct and use of `if` and `else-if` in the body.

#### 4.13 NESTED FOR LOOPS

We can also nest `for` loops to gain more advantage in some situations. The nesting level is not restricted at all. In the body of a `for` loop, any number of sub `for` loop(s) may exist.

#### 4.17 Write a program to demonstrate nested `for` loops.

```
# include <iostream.h>
# include <constream.h>>
void main( )
{
int a,b,c;
clrscr( );
for (a=1;a<=2;a++) /* outer loop */
{
for (b=1;b<=2;b++) /* middle loop */
{
    for (c=1;c<=2;c++) /* inner loop */
        cout <<"\n a=<<a <<" + b="<<b <<" + c="<<c <<" :"<<a+b+c;
        cout <<"\n Inner Loop Over.";
}

cout <<"\n Middle Loop Over.";
}
```

```

cout<<"\n Outer Loop Over.";
}

```

**Explanation:** The above program is executed in the sequence shown below. The total number of iterations are equal to  $2*2*2=8$ . The final output provides 8 results. Table 4.2 shows the working of this program.

#### 4.14 THE WHILE LOOP

Another kind of loop structure in C/C++ is the while loop. It's format is given below.

**Syntax:**

```

while (test condition)
{
    body of the loop
}

```

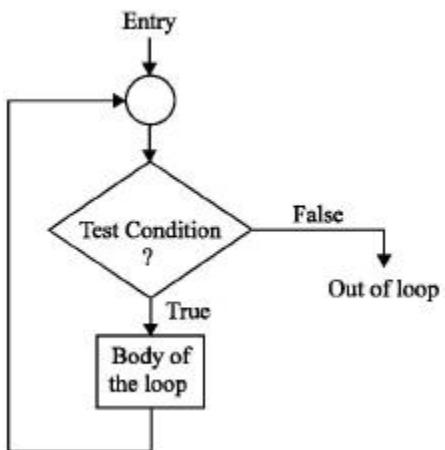
The test condition may be any expression. The loop statements will be executed till the condition is true i.e., the test condition is evaluated and if the condition is true, then the body of the loop is executed. When the condition becomes false the execution will be out of the loop.

The execution of the loop is shown in Figure 4.4.

The block of the loop may contain single statement or a number of statements. The same block can be repeated. The braces are needed only if the body of the loop contains more than one statement. However, it is a good practice to use braces even if the body of the loop contains only one statement.

**Table 4.2** Working of program 4.17

Values of loop variables			Output
Outer	Middle	Inner	
(1) a=1	b=1	c=1	a+b+c=3
2) a=1	b=1	c=2	a+b+c=4
<b>Inner Loop Over</b>			
(3) a=1	b=2	c=1	a+b+c=4
(4) a=1	b=2	c=2	a+b+c=5
<b>Inner Loop Over</b>			
<b>Middle Loop Over</b>			
(5) a=2	b=1	c=1	a+b+c=4
(6) a=2	b=1	c=2	a+b+c=5
<b>Inner Loop Over</b>			
(7) a=2	b=2	c=1	a+b+c=5
(8) a=2	b=2	c=2	a+b+c=6
<b>Inner Loop Over</b>			
<b>Middle Loop Over</b>			
<b>Outer Loop Over.</b>			



**Steps of while loops are as follows:**

- (1) The test condition is evaluated and if it is true, the body of the loop is executed.
- 2) On execution of the body, test condition is repetitively checked and if it is true the body is executed.
- 3) The process of execution of the body will be continued till the test condition becomes false.
- 4) The control is transferred out of the loop.

**Fig. 4.4** The while loop

**4.18 Write a program to add 10 consecutive numbers starting from 1. Use the while loop.**

```

#include <iostream.h>
#include <constream.h>

void main( )
{
    int a=1,sum=0;
    clrscr( );

    while(a<=10)
    {
        cout <<" "<<a;
        sum=sum+a;
        a++;
    }
    cout <<"\n Sum of 10 numbers : "<<sum;
}

```

**OUTPUT :**

**1 2 3 4 5 6 7 8 9 10**

**Sum of 10 numbers : 55**

**Explanation:** In the above program, integer variable a is initialized to 1 and variable sum to 0. The while loop checks the condition for  $a \leq 10$ . The variable a is added to variable sum and each time a is incremented by 1. In each while loop a is incremented and added to sum. When the value of a reaches to 10, the condition given in while loop becomes false. At last the loop is terminated. The sum of the number is displayed.

**4.19 Write a program to calculate the sum of individual digits of an entered number.**

```

#include <iostream.h>
#include <constream.h>
void main( )
{

```

```

int num,t;
clrscr( );
cout<<"\n enter a number : ";
cin>>num;
t=num;
int sum=0;
while(num)           //condition is true until num!=0
{
sum=sum+num%10;
num=num/10;
}
cout<<"\n Sum of the individual digits of the number "<<t<<" is =
"<<sum;
}

OUTPUT

```

**Enter a number : 234**

**Sum of the individual digits of the number 234 is = 9**

**Explanation:** The logic behind the program is to extract each time the LSD(lower significant digit) and divide the number by 10 so as to shift it by one place. To extract the LSD we use the fact that  $\text{num} \% 10$  = the remainder on dividing num by 10.

#### **4.20 Write a program to check whether the entered number is palindrome or not.**

```

# include <iostream.h>
# include <constream.h>
void main( )
{
int num;
clrscr( );
cout<<"\n Enter the number : ";
cin>>num;
int b=0,a=num;
while(a)
{
b=b*10+a%10;
a=a/10;
}
if(num==b)
cout<<"\n The given number is palindrome ";
else
cout<<"\n The given number is not palindrome ";
}

OUTPUT

```

**Enter the number: 121**

**The given number is palindrome**

**Explanation:** Palindrome is a number that is equal to it's reverse, for example, 51715. The program above first simply reverses the number and stores it in b. Then it compares b with original number to tell you whether or not the number is a palindrome. See that in each interaction the LSD of a is being made MSD of b. a is being shifted left and b is shifted right by one digit each in every iteration. Iterations are done until a exhausts or becomes equal to 0.

#### **4.15 THE DO-WHILE LOOP**

The format of do-while loop in C/C++ is given below.

```
do
{
statement/s;
}
while (condition);
```

while (condition); The difference between the while and do-while loop is the place where the condition is to be tested. In the while loop, the condition is tested following the while statement and then the body gets executed. Where as in do-while, the condition is checked at the end of the loop. The do-while loop will execute at least once even if the condition is false initially. The do-while loop executes until the condition becomes false.

#### **4.21 Write a program using do-while loop to print numbers and their cubes up to 10.**

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

void main( )
{
int y,x=1;
clrscr( );

cout <<"\n\tNumbers and their cubes \n";

do
{   y=pow(x,3);
    cout <<"\t" <<x <<"\t\t" <<y <<"\n";
    x++;
}   while (x<=10);
}
```

#### **OUTPUT**

#### **Numbers and their cubes**

**1**

**1**

**2**

**8**

**3**

**27**

**4**

**64**

**5**

**125**

**6**

**216**

**7**

**343**

**8**

**512**

**9**

**729**

**10**

**1000**

**Explanation:** Here, the mathematical function `pow (x, 3)` is used. Its meaning is to calculate the third power of x. With this function we get the value of  $y=x^3$ . For use of the `pow ( )` function we have to include `math.h` header file.

### SUMMARY

- (1) This chapter teaches the basics of control structures of C++ language.
- (2) The C++ control structure covers two sets of statements. The set that performs certain operations repetitively is called as loop statements and the set that makes decision is known as decision-making statements.
- (3) The simple `if` statement executes statement only if the condition is true, otherwise, it follows the next statement. The `else` keyword is used when the expression is not true. The `else` keyword is optional.
- (4) C/C++ has four statements that perform an unconditional control transfer. These are `return ( )`, `goto`, `break`, and `continue`. Of these, `return ( )` is used only in functions. `goto` and `return ( )` may be used anywhere in the program but `continue` and `break` statements may be used only in conjunction with a loop statement. `break` is used most frequently in `switch case`.
- (5) The `for` loop comprises of three actions. The three actions are placed in the `for` statement itself. The three actions ***initialize counter, test condition, and re-evaluation parameters*** are included in one statement. The `for` statement is equivalent to the `while` and `do-while` statement. The only difference between `for` and `while` is that the latter checks the logical condition and then executes the body of the loop, whereas the `for` statement test is always performed at the beginning of the loop. The body of the loop may not be executed at all times if the condition.

### EXERCISES

#### [A] Answer the following questions.

- (1) Explain the need of control structures in C++.
- (2) What are the differences between `break` and `continue` statements?
- (3) Why `goto` statement is not commonly used?
- (4) Explain the working of `if-else` statement.
- (5) Explain the working of `switch ( ) case` statement.

- (6) Explain the role of `break` statement in `switch( ) case`.
- (7) What are the differences between `while` and `do-while` loop statements?
- (8) What is an infinite loop?
- (9) Explain nested if's.
- (10) Explain nested `switch( ) case` statement.
- (11) Explain the use of the keyword `default`.

**[B] Answer the following by selecting the appropriate option.**

- (1) Which of the following loop statement uses two keywords?
  - (a) `do-while` loop
  - (b) `for` loop
  - (c) `while` loop
  - (d) none of the above
- (2) The statement which requires at least one statement followed by it is
  - (a) `default`
  - (b) `continue`
  - (c) `break`
  - (d) `else`
- (3) The loop statement terminated by a semi-colon is
  - (a) `do-while` loop
  - (b) `for` loop
  - (c) `while` loop
  - (d) none of the above
- (4) Every expression always returns
  - (a) 0 or 1
  - (b) 1 or 2
  - (c) -1 or 0
  - (d) none of the above
- (5) The meaning of `if(1)` is
  - (a) always true
  - (b) always false
  - (c) both (a) and (b)
  - (d) none of the above
- (6) The curly braces are not present; the scope of loop statement is
  - (a) one statement
  - (b) two statements
  - (c) four statements
  - (d) none of the above
- (7) In `nested` loop
  - (a) the inner most loop is completed first
  - (b) the outer most loop is completed first
  - (c) both (a) and (b)
  - (d) none of the above

**[C] Attempt the following programs.**

- (1) Write a program to display numbers from 10 to 1 using `for` loop.
- (2) Write a program to calculate the factorial of a given number.
- (3) Write a program to display only even numbers in between 1 to 150.
- (4) Write a program to solve the series  $x=1/2!+1/4!+1/n!$ .
- (5) Write a program to calculate the sum of numbers between 1 to N numbers. The user enters the value of N.
- (6) Write a program to use `break` and `continue` statements.
- (7) Write a program to display alphabets A to Z using `while` loop.
- (8) Write a program to use `break` statement and terminate the loop.
- (9) Write a program to demonstrate the use of `continue` statement.

# 5

CHAPTER

## Functions in C++

C  
H  
A  
P  
T  
E

- [5.1 Introduction](#)
- [5.2 The `main \( \)` Function in C and C++](#)
- [5.3 Parts of Function](#)

- [5.4 Passing Arguments](#)
- [5.5 LValues and RValues](#)
- [5.6 Return by Reference](#)
- [5.7 Returning More Values by Reference](#)
- [5.8 Default Arguments](#)
- [5.9 The const Argument](#)
- [5.10 Inputting Default Arguments](#)
- [5.11 Inline Functions](#)
- [5.12 Function Overloading](#)
- [5.13 Principles of Function Overloading](#)
- [5.14 Precautions with Function Overloading](#)
- [5.15 Library Functions](#)

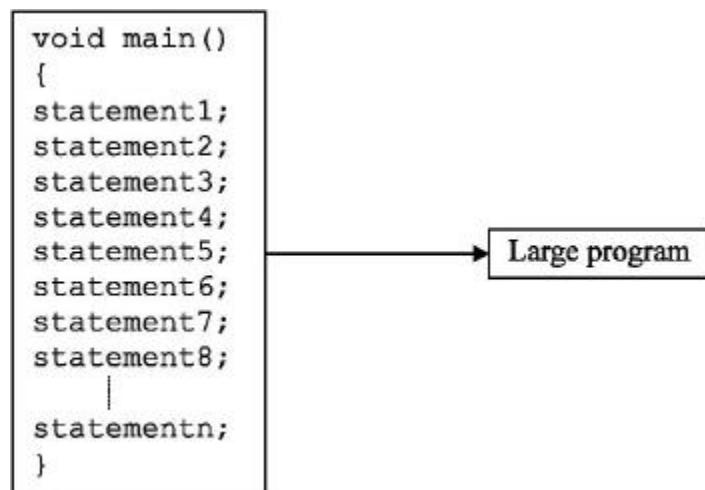
## 5.1 INTRODUCTION

When the concept of function or sub-program was not introduced, the programs were large in size and code got repeated as shown in [Figure 5.1](#). It was very difficult to debug and update these large programs because many bugs were encountered. When the functions and sub-programs are introduced, the programs are divided into a number of segments and code duplication is avoided as shown in [Figure 5.2](#). Bugs in small programs can be searched easily and this step leads to the development of complex programs with functions.

One of the features of C/C++ language is that a large sized program can be divided into smaller ones. The smaller programs can be written in the form of functions. The process of dividing a large program into tiny and handy sub-programs and manipulating them independently is known as ***modular programming***. This technique is similar to divide-

and-conquer technique. Dividing a large program into smaller functions provides advantages to the programmer. The testing and debugging of a program with functions becomes easier. A function can be called repeatedly based on the application and thus the size of the program can be reduced. The message passing between a caller (calling function) and callee (called function) takes place using arguments. The concept of modular approach of C++ is obtained from function. Following are the advantages of functions:

- (a) Support for modular programming
- (b) Reduction in program size
- (c) Code duplication is avoided
- (d) Code reusability is provided
- (e) Functions can be called repetitively
- (f) A set of functions can be used to form Libraries

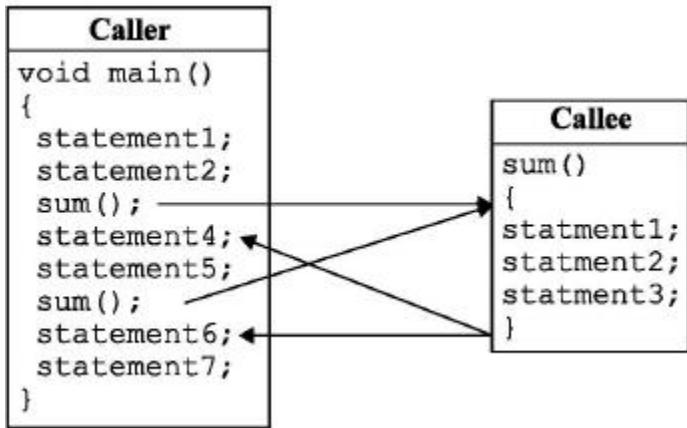


**Fig.5.1** Program without functions

The programs written in C++ like C language are highly dependent on functions. The C++ program is nothing but a combination of one or more functions. Execution of every C++ program starts with user defined function `main( )`. The method of using functions is slightly changed and enhanced in C++ as compared to C. The C++ language adds a few new features to functions like overloading of functions, default arguments etc. which are described ahead.

Like C, C++ supports two types of functions. They are (1) Library functions, and (2) User defined functions. The library functions can be used in any program by including respective header files. The header files must be included using `# include` preprocessor directive. For example, a mathematical function uses `math.h` header file. Various library functions are described at the end of this chapter.

The programmer can also define and use his/her own functions for performing some specific tasks. Such functions are called as user-defined functions. The user-defined functions are those that are defined in a class. They are called as member functions. Various member functions are described further in [Chapter 6](#).



**Fig.5.2** Program with functions

## 5.2 THE MAIN( ) FUNCTION IN C AND C++

In C++, the function `main( )` always returns integer value to operating system by default. The return value of the function `main( )` is used by the operating system to check whether the program is executed successfully or not. If the returned value is zero (0), it means that the program is executed successfully. A *non-zero* value indicates that the program execution was unsuccessful. The statement `return 0` indicates that the program execution is successful. The prototype of `main( )` function in C++ follows below:

```

int main( );
int main(int argc, char *argv[]);
  
```

The return statement is used to return value. If there is no return statement, compiler displays a warning message "Function should return a value". The `main( )` function in C and C++ is explained with examples in [Table 5.1](#) below.

**Table 5.1** Function `main( )` in C and C++

main( ) in C++	main( ) in C
<pre> int main ( ) {     Statement1;     Statement2;     return 0; }   </pre>	<pre> main ( ) {     Statement1;     Statement2; }   </pre>
(a) return type is specified (b) The return statement is used.	(a) No return type is specified and not necessary. (b) No return statement is used.

(c) Can be declared as void. If declared void, no need of return statement.

(c) Can be declared as void.

Every function including main( ) by default returns an integer value, hence the keyword int before function main( ) is not compulsory. In C there is no return type to function main( ). The function can be defined as given in Table 5.1.

The function main( ) can be declared as void and return statement can be omitted. But according to ANSI standard the main( ) function should return a value. Thus, the function declared as int main( ) will surely work on any operating system. The function main( ) neither works as inline nor as overloaded. The inline and overloading of functions are illustrated ahead.

### 5.3 PARTS OF FUNCTION

A function has the following parts:

- (a) Function prototype declaration
- (b) Definition of a function (function declarator)
- (c) Function call
- (d) Actual and formal arguments
- (e) The return statement?

#### (1) FUNCTION PROTOTYPES

We use many built-in functions. The prototypes of these functions are given in the respective header files. We include them using # include directive. In C++, while defining user-defined functions, it is unavoidable to declare its prototype. A prototype statement helps the compiler to check the return and argument types of the function.

A function prototype declaration consists of the function return type, name, and arguments list. It tells the compiler (a) the name of the function, (b) the type of value returned, and (c) the type and number of arguments.

When the programmer defines the function, the definition of function must be like its prototype declaration. If the programmer makes a mistake, the compiler flags an error message. The function prototype declaration statement is always terminated with semi-colon. The following statements are the examples of function prototypes:

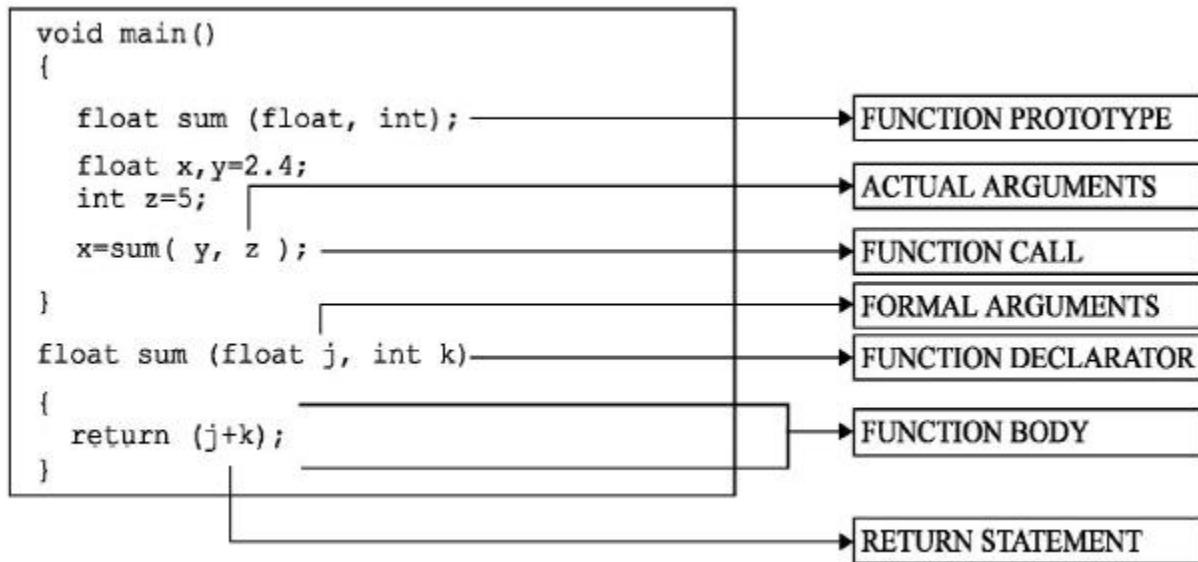
- (a) void show (void);
- (b) float sum (float, int);
- (c) float sum ( float x, int y);

In example (a), the return type is void i.e., the function won't return any value.

The void functions are always without return statement. The void argument means that the function won't require any argument. By default every function returns an integer value. To return a non-integer value, the data type should be mentioned in function prototype and definition. While writing the definition of function the return type must be preceded by the function name and it is optional if return type is default (int).

In example (b) the prototype of function sum( ) is declared. Its return type is float and arguments are float and integer type respectively. It is shown in Figure 5.3.

In example (c) with argument type, argument names are also declared. It is optional and also not compulsory to use the same variable name in the program.?



**Fig.5.3** Parts of function

## (2) FUNCTION DEFINITION

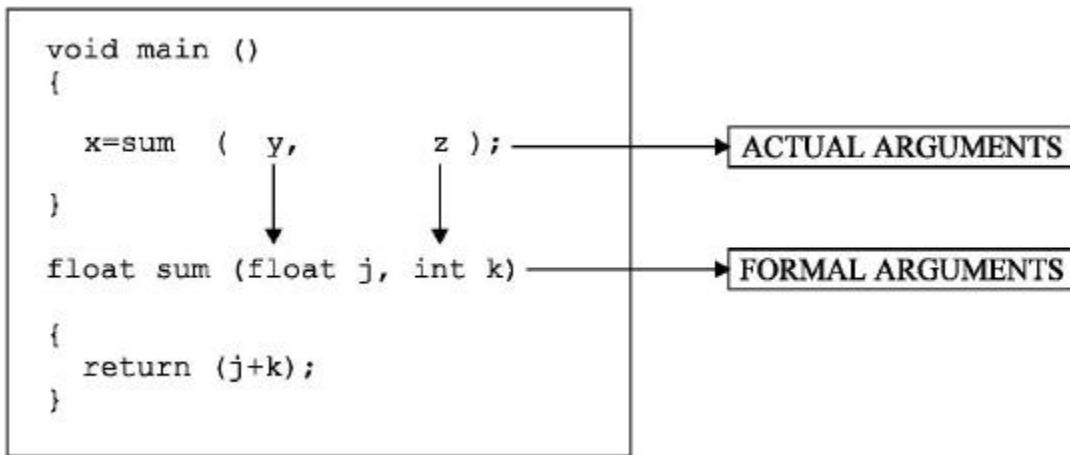
The first line is called function declarator and is followed by function body. The block of statements followed by function declarator is called as function definition. The declarator and function prototype declaration should match each other. The function body is enclosed with curly braces. The function can be defined anywhere. If the function is defined before its caller, then its prototype declaration is optional.

## (3) FUNCTION CALL

A function is a latent body. It gets activated only when a call to function is invoked. A function must be called by its name, followed by argument list enclosed in parenthesis and terminated by semi-colon.

## (4) ACTUAL AND FORMAL ARGUMENTS

The arguments declared in caller function and given in the function call are called as actual arguments. The arguments declared in the function declarator are known as formal arguments. For example, [Figure 5.4](#) shows these concepts.



**Fig.5.4** Actual and formal arguments

As shown in [Figure 5.4](#), variables y and z are actual arguments and j and k are formal arguments. The values of y and z are stored in j and k respectively. The values of actual arguments are assigned to formal arguments. The function uses formal arguments for computing.

## (5) THE RETURN STATEMENT

The `return` statement is used to `return` value to the caller function. The `return` statement returns only one value at a time. When a `return` statement is encountered, compiler transfers the control of the program to caller function. The syntax of `return` statement is as follows:

`return (variable name);` or `return variable name;`

The parenthesis is optional.

### 5.1 Write a program to declare prototype of function `sum()`. Define the function `sum()` exactly similar to its prototype.

```

#include <iostream.h>
#include <constream.h>

int main( )
{
    clrscr( );
    float sum(int, float ); // function prototype
    int a=20;
    float s,b=2.5;
    s=sum(a,b);
    cout <<"Sum=" <<s;
    return 0;
}

float sum (int x, float y)
{
    return (x+y);
}

```

**OUTPUT:****Sum = 22.5**

**Explanation:** In the above program, the prototype of function `sum( )` is declared. The prototype instructs the compiler that the function `sum( )` should return `float` value. The types of arguments used by function `sum( )` are `int` and `float` respectively. The values of variables `a` and `b` are passed to function `sum( )`. These values are assigned to formal arguments `x` and `y`. The sum of `x` and `y` is calculated and returned. The return value of function is assigned to float variable `s`.

**5.2 Write a program to declare and define void function.**

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

void main( )
{
    clrscr( );

    void show(void);
    show( );
}

void show( )
{
    cout <<"\n In show( )";
}
```

**OUTPUT****In show()**

**Explanation:** In the above program, the prototype of function `show( )` is given preceding `void` key-word. The statement `show( )` invokes the `show( )` function that displays the message "In `show( )`".

## 5.4 PASSING ARGUMENTS

The main objective of passing argument to function is message passing. The message passing is also known as communication between two functions i.e., between caller and callee functions. There are three methods by which we can pass values to the function. These methods are.

- (a) Call by value (pass by value),
- (b) Call by address (pass by address), and
- (c) Call by reference (pass by reference).

C++ supports all these three types of passing values to the function whereas C supports only the first two types. The arguments used to send values to function are known as input-arguments. The arguments used to return results are known as output arguments. The arguments used to send as well as return results are known as input-output arguments. While passing values the following conditions should be fulfilled.

The data type and the number of actual and formal arguments should be same both in caller and callee functions. Extra arguments are discarded if they are declared. If the formal

arguments are more than the actual arguments then the extra arguments appear as garbage. Any mismatch in the data type will produce the unexpected result.

### (1) PASS BY VALUE

In this type, value of actual argument is passed to the formal argument and operation is done on the formal arguments. Any change in the formal argument does not effect the actual argument because formal arguments are photocopy of actual arguments. Hence, when function is called by call by value method, it does not affect the actual contents of actual arguments. Changes made in the formal arguments are local to the block of called function. Once control returns back to the calling function the changes made vanish. The following example illustrates the use of call by value:

#### 5.3 Write a program to demonstrate call by value.

```
# include <iostream.h>
# include <constream.h>
void main( )
{
    clrscr( );
    int x,y;
    void change(int , int);
    cout<<"\n Enter Values of X & Y :";
    cin>>x>>y;
    change(x,y);
    cout <<"\n\n In function main ( ) ";
    cout<<"\n Values  X=<<x <<" and  Y= " <<y;
    cout<<"\n Address X=<<(unsigned)&x <<" and  Y= " <<(unsigned)&y;
}

void change(int a, int b)
{
    int k;  k=a;  a=b;  b=k;
    cout <<"\n In function change ( ) ";
    cout<<"\n Values  X=<<a <<" and  Y= " <<b;
    cout<<"\n Address X=<<(unsigned)&a <<" and  Y= " <<(unsigned)&b;

}
```

#### OUTPUT

Enter Values of X & Y :5 4

In function change()

Values X=4 and Y= 5

Address X=4090 and Y= 4092

In function main()

Values X=5 and Y= 4

Address X=4096 and Y= 4094

**Explanation:** In the above program, we are passing values of actual arguments x and y to function change(). The formal arguments a and b of function change() receive these values. The values are exchanged i.e., value of a is assigned to b and vice versa. They are displayed on the screen. When the control returns back to the main(), the changes made in function change() vanish because a and b are local variables of function change()

). In the `main()` the values of `x` and `y` are printed as they are read from the keyboard. In call by value method, the formal argument acts as duplicate of the actual argument. The addresses of actual and formal arguments are different. Thus, changes made with the variables are temporary.

## (2) PASS BY ADDRESS

In this type, instead of passing values, addresses are passed. Function operates on addresses rather than values. Here the formal arguments are pointers to the actual arguments. Hence changes made in the arguments are permanent. The following example illustrates passing the arguments to the function by pass by address method.

### 5.4 Write a program to demonstrate pass by address.

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

void main( )
{
    clrscr( );
    int x,y;
    void change(int* , int* );
    cout<<"\n Enter Values of X & Y :";
    cin>>x>>y;
    change(&x,&y);
    cout <<"\n In main( ) ";
    cout <<"\n Values      X="<<x <<" and  Y="<<y;
    cout <<"\n Addresses X="<<(unsigned)&x <<" and  Y="<<(unsigned)&y;
}
void change(int *a, int *b)
{
    int *k;
    *k=*a;
    *a=*b;
    *b=*k;
    cout <<"\n In change( ) ";
    cout <<"\n Values      X="<<*a <<" and  Y="<<*b;
    cout <<"\n Addresses X="<<(unsigned)a <<" and  Y="<<(unsigned)b;
}
```

### OUTPUT

Enter Values of X & Y :5 4

In change( )

Values X=4 and Y=5

Addresses X=4096 and Y=4094

In main( )

Values X=4 and Y=5

Addresses X=4096 and Y=4094

**Explanation:** In the above example, we are passing addresses of actual arguments to the function `change()`. The formal arguments are declared as pointers in the function declarator of `change()`. The formal arguments receive addresses of actual arguments i.e., formal arguments point to the actual argument. Here the `change()` function operates on the addresses of actual argument through pointers. The addresses of actual

arguments and formal arguments are same. Hence, the changes made in the values are permanent.

### (3) PASSING BY REFERENCE

C passes arguments by value and address. In C++ it is possible to pass arguments by value, address and reference. C++ reference types, declared with & operator are nearly identical but not exactly same to pointer types. They declare aliases for object variables and allow the programmer to pass arguments by reference to functions. The reference decelerator (&) can be used to declare references outside functions.

```
int k = 0;  
int &kk = k; // kk is an alias for k  
kk = 2; // same effect as k = 2
```

This creates the lvalue kk as an alias (assumed name) for k, provided that the initializer is the same type as the reference. Any operation on kk will give the same result as operations on k.

#### *Example*

```
Kk = 5 // assigns 5 to k and  
&kk // returns the address of k.
```

The reference decelerator can also be used to declare reference type parameters within a function.

```
void funcA (int i);  
void funcB (int &kk); // kk is type "reference to int"
```

```
int s=4;  
funcA(s); // s passed by value  
funcB(s); // s passed by reference
```

The s argument passed by reference can be modified directly by funcB whereas, funcA receives a duplicate copy (not actual) of the s argument (passed by value). Due to this reason variable s itself cannot be modified by funcA. When an actual variable s is not appearing in the above example, argument s is passed by value, the matching formal argument in the function obtains a copy of s. Any alteration to this copy inside the scope of the function body are not thrown back in the value of s itself.

Absolutely, the function can return a value that could be used later to change s, but the function cannot directly alter a parameter passed by value. The conventional C method for changing s operates on the address of actual argument (&s), the address of s, rather than s itself. Although &s is passed by value, the function can access s through the copy of &s it receives. Even though the function does not need to alter s, it is still useful (though subject possibly to risky secondary results) to pass &s, especially if s is a large data structure. Passing s directly by value contains the useless copying of the data structure.

#### **5.5 Write a program to pass the value of variable by value, reference, and address and display the result.**

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

```

void main( )
{
clrscr( );
void funA(int s);
void funB(int & );
void funC(int *);

int s=4;           // initial value
funA(s);          // s passed by value
cout <<"\nValue of s= "<<s <<" Addresss of s : "<<unsigned(&s);
funB(s);          // s passed by reference
cout <<"\nValue of s= "<<s <<" Addresss of s : "<<unsigned(&s);
funC(&s);         // s passed by reference (C style )
cout <<"\nValue of s= "<<s <<" Addresss of s : "<<unsigned(&s);
}

void funA( int i)
{i++; }

void funB(int &k)
{
    k++;

}

void funC(int *j) { ++*j; }

```

## **OUTPUT**

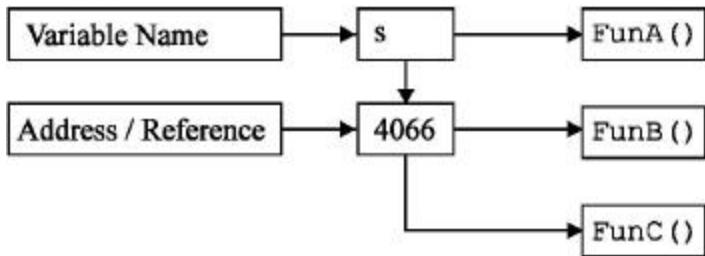
**Value of s= 4 Addresss of s : 4096**

**Value of s= 5 Addresss of s : 4096**

**Value of s= 6 Addresss of s : 4096?**

**Explanation:** In the above example, funA( ), funB( ) and funC( ) i.e., three functions are declared. The integer variable s is declared and initialized to 4. The funA( ) is invoked and value of s is passed by value. In function funA( ) the value of s is incremented. After execution of funA( ) the value of s is printed in main( ) which is same as previous one. Thus, passing variable by value cannot change the contents of the variable.

The value of s is again passed to funB( ). The function funB( ) receives the value of s by reference. The value of s is received by variable k by reference i.e., variable k is an alias for variable s and both have the same memory location. The variable k is incremented. Thus, any change made in reference variable effects the actual variable. Thus, after execution of funB( ) the printed value of s is equal to 5.



**Fig.5.5** Argument passing methods

The third function, `func( )` uses conventional C style. Here, address of the variable `s` is passed to `func( )`. The formal argument `*j` is pointer to the actual argument. Thus, any change made through pointer `j` also reflects on the actual variable. Thus, we can change the value of variable using call by reference. [Figure 5.5](#) explains the methods of passing arguments to function.

In function declarator and function prototype declaration the formal arguments are preceded by & operator. The reference type variable can be used in the same manner like other ordinary variables. The following points related to reference parameters are given below:

- (a) A reference may not be null. It should always refer to a legal variable.
- (b) Once declared, a reference should not be altered to referrer to another object.
- (c) A reference variable does not need any explicit address manipulation to access the actual value of the variable.

## 5.5 LVALUES AND RVALUES

### (1) LVALUES (LEFT VALUES)

A lvalue is an object locator. It is an expression that points to an object. An example of an lvalue expression is `*k` which results to a non-null pointer. A changeable lvalue is an identifier or expression that is related to an object that can be accessed and suitably modified in computer memory. A `const` pointer to a constant is not a changeable lvalue. A pointer to a constant can be altered (not its dereferenced value). A lvalue could suitably stand on the left (the assignment side) of an assignment statement. Now, only changeable lvalue can legally stand on the left of an assignment statement. For example, suppose `x` and `y` are non-constant integer identifiers with appropriate allocated memory. Their lvalues are changeable. The following expressions are legal:

`x = 1;`  
`Y = x + y` are legal expressions.

### (2) RVALUES (RIGHT VALUES)

The statement `x + y` is not a lvalue. `x + y = z` is invalid because the expression on the left is not related to a variable. Such expressions are often called rvalues.

## 5.6 RETURN BY REFERENCE

We have studied the reference variable and it's functioning. A reference allows creating alias for the preexisting variable. A reference can also be returned by the function. A function that returns reference variable is in fact an alias for referred variable. This technique of returning reference is used to establish cascade of member functions calls in operator overloading. Consider the following example:

The program given below illustrates return by reference.

### 5.6 Write a program to return a value by reference.

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

void main( )
{
    clrscr( );
    int & min ( int &j, int &k);
    int a=18,b=11,c;
    c=min (a,b);
    cout <<"Minimum Value = "<<c;
}
```

```
int & min(int &j, int &k)
{
    if (k<j ) return k;
    else
        return j;
}
```

#### OUTPUT

**Minimum Value = 11**

**Explanation:** In the above program, the statement `int & min ( int &j, int &k)` declares prototype of function `min()`. The `&` reference operator is used because the function returns reference to `int` and also receives arguments as reference. The function `min()` receives two integers as reference and returns minimum value out of two by reference.

### 5.7 Write a program to demonstrate return by reference.

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

int & large( int & p, int & q);

void main ( )
{   clrscr( );
    int l,k;
    cout <<"\n Enter values of l and k : ";
    cin>>l>>k;
    large(l,k)=120;
    cout <<" l= "<<l << " k="<<k;
}
```

```

int & large(int & p, int & q)
{
    if (p>q) return p;
    else return q;
}

```

### **OUTPUT**

**Enter values of l and k : 48**

**l=4 k=120**

**Enter values of l and k : 92**

**l= 120 k=2**

**Explanation:** In function main( ), the statement large (l, k)=120; calls the function large( ). It returns reference to the variable containing larger value and assigns the value 120 to it. The return type of function large( ) is int & (reference), it indicates that the call to function large( ) can be written on the left side of the assignment operator. Consequently, the statement large (l,k)=120; is legal and assigns 120 to the variable containing larger value.

## **5.7 RETURNING MORE VALUES BY REFERENCE**

The return( ) statement has one big limitation that it can return only one value at a time. The return( ) statement is only used when values are passed by value to functions. Call by address and call by reference, accesses memory location directly. When the user wants to return more than one value from function, he/she should pass values by address or by reference method. In C, for returning more values, call by address is used. In C++ we have one additional method i.e., call by reference. Consider the following program:

### **5.8 Write a program to return more than one value from function by reference.**

```

#include <iostream.h>
#include <constream.h>

void main( )
{
    int sq,cb,n;
    void more(int &, int &, int);
    cout <<"\n Enter a Number : ";
    cin>>n;
    more(sq,cb,n);
    cout <<"\n Square ="<<sq;
    cout <<"\n Cube   ="<<cb;
}

void more(int & s, int & c,int j)
{
    s=j*j;
    c=j*j*j;
}

```

### **OUTPUT**

**Enter a Number : 2**

**Square =4**

**Cube =8**

**Explanation:** In the above program, the function `more()` is used to perform square and cube of an integer passed. The three variables `sq`, `cb` and `n` of integer type are declared. The number entered by the user is stored in variable `n`. The variables `sq`, `cb` and `n` are passed to function `more()`.

In function `more()`, `s` and `j` are reference variables and `j` is a local variable of function `more()`. The variable `s` is reference variable of `sq` and variable `c` is reference variable of `cb`. The variables `s` and `sq` and `c` and `cb` have the same memory locations. Thus, any value assigned to `s` and `c` can be accessed by `sq` and `cb`. The variable `s` is assigned the square of `j` (`n`) and variable `cb` is assigned with cube of `j` (`n`).

## 5.8 DEFAULT ARGUMENTS

Usually, a function is called with all the arguments as declared in function prototype declaration and function definition. C++ compiler lets the programmer to assign default values in the function prototype declaration/function declarator of the function. When the function is called with less parameter or without parameters, the default values are used for the operations.

It is not allowed to assign default value to any variable, which is in between the variable list. Variables lacking default values are written first, followed by the variables containing default values. This is because the convention of storing variables on the stack in C++ is from right to left.

The default values can be specified in function prototype declaration or function declarator. Normally, the default values are placed in function prototype declaration. In case the function is defined before caller function, then there is no need to declare function prototype. Hence, the default arguments are placed in function declarator. The compiler checks for the default values in function prototype and function declarator and provides these values to those arguments that are omitted in function call.

The default arguments are useful while making a function call if we do not want to take effort for passing arguments that are always same. It is also useful when we update an old function by adding more arguments in it. Using default arguments, the function calls can continue to use previous arguments with the new arguments.

The example given below declares the default arguments.

### **Example**

(a) `int sum (int a,int b=10 ,int c = 15,int d=20)`

In this example, function `sum()` has four arguments. They are `a`, `b`, `c` and `d` of integer types. The variable `b`, `c`, and `d` are initialized with 10, 15 and 20 respectively. These values are used when the function `sum()` is called with fewer arguments.

## 5.9 Write a program to define function `sum()` with default arguments.

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

```
int main( )
{
```

```

clrscr();
int sum (int a,int b=10 ,int c=15,int d=20 ); // Function prototype

    int a=2;
    int b=3;
    int c=4;
    int d=5;

cout <<"Sum=" <<sum(a,b,c,d);
cout <<"\nSum=" <<sum(a,b,c);
cout <<"\nSum=" <<sum(a,b);
cout <<"\nSum=" <<sum(a);
cout <<"\nSum=" <<sum(b,c,d);

return 0;
}

    sum (int j, int k, int l, int m)
{
return (j+k+l+m);
}

```

### **OUTPUT:**

**Sum=14  
Sum=29  
Sum=40  
Sum=47  
Sum=32**

**Explanation:** In the above example, the prototype of variable `sum( )` is declared. The function `sum( )` has four arguments of integer types `a`, `b`, `c` and `d`. The variable `b`, `c` and `d` are initialized with default values 10, 15 and 20 respectively in function prototype declaration. The variable `a` is not initialized. The function `sum( )` is called five times. Each time the resultant sum has different values.

In the first `cout` statement, the function `sum( )` is called with four arguments. In this function call, no default values are used i.e., actual values of variables initialized are considered. Hence, the result is 14.

In the second `cout` statement, the function `sum( )` is called with three arguments (one less argument). In this function call, actual values of starting three variables `a`, `b`, `c` and default value of the fourth variable `d` are considered. Hence, the result is 29.

In the third `cout` statement, the function `sum( )` is called with two arguments (two arguments less). In this function call, actual values of starting two variables `a`, `b` and default values of last two variables `c`, `d` is considered. Hence, the result is 40.

In the fourth `cout` statement, the function `sum( )` is called with one argument. In this function call, the actual value of variable `a` and default values of variables `b`, `c` and `d` i.e., 10, 15 and 20 respectively, are taken into account. The result displayed is 47.

In the last cout statement, the function sum( ) is called with last three arguments. The actual values of b, c, and d variables are 3, 4 and 5 and last default value 20 is together taken into account. Hence the result is 32.

### 5.10 Write a program to find the area of a triangle by using default values and actual values. The formula for area of a triangle is $\frac{1}{2} * \text{base} * \text{height}$ .

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

int main( )
{
    clrscr( );

    float area(int base=3,int height=5 ); // Function prototype

    int base=2;
    int height=7;
    cout <<"Area of Triangle : " << area(base,height);
    cout <<"\nArea of Triangle : " << area (base);
    cout <<"\nArea of Triangle : " <<area(height);
    return 0;
}

float area( int j, int k)
{ return (.5*j*k); }
```

#### OUTPUT:

```
Area of Triangle : 7
Area of Triangle : 5
Area of Triangle : 17.5
```

**Explanation:** In the above program, the prototype of function area( ) is declared with two default values 3 and 5 for variables base and height respectively. In function main( ) the variable base and height are defined and initialized with 2 and 7 respectively. In the first call of the function area( ), function is called with both the arguments i.e., base and height. In this call no default arguments are used. In the second call the function area( ) is called with one argument (one less argument). In this call, one actual and one default value is used. The third call is the same as the second one.

## 5.9 THE CONST ARGUMENT

The constant variable can be declared using const keyword. The const keyword makes variable value stable. The constant variable should be initialized while declaring.

#### Syntax:

- (a) const <variable name> = <value>;
- (b) <function name> (const <type>\*<variable name>;)
- (c) int const x // in valid

(d) int const x =5 // valid

In statement **(a)**, the `const` modifier assigns an initial value to a variable that cannot be changed later by the program.

For example,

```
const age = 40;
```

Any attempt to change the contents of `const` variable `age` will produce a compiler error.

Using pointer, one can indirectly modify a `const` variable as shown below:

```
* (int *) &age = 45;
```

When the `const` variable is used with a pointer argument in a function's parameter list, the function cannot modify the variable that the pointer points to. For example,

```
int cube (const int *x, ...);
```

Here the `cube` function is prohibited from altering the integer variable.

## 5.10 INPUTTING DEFAULT ARGUMENTS

Default arguments are used when function is invoked with fewer arguments than declared. The default values are placed in function prototype or function declarator. It is also possible to directly invoke user-defined function in function prototype or function declarator. The return value of function is considered as default value. The following program illustrates this point:

### 5.11 Write a program to enter default values using function.

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

void main( )
{
    clrscr( );
    int input(void);
    void display ( int = input( ) );

    display(20); // first call with argument
    display( ); // second call with default argument
}
```

```
void display( int j)
{
    cout << "\nInteger is : " << j;
}
```

```
input( )
{
    int k;
    cout << "\n Enter an integer : ";
    cin >>k;
    return k;
}
```

## OUTPUT

Integer is : 20

**Enter an integer : 12**

**Integer is : 12**

**Explanation:** In the above program, user-defined functions `display( )` and `input( )` are declared and defined. The `display( )` function is used to display passed integer on the screen. The `input( )` function when executed asks for an integer. The function `input( )` is used as default value for function `display( )`. When a function `display( )` is invoked without argument, the `input( )` function assigned as default argument in function prototype is executed. User enters a number. Entered number is passed to `display( )` function and displayed on the screen.

## 5.11 INLINE FUNCTIONS

One of the prime factors behind using function is that code duplication in program is avoided and memory space is saved. When a function is defined and invoked, one set of instruction is created in the memory of the system. At each call, the control passes to the subroutine at a specified address in the memory. The CPU stores the memory address of instruction following the function calls into the stack and also pushes the arguments in the stack area. The compiler runs the function, stores the return values in a memory location or register and when execution completes, the control returns to the calling function. After this, execution resumes immediately to the next line following the function call statement. If the same function is called several times, each time the control is passed to the function i.e., the compiler uses the same set of instructions at each call. Due to this passing of control in between caller and callee functions, execution speed of program decreases. By preventing repetitive calls, program execution speed can be increased.

The C++ provides a mechanism called `inline` function. When a function is declared as `inline`, the compiler copies the code of the function in the calling function i.e., function body is inserted in place of function call during compilation. Passing of control between caller and callee functions is avoided. If the function is very large, in such a case `inline` function is not used because the compiler copies the contents in the called function that reduces the program execution speed. The `inline` function is mostly useful when calling function is small. It is advisable to use the `inline` function for only small functions. Inline mechanism increases execution performance in terms of speed. The overhead of repetitive function calls and returning values are removed. On the other hand, the program using `inline` functions needs more memory space since the `inline` functions are copied at every point where the function is invoked.

For defining an `inline` function, the `inline` keyword is preceded by the function name. The `inline` functions are declared below:

```
inline function - name
{
    statement1;
    statement2; // Function body
}
```

**Example:**

```
inline float square ( float k)
{
    return ( k*k );
}
```

The above example can be executed as given below:

```
j=square (2.5);
k=square (1.1+1.4);
```

After the execution of above statements, the values of `j` and `k` will be 6.25.

The `inline` keyword just makes an appeal to the compiler. The compiler may neglect this request if the function defined is too big in size or too convoluted. In such a case the function is treated as normal function.

Following are some situations where inline function may not work:

- (1) The function should not be recursive.
- (2) Function should not contain static variables.
- (3) Function containing control structure statements such as `switch`, `if`, `for` loop etc.
- (4) The function `main()` cannot work as inline.

The inline functions are similar to macros of C. The main limitation with macros is that they are not functions and errors are not checked at the time of compilation. The function offers better type testing and do not contain side effects as present in macros. Consider the following example:

### **5.12 Write a program to calculate square using inline function and macro.**

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

#define SQUARE(v) v * v

inline float square( float j)
{
    return (j*j);
}

void main( )
{
    clrscr( );
    int p=3, q=3, r,s;
    r=SQUARE (++p);
    s=square (++q);
    cout <<" r= "<<r<<"\n"<<" s= "<<s;
}
```

#### **OUTPUT**

```
r= 25
s= 16
```

**Explanation:** In the above program, the function `square()` is declared as inline function. The macro `SQUARE()` expanded into `r=++v *++v`. The variable `p` is incremented only once but in macros expansion it is incremented twice. Therefore, it gives wrong result.

### **5.13 Write a program to define function cube() as inline for calculating cube.**

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

void main( )
{
```

```

clrscr( );
int cube(int);
int j,k,v=5;
j = cube(3) ;
k = cube(v);

cout <<"\n Cube of j="<

```

inline int cube(int h)
{
    return (h*h*h);
}

```


```

## OUTPUT

**Cube of j=27**

**Cube of k=125**

**Explanation:** In the above program, the function `cube()` is declared as inline. The function `cube()` calculates cube of passed number. The function is declared as inline i.e., the statements of function `cube()` are inserted at the point of function call as shown in Figure 5.6.

## 5.12 FUNCTION OVERLOADING

It is possible in C++ to use the same function name for a number of times for different intentions. Defining multiple functions with same name is known as function overloading or function polymorphism. Polymorphism means one function having many forms. The overloaded function must be different in its argument list and with different data types. The examples of overloaded functions are given below. All the functions defined should be equivalent to their prototypes.

```

int sqr (int);
float sqr (float);
long sqr (long);

```

## 5.14 Write a program to calculate square of an integer and float number. Define function `sqr()`. Use function-overloading concept.

```

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

int sqr(int);
float sqr(float);

main( )
{
    clrscr( );
    int a=15;
    float b=2.5;
    cout <<"Square = "<<sqr(a) <<"\n";
    cout <<"Square = "<< sqr(b) <<"\n";
}

```

```

        return 0 ;
}

int sqr(int s)
{
    return (s*s);
}

float sqr (float j)
{
    return (j*j);
}

```

**OUTPUT** Square = 225  
**Square** =6.25

**Explanation:** In the above program, the function `sqr( )` is overloaded for `integer` and `float`. In the first call of the function `sqr( )`, an integer 15 is passed. The compiler executed integer version of the function and returns result 225. In the second call, a float value 2 . 5 is passed to function `sqr( )`. In this call, the compiler executed the float version of the function and returns the result 6 . 25. The selection of function to execute is made at run- time by the compiler according to the data type of variable passed.

**5.15 Write a program to find the area of rectangle, triangle and sphere. Use function overloading.**

```

#include<iostream.h>
#include<constream.h>
#define pi 3.142857142857142857
int calcarea(int length,int breadth);
float calcarea(double base,double height);
float calcarea(double radius);

void main( )
{
    int area1;
    float area2;
    double area3;
    area1=calcarea(10,20);
    area2=calcarea(4.5,2.1);
    area3=calcarea(3.12145);
    clrscr( );
    cout<<"Area of rectangle is : "<<area1<<endl;
    cout<<"Area of traingle is : "<<area2<<endl;
    cout<<"Area of sphere is : "<<area3<<endl;
    getch( );
}
int calcarea(int length,int breadth)
{
    return (length*breadth);
}
float calcarea(double base,double height)
{

    return ((0.5)*base*height);
}

```

```

        float calcarea(double radius)
{
    return ((4/3)*pi*radius*radius*radius);
}

```

#### **OUTPUT:**

**Area of rectangle is : 200  
 Area of traingle is : 4.725  
 Area of sphere is : 95.58589**

**Explanation:** In the above program, function `calcarea( )` is overloaded. It is used for finding the area of a rectangle, triangle and sphere. Explanation is same as the previous problem.

### **5.13 PRINCIPLES OF FUNCTION OVERLOADING**

(1) If two functions have the similar type and number of arguments (data type), the function cannot be overloaded. The return type may be similar or void, but argument data type or number of arguments must be different. For example,

```

(a) sum (int,int,int);
     sum (int,int);

```

Here, the above function can be overloaded. Though data type of arguments in both the functions are similar but number of arguments are different.

```

(b) sum (int,int,int);
     sum (float,float,float);

```

In the above example, number of arguments in both the functions are same, but data types are different. Hence, the above function can be overloaded.

(2) Passing constant values directly instead of variables also result in ambiguity. For example,

```

int sum (int,int);
float sum (float float float);

```

Here, `sum( )` is an overloaded function for integer and float values. If values are passed as follows

```

sum (2,3);
sum (1.1,2.3,4.3);

```

the compiler will flag an error because the compiler cannot distinguish between these two functions. Here, internal conversion of float to int is possible. Hence, in both the above calls integer version of function `sum( )` is executed. To overcome this problem, the user needs to do following things:

(i) Declare prototype declaration of all the overloaded functions before function `main( )`.

(ii) Passing argument-using variables as follows:

```

sum (a,b) ; // a and b are integer variables
sum (e,r,t,y); // e,r,t and y are float variables

```

Refer to program 5.22 for confirmation. Also try this program with direct values and by declaring function prototype inside `main( )`.

(3) The compiler attempts to find an accurate function definition that matches in types and number of arguments and invokes that function. The arguments passed are checked with all declared functions. If matching function is found then that function gets executed.

(4) If there is no accurate match found, compiler makes the implicit conversion of actual argument. For example, `char` is converted to `int` and `float` is converted to `double`. If all above steps fail then compiler performs user built functions.

The following example illustrates this point.

### 5.16 Write a program to overload a function and create a situation such that the compiler does integral conversion.

```
# include <iostream.h>
# include <conio.h>
    int sqr (int);
    float sqr (double );

main( )
{
    clrscr( );
    int a=15;
    float b=3.5;
    cout <<"Square = "<<sqr('A') <<"\n";
    cout <<"Square = "<< sqr(b) <<"\n";
    return 0 ;
}

int sqr (int s) { return (s*s); }
float sqr (double j) { return (j*j); }
```

#### OUTPUT

**Square = 4225**

**Square = 12.25**

**Explanation:** In the above program, the function `square` is overloaded. The first prototype of function `sqr (int)` is proposed for integer variable and second for double variable. In function `main( )`, the 'A' character data type value is passed to the function `sqr( )`. The compiler invokes the integer version of the function `sqr( )` because the `char` data type is compatible with integer. In the second call, a float value is passed to the function `sqr( )`. This time the compiler invokes the double data type version of function `sqr( )`.

When accurate match is not found the compiler makes internal conversion as seen in the above example. C++ automatically attempts to convert the arguments used to call a function into the type of arguments expected by the function.

(5) If internal conversion fails user-defined conversion is carried out with implicit conversion and integral promotion. The user-defined conversion is used with class objects.

(6) Sometime while making internal conversion, ambiguity is created if one data type is compatible with two or more data types. If such situation occurs, the compiler displays an error message.

Consider the following example. Given below are two versions of `sqr( )` function, one is for `long` data type and other for `double` data type.

```
long sqr ( long );
double sqr (double );
sqr (10); // function call
```

If an integer value is passed to the function `sqr( )`, the compiler will fall in confusion about the version to be executed and will result in an error message “Ambiguity between ‘`sqr(long)`’ and ‘`sqr(double)`’”.

(7) If a programs has two versions of functions i.e., one for `float` and second for `double` data type, then if we pass a float number, the double version of function is selected for execution. But the same is not applicable with integer and long integer.

### **5.17 Write a program to define overloaded function `add( )` for integer and float and perform the addition.**

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

int add (int , int, int);
float add ( float , float , float);

int main( )
{
clrscr( );
float fa,fb,fc,fd;
int ia,ib,ic,id;

cout <<"\n Enter values integer values for ia,ib and ic : ";
cin >>ia >>ib >>ic;
cout <<"\n Enter float values for fa,fb and fc : ";
cin >>fa >>fb >>fc;
id=add (ia,ib,ic);
cout <<"\n Addition : " <<id;
fd= add (fa,fb,fc);
cout <<"\n Addition : "<<fd;
return 0;
}
add ( int j, int k, int l) { return (j+k+l); }
float add (float a, float b, float c ) { return (a+b+c); }
```

#### **OUTPUT**

**Enter values integer values for ia,ib and ic : 1 2 4**

**Enter float values for fa,fb and fc : 2.2 3.1 4.5**

**Addition : 7**

**Addition : 9.8**

**Explanation:** In the above program, two versions of function `add( )` are declared. One for integer values and second for float values. Three integer and float values are entered through the keyboard. According to the data type the compiler executes appropriate function. The prototype of functions should be written before `main( )` function. If we write the prototype inside the `main( )`, the float version of the function will not be executed.

### **5.18 Write a program to define overloaded function. Invoke the overloaded function through another overloaded function.**

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

```

int add (void);
float add ( float , float , float);

float fa,fb,fc,fd;
int id;

int main( )
{
    clrscr( );
    cout <<"\n Enter float values for fa,fb and fc : ";
    cin >>fa >>fb >>fc;
    id=add ( );
    cout <<"\n Addition : " <<id;
    fd= add (fa,fb,fc);
    cout <<"\n Addition : "<<fd;

    return 0;
}

add ( )
{
int x;
x=add(fa,fb,fc);
return (x);
}

float add (float a, float b, float c ) { return (a+b+c);}

OUTPUT

```

**Enter float values for fa,fb and fc : 5.5 2.4 8.2**

**Addition : 16**

**Addition : 16.1**

**Explanation:** In the above program, the function `add()` is overloaded.

The `add(void)` function won't require any argument but returns integer. The `float add()` function requires three arguments of `float` type. Here, the arguments are `fa`, `fb`, `fc`, `fd` and `id` and are declared as global so that any function can call them. After entering three float values, the `add(void)` function is called. The `add(void)` function calls the `float` type of `add()`, the `float` type `add()` function calculates addition and returns results to function `add(void)`. The `add(void)` function returns addition as an integer. In the second call, the compiler calls the `float add()` function. This function returns addition of float values.

## 5.14 PRECAUTIONS WITH FUNCTION OVERLOADING

Function overloading is a powerful feature of C++. But, this facility should not be overused. Otherwise it becomes an additional overhead in terms of readability and maintenance.

Following precautions must be taken:

- (1) Only those functions that basically do the same task, on different sets of data, should be overloaded. The overloading of function with identical name but for different purposes should be avoided.

- (2) In function overloading, more than one function has to be actually defined and each of these occupy memory.
- (3) Instead of function overloading, using default arguments may make more sense and fewer overheads.
- (4) Declare function prototypes before `main( )` and pass variables instead of passing constant directly. This will avoid ambiguity that frequently occurs while overloading functions.

## 5.15 LIBRARY FUNCTIONS

(a) `ceil`, `ceill` and `floor`, `floorl`

The functions `ceil` and `ceill` round up the given float number. Where as the functions `floor` and `floorl` round down the float number. They are defined in `math.h` header file. Their declarations are given below:

**Declaration**

```
double ceil (double n);
double floor (double n);
long double ceill (long double (n));
long double floorl (long double (n));
```

The given below program illustrates the working of these functions.

### 5.19 Write a program to round down and up the given float number.

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

void main(void)
{
    clrscr();
    float num = 3.12;
    float d, u;

    d = floor(num);
    u = ceil(num);

    cout <<"\n Original number      is : "<<num;
    cout <<"\n Number rounded up    is : "<<u;
    cout <<"\n Number rounded down is : " <<d;
}
```

#### OUTPUT

**Original number is : 3.12**

**Number rounded up is : 4**

**Number rounded down is : 3**

**Explanation:** In the above program, the `float` variable `num` has a value 3.12. The `floor()` function converts it to the nearest and small integer number than variable `num` and returns it to variable `d` where as the `ceil( )` function converts the `float` number to the nearest and greater number than `num`.

**(b) Modf and modfl**

The function `modf` breaks double into integer and fraction elements and the function `modfl` breaks long double into integer and fraction elements. These functions return the fractional elements of given number. They are declared as given below:

**Declaration**

```
double modf (double n, double *ip);
long double modfl (long double (n), long double *(ip));
```

**5.20 Write a program to separate double number into integer and fractional parts.**

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

int main( )
{
    clrscr( );
    double f, i;
    double num = 211.57;

    f = modf(num, &i);
    cout <<"\n The Complete Number : "<<num;
    cout <<"\n The Integer elements : "<<i;
    cout <<"\n Fractional Elements : "<<f;

    return 0;
}
```

**OUTPUT**

**The Complete Number : 211.57**

**The Integer elements : 211**

**Fractional Elements : 0.57**

**Explanation:** In the above program, the function `modf ( )` separates a double number into separate integer and fractional parts. The second argument of the function is passed by reference that after execution holds the integer part of the number.

**(c) abs, fabs and labs**

The function `abs ( )` returns the absolute value of an integer. The `fabs ( )` returns the absolute value of a floating point number and `labs ( )` returns the absolute value of a long number.

**Declaration:**

```
Int abs (int n);
double fabs (double n);
long int labs (long int n);
```

If `abs` is invoked using `STDLIB.H`, it is executed as a macro that expands to inline code. If we need to use the original `abs` function then undefine the macro using statement `#undef abs` in the program, after `#include <stdlib.h>`.

**(d) norm**

This function is defined in `complex.h` header file and it is used to calculate the square of the absolute value.

**Syntax:** `double norm(complex n);`

### 5.21 Write a program to calculate the square of the complex number using `norm( )` function.

```
# include <iostream.h>
# include <complex.h>
# include <conio.h>
int main ( )
{
    clrscr( );
    double x=-12.5;
    cout <<norm(x);
    return 0;
}
```

#### OUTPUT

6.25

**Explanation:** In the above program, the `norm( )` function calculates the square of complex negative number.

(e) `complex( )`, `real( )`, `imag( )` and `conj( )`

`Complex( )` : This function is defined in `complex.h` header file and it creates complex numbers. Creates a complex number from given real and imaginary parts. The imaginary part is supposed to be 0 if not given. Complex is the constructor for the C++ class `complex`. This function returns the complex number with the given real and imaginary parts.

`Real( )` : Returns real part of the complex number.

`Imag( )` : Returns the imaginary part of the complex number.

`Conj( )` : Returns complex conjugate of a complex number.

**Syntax:** `complex complex(double real, double imag);`

**Syntax:** `double real( complex x);`

**Syntax:** `double imag( complex x);`

**Syntax:** `double conj( complex x);`

### 5.22 Write a program to return real, imaginary part and conjugate number.

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

int main(void)
{
    double a = 4.5, b = 7.4;
    clrscr( );
    complex c = complex(a,b);

    cout << "c = "<< c << "\n";

    cout << " and imaginary real part = " << imag(c) << "\n";
```

```
    cout << "c has complex conjugate = " << conj(c) << "\n";
    return 0;
}
```

## OUTPUT

**C = (4.5, 7.4)**

**and imaginary real part = 7.4**

**c has complex conjugate = (4.5, -7.4)**

**Explanation:** In the above program, variable **a** and **b** of double data type are declared and initialized with 4.5 and 7.4. By applying the **complex( )**, **imag( )** and **conj( )** functions, results are obtained. The results are shown in the output.

## SUMMARY

- (1) The programs written in C++ like C language, highly depends on functions. The C++ program is nothing but a combination of one or more functions.
- (2) In C++, the function **main( )** always returns an integer value to the operating system. The return value of the function **main( )** is used by the operating system to check whether the program is executed successfully or not.
- (3) A function prototype declaration consists of function's return type, name, and arguments list. When the programmer defines the function, the definition of function must be same like its prototype declaration.
- (4) In C++ it is possible to pass arguments by value or by reference. C++ reference type is declared with & operator and it is nearly identical but not exactly same as pointer types. They declare aliases for objects variables and allow the programmer to pass arguments by reference to functions.
- (5) An **lvalue** is an object locator and an expression that indicates an object.
- (6) The statement **x + y** is not an **lvalue**, **x + y = z** is invalid because the expression on the left is not related to a variable. Such expressions are often called **rvalues**.
- (7) C++ lets the programmer to assign default values in the function prototype declaration of the function. When the function is called with lesser parameters then the default values are used for the operations.
- (8) The constant variable can be declared using **const** keyword. The **const** keyword makes variable value stable. The constant variable should be initialized at the time of declaration.
- (9) The C++ provides a mechanism called **inline** function. When a function is declared as **inline** the compiler copies the code of the function in the calling function.
- (10) C++ makes it possible for the programmer to use the same function name for various times for different intentions. This is called as function overloading or function polymorphism.
- (11) The function **modf( )** breaks double into integer and fraction elements and the function **modfl( )** breaks long double into integer and fraction elements.
- (12) The functions **ceil( )** and **ceill( )** round up the given float number where as the functions **floor( )** and **floord( )** round down the float number.
- (13) The function **abs( )** returns the absolute value of an integer. The **fabs( )** returns the absolute value of a floating point number and **labs( )** returns the absolute value of a long number

## EXERCISES

**[A] Answer the following questions.**

- (1) What are the differences between C and C++ functions?
- (2) Describe different parts of a function?
- (3) What are `void` functions?
- (4) What does function prototype mean? Is it compulsory?
- (5) When is function prototype declaration not necessary?
- (6) What are default arguments?
- (7) Where are default arguments assigned?
- (8) How are default arguments entered at run-time?
- (9) What are `inline` functions? Discuss its advantages and disadvantages.
- (10) What are the `rvalue` and `lvalue` in an expression? Explain with examples.
- (11) What is the difference between call by value and call by reference? Illustrate them with examples.
- (12) What are constant arguments?
- (13) How is the value of a constant variable changed?
- (14) What is function overloading?
- (15) What are the rules for defining overloaded functions?
- (16) What precautions should we take while overloading function?
- (17) What is the difference between pointer and reference variable?
- (18) What is the difference between normal function and inline function?
- (19) What are actual and formal arguments?
- (20) How does `return` statement pass more than one value from function?

**[B] Answer the following by selecting the appropriate option.**

- (1) The `main()` function returns an integer value to
  - (a) operating system
  - (b) compiler
  - (c) `main()` function
  - (d) none of the above
- (2) The concept of declaring same function name with multiple definitions is
  - (a) function overloading
  - (b) operating overloading
  - (c) both (a) and (b)
  - (d) none of the above
- (3) The default arguments are used when
  - (a) function is called with less arguments
  - (b) function is `void`
  - (c) when arguments are passed by reference
  - (d) none of the above
- (4) The constant function
  - (a) cannot alter values of a variable
  - (b) can alter values of a constant variable
  - (c) makes its local variable constant
  - (d) none of the above
- (5) The function `abs()` returns
  - (a) absolute value

- (b) negative value  
(c) both (a) and (b)  
(d) none of the above
- (6) The use of parenthesis is optional with one of the following statement
- (a) return  
(b) main  
(c) clrscr  
(d) exit
- (7) Which of the following program will generate an error message:
- ```
void main( )  
{  
    return 0;  
}
```
- (a) main( ) cannot return value  
(b) void keyword is not allowed to main( )  
(c) function should return a non-zero value  
(d) return statement is not allowed
- (8) Which of the following statements are true?
- (1) A return type of void specifies that no value be returned  
(2) Functions by default return int value  
(3) The return type can only be int, char or double
- (a) (1) and (2)  
(b) (1),(2) and (3)  
(c) (1) and (3)  
(d) none of the above
- (9) C++ provides inline functions to facilitate reduce function call overhead, mainly for
- (a) small functions  
(b) large functions  
(c) member functions  
(d) none of the above
- (10) To—— an inline function, we must change it to an outline function.
- (a) debug  
(b) edit  
(c) remove  
(d) comment out
- (11) Identify which of the following function calls are allowed. The prototype declaration  
`is int sum(int j, int k=4, int l=3);`
- (1)sum(2);  
(2)sum(2, 3);  
(3)sum( )  
(4)sum(3, 4, 5);
- (a) 1,2 and 4  
(b) 2,3 and 4  
(c) 1 and 5

(d) 2 and 4

(12) It is possible to overload function

(a) when they have similar name and return type

(b) when they have similar name, return type and different arguments

(c) when they have similar name, different argument type and different return type

(d) when they have similar name, different types of argument, or different number of arguments and return type does not matter

(13) What will be the output of the following program?

```
int sub(int q)
{
    int m;
    m=sub(q-1);
    return (m);
}
void main()
{
    int r=sub(7);
    cout<<r;
}
```

(a) 0  
(b) stack overflow  
(c) compilation error  
(d) - 5

### [C] Attempt the following programs.

(1) Write a program to define a function with default argument. Whenever the function needs default values of arguments, it will prompt the user to enter a default value. Display the values.

(2) Write a program to define a constant variable. Change its value using pointer. Display default and the new assigned values with memory locations.

(3) Write a program to calculate the power of a given number. Define user-defined function power( ). Make it inline.

(4) Write a program to accept a double number. Separate its integer and fractional part.

(5) Write a program to calculate absolute value of long and float number.

(6) Write a program to round down and round up the floating point number.

(7) Write a program to overload the function uabs( ). The function should return absolute value of the given number for data type int and float.

(8) Write a program to enter quantity and rate through the keyboard. Calculate the amount by multiplying quantity and rate. If the fractional part of the amount is greater or equal to .50 then round up the number otherwise round down the number. Enter minimum 10 records. Calculate separately the total fractional parts of amount rounded up and down.

(9) Write a program to display only integer portion of the given float numbers without type casting.

(10) Write a program to accept a float number through the keyboard. Calculate the square of the number. Separate the float number into integer and fractional part.

Individually calculate the square of an integer and fractional parts and add them in another variable. Compare the two squares obtained. Write your observation regarding the squares obtained. (Note: set precision to 2)

(11) Write a program to calculate the square root of 1 to 10 numbers. Display the sum of integer parts and fractional parts of the square roots obtained. (Note: set precision to 2)

(12) Write a program to overload function to convert an `integer` number to an ASCII character and `float` to ASCII string.

(13) Write an `inline` function to display lines of different patterns.

(14) Write a program to find the power of integer number like `pow( )` library function.

(15) Write a program to return more than one values from function. Use call by reference method.

# 6

## CHAPTER

# Classes and Objects

C  
H  
A  
P

—• [6.1 Introduction](#)

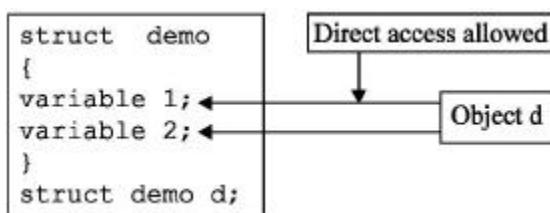
—• [6.2 Structures in C](#)

- [6.3 Structures in C++](#)
- [6.4 Classes in C++](#)
- [6.5 Declaring Objects](#)
- [6.6 The public Keyword](#)
- [6.7 The private Keyword](#)
- [6.8 The protected Keyword](#)
- [6.9 Defining Member Functions](#)
- [6.10 Characteristics of Member Functions](#)
- [6.11 Outside Member Function Inline](#)
- [6.12 Rules for Inline Functions](#)
- [6.13 Data Hiding or Encapsulation](#)
- [6.14 Classes, Objects and Memory](#)
- [6.15 Static Member Variables and Functions](#)
- [6.16 Static Object](#)
- [6.17 Array of Objects](#)
- [6.18 Objects as Function Arguments](#)
- [6.19 friend Functions](#)

- [6.20 The const Member Function](#)
- [6.21 Recursive Member Function](#)
- [6.22 Local Classes](#)
- [6.23 Empty, Static and Const Classes](#)
- [6.24 Member Functions and Non- Member Functions](#)
- [6.25 The main\( \) as a Member Function](#)
- [6.26 Overloading Member Functions](#)
- [6.27 Overloading main\( \) Function](#)
- [6.28 The main\( \), Member Function and Indirect Recursion](#)
- [6.29 Bit Fields and Classes](#)

## 6.1 INTRODUCTION

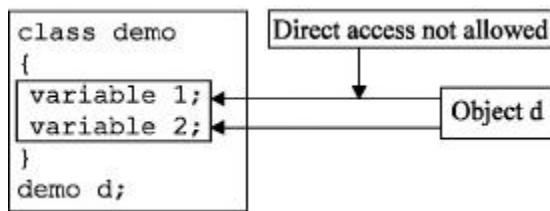
In C, structures are used to combine one or more data types. Functions are not permitted as members in structures. Later objects are declared which hold individual copies of member variables of structure. Objects can access the data members of the structures directly. Any outside function is also able to access the data members of the structure through the object. Thus, there is no security to member variables declared inside the structure in C as shown in [Figure 6.1](#).



**Fig.6.1** Structure in C

In C++, structure also combines functions with data members. C++ introduces a new keyword `class`. A class in C++ is similar to structure. Using class or structure, a programmer can merge one or more dissimilar data types and a new custom data type can be created. A class is nothing but grouping of variables of different data types with functions. Each variable of a class is called as member variable. Functions are called as member functions or methods.

In C++, it is possible to restrict access to data members directly by objects which is not possible in C. It is always a programmer's choice to allow or disallow direct access to data members. The mechanism of restricting access to data outside the class is called as ***data hiding*** or ***encapsulation***. In such a case, only the member functions can access the data. If `class` is used in place of `struct`, it restricts the access to data members as shown in [Figure 6.2](#).



**Fig.6.2** Class in C++

## 6.2 STRUCTURES IN C

Though, it is possible to combine more than one variable using structure in C, the structure has following limitations when used in C:

- (a) Only variables of different data types can be declared in the structure as member, functions are not allowed as members.
- (b) Direct access to data members is possible. It is because by default all the member variables are public. Hence, security to data or data hiding is not provided.
- (c) The `struct` data type is not treated as built in type i.e., use of `struct` keyword is necessary to declare objects.
- (d) The member variables cannot be initialized inside the structure.

The syntax of structure declaration is as follows:

### SYNTAX:

```

struct <struct name>
{
    Variable1;

    Variable2;
};

```

### EXAMPLE:

```

struct item
{
    int codeno;
    float prize;
    int qty;
};

```

In the above example, item is a structure name. The variables codeno, prize and qty are member variables. Thus, a custom data type is created by a combination of one or more variables.

The object of structure **item** can be declared as follows:

```
struct item a,*b
```

The object declaration is same as declaration of variables built in data types. The object **a** and pointer **\*b** can access the member variables of **struct item**. Use of keyword **struct** is necessary.

The operators (**•**) dot and (**->**) arrow are used to access the member variables of struct. The **dot** operator is used when simple object is declared and **arrow** operator is used when object is pointer to structure. The access of members can be accomplished as per the syntax given below:

**[Object name][Operator][Member variable name]**

When an object is a simple variable, access to members is done as below:

```
a.codeno  
a.prize  
a.qty
```

When an object is a pointer to structure then members are accessed as below:

```
a->codeno  
a->prize  
a->qty
```

The following program illustrates the above discussion.

### 6.1 Write a program to declare structure, access and initialize its members using object of structure type.

```
# include <stdio.h>  
# include <conio.h>  
  
struct item      // struct declaration  
{  
    int    codeno;    // codeno=200 not possible as per 6.2 section  
    float  prize;  
    int    qty;  
};  
  
void main( )  
{  
    struct item a,*b;    // object declaration  
    clrscr( );  
    a.codeno=123; // direct access & initialization of member variables  
    a.prize=150.75;  
    a.qty= 150;  
  
    printf ("\n With simple variable");  
    printf ("\n Codeno : %d ",a.codeno);  
    printf ("\n Prize  : %d",a.prize);  
    printf ("\n Qty    : %d",a.qty);  
  
    b->codeno=124; // direct access & initialization of member variables  
    b->prize=200.75;
```

```

b->qty= 75;

printf ("\n\n With pointer variable");
printf ("\n Codeno : %d ",b->codeno);
printf ("\n Prize : %d",b->prize);
printf ("\n Qty : %d",b->qty);
}

```

## **OUTPUT**

### **With simple variable**

**Codeno : 123**

**Prize : 150.75**

**Qty : 150**

### **With pointer to structure**

**Codeno : 124**

**Prize : 200.75**

**Qty : 75**

**Explanation:** The above program is compiled with C compiler. The following discussion is in accordance with C compiler. In the above program, the structure item is declared with three member variables. The initialization of member variables inside the struct is not permitted. The declaration of member variables is enclosed within the curly braces. The struct declaration is terminated by semi-colon.

In function main( ), the objects a and b are declared. Consider the following statement:

```
struct item a,*b; // object declaration
```

The struct must be preceded by structure name. The member variables can be accessed and initialization is done directly by object. The dot and arrow operators are used to access the member variables.

## **6.3 STRUCTURES IN C++**

No doubt, C++ has made various improvements in structure. To know the improvements made in C++, the last program is compiled and executed with C++ compiler. The explanation followed by this program discusses the various improvements.

### **6.2 Write a program to declare struct. Initialize and display contents of member variables.**

```

#include <iostream.h>
#include <constream.h>

struct item      // struct declaration
{
    int codeno;    // codeno=200 not possible
    float prize;
    int qty;
};

void main( )
{

```

```

item a,*b;      // object declaration
clrscr( );
a.codeno=123; // direct access & initialization of member variables
a.prize=150.75;
a.qty= 150;

cout <<"\n With simple variable";
cout <<"\n Codeno : "<<a.codeno;
cout <<"\n Prize   : "<<a.prize;
cout <<"\n Qty     : "<<a.qty;

b->codeno=124; // direct access & initialization of member variables
b->prize=200.75;
b->qty= 75;

cout <<"\n\n With pointer to structure";
cout <<"\n Codeno : "<<b->codeno;
cout <<"\n Prize   : "<<b->prize;
cout <<"\n Qty     : "<<b->qty;
}

```

**Explanation:** The above program is same as the last one. The output is also same, hence not shown. Consider the following statement:

```
item a,*b; // object declaration in c++
```

While declaring an object the keyword **struct** is omitted in C++ which is compulsory in C. The structure item is a user-defined data type. C++ behaves structure data type as built in type and allows variable declaration.

C++ introduces new keyword **class**, which is similar to structure. The other improvements are discussed with use of class in the following section.

## 6.4 CLASSES IN C++

Classes and structure are the same with only a small difference that is explained later. The following discussion of class is applicable to struct too.

The whole declaration of class is given in [Table 6.1](#). The class is used to pack data and function together. The class has a mechanism to prevent direct access to its members, which is the central idea of object-oriented programming. The class declaration is also known as formation of new abstract data type. The abstract data type can be used as basic data type such as **int**, **float** etc. The class consists of member data variables that hold data and member functions that operate on the member data variables of the same class.

**Table 6.1** Syntax and an example of class

| Syntax of class declaration                           | Example of class                                                                   |
|-------------------------------------------------------|------------------------------------------------------------------------------------|
| <pre>class &lt;name of class&gt; {     private:</pre> | <pre>class item      // class declaration {     private:         int codeno;</pre> |

|                                                                                                                                                    |                                                                            |
|----------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------|
| <pre> declaration of variables; prototype declaration of function;  public:  declaration of variables; prototype declaration of function; };</pre> | <pre> float prize; int qty; void values( ); public: void show( ); };</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------|

The `class` and `struct` are keywords. The `class` declaration is same as `struct` declaration. The declaration of a class is enclosed with curly braces and terminated by a semi-colon. The member variables and functions are divided in two sections i.e., `private` and `public`. The `private` and `public` keywords are terminated by colon (:). The object cannot directly access the member variables and functions declared in `private` section but it can access the data member variables and functions declared in `public` section. The private members of a class can only be accessed by public member function of the same class. Different sections of a class are illustrated with examples in following sections.

It is also possible to access private member variables directly like public member variables provided that the class should have at least one public member variable. Both the private and public member variables are stored in consecutive memory locations in the memory. A pointer to member variable provides address of member variable. By applying increment (++) and decrement (--) operations on pointer, we can access all private and public member variables of the class. The object of a class contains address of the first member variable. It can also be used to access the private or public data.

## 6.5 DECLARING OBJECTS

A class declaration only builds the structure of object. The member variables and functions are combined in the class. The declaration of objects is same as declaration of variables of basic data types. Defining objects of class data type is known as ***class instantiation***. When objects are created only during that moment, memory is allocated to them.

Consider the following examples:

- (a) `int x,y,z; // Declaration of integer variables`
- (b) `char a,b,c; // Declaration of character variables`
- (c) `item a,b, *c; // Declaration of object or class type variables`

In example (a) three variables `x`, `y` and `z` of `integer` types are declared. In example (b) three variables `a`, `b` and `c` of `char` type are declared. In the same fashion the third example declares the three objects `a`, `b` and `c` of class `item`. The object `*c` is pointer to class `item`.

An object is an abstract unit having following properties:

- (a) It is individual.
- (b) It points to a thing, either physical or logical that is identifiable by the user.
- (c) It holds data as well as operation method that handles data.

(d) Its scope is limited to the block in which it is defined.

## (1) ACCESSING CLASS MEMBERS

The object can access the public member variables and functions of a class by using operator dot (.) and arrow (->). The syntax is as follows:

**[Object name] [Operator] [Member name]**

To have access to members of class item, the statement would be,

a.show();

where a is an object and show( ) is a member function. The dot operator is used because a is a simple object.

In statement,

c->show();

\*c is pointer to class item; therefore, the arrow operator is used to access the member.

Consider the given example:

```
class item      // class declaration
{
    int codeno;
    float prize;
    int qty;
};
```

We replaced the struct keyword with class. If last few programs are executed with class, they won't work. For example,

```
void main ( )
{
    item a,*b;      // object declaration
    clrscr( );
    a.codeno=123;    // Direct access is not allowed
    a.prize=150.75;
    a.qty= 150;
}
```

The above program will generate error messages, one of them being "item::codeno' is not accessible". This is because the object cannot directly access the member variables of class that is possible with structure. Hence, we can say that the difference between class and struct is that the member variables of struct can be accessed directly by the object, whereas the member variables of class cannot be accessed directly by the object.

## 6.6 THE PUBLIC KEYWORD

In Section 6.3, we have noticed that the object directly accesses the member variables of structure whereas the same is not possible with class members. The keyword **public** can be used to allow object to access the member variables of class directly like structure.

The **public** keyword is written inside the class. It is terminated by colon (:). The member variables and functions declared followed by the keyword **public** can be accessed directly by the object. The declaration can be done as below:

```
class item      // class declaration
{
    public:      // public section begins
    int codeno;
    float prize;
    int qty;
};
```

The following program illustrates the use of `public` keyword with class.

### 6.3 Write a program to declare all members of a class as public. Access the elements using object.

```
# include <iostream.h>
# include <constream.h>
class item
{
    public:           // public section begins
    int codeno;
    float prize;
    int qty;

};

int main( )
{
    clrscr( );
    item one;          // object declaration
    one.codeno=123;   // member initialization
    one.prize=123.45;
    one.qty=150;

    cout <<"\n Codeno    = "<<one.codeno;
    cout <<"\n Prize      ="<<one.prize;
    cout <<"\n Quantity   ="<<one.qty;
    return 0;
}
```

#### OUTPUT

Codeno =123  
Prize =123.449997  
Quantity =150

**Explanation:** In the above program, the members of class item are declared followed by keyword `public`. The object `one` of class item, accesses the member variables directly. The member variables are initialized and values are displayed on the screen.

### 6.7 THE PRIVATE KEYWORD

The `private` keyword is used to prevent direct access to member variables or function by the object. The class by default produces this effect. The structure variables are by default `public`. To prevent member variables and functions of struct from direct access the `private` keyword is used. The syntax of `private` keyword is same as `public`.

The `private` keyword is terminated by colon. Consider the given example.

```
struct item
{
    private:           // private section begins
    int codeno;
    float prize;
    int qty;
};

// end of class
```

```

int main ( )
{
    clrscr( );
    item one;           // object declaration
    one.codeno=123;     // member initialization
    one.price=123.45;
    one.qty=150;
}

```

As soon as the above program is compiled, the compiler will display following error messages:

```

'item::codeno' is not accessible
'item::prize' is not accessible
'item::qty' is not accessible
'item::codeno' is not accessible
'item::prize' is not accessible

```

From the above discussion, we noticed that by default (without applying `public` or `private` keyword) the class members are private (not accessible) whereas the struct members are public (accessible).

The private members are not accessible by the object directly. To access the private members of a class, member functions of the same class are used. The member functions must be declared in the class in public section. An object can access the private members through the public member function.

## 6.8 THE PROTECTED KEYWORD

The access mechanism of `protected` keyword is same as `private` keyword.

The `protected` keyword is frequently used in inheritance of classes. Hence, its detailed description is given in the Chapter ***Inheritance***. **Table 6.2** illustrates the access difference between `private`, `protected` and `public` keywords.

**Table 6.2** Access limits of class members

| Access specifiers | Access permission |              |
|-------------------|-------------------|--------------|
|                   | Class members     | Class object |
| public            | allowed           | allowed      |
| private           | allowed           | disallowed   |
| protected         | allowed           | disallowed   |

## 6.9 DEFINING MEMBER FUNCTIONS

The member function must be declared inside the class. They can be defined in **a) private or public section b) inside or outside the class**. The member functions defined inside the

class are treated as inline function. If the member function is small then it should be defined inside the class, otherwise it should be defined outside the class.

If function is defined outside the class, its prototype declaration must be done inside the class. While defining the function, scope access operator and class name should precede the function name. The following program illustrates everything about member functions and how to access private member of the class.

### (1) MEMBER FUNCTION INSIDE THE CLASS

Member function inside the class can be declared in public or private section. The following program illustrates the use of member function inside the class in public section.

#### 6.4 Write a program to access private members of a class using member function.

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

struct item
{
    private:           // private section starts

        int codeno;
        float price;
        int qty;

    public:            // public section starts

        void show ( )   // member function
    {
        codeno=125;    // access to private members
        price=195;
        qty=200;

        cout <<"\n Codeno  ="<<codeno;
        cout <<"\n Price   ="<<price;
        cout <<"\n Quantity="<<qty;
    }

};

int main( )
{
    clrscr( );
    item one;          // object declaration
    one.show( );       // call to member function

    return 0;
}
```

#### OUTPUT

Codeno =125

**Price =195**

**Quantity=200**

**Explanation:** In the above program, the member function `show( )` is defined inside the class in public section. In function `main( )`, object one is declared. We know that an object has permission to access the public members of the class. The object one invokes the public member function `show( )`. The public member function can access the private members of the same class. The function `show( )` initializes the private member variables and displays the contents on the console. For the sake of understanding only one function is defined.

In the above program, the member function is defined inside the class in public section. Now the following program explains how to define private member function inside the class.

## **(2) PRIVATE MEMBER FUNCTION**

In the last section, we have learnt how to access private data of class using public member function. It is also possible to declare function in private section like data variables. To execute private member function, it must be invoked by public member function of the same class. A member function of a class can invoke any other member function of its own class. This method of invoking function is known as nesting of member function. When one member function invokes other member function, the frequent method of calling function is not used. The member function can be invoked by its name terminated by semi-colon only like normal function. The following program illustrates this point.

### **6.5 Write a program to declare private member function and access it using public member function.**

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

struct item
{
    private:           // private section starts
    int codeno;

    float price;
    int qty;

    void values()      // private member function
    {
        codeno=125;
        price=195;
        qty=200;
    }

    public:           // public section starts

    void show()       // public member function
    {
        values();     // call to private member functions
    }
}
```

```

        cout <<"\n Codeno  ="<<codeno;
        cout <<"\n Price   ="<<price;
        cout <<"\n Quantity="<<qty;
    }

};

int main( )
{
    clrscr( );
    item one;           // object declaration

    // one.values( );    // not accessible

    one.show( );         // call to public member function
    return 0;
}

```

## **OUTPUT**

**Codeno =125**

**Price =195**

**Quantity=200**

**Explanation:** In the above program, the private section of a class item contains one-member function values( ). The function show( ) is defined in public section. In function main( ), one is an object of class item. The object one cannot access the private member function. In order to execute the private member function, the private function must be invoked using public member function. In this example, the public member function show( ) invokes the private member function values( ). In the invocation of function values( ), object name and operator are not used.

## **(3) MEMBER FUNCTION OUTSIDE THE CLASS**

In the previous examples, we observed that the member functions are defined inside the class. The function prototype is also not declared. The functions defined inside the class are considered as inline function. If a function is small, it should be defined inside the class and if large it must be defined outside the class. To define a function outside the class the following care must be taken:

- (1) The prototype of function must be declared inside the class.
- (2) The function name must be preceded by class name and its return type separated by scope access operator.

The following example illustrates the function defined outside the class.

### **6.6 Write a program to define member function of class outside the class.**

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

```
class item
```

```

{
    private:           // private section starts

        int codeno;      // member data variables
        float price;
        int qty;

    public:            // public section starts

        void show (void); // prototype declaration

};

                                // end of class

void item:: show( ) // definition outside the class
{
    codeno=101;
    price=2342;
    qty=122;

    cout <<"\n Codeno ="<<codeno;
    cout <<"\n Price   ="<<price;
    cout <<"\n Quantity="<<qty;
}

int main( )
{
    clrscr( );
    item one;          // object declaration
    one.show( );       // call to public member function
    return 0;
}

```

### **OUTPUT**

**Codeno =101**

**Price =2342**

**Quantity=122**

**Explanation:** In the above program, the prototype of function `show( )` is declared inside the class and followed by it class definition is terminated. The body of function `show( )` is defined inside the class. Class name that it belongs to and its return type, precede the function name. The function declarator of function `show( )` is as follows:

```
void item:: show( )
```

Here, `void` is return type i.e., function is not returning a value. The `item` is a class name. The scope access operator separates the class name and function name and the body of function is defined after it.

## **6.10 CHARACTERISTICS OF MEMBER FUNCTIONS**

- (1) The difference between member and normal function is that the former function can be invoked freely where as the latter function only by using an object of the same class.
- (2) The same function can be used in any number of classes. This is possible because the scope of the function is limited to their classes and cannot overlap one another.
- (3) The private data or private function can be accessed by public member function. Other functions have no access permission.
- (4) The member function can invoke one another without using any object or dot operator.

## 6.11 OUTSIDE MEMBER FUNCTION INLINE

In the last chapter we saw how inline mechanism is useful for small functions. It is a good practice to declare function prototype inside the class and definition outside the class. The inline mechanism reduces overhead relating to accessing the member function. It provides better efficiency and allows quick execution of functions. An inline member function is similar to macros. Call to inline function in the program, puts the function code in the caller program. This is known as inline expansion. Inline functions are also called as open subroutines because their code is replaced at the place of function call in the caller function. The normal functions are known as closed subroutines because when such functions are called, the control passes to the function.

By default, all member functions defined inside the class are inline functions. The member function defined outside the class can be made `inline` by prefixing the keyword `inline` to function declarator as shown in [Figure 6.3](#).

|                     |                          |                         |                 |                                        |
|---------------------|--------------------------|-------------------------|-----------------|----------------------------------------|
| <code>inline</code> | <code>return type</code> | <code>class name</code> | <code>::</code> | <code>function name (signature)</code> |
|---------------------|--------------------------|-------------------------|-----------------|----------------------------------------|

**Fig.6.3** Inline function outside the class

The `inline` is a keyword and acts as function qualifier. The `return type` is functions return type i.e., the function returns values of this type. The `class name` is the name of class that the function belongs to. Scope access operator separates class name and function name. Signature means argument list passed function. The following program illustrates inline function outside the class.

### 6.7 Write a program to declare outside function inline.

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

class item
{
    private:                      // private section starts

        int codeno;                // member data variables
        float price;
        int qty;

    public:                       // public section starts
```

```

void show (void);           // prototype declaration

};

// end of class

inline void item:: show ()   // outside inline function
{
    codeno=213;
    price=2022;
    qty=150;
    cout <<"\n Codeno ="<<codeno;

    cout <<"\n Price   ="<<price;
    cout <<"\n Quantity="<<qty;
}

int main( )
{
    clrscr( );
    item one;      // object declaration
    one.show();     // call to public member function (inline)
    return 0;
}

```

## OUTPUT

**Codeno =213**

**Price =2022**

**Quantity=150**

**Explanation:** The above program is the same as the last one. The only difference is that the function `show()` is defined as inline outside the class. The function declarator is `inline void item:: show()`.

## 6.12 RULES FOR INLINE FUNCTIONS

(1) Use inline functions rarely. Apply only under appropriate circumstances.

(2) Inline function can be used when the member function contains few statements. For example,

```

inline int item :: square (int x)
{
    return (x*x);
}

```

(3) If function takes more time to execute, then it must be declared as inline. The following inline function cannot be expanded as inline:

```

inline void item:: show( )
{
    cout <<"\n Codeno ="<<codeno;
    cout <<"\n Price   ="<<price;
    cout <<"\n Quantity="<<qty;
}

```

The member function that performs input and output operation requires more time. Inline functions have one drawback, the entire code of the function is placed at the point of call in

caller function and it must be noted at compile time. Therefore, inline functions cannot be placed in standard library or run-time library.

### 6.13 DATA HIDING OR ENCAPSULATION

Data hiding is also known as encapsulation. It is a process of forming objects. An encapsulated object is often called as an abstract data type. We need to encapsulate data because the programmer often makes various mistakes and the data gets changed accidentally. Thus, to protect data we need to construct a secure and impassable wall to protect the data. Data hiding is nothing but making data variable of the class or struct **private**. Thus, private data cannot be accessed directly by the object. The objects using public member functions of the same class can access the private data of the class. The keywords **private** and **protected** are used for hiding the data. Table 6.3 shows the brief description of access specifiers. The following program explains data hiding:

#### 6.8 Write a program to calculate simple interest. Hide the data elements of the class using **private** keyword.

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

class interest
{
    private :
        float p_amount;      // principle amount
        float rate;          // rate of interest
        float period;         // number of years
        float interest;
        float t_amount;       // total amount

    public :
        void in( )
        {
            cout <<" Principle Amount : "; cin>>p_amount;
            cout <<" Rate of Interest : "; cin>>rate;
            cout <<" Number of years : "; cin>>period;
            interest=(p_amount*period*rate)/100;
            t_amount=interest+p_amount;
        }

        void show( )
        {
            cout <<"\n Principle Amount : "<<p_amount;
            cout <<"\n Rate of Interest : "<<rate;
            cout <<"\n Number of years : "<<period;
            cout <<"\n Interest           : "<<interest;

            cout <<"\n Total Amount       : "<<t_amount;
        }

};
```

```

int main( )
{
    clrscr();
    interest r;
    r.in();
    r.show();
    return 0;
}
OUTPUT
Principle Amount : 5000
Rate of Interest : 2
Number of years : 3
Principle Amount : 5000
Rate of Interest : 2
Number of years : 3
Interest : 300
Total Amount : 530

```

**Explanation:** In the above program, the class interest is declared with the data members p\_amount, float rate, period, interest and t\_amount of float type. These entire data elements are declared in private section hence it is hidden (encapsulated) and cannot be directly accessed by the object. Here, member function in( ), is used to read data through the keyboard. The function show( ) is used to display the contents of the variables.

**Table 6.3** Access specifiers and their scope

| Access specifiers | Members of the class | Class objects   |
|-------------------|----------------------|-----------------|
| <b>public</b>     | access possible      | access possible |
| <b>private</b>    | access possible      | no access       |
| <b>protected</b>  | access possible      | no access       |

As described in **Table 6.3**, the class object can access public member of the class directly without use of member function. The private and protected mechanism do not allow objects to access data directly. The object can access private or protected members only through public member functions of the same class. The following program explains the working of the above keywords.

**6.9 Write a program to declare class with private, public and protected sections. Declare object and access data elements of these different sections.**

```

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

```

```

class access
{
    private :
    int p;

void getp( )
{
    cout <<" In pget( ) enter value of p :";
    cin>>p;
}
public:
int h;

void geth( )
{
    cout <<" In geth( ) : "<<endl;
    getp( );
    getm( );

cout <<" p = "<<p <<" h = "<<h <<" m = "<<m;
}

protected :
int m;
void getm( )
{
    cout <<" In mget( ) enter value of m : ";
    cin>>m;
}

};

void main( )
{
    clrscr( );
    access a;// object declaration
        // a.p=2; // access to private member is not possible
        // a.pget( ) // _____
        // a.m=5; // access to protected member is not possible
        // a.mget( ) // _____
    a.h=4; // direct access to public member is possible
    a.geth( );

}

```

## **OUTPUT**

**In geth () :**

**In pget () enter value of p :7**

**In mget () enter value of m :4**

**p = 7 h = 4 m = 4**

**Explanation:** In the above program, the class access is declared with private, protected and public sections. Each section holds one integer variable and one member function. The object cannot directly access the data variable and member function of the private and protected sections. The object can access only the public section of the class and through public section it can access the private or protected sections. Here, the object a accesses the public variable h and initializes it with 4 whereas the private and protected variables are accessed using member functions. The function getp( ) and getm( ) are invoked by the public member function geth( ). The geth( ) also displays the contents of the variables on the screen.

## 6.14 CLASSES, OBJECTS AND MEMORY

Objects are the identifiers declared for class data type. Object is a composition of one or more variables declared inside the class. Each object has its own copy of public and private data members. An object can have access to its own copy of data members and have no access to data members of other objects.

Only declaration of a class does not allocate memory to the class data members. When an object is declared memory is reserved for data members only and not for member functions.

Consider the following program.

### 6.10 Write a program to declare objects and display their contents.

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

class month
{
public:
char *name;
int days;
};

// end of class

int main ( )
{
clrscr( );
month M1,M3;           // object declaration
M1.name="January";
M1.days=31;
M3.name="March";
M3.days=31;

cout <<"\n Object M1 ";
cout <<"\n Month name : "<<M1.name <<" Address
:<<(unsigned)&M1.name;
cout <<"\n Days :" <<M1.days <<"\t\t Address : "<<(unsigned)&M1.days;
```

```

cout <<"\n\n Object M3 ";
cout <<"\n Month name : "<<M3.name <<"\t Address :
    "<<(unsigned)&M3.name;
cout <<"\n Days :" <<M3.days <<"\t\t Address : "<<(unsigned)&M3.days;

return 0;
}

```

## OUTPUT

### Object M1

**Month name : January Address :65522**

**Days : 31 Address : 65524**

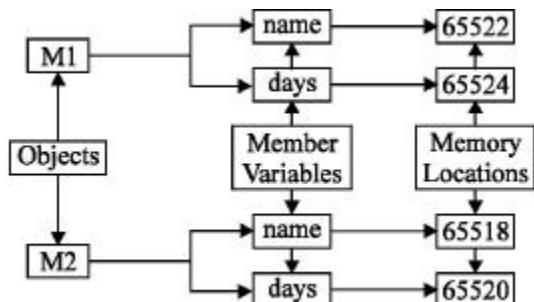
### Object M3

**Month name : March Address : 65518**

**Days : 31 Address : 65520**

**Explanation:** M1 and M3 are objects of class month. Separate memory is allocated to each object. The contents and address of the member variables are displayed in the output. [Figure 6.4](#) shows it more clearly.

From the last program it is clear that memory is allocated to data members. What about functions? Member functions are created and memory is allocated to them only once when a class is declared. All objects of a class access the same memory location where member functions are stored. Hence, separate copies of member functions are not present in every object like member variables. The following program and [Figure 6.5](#) illustrates this.



**Fig.6.4** Memory occupied by objects

## 6.11 Write a program to display the size of the objects.

```

# include <iostream.h>
# include <constream.h>

class data
{
    long i;      // By default private
    float f;
    char c;
};

int main( )

```

```

{
    clrscr( );
    data d1,d2;
    cout <<endl<<" Size of object d1 = "<<sizeof(d1);
    cout <<endl<<" Size of object d2 = "<<sizeof(d2);
    cout <<endl<<" Size of class           ="<<sizeof(data);
    return 0;
}

```

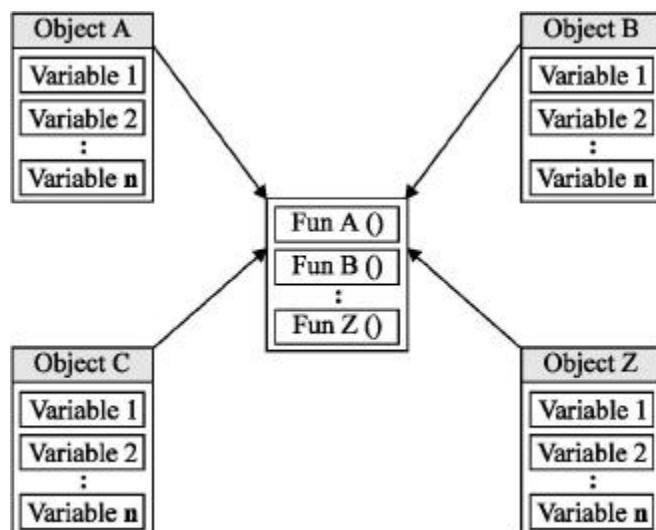
## OUTPUT

**Size of object d1 = 9**

**Size of object d2 = 9**

**Size of class = 9**

**Explanation:** In the above program, the class data has three member variables of long, float and char type. d1 and d2 are objects of the class data. The sizeof( ) operator displays the size of objects. The size of any object is equal to sum of sizes of all the data members of the class. In the class data, the data type long occupies 4 bytes, float occupies 4 bytes and char occupies 1 byte. Their sum is 9 that is the size of an individual object.



**Fig.6.5** Data members and member functions in memory

The member functions are not considered in the size of the object. All the objects of a class use the same member functions. Only one copy of member function is created and stored in the memory whereas each object has its own set of data members.

## 6.15 STATIC MEMBER VARIABLES AND FUNCTIONS

### (1) STATIC MEMBER VARIABLES

We have noticed earlier that each object has its separate set of data member variables in memory. The member functions are created only once and all objects share the functions. No separate copy of function of each object is created in the memory like data member variables.

It is possible to create common member variables like function using the `static` keyword. Once a data member variable is declared as `static`, only one copy of that member is created for the whole class. The `static` is a keyword used to preserve value of a variable. When a variable is declared as `static` it is initialized to zero. A static function or data element is only recognized inside the scope of the present class.

In the earlier version of Turbo C++, it was not necessary to define static data members explicitly. It was linkers' responsibility to find undefined static data. The linker would implicitly define the static data and allocate them required memory without showing error message. In the new versions of Turbo C++, it is necessary to explicitly define static members.

#### **SYNTAX:**

```
Static <variable definition> ;  
Static <function definition>;
```

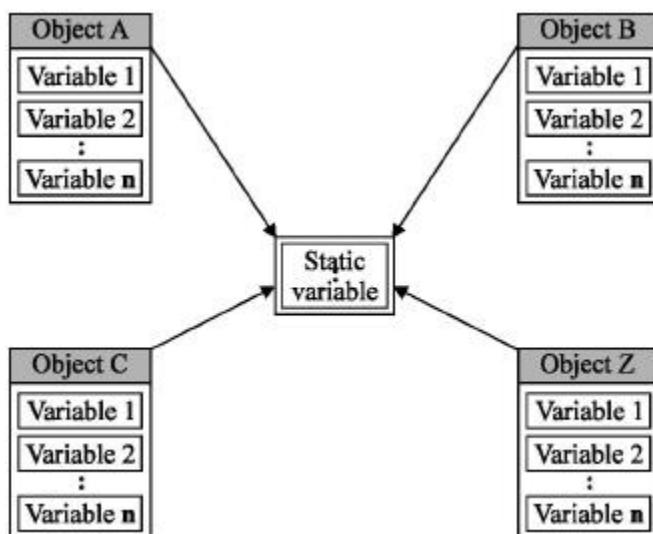
If a local variable is declared with `static` keyword, it preserves the last value of the variable. A static data item is helpful when all the objects of the class share a common data. The static data variable is accessible within the class, but its value remains in the memory throughout the whole program. [Figure 6.6](#) shows static members in memory.

#### **EXAMPLES:**

```
Static int c;  
Static void display ( ) {}  
(a) int sumnum :: c=0;
```

The class and scope of the static member variable is defined outside the class declaration as shown in statement **(a)**. The reasons are:

- (1) The static data members are associated with the class and not with any object.
- (2) The static data members are stored individually rather than an element of an object.
- (3) The static data member variable must be initialized otherwise the linker will generate an error.
- (4) The memory for static data is allocated only once.
- (5) Only one copy of static member variable is created for the whole class for any number of objects. All the objects have common static data member.



**Fig.6.6** Static members in memory

**6.12 Write a program to declare static data member. Display the value of static data member.**

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

class number
{
    static int c;

public:

void count ( )
{
    ++c;
    cout <<"\n c="<<c;
}

};

int number :: c=0; // initialization of static member variable

int main( )
{
    number a,b,c;
    clrscr( );
    a.count( );
    b.count( );
    c.count( );
    return 0;
}
```

**OUTPUT**

c=1  
c=2  
c=3

**Explanation:** In the above program, the class `number` has one static data variable `c`. The `count()` is a member functions increment value of static member variable `c` by one when called. The statement `int number :: c=0` initializes the static member with 0. It is also possible to initialize the static data members with other values. In the function `main()`, `a`, `b` and `c` are three objects of class `number`. Each object calls the function `count()`. At each call to the function `count()` the variable `c` gets incremented and the `cout` statement displays the value of variable `c`. The objects `a`, `b` and `c` share the same copy of static data member `c`.

**6.13 Write a program to show difference between static and non-static member variables.**

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

```

# include <conio.h>

class number
{
    static int c;           // static variable
    int k;                 // non-static variable

public:

void zero( )
{
    k=0;
}

void count( )
{
    ++c;
    ++k;

    cout <<"\n Value of c = "<<c <<" Address of c = "<<(unsigned)&c;
    cout <<"\n Value of k = "<<k <<" Address of k = "<<(unsigned)&k;
}

};

int number :: c=0; // initialization of static member variable

int main( )
{
    number A,B,C;

    clrscr( );

    A.zero( );
    B.zero( );
    C.zero( );

    A.count( );
    B.count( );
    C.count( );

    return 0;
}

```

## **OUTPUT**

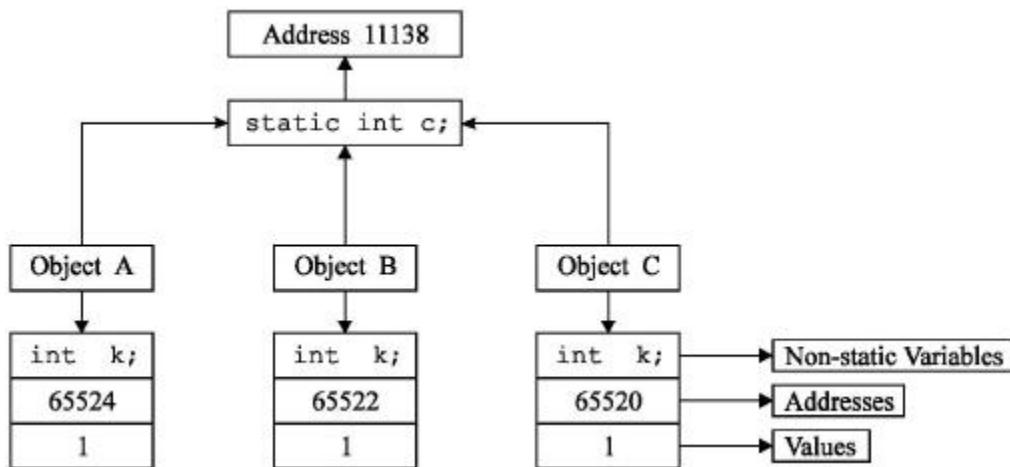
**Value of c = 1 Address of c = 11138**  
**Value of k = 1 Address of k = 65524**  
**Value of c = 2 Address of c = 11138**  
**Value of k = 1 Address of k = 65522**  
**Value of c = 3 Address of c = 11138**  
**Value of k = 1 Address of k = 65520**

**Explanation:** This program compares between static and non-static member variables. The class number has two-member variables c and k. The variable c is declared as static and k

is a normal variable. The function `zero( )` is used to initialize the variable `k` with zero. The static member variable `c` is initialized with zero as shown below:

```
int number :: c=0; // initialization of static member variable
```

The function `count( )` is used to increment values of `c` and `k`. In function `main( )`, `A`, `B` and `C` are objects of class `number`. The function `zero( )` is invoked three times by object `A`, `B` and `C`. Each object has its own copy of variable `k` and hence, each object invokes the function `zero( )` to initialize its copy of `k`. The static member variable `c` is common among the objects `A`, `B` and `C`. Hence, it is initialized only once. [Figure 6.7](#) shows the object and member variables in memory.



**Fig.6.7** Static and non-static members

#### 6.14 Write a program to enter a number. Count the total number of digits from 0 to 9 occurring from 1 to entered number.

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

class digit
{
static int num[10];
public :

void check(int n);
void show( );
void input( );
void ini( );
};

int digit :: num[]={0,0,0,0,0,0,0,0,0,0};

void digit :: show( )
{
    for (int j=0;j<10;j++)
    {
```

```

        if (num[j]==0) continue;
        cout <<"\n  Number "<<j <<" occurs " <<num[j] <<" times";
    }

void digit :: ini( )
{
for (int k=0;k<10;k++)
num[k]=0;
}

void digit :: input( )
{
int x,y;
cout <<endl<<"\n Enter a Number : ";
cin >>y;
check(y);
}

void digit :: check (int u)
{
int m;

while (u!=0)
{
    m=u%10;
    num[m]++;
    u=u/10;
}
}

void main( )
{
clrscr( );
digit d;

//d.ini( );
d.input( );
d.show( );
}

```

## **OUTPUT**

**Enter a Number : 22151**  
**Number 1 occurs 2 times**  
**Number 2 occurs 2 times**  
**Number 5 occurs 1 times**

**Explanation:** In the above program, the class `digit` is declared with one static array member `num [10]` and four member functions `check( )`, `show( )`, `input( )` and `ini( )`. The function `input( )` reads an integer through the keyboard. The entered number is passed to function `check( )`. The function `check( )` is invoked by function `input( )`. The function `check( )` separates individual digits of the entered number using repetitive modular division and division operation. The separated digits are

counted and the count value is stored in the array num[10] according to the element number. The function `show( )` displays the contents of array `num[]`. The function `ini( )` is declared and when called, initializes all array elements with zero. In case the array is not declared as static this function is useful. Here, in this program the array is static, hence we initialize it with the statement `int digit:: num[]={0,0,0,0,0,0,0,0,0,0}`. If this statement is removed, we need to call the function `ini( )`.

## (2) STATIC MEMBER FUNCTIONS

Like member variables, functions can also be declared as static. When a function is defined as static, it can access only static member variables and functions of the same class. The non-static members are not available to these functions. The static member function declared in public section can be invoked using its class name without using its objects. The `static` keyword makes the function free from the individual object of the class and its scope is global in the class without creating any side effect for other part of the program. The programmer must follow the following points while declaring static function:

- (1) Just one copy of static member is created in the memory for entire class. All objects of the class share the same copy of static member.
- (2) Static member functions can access only static data members or functions.
- (3) Static member functions can be invoked using class name.
- (4) It is also possible to invoke static member functions using objects.
- (5) When one of the objects changes the value of data member variables, the effect is visible to all the objects of the class.

### 6.15 Write a program to declare static member functions and call them from the `main( )` function.

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

class bita
{
private :
    static int c;
public :

    static void count( ) { c++; }

    static void display( )
    {
        cout <<"\nValue of c : "<<c;
    }

};

int bita ::c=0;

void main( )
```

```

{
    clrscr( );
    bita:: display( );
    bita::count( );
    bita::count( );
    bita::display( );
}

```

### **OUTPUT:**

**Value of c : 0**

**Value of c : 2**

**Explanation:** In the above program, the member variable `c` and functions of class `bita` are static. The function `count()` when called, increases the value of static variable `c`. The function `display()` prints the current value of the variable `c`. The static function can be called using class name and scope access operator as per statements given next.

```

bita::count();      // invokes count() function
bita::display();   // invokes display() function

```

### **(3) STATIC PRIVATE MEMBER FUNCTION**

Static member function can also be declared in private section. The private static function must be invoked using static public function. The following program illustrates the point.

#### **6.16 Write a program to define private static member function and invoke it.**

```

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

class bita
{
    private :
        static int c;

    static void count( ) { c++; }

public:

    static void display( )
    {
        count();           // Call to private static member function
        cout << "\nValue of c : " << c;
    }
};

int bita ::c=0;

void main( )
{
    clrscr( );
    bita:: display( );
    bita::display( );
}

```

**OUTPUT**

**Value of c : 1**

**Value of c : 2**

**Explanation:** In the above program, `count( )` is a private static member function. The public static function `display( )` invokes the private static function `count( )`. The function `display( )` also displays the value of static variable `c`.

#### (4) STATIC PUBLIC MEMBER VARIABLE

The static public member variable can also be initialized in function `main( )` like other variables. The static member variable using class name and scope access operator can be accessed. The scope access operator is also used when variables of same name are declared in global and local scope. The following program illustrates this:

#### 6.17 Write a program to declare static public member variable, global and local variable with the same name. Initialize and display their contents.

```
# include <iostream.h>
# include <constream.h>
int c=11;      // global variable

class bita
{
public:
    static int c;
};

int bita ::c=22;    // class member variable

void main( )
{
    clrscr( );

    int c=33;      // local variable

cout<<"\nClass member    c = "<<bita::c;
cout <<"\nGlobal variable c = "<<::c;
cout  <<"\nLocal variable  c = "<<c;
}
```

#### OUTPUT

**Class member c = 22**

**Global variable c = 11**

**Local variable c = 33**

**Explanation:** In the above program, the variable `c` is declared and initialized in three different scopes such as global, local and inside the class. The variable `c` declared inside is static variable and initialized to 22. The global variable `c` is initialized to 11 and local variable `c` is initialized to 33.

**Static Member Variable** The value of static variable is displayed using variable name preceded by class name and scope access operator as shown in the statement `cout<<"\nClass member c = "<<bita::c;`.

**Global Variable** The global variable can be accessed using variable name preceded by scope access operator as shown in the statement `cout << "\nGlobal variable c = "<<::c;`.

**Local Variable** The local variable can be accessed only by putting its name as shown in the statement `cout << "\nLocal variable c = "<<c;`.

## 6.16 STATIC OBJECT

In C, it is common to declare variable static that gets initialized to zero. The object is a composition of one or more member variables. There is a mechanism called constructor to initialize member variables of the object to desired values. The constructors are explained in next chapter. The keyword `static` can be used to initialize all class data member variables to zero. Declaring object itself as static can do this. Thus, all its associated members get initialized to zero. The following program illustrates the working of static object.

### 6.18 Write a program to declare static object. Display its contents.

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

class bita
{
    private:
        int c;
        int k;

    public :

        void plus( )
        {
            c+=2;
            k+=2;
        }

        void show( )
        {
            cout << " c= "<<c<<"\n";
            cout << " k= "<<k;
        }
};

void main( )
{
    clrscr( );
    static bita A;
```

```
    A.plus( );
    A.show( );
}
```

## OUTPUT

c= 2

k= 2

**Explanation:** The class `b1ta` has two-member variables `c` and `k` and two member functions `plus()` and `show()`. In function `main()`, the object `A` is declared. It is also declared as static. The data member of object `A` gets initialized to zero. The function `plus()` is invoked, which adds two to the value of `c` and `k`. The function displays value of `c` and `k`. Declaring objects static, does not mean that the entire class is static including member function. The declaration of static object removes garbage of its data members and initializes them to zero.

## 6.17 ARRAY OF OBJECTS

Arrays are collection of similar data types. Arrays can be of any data type including user-defined data type, created by using `struct`, `class` and `typedef` declarations. We can also create an array of objects. The array elements are stored in continuous memory locations as shown in [Figure 6.8](#). Consider the following example:

```
class player
{
private:
    char name [20];
    int age;

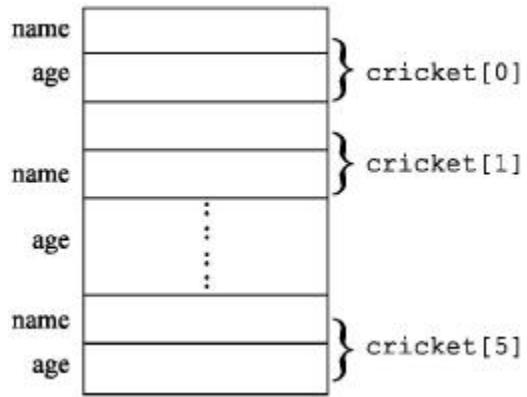
public:

    void input (void);
    void display (void);
};
```

In the example given above `player` is a user-defined data type and can be used to declare an array of object of type `player`. Each object of an array has its own set of data variables.

```
player cricket[5];
player football[5];
player hockey[5];
```

As shown above, arrays of objects of type `player` are created. The array `cricket[5]` contains name, age and information for five objects. The next two declarations can maintain the same information for other players in arrays `hockey[5]` and `football[5]`. These arrays can be initialized or accessed like an ordinary array. The program given below describes the working of array of objects.



**Fig.6.8** Array of objects

### 6.19 Write a program to declare array of objects. Initialize and display the contents of arrays.

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

class player
{
    private:
        char name [20];
        int age;
    public:
        void input (void);
        void display (void);
};

void player :: input ( )
{
    cout <<"\n Enter Player name : ";
    cin>>name;
    cout <<" Age : ";
    cin>>age;
}

void player :: display ( )
{
    cout <<"\n Player name : "<<name;
    cout <<"\n Age          : "<<age;
}

int main( )
{
    clrscr( );

    player cricket[3]; // array of objects
```

```

cout <<"\n Enter Name and age of 3 players ";
for (int i=0;i<3;i++)
    cricket[i].input ( );

for (i=0;i<3;i++)
    cricket[i].display ( );

return 0;
}

```

## **OUTPUT**

**Enter Name and age of 3 players**

**Enter Player name : Sachin**

**Age : 29**

**Enter Player name : Rahul**

**Age : 28**

**Enter Player name : Saurav**

**Age : 30**

**Player name : Sachin**

**Age : 29**

**Player name : Rahul**

**Age : 28**

**Player name : Saurav**

**Age : 30**

**Explanation:** In the above program, the member function `input( )` reads information of players. The `display( )` function displays information on the screen. In function `main( )`, the statement `player cricket[3];` creates an array `cricket[3]` of three objects of type `player`. The `for` loops are used to invoke member functions `input( )` and `display( )`, using array of objects.

## **6.18 OBJECTS AS FUNCTION ARGUMENTS**

Similar to variables, objects can be passed on to functions. There are three methods to pass an argument to a function as given below:

(a) Pass-by-value—In this type a copy of an object (actual object) is sent to function and assigned to object of callee function (Formal object). Both actual and formal copies of objects are stored at different memory locations. Hence, changes made in formal objects are not reflected to actual objects.

(b) Pass-by-reference — Address of object is implicitly sent to function.

(c) Pass-by-address — Address of the object is explicitly sent to function.

In pass by reference and address methods, an address of actual object is passed to the function. The formal argument is reference pointer to the actual object. Hence, changes made in the object are reflected to actual object. These two methods are useful because an address is passed to the function and duplicating of object is prevented.

The examples given below illustrate both methods of passing objects to the function as an argument.

### 6.20 Write a program to pass objects to the function by pass-by-value method.

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

class life
{
    int mfgyr;
    int expyr;
    int yr;
public :

    void getyrs( )
    {
        cout <<"\nManufacture Year : ";
        cin  >>mfgyr;
        cout <<"\n Expiry Year      : ";
        cin  >>expyr;
    }

    void period ( life);
};

void life :: period (life y1)
{
    yr=y1.expyr-y1.mfgyr;
    cout <<"Life of the product   : " <<yr <<" Years";
}

void main( )
{
    clrscr( );
    life a1;
    a1.getyrs( );
    a1.period(a1);
}
```

#### OUTPUT:

Manufacture Year : 1999

Expiry Year : 2002

Life of the product : 3 Years

**Explanation:** In the above program, the class `life` is declared with three member integer variables. The function `getyrs( )` reads the integers through the keyboard. The function `period( )` calculates the difference between the two integers entered. In function `main( )`, `a1` is an object to the class `life`. The object `a1` calls the function `getyrs( )`. Immediately after this, the same object (`a1`) is passed to the function `period( )`. The function `period( )` calculates the difference between two integers (dates) using the two data members of the same class. Thus, an object can be passed to the

function. To pass an object by reference, the prototype of function `period( )` should be as follows:

```
void period( life &);
```

### 6.21 Write a program to pass objects to the function pass-by-address method.

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

class life
{
    int mfgyr;
    int expyr;
    int yr;
public :

void getyrs( )
{
    cout <<"\nManufacture Year : ";
    cin  >>mfgyr;
    cout <<"\n Expiry Year      : ";
    cin  >>expyr;
}

    void period ( life* );
};

void life :: period (life *y1)
{
    yr=y1->expyr-y1->mfgyr;
    cout <<"Life of the product   : " <<yr;
}

void main( )
{
    clrscr( );
    life a1;
    a1.getyrs( );
    a1.period(&a1);
}
```

#### OUTPUT:

Manufacture Year : 1999

Expiry Year : 2002

Life of the product : 3 Years

**Explanation:** The above program is same as the previous one. In this program, the object `a1` is passed by address. Consider the following statements:

- (a) `void period ( life* );`
- (b) `a1.period (&a1);`
- (c) `Yr=y1->expyr-y1->mfgyr;`

The statement **(a)** is the prototype of function `period( )`. In this statement, the deference operator (\*) indicates that the function will accept address of the actual argument. The statement **(b)** is used to pass the address of the argument to the function `period( )`. The statement **(c)** is used to access the member variables of the

class. When an object is a pointer to the class members, then its elements are accessed by using `->` (arrow) operator. In such case use of `dot` operator(.) is invalid.

## 6.19 FRIEND FUNCTIONS

The central idea of encapsulation and data hiding concept is that any non-member function has no access permission to the private data of the class. The private members of the class are accessed only from member functions of that class. Any non-member function cannot access the private data of the class.

C++ allows a mechanism, in which a non-member function has access permission to the private members of the class. This can be done by declaring a non-member function `friend` to the class whose private data is to be accessed. Here `friend` is a keyword. Consider the following example:

```
class ac
{ private:
    char name [15];
    int acno;
    float bal;
public:
    void read( );
friend void showbal( );
};
```

The keyword `friend` must precede the function declaration whereas function declarator must not. The function can be defined at any place in the program like normal function. The function can be declared as friend function in one or more classes. The keyword `friend` or scope access operator must not precede the definition of `friend` function. The declaration of friend function is done inside the class in private or public part and a function can be declared as friend function in any number of classes. These functions use objects as arguments. Let us see the statement given below.

```
(a) friend void :: showbal (ac a) // Wrong function definition
{
    statement1;
    statement2;
}
```

The above declaration of function is wrong because the function declarator precedes the keyword `friend`:

The friend functions have the following properties:

- (a) There is no scope restriction for the friend function; hence they can be called directly without using objects.
- (b) Unlike member functions of class, friend function cannot access the member directly. On the other hand, it uses `object` and `dot` operator to access the private and public member variables of the class.
- (c) By default, friendship is not shared (mutual). For example, if class X is declared as friend of Y, this does not mean that Y has privileges to access private members of class X.
- (d) Use of friend functions is rarely done, because it violates the rule of encapsulation and data hiding.
- (e) The function can be declared in public or private sections without changing its meaning.

**6.22 Write a program to access private data using non-member function. Use friend function.**

```

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

class ac
{
    private:
        char name[15];
        int acno;
        float bal;

public:

void read( )
{
    cout <<"\nName      :" ;
    cin >>name;
    cout <<"\nA/c No. :" ;
    cin >>acno;
    cout <<"\n Balance   :" ;
    cin >>bal;
}

friend void showbal(ac ); // friend function declaration
};

void showbal (ac a)
{
    cout <<"\n Balance of A/c no. " <<a.acno <<" is Rs." <<a.bal;
}

int main( )
{
    ac k;
    k.read( );
    showbal(k); // call to friend function
    return 0;
}

```

#### **OUTPUT:**

**Name :Manoj**

**A/c No. :474**

**Balance :40000**

**Balance of A/c no. 474 is Rs.40000**

**Explanation:** In the above program, class ac is declared. It has three member variables and one member function. Also, inside the class ac, showbal( ) is a function, which is declared as friend of the class ac. Once the outside function is declared as friend to any class, it gets an authority to access the private data of that class. The function read( ),

reads the data through the keyboard such as name, account number and balance.  
The friend function showbal( ) displays balance and acno.

**6.23 Write a program to declare friend function in two classes. Calculate the sum of integers of both the classes using friend sum( ) function.**

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

class first;

class second
{
    int s;

public :

void getvalue( )
{
    cout <<"\nEnter a  number : ";
    cin >>s;
}

friend void sum (second, first);

};

class first
{

int f;
public :

void getvalue( )
{
    cout <<"\nEnter a  number : " ;
    cin >>f;
}

friend void sum (second , first);
;

void sum (second d, first t)
{
    cout <<"\n Sum of two numbers : " <<t.f + d.s;
}

void main( )
{
```

```

        clrscr( );
        first a;
        second b;
        a.getvalue( );
        b.getvalue( );
        sum(b,a);
    }

```

**OUTPUT :**

Enter a number : 7

Enter a number : 8

**Sum of two numbers : 15**

**Explanation:** In the above program, two classes `first` and `second` are declared with one integer and one member function in each. The member function `getvalue( )` of both classes reads integers through the keyboard. In both the classes, the function `sum( )` is declared as friend. Hence, this function has an access to the members of both the classes. Using `sum( )` function, addition of integers is calculated and displayed.

**6.24 Write a program to exchange values between two classes. Use friend functions.**

```

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

class second;

class first
{
    int j;

public:
    void input( )
    {
        cout <<"Enter value of j : ";
        cin  >>j;
    }

    void show (void)
    {
        cout <<"\n Value of J = ";
        cout <<j <<"\n";
    }
    friend void change (first &, second &);
};

class second
{
    int k;
public :
    void input( )
    {
        cout <<"\nEnter value of k : ";
    }
}

```

```

    cin >>k ;
}

void show (void)
{
cout <<" Value of K = ";

cout <<k ;

}
friend void change (first & , second &);

};

void change ( first &x, second &y)
{
    int tmp=x.j;
    x.j=y.k;
    y.k=tmp;
}

main( )
{
    clrscr( );
    first c1;
    second c2;
    c1.input( );
    c2.input( );
    change (c1,c2);
    cout <<"\nAfter change values are" <<"\n";
    c1.show( );
    c2.show( );
    return 0;
}

```

### **OUTPUT:**

**Enter value of j : 4**

**Enter value of k : 8**

**After change values are**

**Value of J = 8**

**Value of K = 4**

**Explanation:** In the above program, two classes `first` and `second` are defined. Each class contains one-integer variable and two member functions. The function `input( )` is used to read an integer through the keyboard. The function `show( )` is used to display the integer on the screen. The function `change( )` is declared as friend function for both the classes. Passing values by reference of member variables of both the classes, values are exchanged.

**6.25 Write a program to declare three classes. Declare integer array as data member in each class. Perform addition of two data member arrays into array of third class.**

**Use friend function.**

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

```

class B;
class C;

class A
{
    int a[5];
public :
void input( );
friend C sum (A,B,C);
};

void A :: input( )
{
    int k;
    cout <<"\n Enter five integers : ";
    for (k=0;k<5;k++)
        cin>>a[k];
}

class B
{
    int b[5];
public :
void input( );
friend C sum (A,B,C);
};

void B:: input( )
{
    int k;
    cout <<"\n Enter five integers : ";
    for (k=0;k<5;k++)
        cin>>b[k];
}

class C
{
    int c[5];
public :
void show( );
friend C sum (A,B,C);
};

void C :: show( )
{
    cout <<"\n\t Addition      : ";
    for (int k=0;k<5;k++)
        cout <<" " <<c[k];
}

C sum (A a1 ,B b1, C c1)
{
    for (int k=0;k<5;k++)
        c1.c[k]=a1.a[k]+b1.b[k];
    return c1;
}

void main( )
{
    clrscr( );
}

```

```

A a;
B b;
C c;

a.input( );
b.input( );
c=sum(a,b,c);
c.show( );
}

```

## OUTPUT

Enter five integers : 5 4 8 7 5

Enter five integers : 2 4 1 2 3

Addition : 7 8 9 9 8

**Explanation:** In the above program, three classes A, B and C are declared. Each class contains single integer array as data member. They are a [5], b [5] and c [5] respectively. Classes A and B contain member function input ( ) to read integers. The function sum ( ) is declared as friend in all the three classes. This function performs addition of arrays of class A and B and stores results in the array of class C. The result obtained is returned in main ( ) where the return value is assigned to object c. In main ( ) a, b and c are objects of classes A, B and C respectively. The member function show ( ) of class C displays the contents of object c.

## FRIEND CLASSES

It is possible to declare one or more functions as friend functions or an entire class can also be declared as friend class. When all the functions need to access another class in such a situation we can declare an entire class as friend class. The friend is not transferable or inheritable from one class to another. Declaring class A to be a friend of class B does not mean that class B is also a friend of class A i.e., friendship is not exchangeable. The friend classes are applicable when we want to make available private data of a class to another class.

### 6.26 Write a program to declare friend classes and access the private data.

```

#include <iostream.h>
#include <constream.h>
class B;

class A
{
    private :
    int a;
    public :
    void asset( ) {a=30;}
    void show (B);
};

class B
{
    private :
    int b;

```

```

public :
void b_set( ) { b=40 ; };
friend void A :: show (B bb);
};

void A :: show (B b)

{
    cout <<"\n a = "<<a;
    cout <<"\n b = "<<b.b;
}

void main( )
{
    clrscr( );
    A a1;
    a1.aset( );
    B b1;
    b1.bset( );
    a1.show(b1);
}

```

### **OUTPUT**

**a = 30**

**b = 40**

### **OUTPUT**

**a = 30**

**b = 40**

**Explanation:** In the above program, two classes A and B are declared. Here, class A is a friend of class B. The member function of class A can access the data of class B. Thus, the show( ) function displays the values of data members of both the classes.

## **6.20 THE CONST MEMBER FUNCTION**

The member functions of a class can also be declared as constant using `const` keyword. The constant functions cannot modify any data in the class. The `const` keyword is suffixed to the function prototype as well as in function definition. If these functions attempt to change the data, compiler will generate an error message.

### **6.27 Write a program to declare `const` member function and attempt any operation within it.**

```

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

class A

{
    int c;
    public :

```

```

void add (int a,int b) const
{
// c=a+b;           // invalid statement
a+b;
cout <<"a+b = "<<_AX ;
}

int main( )
{
    clrscr( );
    A a;
    a .add(5,7);
    return 0;
}

```

### **OUTPUT**

**a+b = 12**

**Explanation:** In the above program, class A is declared with one member variable c and one constant member function add( ). The add( ) function is invoked with two integers. The constant member function cannot perform any operation. Hence, the expression c=a+b will generate an error. The expression a+b is valid and cannot alter any value. The result obtained from the equation a+b is displayed using CPU register.

## **6.21 RECURSIVE MEMBER FUNCTION**

Like C, C++ language also supports recursive features i.e., function is called repetitively by itself. The recursion can be used directly or indirectly. The direct recursion function calls itself till the condition is true. In indirect recursion, a function calls another function and then the called function calls the calling function. The recursion with member function is illustrated in the following program.

**6.28 Write a program to calculate triangular number by creating a member function. Call it recursively.**

```

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

class  num

{
public :

tri_num(int m)
{ int f=0;
    if (m==0)
        return(f);
    else
        f=f+m+tri_num(m-1);
    return (f);
}

};


```

```

void main( )
{
    clrscr( );
num a;
int x;
cout <<"\n Enter a number : ";
cin>>x;
cout<<"Triangular number of "<<x<<" is : "<<a.tri_num(x);
}

```

## OUTPUT

Enter a number : 5

Triangular number of 5 is : 15

**Explanation:** In the above program, the class num is declared with one member function tri\_num( ). This function is used to calculate the triangular number of the entered number. The triangular number is nothing but sum from 1 to that number. In function main( ), a number is read through the keyboard and it is passed to function tri\_num( ) which is invoked by object a of class num. The function tri\_num( ) is invoked and tri\_num( ) invokes itself repetitively till the value of m becomes 0. The variable f holds the cumulative total of successive numbers and return( ) statement returns value of f in function main( ), where it displays triangular number on the screen.

## 6.22 LOCAL CLASSES

When a class is declared inside a function they are called as local classes. The local classes have access permission to global variables as well as static variables. The global variables need to be accessed using scope access operator when the class itself contains member variable with same name as global variable. The local classes should not have static data member and static member functions. If at all they are declared, the compiler provides an error message. The programs given next illustrate the local classes.

### 6.29 Write a program to define classes inside and outside main( ) function and access the elements.

```

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

class A
{
private :
    int a;
public :

    void get( )
    {
        cout <<"\n Enter value for a : ";
        cin >>a;
    }

    void show( )
    {   cout <<endl<<" a = " <<a; }
};


```

```

main( )
{
    clrscr( );

    class B
    {
        int b;
        public :

        void get( )
        {
            cout <<"\n Enter value for b : ";
            cin >>b;
        }

        void show( )
        {
            cout <<" b = " <<b;
        }
    };

    A j;
    B k;

    j.get( );
    k.get( );
    j.show( );
    k.show( );
    return 0;
}

```

### **OUTPUT**

**Enter value for a : 8**

**Enter value for b : 9**

**a = 8 b = 9**

**Explanation:** In the above program, class A is declared before main( ) function as usual. Class B is declared inside the main( ) function. Both the functions have two member functions, get( ) and show( ). The get( ) function reads integers through the keyboard. The show( ) function displays the values of data members on the screen.

### **6.30 Write a program to declare global variables, read and display data using member functions.**

```

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

int j, k, l, m; // global variable

```

```

class A
{
    private :
    int a;
    int j;
    public :

        void get( )

        {
            cout <<"\n Enter value for a,j,j and k : ";
            cin >>a >>j>>::j >>k;
        }

        void show ( )
        {
            cout <<endl<<"a= " <<a<<" j=" <<j <<"::j=" <<::j <<"k=" <<k ;
        }
};

int main( )
{
    clrscr( );

    class B
    {
        int b;
        int l;
        public :

            void get( )
            {
                cout <<"\n Enter value for b,l,l and m : ";
                cin >>b >>l>>::l>>m;
            }

            void show( )
            {
                cout <<"\n b = " <<b <<" l = " <<l <<" ::l = " <<::l <<" m = " <<m;
            }
    };

    A x;
    B y;

    x.get( );
    y.get( );
    x.show( );
    y.show( );

    return 0;
}

```

```
}
```

**OUTPUT**

Enter value for a,j,j and k : 1 2 3 4

Enter value for b,l,l and m : 5 6 4 3

a = 1 j = 2 ::j = 3 k = 4  
b = 5 l = 6 ::l = 4 m = 3

**Explanation:** The above program is same as the previous one. In addition, in this program global variables *j*, *k*, *l*, and *m* are declared. The member functions *get()* and *show()* read and display values of member variables as well as global variables. Here, both the classes contain a single data member variable with same name as global variables. Thus, to access the global variable where ever necessary, scope access operator is used.

## 6.23 EMPTY, STATIC AND CONST CLASSES

The classes without any data members or member functions are called as empty classes. These types of classes are not frequently used. The empty classes are useful in exception handling. The syntax of empty class is as follows:

### Empty Classes

```
class nodata { };  
class vacant { };
```

We can also precede the declaration of classes by the keywords *static*, *const*, *volatile* etc. But there is no effect in the class operations. Such declaration can be done as follows:

### Classes and other keywords

```
static class boys { };  
const class data { };  
volatile class area{ };
```

## 6.24 MEMBER FUNCTIONS AND NON-MEMBER FUNCTIONS

So far we have used non-member function *main()* for declaring objects and calling member functions. Instead of *main()* other non-member functions can also be used. The member function can also invoke non-member function and vice versa. When a member function calls to non-member function, it is necessary to put its prototype inside the calling function or at the beginning of the program. It is a better practice to put prototype at the beginning of the program. It is also possible to put definition of the non- member function before class declaration. This method allows member function to invoke outside non- member function without the need of prototype declaration. But this approach creates problem when an outside non-member function attempts to invoke member function. We know that member functions can be called using object of that class. If a non-member function is defined before class declaration, it is not possible to create object in that function. Hence, the best choice is to put prototype of the non- member function at the beginning of the program that makes easy for both non-member functions and member functions to call each other. The following program explains practically whatever we learned about member function and non- member function in this section.

### 6.31 Write a program to call a member function using non-member function.

```

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

void moon(void); // Function prototype declaration

class mem
{
public :

    void earth ( ) { cout <<"On earth"; }

};

void main ( )
{
    clrscr( );
    mem k;
    moon( );
}

void moon( )
{
    mem j;
    j. earth( );
    cout <<endl<<"On moon   ";
}

```

## **OUTPUT**

**On earth**

**On moon**

**Explanation:** In the above program, `moon()` is a non-member function and its prototype is declared at the beginning of the program. The function `main()` calls the function `moon()`. In function `moon()`, object `j` of type class `mem` is declared and a member function `earth()` is invoked. Thus, non-member function calls the member function.

## **6.25 THE MAIN( ) AS A MEMBER FUNCTION**

We know that the function `main()` is the starting execution point of every C/C++ program. The `main()` can be used as a member function of the class. But the execution of program will not start from this member function. The compiler treats member function `main()` and the user-defined `main()` differently. No ambiguity is observed while calling a function. The following program narrates this concept:

### **6.32 Write a program to make `main()` as a member function.**

```

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

class A
{

```

```

public:

void main( )
{
    cout << endl << "In member function main( )";
}
};

void main( )
{
    clrscr( );
    A *a;
    a->main( );
}

```

## OUTPUT

**In member function main( )**

**Explanation:** In the above program, class A is declared and has one member function main( ). In the non-member function main( ), the object a invokes the member function main( ) and a message is displayed as shown in the output.

## 6.26 OVERLOADING MEMBER FUNCTIONS

Member functions are also overloaded in the same fashion as other ordinary functions. We learned that overloading is nothing but one function is defined with multiple definitions with same function name in the same scope. The program given below explains the overloaded member function.

### 6.33 Write a program to overload member function of a class.

```

# include <iostream.h>
# include <stdlib.h>
# include <math.h>
# include <conio.h>

class absv
{
public :
    int num(int);
    double num (double);
};

int absv:: num(int x)
{
    int ans;
    ans=abs(x);
    return (ans);
}

double absv :: num(double d)
{

```

```

        double ans;
        ans=fabs(d);
        return(ans);
    }

int main( )
{
    clrscr( );
    absv n;
    cout <<"\n Absolute value of -25 is "<<n.num(-25);
    cout <<"\n Absolute value of -25.1474 is "<<n.num(-25.1474);
    return 0;
}

```

## OUTPUT

**Absolute value of -25 is 25**

**Absolute value of -25.1474 is 25.1474**

**Explanation:** In the above program, the class `absv` has member function `num()`.

The `num()` function is overloaded for integer and double. In function `main()`, the object `n` invokes the member function `num()` with one value. The compiler invokes the overloaded function according to the value. The function returns the absolute value of the number.

## 6.27 OVERLOADING MAIN( ) FUNCTION

In the last two subtitles we learnt how to make `main()` as member function and how to overload member function. Like other member function, `main()` can be overloaded in the class body as a member function. The following program explains this concept.

**6.34 Write a program to declare main( ) as a member function and overload it.**

```

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

class A
{
public:

    void main(int i)
    {
        cout <<endl<<"In main (int) :"<<i;
    }

    void main (double f)
    {
        cout <<"\nIn main(double) :"<<f;
    }

    void main(char *s)
    {
        cout <<endl<<"In main (char ) : "<<s;
    }
}

```

```

};

void main( )
{
    clrscr( );
    A *a;
    a->main(5);

    a->main(5.2);
    a->main("C++");
}

```

## **OUTPUT**

**In main (int) :5**

**In main(double) :5.2**

**In main (char ) : C++**

**Explanation:** This program is same as the last one. Here, the `main( )` function is used as a member function and it is overloaded for integer, float and character.

It is not possible to overload the non-member `main( )` function, which is the source of C/C++ program and hence the following program will not be executed and displays the error message “Cannot overload ‘main’”.

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

```

void main( )
{ }
main (float x, int y)
{
    cout <<x<<y;
    return 0;
}
```

The `main( )` is the only function that cannot be overloaded.

## **6.28 THE MAIN ( ), MEMBER FUNCTION AND INDIRECT RECURSION**

When a function calls itself then this process is known as recursion or direct recursion. When two functions call each other repetitively, such type of recursion is known as indirect recursion. Consider the following program and explanation to understand the indirect recursion using OOP. The program without using OOP is also described for the sake of C programmers and those who are learning C++.

### **6.35 Write a program to call function `main( )` using indirect recursion. // Indirect Recursion Using OOP //**

```

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

int main (int);

class rec
```

```

{
    int j;
public:
    int f;

    rec (int k, int i)
    {
        clrscr( );
        cout<<"[ ";
        f=i;
        j=k;
    }

    ~rec ( )
    {   cout <<"\b\b] Factorial of number : "<<f ; }

void pass( )
{
    cout<<main (j--)<<" * ";
}

};

rec a(5,1);

main(int x)
{
    if (x==0)
        return 0;

    a.pass( );
    a.f=a.f*x;
    return x;
}

```

## **OUTPUT**

**[ 0 \* 1 \* 2 \* 3 \* 4 \* 5 ] Factorial of number : 120**

**Explanation:** In the above program, class `rec` is declared with constructor, destructor, member function `pass()` and two integer variables. The integer variable `j` is private and `f` is public. The function `main()` is defined with one integer argument. Usually, `main()` with arguments is used for the applications used on the dos prompt. In this program, `main()` is called recursively by member function `pass()`. When a function call is made, the value of member data variable is decreased first and then passed. Thus, the `main()` passes value to itself. In function `pass()`, `main()` function is invoked and its return value is displayed. The public data member is directly used and by applying multiplication operation, factorial of a number is calculated.

Generally, objects are declared inside the function `main()`. But in this program, function `main()` is used in recursion. Hence, if we put the object declaration statement

inside the `main( )`, in every call of `main( )` object is created and the program does not run properly. To avoid this, the object is declared before `main( )`.

Constructor is used to initialize data members as well as to clear the screen. Destructor is used to display the factorial value of the number. All the statements that we frequently put in `main( )` are written outside the `main( )`.

Before the class declaration, prototype of `main( )` is given and this is because the member functions don't know about `main( )` and the prototype declaration provides information about `main( )` to member function.

For C programmers the above program in C style is explained below.

### **6.36 Write a program to call function `main( )` using indirect recursion in C style. // Indirect Recursion in C style //**

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

int m=5;
int f=1;
int j;
main (int x)
{
    void pass (void);

    if (x==0)
    { clrscr();
        cout << endl << "Factorial of number =" << f;
        return 0;
    }
    f=f*x;
    pass();
}

return x;
}

void pass ( ) { main (m--); }
```

#### **OUTPUT**

#### **Factorial of number =120**

**Explanation:** The logic of the program is same as the last one. The user-defined function `pass( )` has only one job i.e., to invoke function `main( )`. The if conditions inside `main( )`, checks the value of variable `x`. If value of `x` is zero, the `if` block is executed that displays factorial of the number and terminates the program.

## **6.29 BIT FIELDS AND CLASSES**

Bit field provides exact amount of bits required for storage of values. If a variable value is 1 or 0, we need a single bit to store it. In the same way if the variable is expressed between 0 and 3, then two bits are sufficient for storing these values. Similarly if a variable assumes

values between 0 and 7 then three bits will be enough to hold the variable and so on. The number of bits required for a variable is specified by non-negative integer followed by colon.

To hold the information we use the variables. The variables occupy minimum one byte for `char` and two bytes for `integer`. Instead of using complete integer if bits are used, space of memory can be saved. For example, to know the information about vehicles, following information has to be stored in the memory.

- (1) PETROL VEHICLE
- (2) DIESEL VEHICLE
- (3) TWO-WHEELER VEHICLE
- (4) FOUR-WHEELER VEHICLE
- (5) OLD MODEL
- (6) NEW MODEL

In order to store the status of the above information, we may need two bits for type of fuel as to whether vehicle is of petrol or diesel type. Three bits for its type as to whether the vehicle is two or four-wheeler. Similarly, three bits for model of the vehicle. Total bits required for storing the information would be 8 bits i.e., 1 byte. It means that the total information can be packed into a single byte. Eventually bit fields are used for conserving the memory. The amount of memory saved by using bit fields will be substantial which is proved from the above example.

However, there are restrictions on bit fields when arrays are used. Arrays of bit fields are not permitted. Also the pointer cannot be used for addressing the bit field directly, although use of the member access operator (`->`) is acceptable. The unnamed bit fields could be used for padding as well as for alignment purposes.

- (1) Bits fields should have integral type. A pointer array type is now allowed.
- (2) Address of bit fields cannot be obtained using `&` operator.

The class for the above problem would be as follows:

```
class vehicle
{
    unsigned type: 3;
    unsigned fuel: 2;
    unsigned model: 3;
};
```

The colon(:)in the above declaration tells to the compiler that bit fields are used in the class and the number after it indicates how many bits are required to allot for the field. Simple program is illustrated below.

### 6.37 Write a program to use bit fields with classes and display the contents of the bit fields.

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

# define PETROL 1
# define DIESEL 2
# define TWO_WH 3
# define FOUR_WH 4
# define OLD 5
# define NEW 6
```

```

class vehicle
{
    private :
        unsigned type : 3;
        unsigned fuel : 2;
        unsigned model :3;

public :

    vehicle ( )
    {
        type=FOUR_WH;
        fuel=PETROL;
        model=NEW;
    }

    void show ( )
    {
        if (model==NEW)

            cout <<"\n      New Model ";
        else
            cout <<"\n      Old Model ";

        cout <<"\n Type of Vehicle : "<< type;
        cout <<"\n Fuel           : "<<fuel;
    }
};

void main( )
{
    clrscr( );

    vehicle v;
    cout<<" Size of Object   : "<<sizeof(v)<<endl;
    v.show( );
}

```

## **OUTPUT**

**Size of Object : 1**

**New Model**

**Type of Vehicle : 4**

**Fuel : 1**

**Explanation:** In the above program, using # define macros are declared. The information about the vehicle is indicated between integers 1 to 6. The class vehicle is declared with bit fields. The number of **bits** required for each member is initialized. As per the program, type of vehicle requires 3 bits, fuel requires 2 bits and model requires 3 bits. An object v is declared. The constructor initializes bit fields with data. The output of the

program displays integer value stored in the bit fields, which can be verified with macro definitions initialized at the beginning of the program.

### SUMMARY

- (1) A class in C++ is similar to structure in C. Using class or structure, a programmer can merge one or more dissimilar data types and a new custom data type can be created.
- (2) In C++, classes and structures contain member variables and member functions in their declarations with private and public access blocks that restrict the unauthorized use. The defined classes and structures further can be used as custom data type in the program to declare objects.
- (3) In C++ `private` and `public` are two new keywords. The `private` keyword is used to protect specified data and functions from illegal use whereas the `public` keyword allows access permission.
- (4) The member function can be defined as (a) private or public (b) inside the class or outside the class.
- (5) To access private data members of a class, member functions are used.
- (6) The difference between member function and normal function is that the normal can be invoked freely whereas the member function can be invoked only by using the object of the same class.
- (7) `static` is the keyword used to preserve value of a variable. When a variable is declared as static, it is initialized to zero. A static function or data element is only recognized inside the scope of the present class.
- (8) When a function is defined as static, it can access only static member variables and functions of the same class. The static member functions are called using its class name without using its objects.
- (9) The member functions of a class can also be declared as constant using `const` keyword. The constant functions cannot modify any data in the class.
- (10) An object of a class can be passed to the function as arguments like variables of other data type. When an object is passed by value then this method is called as pass by value whereas when reference of an object is passed to the function then this method is called as pass by reference.
- (11) When a class is declared inside the function such classes are called as local classes. The local classes have access permission to global variables as well as static variables. The local classes should not have static data members and functions.

### EXERCISES

#### [A] Answer the following questions.

- (1) Explain class and struct with their differences.
- (2) Which operators are used to access members?
- (3) Explain the uses of `private` and `public` keywords. How are they different from each other?
- (4) Explain features of member functions.
- (5) What are static member variables and functions?
- (6) How are static variables initialized? Explain with the statement.
- (7) What are `friend` functions and `friend` classes?
- (8) How are `static` functions and `friend` functions invoked?
- (9) What do you mean by `constant` function?

- (10) What are local classes?
- (11) What is recursion?
- (12) What are bit fields?
- (13) List the keywords terminated by colon with their use.
- (14) Can member functions be private?
- (15) What is the concept of data hiding? What are the advantages of its applications?
- (16) Is it possible to access private data members without using member function? If yes, explain the procedure with an example.
- (17) What are static objects?
- (18) What is the difference between object and variable?

**[B] Answer the following by selecting the appropriate option.**

- (1) The members of a class are by default
  - (a) private
  - (b) public
  - (c) protected
  - (d) none of the above
- (2) The members of struct are by default
  - (a) public
  - (b) private
  - (c) protected
  - (d) none of the above
- (3) The private data of any class is accessed by
  - (a) only public member function
  - (b) only private member function
  - (c) both (a) and (b)
  - (d) none of the above
- (4) When the class is declared inside the function, it is called as
  - (a) local class
  - (b) global class
  - (c) both (a) and (b)
  - (d) none of the above
- (5) A non-member function that can access the private data of class is known as
  - (a) friend function
  - (b) static function
  - (c) member function
  - (d) library function
- (6) Encapsulation means
  - (a) protecting data
  - (b) allowing global access
  - (c) data hiding
  - (d) both (a) and (c)
- (7) The size of object is equal to
  - (a) total size of member data variables
  - (b) total size of member functions
  - (c) both (a) and (b)
  - (d) none of the above

- (8) In the prototype `void sum( int &);` arguments are passed by  
(a) value  
(b) reference  
(c) address  
(d) none of these

**[C] Attempt the following programs.**

- (1) Write a program to declare a class with three integer public data variables. Initialize and display them.
- (2) Write a program to declare private data member variables and public member function. Read and display the values of data member variables.
- (3) Write a program to declare private data member and a function. Also declare public member function. Read and display the data using private function.
- (4) Write a program to declare three classes S1, S2, and S3. The classes have a private data member variable of character data type. Read strings for the classes S1 and S2. Concatenate the strings read and assign it to the data member variable of class S3.
- (5) Write a program to enter positive and negative numbers. Enter at least 10 numbers. Count the positive and negative numbers. Use classes and objects.
- (6) Write a program to declare class with private member variables. Declare member function as static. Read and display the values of member variables.
- (7) Write a program to declare a class `temp_ture` as given below. Declare array of 5 objects. Read and display the data using array.

```
class temp_ture
{
    private:
    char date[12], ct1[15], ct2[15], ct3[15], ct4[15];
    int temp [4]
}
```

- (8) Write a program to declare a class with two integers. Read values using member function. Pass the object to another member function. Display the differences between them.
- (9) Write a program to define three classes. Define a `friend` function. Read and display the data for three classes using common functions and `friend` functions.
- (10) Write a program to define a local class. Also define global variables with the same name as that of member variables of class. Read and display the data for global and member variables.
- (11) Write a program to generate fibonacci series using recursion with member function.
- (12) Write a program to overload member function and display `string`, `int` and `float` using overloaded function.
- (13) Write a program to calculate sum of digits of entered number using indirect recursion. Recursion should be in between `main()` and member function.
- (14) Write a program to display string in reverse using recursion.
- (15) Write a program to count number of vowels present in the entered string.
- (16) Write a program to find largest out of ten numbers.
- (17) Write a program to count numbers between 1 to 100, which are not divisible by 2, 3 and 5.

**[D] Trace the bugs in the following programs.**

```
(1)
class data
{
private ;
};

(2)
class data
{
private:
int x=20;
}

(3)
# include <iostream.h>
# include <conio.h>

class data
{ int x; };

void main( )
{ data A;
A.x=30; }

(4)
# include <iostream.h>
# include <conio.h>

class data
{ int x;
public:
void show (void);
};
void show ( ) { cout <<"\n In show ( )"; }

void main( )
{ data A;
A.show ( ); }
```

# Constructors and Destructors

- [7.1 Introduction](#)
- [7.2 Constructors and Destructors](#)
- [7.3 Characteristics of Constructors and Destructors](#)
- [7.4 Applications with Constructors](#)
- [7.5 Constructors with Arguments](#)
- [7.6 Overloading Constructors](#)
- [7.7 Constructors with Default Arguments](#)
- [7.8 Copy Constructors](#)
- [7.9 The const Objects](#)
- [7.10 Destructors](#)
- [7.11 Calling Constructors and Destructors](#)
- [7.12 Qualifier and Nested Classes](#)
- [7.13 Anonymous Objects](#)

- [7.14 Private Constructors and Destructors](#)
- [7.15 Dynamic Initialization using Constructors](#)
- [7.16 Dynamic Operators and Constructors](#)
- [7.17 The main\( \) as a Constructor and Destructor](#)
- [7.18 Recursive Constructor](#)
- [7.19 Program Execution before main\( \)](#)
- [7.20 Constructor and Destructor with Static Members](#)
- [7.21 Local vs Global Object](#)

## 7.1 INTRODUCTION

When a variable is declared and if not initialized, it contains garbage value. The compiler itself cannot carry out the process of initialization of a variable. The programmer needs explicitly to assign a value to the variable. Initialization prevents the variable from containing garbage value.

Consider an example

```
float height; // variable declaration
height=5.5; // assigning value to variable
```

In the above example, `height` is a variable of `float` type. It holds any garbage value before initialization. In the next statement, variable `height` is initialized with 5.5.

C++ handles abstract data type, which is the combination of one or more basic data type. An object holds copies of one or more individual data member variables. When an object is created, its data member contains garbage value.

We learned in the last chapter that declaring static member variables facilitates programmer to initialize member variables with desired values. The drawback of static members is that only one copy of static member is created for entire class. All objects share the same copy, which do not provide security. And the main disadvantage of static object is that its value remains in the memory throughout the program.

## 7.2 CONSTRUCTORS AND DESTRUCTORS

In the last chapter, we defined a separate member function for reading input values for data members. Using object, member function is invoked and data members are initialized.

The programmer needs to call the function. C++ provides a pair of in-built special member functions called ***constructor*** and ***destructor***. The constructor constructs the objects and destructor destroys the objects. In operation, they are opposite to each other. The compiler automatically executes these functions. The programmer does not need to make any effort for invoking these functions.

The C++ run-time arrangement takes care of execution of constructors and destructors. When an object is created, constructor is executed. The programmer can also pass values to the constructor to initialize member variables with different values. The destructor destroys the object. The destructor is executed at the end of the function when objects are of no use or goes out of scope. It is optional to declare constructor and destructor. If the programmer does not define them, the compiler executes implicit constructor and destructor.

Constructors and destructors are special member functions. Constructors and destructors decide how the objects of a class are created, initialized, copied, and destroyed. Their names are distinguished from all other member functions because their name is same as the class they belong to. The only difference is that destructor is preceded by ~ (tilde) operator.

Constructors and destructors have many attributes as that of normal member functions. We can declare and define them within the class, or declare them within the class and define them outside, but they have few unique characteristics.

## (1) CONSTRUCTORS

If a class *B* has one or more constructors, one of them is invoked each time when we define an object *b* of class *B*. The constructor creates object *b* and initializes it. Constructors are also called when local or temporary objects of a class are created.

***Example***

```
B( ) { }
```

## (2) DESTRUCTORS

Destructors are opposite to the constructor. The process of destroying the class objects created by constructors is done in destructor. The destructors have the same name as their class, preceded by a tilde (~). A destructor is automatically executed when object goes out of scope. It is also invoked when delete operator is used to free memory allocated with class pointer. Like constructor, it is not possible to define overloaded destructor and passing arguments to them. The class can have only one destructor. Destructors are called when these objects go out of scope.

***Example***

```
~B( ) { }
```

## 7.3 CHARACTERISTICS OF CONSTRUCTORS AND DESTRUCTORS

### (1) CONSTRUCTORS

- (1) Constructor has the same name as that of the class it belongs.
- (2) Constructor is executed when an object is declared.
- (3) Constructors have neither return value nor void.

(4) The main function of constructor is to initialize objects and allocate appropriate memory to objects.

(5) Though constructors are executed implicitly, they can be invoked explicitly.

(6) Constructor can have default and can be overloaded.

(7) The constructor without arguments is called as default constructor.

## **(2) DESTRUCTORS**

(1) Destructor has the same name as that of the class it belongs to and preceded by ~ (tilde).

(2) Like constructor, the destructor does not have return type and not even void.

(3) Constructor and destructor cannot be inherited, though a derived class can call the constructors and destructors of the base class.

(4) Destructors can be virtual, but constructors cannot.

(5) Only one destructor can be defined in the destructor. The destructor does not have any argument.

(6) The destructors neither have default values nor can be overloaded.

(7) Programmer cannot access addresses of constructors and destructors.

(8) TURBO C++ compiler can define constructors and destructors if they have not been explicitly defined. They are also called on many cases without explicit calls in program. Any constructor or destructor created by the compiler will be public.

(9) Constructors and destructors can make implicit calls to operators new and delete if memory allocation/ de-allocation is needed for an object.

(10) An object with a constructor or destructor cannot be used as a member of a union.

## **7.4 APPLICATIONS WITH CONSTRUCTORS**

The initialization of member variables of class is carried out using constructors. The constructor also allocates required memory to the object. An example of constructor follows next.

```
class num
{
    private:
        int a, b,c;

    public:

        num (void);      // declaration of constructor
        - - - - -
        - - - - -
};

num :: num (void)      // definition of constructor
{
    a=0;b=0;c=0;          // value assignment
}

main ( )
{
    class num x;
}
```

In the above example, class num has three member integer variables *a*, *b* and *c*. The declaration of constructor can be done inside the class and definition outside the class. In definition, the member variables of a class num are initialized to zero.

In the function `main( )`, *x* is an object of type class num. When an object is created, its member variables (private and public) are automatically initialized to the given value. The programmer need not write any statement to call the constructor. The compiler automatically calls the constructors. If the programmer writes a statement for calling a constructor, a constructor is called again. If there is no constructor in the program, in such a case the compiler calls dummy constructor. The constructor without argument is called as default constructor.

### 7.1 Write a program to define a constructor and initialize the class data member variables with constants.

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

class num
{

private:
    int a,b,c;
public:
    int x;

    num(void); // declaration of constructor

    void show()
    { cout <<"\n x = " <<x <<" a= " <<a <<" b= " <<b <<" c= " <<c; }
};

num :: num (void) // definition of constructor
{
    cout <<"\n Constructor called";
    x= 5 ; a= 0 ;b = 1;c = 2;
}

main( )
{
    clrscr( );
    num x;
    x.show( );
    return 0;
}
```

#### OUTPUT

**Constructor called**

**x = 5 a= 0 b= 1 c= 2**

**Explanation:** In the above program, the class num is declared with four integers *a*, *b*, *c* and *x*. The variables *a*, *b* and *c* are *private* and *x* is a *public* variable. The class also has `show( )` function and constructor prototype declaration. The function `show( )` displays the contents of the member variables on the screen. The definition of a constructor is done outside the class. In the function `main( )`, *x* is an object of class num. When an object is created, constructor is automatically invoked and member variables are initialized to given values as per the constructor definition. The values of variables *x*, *a*,

b and c are as per the output shown. The compiler calls the constructor for every object created. For each object the constructor is executed once i.e., the number of times of the execution of constructor is equal to the number of objects created. The program given below explains this point.

### 7.2 Write a program to show that for each object constructor is called separately.

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

{
private:
int a;

public:
num (void) // definition of constructor
{
    cout << "\n Constructor called.";
    a=1;
    cout << " a = " << a;
}

};

main( )
{
    clrscr( );
    num x,y,z;

    num a[2];
    return 0;
}
```

#### OUTPUT:

```
Constructor called. a = 1
```

**Explanation:** In the above program, x, y and Z are objects of class num. A [ 2 ] is an array of objects. For each individual object constructor is called. The total number of objects declared are five; hence, the constructor is called five times.

### 7.3 Write a program to read values through the keyboard. Use constructor.

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

class num
{
private:
int a,b,c;

public:
num(void); // declaration of constructor

void show( )
{   cout << "\n<<" a= " <<a <<" b= " <<b <<" c= " <<c; }

};
```

```

num :: num (void) // definition of constructor
{
    cout <<"\n Constructor called";
    cout <<"\n Enter Values for a,b and c : ";
    cin >> a>>b>>c;
}
main( )
{
    clrscr( );
    class num x;
    x.show( );

    return 0;
}

```

## OUTPUT

**Constructor called**

**Enter Values for a,b and c : 1 5 4**

**a = 1 b = 5 c = 4**

**Explanation:** The above program is the same as the previous one. In this program, whenever an object is created, the constructor is called and it reads the integer values through the keyboard. Thus, entered constants are assigned to member variables of the class. Here, the constructor is used like other functions.

## 7.5 CONSTRUCTORS WITH ARGUMENTS

In the previous example, constructors initialize the member variables with given values. It is also possible to create constructor with arguments and such constructors are called as parameterized constructors. For such constructors, it is necessary to pass values to the constructor when object is created. Consider the example given next.

```

class num
{
private:
    int a, b, c;
public:

num (int m, int j , int k) ; // declaration of constructor with arguments
- - - - -
- - - - -
};

num :: num (int m, int j, int k) // definition of constructor with arguments
{
    a=m;
    b=j;
    c=k;
}

main( )
{
    class num x=num (4,5,7); // Explicit call
    class num y (9,5,7); // Implicit call
}

```

In the above example, the declaration and definition of a constructor contains three integer arguments. In the definition the three arguments m, j and k are assigned to member variables a, b and c.

In a situation like this when class has a constructor with arguments, care is to be taken while creating objects. So far, we created the object using the following statement:

```
num x
```

This statement will not work. We need to pass required arguments as per the definition of constructor. Hence, to create object the statement must be as given below:

- (a) `class num x=num (4,5,7); // Explicit call`
- (b) `class num y (9,5,7); // Implicit call`

The statements (a) and (b) can be used to create objects, that not only create objects but also pass given values to the constructor. The method (a) is called as explicit call and the method (b) is called as implicit call.

#### 7.4 Write a program to create constructor with arguments and pass the arguments to the constructor.

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

class num
{
private:
int a,b,c;
public:

num(int m, int j , int k) ; // declaration of constructor with arguments

void show( )
{
cout <<"\n a= "<<a <<" b= "<<b <<" c= "<<c;
}

};

num :: num (int m, int j , int k) // definition of constructor with arguments
{
a=m;
b = j ;
c=k ;
}

main( )
{
clrscr( );
num x=num(4,5,7); // Explicit call
num y(1,2,8); // Implicit call
x.show( );
y.show( );
return 0;
}
```

#### OUTPUT

**a= 4 b= 5 c= 7**

**a= 1 b= 2 c= 8**

**Explanation:** In the above program, *x* and *y* are objects of class *num*. When objects are created, three values are passed to the constructor. These values are assigned to the member variables. The function *show( )* displays the contents of member variables.

#### 7.6 OVERLOADING CONSTRUCTORS

Like functions, it is also possible to overload constructors. In previous examples, we declared single constructors without arguments and with all arguments. A class can contain more than one constructor. This is known as constructor overloading. All constructors are defined with the same name as the class. All the constructors contain different number of arguments. Depending upon number of arguments, the compiler executes appropriate constructor. Table 7.1 describes object declaration and appropriate constructor to it.

Consider the following example:

```
class num
{
    private:
        int a;
        float b;
        char c;
    public:
        (a) num (int m, float j , char k);
        (b) num (int m, float j);
        (c) num ( );
        (d) class num x (4,5.5,'A');
        (e) class num y (1,2.2);
        (f) class num z;
```

**Table 7.1** Overloaded constructors

| Constructor declaration                                                      | Object declaration                                       |
|------------------------------------------------------------------------------|----------------------------------------------------------|
| (a) num(int m, float j , char k);<br>(b) num(int m, float j);<br>(c) num( ); | (d) num x(4,5.5,'A');<br>(e) num y(1,2.2);<br>(f) num z; |

In the above example, statements (a), (b) and (c) are constructor declarations and (d), (e) and (f) are the object declarations. The compiler decides which constructor to be called depending on number of arguments present with the object.

When object *x* is created, the constructor with three arguments is called because the declaration of an object is followed by three arguments. For object *y*, constructor with two arguments is called and lastly object *Z*, which is without any argument, is candidate for constructor without argument.

### 7.5 Write a program with multiple constructors for the single class.

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

class num
{
    private:
        int a;

        float b;
        char c;

    public:
        num(int m, float j , char k);
        num (int m, float j);
        num( );
```

```

void show( )
{
    cout <<"\n\t a= "<<a <<" b= "<<b <<" c= "<<c;
}
};

num :: num (int m, float j , char k)
{
    cout <<"\n Constructor with three arguments";
    a=m;
    b=j;
    c=k;
}

num :: num (int m, float j)
{
    cout <<"\n Constructor with two arguments";
    a=m;
    b=j;
    c=' ';
}

num :: num( )
{
    cout <<"\n Constructor without arguments";
    a=b=c=NULL;
}
main( )
{
    clrscr( );
    class num x(4,5.5,'A');

    x.show( );
    class num y(1,2.2);
    y.show( );
    class num z;
    z.show ( );
    return 0;
}

```

## **OUTPUT**

**Constructor with three arguments**

**a= 4 b= 5.5 c= A**

**Constructor with two arguments**

**a= 1 b= 2.2 c=**

**Constructor without arguments**

**a= 0 b= 0 c=**

**Explanation:** In the above program, three constructors are declared. The first constructor is with three arguments, second with two and third without any argument. While creating objects, arguments are passed. Depending on the number of arguments the compiler decides which constructor is to be called. In this program x, y and z are three objects created. The x object passes three arguments. The y object passes two arguments and z object passes no arguments. The function show( ) is used to display the contents of the class members.

## 7.7 CONSTRUCTORS WITH DEFAULT ARGUMENTS

Like functions, it is also possible to declare constructors with default arguments. Consider the following example.

```
power (int 9, int 3);
```

In the above example, the default value for the first argument is nine and three for second.

```
power p1 (3);
```

In this statement, object p1 is created and nine raise to 3 expression n is calculated. Here, one argument is absent hence default value 9 is taken, and its third power is calculated.

Consider the example on the above discussion given below.

**7.6 Write a program to declare default arguments in constructor. Obtain the power of the number.**

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

class power

{
    private:
        int num;
        int power;
        int ans;

    public :

power (int n=9,int p=3); // declaration of constructor with default arguments

    void show( )
    {
        cout <<"\n"<<num <<" raise to "<<power <<" is " <<ans;
    }
};

power :: power (int n,int p )
{
    num=n;
    power=p;
    ans=pow(n,p);
}

main( )
{
    clrscr( );
    class power p1,p2(5);
    p1.show( );
    p2.show( );
    return  0;
}
```

### OUTPUT

9 raise to 3 is 729

5 raise to 3 is 125

**Explanation:** In the above program, the class power is declared. It has three integer member variables and one member function show( ). The show( ) function is used to display the values of member data variables. The constructor of class power is declared with two default arguments. In the function main( ), p1 and p2 are two objects of class power. The p1 object is created without argument. Hence, the constructor uses default arguments in pow( ) function. The p2 object is created with one argument. In this call of constructor, the second argument is taken as default. Both the results are shown in output.

## 7.8 COPY CONSTRUCTORS

The constructor can accept arguments of any data type including user-defined data types and an object of its own class. Consider the examples given in Table 7.2.

**Table 7.2** Copy constructors

| <i>Statement (a)</i>                                                  | <i>Statement (b)</i>                                                      |
|-----------------------------------------------------------------------|---------------------------------------------------------------------------|
| <pre>class num { private: ----- ----- ----- public: num(num); }</pre> | <pre>class num { private: ----- ----- ----- public: num(num...) ; }</pre> |

In **statement (a)** an argument of the constructor is same as that of its class. Hence, this declaration is wrong. It is possible to pass reference of object to the constructor. Such declaration is known as copy constructor. The **statement (b)** is valid and can be used to copy constructor.

When we pass an object by value into a function, a temporary copy of that object is created. All copy constructors require one argument, with reference to an object of that class. Using copy constructors, it is possible for the programmers to declare and initialize one object using reference of another object. Thus, whenever a constructor is called a copy of an object is created.

### 7.7 Write a program to pass an object with reference to constructor. Declare and initialize other objects.

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

class num
{
    int n;
public :

    num ( ) { } // constructor without argument
    num (int k) { n=k; } // constructor with one argument

    num (num &j) // copy constructor
{
```

```

        n=j.n;
    }

void show (void) { cout <<n; }
};

main( )
{
    clrscr( );
    num J(50);
    num K(J);
    num L=J;
    num M;
    M=J;
    cout <<"\n Object J Value of n : ";
    J.show( );
    cout <<"\n Object K Value of n : ";
    K.show( );
    cout <<"\n Object L Value of n : ";
    L.show( );
    cout <<"\n Object M Value of n : ";
    M.show( );
    return 0;
}

```

## **OUTPUT**

**Object J Value of n : 50**  
**Object K Value of n : 50**  
**Object L Value of n : 50**  
**Object M Value of n : 50**

**Explanation:** In the above program, class num is declared with one integer member variable n and three constructors. In function main( ), the object J is created and 50 is passed to constructor. It is passed by value; hence, constructor with one argument is invoked. When object K is created with one object the copy constructor is invoked, object is passed, and data member is initialized. The object L is created with assignment with object J; this time also copy constructor is invoked. The compiler copies all the members of object J to destination object L in the assignment statement num L=J. When object is created without any value, like M, default constructor is invoked. The copy constructor is not invoked even if the object J is assigned to object M.

The data member variables that are dynamically allocated should be copied to the target object explicitly by using assignment statement or by copy constructor as per the given statement.

num L=J; / / copy constructor is executed

Consider the statement

M=J;

Here, M and J is predefined object. In this statement, copy constructor is not executed. The member variables of object J are copied to object M member-by- member. An assignment statement assigns value of one entity to another.

The statement num L=J; initializes object L with J during definition. The member variables of J are copied member-by-member into object L. This statement invokes constructor. This statement can be written as num L (J), which we frequently use to pass values to the constructor.

## 7.9 THE CONST OBJECTS

In the last chapter, we have studied the constant functions. The `const` declared functions do not allow the operations that alter the values. In the same fashion, we can also make the object constant by the keyword `const`. Only constructor can initialize data member variables of constant object. The data members of constant objects can only be read and any effort to alter values of variables will generate an error. The data members of constant objects are also called as read only data members. The constant object can access only constant functions. If constant object tries to invoke a non-member function, an error message will be displayed.

### 7.8 Write a program to declare constant object. Also, declare constant member function and display the contents of member variables.

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

class ABC
{
    int a;
public :

    ABC (int m)
    { a=m; }
    void show ( ) const
    { cout <<"a = "<<a; }
};

int main ( )
{
    clrscr( );
    const ABC x(5);
    x.show( );

return 0;
}
```

#### OUTPUT

A=5

**Explanation:** In the above program, class `ABC` is declared with one member variable and one constant member function `show( )`. The constructor `ABC` is defined to initialize the member variable. The `show( )` function is used to display the contents of member variable. In `main( )`, the object `x` is declared as constant with one integer value. When object is created, the constructor is executed and value is assigned to data member. The object `x` invokes the member function `show( )`.

## 7.10 DESTRUCTORS

Destructor is also a special member function like constructor. Destructors destroy the class objects created by constructors. The destructors have the same name as their class, preceded by a tilde (~).

For local and non-static objects, the destructor is executed when the object goes out of scope. In case the program is terminated by using `return` or `exit( )` statements, the

destructor is executed for every object existing at that time. It is not possible to define more than one destructor. The destructor is only one way to destroy the object. Hence, they cannot be overloaded.

A destructor neither requires any argument nor returns any value. It is automatically called when object goes out of scope. Destructor releases memory space occupied by the objects. The program given below explains the use of destructor.

### 7.9 Write a program to demonstrate execution of constructor and destructor.

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

struct text
{
    text( ) // Constructor
    {
        cout <<"\n Constructor executed." ;
    }

    ~text( ) // Destructor
    {
        cout <<"\n Destructor executed." ;
    }
};

void main( )
{
    clrscr ( );
    text t; // Object declaration
}
```

#### OUTPUT

**Constructor executed.**

**Destructor executed.**

**Explanation:** In the above program, the class `text` contains constructor and destructor. In function `main( )`, object `t` is executed. When object `t` is declared, constructor is executed. When object goes out of scope, destructor is executed.

### 7.10 Write a program to create an object and release them using destructors.

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

int c=0; // counter for counting objects created and destroyed.

class num
{
    public :

    num( )
    {
        c++;
        cout <<"\n Object Created : Object("<<c <<") ";
    }

    ~num( )
    {
        cout <<"\n Object Released : Object("<<c <<") ";
    }
}
```

```

    c-- ;
}
};

main( )
{
    clrscr( );

    cout <<"\n In main( ) \n"; num a,b;
    cout <<"\n\n In Block A\n";
{
    class num c;
}
cout <<"\n\n Again In main( )\n";
return 0;
}

```

### **Output**

**In main( )**

**Object Created : Objecte(1)**

**Object Created : Object (2)**

**In Block A**

**Object Created : Object(3)**

**Object Released : Object(3)**

**Again In main( )**

**Object Released : Object (2)**

**Object Released : Object(1)**

**Explanation:** In the above program, the variable *c* is initialized with zero. The variable *c* is incremented in constructor and decremented in destructor. When objects are created the compiler calls the constructor. Objects are destroyed or released when destructor is called. Thus, the value of a variable changes as the constructors and destructors are called. The value of a variable of *c* shows number of objects created and destroyed. In this program the object *a* and *b* are created in *main( )*. The object *c* is created in another block i.e., in block A. The object *c* is created and destroyed in the same block. The objects are local to the block in which they are defined. The object *a* and *b* are released as the control passes to *main( )* block. The object created last is released first.

## **7.11 CALLING CONSTRUCTORS AND DESTRUCTORS**

The compiler automatically calls the constructor and destructor. We can also call the constructor and destructor in the same fashion as we call the normal user-defined function. The calling methods are different for constructors and destructors. In practice, it may not be useful but for the sake of understanding, few examples are illustrated on this concept.

### **7.11 Write a program to invoke constructor and destructor.**

```

# include <iostream.h>
# include <conio.h>
class byte
{

```

```

int bit;
int bytes;

public :

byte( )
{
    cout <<"\n Constructor invoked";
    bit=64;
    bytes=bit/8;
}

~byte( )
{
    cout <<"\n Destructor invoked ";
    cout <<"\n Bit = "<<bit;
    cout <<"\n Byte = "<<bytes;
}
};

int main ( )
{
    clrscr( );
    byte x;
    byte( );           // calling constructor

    // x.byte( ) ;      // invalid statement
    // x.byte::byte( ) // valid statement
    // ~ byte( );       // invalid statement
    // byte.~byte( );   // invalid statement
    // x.~byte( );      // Member identifier expected

    x.byte::~byte( );

    return 0;
}

```

## **OUTPUT**

**Constructor invoked**

**Constructor invoked**

**Destructor invoked**

**Bit = 64**

**Byte = 8**

**Destructor invoked**

**Bit = 64**

**Byte = 8**

**Destructor invoked**

**Bit = 64**

**Byte = 8**

**Explanation:** In the above program, the class `byte` is declared with two integer members. The class has constructors and destructors. In function `main( )`, `x` is an object of class `byte`. When an object is created the compiler automatically invokes the constructor. The program also invokes constructor. Thus, constructor is executed twice i.e., implicitly and explicitly. The constructor is called without the help of object of that class. If we try to

call the constructor with the help of object x as per the statement x.byte( ), the compiler displays an error message.

The destructor cannot be invoked like constructor. Calling destructor requires the help of object and class name. Only object of the class alone is not able to invoke the destructor, the object needs help of the class. The statement x. ~byte( ) is invalid, the compiler displays the error message Member identifier expected. The destructor can be invoked using the statement x.byte::~byte( ). Here, class name is also specified. The constructor and destructor can call each other. The statement byte::byte( ) is used for calling destructor ~byte( ) within the body of constructor byte( ). The destructor ~byte( ) can call the constructor byte( ) using the statement byte( ). The constructor and destructor may contain condition statements; such destructors or constructors are called as conditional constructor or conditional destructor. The conditional constructor can be created using if-else or switch( ) statements. The example given below illustrates this.

### 7.12 Write a program to define conditional constructor and destructor.

```
# include <iostream.h>
# include <conio.h>
int c=0;

class byte

{
    int bit;
    int bytes;
public :

    ~byte( ); // Prototype declaration

    byte ( )
    { cout <<"\n Constructor invoked";
      bit=64;
      bytes=bit/8;
      byte:: ~byte( ); // call to destructor
    }
};

byte::~byte( )
{
    if (c==0)
    {
        // byte( ) // call to constructor
        cout <<"\n Destructor invoked ";
        cout <<"\n Bit = "<<bit;
        cout <<"\n Byte = "<<bytes;
    }
    c++;
}

int main( )
{
    clrscr( );
    byte x,y;
    return 0;
}
```

## OUTPUT

**Constructor invoked**

**Destructor invoked**

**Bit = 64**

**Byte = 8**

**Constructor invoked**

**Explanation:** In the above program, x and y are objects of class byte. The compiler automatically invokes the constructor. The constructor invokes the destructor. The variable c is declared and initialized with zero. It is a global variable declared before main( ) and can be used anywhere in the program. The destructor contains conditional statement if. The if block is executed only when c is zero. When first time destructor is called, the if block is executed and c is turned to non-zero value. Next time when destructor is called the if condition evaluates false and the if block is not executed. Though destructor is called, no result is displayed on the screen. Remember the constructors and destructors are executed for equal number of times. It is possible for the programmer to use the constructor and destructor like other user-defined functions.

## 7.12 QUALIFIER AND NESTED CLASSES

The class declaration can also be done inside the other class. While declaring object of such class, it is necessary to precede the name of the outer class. The name of outer class is called as qualifier name and the class defined inside is called as nested class.

The use of nested classes enhances the power of abstract data type and helps to construct more powerful data structure. The qualifier (host class) and nested class follow the same access conditions. The nested classes are useful to hide particular class and its objects within a qualifier class. The following program illustrates this.

### 7.13 Write a program to declare classes within classes and access the variables of both the classes.

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

class A          // Host class or qualifier class
{
public :
    int x;

    A( ) { x=10; }

class B          // Nested class
{
public:
    int y;

    B( ) { y=20; }

void show( )
{
    A a;
    cout <<"x="<<a.x<<endl;
```

```

        cout <<"y="<<y<<endl;
    }

};      // End of nested (inner) class

};      // End of qualifier (outer) class

void main( )
{
    clrscr( );
    A::B b;
    b.show( );
}

```

**OUTPUT**

**x=10**

**y=20**

**Explanation:** In the above program, the class B is declared inside the class A. All the members of both the classes are declared public so that they can be easily accessed. Both the classes have constructors to initialize the member variables. The class B also has one member function show( ) to display the contents on the screen.

In function main( ), A::B b declares object of class B. The name of class A is preceded because the class B is inside the class A. If we declare an object using the statement B b the program will be executed with a warning message "Use qualified name to access nested type 'A::B' ". Here, the class A acts as a qualified class.

The show( ) function is a member of class B. Due to this variable y is its member variable and can be directly accessed. To access the variable of a qualified class A, object of class A a is declared and through this object variable x is accessed. The data variables are public. In case private, we can access member variables using member functions. There is no limit for declaring classes within classes. The following program explains this.

#### 7.14 Write a program to declare multiple qualifier classes and declare an object of every class.

```

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

class A
{

public:
    int x;
    A( ) { x=5;
        cout <<" x= "<<x;
    }

    class B
    {
        public :
        int y;
        B( )
        {
            y=10;
            cout <<" y = "<<y;
        }
    }
}
```

```

class C
{
    public :
    int z;
    C( )
    {
        z=15;
        cout << " z =" << z;
    }
};

void main( )
{
    clrscr( );

    A a;           // outer class object
    A::B b;         // middle class object
    A::B::C c;       // inner class object
}

```

### OUTPUT

**x= 5 y=10 z =15**

**Explanation:** In the above program, classes A, B and C are declared. The class B is declared inside the class A. The class C is declared inside the class B. A and B are qualified classes for class C. The class A is qualified class for class B. Each class has a constructor that initializes and displays the contents of the object.

### 7.13 ANONYMOUS OBJECTS

Objects are created with names. It is possible to create objects without name and such objects are known as ***anonymous objects***. When constructors and destructors are invoked, the data members of the class are initialized and destroyed respectively. Thus, without object we can initialize and destroy the contents of the class. All these operations are carried out without object or we can also assume that the operations are carried out using an anonymous object, which exists but is hidden. The second thought is correct because if we apply pointer `this` (explained in Chapter “**Pointers and Arrays**”) we get the address of the object. It is not possible to invoke member function without using objects but we can invoke special member functions ***constructors*** and ***destructors*** which compulsorily exist in every class. Thus, without use of ***constructor*** and ***destructor*** the theory of ***anonymous object*** cannot be implemented in practical. Consider the following example.

### 7.15 Write a program to create anonymous object. Initialize and display contents of member variables.

```

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

class noname
{
    private :
    int x;
    public:

```

```

noname (int j)
{
    cout <<"\n In constructor";
    x=j;
    show( );
}
noname ( )
{
    cout <<"\n In constructor";
    x=15;
    show( );

}
~noname( ) { cout <<"\n In destructor "; }
noname *const show( )
{
    cout <<endl<<" x : "<<x;
    return this;
}

};

void main( )
{
    clrscr( );
    noname( );
    noname(12);
}

```

## OUTPUT

**In constructor**

**x:15**

**In destructor**

**In constructor**

**x:12**

**In destructor**

**Explanation:** In the above program, class noname is declared with one integer data member. The class also has constructor, destructor, and member function. The member function show( ) is used to display the contents on the screen.

In function main( ), no object is created. The constructor is called directly. Calling constructor directly means creating anonymous objects. In first call of the constructor, the member variable is initialized with 15. The constructor invokes the member function show( ) that displays the value of x on the screen.

It is not possible to create more than one anonymous object at a time. When constructor execution ends, destructor is executed to destroy the object. Again, the constructor with one argument is called and integer is passed to it. The member variable is initialized with 12 and it is displayed by show( ) function. Finally, the destructor is executed that marks the end of program.

## TIP

Calling constructors directly means creating anonymous objects.

## 7.14 PRIVATE CONSTRUCTORS AND DESTRUCTORS

We know that when a function is declared as private, it could be invoked by the public function of the same class. So far, we've declared constructors and destructors as public. The constructor and destructor can be declared as private. The constructor and destructor are automatically executed when an object is executed and when an object goes out of scope. Nevertheless, when the constructors and destructors are private, they cannot be executed implicitly, and hence, it is a must to execute them explicitly. The following program is illustrated concerning this.

### 7.16 Write a program to declare constructor and destructor as private and call them explicitly.

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

class A
{
    private:
        int x;

    ~A( ) { cout <<"\n In destructor ~A( )"; }

    A( ) { x=7; cout <<"\n In constructor A( )"; }

    public:

        void show( )
        {
            this->A::A( ); // invokes constructor
            cout <<endl << " x = "<<x;
            this->A::~A( ); // invoke destructor
        }
};

void main( )
{
    clrscr( );
    A *a;
    a->show( );
}
```

#### OUTPUT

In constructor A()

x = 7

In destructor ~A()

**Explanation:** In the above program, class A is declared. It has one data member, constructor, and destructor. The function show( ) is declared in public section. In function main( ), a is a pointer to class A. When pointer object is created, no constructor is executed. When show( ) function is invoked, the constructor is invoked and its return value is assigned to pointer \*this. Every member function holds this pointer that point to the calling object. Here, the pointer points to object a. When the pointer invokes the constructor, constructor is invoked. Consider the statement this=this->A::A( ). The statement invokes the zero argument

constructors. The contents is displayed by the `cout` statement and again the `this` pointer invokes the destructor.

## 7.15 DYNAMIC INITIALIZATION USING CONSTRUCTORS

After declaration of the class data member variables, they can be initialized at the time of program execution using pointers. Such initialization of data is called as dynamic initialization. The benefit of dynamic initialization is that it allows different initialization modes using overloaded constructors. Pointer variables are used as argument for constructors. The following example explains dynamic initialization using overloaded constructor.

### 7.17 Write a program to initialize member variables using pointers and constructors.

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

class city
{
    char city[20];
    char state[20];
    char country[20];

public:
    city( ) {     city[0]=state[0]=country[0]=NULL; }

    void display( char *line);

    city(char *cityn)
    {

        strcpy(city, cityn);
        state[0]=NULL;
    }

    city(char *cityn,char *staten)
    {
        strcpy(city,cityn);
        strcpy(state,staten);
        country[0]=NULL;
    }

    city(char *cityn,char *staten, char *countryn)
    {
        _fstrcpy(city,cityn);
        _fstrcpy(state,staten);
        _fstrcpy(country,countryn);
    }
};

void city:: display (char *line)
{
    cout <<line<<endl;
    if (_fstrlen(city))      cout<<"City      : "<<city<<endl;
```

```

    if (strlen(state))    cout <<"State    : "<<state <<endl;
    if (strlen(country)) cout <<"Country : "<<country <<endl;
}

void main( )
{
    clrscr( );
    city c1("Mumbai"),
          c2("Nagpur", "Maharashtra"),
          c3("Nanded", "Maharashtra", "India"),
          c4('\'0', '\'0', '\'0');

    c1.display("=====*=====");
    c2.display("=====*=====");
    c3.display("=====*=====");
    c4.display("=====*=====");
}

```

## **OUTPUT**

```
=====*=====
```

**City : Mumbai**

```
=====*=====
```

**City : Nagpur**

**State : Maharashtra**

```
=====*=====
```

**City : Nanded**

**State : Maharashtra**

**Country : India**

```
=====*=====
```

**Explanation:** In the above program, the class `city` is declared with three-character arrays `city`, `state` and `country`. The class `city` also has four constructors. They are zero-argument constructor, one argument constructor, two-argument constructor, and three-argument constructor. The `display ( )` member function is used to display the contents on the screen.

In function `main ( )`, `c1`, `c2`, `c3` and `c4` are declared and strings are passed. According to the number of arguments, respective constructors are executed. The function `display ( )` is called by all the four objects and information displayed is as per shown in the output. While calling function `display ( )`, the line format “====” is passed which is displayed before displaying the information. The `display ( )` function also contains `if` statements that checks the length of the string and displays the string only when the variable contains string. In case the data variable contains NULL, the string will not be displayed. The object `c4` is initialized with NULL character. The use of `c4` object is only to display line at the end of program.

## **7.16 DYNAMIC OPERATORS AND CONSTRUCTORS**

When constructors and destructors are executed they internally use `new` and `delete` operators to allocate and de-allocate memory. The dynamic construction

means allocation of memory by constructor for objects and dynamic destruction means releasing memory using the destructor. Consider the following program that explains the use of new and delete operator with constructors and destructors.

### 7.18 Write a program to use new and delete operators with constructor and destructor.

Allocate memory for given number of integers. Also release the memory.

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

int number (void);

class num
{
    int *x;

    int s;
public :

num( )
{
    s=number( );
    x= new int [s];
}

~num( ) { delete x; }

void input( );
void sum( );

};

void num :: input( )
{
    for ( int h=0;h<s;h++)
    {
        cout <<"Enter number ["<<h+1<< "] : ";
        cin>>x[h];
    }
}

void num :: sum( )
{
int adi=0;
for ( int h=0;h<s;h++)
adi+=x[h];
cout <<" sum of elements = "<<adi;
}

number( )
{
    clrscr( );
    int n;
    cout <<" How many numbers : ";
    cin>>n;
    return n;
```

```
}
```

```
void main( )
```

```
{
```

```
    num n1;
```

```
    n1.input( );
```

```
    n1.sum( );
```

```
}
```

## OUTPUT

How many numbers : 3

Enter number [1] : 1

Enter number [2] : 4

Enter number [3] : 5

sum of elements = 10

**Explanation:** In the above program, class num is declared with two variables and they are integer pointer \*x and integer variable s. The class also contains constructor and destructor. The responsibility of constructor is to allocate memory using new operator for given number of integers by the user. The non-member function number( ) is used to input number of elements through the keyboard and entered numbers are stored in the variable s. The variable s is used with the new operator to allocate memory. The memory is allocated to the pointer x.

The member function input( ) reads elements and the for loop repeats the statement `cin>>x[h]` for s (number of total elements) times. The `x[h]` is used like array. The x is the starting address and h indicates the successive locations. The address of pointer x remains unchanged. The value of h is added to value of x. In this way, all these numbers are stored in continuous memory locations. If you are still confused as to how successive memory locations are accessed, go through the pointer arithmetic operations. The sum( ) function is used to perform addition of all the entered numbers and displays the same on the screen. The destructor is finally executed which releases the memory using delete operator.

## TIP

In the above program, if we remove the operators new and delete, the program will work successfully. However, this is suitable for small programs, developed for demonstration purpose. In real application, it needs large amount of memory to be allocated or released. It is very essential to check the memory before doing the process. For example, when any program is loaded in the memory, the operating system checks that available system resources are enough and not to load the requested application by the user. If memory is insufficient, compiler displays some error messages to the user.

## 7.17 THE MAIN( ) AS A CONSTRUCTOR AND DESTRUCTOR

The constructors and destructors have the same name as their class. To use main( ) as a constructor and destructor we need to define class with the name main. So far, we have declared objects of classes without using the keyword class or struct because C++ treats classes like built-in data types. The use of keyword class or struct is only compulsory when the class name is main. This is because execution of program starts with the function main( ). The following program clears this point.

### 7.19 Write a program to declare class with name main.

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

class main
{
public:

main( ) { cout <<"\n In constructor main ( )"; }

~main( ) { cout <<"\n In destructor main ( )"; }

};

void main( )
{
    clrscr( );
    class main a;
}
```

**OUTPUT** In constructor main ()

In destructor main ()

**Explanation:** In the above program, the class is declared with the name main. Hence, it is compulsory to use keywords `class` or `struct` while declaring objects.

#### TIP

When the class name is main, it is compulsory to use keyword `class` or `struct` to declare objects.

### 7.18 RECURSIVE CONSTRUCTOR

Like normal and member functions, constructors also support recursion. The program given next explains this.

### 7.20 Write a program to invoke constructor recursively and calculate the triangular number of the entered number.

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

class tri_num
{
    int f;
    public :

tri_num( ) { f=0; }

void sum( int j) {    f=f+j;    }

tri_num(int m)
{
    if (m==0)
    {
        cout <<"Triangular number : "<<f;
```

```

        exit(1);
    }

    sum(m);
    tri_num::tri_num(--m);
}

;

void main( )
{
clrscr( );

tri_num a;
a.tri_num::tri_num(5);
}

```

## OUTPUT

Triangular number : 15

**Explanation:** In the above program, class `tri_num` is declared with private integer `f` and member function `sum()`. The class also has zero-argument constructor and one-argument constructor.

In function `main()`, `a` is an object of `tri_num` class. When object `a` is declared, the zero-argument constructor is executed. The object `a` invokes the constructor explicitly and passes integer value to it. The passed value is received by the variable `m` of the constructor. The `if` statement checks the value of variable `m` and if it is zero the triangular number is displayed and program is terminated.

The `sum()` function is called by the constructor and cumulative sum is calculated and stored in the member variable `f`. Followed by the call of function `sum()`, the constructor `tri_num()` is executed and the value of `m` is decreased by one. Thus, recursion takes places and each time value of `m` decreases. When the value of `m` becomes zero, the program terminates. We get the triangular number.

## TIP

Recursion is very useful hence readers are requested to make it more clear by stepwise executing the program, in a single step. This will clear the flow of program. First, try the recursion with normal function in C style and after perfect understanding try with member function and constructors, but do not drop it.

## 7.19 PROGRAM EXECUTION BEFORE MAIN ( )

It is impossible to execute a program without `main()`. The declaration of objects is allowed before `main()`. We know that when an object is created constructors and destructors are executed. The constructor can call other member functions. When an object is declared before `main()` it is called as global object. For global object constructor is executed before `main()` function and destructor is executed after the completion of execution of `main()` function. The following program illustrates this.

## 7.21 Write a program to declare global object and observe execution of constructor.

```

#include <iostream.h>
#include <constream.h>

struct text

```

```

    {
        text( )      // Constructor
    {
        cout <<"\n Constructor executed.";
    }

    ~text( )     // Destructor
    {
        cout <<"\n Destructor executed.";
    }
}

text t;      // Global object
void main( )
{   }

```

## **OUTPUT**

**Constructor executed.**

**Destructor executed.**

**Explanation:** In the above program, object `t` is declared before `main()`. It is a global object. As soon as an object is declared, its constructor is invoked immediately. However, the execution of every program starts from function `main()`, but for the object declared before `main()`, constructor is executed before execution of `main()`. The destructor for such object is executed after the complete execution of function `main()`. The constructor can invoke other member functions. These member functions are also executed before `main()`. Following program explains this.

### **7.22 Write a program to declare object before main() and invoke member function.**

```

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

class A
{
private:
    char *y;
    int x;

public :
    A( )
    {
        clrscr( );
        cout <<"\n In constructor ";
        x=15;
        show( );    // invokes member function
    }

    void show( ) { cout<<endl<<" In show( )   x="<<x; }

    ~A( ) { cout<<endl<<" In destructor"; }

};

A a;

void main( ) { }

```

## **OUTPUT**

**In constructor**

**In show ( ) x=15**

**In destructor**

**Explanation:** In the above program, class A is declared with member variable character pointer and integer x. The class also has constructor, destructor and member function show( ). Before main( ), object a is declared and constructor is executed that initializes the member variables. The constructor invokes the member function show( ). Finally, the destructor is executed.

## 7.20 CONSTRUCTOR AND DESTRUCTOR WITH STATIC MEMBERS

Every object has its own set of data members. When a member function is invoked, only copy of data member of calling object is available to the function. Some times it is necessary for all the objects to share data fields which is common for all the objects. If the member variable is declared as static, only one copy of such member is created for entire class. All objects access the same copy of static variable. The static member variable can be used to count number of objects declared for a particular class. The following program helps you to count number of objects declared for a class.

### 7.23 Write a program to declare static member variable. Count number of objects created and destroyed.

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

class man
{
    static int no;

    char name;
    int age;

public :
man( )
{
no++;
cout <<"\n Number of Objects exist: "<<no;
}
~ man( )
{
--no;
cout <<"\n Number of objects exist : "<<no;
}

};

int man:: no=0;

void main( )
{
clrscr( );

man A,B,C;
cout <<"\n Press any key to destroy object ";
```

```

    getch( );
}

OUTPUT

Number of Objects exist: 1
Number of Objects exist: 2
Number of Objects exist: 3
Press any key to destroy object
Number of objects exist: 2
Number of objects exist: 1
Number of objects exist: 0

```

**Explanation:** In this program, the class `man` has one static data member `no`. The static data member is initialized to zero. Only one copy of static data member is created and all objects share the same copy of static data member.

In function `main()`, objects `A`, `B` and `C` are declared. When objects are declared constructor is executed and static data member `no` is increased with one. The constructor also displays the value of `no` on the screen. The value of static member shows us the number of objects present. When the user presses a key, destructor is executed, which destroys the object. The value of static variable is decreased in the destructor. The value of static member variable shows number of existing objects.

## 7.21 LOCAL VS GLOBAL OBJECT

The object declared outside all function bodies is known as global object. All functions can access the global object. The object declared inside a function body is known as local object. The scope of local object is limited to its current block.

When global and local variables are declared with the same name, in such a case the scope access operator is used to access the global variable in the current scope of local variable. We know that the local variable gets first precedence than global variable. The same is applicable for objects. When a program contains global and local objects with the same name, the local object gets first preference in its own block. In such a case, scope access operator is used with global object. The following program illustrates this.

## 7.24 Write a program to show the difference between local and global object.

```

# include <iostream.h>
# include <constream.h>

class text
{
public:

void show (char *c)
{
    cout <<"\n" << c;
}

};

text t; // global object declaration

void main( )

```

```
{  
    text t;          // local object declaration  
    ::t.show("Global"); // call using global object  
    t.show("Local");   // call using local object  
}
```

## OUTPUT

Global

Local

**Explanation:** In the above program, the object `t` is declared in local as well as global scope. In function `main()`, using scope access operator and `t`, the function `show()` is invoked. The first time function `show()` is invoked using global object. The second call to function `show()` is made by local object. The local object does not require scope access operator. The scope access operator is used only when the same name for object is used in global and local scopes. If scope access operator is not used before object, for both times function `show()` is invoked by local object.

## SUMMARY

- (1) C++ provides a pair of in-built functions called **constructors** and **destructors**. The compiler automatically executes these functions. When an object is created constructor is executed. The programmer can also pass values to the constructor to initialize member variables with different values. The destructor destroys the object. It is executed at the end of program when objects are of no use.
- (2) Constructors and destructors decide how the objects of a class are created, initialized, copied, and destroyed. They are member functions. Their names are distinguished from all other member functions because they have the same name as the class they belong to.
- (3) It is also possible to create constructor with arguments like normal functions.
- (4) Like functions, it is also possible to overload constructors and assign default arguments.
- (5) When we pass an object by value into a function, a temporary copy of that object is created. All copy constructors require one argument, with reference to an object of that class. Using copy constructors, it is possible for the programmers to declare and initialize one object using reference of another object. Thus, whenever a constructor is called a copy of an object is created.
- (6) We can also make the object constant by the keyword `const`. Any effort to alter values of variables made by it will generate an error. The constant object can access only constant functions.
- (7) The compiler automatically calls the constructor and destructor. We can also call the constructor and destructor in the same fashion as we call the normal user-defined function. The calling methods are different for constructors and destructors.
- (8) Objects are created with names. It is not possible to declare objects without name. Such objects are known as **anonymous objects**.
- (9) When the constructor and destructor are private, they cannot be executed implicitly. Hence, it is a must to execute them explicitly.
- (10) We can also call the constructor and destructor in the same fashion as we call the normal user-defined function.
- (11) The class declaration can also be done inside the other class. While declaring object of such class, it is necessary to precede the name of the outer class. The name of outer class is called as qualifier name and the class defined inside is called as nested class.

- (12) The dynamic construction means allocation of memory by constructor for objects and dynamic destruction means releasing memory using the destructor.
- (13) To use `main()` as a constructor and destructor we need to define class with the name **main**.
- (14) Like normal and member functions, constructors also support recursion.
- (15) For global objects constructor is executed before `main()` function and destructor is executed after the completion of execution of `main()` function.
- (16) The object declared outside all function bodies is known as global object. All functions can access the global object. The object declared inside a function body is known as local object.

### EXERCISES

**[A] Answer the following questions.**

- (1) What are constructors and destructors?
- (2) Explain the characteristics of constructors and destructors?
- (3) Explain constructors with arguments. How are arguments passed to the constructor?
- (4) What do you mean by overloading of constructors? How does it benefit the programmer?
- (5) Explain constructor with default arguments?
- (6) What is copy constructor?
- (7) What are constant objects? How they are declared?
- (8) How are constructors and destructors called explicitly?
- (9) What is the difference between calling methods for constructor and destructor?
- (10) Is it possible for a constructor and destructor to call each other?
- (11) What are conditional constructors and destructors?
- (12) What is anonymous object?
- (13) What are nested and qualifier classes?
- (14) What is the difference between local object and global object?
- (15) What is static object? How is it different from normal object?
- (16) How are private constructors and destructors executed?
- (17) How will you declare constant object?
- (18) What is default constructor?
- (19) What is parameterized constructor?
- (20) Explain negative aspect of static objects.
- (21) What are local and global objects?

**[B] Answer the following by selecting the appropriate option.**

- (1) Constructors and destructors are automatically invoked by
  - (a) compiler
  - (b) operating system
  - (c) `main()` function
  - (d) object
- (2) Constructor is executed when
  - (a) object is declared
  - (b) object is destroyed
  - (c) both (a) and (b)

- (d) none of the above  
 (3) The destructor is executed when  
 (a) object goes out of scope  
 (b) when object is not used  
 (c) when object contains nothing  
 (d) none of the above

- (4) When memory allocation is essential, the constructor makes implicit call to  
 (a) new operator  
 (b) *malloc()*  
 (c) *memset()*  
 (d) random access memory

- (5) Destructors can be  
 (a) overloaded  
 (b) of any data type  
 (c) able to return result  
 (d) explicitly called

- (6) Constructor has the same name as  
 (a) the class they belong to  
 (b) the current program file name  
 (c) class name and preceded by ~  
 (d) both (a) and (c)

- (7) The following program displays

```
# include <iostream.h>
class A
{
    int x;
public:
    A( ) {x=10; }
    ~A( ) {}
};

void main ( )
{
    A a;
    cout<<(unsigned)a.A::A( );
}
```

- (a) address  
 (b) value  
 (c) both (a) and (b)  
 (d) none of the above

- (8) The following program returns address of

```
# include <iostream.h>

class A
{ public:
    A( ) {} ~A( ) {}
    A *display ( ) { return (&A( )); } };

void main ( )
{
    void *p;
    clrscr( );
```

```
A a;  
p=a.display( );  
cout <<p;  
}
```

- (a) constructor
- (c) first element if any
- (b) display ( )
- (d) none of the above

#### **[C] Attempt the following programs.**

- (1) Write a program to declare a class with private data members. Accept data through constructor and display the data with destructor.
- (2) Write a program to pass an object to constructor and carry out copy constructor. Display contents of all the objects.
- (3) Write a program to declare a class with three data members. Declare overloaded constructors with no arguments, one argument, two arguments, and three arguments. Pass values in the object declaration statement. Create four objects and pass values in such a way that the entire four constructors are executed one by one. Write appropriate messages in constructor and destructor so that the execution of the program can be understood.
- (4) Write a program to declare a class with two data members. Also, declare and define member functions to display the content of the class data members. Create object A. Display the contents of object A. Again initialize the object A using explicit call to constructor. This time pass appropriate values to constructor. Display the contents of object A using member function.
- (5) Write a program to call constructor recursively. Calculate factorial of a given number.
- (6) Write a program to create array of strings. Read and display the strings using constructor and destructor. Do not use member functions.
- (7) Write a program to create object without name.
- (8) Write a program to declare global and local objects with the same name. Access member functions using both the objects?

#### **[D] Find errors in the following programs.**

(1)

```
class text  
{  text ( )    { cout<<" Start";      }  
   ~text ( )    {   cout <<"\n End"; }  };
```

```
void main ( )    { text t; }
```

(2)

```
struct text  
{public:  
  
text (char *c)  
{ cout <<"\n"<<c; } };
```

```
void main ( ) { text t; }
```

(3)

```
class text  
{  char *c;  
  public:
```

```
text (*c) { cout <<"\n" <<c; } };  
void main ( ) { text t("I WON"); }
```

# 8

## CHAPTER

# Operator Overloading and Type Conversion

- [8.1 Introduction](#)
- [8.2 The Keyword Operator](#)
- [8.3 Overloading Unary Operators](#)
- [8.4 Operator Return Type](#)

- [8.5 Constraint on Increment and Decrement Operators](#)
- [8.6 Overloading Binary Operators](#)
- [8.7 Overloading with friend Function](#)
- [8.8 Type Conversion](#)
- [8.9 Rules for Overloading Operators](#)
- [8.10 One Argument Constructor and Operator Function](#)
- [8.11 Overloading Stream Operators](#)

## 8.1 INTRODUCTION

Operator overloading is an important and useful feature of C++. The concept of operator overloading is quite similar to that of function overloading.

An operator is a symbol that indicates an operation. It is used to perform operation with constants and variables. Without an operator, programmer cannot build an expression.

C++ frequently uses user-defined data types, which is a combination of one or more basic data types. C++ has an ability to treat user-defined data type like the one they were built-in type. User-defined data types created from class or struct are nothing but combination of one or more variables of basic data types. The compiler knows how to perform various operations using operators for built-in types; however, for the objects those are instance of the class, the operation routine must be defined by the programmer.

For example, in traditional programming languages the operators such as +, - , <=, >= etc., can be used only with basic data types such as `int` or `float` etc. The operator `+` (plus) can be used to perform addition of two variables, but the same is not applicable for objects. The compiler cannot perform addition of two objects. The compiler would throw an error if addition of two objects is carried out. The compiler must be made aware about addition process of two objects. When an expression including operation with objects is encountered, a compiler searches for the definition of the operator, in which a code is written to perform operation with two objects. Thus, to perform operation with objects we need to redefine the definition of various operators. For example, for addition of objects *A* and *B*, we need to define operator `+` (plus). Re-defining the operator plus does not change its natural meaning. It can be used for both variables of built-in data types as well as objects of user-defined data types.

Operator overloading is one of the most valuable concepts introduced by C++ language. It is a type of polymorphism. Polymorphism permits to write multiple definitions for functions

and operators. C++ has number of standard data types like `int`, `float`, `char` etc. The operators `+`, `-`, `*` and `=` are used to carry operations with these data types. Operator overloading helps programmer to use these operators with the objects of classes. The outcome of operator overloading is that objects can be used in a natural manner as the variables of basic data types. Operator overloading provides the capability to redefine the language in which working operator can be changed.

Consider an example,

```
a=c+d;  
c=a-d;
```

where `a`, `c` and `d` are variables of basic data types like `int` or `float`. The use of operators `+`, `-` and `=` is valid. However, if we try these operators with the object, the compiler displays error message "Illegal structure operation".

```
class number  
{  
public:  
    int x;  
    int y;  
};
```

For example, `A`, `B` and `C` are three objects of class `number`. Each object holds individual copy of member variable `x` and `y`. We want to perform addition of `A` and `B` and store the result in `C`. For the sake of understanding the member variables are declared in public section. The addition of `A` and `B` means addition of member variables of `A` and member variables of `B`. The result of this operation will be stored in member variables of `C`. This feature can be implemented as shown below:

### 8.1 Write a program to perform addition of two objects and store the result in third object. Display contents of all the three objects.

```
# include <iostream.h>  
# include <constream.h>  
  
class number  
{  
public:  
    int x;  
    int y;  
  
    number( ) { } // ZERO ARGUMENT CONSTRUCTOR  
  
    number ( int j,  int k ) // TWO ARGUMENT CONSTRUCTOR  
    {  
        x=j;  
        y=k;  
    }  
  
    void show( )  
    {  
        cout <<"\n x="<<x <<" y="<<y;  
    }  
  
};  
void main( )  
{  
    clrscr ( );  
    number A(2,3), B(4,5), C ;
```

```

A.show( );
B.show( );

C.x=A.x+B.x; // ADDITION OF TWO OBJECTS
C.y=A.y+B.y; // USING MEMBER VARIABLES DIRECTLY

C.show();
}

```

## OUTPUT

x=2 y=3  
x=4 y=5  
x=6 y=8

**Explanation:** In the above program, A, B and C are objects of class number. Using constructor, objects are initialized. Consider the following statements:

```
C.x=A.x+B.x;
C.y=A.y+B.y;
```

In the above statements, addition of members of objects A and B is performed and stored in C. Each member variable is accessed individually and stored in member variable of C. For example, member x of A and member x of B are added and stored in x of C. Similarly, addition of other members is carried out.

In this program we cannot perform the operation C=A+B. The operation with objects is complicated because it involves operation of one or more data member variables which are part of objects.

## OPERATOR OVERLOADING

The capability to relate the existing operator with a member function and use the resulting operator with objects of its class as its operands is called operator overloading.

### 8.2 THE KEYWORD OPERATOR

The keyword operator defines a new action or operation to the operator.

#### Syntax:

```
Return type operator operator symbol (parameters )
{
    statement1;
    statement2;
}
```

The keyword 'operator', followed by an operator symbol, defines a new (overloaded) action of the given operator.

#### Example:

```
number operator + (number D)
{
    number T;
    T.x=x+D.x;
    T.y=y+D.y;
    return T;
}
```

Overloaded operators are redefined within a C++ class using the keyword operator followed by an operator symbol. When an operator is overloaded, the produced symbol is called the operator function name. The above declarations provide an extra meaning to the

operator. Operator functions should be either member functions or friend functions. A friend function requires one argument for unary operators and two for binary operators. The member function requires one argument for binary operators and no argument for unary operators. When the member function is called, the calling object is passed implicitly to the function and hence available for member function. While using friend functions, it is essential to pass the objects by value or reference. The prototype of operator functions in classes can be written as follows:

- (a) void operator ++( );
- (b) void operator --( );
- (c) void operator - ( );
- (d) num operator+(num);
- (e) friend num operator \* (int ,num);
- (f) void operator = (num);

Operator overloading can be carried out in the following steps:

- (a) Define a class which is to be used with overloading operations.
- (b) The public declaration section of the class should contain the prototype of the function operator( ).
- (c) Define the definition of the operator ( ) function with proper operations for which it is declared.

## 8.2 Write a program to perform addition of two objects using operator keyword.

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

class number
{
public:
    int x;
    int y;

    number( ) { } // ZERO ARGUMENT CONSTRUCTOR

    number(int j, int k ) // TWO ARGUMENT CONSTRUCTOR
    {
        x=j;
        y=k;
    }

    number operator + ( number D)
    {
        number T;
        T.x=x+D.x;
        T.y=y+D.y;
        return T;
    }

    void show( )
{
```

```

        cout <<"\n x="<<x <<" y="<<y;
}

};

void main( )

{
    clrscr ( )
    number A(2,3),B(4,5),C;
    A.show ( );
    B.show ( );
    C=A+B;
    C.show( );
}

```

### **OUTPUT**

**x=2 y=3**

**x=4 y=5**

**x=6 y=8**

**Explanation:** In the above program, A, B and C are objects of class number. Here, the addition has been performed using statement C=A+B. Remember, in the last program we were not able to execute this statement. Instead of this, two separate statements were used to perform addition.

In this program, the statements that perform addition operation of each individual member of objects are written in function operator. The operator has return type and single argument. It also uses a local object (T) to hold addition as long as the operator function is active. Whenever the statement C=A+B is executed, the compiler searches for definition of operator +. The object A invokes the operator function and object B is passed as argument. The copy of object B is stored in the formal argument D. The member variables of A are directly available in operator function as the function is invoked by the same object. The addition of individual members are carried out and stored in member variable of object T. The return type of operator function is same as that of its class. The function returns object T and it is assigned to variable C.

## **8.3 OVERLOADING UNARY OPERATORS**

Overloading devoid of explicit argument to an operator function is called as unary operator overloading. The operator ++, --, and - are unary operators. The unary operators ++ and -- can be used as **prefix** or **suffix** with the functions. These operators have only single operand. The examples given below illustrate the overloading of unary operators.

### **8.3 Write a program to increment member variables of object. Overload unary ++ operator.**

```

# include <iostream.h>
# include <constream.h>

class num
{
    private :
    int a,b,c,d;

    public :

```

```

num ( int j, int k, int m ,int l)
{
a=j;
b=k;
c=m;
d=l;
}

void show(void);
void operator ++( );

};

void num :: show( )
{
    cout <<" A= "<<a <<" B= " <<b <<" C = "<<c <<" D = "<<d;
}

void num :: operator ++( )
{
    ++a; ++b; ++c; ++d;
}

main( )
{
    clrscr( );
    num X(3,2,5,7);
    cout <<"\n Before Increment of X : ";
    X.show( );
    ++X;
    cout <<"\n After Increment of X : ";
    X.show( );
    return 0;
}

```

## OUTPUT

**Before Increment of X : A= 3 B= 2 C = 5 D = 7**

**After Increment of X : A= 4 B= 3 C = 6 D = 8**

**Explanation:** In the above example, the class num contains four integer variables a, b, c and d. The class also has two-member functions show( ) and operator ++( ) and one parameterized constructor. The constructor is used to initialize object. The show( ) displays the contents of the member variables. The operator ++( ) overloads the unary operator++. When this operator is used with integer or float variables, its value is increased by one. In this function, ++ operator precedes each member variable of class. This operation increments the value of each variable by one.

In function main( ), the statement ++X calls the function operator ++( ), where, X is an object of the class num. The function can also be called using statement X.operator ++( ). In the output, values of member variables before and after increment operations are displayed.

## 8.4 Write a program to overload - operator.

```

# include <iostream.h>
# include <conio.h>
class num
{

```

```

private :
    int a,b,c,d;
public :
num ( int x, int y, int z, int w)
{
    a=x;
    b=y;
    c=z;
    d=w;
}
void show(void);
void operator -( );
};

void num :: show( )
{
    cout <<"A= "<<a <<" B= " <<b <<" C = "<<c <<" D = "<<d;
}

void num :: operator -( )
{
a=-a;
b=-b;

c=-c;
d=-d;
}
main( )
{
    clrscr( );
    num X(2,2,8,4);

    cout <<"\nBefore Negation of X : ";
    X.show( );
    -X;
    cout <<"\nAfter Negation of X : ";
    X.show( );
    return 0;
}

```

## **OUTPUT**

**Before Negation of X : A= 2 B= 2 C = 8 D = 4**

**After Negation of X : A= -2 B= -2 C = -8 D = -4**

**Explanation:** The above program is same as the previous one. Here, the operator – is overloaded. The statement `-X` calls the function `operator -( )`. The function `operator -( )` makes all the member variables negative. The function `show( )` displays the values of member variables on the screen.

## **8.4 OPERATOR RETURN TYPE**

In the last few examples we declared the operator () of void types i.e., it will not return any value. However, it is possible to return value and assign to it other objects of the same type. The return value of operator is always of class type, because the operator overloading is only for objects. An operator cannot be overloaded for basic data types. Hence, if the operator returns any value, it will be always of class type. Consider the following program.

## **8.5 Write a program to return values from operator( ) function.**

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

```

# include <conio.h>

class plusplus
{
    private :
    int num;

    public :
    plusplus( ) { num=0; }

    int getnum( ) { return num; }

    plusplus operator ++ (int)
    {
        plusplus tmp;
        num=num+1;
        tmp.num=num;
        return tmp;
    }
};

void main( )
{
    clrscr( );
    plusplus p1, p2;
    cout <<"\n p1 = "<<p1.getnum( );
    cout <<"\n p2 = "<<p2.getnum( );
    p1=p2++;
    cout <<endl<<" p1 = "<<p1.getnum( );
    cout <<endl<<" p2 = "<<p2.getnum( );
    p1++;
    // p1++=2;
    cout <<endl<<" p1 = "<<p1.getnum( );
    cout <<endl<<" p2 = "<<p2.getnum( );
}

```

## **OUTPUT**

```

p1 = 0
p2 = 0
p1 = 1
p2 = 1
p1 = 2
p2 = 1

```

**Explanation:** In the above program, class `plusplus` is declared with one private integer `num`. The class constructor initializes the object with zero. The member function `getnum( )` returns current value of variable `num`. The operator `++( )` is overloaded and it can handle as postfix increment of the objects. In case of prefix incrimination it will flag an error.

The `p1` and `p2` are objects of the class `plusplus`. The statement `p1=p2++`, first increments the value of `p2` and then assigns it to the object `p1`. The values displayed will be one for the objects. The object `p1` is increased. This time the values of object displayed will be two and one.

## **8.5 CONSTRAINT ON INCREMENT AND DECREMENT OPERATORS**

When an operator (increment/decrement) is used as prefix with object, its value is incremented/ decremented before operation and on the other hand the postfix use of operator increments/decrements the value of variable after its use.

When ++ and -- operators are overloaded, no difference exists between postfix and prefix overloaded operator functions. The system has no way of determining whether the operators are overloaded for postfix or prefix operation. Hence, the operator must be overloaded in such a way that it will work for both prefix and postfix operations. The ++ or -- operator overloaded for prefix operation works for both prefix as well as postfix operations but with a warning message, but not vice-versa. To make a distinction between prefix and postfix notation of operator, a new syntax is used to indicate postfix operator overloading function. The syntaxes are as follows:

```
Operator ++( int ) // postfix notation
Operator ++( ) // prefix notation
```

The argument followed by operator (++) or (--) should have type 'int'. When a postfix operator ++ or operator -- is declared, the last parameter must be declared with the type int. No other types such as float, long etc., are allowed. We can use this operator with all types of variables including float, long etc. Declaring int does not mean that it is only for integer type. The following program illustrates overloading of ++ operator in postfix and prefix style.

### 8.6 Write a program to overload ++ and -- operator for prefix and postfix use.

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

class number
{
    float x;
public :
    number ( float k ) { x=k; }

void operator ++ (int) // postfix notation
{ x++; }

void operator -- ( ) // prefix notation
{ --x; }

void show( ) { cout <<"\n x=" <<x; }

};

void main( )

{
    clrscr( );
    number N(2.3);
    cout <<"\n Before Incrimination: ";
    N.show ( );
    cout <<"\n After Incrimination: ";
    N++; // postfix increment
    N.show( );
    cout <<"\n After Decrementation:" ;
    --N; // prefix decrement
    N.show( );
}
```

## OUTPUT

**Before incrimination:**

**x=2.3**

**After incrimination:**

**x=3.3**

**After decrementation:**

**x=2.3**

**Explanation:** In this program, operator ++ and -- are overloaded. The ++ operator is overloaded for postfix use and -- operator is overloaded for prefix use. You can see the keyword (`int`) is followed by the operator ++, which is necessary for postfix notation of operator. The operator -- is overloaded for prefix operation. Here, a value of `float` member variable is incremented and decremented.

## 8.6 OVERLOADING BINARY OPERATORS

Overloading with a single parameter is called as binary operator overloading. Like unary operators, binary operator can also be overloaded. Binary operators require two operands. Binary operators are overloaded by using member functions and friend functions.

### (1) OVERLOADING BINARY OPERATORS USING MEMBER FUNCTIONS

If overloaded as a member function they require one argument. The argument contains value of the object, which is to the right of the operator. If we want to perform the addition of two objects `o1` and `o2`, the overloading function should be declared as follows:

```
operator (num o2);
```

Where, `num` is a class name and `o2` is an object.

To call function operator the statement is as follows:

```
o3=o1+o2;
```

We know that a member function can be called by using class of that object. Hence, the called member function is always preceded by the object. Here, in the above statement, the object `o1` invokes the function `operator ()` and the object `o2` is used as an argument for the function. The above statement can also be written as follows:

```
o3=o1.operator +(o2);
```

Here, the data members of `o1` are passed directly and data members of `o2` are passed as an argument. While overloading binary operators, the left-hand operand calls the operator function and right-hand operand is used as an argument.

### (2) OVERLOADING BINARY OPERATORS USING FRIEND FUNCTIONS

The friend can be used alternatively with member functions for overloading of binary operators. The friend function requires two operands to be passed as arguments.

```
o3=o1+o2;
```

```
o3=operator +(o1,o2);
```

Both the above statements have the same meaning. In the second statement, two objects are passed to the operator function.

The use of member function and friend function produces the same result. Friend functions are useful when we require performing an operation with operand of two different types. Consider the statements:

```
X =Y+3;  
X =3+Y;
```

Where, X and Y are objects of same type. The first statement is valid. However, the second statement will not work. The first operand must be an object of the same class. This problem can be overcome by using friend function. The friend function can be called without using object. The friend function can be used with standard data type as left-hand operand and with an object as right-hand operand.

Following programs are illustrated based on the above discussion.

### 8.7 Write a program to overload + binary operator.

```
# include <iostream.h>  
# include <conio.h>  
  
class num  
{  
    private :  
        int a,b,c,d;  
  
    public :  
  
        void input(void);  
        void show(void);  
        num operator+(num);  
  
};  
  
void num :: input()  
{  
    cout <<"\n Enter Values for a,b,c and d : ";  
    cin >>a>> b>>c>>d;  
}  
  
void num :: show()  
{  
    cout <<" A= "<<a <<" B= " <<b <<" C = "<<c <<" D = "<<d <<"\n";  
}  
  
num num :: operator +(num t)  
{  
    num tmp;  
  
    tmp.a=a+t.a;  
    tmp.b=b+t.b;  
    tmp.c=c+t.c;  
    tmp.d=d+t.d;  
    return (tmp);  
}  
  
main()  
{
```

```

clrscr( );
num X,Y,Z;
cout <<"\n Object X";
X.input( );
cout <<"\n Object Y";
Y.input( );
Z=X+Y;
cout <<"\nX : ";
X.show( );
cout <<"Y : ";
Y.show( );
cout <<"Z : ";

Z.show( );
return 0;
}

```

## **OUTPUT**

**Object X**

**Enter Values for a,b,c and d : 1 4 2 1**

**Object Y**

**Enter Values for a,b,c and d : 2 5 4 2**

**X : A= 1 B= 4 C = 2 D = 1**

**Y : A= 2 B= 5 C = 4 D = 2**

**Z : A= 3 B= 9 C = 6 D = 3**

**Explanation:** In the above program, binary operator + is overloaded. Using the overloading operator +, addition of member variables of two objects are performed and results are assigned to member variables of third object. In this program X, Y and Z are objects of class num. The statement Z=X+Y invokes the operator function. In this statement the object Y is assigned to object t of operator function and member variables of X are accessed directly. The object tmp is used for holding the result of addition and it is returned to object Z after function execution. The function show( ) displays the values of three objects.

## **8.8 Write a program to perform multiplication using an integer and object.**

**Use friend function.**

```

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

class num
{
    private :
    int a,b,c,d;

    public :

    void input(void);
    void show(void);
    friend num operator * (int ,num); // friend function declaration
};

void num :: input( )

```

```

{
    cout <<"\n Enter Values for a,b,c and d : ";
    cin >>a>> b>>c>>d;
}

void num :: show( )
{
    cout <<" A= "<<a <<" B= " <<b <<" C = "<<c <<" D = "<<d <<"\n";
}

num operator * (int a, num t)
{
    num tmp;

    tmp.a=a*t.a;
    tmp.b=a*t.b;
    tmp.c=a*t.c;
    tmp.d=a*t.d;
    return (tmp);
}
main( )
{
    clrscr( );
    num X,Z;
    cout <<"\n Object X";
    X.input( );
    Z=3*X;
    cout <<"\nX : ";
    X.show( );
    cout <<'Z : ';
    Z.show( );

    return 0;
}

```

## **OUTPUT**

### **Object X**

**Enter Values for a,b,c and d : 1 2 2 3**

**X : A= 1 B= 2 C = 2 D = 3**

**Z : A= 3 B= 6 C = 6 D = 9**

**Explanation:** In the above program, the equation  $Z=3*X$  contains integer and class object. We know that the left-hand operand is always used to invoke the function and the right-hand operand is passed as an argument. In such type of equations member functions are not useful because the left-hand operand is integer and cannot invoke the function. Hence, the function `operator * ( )` is declared as `friend`. The `friend` function calls the `operator * ( )` and carries the multiplication of each member variable by three. The results are displayed in the output.

## **8.7 OVERLOADING WITH FRIEND FUNCTION**

Friend functions are more useful in operator overloading. They offer better flexibility which is not provided by the member function of the class. The difference between member function and friend function is that the member function takes arguments explicitly. Quite

the opposite, the friend function needs the parameters to be explicitly passed. The syntax of operator overloading with friend function is as follows:

```
friend return-type operator operator-symbol(variable1, variable2)
{
    statement1;
    statement2;
}
```

The keyword **friend** precedes function prototype declaration. It must be written inside the class. The function can be defined inside or outside the class. The arguments used in friend functions are generally objects of the friend classes. A friend function is similar to normal function. The only difference is that friend function can access private members of the class through the objects. Friend function has no permission to access private members of a class directly. However, it can access the private members via objects of the same class.

### 8.9 Write a program to overload unary operator using friend function.

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

class complex
{
    float real,imag;
public:

complex( )      // zero argument constructor
{
    real=imag=0;
}

complex (float r, float i) // two argument constructor
{
    real=r;
    imag=i;
}

friend complex operator - ( complex c)
{
    c.real=-c.real;
    c.imag=-c.imag;
    return c;
}

void display( )
{
    cout <<"\n Real : "<<real;
    cout <<"\n Imag : "<<imag;
}
;

void main( )
{
    clrscr( );
    complex c1(1.5,2.5),c2;
    c1.display( );
    c2=-c1 ;
    cout <<"\n\n After Negation \n";
    c2.display( );
}
```

```
OUTPUT
Real : 1.5
Imag : 2.5
After Negation
Real : -1.5
Imag : -2.5
```

**Explanation:** In the above program, operator -- is overloaded using friend function. The operator function is defined as friend. The statement `c2=-c1` invokes the operator function. This statement also returns the negated values of `c1` without affecting actual value of `c1` and assigns it to object `c2`.

The negation operation can also be used with an object to alter its own data member variables. In such a case the object itself acts as a source and destination object. This can be accomplished by sending reference of object. Program 8.10 illustrates this.

### **8.10 Write a program to pass reference of an object to operator function.**

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

class complex
{
    float real,imag;
public:
complex (float r, float i) // two argument constructor
{
    real=r;
    imag=i;
}

friend void operator - ( complex & c)
{
    c.real=-c.real;
    c.imag=-c.imag;
}

void display( )
{
    cout <<"\n Real : "<<real;
    cout <<"\n Imag : "<<imag;
}
};

void main( )
{
    clrscr( );
    complex c1(1.5,2.5);
    c1.display( );
    -c1;
    cout <<"\n\n After Negation \n";
    c1.display( );
}
```

**OUTPUT****Real : 1.5****Imag : 2.5****After Negation****Real : -1.5****Imag : -2.5**

**Explanation:** In the above program, the object `c1` itself acts as source and destination object. The reference of object is passed to operator function. The object `c` is a reference object of `c1`. The values of object `c` are replaced by itself by applying negation.

## 8.8 TYPE CONVERSION

When constants and variables of various data types are clubbed in a single expression, automatic type conversion takes place. This is so for basic data types. The compiler has no idea about the user-defined data types and about their conversion to other data types. The programmer should write the routines that convert basic data types to user-defined data types or vice versa. There are three possibilities of data conversion as given below:

- (1) Conversion from basic data type to user-defined data type ( class type).
- (2) Conversion from class type to basic data type.
- (3) Conversion from one class type to another class type.

### (1) CONVERSION FROM BASIC TO CLASS TYPE

The conversion from basic to class type is easily carried out. It is automatically done by the compiler with the help of in-built routines or by applying typecasting. In this type the left-hand operand of = sign is always class type and right-hand operand is always basic type. The program given below explains the conversion from basic to class type.

**8.11 Write a program to define constructor with no argument and with float argument. Explain how compiler invokes constructor depending on data type.**

```
# include <iostream.h>
# include <conio.h>
class data
{   int x;
    float f;

public :
    data ( )
    {
        x=0;
        f=0;
    }

    data ( float m)
    {
        x=2;
        f=m;
    }
}
```

```

void show( )
{
    cout <<"\n x= "<<x <<" f = "<<f;
    cout <<"\n x= "<<x <<" f = "<<f;
}
};

int main( )
{
    clrscr( );
    data z;
    z=1;
    z.show( );
    z=2.5;
    z.show( );
    return 0;
}

```

### **OUTPUT**

```

x= 2 f = 1
x= 2 f = 1
x= 2 f = 2.5
x= 2 f = 2.5

```

**Explanation:** In the above program, the class data has two member variables each of integer and float types respectively. It also has two constructors one with no argument and second with float argument. The member function `show()` displays the contents of the data members. In function `main()`, `z` is an object of class `data`. When `z` is created the constructor with no argument is called and data members are initialized to zero. When `z` is initialized to one the constructor with float argument is invoked. The integer value is converted to float type and assigned member variable `f`. Again when `z` is assigned to 2.5, same process is repeated. Thus, the conversion from basic to class type is carried out.

## **(2) CONVERSION FROM CLASS TYPE TO BASIC DATA TYPE**

In the previous example, we studied how compiler makes conversion from basic to class type. The compiler does not have any knowledge about the user-defined data type built using classes. In this type of conversion, the programmer explicitly needs to tell the compiler how to perform conversion from class to basic type. These instructions are written in a member function. Such type of conversion is also known as overloading of type cast operators. The compiler first searches for the operator keyword followed by data type and if it is not defined, it applies the conversion functions. In this type, the left-hand operand is always of basic data type and right-hand operand is always of class type. While carrying this conversion, the statement should satisfy the following conditions:

- (1) The conversion function should not have any argument.
- (2) Do not mention return type.
- (3) It should be a class member function.

### **8.12 Write a program to convert class type data to basic type data.**

```

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

class data

```

```

{
    int x;
    float f;

public :
    data( )
    {
        x=0;
        f=0;
    }
operator int( )
{
    return (x);
}

operator float( )
{ return f; }

data ( float m)
{
    x=2;
    f=m;
}

void show( )
{
    cout <<"\n x= "<<x <<" f = "<<f;
    cout <<"\n x= "<<x <<" f = "<<f;
}
};

int main( )
{
    clrscr( );
    int j;
    float f;
    data a;
    a=5.5;

    j=a; // operator int ( ) is executed
    f=a; // operator float ( ) is executed

    cout <<"\n Value of j : "<<j;
    cout <<"\n Value of f : "<<f;
    return 0;
}

```

## **OUTPUT**

**Value of j : 2**

**Value of f : 5.5**

**Explanation:** In the above program, the class `data` has two member variables each of `integer` and `float` data type. It also contains constructors as per described in the last example. In addition, it contains overloaded data types `int` and `float`. These functions are useful for conversion of data from class type to basic type. Consider the following statements:

(a) `j=a;`

(b) `f=a;`

In the first statement object `a` is assigned to integer variable `j`. We know that class type data is a combination of one or more basic data types. The class contains two-member functions `operator int()` and `operator float()`. Both these functions are able to convert data types from class to basic. In statement (a) the variable `j` is of integer type, the function `operator int()` is invoked and integer value data member is returned. In statement (b), `f` is of `float` type, the member function `operator float()` is invoked.

### **(3) CONVERSION FROM ONE CLASS TYPE TO ANOTHER CLASS TYPE**

When an object of one class is assigned to object of another class, it is necessary to give clear-cut instructions to the compiler. How to make conversion between these two user-defined data types? The method must be instructed to the compiler. There are two ways to convert object data type from one class to another. One is to define a conversion operator function in source class or a one-argument constructor in a destination class. Consider the following example:

`X=A;`

Here, `X` is an object of class `XYZ` and `A` is an object of class `ABC`. The class `ABC` data type is converted to class `XYZ`. The conversion happens from class `ABC` to `XYZ`. The `ABC` is a source class and `XYZ` is a destination class.

We know the operator function `operator data-type()`. Here, data type may be built-in data type or user-defined data type. In the above declaration, the data type indicates target type of object. Here, the conversion takes place from class `ABC` (source class) to class `XYZ` (destination class).

#### **8.13 Write a program to convert integer to date and vice versa using conversion function in source class.**

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

class date
{
    char d[10];
    public :
date( ) { d[0]=NULL; }

date(char *e) { strcpy(d,e); }

void show( ) { cout <<d; }
};

class dmy
{

    int mt,dy,yr;
    public:
dmy ( ) { mt=dy=yr=0; }
```

```

dmy (int m, int d, int y)
{
    mt=m;
    dy=d;
    yr=y;
}
operator date( )
{
    char tmp[3],dt[9];

    itoa(dy,dt,10);
    strcat(dt,"-");

    itoa(mt,tmp,10);

    strcat(dt,tmp);
    strcat(dt,"-");

    itoa(yr,tmp,10);
    strcat(dt,tmp);
    return (date(dt));
}

void show( )
{
    cout <<dy<<" "<<mt<<" "<<yr;
}
};

int main( )
{
clrscr( );
date D1;
dmy D2(1,7,99);

D1=D2;
cout <<endl<<"D1=";
D1.show( );
cout <<endl<<"D2=";
D2.show( );
return 0;
}
OUTPUT
D1=7-1-99
D2=7 1 99

```

**Explanation:** In the above program, `date` and `dmy` are two classes declared. In function `main( )`, `D1` is an object of class `date` and `D2` is an object of class `dmy`. The object `D2` is initialized.

The statement `D1=D2` initializes `D1` with `D2`. Here, both the objects `D1` and `D2` are of different types hence the conversion function `date( )` is called to perform the conversion from one object to another object.

**8.14 Write a program to convert integer to date and vice versa using conversion function in destination class.**

```

# include <iostream.h>
# include <conio.h>
# include <stdlib.h>
# include <string.h>

class dmy
{
    int d,m,y;

    public :

dmy( ) { d=m=y=0; }

dmy(int da, int ma, int ya) { d=da; m=ma; y=ya; }

int day( ) { return (d); }

int month( ) { return (m); }

int year( ) { return (y); }

void show( ) { cout <<d<<" "<<m<<" "<<y; }
};

class date
{
    private :
    char dts[9];
    public :
    date( ) { dts[0]=0; }

date ( char *e) { strcpy (dts,e); }

void show( ) { cout <<dts; }

date ( dmy k)
{
    int d=k.day( );      // first member function
    int m=k.month( );    // second member function
    int y=k.year( );     // third member function

    char tmp[3];
    itoa(d,dts,10);
    strcat(dts,"-");
    itoa(m,tmp,10);
    strcat(dts,tmp);
    strcat(dts,"-");
    itoa(y,tmp,10);
    strcat(dts,tmp);
}

};

int main( )
{
    clrscr( );
    date D1;
    dmy D2(1,3,77);
    D1=D2;
}

```

```

    cout<<endl<<"D1=";
    D1.show( );
    cout <<endl<<"D2=";
    D2.show( );
    return 0;
}

```

## **OUTPUT**

**D1=1-3-77**

**D2=1 3 77**

**Explanation:** In the above program, as soon as the statement `D1=D2` is executed, the one-argument constructor defined in the class `date` is invoked. The one argument constructor carries the conversion. The constructor calls three-member function of `dmy` class to get the day, month, and year of the date. Here, in this program conversion is done by using the constructor in destination object.

## **8.15 Write a program to convert class data type to another class data type.**

```

# include <iostream.h>
# include <conio.h>
class minutes
{
    int m;

public :
    minutes( )
    { m=240; }

    get( )
    { return (m); }

    void show( )
    { cout <<"\n Minutes = "<<m; }

};

class hours
{
int h;
public :
void operator = (minutes x);
void show ( )
{ cout <<"\n Hours = "<<h; }

};

void hours:: operator = (minutes x)
{
    h=x.get( )/60;
}

int main( )

```

```

{
    clrscr( );
    minutes minute;
    hours hour;
    hour=minute;
    minute.show( );
    hour.show( );
return 0;
}

```

## **OUTPUT**

**Minutes = 240**

**Hours = 4**

**Explanation:** In the above program, two classes are declared. The class `minutes` has one integer member variable `m` and two member functions `get()` and `show()`. It also contains constructor without argument. The class `hours` has one integer member variable and `show()` member function. The class `hours` contains overloaded operator function. In function `main()`, `minute` is an object of class `minutes` and `hour` is an object of class `hours`. The program converts minutes to hours. The equation `hour=minute` invokes the operator function. In function `operator()`, `x` is an object of class `minutes`. The object `x` invokes the member function `get()` that returns total number of minutes. Number of hours is obtained by dividing the total number of minutes by 60. The equation `h=x.get()/60` performs this task and assigns result to `h`. Thus, the result of the program is as per given above.

## **8.9 RULES FOR OVERLOADING OPERATORS**

- Overloading of an operator cannot change the basic idea of an operator. When an operator is overloaded, its properties like syntax, precedence, and associativity remain constant. For example `A` and `B` are objects. The following statement

`A+=B;`

assigns addition of objects `A` and `B` to `A`. The overloaded operator must carry the same task like original operator according to the language. The following statement must perform the same operation like the last statement.

`A=A+B;`

- Overloading of an operator must never change its natural meaning. An overloaded operator `+` can be used for subtraction of two objects, but this type of code decreases the utility of the program. Remember that the aim of operator overloading is to comfort the programmer to carry various operations with objects. Consider the following program.

## **8.16 Misuse of operator overloading. Perform subtraction using + operator.**

```

# include <iostream.h>
# include <constream.h>

class num
{
    int x;
public:

    num( ) { x=0; }
    num( int k) { x=k; }

```

```

num operator + ( num n)
{
    num s;
    s.x=x-n.x;
    return s;
}

void show( ) { cout <<"\n x="<<x; }

};

void main( )

{
    clrscr( );
    num a(10), b(5),c;
    c=a+b;
    c.show( );
}

```

## OUTPUT

X=5

**Explanation:** In the above program, the operator + is overloaded. It performs subtraction. Such type of misuse must be avoided while overloading operators. The programmer thought that it would perform addition but in reality, it performs subtraction.

- The overloaded operator should contain one operand of user-defined data type. Overloading operators are only for classes. We cannot overload the operator for built-in data types.
- Overloaded operators have the same syntax as the original operator. They cannot be prevailing over the original operators.
- There is no higher limit to the number of overloading for any operator. An operator can be overloaded for a number of times if the arguments are different in each overloaded operator function.
- Operator overloading is applicable within the scope (extent) in which overloading occurs.
- Only existing operators can be overloaded. We cannot create a new operator.
- C++ has wide range of operators. However, few operators cannot be overloaded to operate in the same manner like built-in operators. The operators given in Table 8.1 cannot be overloaded.

**Table 8.1** Non-overloadable operators

| Operator | Description                |
|----------|----------------------------|
| .        | Member operator            |
| .*       | Pointer to member operator |

|           |                       |
|-----------|-----------------------|
| ::        | Scope access operator |
| ?:        | Conditional operator  |
| Sizeof( ) | Size of operator      |
| # and ##  | Preprocessor symbols  |

For example, operators such as ?:, :: and .\* are combinations of more than one symbol. The condition operator needs three arguments. It is only one operator that requires three arguments. Hence, above operators cannot be overloaded.

- The operators given in [Table 8.2](#) cannot be overloaded using `friend` function.
- When unary operators are overloaded using member functions, it requires no explicit friend argument and returns no value whereas when it is overloaded using friend function, it requires one reference argument.
- When binary operators are overloaded using `friend` functions, it requires two arguments whereas when overloaded using member function, it requires one argument.

**Table 8.2** Non-overloadable operators with friend function

| Operator | Description                      |
|----------|----------------------------------|
| ()       | Function call delimiter/operator |
| =        | Assignment operator              |
| []       | Subscripting operator            |
| ->       | Class member access operator     |

## 8.10 ONE ARGUMENT CONSTRUCTOR AND OPERATOR FUNCTION

We know that a single argument constructor or an operator function could be used for conversion of objects of different classes. A large range of classes as class libraries are available with the compiler, however, they are linked with the main program. Their source code is invisible to us. Only objects of these classes can be used. The user cannot change the in-built classes. The problem occurs when programmer attempts conversion from object of class declared by him/her to type of in-built class. This problem can be avoided by defining conversion routine in the user-defined class. The conversion routine may be single

argument constructor or an operator function. It depends on the object whether it is source or destination object. Table 8.3 describes conversion type and place of routine to be defined, followed by description.

**Table 8.3** Conversion types

| Sr.No. | Conversion type | Routine in destination class | Routine in source class |
|--------|-----------------|------------------------------|-------------------------|
| A      | Class to class  | Constructor                  | Conversion function     |
| B      | Class to basic  | Not applicable               | Conversion function     |
| C      | Basic to class  | Constructor                  | Not applicable          |

(A) In case both the source and destination objects are of user-defined type, the conversion routine can be carried out using operator function in source class or using constructor in destination class.

(B) If the user-defined object is a destination object, the conversion routine should be carried out using single argument constructor in the destination object's class.

(C) In case the user-defined object is a source object, the conversion routine should be carried out using an operator function in the source object's class.

Defining multiple conversion routines puts the compiler in an uncertain condition. The compiler fails to select appropriate conversion routines. For example, if one argument constructor is present in destination class and operator function in source class, the compiler cannot select appropriate routines. Hence, while defining conversion routines, follow the conditions given in Table 8.3.

## 8.11 OVERLOADING STREAM OPERATORS

The predefined objects `cin` and `cout` are used to perform various input/ output operations in C++. The extraction operator (`>>`) is used with `cin` object to carry out input operations. The insertion operator (`<<`) is used with `cout` object to carry out output operations. In Chapter "***Input and Output in C++***", we learnt how to create objects similar to `cin` and `cout`. It is also possible to overload both these extraction and insertion operator with `friend` function.

The syntax for overloading (`<<`) insertion operator is as follows:

```
friend ostream & operator << ( ostream & put , v1)
{
    // code
    return put;
}
```

The keyword `friend` precedes the declaration. The `ostream` is an output stream class followed by reference and keyword operator. The `put` is an output stream object like `cout`. The `v1` is a user-defined class object. The following program explains overloading of insertion operator with friend function.

### 8.17 Write a program to overload insertion operator (`<<`) with friend function.

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

class string
{
```

```

char *s;
public:

    string ( char *k)
    {
        s=k;
    }

friend ostream & operator << (ostream &put, string & k) {
    put <<k.s;
    return put;
}
};

int main( )

{
    clrscr( );
    string s("INDIA");
    cout<<s;
    return 0;
}

```

## **OUTPUT**

**INDIA**

**Explanation:** In the above program, the insertion operator `<<` is overloaded with `friend` function. The overloaded operator allows us to display contents of objects directly using `cout` statement. The statement `cout<<s;` displays contents of object `s` on the screen.

In the same way, extraction operator can be overloaded. The syntax for overloading extraction operator follows below:

```

friend istream & operator >> ( istream & get, v2)
{
    // code
    return get;
}

```

The following program explains overloading of extraction operator.

### **8.18 Write a program to overload extraction operator using friend function.**

```

# include <iostream.h>
# include <constream.h>

class string
{
    char *s;

public:

friend istream & operator >> (istream &get, string & k)
{
    cout <<"\n Enter a string : ";
    get >>k.s;
    return get;
}

};

int main( )

```

```

{
    clrscr( );
    string s;
    cin>>s; // input string
    return 0;
}

```

## OUTPUT

### BEST LUCK

**Explanation:** This program is the same as the last one. Here, extraction operator is overloaded. The object `s` is directly used with `cin` statement. After execution of this statement, the overloaded operator is invoked.

## SUMMARY

- (1) Operator overloading is one of the most helpful concepts introduced by the C++ language. Operator overloading provides the capability to redefine the language in which working of operator can be changed.
- (2) Overloaded operators are redefined within a C++ class using the keyword `operator` followed by an operator symbol. When an operator is overloaded, the produced symbol is called the operator function name.
- (3) Overloading of operator cannot change the basic idea of an operator. When an operator is overloaded, its properties like syntax, precedence, and associativity remain constant.
- (4) The keyword `operator` defines a new action or operation to the operator.
- (5) The operators `++`, `--`, and `-` are unary operators. The unary operators `++` and `--` can be used as prefix or suffix with the functions. These operators have only single operand.
- (6) Binary operators require two operands. Binary operators are overloaded by using `member` functions and `friend` functions.
- (7) The conversion routine may be single argument constructor or an operator function. It depends on the object whether it is source or destination object.
- (8) Defining multiple conversion routines puts the compiler in an uncertain condition. The compiler fails to select appropriate conversion routines.
- (9) There are three possibilities of data conversion. They are given below:
  - (i) **Conversion from basic data type to user-defined data type (class type)**- The conversion from basic to class type is easily carried out. It is automatically done by the compiler with the help of in-built routines or by applying type casting.
  - (ii) **Conversion from class type to basic data type**- The compiler does not have any knowledge about the user-defined data type built using classes. In this type of conversion the programmer, needs explicitly to tell the compiler how to perform conversion from class to basic type. These instructions are written in a member function. Such type of conversion is also known as overloading of type cast operators.
  - (iii) **Conversion from one class type to another class type**- When an object of one class is assigned to an object of another class it is necessary to give clear-cut instructions to the compiler about how to make conversion between these two user-defined data types. Using constructor or conversion this function can be performed.
- (10) The conversion routine may be single argument constructor or an operator function. It depends on the object whether it is source or destination object. [Table 8.3](#) describes conversion type and place of routine to be defined.

## EXERCISES

**[A] Answer the following questions.**

- (1) What do you mean by operator overloading?
- (2) What is the use of the keyword `operator`?
- (3) What are the rules for overloading operators?
- (4) What is the difference between operator overloading and function overloading?
- (5) What is the difference between overloading of binary operators and unary operators?
- (6) How are `friend` functions used to carry out overloading of operators? In which situation are they helpful?
- (7) Explain conversion of data from basic to class type. Explain the role of compiler.
- (8) Explain conversion from class to basic type.
- (9) Explain conversion from class type to class type.
- (10) What are source and destination objects?
- (11) List the keywords that cannot be overloaded.
- (12) Describe rules for operator overloading.
- (13) Write syntaxes for overloading extraction and insertion operator with friend function.

**[B] Answer the following by selecting the appropriate option.**

- (1) The keyword `operator` is used to overload an
  - (a) operator
  - (b) function
  - (c) class
  - (d) none of the above
- (2) Consider the equation  $Z=3*X$ , to overload the `*` operator one of the following function `A` is used. `Z` and `X` are objects of the same class.
  - (a) `friend`
  - (b) `virtual`
  - (c) `public`
  - (d) none of the above
- (3) Which one of the following conversions is automatically carried by the compiler
  - (a) conversion from basic data type to user-defined data type (class type)
  - (b) conversion from class type to basic data type
  - (c) conversion from one class type to another class type
  - (d) none of the above
- (4) Which one of the following operator cannot be overloaded
  - (a) dot operator `(.)`
  - (b) plus operator `(+)`
  - (c) & ampersand operator
  - (d) `--` operator
- (5) In postfix overloading of operator `(++ or --)`, the last argument should have type
  - (a) `int`
  - (b) `void`
  - (c) `float`
  - (d) `long`
- (6) `A`, `B` and `C` are objects of same class. To execute the statement `C=A+B`, the operator must be overloaded.

- (a) +
  - (b) =
  - (c) both (a) and (b)
  - (d) none of the above
- (7) The operator function returns value of
- (a) basic type
  - (b) void type
  - (c) class type
  - (d) all types

**[C] Attempt the following programs.**

- (1) Write a program to overload < operator and display the smallest number out of two objects.
- (2) Write a program to overload = operator. Assign values of data members of one object to another object of the same type.
- (3) Write a program to overload == operator. Compare two objects using overloaded operator.
- (4) Write a program to carry conversion from class type to basic data type.
- (5) Write a program to carry conversion from one class data type to another class type.
- (6) Write a program to evaluate the equation  $A=B^3$ , where A and B are objects of same class. Use friend function.
- (7) Write a program to pass reference of object to operator function and change the contents of object. Use single object as source and destination object.
- (8) Write a program to declare two classes rupees and dollars. Declare objects of both the classes. Convert rupees to dollars and vice versa. Perform conversion using user-defined conversion routines.
- (9) Write a program to convert square to square root and vice versa.
- (10) Write a program to declare matrix class and perform addition of matrix class objects.

**[D] Find bugs in the following programs.**

```
(1)
Class num
{
    int x;
    public:

    num ( int k) { x=k; }

    int operator + ( num n)
    {
        num s(0);
        s.x=x-n.x;
        return x; }
    };

void main ( )
{
    num a(1), b(5),c(0);
    c=a+b; }

(2)
struct num
{
    int x;
    num ( int k) { x=k; }
```

```
void operator ++ ( )
{ ++x; }

void main ( )
{
    num b(2);
    b++;
    cout <<b.x;
}
```

# 9

## CHAPTER

# Inheritance

C  
H  
A  
P  
T  
E  
R  
O  
U  
T

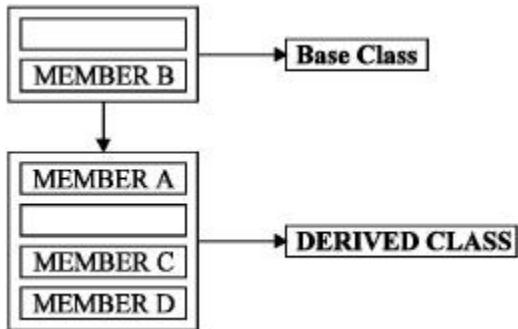
- [9.1 Introduction](#)
- [9.2 Access Specifiers and Simple Inheritance](#)
- [9.3 Protected Data with Private Inheritance](#)
- [9.4 Types of Inheritances](#)
- [9.5 Single Inheritance](#)

- [9.6 Multilevel Inheritance](#)
- [9.7 Multiple Inheritance](#)
- [9.8 Hierarchical Inheritance](#)
- [9.9 Hybrid Inheritance](#)
- [9.10 Multipath Inheritance](#)
- [9.11 Virtual Base Classes](#)
- [9.12 Constructors, Destructors and Inheritance](#)
- [9.13 Object as a Class Member](#)
- [9.14 Abstract Classes](#)
- [9.15 Qualifier Classes and Inheritance](#)
- [9.16 Common Constructor](#)
- [9.17 Pointers and Inheritance](#)
- [9.18 Overloading Member Function](#)
- [9.19 Advantages of Inheritance](#)
- [9.20 Disadvantages of Inheritance](#)

## 9.1 INTRODUCTION

Inheritance is one of the most useful and essential characteristics of object-oriented programming. The existing classes are main components of inheritance. The new classes are created from existing one. The properties of existing classes are simply extended to the

new classes. The new classes created using such methods are known as derived classes and the existing classes are known as base classes as shown in [Figure 9.1](#). The relationship between the base and derived class is known as *kind of relationship*. The programmer can define new member variables and functions in the derived class. The base class remains unchanged. The object of derived class can access members of base as well as derived class. On the other hand, the object of base class cannot access members of derived classes. The base classes do not know about their subclasses.



**Fig. 9.1** Inheritances

The term **reusability** means reuse of properties of base class in the derived classes. Reusability permits to reuse members of the previous class. In [Figure 9.1](#) the base class is reused. Reusability is achieved using inheritance. Inheritance and reusability are not different from each other. The outcome of inheritance is reusability. This chapter deals with implementation of the mechanism of inheritance and understand the features of it by illustrating several examples. The property of inheritance is slightly different in C++ as compared to other languages. C++ allows inheriting all the properties of previous class and there is a flexibility to add new members in the derived class, which are not available in the parent class. The base class is called as superclass, parent or ancestor and derived class as subclass, child or descendent. It is also possible to derive a class from previously derived class. A class can be derived from more than one class.

### INHERITANCE

The procedure of creating a new class from one or more existing classes is termed as inheritance.

## 9.2 ACCESS SPECIFIERS AND SIMPLE INHERITANCE

Before elaborating inheritance mechanism it is necessary to recall our memory for access rules studied in the chapter **Classes and Objects**. We have studied the access specifiers private and public in details and briefly studied the protected access specifiers previously. The public members of a class can be accessed by objects directly outside the class i.e., objects access the data member without member function of the class. The private members of the class can only be accessed by public member function of the same class. The protected access specifier is same as private. The only difference is that it allows its

derived classes to access protected members directly without member functions. This difference is explained later with suitable examples.

A new class can be defined as per the syntax given below. The derived class is indicated by associating with the base class. A new class also has its own set of member variables and functions. The syntax given next creates the derived class.

```
class name of the derived class: access specifiers - name of the base class
{
    _____ // Member variables of new class (derived class)
}
```

The derived class name and base class names are separated by colon (:). The access specifiers may be private or public. The keyword private or public is specified followed by colon. Generally, the access specifier is to be specified. In the absence of access specifier, the default is private. The access specifiers decide whether the characteristics of the base classes are derived privately or publicly. The derived class also has its own set of member variables and functions. The following are possible syntaxes of declaration:

(1)

```
class B : public A
{
    // members of class B
};
```

In the above syntax, class A is a base class, class B is a derived class. Here, class B is derived publicly.

(2)

```
class B : private A // private derivation
{
    // members of class B
};
```

(3)

```
class B: A // by default private derivation
{
    // members of class B
};
```

(4)

```
class B : protected A // same as private
{
    // members of class B
};
```

(5)

```
Struct B : A // public derivation
{
    // Members of class B
};
```

In the above syntaxes, the class B is derived privately from base class A. If no access specifier is given, the default mode is private. The use of protected access specifier give same results as private derivation. If struct is used instead of class, the default derivation is public. The following points are important to note:

(a) When a public access specifier is used, (example 1) the public members of the base class are public members of the derived class. Similarly, the protected members of the base class are protected members of the derived class.

(b) When a private access specifier is used, public and protected members of the base class are private members of the derived class.

## **(1) PUBLIC INHERITANCE**

A class can be derived publicly or privately. No third type exists. When a class is derived publicly, all the public members of base class can be accessed directly in the derived class whereas in private derivation, an object of derived class has no permission to access even public members of the base class directly. In such a case, the public members of the base class can be accessed using public member functions of the derived class.

In case the base class has private member variables and a class derived publicly, the derived class can access the member variables of base class using only member functions of the base class. The public derivation does not allow the derived class to access private member variables of the class directly as is possible for public member variables. The following example illustrates public inheritance where base class members are declared as public and private.

**9.1 Write a program to derive a class publicly from base class. Declare the base class with its member under public section.**

```
# include <iostream.h>
# include <constream.h>
                                // PUBLIC DERIVATION //
class A          // BASE CLASS
{
public:
    int x;
};

class B: public A // DERIVED CLASS
{
public:
    int y;
};

void main( )
{
    clrscr( );
    B b;           // DECLARATION OF OBJECT
    b.x=20;
    b.y=30;

    cout <<"\n Member of A : "<<b.x;
    cout <<"\n Member of B : "<<b.y;
}
```

### **OUTPUT**

**Member of A : 20**

**Member of B : 30**

**Explanation:** In the above program, two classes are defined containing one public member variable each. The class B is derived publicly from the class A. Consider the statement

```
class B : public A
```

The above statement is used to derive the new class. The keyword `public` is used to derive the class publicly. The access specifier is followed by the base class name.

In function `main( )`, `b` is an object of class B. The object `b` can access the members of class A as well as of B through the following statements:

```
b.x=20; // Access to base class members  
b.y=30; // Access to derived class members
```

Thus, the derived class holds members of base class and its object has permission to access members of base class.

## 9.2 Write a program to derive a class publicly from base class. Declare the base class member under private section.

```
# include <iostream.h>  
# include <constream.h>  
  
// PUBLIC DERIVATION //  
  
class A // BASE CLASS  
{  
    private:  
        int x;  
    public :  
  
        A ( ) { x=20; }  
  
        void showx ( )  
        {  
            cout <<"\n x=" <<x;  
        }  
  
};  
  
class B : public A // DERIVED CLASS  
{  
    public:  
        int y;  
  
        B ( ) { y=30; }  
  
        void show ( )  
        {  
            showx ( );  
            cout <<"\n y=" <<y;  
        }  
  
};
```

```

void main ( )
{
    clrscr( );
    B b;           // DECLARATION OF OBJECT
    b.show( );
}

```

## **OUTPUT**

**x=20**

**y=30**

**Explanation:** In the above program, the class A has one private member, default constructor and member function showx(). The class B is derived from the class A publicly. The class B contains one public member variable y, default constructor and member function show().

In function main(), b is an object of derived class B. Though, the class B is derived publicly from class A, the status of members of base class remains unchanged. The object b can access public members, but cannot access the private members directly. The private members of the base class can be accessed using public member function of the base class. The object b invokes the member function show() of derived class. The function show() invokes the showx() function of base class.

The object b can access member functions defined in both base and derived class. The following statements are valid:

```

b.showx( );      // Invokes member function of base class
b.show( );       // Invokes member function of derived class

```

The constructor and member functions of derived class cannot access the private members. Due to this, separate constructor and member functions are defined in both base and derived class.

## **(2) PRIVATE INHERITANCE**

The object of privately derived class cannot access the public members of the base class directly. Hence, member functions are used to access the members.

**9.3 Write a program to derive a class privately. Declare the member of base class under public section.**

```

// PRIVATE INHERITANCE //

# include <iostream.h>
# include <constream.h>

class A           // BASE CLASS
{
public:
    int x;
};


```

```

class B : private A // DERIVED CLASS
{
public:
int y;

B( )
{ x=20;
y=40;
}

void show( )

{ cout <<"\n x="<<x;
cout <<"\n y="<<y;
}
};

void main ( )
{

```

```

clrscr( );
B b; // DECLARATION OF OBJECT
b.show( );
}

```

## **OUTPUT**

**x=20**

**y=40**

**Explanation:** In the above program, the class B is derived privately from class A. The member variable x is a public member of base class. However, the object b of derived class cannot access directly the variable x. The following statement is invalid:

```
b.x=30; // cannot access
```

The class B is derived privately. Hence, its access is restricted. The member function of derived class can access the members of base class. The function show() does the same. From the last program it is clear that handling public data of a class is not a tough task for the programmer. However, for accessing private data members, difficulties are faced by the programmer, because they cannot be accessed without member functions. Even the derived class has no permission to access private data directly. Hence, the programmer needs to define separate constructor and member functions to access the private data of that particular base class. Due to this, the length of the program increases.

## **9.3 PROTECTED DATA WITH PRIVATE INHERITANCE**

The member functions of derived class cannot access the private member variables of base class. The private members of base class can be accessed using public member functions of the same class. This approach makes a program lengthy. To overcome the problem associated with private data, the creator of C++ introduced another access specifier

called **protected**. The **protected** is same as **private**, but it allows the derived class to access the private members directly.

Consider the example given on the next page.

#### **9.4 Write a program to declare protected data in base class. Access data of base class declared under protected section using member functions of derived class.**

```
// PROTECTED DATA //  
  
# include <iostream.h>  
# include <constream.h>  
  
class A           // BASE CLASS  
{  
    protected:      // protected declaration  
        int x;  
};  
  
class B : private A // DERIVED CLASS  
{  
    int y;  
public:  
    B ( )  
    { x=30;  
        y=40;  
    }  
  
    void show( )  
    {  
        cout <<"\n x=" <<x;  
        cout <<"\n y=" <<y;  
    }  
};  
void main( )  
{  
    clrscr( );  
    B b;           // DECLARATION OF OBJECT  
    b.show( );  
}
```

#### **OUTPUT**

**x=30**

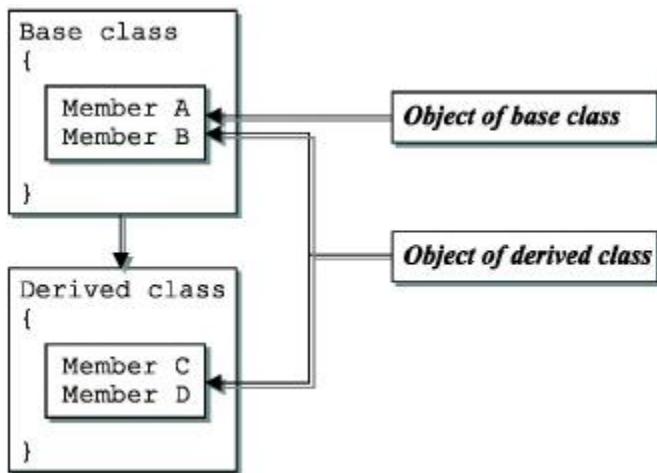
**y=40**

**Explanation:** In this program, the data member variable of class A is declared under **protected** section. The class A contains neither default constructor nor any member function. The class B is derived from class A. The member functions of derived class B can access the protected member of base class.

Thus, the **protected** mechanism reduces the program size. The derived class doesn't need to depend upon member function of base class to access the data. The **protected** data protects data from direct use and allows only derived classes to access the data.

The **protected** access specifier must be used when we know in advance that a particular class can be used as a base class. The classes that can be used as base class, their data members must be declared as **protected**. In such classes programmer does not need to define functions. The member functions of base class are rarely useful.

In the inheritance mechanism, the derived classes have more number of members as compared to base class. The derived class contains properties of base class and few of its own. The object of derived class can access members of both base as well as derived class. However, the objects of base class can access the members of only base class and not of any derived class as shown in [Figure 9.2](#). Hence, the programmer always uses objects of derived classes and performs operations. The member functions of base class cannot access the data of derived class. Obviously, the programmer uses objects of derived classes and possibly avoids defining member functions in base class.



**Fig.9.2** Difference between base and derived class objects

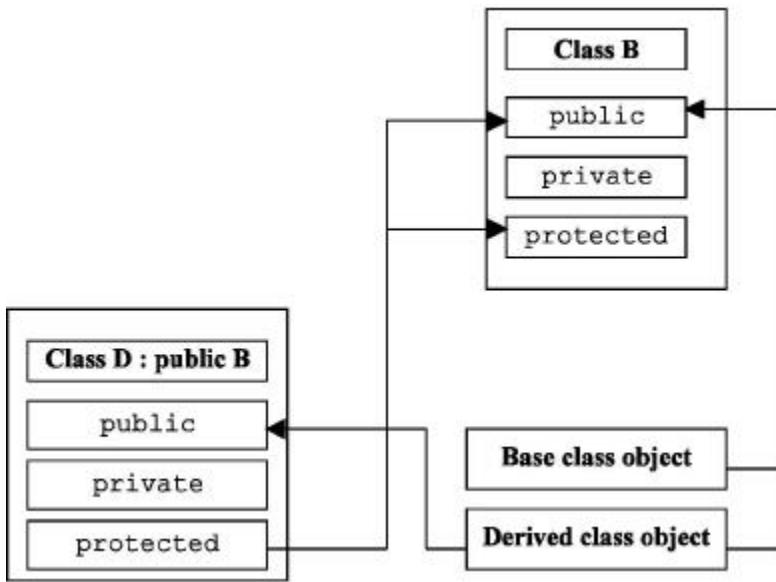
Now that we've learnt the access specifiers in detail let us revise few points related to `private`, `public` and `protected` keywords. [Table 9.1](#) gives description of access specifiers followed by explanation. [Figure 9.3](#) gives pictorial representation of access control.

**Table 9.1** Access specifiers with their scopes

| Sr.No. | Base class access mode | Derived class access mode  |                            |                            |
|--------|------------------------|----------------------------|----------------------------|----------------------------|
|        |                        | private derivation         | public derivation          | protected derivation       |
| A      | <code>public</code>    | <code>private</code>       | <code>public</code>        | <code>protected</code>     |
| B      | <code>private</code>   | <code>Not inherited</code> | <code>Not inherited</code> | <code>Not inherited</code> |
| C      | <code>protected</code> | <code>private</code>       | <code>protected</code>     | <code>protected</code>     |

- (A) Accessible to member functions of the same class, derived class, and using objects. When a class is derived by private derivation, public members of the base class are private in the derived class.
- (B) Accessible to member functions inside its own class but not in derived class. The derived class cannot access the private members of the base class directly. The private members of the base class can be accessed only by using public member functions of the same class.

(C) Accessible to member functions of base and derived class. When a class is derived privately, the protected members of the base class become private and when derived publicly, the protected members remain protected in the derived class.



**Fig. 9.3** Access scopes of class members

The syntax of `public`, `private` and `protected` access specifiers are as follows:

**Syntax :**

```

public: <declarations>
private: <declarations>
protected: <declarations>

```

If member variables of a class are protected, its scope is the same as for private. A protected member can be supposed as a hybrid of public and private members. A protected member is public for its derived class and private for other class members. Member functions and friends can use the member classes derived from the declared class only in objects of the derived type. It is possible to override the default struct access with private or protected. However, it is not possible to override the default union access. [Figure 9.5](#) shows pictorial representation of access control in classes.

- (1) All private members of the class are accessible to public members of the same class. They cannot be inherited.
- (2) The derived class can access private members of the base class using member function of the base class.
- (3) All the protected members of the class are available to its derived classes and can be accessed without the use of member functions of base class. In other words, we can say that all protected members act as public for the derived class.
- (4) If any class is prepared for deriving classes, it is advisable to declare all members of base class as protected so that derived classes can access the members directly.
- (5) All the public members of the class are accessible to its derived class. There is no restriction for accessing elements.

(6) The access specifier required while deriving classes is either `private` or `public`. If not specified, `private` is default for classes and `public` for structures.

(7) Constructors and destructors are declared in `public` section of the class. If declared in `private` section the object declared will not be initialized and compiler will flag an error.

(8) The `private`, `public`, and `protected` sections (visibility sections) can be defined several times in any sequence. In [Figure 9.4](#) you can observe that `public` section is declared twice, `private` section is defined at the end and `protected` section is declared in between two `public` sections. The sections can be declared in any sequence for any number of times.

```
class <class name>
{
    public:           // public section
    <declarations>
    protected:        // protected section
    <declarations>
    public:           // public section
    <declarations>
    private:          // private section.
    <declarations>
};
```

**Fig. 9.4** Visibility sections

### (1) MEMBER FUNCTIONS SCOPE

The following type of functions can have access to the `protected` and `private` members of the class. [Table 9.2](#) describes these functions. [Figure 9.5](#) shows access scope in classes.

**Table 9.2** Access controls of functions

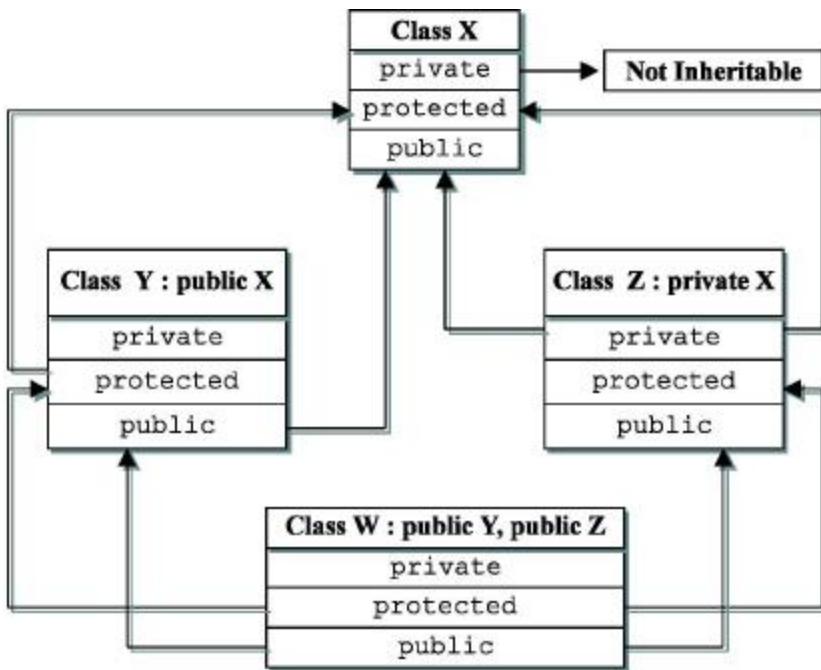
| Sr. No. | Type of Functions     | Access modes |           |        |
|---------|-----------------------|--------------|-----------|--------|
|         |                       | private      | protected | public |
| A       | Class member function | ✓            | ✓         | ✓      |
| B       | Derived class member  | ✗            | ✓         | ✓      |
| C       | Friend function       | ✓            | ✓         | ✓      |
| D       | Friend class member   | ✓            | ✓         | ✓      |

(A) The class member function can access the `private`, `protected` and `public` members.

(B) The derived class member function cannot access the `private` members of the base class directly. However, the `private` members can be accessed using member functions of the base class.

(C) The friend function of a friend class can access the `private` and `protected` members.

(D) The member function of a friend class can access the `private` and `protected` members of the class.



**Fig. 9.5** Access scope in classes

#### 9.4 TYPES OF INHERITANCES

So far we've learnt examples of simple inheritance that uses one base class and one derived class. The process of inheritance can be a simple one or may be complex.

This depends on the following points:

(1) Number of base classes: The program can use one or more base classes to derive a single class.

(2) Nested derivation: The derived class can be used as base class and new class can be derived from it. This can be possible to any extent.

Depending on the above points inheritance is classified as follows:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance
- Multi-path Inheritance

The different types of inheritance are described in [Figure 9.6](#). The base classes are at the top level and derived classes at the bottom. The arrow pointed from top to bottom indicates that properties of base classes are inherited by the derived class and the reverse is not applicable.

##### (1) SINGLE INHERITANCE

When only one base class is used for derivation of a class and the derived class is not used as base class, such type of inheritance between one base and derived class is known as **single inheritance**. [Figure 9.6 \(a\)](#) indicates single inheritance.

## (2) MULTIPLE INHERITANCE

When two or more base classes are used for derivation of a class, it is called **multiple inheritance**. Figure 9.6 (b) indicates multiple inheritance.

## (3) HIERARCHICAL INHERITANCE

When a single base class is used for derivation of two or more classes, it is known as **hierarchical inheritance**. Figure 9.6 (c) indicates hierarchical inheritance.

## (4) MULTILEVEL INHERITANCE

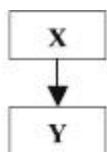
When a class is derived from another derived class i.e., derived class acts as a base class, such type of inheritance is known as **multilevel inheritance**. Figure 9.6 (d) indicates multilevel inheritance.

## (5) HYBRID INHERITANCE

The combination of one or more type of inheritance is known as **hybrid inheritance**. Figure 9.6 (e) indicates hybrid inheritance.

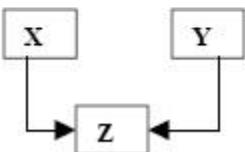
## (6) MULTIPATH INHERITANCE

When a class is derived from two or more classes that are derived from same base class such type of inheritance is known as **multipath inheritance**. Figure 9.6 (f) indicates multipath inheritance.



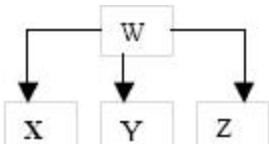
X is a base class. Y is a derived class. This type involves one base and derived class. Further, no class is derived from Y.

(a) Single Inheritance



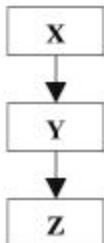
X and Y are base classes. Z is a derived class. Class Z inherits properties of both X and Y. Further, Z is not used as a base class.

(b) Multiple Inheritance



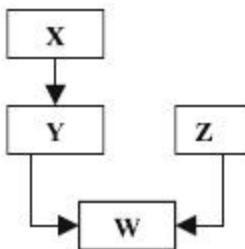
W is only one base class. X, Y and Z are derived classes. Further, X, Y and Z are not used for deriving a class.

(c) Hierarchical Inheritance



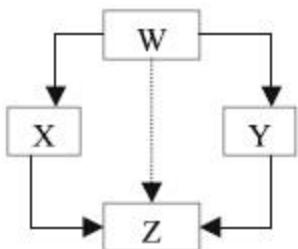
X is a base class. Y is derived from X. Further, Z is derived from Y. Here, Y is not only a derived class but also a base class. Further, Z can be used as a base class.

(d) Multilevel Inheritance



In this type, two types of inheritance is used i.e., single and multiple inheritance. Class Y is derived from class X. It is a single type of inheritance. Further, the derived class Y acts as a base class. The class W is derived from base classes Y and Z. This type of inheritance that uses more than one base class is known as multiple inheritance. Thus, combination of one or more type of inheritance is called as hybrid inheritance.

(e) Hybrid Inheritance



W is a base class. The classes X and Y are derived from W base class. Both X and Y inherit properties of class W. Further, class Z is derived from X and Y. X and Y have same copies of members inherited from W. Here, ambiguity is generated. Hence, virtual keyword is used to avoid ambiguity. The virtual mechanism is explained later.

(f) Multipath Inheritance

**Fig. 9.6** Types of Inheritances

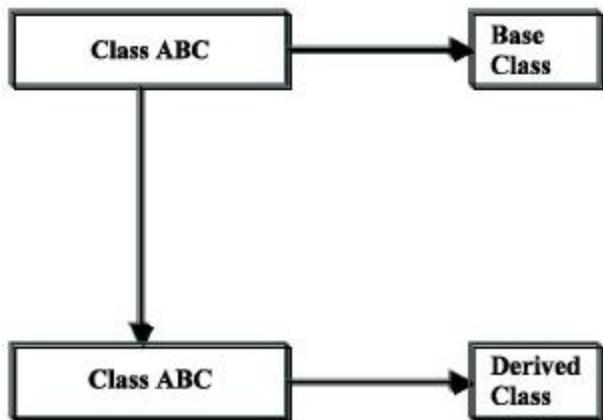
## 9.5 SINGLE INHERITANCE

When only one class is derived from a single base class such derivation of a class is known as **single inheritance**, further, the derived class is not used as a base class. This type of inheritance uses one base and one derived class.

The new class is termed as derived class and the old class is called as base class as shown in [Figure 9.7](#). A derived class inherits data member variables and functions of base class. However, constructors and destructors of base class are not inherited in derived class.

The newly created class receives entire characteristics from its base class. In single inheritance, there are only one base class and derived class. The single inheritance is not as complicated as compared to other types of inheritances.

In the above diagram, class ABC is a base class and class abc is derived class. The arrow shows that class abc is derived from class ABC. The program given below illustrates single inheritance.



**Fig. 9.7** Single inheritance

### 9.5 Write a program to show single inheritance between two classes.

```
# include <iostream.h>
# include <constream.h>
class ABC
{
protected:
    char name[15];
    int age;
};

class abc : public ABC      //  public derivation
{
    float height;
    float weight;

public:

void getdata( )
{
    cout <<"\n Enter Name and Age : ";
    cin >>name>>age;

    cout <<"\n Enter Height and Weight : ";
    cin >>height >>weight;
}

void show( )
```

```

{
    cout <<"\n Name : "<<name <<"\n Age : "<<age<<" Years";
    cout <<"\n Height : "<<height <<" Feets"<<"\n Weight :
        " <<weight <<" Kg.";
}
};

void main( )
{
    clrscr( );
    abc x;
    x.getdata( );           // Reads data through keyboard.
    x.show( );             // Displays data on the screen.
}

```

## OUTPUT

Enter Name and Age : Santosh 24

Enter Height and Weight : 4.5 50

Name : Santosh

Age : 24 Years

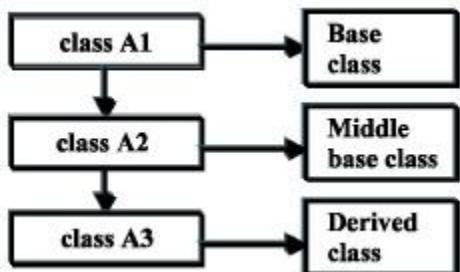
Height : 4.5 Feets

Weight : 50 Kg.

**Explanation:** In the above program, two classes ABC and abc are declared. The class ABC has two protected data members name and age. The class abc has two float data members height and weight with two-member functions getdata() and show(). The class abc is derived from class ABC. The statement class abc: public ABC defines the derived class abc. In function main(), x is an object of derived class abc. The object x invokes member functions getdata() and show(). This function reads and displays data respectively.

## 9.6 MULTILEVEL INHERITANCE

The procedure of deriving a class from derived class is named as **multilevel inheritance**.



**Fig. 9.8** Multilevel inheritance

In the Fig. 9.8 class A3 is derived from class A2. The class A2 is derived from class A1. The class A3 is derived class. The class A2 is a derived class as well as base class for class A3. The class A2 is called as intermediate base class. The class A1 is a base class of

classes A2 and A3. The series of classes A1, A2, and A3 is called as inheritance pathway as shown in Figure 9.8.

### 9.6 Write a program to create multilevel inheritance. Create classes A1, A2, and A3.

```
// Multilevel inheritance //  
  
# include <iostream.h>  
# include <constream.h>  
  
class A1           // Base class  
{  
protected :  
char name[15];  
int age;  
};  
  
class A2 : public A1 // Derivation first level  
{  
protected :  
float height;  
float weight;  
};  
  
class A3 : public A2 // Derivation second level  
{  
protected :  
char sex;  
public :  
  
void get( )          // Reads data  
{  
cout <<"Name    : "; cin >>name;  
cout <<"Age     : "; cin >>age;  
cout <<"Sex     : "; cin >>sex;  
cout <<"Height  : "; cin >>height;  
cout <<"Weight  : "; cin >>weight;  
}  
  
void show( )        // Displays data  
{  
cout <<"\nName   : " <<name;  
cout <<"\nAge    : " <<age <<" Years";  
cout <<"\nSex    : " <<sex;  
cout <<"\nHeight : " <<height <<" Feets";  
cout <<"\nWeight : " <<weight <<" Kg.>";  
}  
};  
  
void main( )  
{  
clrscr( );  
A3 x;           // Object Declaration  
x.get( );       // Reads data
```

```
x.show( ); // Displays data  
}
```

## OUTPUT

Name : Balaji

Age : 26

Sex : M

Height : 4

Weight : 4.9

Name : Balaji

Age : 26 Years

Sex : M

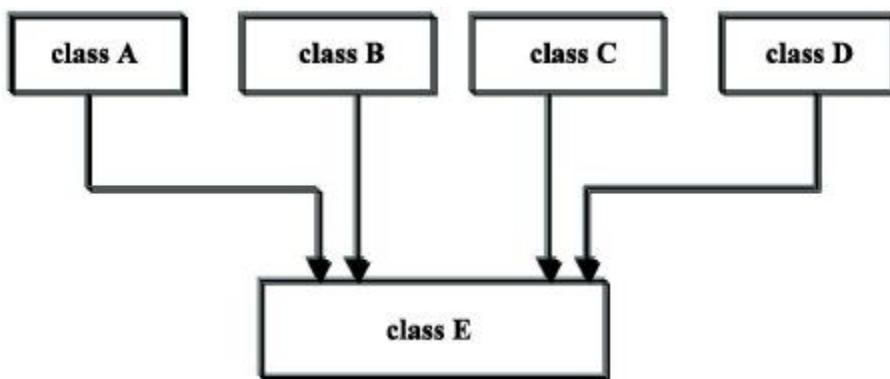
Height : 4 Feets

Weight : 4.9 Kg.

**Explanation:** In the above program, the class A1, A2, and A3 are declared. The member variables of all these classes are protected. The class A2 is derived from class A1. The class A3 is derived from class A2. Thus, the class A2 acts as derived class as well as base class. The function get( ) reads data through the keyboard and function show( ) displays data on the screen. Both the functions are invoked using object x of class A3.

## 9.7 MULTIPLE INHERITANCE

Multiple inheritance is the latest addition to the C++ language. When a class is derived from more than one class then this type of inheritance is called as **multiple inheritance**. A class can be derived by inheriting properties of more than one class. Properties of various predefined classes are transferred to single derived class. Figure 9.9 shows multiple inheritance.



**Fig.9.9** Multiple inheritance

### 9.7 Write a program to derive a class from multiple base classes.

```
// Multiple Inheritance //
```

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

```

# include <constream.h>

class A { protected : int a; }; // class A declaration
class B { protected : int b; }; // class B declaration
class C { protected : int c; }; // class C declaration
class D { protected : int d; }; // class D declaration

// class E : public A, public B, public C, public D

class E : public A,B,C,D // Multiple derivation
{
    int e;
    public :

        void getdata( )
    {
        cout <<"\n Enter values of a,b,c & d & e : ";
        cin >>a>>b>>c>>d>>e;
    }

    void showdata( )
    {

cout <<"\n a="<<a <<" b = "<<b <<" c = "<<c <<" d= "<<d <<" e= "<<e;
    }
};

void main ( )
{
    clrscr( );
    E x;
    x.getdata( ); // Reads data
    x.showdata( ); // Displays data
}

```

## OUTPUT

**Enter values of a,b,c & d & e : 1 2 4 8 16**

**a=1 b = 2 c = 4 d= 8 z= 16**

**Explanation:** In the above program, classes A, B, C, D, and E are declared each with one integer member variable. The class E has two-member functions, `getdata()` and `showdata()`. The `getdata()` is used to read integers through the keyboard and `showdata()` is used to display the contents on the screen. The class E is derived from the classes A, B, C, and D. The classes A, B, C, and D all together act as a base class. The derivation is carried out with the statement `class E:public A, B, C, and D.` The class members of A are publicly derived and members of other classes are privately derived. To derive all class members publicly, the statement would be as `class Z:public A, public B,public C,public D.` Remember that the meaning of both the statements is not the same. Most programmers refer to the second format as it reduces the code.

## 9.8 HIERARCHICAL INHERITANCE

We know that in inheritance, one class could be inherited from one or more classes. In addition, new members are added to the derived class. Inheritance also supports hierarchical arrangement of programs. Several programs require hierarchical arrangement of classes, in which derived classes share the properties of base class. Hierarchical unit shows top down style through splitting a compound class into several simple sub classes. Figure 1.16 given in Chapter “***Introduction to C++***”, is a perfect example of hierarchy of classes. The program based on hierarchical inheritance is illustrated below.

### 9.8 Write a program to show hierarchical inheritance.

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

class red
{
public:
    red ( ) {cout<<" Red   ";};
};

class yellow
{
public :
    yellow( ) { cout <<" Yellow "; }
};

class blue
{
public:
    blue ( ) { cout <<" Blue "; }
};

class orange : public red, public yellow
{
public :
    orange( ) { cout <<" = Orange "; }
};

class green : public blue, public yellow
{
public:
    green( ) { cout <<" = Green "; }

};

class violet : public red, public blue
{
public:
    violet( ) { cout <<" = Violet "; }
};
```

```

class reddishbrown : public orange, public violet
{
public:
    reddishbrown( ) { cout << " = Reddishbrown "; }
};

class yellowishbrown : public green, public orange
{
public:
    yellowishbrown( ) { cout << " = Yellowishbrown "; }
};

class bluishbrown : public violet, public green
{
public:
    bluishbrown( ) { cout << " = Bluishbrown "; }
};

void main( )
{
    clrscr( );
    reddishbrown r;
    endl(cout);

    bluishbrown b;
    endl(cout);

    yellowishbrown y;
    endl(cout);
}

```

## **OUTPUT**

**Red Yellow = Orange Red Blue = Violet = Reddishbrown**

**Red Blue = Violet Blue Yellow = Green = Bluishbrown**

**Blue Yellow = Green Red Yellow = Orange = Yellowishbrown**

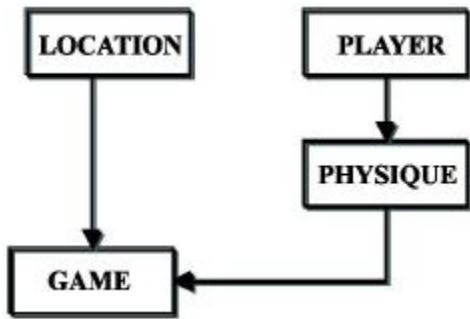
**Explanation:** For each colour separate class with constructor is declared. The classes `red`, `blue`, and `yellow` are base classes. The class `orange` is derived from classes `red` and `yellow`. The class `green` is derived from `blue` and `yellow`. The class `violet` is derived from `red` and `blue`.

The class `reddishbrown` is derived from classes `orange` and `violet`. The class `yellowishbrown` is derived from classes `green` and `orange` and lastly the class `bluishbrown` is derived from `violet` and `green`.

In function `main()`, objects of classes `reddishbrown`, `yellowishbrown`, and `bluishbrown` are declared. When objects are declared constructors are executed from base to derived classes. Constructor when executed displays the class name (colour name). The output shows names of different colours and resulting colour after their combination.

## 9.9 HYBRID INHERITANCE

The combination of one or more types of inheritance is known as **hybrid inheritance**. Sometimes, it is essential to derive a class using more types of inheritances. Figure 9.10 shows hybrid inheritance. In the diagram given below the class GAME is derived from two base classes i.e., LOCATION and PHYSIQUE. The class PHYSIQUE is also derived from class player.



**Fig. 9.10** Hybrid inheritance

### 9.9 Write a program to create a derived class from multiple base classes.

```
// Hybrid Inheritance //  
  
# include <iostream.h>  
# include <constream.h>  
  
class PLAYER  
{  
protected :  
char name[15];  
char gender;  
int age;  
};  
  
class PHYSIQUE : public PLAYER  
{  
protected :  
float height;  
float weight;  
};  
  
class LOCATION  
{  
protected :  
char city[10];  
char pin[7];  
};  
  
class GAME : public PHYSIQUE, LOCATION  
{
```

```

protected :
char game[15];
public :
void getdata( )
{
    cout <<" Enter Following Information\n";
    cout <<"Name : "; cin>> name;
    cout <<"Gender : "; cin>>gender;
    cout <<"Age : "; cin>>age;
    cout <<"Height : "; cin>>height;
    cout <<"Weight : "; cin>>weight;
    cout <<"City : "; cin>>city;
    cout <<"Pincode : "; cin>>pin;
    cout <<"Game : "; cin>>game;
}
void show( )
{
    cout <<"\n Entered Information";
    cout <<"\nName : "; cout<<name;
    cout <<"\nGender : "; cout<<gender;
    cout <<"\nAge : "; cout<<age;
    cout <<"\nHeight : "; cout<<height;
    cout <<"\nWeight : "; cout<<weight;
    cout <<"\nCity : "; cout<<city;
    cout <<"\nPincode : "; cout<<pin;
    cout <<"\nGame : "; cout<<game;
}
};

int main( )
{
    clrscr( );
    GAME G;
    G.getdata( );
    G.show( );
    return 0;
}

```

## OUTPUT

**Enter Following Information**

**Name : Mahesh**

**Gender : M**

**Age : 25**

**Height : 4.9**

**Weight : 55**

**City : Nanded**

**Pincode : 431603**

**Game : Cricket**

**Entered Information**

**Name : Mahesh**

**Gender : M**

**Age : 25**

**Height : 4.9**

**Weight : 55**

**City : Nanded**  
**Pincode : 431603**  
**Game : Cricket**

**Explanation:** In the above program, PLAYER, PHYSIQUE, LOCATION, and GAME classes are defined. All the data members of these four classes are protected. The class PHYSIQUE is derived from base class PLAYER. The class GAME is derived from PHYSIQUE and LOCATION i.e., the derived class GAME has two base classes PHYSIQUE and LOCATION. The class LOCATION is a separate class and not derived from any other class. The `getdata()` and `show()` are functions of class GAME which read and display data respectively. In `main()`, the object G of class GAME calls these functions one by one to read and write the data. The output of the program is shown at the end of the above program.

## 9.10 MULTIPATH INHERITANCE

When a class is derived from two or more classes, which are derived from the same base class such type of inheritance is known as **multipath inheritance**. Multipath inheritance consists of many types of inheritances such as multiple, multilevel and hierarchical as shown in Figure 9.11.

**Consider the following example:**

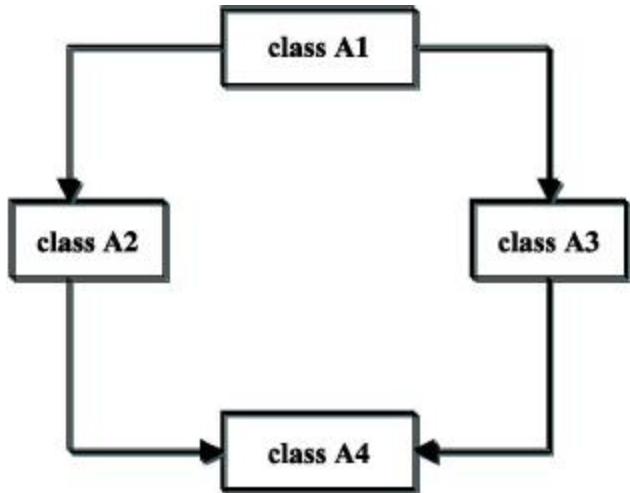
```
class A1
{
protected:
    int a1;
};

class A2 : public A1
{

protected:
    int a2;
};

class A3: public A1
{
protected:    int a3;
};

class A4: public A2,A3
{int a4; }
```



**Fig. 9.11** Ambiguity in classes

In the given example, class A2 and A3 are derived from the same base class i.e., class A1 (hierarchical inheritance). The classes A2 and A3 both can access variable a1 of class A1. The class A4 is derived from class A2 and class A3 by multiple inheritances. If we try to access the variable a1 of class A, the compiler shows error messages as given below.

- (a) Error PRG58.CPP 30: Member is ambiguous: 'A1::a1' and 'A1::a1'
- (b) Error PRG58.CPP 37: Member is ambiguous: 'A1::a1' and 'A1::a1'

In the above example, we can observe all types of inheritances i.e. multiple, multilevel and hierarchical. The derived class A4 has two sets of data members of class A1 through the middle base classes A2 and A3. The class A1 is inherited twice.

## 9.11 VIRTUAL BASE CLASSES

To overcome the ambiguity occurred due to multipath inheritance, C++ provides the keyword `virtual`. The keyword `virtual` declares the specified classes `virtual`. The example given below illustrates the `virtual` classes.

```

class A1
{
protected:
    int a1;
};

class A2 : virtual public A1      // virtual class declaration
{
protected:
    int a2;
};

class A3: virtual public A1      // virtual class declaration
{
protected:
    int a3;
};
  
```

```
class A4: public A2,A3
{
int a4;
};
```

When classes are declared as `virtual`, the compiler takes necessary precaution to avoid duplication of member variables. Thus, we make a class `virtual` if it is a base class that has been used by more than one derived class as their base class.

Let us consider an example that will give a clear idea of virtual classes as well as necessary care to be taken while declaring them.

(a) A base class cannot be specified more than once in a derived class.

```
class A
{ _____ };
_____ ;
```

```
class A1: A, A { ... }; // Illegal Declaration
```

Here, class `A` is specified twice. Hence, it is an illegal declaration.

(b) A base class can be indirectly passed to the derived class more than once.

```
class L: public A { ... }
class K: public A { ... }
class J: public L, public K { ... } // Legal declaration
```

In the above case (b), each object of class `J` will contain two sub-objects of class `A` through classes `L` and `K`.

(c) Case (b) causes some problems. To avoid duplication we can add the keyword `virtual` to a base class specifier.

**For example,**

```
class L : virtual public A { ... }
class K : virtual public A { ... }
class J : public K, public J { ... }
```

The `virtual` keyword always appears before the class name. `A` is now a virtual base class, and class `J` has only one sub-object of class `A`.

## 9.10 Write a program to declare virtual base classes. Derive a class using two virtual classes.

```
// VIRTUAL BASE CLASSES //

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

class A1
{
protected:
int a1;
};

class A2 : public virtual A1 // virtual declaration
{
protected :
int a2;
};
```

```

class A3 : public virtual A1 // virtual declaration
{
protected:
    int a3;
};

class A4 : public A2,A3 // virtual declaration
{
int a4;
public :

void get( )
{
cout <<"Enter values for a1, a2,a3 and a4 : ";
cin >>a1>>a2>>a3>>a4;
}

void put( )
{
cout <<"a1= "<<a1 <<" a2 = "<<a2 <<" a3 = "<<a3 <<"a4 = "<<a4;
}
};

void main( )
{
    clrscr( );
    A4 a;
    a.get( ); // Reads data
    a.put( ); // Displays data
}

```

## **OUTPUT**

**Enter values for a1 a2,a3 and a4 : 5 8 7 3  
a1= 5 a2 = 8 a3 = 7 a4 = 3**

**Explanation:** In the above program, the classes A1, A2, A3, and A4 are declared and each contains one protected member variable. The class A4 has two member functions get( ) and put( ). The get( ) function reads integers through the keyboard. The put( ) function displays the contents of the member variables on the screen. The classes A2 and A3 are derived from class A1. While deriving these two classes, the class A1 is declared as virtual as per the following statements:

- (a) class A2 : public virtual A1 // derivation of class A2
- (b) class A3 : public virtual A1 // derivation of class A3
- (c) class A4: public A2,A3 // derivation of class A4

The class A4 is derived from two classes A2 and A3 as per the statement (c). In function main( ), the object a of class A4 invokes the member function get( ) and put( ).

## **9.12 CONSTRUCTORS, DESTRUCTORS AND INHERITANCE**

The constructors are used to initialize member variables of the object and the destructor is used to destroy the object. The compiler automatically invokes constructors and destructors. The derived class does not require a constructor if the base class contains

zero-argument constructor. In case the base class has parameterized constructor then it is essential for the derived class to have a constructor. The derived class constructor passes arguments to the base class constructor. In inheritance, normally derived classes are used to declare objects. Hence, it is necessary to define constructor in the derived class. When an object of a derived class is declared, the constructors of base and derived classes are executed.

In inheritance, destructors are executed in reverse order of constructor execution. The destructors are executed when an object goes out of scope. To know the execution of constructor and destructor let us study the following program.

**9.11 Write a program to show sequence of execution of constructor and destructor in multiple inheritance.**

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

class A      // base class

{
public :
A ( )
{ cout <<"\n Zero-argument constructor of base class A"; }

~A( )

{ cout<<"\n Destructor of the class A"; }

};

class B  //  base class
{
public:
B( )
{ cout<<"\n Zero-argument constructor of base class B "; }

~B( ) { cout <<"\n Destructor of the class B " ; }

};

class C : public A, public B    // Derivation of class
{
public:
C( )
{ cout <<"\n Zero-argument constructor of derived class C"; }

~C( )   { cout<<"\n Destructor of the class C"; }

};
```

```
void main( )
{
clrscr( );
C objc; // Object declaration
}
```

## OUTPUT

**Zero-argument constructor of base class A**  
**Zero-argument constructor of base class B**  
**Zero-argument constructor of derived class C**  
**Destructor of the class C**  
**Destructor of the class B**  
**Destructor of the class A**

**Explanation:** In the above program, class A and B are two classes. The class C is derived from classes A and B. The constructors of base classes are executed first followed by derived class. The destructors of derived classes are executed first followed by base class.

## (1) BASE AND DERIVED CLASSES WITH CONSTRUCTOR

### 9.12 Write a program to declare both base and derived classes with constructors.

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

class I
{
public :

I( )
{
    cout <<"\n In base class constructor ";
}

class II : public I
{
public:
II( )      cout <<"\n In derived class constructor\n";
}

};

void main( )
{
    clrscr( );
    II i;
}
```

## OUTPUT

### In base class constructor

### In derived class constructor

**Explanation:** In the above program, both the base and derived classes contain constructors. When the object of derived class type is declared and constructors of both the classes are executed, the constructor of base class is executed first and then the constructor of derived class.

## (2) BASE CLASS WITH VARIOUS CONSTRUCTORS AND DERIVED CLASS WITH ONE CONSTRUCTOR

### 9.13 Write a program to declare multiple constructors in base class and single constructor in derived class.

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

class I
{
public :
    int x;

    I( )      { cout <<"\nZero argument base class constructor "; }

    I( int k)
    {
        cout <<"\nOne argument base class constructor";
    }

};

class II : public I
{
    int y;
public:

    II (int j)
    {
        cout <<"\nOne argument derived class constructor ";
        y=j;
    }
};

void main( )
{
    clrscr( );
    II i(2);
}
```

## OUTPUT

**Zero argument base class constructor**

**One argument derived class constructor**

**Explanation:** In the above program, the class I has zero and one argument constructor. The class II has only one argument constructor. The object i is declared with an integer. The zero argument constructor of base class and one argument constructor of derived class are executed.

## (3) BASE AND DERIVED CLASSES WITHOUT DEFAULT CONSTRUCTOR

**9.14 Write a program to declare base and derived class without default constructor.**

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

class I
{
public :
    int x;

    I ( int k)
    {
        x=k;
        cout <<"\nOne argument base class constructor";    }

};

class II : public I
{
    int y;
public:

II (int j) : I(j)
{
    cout <<"\nOne argument derived class constructor ";
    y=j;
}
};

void main( )
{
    clrscr( );
    II i(2);
}
```

## OUTPUT

**One argument base class constructor**

**One argument derived class constructor**

**Explanation:** In the above program, no default constructor is declared. In class II one argument constructor is declared and base class constructor is explicitly invoked. When object i is declared, one argument constructor of both the classes is executed. In the absence of explicit call of base class constructor the compiler will display the error message "Cannot find default constructor to initialize base class 'I'".

#### (4) CONSTRUCTORS AND MULTIPLE INHERITANCE

**9.15 Write a program to derive a class using multiple base classes. Observe the execution of constructor when object of derived class is declared.**

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

class I
{
public:
    I( ) { cout <<"\n Zero argument constructor of base class I "; }

};

class II
{
public:
    II( ) { cout<<"\n Zero argument constructor of base class II "; }

};

class III : public II,I
{
public:
    III( )
    { cout<<"\n Zero argument constructor of base class III "; }

};

void main( )
{
    clrscr( );
    III i;
}
```

#### OUTPUT

Zero argument constructor of base class II

### **Zero argument constructor of base class I**

### **Zero argument constructor of base class III**

**Explanation:** The classes I and II are base classes of derived class III. In function main(), i is an object of derived class III. The execution of constructors depends on the sequence given while deriving a class as per the following statement:

```
class III : public II, I
```

Here, the class II is the first base class and class I is the second base class and the execution sequence of constructors is shown in the output.

## **(5) CONSTRUCTORS IN MULTIPLE INHERITANCES WITH EXPLICIT CALL**

### **9.16 Write a program to derive a class using multiple base classes. Invoke the constructors of base classes explicitly.**

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

class I
{
public:
    I() { cout << "\n Zero argument constructor of base class I "; }

class II
{
public:
    II() { cout << "\n Zero argument constructor of base class II "; }

class III : public II, I
{
public:
    III() : II(), I()
    { cout << "\n Zero argument constructor of base class III "; }

void main()
{
    clrscr();
    III i;
}
```

### **OUTPUT**

**Zero argument constructor of base class II**

### **Zero argument constructor of base class I**

### **Zero argument constructor of base class III**

**Explanation:** In this program, in class III explicitly, the constructors of both the base classes II and I are invoked. The execution sequence can be observed in the output. The execution sequence of constructor depends on the sequence of base classes and not according to the explicit calls.

## **(6) MULTIPLE INHERITANCE AND VIRTUAL CLASS**

**9.17 Write a program to derive a class using multiple base classes. Invoke the constructors of base classes explicitly. Declare any one base class as virtual.**

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

class I
{
public:
    I( ) { cout <<"\n Zero argument constructor of base class I "; }

};

class II
{
public:

    II( ) { cout<<"\n Zero argument constructor of base class II "; }

};

class III : public II, virtual I
{
public:

    III( ) : II( ), I( )
    { cout<<"\n Zero argument constructor of base class III "; }

};

void main( )
{
    clrscr( );
    III i;
}
```

### **OUTPUT**

**Zero argument constructor of base class I**

**Zero argument constructor of base class II**

**Zero argument constructor of base class III**

**Explanation:** In this program, the base class I is declared as virtual class while deriving the class. The constructor of virtual class is executed first. The execution of constructor here is not according to the sequence of base class.

## (7) EXECUTION OF CONSTRUCTORS IN MULTILEVEL INHERITANCE

### 9.18 Write a program to derive a class using multilevel inheritance and observe the execution sequence of constructors.

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

class I
{
public:
    I( ) { cout <<"\n Zero argument constructor of base class I "; }
};

class II : public I
{
public:
    II( ) { cout <<"\n Zero argument constructor of base class II "; }
};

class III : public II
{
public:
    III( ) { cout <<"\n Zero argument constructor of base class III "; }
};

void main( )
{
    clrscr( );
    III ii;
}
```

#### OUTPUT

**Zero argument constructor of base class I**

**Zero argument constructor of base class II**

**Zero argument constructor of base class III**

**Explanation:** In this program, the class II is derived from class I. The class III is derived from class II. The class II is base as well as derived class. In function main ( ), ii is an object of class III. The constructors are executed from base to derived classes as shown in the output. Table 9.3 shows order of execution of constructors.

Table 9.3 Execution sequences of constructors

| Statements          | Sequence of execution                                              | Remarks            |
|---------------------|--------------------------------------------------------------------|--------------------|
| Class II : public I | I( ) – Base class constructor<br>II( ) - Derived class constructor | Single inheritance |

|                                                                                                      |                                                                                                                                                                                                                                         |                                                  |
|------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------|
| Class III: public I, II                                                                              | I ( ) – Base class constructor<br>II ( )- Base class constructor<br>III ( )- Derived class constructor                                                                                                                                  | Multiple inheritance                             |
| Class III: public I, virtual<br>II<br><br>class II : public I<br>{};<br><br>class III: public II {}; | II ( ) – Virtual class constructor<br>I ( ) – Base class constructor<br>III ( ) – Derived class constructor<br>I ( ) – First base class constructor<br>II ( ) – second base class<br>constructor<br>III ( ) – derived class constructor | Multiple inheritance<br><br>Multiple inheritance |

### 9.13 OBJECT AS A CLASS MEMBER

Properties of one class can be used in another class using inheritance or using object of a class as a member in another class. Declaring object as a class data member in another class is also known as **delegation**. When a class has an object of another class as its member, such class is known as container class.

In inheritance, derived classes can use members of base class. Here, the derived class is a kind of base class. The programmer can also add new members to the derived class.

In delegation, the class is composed from objects of another classes. The composed class uses properties of other classes through their objects. This kind of relationship is known as **has-a-relationship** or **containership**.

```
class I
{
    ***** // members
    *****
}

class II
{
    I J; // object of class I as member
    *****
}
```

### 9.19 Write a program to use object of one class in another class as a member.

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

```
class I
{
public:
    int x;

I()
```

```

    {
        cout <<"\n Constructor of class I";
        x=20;
    }
};

class II
{
public:
    int k;
    I y;

II( )
{
    k=30;
    cout <<"\n Constructor of class II";
}
}

void show( ) { cout <<"\n x= "<<y.x <<" k = "<<k; }
};

void main( )
{
    clrscr( );
    II ii;
    ii.show( );
}

```

## **OUTPUT**

**Constructor of class I**

**Constructor of class II**

**x= 20 k = 30**

**Explanation:** In the above program, class II contains integer k and object y of class I. In function main(), object ii is an object of class II. The constructor of class I is executed first because when compiler reaches class II, it finds an object of class I. We know that object declaration always executes constructors of that class. Thus, object of class I is declared and constructor of class I is executed. The constructor of class II is executed and variable of class II is initialized. The member function show() displays the contents of x and k. The contents of class I member is obtained from object y. The dot (.) operator is used to access the elements of class I.

## **9.20 Write a program to access member variables of base class using object, scope access operator and direct.**

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

```
class A1
{
public :
    char name[15];
    int age;
```

```

};

class A2 : public A1
{
private :
A1 a;
float height;
float weight;
public :

A2( )
{
clrscr( );

cout <<"Access Using Scope Access operator\n";

cout <<"Name      : "; cin>>A1::name;
cout <<"Age       : "; cin>>A1::age;

cout <<"Access Using object of the class\n";

cout <<"Name      : "; cin >>a.name;
cout <<"Age       : "; cin >>a.age;
cout <<"Access Using direct member variables\n";

cout <<"Name      : "; cin >>name;
cout <<"Age       : "; cin >>age;

cout <<"Height   : "; cin >>height;
cout <<"Weight   : "; cin >>weight;
}

~A2( )

{
cout <<"\nDisplay using Scope Access operator\n";

cout <<"\nName   : " <<A1::name;
cout <<"\nAge    : " <<A1::age;

cout <<"\nDisplay Using object of the class\n";

cout <<"\nName   : " <<a.name;
cout <<"\nAge    : " <<a.age;

cout <<"\nAccess Using direct member variables\n";
}

```

```

cout <<"\nName : " <<name;
cout <<"\nAge : " <<age;

cout <<"\nHeight : " <<height;
cout <<"\nWeight : " <<weight;
}
};

void main( )
{
A2 x;
}

```

### **OUTPUT**

**Name : Ajay**

**Age : 21**

**Access Using object of the class**

**Name : Amit**

**Age : 20**

**Access Using direct member variables**

**Name : Arun**

**Age : 19**

**Height : 5.5**

**Weight : 31**

**Display using Scope Access operator**

**Name : Arun**

**Age : 19**

**Display Using object of the class**

**Name : Amit**

**Age : 20**

**Access Using direct member variables**

**Name : Arun**

**Age : 19**

**Height : 5.5**

**Weight : 31**

**Explanation:** In the above program, A1 and A2 are two classes. The class A2 is derived from class A1. Class A1 has two public member variables. Class A2 has three private members. One of its member is an object of class A1 i.e., object a. The object a holds its separate set of member variables of class A1. There are three ways to access member variables of base class A1. Table 9.4 describes status of access of member variables of class A1 through derived class A2 when we use access specifiers such as private, public and protected.

**Table 9.4** Access specifiers

| Access (using)           | Public   | Private      | Protected    |
|--------------------------|----------|--------------|--------------|
| Scope Access :: operator | Possible | Not Possible | Possible     |
| Object                   | Possible | Not Possible | Not Possible |
| Direct                   | Possible | Not Possible | Possible     |

(1) The first way is to use scope access operator as given below:

```
cin>>A1::name;
cin>>A1::age;
```

In the above statements, A1 is a class name and name and age are member variables of class A1. When access specifier is public or protected the above statements are valid.

(2) The second method is to use objects of the base class as given below:

```
cin >>a.name;
cin >>a.age;
```

In the above statements, member variables of class A1 are accessed using objects of the same class, which is the member of the derived class. This is possible only when the member variables of class A1 are public and not possible if the member variables of class A1 are protected or private.

(3) The third method uses directly the member variables. Direct access is possible when access specifier is public or protected, as per the following statements.

```
cin >>name;
cin >>age;
```

**9.21 Write a program to derive a class from two base classes. Use objects of both the classes as member variables for derived class. Initialize and display the contents of classes using constructor and destructor.**

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

```
class A
{
public :
    int a1;
};
```

```
class B
{
public:
    int b1;
};
```

```
class AB
{
public :
    A a;
    B b;
public :
```

```
AB ( )
```

```

    {
        a.a1=65;
        b.b1=66;
        cout <<"a1 = "<<a.a1 <<" b1 = "<<b.b1;
    }

    ~AB( ){ };
};

int main( )
{
    clrscr( );
    AB ab;

    return 0;
}

```

## OUTPUT

**a1=65 b1=66**

**Explanation:** In the above program, class A and class B are declared each with one integer. Class AB is declared which contains objects of A and B classes as member variables. The constructor of class AB initializes member variables of class A and B. The constructor also displays contents on the screen. Finally, the destructor destroys the object.

## 9.14 ABSTRACT CLASSES

When a class is not used for creating objects it is called as abstract class. The abstract class can act as a base class only. It is a lay out abstraction in a program and it allows a base on which several levels of inheritance can be created. The base classes act as foundation of class hierarchy. An abstract class is developed only to act as a base class and to inherit and no objects of these classes are declared. An abstract class gives a skeleton or structure, using which other classes are shaped. The abstract class is central and generally present at the starting of the hierarchy. The hierarchy of classes means chain or groups of classes being involved with one another. In the last program, class A is an abstract class because no instance (object) of class A is declared.

## 9.15 QUALIFIER CLASSES AND INHERITANCE

The following program explains the behaviour of qualifier classes and the classes declared within them with inheritance.

### 9.22 Write a program to create derived class from the qualifier class.

```

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

class A
{
public:
    int x;
}

```

```

A ( ) {}

    class B
    {

        public:
        int y;
        B( ) {}
    };

};

class C : public A,A::B
{
    public:
    int z;

    void show( )
    {
        cout << endl << "x = " << x << " y = " << y << " z = " << z;
    }

    C (int j,int k, int l)
    {
        x=j;
        y=k;
        z=l;
    }
};

void main( )
{
    clrscr( );
    C c(4,7,1);
    c.show( );
}

```

### **OUTPUT**

**x = 4 y =7 z = 1**

**Explanation:** In the above program, class B is defined inside class A. The class A is a qualifier class of class B. The class C is inherited from the classes A and B. In the statement class C: public A, A::B, class C is inherited from A and B. To access class B, it is preceded by the qualifier class A and scope access operator. If we mention only class A, class B won't be considered for inheritance. Similarly, if we mention only class B, the qualifier class A will not be considered for inheritance.

### **9.16 COMMON CONSTRUCTOR**

When a class is declared, constructor is also declared inside the class in order to initialize data members. It is not possible to use a single constructor for more classes. Every class has its own constructor and destructor with the same name as class. When a class is derived from another then it is possible to define a constructor in derived class and data members of both base and derived classes can be initialized. It is not essential to declare constructor

in a base class. Thus, the constructor of the derived class works for its base class and such constructors are called as **common constructors**.

### 9.23 Write a program to initialize member variables of both base and derived classes using a constructor of derived class.

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

class A
{
protected:
    int x;
    int y;

};

class B : private A
{
public:
    int z;

B( ) {    x=1,y=2,z=3;
    cout <<"x= " <<x << " y =" <<y << " z=" <<z; }

};

void main( )
{
    clrscr( );
    B b;
}
```

#### OUTPUT

**x= 1 y =2 z=3**

**Explanation:** In the above program, class A and class B are declared. Class B is derived from class A. The constructor of class B initializes member variables of both classes. Hence, it acts as a common constructor of both base and derived class.

## 9.17 POINTERS AND INHERITANCE

The **private** and **public** member variables of a class are stored in successive memory locations. A pointer to public member variable gives us access to private member variables. The same is true for derived class. The member variables of base class and derived class are also stored in successive memory locations. The following program explains the mechanism of accessing private data members of the base class using the address of public member variable of derived class using pointer. Here, no member functions are used.

### 9.24 Write a program to access private member variables of base class using pointers.

```

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

class A
{
private:
    int x;
    int y;

public:
    A( ) {
        x=1;
        y=2;
    }
};

class B : private A
{
public:
    int z;

B( ) { z=3; }

};

void main( )
{
    clrscr( );
    B b; // object declaration
    int *p;// pointer declaration
    p=&b.z; // address of public member variabe is stored in pointer

    cout<<endl<<" Address of z : "<<(unsigned)p <<" " <<"Value of z :"<<*p;
    p--; // points to previous location
    cout<<endl<<" Address of y : "<<(unsigned)p <<" " <<"Value of y :"<<*p;
    p--;
    cout<<endl<<" Address of x : "<<(unsigned)p <<" " <<"Value of x :"<<*p;
}

```

## OUTPUT

**Address of z : 65524 Value of z :3**

**Address of y : 65522 Value of y :2**

**Address of x : 65520 Value of x :1**

**Explanation:** In the above program, class A contains two private member variables x and y. The constructor initializes the member variables. Class B is derived from class A. The class B has one public member variable. In function main( ), b is an object of class B. The pointer p is an integer pointer. The address of member variable z of derived class B is assigned to pointer p. By applying decrease operation, we get the previous memory locations where member variables of base classes are stored. The values of all member variables with their addresses are displayed.

## 9.18 OVERLOADING MEMBER FUNCTION

The derived class can have the same function name as base class member function. An object of the derived class invokes member functions of the derived class even if the same function is present in the base class.

### 9.25 Write a program to overload member function in base and derived class.

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

class B
{
public:

    void show( )
    {
        cout <<"\n In base class function ";
    }
};

class D : public B
{
public:

    void show( )
    {
        cout <<"\n In derived class function";
    }
};

int main( )
{
    clrscr( );
    B b;           // b is object of base class
    D d;           // d is object of derived class

    b.show( );      // Invokes Base class function
    d.show( );      // Invokes Derived class function
    d.B::show( );   // Invokes Base class function
    return 0;
}
```

### OUTPUT

In base class function  
In derived class function  
In base class function

**Explanation:** In this program, class D is derived from class B. Both the classes have the same function `show()` as member function. In function `main()`, objects of both B and D classes are declared. The object b invokes the member function `show()`. The object b is object of base class, hence, it invokes the member function `show()` of base class. In addition, the objects of base class cannot invoke the member functions of derived class, because the base class does not have information about the classes derived under it. The object d of derived class invokes the function `show()`. When function with same name and argument list are present in both the base and derived classes, the objects of derived class give first priority to function of its own class. Thus, the statement `d.show()` invokes function of derived class. To invoke the function of base class with the object of derived class, the class name and scope access operator are preceded before the matching function name. From the above program, it is also clear that we can declare objects of both base and derived classes. The objects of both base and derived classes are independent of each other.

## 9.19 ADVANTAGES OF INHERITANCE

- (1) The most frequent use of inheritance is for deriving classes using existing classes, which provides reusability. The existing classes remain unchanged. By reusability, the development time of software is reduced.
- (2) The derived classes extend the properties of base classes to generate more dominant object.
- (3) The same base classes can be used by a number of derived classes in class hierarchy.
- (4) When a class is derived from more than one class, all the derived classes have the same properties as that of base classes.

## 9.20 DISADVANTAGES OF INHERITANCE

- (1) Though object-oriented programming is frequently propagandized as an answer for complicated projects, inappropriate use of inheritance makes a program more complicated.
- (2) Invoking member functions using objects create more compiler overheads.
- (3) In class hierarchy various data elements remain unused, the memory allocated to them is not utilized.

## SUMMARY

- (1) Inheritance is one of the most useful and essential characteristics of object-oriented programming language. Inheritance allows the programmer to utilize the previously defined classes with newer ones. The new class is assembled using properties of existing classes.
- (2) The procedure of developing a new class from an old class is termed as inheritance.
- (3) The new class is termed as derived class and the old class is called base class.
- (4) Single inheritance When a new class is derived from only one base class such type of inheritance is called as single inheritance.
- (5) Multilevel inheritance The procedure of deriving a class from derived class is named as multilevel inheritance.
- (6) Multiple inheritance or Hierarchical inheritance When a class is derived from more than one class then this type of inheritance is called as multiple inheritance or hierarchical inheritance.

(7) Hybrid inheritance When a class is derived from another one or more base classes, this process is termed as hybrid inheritance.

(8) When classes are declared as `virtual`, the compiler takes essential caution to avoid duplication of member variables. Thus, we make a class `virtual` if it is a base class that has been used by more than one derived class as their base class.

(9) Execution of constructor in inheritance The execution of constructor takes place from base class to derived class.

(10) Execution of destructor in inheritance The execution of destructors is in opposite order as compared to constructors i.e., from derived class to base class.

(11) Public If member variables of a class are public, any function can access them. In C++, members of struct and union are by default public.

(12) Private If member variables of a class are private, member functions and friends can only access them, if they are declared in the same class. Members of a class are by default private.

(13) Protected If member variables of a class are protected, its scope is the same as for private. In addition, member functions and friends can use the member classes derived from the declared class but only in objects of the derived type.

(14) When a class is not used for creating objects, it is called as abstract class.

(15) The constructor of the derived class works for its base class; such constructors are called as common constructors.

(16) The derived class can have the same function name as the base class member function. An object of the derived class invokes member functions of the derived class even if the same function is present in the base class.

(17) Properties of one class can be used in another class using inheritance or using objects of a class as a member in another class. Declaring objects as a class data member in another class is also known as delegation. When a class has an object of another class as its member, such class is known as container class.

## EXERCISES

### [A] Answer the following questions.

- (1) What do you mean by inheritance?
- (2) What do you mean by base class and derived class?
- (3) Describe various types of inheritances with examples.
- (4) What is the difference between single and multilevel inheritance?
- (5) What is the difference between multilevel and hybrid inheritance?
- (6) How are constructors and destructors executed in multilevel inheritance?
- (7) What is the use of `virtual` keyword?
- (8) What do you mean by `virtual` classes?
- (9) What are `abstract` classes?
- (10) Describe the use of public, private and protected access specifiers.
- (11) What are `nested` classes?
- (12) Explain the mechanism for accessing private member of the base class using pointers.
- (13) What is the difference between private and protected access specifiers?
- (14) Do you think that accessing private data using pointers is a limitation of encapsulation?
- (15) How do structure and class provide inheritance differently?

- (16) What is the difference between private and protected inheritance?
- (17) What are the advantages and disadvantages of inheritance?
- (18) What do you mean by object delegation?
- (19) What is a common constructor?
- (20) Explain hierarchical inheritance.

**[B] Answer the following by selecting the appropriate option.**

- (1) What will be the output of the following program? class A { public: int a;

```
class A
{ public:
    int a;
    A( ) {a=10; }
};
```

```
class B : public A
{ public:
    int b;
    B( ) { b=20; }
    ~ B( ) { cout <<"\n a= "<<a <<" b = "<<b; }
};
```

void main ( ) { B( ); }

(a) a = 10 b = 20

(b) a = 20 b = 10

(c) a = 30 b = 30

(d) none of the above

- (2) Identify the access specifier

(a) public

(b) virtual

(c) void

(d) class

- (3) An object a cannot access the variable class A

```
class A
{ public:
    int a;
    private:
    int b;
    public:
    A( ){a=10,b=20; }
};
```

void main( ){ A a; }

(a) b

(b) a

(c) both (a) and (b)

(d) both a and b are accessible

- (4) Private data members of a class can be accessed by

(a) public member functions of the same class

(b) directly by the object

(c) private member function of the same class

(d) none of the above

(5) In multilevel inheritance, the middle class acts as

(a) base class as well as derived class

(b) only base class

(c) only derived class

(d) none of the above

(6) In single inheritance, constructors are executed from

(a) base class to derived class

(b) derived class to base class

(c) both (a) and (b)

(d) none of the above

(7) In the following program object of which class can access all member variables?

```
struct A { int x; };
```

```
struct B : A { int y; };
```

```
struct C : B { int z; };
```

```
struct D : C { int k; };
```

(a) object of class D

(b) object of class B

(c) object of class C

(d) object of class A

The protected keyword allows

(a) derived class to access base class members directly

(b) prevents direct access to public members

(c) allows objects to access private members

(d) all of the above

(9) The class is declared `virtual` when

(a) two or more classes involved in inheritance have common base class

(b) more than one class is derived

(c) we want to prevent a base class from inheritance

(d) none of the above

(10) The ambiguity of members normally occurs in

(a) single inheritance

(b) multilevel inheritance

(c) multiple inheritance

(d) none of the above

(11) In the following program class A is an

```
class A { int x; };
```

```
class B : A { int y; };
```

```
void main () { B b; }
```

(a) abstract class

(b) virtual class

(c) derived class

(d) none of the above

(12) Class A is a base class of class B. The relationship between them is

(a) kind of relationship

(b) has a relationship

(c) is a relationship

(d) none of the above

**[C] Attempt the following programs.**

(1) Write a program to define three classes A, B and C. Each class contains private data members. Derive class C from A and B by applying multiple inheritance. Read and display the data using constructors and destructors.

(2) Write a program to declare classes X, Y and Z. Each class contains one character array as a data member. Apply multiple inheritances. Concatenate strings of classes X and Y and store it in the class Z. Show all the three strings. Use constructors and destructors.

(3) Write a program to calculate the salary of a medical representative based on the sales. Bonus and incentives to be offered to him will be based on total sales. If the sale exceeds Rs.1,50,000/- follow the particulars of Column (1) otherwise (2). Apply conditional constructor and destructor.

| Column 1                  | Column 2                 |
|---------------------------|--------------------------|
| Basic = Rs. 3000          | Basic = Rs. 3000         |
| HRA = 20% of basic        | HRA = 20% of basic       |
| DA = 110% of basic        | DA = 110% of basic       |
| Conveyance = Rs.500       | Conveyance = Rs.500      |
| Incentives = 10% of sales | Incentives = 5% of sales |
| Bonus = Rs. 1500          | Bonus = Rs. 1000         |

(4) Write a program to calculate energy bill. Read the starting and ending meter reading. The charges are as given below:

| No. of units consumed | Rates in (Rs.) |
|-----------------------|----------------|
| 200 - 500             | 4.50           |

|               |      |
|---------------|------|
| 100 - 200     | 3.50 |
| Less than 100 | 2.50 |

(5) The price of trophy depends on the type of material from which it is created. The following table gives the list of prizes. Write a program to input serial number of the material. In addition, display the material and its prize.

| Serial no. | Material | Prize of trophy |
|------------|----------|-----------------|
| 01         | Gold     | Rs. 20,000/-    |
| 02         | Silver   | Rs. 9,500/-     |
| 03         | Steel    | Rs. 5,000/-     |
| 04         | Bronze   | Rs. 500/-       |
| 05         | Pewter   | Rs. 750/-       |

(6) A newspaper agent pays two variant rates for the delivery of newspapers. A delivery boy can earn Rs.2/- on a morning delivery but only Re. 1.50/- on an evening delivery. Boys are salaried either for a morning delivery or for an evening paper round but not for both. Write a program to read (input) the code for round (0 for morning and 1 for evening) and the number of total paper rounds in one week done by a boy and compute his earnings and display it.

(7) The postage for ordinary post is Rs. 2/- for the first 15 grams and Re. 1 for each additional 10 grams. Write a program to calculate the charge of the postage for a post weighing N grams. Read the weights of N packets and display the total amount of postage.

(8) Declare a class of vehicle. Derived classes are two-wheeler, three-wheeler, and four-wheeler. Display the properties of each type of vehicle using member functions of classes.

(9) Create classes country, state, city and village. Arrange these classes in hierarchical manner.

#### [D] Find bugs in the following programs.

(1)

```

struct A { int x; };
struct B { int y; };
class C : public A, B {};

void main ()
{
    C c;
    c.x=20;
    c.y=30;
}

```

(2)

```

struct A { int x; };
struct B : A { int y; };
struct C : A { int z; };

```

```

struct D : B,C { int k; };
void main ( )
{
    D d;
    d.x=20;
}

(3)
struct A
{
    int x;

    struct B { int y; } b;
};

struct C : A::B,A
{
    int z;
};

void main ( )
{
    C c;
    c.x=40;
    c.y=20;
}

```

(4)

```

struct A { int x; };
class B : A
{
    A a;
};

void main ( )
{
    B b;
    b.x=20;
    b.a.x=30;
}

```

# 10

CHAPTER

# Pointers and Arrays

C  
H  
A  
P  
T  
E  
R  
O  
U  
T  
L  
I  
N  
E

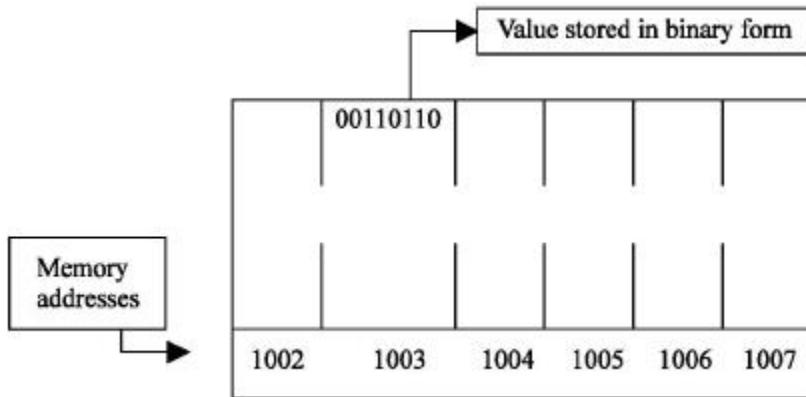
- 10.1 Introduction
- 10.2 Pointer Declaration
- 10.3 Void Pointers
- 10.4 Wild Pointers
- 10.5 Pointer to Class
- 10.6 Pointer to Object
- 10.7 The `this` Pointer
- 10.8 Pointer to Derived Classes and Base Classes
- 10.9 Pointer to Members
- 10.10 Accessing Private Members with Pointers
- 10.11 Direct Access to Private Members
- 10.12 Address of Object and Void Pointers
- 10.13 Arrays

- 10.14 Characteristics of Arrays
- 10.15 Initialization of Arrays Using Functions
- 10.16 Arrays of Classes

## 10.1 INTRODUCTION

Most of the learners feel that pointer is a puzzling topic. However, pointers can make programs quicker, straightforward and memory efficient. C/C++ gives more importance to pointers. Hence, it is important to know the operation and applications of pointers. Pointers are used as tool in C/C++. Like C, in C++ variables are used to hold data values during program execution. Every variable when declared occupies certain memory locations. It is possible to access and display the address of memory location of variables using ‘&’ operator. Memory is arranged in series of bytes. These series of bytes are numbered from zero onwards. The number specified to a cell is known as memory address. Pointer variable stores the memory address of any type of variable. The pointer variable and normal variable should be of the same type. The pointer is denoted by (\*) asterisk symbol.

A byte is nothing but combination of eight bits as shown in Figure 10.1. The binary numbers 0 and 1 are known as bits. Each byte in the memory is specified with a unique (matchless) memory address. The memory address is an unsigned integer starting from zero to uppermost addressing capacity of the microprocessor. The number of memory locations pointed by a pointer depends on the type of pointer. The programmer should not worry about the addressing procedure of variables .The compiler takes the procedure itself. The pointers are either 16 bits or 32 bits long.



**Fig. 10.1** Memory representation

The allocation of memory during program run-time is called as *dynamic memory allocation*. Such type of memory allocation is essential for data structures and can efficiently handle them using pointers. Another reason to use pointers is in array. Arrays are used to store more values. Actually, the name of array is a pointer. One more reason to use pointers is command-line arguments. Command line arguments are passed to programs and are stored in an array of pointers `argv[]`. The command line arguments and dynamic memory allocation are illustrated in the forthcoming chapters.

## POINTER

A pointer is a memory variable that stores a memory address. Pointers can have any name that is legal for other variables and it is declared in the same fashion like other variables but it is always denoted by '\*' operator.

### (1) FEATURES OF POINTERS

- (1) Pointers save the memory space.
- (2) Execution time with pointer is faster because data is manipulated with the address i.e., direct access to memory location.
- (3) The memory is accessed efficiently with the pointers. The pointer assigns the memory space dynamically that leads you to reserve a specific bytes in the memory.
- (4) Pointers are used with data structures. They are useful for representing two-dimensional and multi-dimensional arrays.
- (5) In C++, a pointer declared to a base class could access the objects of derived class. Whereas a pointer to derived class cannot access the objects of base class. The compiler will generate an error message cannot convert '`A*`' to '`B *`' where `A` is the base class and `B` is the derived class.

## 10.2 POINTER DECLARATION

Pointer variables can be declared as below:

**Example**

```
int *x;  
float *f;  
char *y;
```

(1) In the first statement ‘x’ is an integer pointer and it informs to the compiler that it holds the address of any integer variable. In the same way, ‘f’ is a float pointer that stores the address of any float variable and ‘y’ is a character pointer that stores the address of any character variable.

(2) The **indirection operator** (\*) is also called the **deference operator**. When a pointer is dereferenced, the value at that address stored by the pointer is retrieved.

(3) Normal variable provides direct access to their own values whereas a pointer indirectly accesses the value of a variable to which it points.

(4) The **indirection operator** (\*) is used in two distinct ways with pointers, declaration and deference.

(5) When a pointer is declared, the star indicates that it is a pointer, not a normal variable.

(6) When the pointer is dereferenced, the indirection operator indicates that the value at that memory location stored in the pointer is to be accessed rather than the address itself.

(7) Also note that \* is the same operator that can be used as the multiplication operator. The compiler knows which operator to call, based on context.

(8) The ‘&’ is the address operator and it represents the address of the variable. The address of any variable is a whole number. The operator ‘&’ immediately preceding the variable returns the address of the variable. In the example given below ‘&’ is immediately preceding the variable ‘num’ which provides address of the variable.

### 10.1 Write a program to display the address of the variable.

```
# include <iostream.h>  
# include <conio.h>  
  
main( )  
{  
    int n;  
    clrscr( );  
    cout <<"Enter a Number = ";
```

```

    cin >>n;
    cout <<"Value of n = "<<n;
    cout <<"Address of n= " <<(unsigned)&n;
    getch( );
}

```

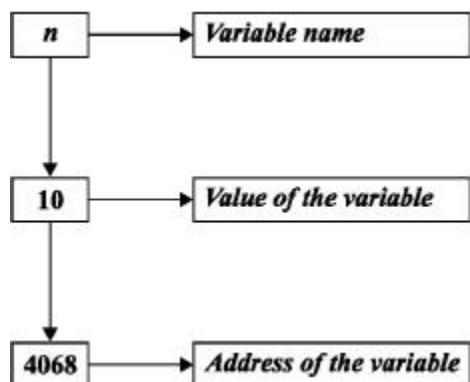
**OUTPUT :**

**Enter a Number = 10**

**Value of n = 10**

**Address of n=4068**

**Explanation:** The memory location of a variable is system dependent. Hence, address of a variable cannot be predicted immediately. In the above example, the address of the variable ‘n’ observed is 4068. In Figure 10.2 three blocks are shown related to the above program. The first block contains variable name. The second block represents the value of the variable. The third block is the address of the variable ‘n’ where 10 is stored. Here, 4068 is the memory address. The address of variable depends on various things for instance memory model, addressing scheme and present system settings.



**Fig. 10.2** Variable and its memory address

**10.2 Write a program to declare a pointer. Display the value and address of the variable using pointers.**

```

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

void main( )
{
    int *p;
    int x=10;
    p=&x;
}

```

```

clrscr( );
printf ("\n  x=%d  &x=%u \t(Using printf( ))",x,p);

cout  <<"\n  x="<<x << "  &x="  <<&x <<"    (Using cout( ))";
printf ("\n  x=%d  &x=%u \t(Using pointer)",*p,p);
cout  <<"\n *p="<<*p <<"\t&p=" <<p <<"\t(Contents of pointer)";
}

```

## OUTPUT

```

x = 10 &x = 65524    (Using printf( ))
x = 10 &x = 0x8f94fff4 (Using cout( ))
x = 10 &x = 65524    (Using pointer)
*p = 10 &p = 0x8f94fff4 (Contents of pointer)

```

**Explanation:** In the above program, `*p` is an integer pointer and `x` is an integer variable. The address of variable `x` is assigned to pointer `p`.

Ampersand (`&`) operator preceded with variable displays the memory location of that variable. The `printf( )` and `cout( )` statements display address in different formats. The `printf( )` statement displays the address as an unsigned integer whereas the `cout( )` statement displays the address in hexadecimal format. We can access the contents of variable `x` using pointer `p`. Output of the program is shown above.

### 10.3 Write a program to display memory address of a variable. Typecast the memory address from hexadecimal to unsigned integer.

```

#include<iostream.h>
#include<conio.h>
void main( )
{
    int a;
    float b;
    char c;
    clrscr( );
    cout<<"\n Enter integer number = ";
    cin>>a;
    cout<<"\n Enter float number     = ";
    cin>>b;
    cout<<"\n Enter character          = ";
    cin>>c;
    cout<<"\nThe entered int      is = "<<a;
    cout<<"\nThe entered float   is = "<<b;

    cout<<"\nThe entered char    is = "<<c;
    cout<<"\n\nThe entered number is stored at location =
"<<(unsigned)&a;
    cout<<"\n\nThe entered float  is stored at location=
"<<(unsigned)&b;

```

```
cout<<"\nThe entered char    is stored at location =  
"<<(unsigned)&c;  
getch( );  
}
```

## OUTPUT

```
Enter integer number = 34  
Enter float number  = 343.34  
Enter character   = g  
The entered int is = 34  
The entered float is = 343.339996  
The entered char is = g  
  
The entered number is stored at location = 4096  
The entered float is stored at location= 4092  
The entered char is stored at location = 4091
```

**Explanation:** In the above program, variables `a`, `b` and `c` of `int`, `float` and `char` type are declared. The entered values with their addresses are displayed. The addresses of the variables are displayed by preceding ampersand (`&`) operator with variable name. The typecasting syntax `(unsigned)` is done to typecast hexadecimal address into unsigned integer. Here, you can observe the addresses in descending order. This is because all automatic variables are stored in stack. The stack always starts from top to lower memory addresses.

## 10.3 VOID POINTERS

Pointers can also be declared as void type. Void pointers cannot be dereferenced without explicit type conversion. This is because being void the compiler cannot determine the size of the object that the pointer points to. Though void pointer declaration is possible, void variable declaration is not allowed. Thus, the declaration `void p` will display an error message “Size of ‘p’ is unknown or zero” after compilation.

It is not possible to declare void variables like pointers. Pointers point to an existing entity. A void pointer can point to any type of variable with proper typecasting. The size of void pointer displayed will be two. When pointer is declared as void, two bytes are allocated to it. Later using typecasting, number of bytes can be allocated or de-allocated. Void variables cannot be declared because memory is not allocated to them and there is no place to store the address. Therefore, void variables cannot actually serve the job they are made for.

**10.4 Write a program to declare a void pointer. Assign address of int, float, and char variables to the void pointer using typecasting method. Display the contents of various variables.**

```
// void pointers //
# include <stdio.h>
# include <iostream.h>
# include <conio.h>

int p;
float d;
char c;

void *pt = &p;           // pt points to p

void main (void)
{
    clrscr( );
    *(int *) pt = 12;
    cout <<"\n p="<<p;
    pt = &d;           // pt points to d
    *(float *)pt = 5.4;
    cout <<"\n r="<<d;
    pt=&c;           // pt points to c
    *(char*)pt='S';
    cout <<"\n c="<<c;
}
```

## **OUTPUT**

**p=12**

**r=5.4**

**c=S**

**Explanation:** In the above example, variables p, d, and c are variables of type int, float, and char respectively. Pointer pt is a pointer of type void. These entire variables are declared before main( ). The pointer is initialized with the address of integer variable p i.e., the pointer p points to variable x.

The statement \*(int \*) pt = 12 assigns the integer value 12 to pointer pt i.e., to a variable p. The contents of variable p are displayed using the succeeding statements. The declaration \*(int \*) tells the compiler the value assigned is of integer type. Thus, assignment of float and char type is carried out. The statements \*(int \*) pt = 12, \*(float \*) pt =

5 . 4 and `* (char*) pt='S'` help the compiler to exactly determine the size of data types.

## 10.4 WILD POINTERS

Pointers are used to store memory addresses. An improper use of pointers creates many errors in the program. Hence, pointers should be handled cautiously. When pointer points to an unallocated memory location or to data value whose memory is deallocated, such a pointer is called as wild pointer. The wild pointer generates garbage memory location and pendent reference. When a pointer pointing to a memory location gets vanished, the memory turns into garbage memory. It indicates that memory location exists but pointer is destroyed. This happens when memory is not de-allocated explicitly.

The pointer becomes wild pointer due to the following reasons:

- (1) Pointer declared but not initialized
- (2) Pointer alteration
- (3) Accessing destroyed data

(1) When a pointer is declared and not initialized, it holds an illicit address. It is very hard to manipulate such a pointer. Consider the following example:

### 10.5 Write a program to use wild pointer.

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

int main( )
{
    clrscr( );
    int *x;
    for (int k=0;k<10;k++)
        cout <<x[k]<<" ";
    return 0;
}
```

#### OUTPUT

**28005 27760 29793 29541 29728 8303 25954 28704 25205 26988**

**Explanation:** In the above program, pointer `x` is declared and not initialized. Using `for` loop the location containing pointer is increased and successive addresses are displayed.

(2) The careless assignment of new memory location in a pointer is called as pointer alteration. This happens when other wild pointer accesses the location of a legal pointer. The wild pointer converts a legal pointer to wild pointer.

(3) Sometimes the pointers attempt to access data that has no longer life. The program given next illustrates this:

### 10.6 Write a program to show display output when a pointer accesses a temporary data of the memory.

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

char *instring( );
char *inchar( );

void main( )
{
    clrscr( );
    char *ps,*pc;
    ps=instring( );
    pc=inchar( );
    cout <<" String      : "<<*ps<<endl;
    cout <<" Character   : "<<*pc<<endl;
}

char *instring( )
{
    char str[]="cpp ";
    return str;
}

char *inchar( )
{
    char g;
    g='D';
    return &g;
}
```

#### OUTPUT

String : c

Character :

**Explanation:** In the above program, ps and pc are character pointers. The function instring( ) and inchar( ) returns reference (base address of the string or character) and they are stored in the pointer ps and pc respectively. In both the functions the character variables str and g are local. When the control exists from these function and returns to main( ), local variables inside the user-defined functions are

destroyed. Thus, the pointers `ps` and `pc` point to the data that is destroyed. The contents displayed by the pointers are shown in the output.

## 10.5 POINTER TO CLASS

We know that pointer is a variable that holds the address of another data variable. The variable may be of any data type i.e., `int`, `float` or `double`. In the same way, we can also define pointer to class. Here, starting address of the member variables can be accessed. Such pointers are called `class` pointers.

**Example:**

```
class book
{char name [25];
 char author [25];
 int pages;
};
```

- a) `class book *ptr;`
- or
- a) `struct book *ptr;`

In the above example, `*ptr` is pointer to class `book`. Both the statements (a) and (b) are valid. The syntax for using pointer with member is given below.

1) `ptr->name` 2) `ptr->author` 3) `ptr->pages`.

By executing these three statements, starting address of each member can be estimated.

## 10.7 Write a program to declare a class. Declare pointer to class.

**Initialize and display the contents of the class member.**

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

    class man
    {
        public :
        char name[10];
        int age;
    };
}

man m={"RAVINDRA", 15};
man *ptr;
```

```

ptr=& (man) m;

// *ptr=(man) m;
// *ptr=man (m);
// *ptr=m;
//ptr=&m;

clrscr( );
cout <<"\n" <<m.name <<" " <<m.age;
cout <<"\n" <<ptr->name <<" " <<ptr->age;

}

```

## OUTPUT

**RAVINDRA 15**

**RAVINDRA 15**

**Explanation:** In the above program, the pointer `ptr` points to the object `m`. The statement `ptr=& (man) m;` assigns address of first member element of the class to the pointer `ptr`. Using the `dot` operator and `arrow` operator, contents can be displayed. The display of class contents is possible because the class variables are declared as `public`. The statements given below can also be used to assign address of objects to the pointer.

- `ptr=& (man) m;`
- `*ptr=(man) m;`
- `*ptr=man (m);`
- `*ptr=m;`
- `ptr=&m;`

## 10.6 POINTER TO OBJECT

Like variables, objects also have an address. A pointer can point to specified object. The following program illustrates this.

**10.8 Write a program to declare an object and pointer to the class.  
Invoke member functions using pointer.**

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

```

class Bill
{
    int    qty;
    float price;

    float amount;

public :

    void getdata (int a, float b, float c)
    {
        qty=a;
        price=b;
        amount=c;
    }

    void show( )
    {
        cout <<"Quantity      : " <<qty <<"\n";
        cout <<"Price         : " <<price <<"\n";
        cout <<"Amount        : " <<amount <<"\n";
    }
};

int main( )
{
    clrscr( );
    Bill s;
    Bill *ptr =&s;
    ptr->getdata(45,10.25,45*10.25);
    (*ptr).show( );
    return 0;
}

```

## **OUTPUT**

**Quantity :45**  
**Price :10.25**  
**Amount :461.25**

**Explanation:** In the above program, the class Bill contains two float and one integer member. The class Bill also contains member functions getdata ( ) and show( ) to read and display the data. In function main( ), s is an object of class Bill and ptr is pointer to the same class. The address of

object s is assigned to pointer ptr. Using pointer ptr with arrow operator (->) and dot operator (.) , member functions are invoked. The statements used for invoking functions are given next.

```
ptr->getdata (45,10.25,45*10.25);  
(*ptr).show ( );
```

Here, both the pointer declarations are valid. In the second statement, `ptr` is enclosed in the bracket because the dot operator (.) has higher precedence as compared to the indirection operator (\*) .

### **10.9 Write a program to create dynamically an array of objects of class type. Use new operator.**

```
# include <iostream.h>  
# include <conio.h>  
  
class Bill  
{  
    int    qty;  
    float price;  
    float amount;  
  
public :  
  
void getdata (int a, float b, float c)  
{  
    qty=a;  
    price=b;  
    amount=c;  
}  
  
void show( )  
{  
    cout <<"Quantity    : " <<qty <<"\n";  
    cout <<"Price      : " <<price <<"\n";  
    cout <<"Amount     : " <<amount <<"\n";  
}  
  
};  
  
int main( )  
{  
    clrscr( );
```

```

Bill *s= new Bill[2];

Bill *d =s;
int x,i;
float y;

for (i=0;i<2;i++)
{
    cout <<"\nEnter Quantity and Price : ";
    cin >>x >>y;
    s->getdata(x,y,x*y);
    s++;
}

for (i=0;i<2;i++)
{
    cout <<endl;
    d->show( );
    d++;
}

return 0;
}

```

## **OUTPUT**

**Enter Quantity and Price : 5 5.3**

**Enter Quantity and Price : 8 9.5**

**Quantity : 5**

**Price : 5.3**

**Amount : 26.5**

**Quantity : 8**

**Price : 9.5**

**Amount : 76**

**Explanation:** In the above program, the class Bill is same as in the previous example. In main( ), using new memory allocation operator, memory required for two objects is allocated to pointer s i.e., 10 bytes. The first for loop accepts the data through the keyboard. Immediately after this, the data is sent to the member function getdata( ). The pointer s is incremented. After incrementation, it points to the next memory location of its type. Thus,

two records are read through the keyboard. The second `for` loop is used to display the contents on the screen. Here, the function `show( )` is invoked. The logic used is same as in the first loop. The functions are invoked using pointer and it is explained in the previous example.

## 10.7 THE THIS POINTER

Use of this pointer is now outdated. The objects are used to invoke the non-static member functions of the class. For example, if `p` is an object of class `P` and `get( )` is a member function of `P`, the statement `p.get( )` is used to call the function. The statement `p.get( )` operates on `p`. In the same way if `ptr` is a pointer to `P` object, the function called `ptr->get( )` operates on `*ptr`.

However, the question is, how does the member function `get( )` understand which `p` it is functioning on? C++ compiler provides `get( )` with a pointer to `p` called `this`. The pointer `this` is transferred as an unseen parameter in all calls to non-static member functions. The keyword `this` is a local variable that is always present in the body of any non-static member function. The keyword `this` does not need to be declared. The pointer is rarely referred to explicitly in a function definition. However, it is used implicitly within the function for member references. For example, if `p.get(k)` is called, where `k` is a member of `P`, the keyword `this` is set to `&p` and `k` is set to `this->k`, which is equivalent to `p.k`. Figure 10.3 shows working of this pointer.

### 10.10 Write a program to use `this` pointer and return pointer reference.

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

class number
{
    int num;
public :

void input( )
{
    cout <<"\n Enter a Number : ";
    cin >>num;
}
```

```

void show( )
{   cout <<"\n The Minimum Number : "<<num; }

number min( number t)
{if (t.num<num)
    return t;
else
    return *this;      }

};

void main( )
{
clrscr( );

number n,n1,n2;
n1.input( );
n2.input( );
n=n1.min(n2);
n.show( );
}


```

## **OUTPUT**

**Enter a Number : 152**

**Enter a Number : 458**

**The Minimum Number : 152**

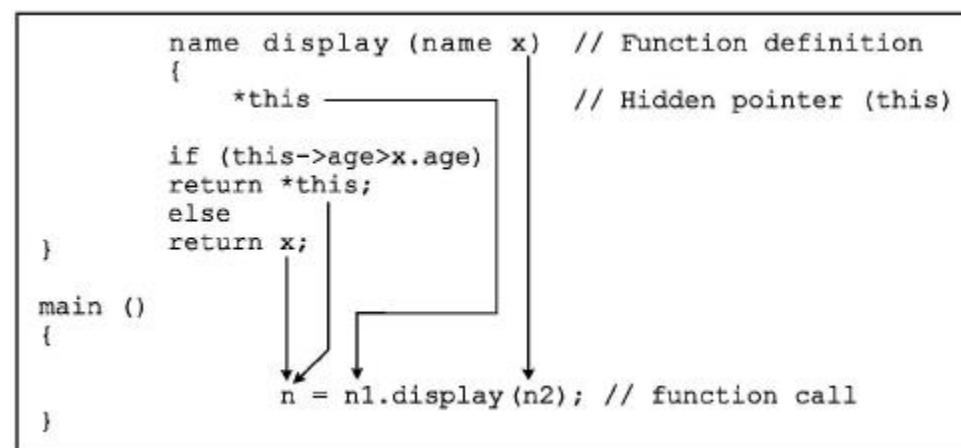
**Explanation:** In the above program, the class `number` contains one integer number variable `num`. The class also contains member functions `input()`, `show()` and `min()`. The `input()` function reads an integer through the keyboard. The `show()` function displays the contents on the screen. The `min()` function finds minimum number out of two. The variables `n`, `n1`, and `n2` are objects of class `number`.

In function `main()`, the objects `n1` and `n2` calls the function `input()` and read integer. Both the objects `n1` and `n2` are passed to function `min()` with the statement `n=n1.min(n2)`. The object `n` receives the returned value by the function `min()`. The object `n1` calls the function `min()`. The object `n2` is passed as an argument in function `min()`.

In function num( ), the formal object t receives the contents of argument n2 . In the same function, the object n1 is also accessible. We know that the pointer this is present in the body of every non-static member function. The pointer this points to the object n1 . Using pointer this we can access the individual member variables of object n1 . The if statement compares the two objects and return statement returns the smaller objects.

## 10.8 POINTER TO DERIVED CLASSES AND BASE CLASSES

It is possible to declare a pointer which points to the base class as well as the derived class. One pointer can point to different classes. For example, X is a base class and Y is a derived class. The pointer pointing to X can also point to Y .



**Fig.10.3** Working of this

## 10.11 Write a program to declare a pointer to the base class and access the member variable of base and derived class.

```

// Pointer to base object //

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

class A
{
public :
    int b;
    void display( )
    {

```

```

        cout <<"b = " <<b <<"\n";
    }
};

class B : public A
{
public :
int d;
void display( )
{
    cout <<"b= " <<b <<"\n" <<" d=" <<d <<"\n";
}

};

main( )
{
    clrscr( );
A *cp;
A base;
cp=&base;
cp->b=100;
// cp->d=200; Not Accessible
cout <<"\n cp points to the base object \n";
cp->display( );

B b;

cout <<"\n cp points to the derived class \n";
cp=&b;
cp->b=150;
// cp->d=300; Not accessible
cp->display( );
return 0;
}

```

**OUTPUT**

**cp points to the base object**

**b = 100**

**cp points to the derived class**

**b = 150**

**Explanation:** In the above program, A and B are two classes with one integer member variable and member function. The class B is derived from class A. The pointer cp points to the class A. The variable base is an object of the class A. The address of object base is assigned to pointer cp. The pointer cp can access the member variable b, a member of base class but cannot access variable d, a member of derived class. Thus, the following statement is invalid.

(a) cp->d=200;

The variable b is an object of class B. The address of object b is assigned to the pointer cp. However, b is an object of derived class and access to member variable d is not possible. Thus, the following statement is invalid.

(b) cp->d=300;

Here, cp is pointer to the base class and could not access the members of derived class.

### 10.12 Write a program to declare a pointer to the derived class and access the member variables of base and derived class.

```
// Pointer to derived object //

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

class A
{
public :
    int b;
    void display( )
    {
        cout <<"b = " <<b <<"\n";
    }
};

class B : public A
{
public :
    int d;
    void display( )
    {
        cout <<"\tb= " <<b <<"\n" <<"\td= " <<d <<"\n";
    }
};
```

```

};

main( )
{
clrscr( );
B *cp;
B b;

cp=&b;
cp->b=100;
cp->d=350;

cout <<"\n cp points to the derived object \n";
cp->display( );
return 0;
}

```

## **OUTPUT**

**cp points to the derived object**

**b= 100**

**d= 350**

**Explanation:** The above program is same as the previous one. The only difference is that the pointer `cp` points to the objects of derived class. The pointer `cp` is pointer to class `B`. The variable `b` is an object of class `B`. The address of `b` is assigned to the pointer `cp`. The pointer `cp` can access the member variables of both base and derived classes. The output of the program is shown above.

**10.13 Write a program to declare object and pointer to class. Assign value of pointer to object. Display their values. Also carry out conversion from basic type to class type.**

```

# include <iostream.h>
# include <conio.h>
# include <alloc.h>
class A
{
private :
int a;
public:
A ( ) { a=30; }

```

```

void show ( ) { cout <<"\n a = "<<a ;}

A (int x) { this->a=x; }

};

int main( )
{
clrscr( );

A k,*b,a;
*b=50;
k=*b;
    b->show( );
    a.show( );
    k.show( );
    return 0;
}

```

## **OUTPUT**

```

a = 50
a = 30
a = 50

```

**Explanation:** In the above program, a and k are objects of class A . The \*b is a pointer object. The objects a and k are initialized to 30 by the constructor. The statement \*b=50 assigns 50 to data member a . This is carried out by the conversion constructor A (int) . The value of \*b is assigned to object k . The member function show( ) is called by all the three objects. The output of the program is given above.

## **10.9 POINTER TO MEMBERS**

It is possible to obtain address of member variables and store it to a pointer. The following programs explain how to obtain address and accessing member variables with pointers.

### **10.14 Write a program to initialize and display the contents of the structure using dot (.) and arrow (->) operator.**

```

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

```

```

int main ( )
{
    clrscr( );

    struct c
    {
        char *name;
    } ;

    c b, *bp;
    bp->name=" CPP";
    b.name="C &";
    cout<<b.name;
    cout<<bp->name;
    return 0;
}

```

## **OUTPUT**

### **C & CPP**

**Explanation:** In the above program, structure `c` is declared. The variable `b` is object and `bp` is a pointer object to structure `c`. The elements are initialized and displayed using dot (.) and arrow (->) operators. These are the traditional operators which are commonly used in C. C++ allows two new operators `.*` and `->*` to carry the same task. These C++ operators are recognized as pointer to member operators.

#### **10.15 Write a program to initialize and display the contents of the structure using `.*` and arrow `->*` operator.**

```

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

struct data
{
    int x;
    float y;
} ;

int main ( )
{
    clrscr( );
    int data ::*xp=&data::x;
    float data::*yp=&data::y;
}

```

```

data d={11,1.14};
cout <<endl << d.*xp << endl << d.*yp;

data *pp;

pp=&d;

cout << endl << pp->*xp << endl << pp->*yp;

d.*xp=22;
pp->*yp=8.25;
cout << endl << d.*xp << endl << d.*yp;
cout << endl << pp->*xp << endl << pp->*yp;
return 0;
}

```

## OUTPUT

```

11
1.14
11
1.14
22
8.25
22
8.25

```

**Explanation:** In the above program, struct data is defined and it has two data members of integer and float type. Consider the following statements:

```

int data ::*xp=&data::x;
float data::*yp=&data::y;

```

The `*xp` and `*yp` are pointers. The class name followed by scope access operator points that they are pointers to member variables of structures `int x` and `float y`. The rest part of the statement initializes the pointers with addresses of `x` and `y` respectively.

The rest part of the program uses operators `.*` and `->*` to initialize and access the member elements in the same fashion like the operator dot (`.`) and arrow (`->`).

**10.16 Write a program to declare and initialize an array. Access and display the elements using . \* and ->\* operators.**

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

struct data
{int x;
 float y;
};

int main( )
{clrscr( );
 int data ::*xp=&data::x;
 float data::*yp=&data::y;

data darr[]={ {12,2.5},
{58,2.4},
{15,5.7} };

for ( int j=0;j<=2;j++)
cout <<endl << darr[j].*xp << " " << darr[j].*yp;
return 0;
}
```

## OUTPUT

**12 2.5**

**58 2.4**

**15 5.7**

**Explanation:** The above program is the same as the last one. An array darr[] is initialized. The for loop and the operator . \* accesses the elements and elements are displayed on the console.

**10.17 Write a program to declare variables and pointers as members of class and access them using pointer to members.**

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

class data
{
public:
```

```

        int x;
        float y;
        int *z;
        float *k;
        int **l;
        float **m;
    };

void main( )
{
    clrscr( );
    int data::*xp=&data::x;
    float data ::*yp=&data::y;

    int *data::*zp=&data::z;
    float *data::*kp=&data::k;

    int **data::*lp=&data::l;
    float **data::*mp=&data::m;

    data ob={51,4.58,&ob.x,&ob.y,&ob.z,&ob.k};
    data *pp;
    pp=&ob;

    cout <<endl<<ob.*xp<<endl<<ob.*yp;
    cout <<endl<<* (ob.*zp)<<endl<<* (ob.*kp);
    cout<<endl<<** (ob.*lp)<<endl<<** (ob.*mp);

    cout <<endl<<(pp->*xp)<<endl<<(pp->*yp);
    cout<<endl<<* (pp->*zp)<<endl<<* (pp->*kp);
    cout <<endl<<** (pp->*lp)<<endl<<** (pp->*mp);

    * (ob.*zp)=11;
    ** (pp->*mp)=8.24;

    cout <<endl<<ob.*xp<<endl<<ob.*yp;
}

```

## **OUTPUT**

**51**

**4.58**

**51**

```
4.58  
51  
4.58  
51  
4.58  
51  
4.58  
51  
4.58  
11  
8.24
```

**Explanation:** In the above program, the class and data are declared and all members are public. The ob is an object of the class data and it is initialized. The variable pp is a pointer and initialized with address of object ob. The rest program statements use the pointer to member operators and display the contents of the class on the screen.

**TIP**

Pointers to members are not associated with any particular object.

**10.18 Write a program that invokes member functions using pointer to functions.**

```
# include <iostream.h>  
# include <conio.h>  
  
class data  
{  
    public :  
  
        void joy1( ) { cout << endl << (unsigned) this << " in joy1"; }  
  
        void joy2( ) { cout << endl << (unsigned) this << " in joy2"; }  
  
        void joy3( ) { cout << endl << (unsigned) this << " in joy3" << endl;  
    }  
  
};  
  
void main( )
```

```

{
    clrscr( );
    data od[4];

    void (data ::*m[3])( )={&data::joy1,&data::joy2,&data::joy3};

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

    {
        for (int y=0;y<=2;y++)      (od[x].*m[y])( );
    }

}

```

## **OUTPUT**

**65522 in joy1**

**65522 in joy2**

**65522 in joy3**

**65523 in joy1**

**65523 in joy2**

**65523 in joy3**

**65524 in joy1**

**65524 in joy2**

**65524 in joy3**

**65525 in joy1**

**65525 in joy2**

**65525 in joy3**

***Explanation:*** In the above program, class data has three-member

variables joy1, joy2, and joy3. The array m[3] () holds the addresses of member functions. The nested for loops and statement within it invokes member functions.

### **10.19 Write a program to declare pointer to member variable and display the contents of the variable.**

```

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

class A
{

```

```

public :
int x;
int y;
int z;
};

void main( )
{
clrscr( );
A a;
int *p;
a.x=5;
a.y=10;
a.z=15;
p=&a.x;
cout<<endl<<"x= "<<*p;
p++;
cout<<endl<<"y= "<<*p;
p++;
cout<<endl<<"z= "<<*p;
}

```

## **OUTPUT**

**x= 5**

**y= 10**

**z= 15**

***Explanation:*** In the above program the class A is declared with three public member variables x, y and z. In function main( ), a is an object of class A and p is an integer pointer. The three member variables are initialized with 5, 10 and 15 using assignment operation.

The address of variable x is assigned to a pointer. The postfix incrementation operation gives successive memory locations, and values stored at those locations are accessed and displayed. Thus, member variables are stored in successive memory locations. Using the same concept we can also access private members.

## **10.10 ACCESSING PRIVATE MEMBERS WITH POINTERS**

In the last topic we learned how to access public members of a class using pointers. The public and private member variables are stored in successive memory locations. The following program explains how private members can also be accessed using pointer.

## **10.20 Write a program to access private members like public members of the class using pointers.**

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

class A

{
private:
int j;
public :
int x;
int y;
int z;
A ( ) { j=20; }
};

void main( )
{
    clrscr( );
    A a;
    int *p;
    a.x=11;
    a.y=10;
    a.z=15;
    p=&a.x;
    p--;
    cout<<endl<<"j = "<<*p;
    p++;
    cout<<endl<<"x= "<<*p;
    p++;
    cout<<endl<<"y= "<<*p;
    p++;
    cout<<endl<<"z= "<<*p;
}
```

### **OUTPUT**

**j = 20**  
**x = 11**  
**y = 10**  
**z = 15**

**Explanation:** This program is the same as the last one. In addition, the class A has one private member variable. The private member variables can be accessed using functions and no direct access is possible to them. However, using pointers we can access them like public member variables.

## 10.11 DIRECT ACCESS TO PRIVATE MEMBERS

So far, we have initialized private members of the class using constructor or member function of the same class. In the last sub-topic we also learned how private members of the class can be accessed using pointers. In the same way, we can initialize private members and display them. Obtaining the address of any public member variable and using address of the object one can do this. The address of the object contains address of the first element. Programs concerning this are explained below:

### 10.21 Write a program to initialize the private members and display them without the use of member functions.

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

class A
{
    private :
    int x;
    int y;
};

void main( )
{
    clrscr( );
    A a;
    int *p=(int*)&a;
    *p=3;
    p++;
    *p=9;
    p--;
    cout <<endl <<"x= "<<*p;
    p++;
    cout <<endl<<"y= "<<*p;
}
```

### OUTPUT

```
x= 3
y= 9
```

**Explanation:** In the above program, a is an object of class A. The address of the object is assigned to integer pointer p applying typecasting. The pointer p points to private member x. Integer value is assigned to \*p i.e., x. By

increase, operation next memory location is accessed and nine is assigned i.e., *y*. The *p*- – statement sets memory location of *x* using *cout* statement and contents of pointer is displayed.

## 10.12 ADDRESS OF OBJECT AND VOID POINTERS

The size of object is equal to number of total member variables declared inside the class. The size of member function is not considered in object. The object itself contains address of first member variable. By obtaining the address we can access the member variables directly, no matter whether they are private or public. The address of object cannot be assigned to the pointer of the same type. This is because increase operation on pointers will set the address of object according to its size (size of object= size of total member variables). The address of object should be increased according to the basic data type. To overcome this situation a void pointer is useful. The type of void pointer can be changed at run-time using typecasting syntax. Thus, all the member variables can be accessed directly. Consider the following program:

### 10.22 Write a program to declare void pointers and access member variables using void pointers.

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

class A
{
protected:
    int x;
    int y;

};

class B : public A
{
public :
    int z;
    B ( ) { x=10; y=20; z=30; }

};

void main ( )
```

```

{
    clrscr( );           // clears the screen
    B b;                // object declaration
    int j;               // declaration of integer j
    void *p=&b;          // declaration & initialization of void
pointer
    for (j=0;j<3;j++)   // for loop executed for three times
        printf ("\n member variable [ %d ] = %d",j+1,*((int*)p+j));
        // cout<<j+1<<" " <<*((int*)p+j)<<endl;
}

```

## OUTPUT

**member variable [1] = 10  
 member variable [2] = 20  
 member variable [3] = 30**

**Explanation:** In the above program, b is an object of derived class B. The address of object b is assigned to void pointer p. Using for loop and typecasting operation, integer values are accessed. The address of void pointer is increased by two bytes. The cout statement now and then will not display the result properly and hence, the printf( ) statement is used. The typecasting operation is done according to the data type of member variables. Here, all the class member variables are of integer type. In case, the class contains variables of integer, float and char type, then it is necessary to typecast according to the sequence of the member variables.

## 10.13 ARRAYS

### ARRAYS

Array is a collection of elements of similar data types in which each element is unique and located in separate memory locations.

#### (1) ARRAY DECLARATION AND INITIALIZATION

Declaration of an array is shown below:

```

Declaration
int a [5];

```

It tells to the compiler that 'a' is an integer type of array and it must store five integers. The compiler reserves two bytes of memory for each integer array element. In the same way array of different data types are declared as below:

```

Array Declaration
char ch [10];
float real [10];

```

```
long num [5];
```

The array initialization is done as given below:

Array Initialization

```
int a [5]={1,2,3,4,5};
```

Here, five elements are stored in an array 'a'. The array elements are stored sequentially in separate locations. Then question arises how to call individually to each element from this bunch of integer elements. Reading of array elements begins from zero.

Array elements are called with array name followed by element numbers.

Table 10.1 explains the same.

**Table 10.1** Calling array elements

**a[0] refers to 1st element i.e. 1**

**a[1] refers to 2nd element i.e. 2**

**a[2] refers to 3rd element i.e. 3**

**a[3] refers to 4th element i.e. 4**

**a[4] refers to 5th element i.e. 5**

## 10.14 CHARACTERISTICS OF ARRAYS

(1) The declaration `int a [5]` is nothing but creation of five variables of integer type in memory. Instead of declaring five variables for five values, the programmer can define them in an array.

(2) All the elements of an array share the same name, and they are distinguished from one another with the help of element number.

(3) The element number in an array plays major role in calling each element.

(4) Any particular element of an array can be modified separately without disturbing other elements.

```
int a [5]={1,2,3,4,8};
```

If the programmer needs to replace 8 with 10, he/she is not required to

change all other numbers except 8. To carry out this task the statement `a[4]=10` can be used. Here the other three elements are not disturbed.

(5) Any element of an array `a [ ]` can be assigned/equated to another ordinary variable or array variable of its type.

**For example**

```
b= a [2];  
a [2]=a [3];
```

(a) In the statement `b= a [2]` or vice versa, value of `a [2]` is assigned to `b`, where `b` is an integer.

(b) In the statement `a [2]=a [3]` or vice versa, value of `a [2]` is assigned to `a [3]`, where both the elements are of the same array.

(6) The array elements are stored in continuous memory locations. The amount of storage required for holding elements of the array depends upon its type and size. The total size in bytes for a single dimensional array is computed as shown below.

Total bytes = `sizeof(data type)` X size of array

The rules for array declaration are the same in **C** and **C++**. The only difference is in the declaration of strings i.e., character array.

| <b>Program executed in C</b>                                                                                                                                                                                          | <b>Program executed in C++</b>                                                                                                                                                                                        |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre># include &lt;stdio.h&gt;<br/># include &lt;conio.h&gt;<br/><br/>void main( )<br/>{<br/>char text[3]="xyz";<br/>char txt[3]="abc";<br/>clrscr( );<br/>printf ("\n %s",text);<br/>printf ("%s", text)<br/>}</pre> | <pre># include &lt;stdio.h&gt;<br/># include &lt;conio.h&gt;<br/><br/>void main( )<br/>{<br/>char text[3]="xyz";<br/>char txt[3]="abc";<br/>clrscr( );<br/>printf ("\n %s",text);<br/>printf ("%s", text)<br/>}</pre> |
| <b>OUTPUT</b>                                                                                                                                                                                                         | <b>OUTPUT</b>                                                                                                                                                                                                         |
| <b>xyz_abcW__</b>                                                                                                                                                                                                     | <b>xyzA</b>                                                                                                                                                                                                           |

**Explanation:** Both the above programs are same. The same programs are executed in C (Ver 2.0) and C++ (ver 3.0). The character arrays are declared exactly equal to the length of the string. The account of null character

is not taken. That's why in C the `printf( )` statement displays the contents of the next character array followed by the character array. In C++, the `printf( )` statement displays one garbage value followed by the original string. This is so far the difference of string initialization in C and C++. It is a better practice to always count an account of null characters. Thus, the above character arrays can be correctly declared as below:

- (a) `char text [4] = "xyz";`
- (b) `char txt [4] = "abc";`

Now the correct output can be obtained.

## 10.15 INITIALIZATION OF ARRAYS USING FUNCTIONS

The programmers always initialize the array using statement like `int d[ ]={1, 2, 3, 4, 5}`. Instead of this the function can also be directly called to initialize the array. The program given below illustrates this point.

### 10.23 Write a program to initialize an array using functions.

```
# include <stdio.h>
# include <conio.h>
main( )
{
int k,c( ),d[]={c( ),c( ),c( ),c( ),c( )};
clrscr( );
printf ("\n Array d[] elements are :");
for (k=0;k<5;k++)
printf ("%2d",d[k]);

return (NULL);
}

c( )
{
static int m,n;
m++;
printf ("\nEnter Number d[%d] : ",m);
scanf ("%d",&n);
return(n);
}
```

#### OUTPUT:

Enter Number d[1] : 4

```
Enter Number d[2] : 5
Enter Number d[3] : 6
Enter Number d[4] : 7
Enter Number d[5] : 8
```

**Array d[] elements are : 4 5 6 7 8**

**Explanation:** Function can be called in the declaration of arrays. In the above program, d[ ] is an integer array and c( ) is a user-defined function. The function c( ) when called reads value through the keyboard. The function c( ) is called from an array i.e., the values returned by the function are assigned to the array. The above program will not work in C.

## 10.16 ARRAYS OF CLASSES

We know that array is a collection of similar data types. In the same way, we can also define array of classes. In such type of array, every element is of class type. Array of class objects can be declared as shown below:

```
class stud
{
    public:
        char name [12];           // class declaration
        int rollno;
        char grade[2];
};

class stud st[3];           // declaration of array of class objects
```

In the above example, st[3] is an array of three elements containing three objects of class stud. Each element of st[3] has its own set class member variables i.e., char name[12], int rollno and char grade[2]. A program is explained as given below:

**10.24 Write a program to display names, rollnos, and grades of 3 students who have appeared in the examination. Declare the class of name, rollnos, and grade. Create an array of class objects. Read and display the contents of the array.**

```
# include <stdio.h>
# include <conio.h>
# include <iostream.h>
void main( )
{
int k=0;
class stud
{
```

```

public :
char name[12];
int rollno;
char grade[2];
};

class stud st[3];
while (k<3)
{clrscr( );
gotoxy(2,4);
cout <<"Name      : ";
gotoxy(17,4);
cin >>st[k].name;

gotoxy(2,5);
cout <<"Roll No. : ";

gotoxy(17,5);
cin >>st[k].rollno;
gotoxy(2,6);
cout <<"Grade     : ";
gotoxy(17,6);
cin>>st[k].grade;
st[k].grade[1]='\0';
puts ("  press any key..");
getch( );
k++;
}
k=0;
clrscr( );
cout<<"\nName\tRollno Grade\n";
while (k<3)
{
cout<<st[k].name<<"\t"<<st[k].rollno<<" \t"<<st[k].grade<<"\n";
k++;
}
}

```

## **OUTPUT**

| <b>Name</b>   | <b>Rollno</b> | <b>Grade</b> |
|---------------|---------------|--------------|
| <b>Balaji</b> | <b>50</b>     | <b>A</b>     |
| <b>Manoj</b>  | <b>51</b>     | <b>B</b>     |
| <b>Sanjay</b> | <b>55</b>     | <b>C</b>     |

**Explanation:** In the above program, class stud is declared. Its members are `char name[12];` `int rollno` and `char grade[2]` and all are public. The array `st[3]` of class stud is declared. The first while loop and `cin( )` statements within the first while loop are used for repetitive data reading. The second while loop and `printf( )` statements within it display the contents of array. In the above program all the member variables are public. If the member variables are private, the above program will not work. Then, we need to declare member functions to access the data.

### SUMMARY

- (1) Like C, in C++ variables are used to hold data values during program execution. Every variable when declared occupies certain memory locations.
- (2) Pointer saves the memory space. The execution time with pointer is faster because data is manipulated with the address.
- (3) The *indirection operator* (`*`) is also called the *deference operator*. When a pointer is dereferenced, the value at that address stored by the pointer is retrieved.
- (4) The `&` is the address operator and it represents the address of the variable. The address of any variable is a whole number.
- (5) Pointers can also be declared as void type. Void pointers cannot be dereferenced without explicit type conversion. This is because being void the compiler cannot determine the size of the object the pointer points to.
- (6) The pointer `this` is transferred as an unseen parameter in all calls to non-static member functions. The keyword `this` is a local variable, always present in the body of any non-static member function.
- (7) Array is a collection of similar data types in which each element is a unique one and located in separate memory locations.
- (8) The array elements are stored in continuous memory locations. The amount of storage required for holding elements of an array depends upon its type and size.

### EXERCISES

#### [A] Answer the following questions.

- (1) What are pointers?
- (2) What is void pointer?
- (3) What is wild pointer?
- (4) What is `this` pointer?
- (5) What are the features and uses of pointers?
- (6) In which situation does the pointer become harmful?

- (7) Explain any two characteristics of pointers.
- (8) What are arrays?
- (9) How can array initialization be carried out using function?
- (10) Explain any three characteristics of arrays.
- (11) How can private members be accessed using pointers?
- (12) Why is declaration of void variable not permitted?

**[B] Answer the following by selecting the appropriate option.**

- (1) The pointer holds
  - (a) address of the variable
  - (b) value of the variable
  - (c) both (a) & (b)
  - (d) none of the above
- (2) An integer type pointer can hold only address of
  - (a) integer variable
  - (b) float variable
  - (c) any variable
  - (d) none of the above
- (3) The address of the variable is displayed by the symbol
  - (a) & (ampersand)
  - (b) \* (asterisk)
  - (c) (not operator)
  - (d) new operator
- (4) The `sizeof( )` object is equal to
  - (a) total size of data member variables
  - (b) total size of member functions
  - (c) size of large element
  - (d) none of the above
- (5) Array elements are stored in
  - (a) continuous memory locations
  - (b) different memory locations
  - (c) CPU registers
  - (d) none of the above
- (6) Private member variables can be accessed directly using
  - (a) pointers
  - (b) arrays
  - (c) `this` pointer
  - (d) none of the above
- (7) The object itself is a

- (a) pointer
  - (b) variable
  - (c) class member
  - (d) none of the above
- (8) The size of void pointers is
- (a) 2 bytes
  - (b) 0 bytes
  - (c) 4 bytes
  - (d) none of the above
- (9) The `this` is present in every
- (a) member function
  - (b) non-member function
  - (c) every object
  - (d) all of the above
- (10) The array name itself is a
- (a) pointer
  - (b) reference
  - (c) variable
  - (d) object

(11) In the following program the statement `a=& (b=&x) ;`

```
void main ( )
{
    int x=10;
    int **a,*b;
    a=& (b=&x) ;
    cout<<"x="<<**a;
    cout<<"\nx="<<*b;
}
```

- (a) address of `b` is assigned to `a`
- (b) address of `x` is assigned to `a`
- (c) pointer `a` points to `b`
- (d) both (a) and (c)

### **[C] Attempt the following programs.**

- (1) Write a program to declare integer pointer. Store 10 numbers in the pointer. Use new operator to allocate memory for 10 integers. Read and display the numbers.
- (2) Try the above program without allocating memory. Use increment operator to get successive location in the memory. While displaying numbers, retrieve previous memory locations using decrement operator. (Tip: The

program without new operator shows warnings, for the time being ignore it, but in practical application new operator is essential)

(3) Write a program to declare float and integer arrays. Read five float numbers. Separate integer and fractional parts. Store the integers separated in integer array and fractional in float array. Display the contents of both the arrays.

(4) Write a program to find the determinant of a 2 X 2 matrix using pointers.

B(1, 1) B(1, 2)

B(2, 1) B(2, 2)

The determinant of the matrix is  $B_{1,1} \cdot B_{2,2} - B_{1,2} \cdot B_{2,1}$ .

(5) Write a program to find the number of zero elements in 3 X 3 matrix.

Read and display the element. Also, write this program using pointers.

(6) The table given below gives the sales of four items of a shop.

| Day       | Item1   | Item2  | Item3  | Item4  |
|-----------|---------|--------|--------|--------|
| Monday    | 1785.25 | 400.50 | 145.52 | 630.80 |
| Tuesday   | 1575.00 | 375.85 | 258.50 | 540.80 |
| Wednesday | 1910.00 | 180.45 | 456.54 | 985.50 |
| Thursday  | 1745.80 | 254.58 | 452.80 | 150.75 |
| Friday    | 1948.85 | 458.95 | 459.50 | 75.00  |
| Saturday  | 1452.95 | 258.45 | 500    | 150.75 |

(a) Calculate the total daily sales per day.

(b) Calculate the weekly sales of each item.

(c) Calculate the average daily sales of each item.

(d) Calculate the total weekly sales of the shop.

(7) Write a program to find the sum of diagonal elements of 3 X 3 matrix.

(8) Write a program to declare array of character type. Store ten names in array. Display the names in ascending order.

(9) Write a program to declare two pointers to the same variable. Display the values of variable using both the pointers.

(10) Write a program to declare an integer array. Display the contents of array using pointer.

## [D] Find bugs in the following programs.

(1)

```
class set  
{ int x; };
```

```
void main( )  
{  
    set a;
```

```
    cout << endl << (unsigned) &a.x;
}
```

**(2)**

```
void main ( )
{
    int x=20;
    float *p;
    p=&x;
    cout<<x;
}
```

**(3)**

```
void main ( )
{
    void y(int);
    y =30;
    cout<<y;
}
```

**(4)**

```
void main ( )
{
    void *y;
    (int) y =30;
    cout<<(unsigned)y;
}
```

**(5)**

```
void main ( )
{
    char *x;
    int u=10;
    x=&u;
}
```

**(6)**

```
void main ( )
{
    int x=10;
    int *a,*b;
    a=&(b=&x);
    cout<<"x="<<*a;
    cout<<"\nx="<<*b;
}
```

# 11

CHAPTER

## C++ and Memory

C  
H  
A  
P  
T  
E  
R  
  
O  
U  
T  
L  
I  
N  
E

- [11.1 Introduction](#)
- [11.2 Memory Models](#)
- [11.3 The new and delete Operators](#)
- [11.4 Heap Consumption](#)
- [11.5 Overloading new and delete Operators](#)
- [11.6 Execution Sequence of Constructors and Destructors](#)
- [11.7 Specifying Address of an Object](#)
- [11.8 Dynamic Objects](#)

## —• 11.9 Calling Convention

### 11.1 INTRODUCTION

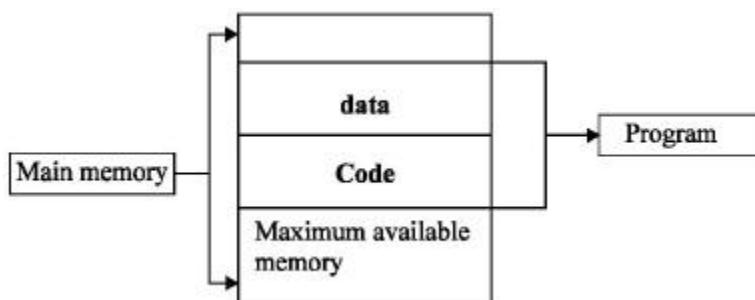
Memory is one of the critical resources of a computer system. One must be conscious about the memory modules while an application is being developed. In this chapter, we will learn stack, heap, dynamic object, and calling conventions.

### 11.2 MEMORY MODELS

The memory model sets the supportable size of code and data areas as shown in Figure 11.1. Before compiling and linking the source code, we need to specify the appropriate memory model. Using memory models, we can set the size limits of the data and code. C/C++ programs always use different segments for code and data. The memory model you opt decides the default method of memory addressing. The default memory model is small. Table 11.1 describes properties of all memory models.

#### (1) TINY

Use the tiny model when memory is at an absolute premium. All four segment registers (CS, DS, ES, SS) are initialized with same address and all addressing is accomplished using 16 bits. Total memory capacity in this case is 64 K bytes. In short, memory capacity is abbreviated as KB. This means that the code, data, and stack must all fit within the same 64 KB segment. Programs are executed quickly if this model is selected. Near pointers are always used. Tiny model programs can be converted to .COM format.



**Fig 11.1** Memory Organization

#### (2) SMALL

All code should fit in a single 64 KB segment and all data should fit in a second 64 KB segment. All pointers are 16 bits in length. Execution speed is same as tiny model. Use this model for average size programs. Near pointers are always used.

### **(3) MEDIUM**

All data should fit in a single 64 KB segment. However, the code is allowed to use multiple segments. All pointers to data are 16 bits, but all jumps and calls require 32 bit addresses. With this, access to data is fast. However, slower program execution is observed with this model. This, model is suitable for big programs that do not keep a large amount of data in memory. Far pointers are used for code but not for data.

### **(4) COMPACT**

All code should fit in 64 KB segment but the data can use multiple segments. However, no data item can surpass 64 KB. All pointers to data are 32 bits, but jumps and calls can use 16 bit addresses. Slow access to data and quick code execution will be observed in this setting.

Compact model is preferred if your program is small but you require pointing large amount of data. The compact model is the opposite of the medium model. Far pointers are preferred for data but not for code. Code is limited to 64 KB, while data has a 1 MB range. All functions are near by default and all data pointers are far by default.

### **(5) LARGE**

Both code and data are allowed to use multiple segments. All pointers are 32 bits in length. However, no single data item can exceed 64 KB. Code execution is slower.

Large model is preferred for very big programs only. Far pointers are used for both code and data, specifying both a 1 MB range. All functions and data pointers are far by default.

### **(6) HUGE**

Both code and data are allowed to use multiple segments. Every pointer is of 32 bits in length. Code execution is the slowest.

Huge model is preferred for very big programs only. Far pointers are used for both code and data. Turbo C++ usually bounds the size of all data to 64 K. The huge memory model sets aside that limit, permitting data to hold more than 64 K. The huge model permits multiple data segments (each 64 K in size), up

to 1 MB for code, and 64 K for stack. All functions and data pointers are supposed to be far.

**Table 11.1** Memory models

| Sr. No. | Memory Model | Segments |           |            | Type of pointer |      |
|---------|--------------|----------|-----------|------------|-----------------|------|
|         |              | Code     | Data      | Stack      | Code            | Data |
| 1       | Tiny         | 64 K     |           |            | near            | near |
| 2       | Small        | 64 K     | 64 K      |            | near            | near |
| 3       | Medium       | 1 MB     | 64 K      |            | far             | near |
| 4       | Compact      | 64 K     | 1 MB      |            | near            | far  |
| 5       | Large        | 1 MB     | 1 MB      |            | far             | far  |
| 6       | Huge         | 1 MB     | 64 K each | 64 K stack | far             | far  |

## (7) SEGMENT AND OFFSET ADDRESS

Every address has two parts, segment and offset. We can separate these address parts using following two macros defined in dos.h header file.

`FP_SEG( )` – This macro is used to obtain segment address of the given pointer variable.

`FP_OFF( )` – This macro is used to obtain offset address of the given pointer variable.

The following program illustrates working of both these macros.

### 11.1 Write a program to obtain segment and offset address.

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

void main ( )
{
    clrscr( );
    int ch;
    cout<<"\n Complete Address of ch : "<<&ch;
    cout <<"\n Segment address : "<<hex<<FP_SEG(&ch);
    cout <<"\n Offset address : "<<hex<<FP_OFF(&ch);
}
```

### OUTPUT

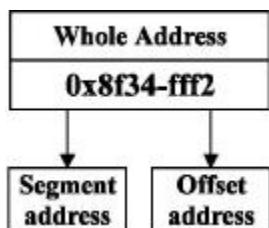
**Complete Address of ch : 0x8f34fff2**

**Segment address : 8f34**

**Offset address : fff2**

**Explanation:** In the above program, variable `ch` is declared. The first `cout` statement displays whole address of the variable `ch` i.e., `0x8f34fff2`.

The macro `FP_SEG( )` returns segment address. The statement used is `FP_SEG(&ch)`. Here, the address of `ch` is passed to the macro. In the same way, offset address is obtained using statement `FP_OFF(&ch)`. You can observe the complete address as displayed above. The separated addresses are also same. [Figure 11.2](#) makes it clearer.



**Fig. 11.2** Segment and offset address

The `far`, `huge` and `near` are keywords. They are summarized in [Table 11.2](#).

**The far pointer:** A `far` pointer is a 32 bit pointer and contains both segment and offset as

**The huge pointer:** A `huge` pointer is 32 bits long and contains both segment and offset add

**The near pointer:** A `near` pointer is 16 bits long and uses the contents of CS or DS register part. The offset part of the address is stored in 16 bits near pointer.

**Table 11.2** Keywords summary

| KEYWORD           | DATA           | CODE                   | POINTER ARITHMETIC |
|-------------------|----------------|------------------------|--------------------|
| <code>near</code> | 16 bit address | 16 bit address         | 16 bit             |
| <code>far</code>  | 32 bit address | 32 bit address         | 16 bit             |
| <code>huge</code> | 32 bit address | Keyword not applicable | 32 bit             |

## 11.2 Write a program to declare `far`, `near` and `huge` pointers. Display their sizes.

```

# include <iostream.h>
# include <constream.h>

void main( )
{
    clrscr( );
    char far *f; // far pointer declaration
    char near *n; // near pointer declaration
    char huge *h; // huge pointer declaration
  
```

```

    cout <<"\n Size of far pointer : "<<sizeof(f);
    cout <<"\n Size of near pointer : "<<sizeof(n);
    cout <<"\n Size of huge pointer : "<<sizeof(h);
}

```

## OUTPUT

**Size of far pointer: 4**

**Size of near pointer: 2**

**Size of huge pointer: 4**

**Explanation:** In the above program, the pointer f, n, and h are declared. In pointer declaration, the pointers are preceded by keywords far, near and huge. The sizeof( ) operator displays the size of pointers in bytes.

## 11.3 Write a program to use far pointer.

```

#include <iostream.h>
#include <constream.h>

void main( )
{
    clrscr( );
    char far *s;           // pointer declaration
    s=(char far*)0xB8000000L; // starting address
    *s='W';                // displays character on the screen
}

```

## OUTPUT

**W**

**Explanation:** Character pointer \*s is declared. It is a far pointer. The address 0xB8000000L is assigned to it. It is starting address outside the data segment. The address is converted to char far\* type by applying type casting. The statement \*s='W' displays the character on the screen.

## 11.3 THE NEW AND DELETE OPERATORS

So far, we have used the new and delete operators in short programs, but for applying them in huge applications we need to understand them completely. A small mistake in syntax may cause a critical error and possibly corrupt memory heap. Before studying memory heap in detail let us repeat few points regarding new and delete operators.

- (1) The new operator not only creates an object but also allocates memory.
- (2) The new operator allocates correct amount of memory from the heap that is also called as a free store.

(3) The object created and memory allocated by using `new` operator should be deleted by the `delete` operator otherwise such mismatch operations may corrupt the heap or may crash the system. According to ANSI standard, it is a valid outcome for this invalid operation and the compiler should have routines to handle such errors.

(4) The `delete` operator not only destroys object but also releases allocated memory.

(5) The `new` operator creates an object and it remains in the memory until it is released using `delete` operator. Sometimes the object deleted using the `delete` operator remains in memory.

(6) If we send a null pointer to `delete` operator, it is secure. Using `delete` to zero has no result.

(7) The statement `delete x` does not destroy the pointer `x`. It destroys the object associated with it.

(8) Do not apply C functions such as `malloc()`, `realloc()` or `free()` with `new` and `delete` operators. These functions are unfit to object-oriented techniques.

(9) Do not destroy the pointer repetitively or more than one time. First time the object is destroyed and memory is released. If for the second time the same object is deleted, the object is sent to the destructor and no doubt, it will corrupt the heap.

(10) If the object created is not deleted, it occupies the memory unnecessarily. It is a good habit to destroy the object and release the system resources.

Table 11.3 shows the difference between `new` and `malloc()`.

**Table 11.3** Difference between `new` and `malloc()`

| <code>new</code>                                                                                            | <code>malloc()</code>                                                                  |
|-------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| Creates objects<br>Returns pointer of relevant type<br>It is possible to overload <code>new</code> operator | Allocates memory<br>Returns void pointer<br><code>malloc()</code> cannot be overloaded |

#### **11.4 Write a program to allocate memory to store 3 integers.**

**Use `new` and `delete` operators for allocating and deallocating memory.**

**Initialize and display the values.**

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

```

int main( )
{
    clrscr( );
    int i, *p;
    p = &i;
    p= new int[3];

    *p=2;          // first element

    *(p+1)=3;    // second element
    *(p+2)=4;    // third element

    cout <<"Value    Address";
    for (int x=0;x<3;x++)
        cout <<endl<<*(p+x)<<"\t"<<(unsigned) (p+x);

    delete []p;

    return 0;
}

```

## OUTPUT

| <b>Value</b> | <b>Address</b> |
|--------------|----------------|
| <b>2</b>     | <b>3350</b>    |
| <b>3</b>     | <b>3352</b>    |
| <b>4</b>     | <b>3354</b>    |

**Explanation:** In the above program, integer variable *i* and pointer *\*p* are declared. The pointer *p* is initialized with address of variable *i* and using *new* operator memory for three integers is allocated to it. The pointer *\*p* can hold three integers in successive memory locations. The pointer variable is initialized with numerical values. The for loop is used to display the contents of the pointer *\*p* *delete* operator releases the memory.

**11.5 Write a program to allocate memory for two objects. Initialize and display the contents and deallocate the memory.**

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

```

# include <conio.h>

struct boy
{
    char *name;
    int age;
};

int main( )
{
    clrscr( );
    boy *p;

    p=new boy[2];
    p->name="Rahul";
    p->age=20;

    (p+1)->name="Raj";
    (p+1)->age=21;
    for (int x=0;x<2;x++)
    {
        cout <<"\nName : "<<(p+x)->name<<endl<<"Age : "<<(p+x)->age;

    }
    delete []p;

    return 0;
}

```

## OUPUT

**Name : Rahul**

**Age : 20**

**Name : Raj**

**Age : 21**

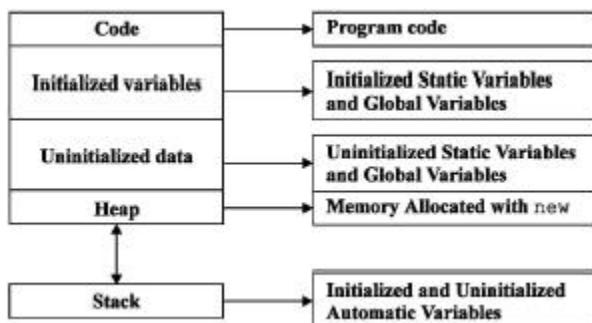
**Explanation:** In the above program, structure boy is declared and memory for two objects is allocated to the pointer p. The pointer p is initialized and the first for loop displays the contents of the pointer on the screen. Finally, the delete operator de-allocates the memory.

## 11.4 HEAP CONSUMPTION

The heap is used to allocate memory during program execution i.e., run-time. In assembly language, there is no such thing as a heap. All memory is for programmer and he/she can use it directly. In various ways, C/C++ has a better programming environment than assembly language. However, a cost has to be paid to use either C or C++. The cost is separation from machine. We cannot use memory anywhere; we need to ask for it. The memory from which we receive is called the heap.

The C/C++ compiler may place automatic variables on the stack. It may store static variables earlier than loading the program. The heap is the piece of memory that we allocate. [Figure 11.3](#) shows view of heap.

Local variables are stored in the stack and code is in code space. The local variables are destroyed when a function returns. Global variables are stored in the data area. Global variables are accessible by all functions. The heap can be thought of as huge section of memory in which large number of memory locations are placed sequentially.



**Fig. 11.33** Heap review

As stated earlier all local variables are stored in the stack. As soon as the function execution completes, local variables are destroyed and the stack becomes empty. The heap is not cleared until program execution completes. It is the user's task to free the memory. The memory allocated from heap remains available until the user explicitly deallocates it.

While solving problems related to memory allocation, we always believe that there is large memory available and the heap is at no time short of memory. It is bad for a program to depend on such guess work which may cause an error in application. In C, the function, `malloc( )` is used to allocate memory and if this function fails to allocate memory, returns null pointer. By checking the return value of function `malloc( )`, failure or success of the memory allocation is tested and appropriate sub-routines are executed.

C++ allows us to apply the same logic with `new` operator. C++ allows us to use two function pointers known as `_new_handler` and `set_new_handler`. The `_new_handler` holds a pointer to a function. It requires no arguments and returns void.

If operator `new` fails to allocate the memory requested, it will invoke the function `*_new_handler` and again it will attempt the memory allocation. By default, the function `*_new_handler` directly closes the program. It is also possible to substitute this handler with a function that releases memory. This can be accomplished by executing the function `set_new_handler` directly that returns a pointer to the handler.

### **11.6 Write a program to use `set_new_handler` function.**

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

void m_warn( )
{
    cerr << "\n Cannot allocate!";
    exit(1);
}

int main( )
{
    clrscr( );

    set_new_handler(m_warn);
    char *k = new char[50];
    cout << "\n First allocation: k = " << hex << long(k);
    k = new char[64000U];
    cout << "\n Second allocation: k = " << hex << long(k);
    set_new_handler(0); // Reset to default.

    return 0;
}
```

### **OUPUT**

**First allocation: k = 8fa40d48**

**Cannot allocate!**

**Explanation:** In the above program, the `set_new_handler` is initialized with the function `m_warn()`. The `m_warn()` displays warning message.

The variable `p` is a character pointer and by using `new` operator, memory for 50 characters is allocated to the pointer `p`. The `cout` statement displays the starting address of the memory allocated.

Consider the statement `p = new char [64 00 0U]`. In this statement the `new` operator attempts to allocate memory to pointer `p`. In case the `new` operator fails to allocate the memory, it calls the `setnew_handler` and `m_warn( )` function is executed.

## 11.5 OVERLOADING NEW AND DELETE OPERATORS

In C++ any time when we are concerned with memory allocation and deallocation, the `new` and `delete` operators are used. These operators are invoked from compiler's library functions. These operators are part of C++ language and are very effective. Like other operators the `new` and `delete` operators are overloaded. The program given next illustrates this.

### 11.7 Write a program to overload new and delete operators.

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

void main( )
{
    clrscr( );
    void warning( );
    void *operator new (size_t, int);
    void operator delete (void*);
    char *t= new ('#') char [10];
    cout <<endl<<"First allocation : p=""<<(unsigned)long(t)<<endl;

    for (int k=0;k<10;k++)
        cout <<t[k];

    delete t;
    t=new ('*') char [64000u];
    delete t;
}

void warning( )
{
```

```

        cout <<"\n insufficient memory";
        exit(1);
    }

void *operator new (size_t os, int setv)
{
    void *t;
    t=malloc(os);
    if (t==NULL)    warning( );
    memset(t, setv, os);
    return (t);
}

void operator delete(void *ss) { free(ss); }

```

## OUPUT

**First allocation : p=3376**

#####

### Insufficient memory

**Explanation:** In the above program, the `new` and `delete` operators are overloaded. The `size_t` is used to determine the size of object.

The `new` operator calls `warning()` function when `malloc()` function returns null.

Consider the statement `t=new('*') char[64000u]`. When memory allocation is requested by this statement, the `new` operator fails to allocate memory and calls the function `warning()`. The `delete` operator when called releases the memory using `free()` function. Internally, in this program `malloc()` and `free()` functions are used to allocate and deallocate the memory.

## 11.6 EXECUTION SEQUENCE OF CONSTRUCTORS AND DESTRUCTORS

Overloaded `new` and `delete` operators function within the class and are always static. We know that static function can be invoked without specifying object. Hence, this pointer is absent in the body of static functions. The compiler invokes the overloaded `new` operator and allocates memory before executing constructor. The compiler also invokes overloaded `delete` operator function and deallocates memory after execution of destructor. The following program illustrates this concept.

**11.8 Write a program to display the sequence of execution of constructors and destructors in classes when new and delete operators are overloaded.**

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

class boy
{
    char name[10];
public :
    void *operator new (size_t );
    void operator delete (void *q);
boy( );
~boy( );

};

char limit[sizeof(boy)];

boy::boy( )

{
    cout << endl << "In Constructor";
}

boy::~boy( )
{
    cout << endl << "In Destructor";
}

void *boy :: operator new (size_t s )
{
    cout << endl << "In boy ::new operator";
    return limit;
}

void boy::operator delete (void *q)
{   cout << endl << "In boy ::delete operator"; }
```

```
void main( )
{
    clrscr( );
    boy *e1;
    e1 = new boy;
    delete e1;
}
```

## OUPUT

**In boy ::new operator**

**In Constructor**

**In Destructor**

**In boy ::delete operator**

**Explanation:** In this program, the new and delete operators are overloaded. The class also has constructor and destructor. The overloaded new operator function is executed first followed by class constructor. We know that constructor is always used to initialize members. The constructor also allocates memory by calling new operator implicitly. Here, the new operator is overloaded. As soon as control of program reaches to constructor it invokes overloaded new operator before executing any statement in constructor. Similarly, when object goes out of scope destructor is executed. The destructor invokes overloaded delete operator to release the memory allocated by new operator.

## 11.7 SPECIFYING ADDRESS OF AN OBJECT

The compiler assigns address to objects created. The programmer has no power over address of the object. However, it is possible to create object at memory location given by the program explicitly. The address specified must be big enough to hold the object. The object created in this way must be destroyed by invoking destructor explicitly. The following program clears this concept.

### 11.9 Write a program to create object at given memory address.

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

class data
{
    int j;
public:
```

```

data ( int k)    { j=k; }
~data ( )        { }

void *operator new ( size_t, void *u)
{
    return (data *)u;
}

void show( )     { cout<<"j="<

```

## OUPUT

**Address of object : 0x8f800420**

**j=10**

**Explanation:** In the above program, the operator new is overloaded. The void pointer \*add is declared and initialized with address 0x420. The pointer object \*p is declared and address is assigned to it. The new operator is also invoked to allocate memory. In the same statement constructor is also invoked and a value is passed to it. The member function show( ) displays the value of a member variable. Finally, the statement t->data::~data( ); invokes the destructor to destroy the object. We can confirm the address of object by displaying it. The address of object would be same as the specified one.

## 11.8 DYNAMIC OBJECTS

C++ supports dynamic memory allocation. C++ allocates memory and initializes the member variables. An object can be created at run-time. Such

object is called as dynamic object. The construction and destruction of dynamic object is explicitly done by the programmer. The dynamic objects can be created and destroyed by the programmer. The operator `new` and `delete` are used to allocate and deallocate memory to such objects. A dynamic object can be created using `new` operator as follows:

```
ptr= new classname;
```

The `new` operator returns the address of object created and it is stored in the pointer `ptr`. The variable `ptr` is a pointer object of the same class. The member variables of object can be accessed using pointer and `->` (arrow) operator. A dynamic object can be destroyed using `delete` operator as follows:

```
delete ptr;
```

It destroys the object pointed by pointer `ptr`. It also invokes the destructor of a class. The following program explains creation and deletion of dynamic object.

### 11.10 Write a program to create dynamic object.

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

class data
{
    int x,y;

public:
data( )
{
    cout<<"\n Constructor ";
    x=10;
    y=50;
}

~data( ) { cout<<"\n Destructor"; }

void display( )
{
    cout <<"\n x="<<x;
    cout&lt;&lt;"\n y="<<y;
}</pre>
```

```

};

void main( )
{
    clrscr( );
    data *d;           // declaration of object pointer
    d= new data;       // dynamic object

    d->display( );
    delete  d;         // deleting dynamic object
}

```

## OUTPUT

### Constructor

**x=10**

**y=50**

### Destructor

**Explanation:** The d is a pointer object. The statement `d= new data` creates an anonymous object and assigns its address to pointer d. Such creation of object is called as dynamic object. The constructor is executed when dynamic object is created. The statement `d->display( )` invokes the member function and contents of members are displayed. The statement `delete d` destroys the object by releasing memory and invokes destructor.

## 11.9 CALLING CONVENTION

Calling convention means how parameters are pushed on the stack when a function is invoked. Table 11.4 describes calling convention method of few languages.

**Table 11.4** Calling convention

| Language | Order                            | Parameter passed     |
|----------|----------------------------------|----------------------|
| Basic    | In order (left to right)         | By address           |
| Fortran  | In order (left to right)         | By address           |
| C        | In reverse order (right to left) | By value and address |

|        |                                  |                                 |
|--------|----------------------------------|---------------------------------|
| C++    | In reverse order (right to left) | By value, address and reference |
| Pascal | In order (left to right)         | As values                       |

In C/C++ parameters are passed from right to left whereas in other languages such as Basic, Fortran calling convention is from left to right order. The following program explains calling convention of C++.

### 11.11 Write a program to demonstrate calling convention of C++.

!

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

void main( )
{
clrscr( );
void show (int,int);
int x=2,y=3;
show (x,y);
}

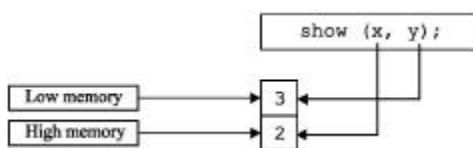
void show (int x, int y)
{
cout<<x;
cout<<endl;
cout<<y;
}
```

#### OUPUT

2

3

**Explanation:** In this program, integer variables x and y are initialized to two and three respectively. The function show ( ) is invoked and x and y are passed as per the statement show (x, y). The parameters are passed from right to left. The variable y is passed first followed by x. The values of variables with their position in the stack are displayed in [Figure 11.4](#).



**Fig. 11.4** Stack with arguments

The value of right most variable is pushed first followed by other variables in sequence.

### SUMMARY

- (1) The memory model sets the supportable size of code and data areas.
- (2) The object created and memory allocated by using `new` operator should be deleted and memory be released by the `delete` operator otherwise such mismatch operation may corrupt the heap or may crash the system. According to ANSI standard, it is a valid outcome for this invalid operation and the compiler should have routines to handle such errors.
- (3) The heap is used to allocate memory during program execution i.e., run-time. The C/C++ compiler may place automatic variables on the stack. It may store static variables before loading the program. The heap is the piece of memory that we allocate.
- (4) In C++, whenever memory allocation and deallocation is done the `new` and `delete` operators are used. These operators are part of C++ language and are very effective. Like other operators the `new` and `delete` operators are also overloaded.
- (5) When constructors and destructors are executed, `new` and `delete` operators are invoked. When these operators are overloaded, the compiler invokes these overloaded `new` and `delete` operators.
- (6) An object can be created at run-time and such an object is called as dynamic object. The construction and destruction of dynamic object is explicitly done by the programmer.
- (7) Calling convention means how parameters are pushed on the stack when a function is invoked. In C/C++ parameters are passed from right to left whereas in Basic, Fortran parameters are passed from left to right.

### EXERCISES

#### [A] Answer the following questions.

- (1) What are the different types of memory models?
- (2) Explain the properties of `new` and `delete` operators.
- (3) What do you mean by heap? Explain in detail about it.
- (4) Explain overloading of `new` and `delete` operators.
- (5) What do you mean by dynamic objects? How are they created?
- (6) What do you mean by calling conventions?
- (7) Explain the process of calling convention in C/C++.
- (8) Explain the use of `set_new_handler`.

(9) Explain the difference between the operator `new` and `malloc( )` function.

(10) What are segment and offset addresses?

(11) Explain the use of macros `FP_SEG( )` and `FP_OFF( )`.

**[B] Answer the following by selecting the appropriate option.**

(1) The dynamic objects are created

- (a) at run-time
- (b) compile time
- (c) both (a) and (b)
- (d) none of the above

(2) The `new` operator is used to

- (a) allocate memory
- (b) deallocate memory
- (c) delete object
- (d) none of the above

(3) The `delete` operator is used to

- (a) allocate memory
- (b) deallocate memory
- (c) create object
- (d) none of the above

(4) In C/C++, when function is called, parameters are passed

- (a) from right to left
- (b) from left to right
- (c) from top to bottom
- (d) none of the above

(5) In C++, it is possible to pass values to function by

- (a) call by value
- (b) call by address
- (c) call by reference
- (d) all of the above

**[C] Attempt the following programs.**

(1) Write a program to create dynamic object.

(2) Write a program to allocate memory for 10 integers to a pointer. Assign and display 10 integers. Also, destroy the object.

(3) Write a program to invoke a function with parameters. How are parameters passed? Explain whether they pass from right to left or left to right.

(4) Write a program to create object at specified memory location.

# 12

CHAPTER

## BINDING, POLYMORPHISM AND VIRTUAL FUNCTIONS

C  
H  
A  
P  
T  
E  
R

- [12.1 Introduction](#)
- [12.2 Binding in C++](#)
- [12.3 Pointer to Derived Class Objects](#)
- [12.4 Virtual Functions](#)

- [12.5 Rules for Virtual Functions](#)
- [12.6 Array of Pointers](#)
- [12.7 Pure Virtual Functions](#)
- [12.8 Abstract Classes](#)
- [12.9 Working of Virtual Functions](#)
- [12.10 Virtual Functions in Derived Classes](#)
- [12.11 Object Slicing](#)
- [12.12 Constructors and Virtual Functions](#)
- [12.13 Virtual Destructors](#)
- [12.14 Destructors and Virtual Functions](#)

## 12.1 INTRODUCTION

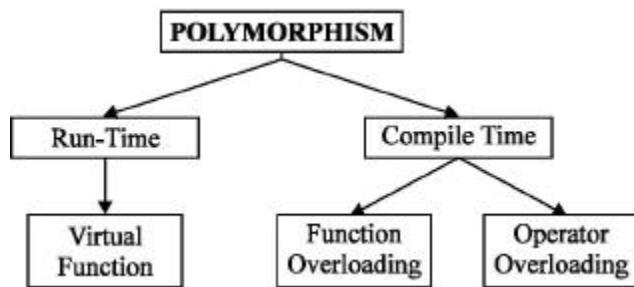
This topic is absolutely new and those who are new to object-oriented programming, must study this chapter in detail. An attempt has been made to illustrate every point of this new topic in an easy way and a bit complicated subtopics are explained in a simple way.

The word *poly* means *many* and *morphism* means *several forms*. Both the words are derived from Greek language. Thus, by combining these two words a whole new word is created called as **polymorphism** i.e., various forms. We have learnt overloading of functions and operators. It is also one-type of polymorphism. The information pertaining to various overloaded member functions is to be given to the compiler while compiling. This is called as early binding or static binding. In C++ function can be bound either at compile time

or run time. Deciding a function call at compiler time is called compile time or early or static binding. Deciding function call at runtime is called as runtime or late or dynamic binding. Dynamic binding permits suspension of the decision of choosing suitable member functions until run-time. Two types of polymorphism are shown in [Figure 12.1](#)

## POLYMORPHISM

Polymorphism is the technique in which various forms of a single function can be defined and shared by various objects to perform the operation.



**Fig.12.1** Polymorphism in C++

## 12.2 BINDING IN C++

Though C++ is an object-oriented programming language, it is very much inspired by procedural language. C++ supports two types of binding, static or early binding and dynamic or late binding.

### (1) STATIC (EARLY) BINDING

Even though similar function names are used at many places, but during their references their position is indicated explicitly. Their ambiguities are fixed at compile time. Consider the following example:

```
class first // base class
{
    int d;
public:
    void display ( ) { —— } // base class member function
};

class second : public first // derived class
{
    int k :
public:
    void display ( ) { —— } // member function of derived class
}
```

In the above program, base and derived classes have the same member function `display( )`. This is not an overloaded function because their prototype is same and they are defined in different classes.

## (2) DYNAMIC (LATE) BINDING

Dynamic binding of member functions in C++ can be done using the keyword `virtual`. The member function followed by the `virtual` keyword is called as **virtual function**. Consider the following example:

```
class first                                // base class
{
    int d;
public:
    virtual void display ( ) { —— } // base class member function
};
class second: public first                  // derived class
{
    int k;
public:
    virtual void display ( ) { —— } // member function of derived
class
}
```

In the above example, base and derived classes have the same member function `display( )` preceded by the keyword `virtual`. The various forms of virtual functions in base and derived classes are dynamically bound. The references are detected from the base class. All virtual functions in derived classes are believed as virtual, supposing they match the base class function exactly in number and types of parameters sent. If there is no match between these functions, they will be assumed as overloaded functions. The virtual function must be defined in public section. If the function is declared `virtual`, the system will use dynamic (late) binding, which is carried out at run-time otherwise, early or compile time binding is used.

### 12.1 Write a program to invoke function using scope access operator.

```
# include <iostream.h>
# include <conio.h>
class first
{
    int b;
public :
    first( ) { b=10; }
    void display ( ) { cout <<"\n b = " <<b; }
};
class second : public first

{
```

```

int d ;
public :
second( ) { d=20; }
void display ( ) { cout <<"\n d = "<<d; }
};

int main ( )
{
    clrscr( );
    second s;
    s.first::display( );
    s.display( );
    return 0;
}

```

## OUTPUT

**b = 10**

**d = 20**

**Explanation:** In the above program, the class `first` is a base class and class `second` is a derived class. Both the classes contain one integer data member and member function. The `display( )` function is used to display the content of the data member. Both the classes contain the same function name. In function `main( )`, `s` is an object of derived class `second`. Hence, in order to invoke the `display( )` function of base class, scope access operator is used. When base and derived classes have the same function name then it is very essential to provide information to the compiler at run-time about the member functions. The mechanism that provides run-time selection of function is called as **polymorphism**. The `virtual` keyword plays an important role in this process. Before introducing virtual functions let us have a look at the program given below. The following program shows what happens when functions are not declared `virtual`. This is an example of early binding.

## 12.2 Write a program to invoke member functions of base and derived classes using pointer of base class.

```

# include <iostream.h>
# include <conio.h>
class first
{ int b;
public :
first( ) { b=10; }

void display( ) { cout <<"\n b = "<<b; }
};

class second : public first
{ int d ;

```

```

public :
second( ) { d=20; }
void display( ) { cout <<"\n d = "<<d; }
};

int main( )
{
    clrscr( );
    first f,*p;
    second s;
    p=&f;
    p->display( );
    p=&s;
    p->display( );
    return 0;
}

```

## **OUTPUT**

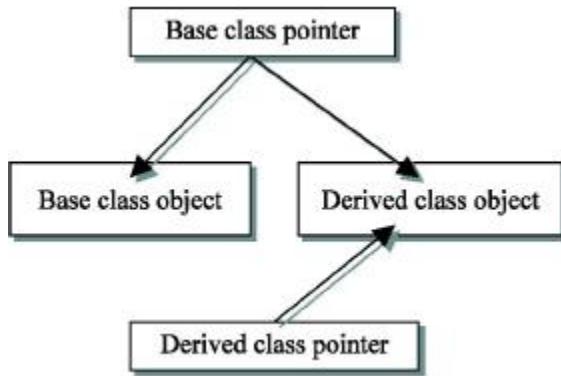
**b = 10**

**b = 10**

**Explanation:** In the above program, the class `first` is a base class and class `second` is a derived class. The variable `f` is an object of base class and `s` is an object of derived class. The pointer `*p` is an object pointer of base class. Address of base class object is assigned to pointer `p` and `display( )` function is called. Again, address of derived class is assigned to pointer `p` and `display( )` function is called. Both times `display( )` function of base class is executed. This is because second call, `p` contains address of object of derived class, but the `display( )` function of base class replaces the existence of `display( )` function of derived class. In order to execute various forms of the same function defined in base and derived classes, run-time binding is necessary and it can be achieved using `virtual` keyword.

## **12.3 POINTER TO DERIVED CLASS OBJECTS**

Inheritance provides hierarchical organization of classes. It also provides hierarchical relationship between two objects and indicates the shared properties between them. All derived classes inherit properties from the common base class. Pointers can be declared to point base or derived classes. Pointers to object of base class are type compatible with pointers to object of derived class. A base class pointer can also point to objects of base and derived classes. In other words, a pointer to base class object can point to objects of derived classes whereas a pointer to derived class object cannot point to objects of base class object as shown in [Figure 12.2](#).



**Fig.12.2** Type-compatibility of base and derived class pointers

From Figure 12.2 it is clear that a base class pointer can point to objects of base and derived classes whereas a derived class pointer cannot point to objects of base class.

### 12.3 Write a program to access members of base and derived classes using pointer objects of both classes.

```

#include <iostream.h>
#include <constream.h>

class W
{
protected:
int w;
public:

W (int k) { w=k; }

void show( )
{
cout <<"\n In base class W ";
cout<<"\n W="<<w; };

};

class X : public W
{
protected:
int x;

```

```

public:

X (int j, int k) : W( j)
{
    x=k;
}

void show( )
{
    cout <<"\n In class X ";
    cout<<"\n w="<

## OUTPUT


```

**In base class W**

**W=20**

**In base class W**

```
W=5  
In class X  
w=5  
x=2  
In class X  
w=3  
x=4
```

**Explanation:** In the above program, the class W is a base class. X is derived from W and class Y is derived from class X. Here, the type of inheritance is multilevel inheritance. The variable \*b is a pointer object of class W. The statement b = new W (20) creates a nameless object and returns its address to pointer b. The show( ) function is invoked by the pointer object b. Here, the show( ) function of base class W is invoked. Using delete operator the pointer b is deleted.

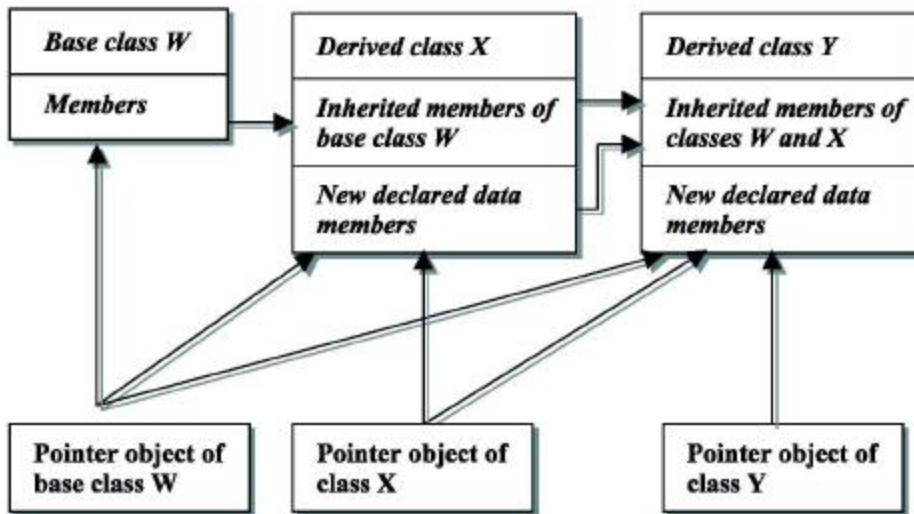
The statement b = new X(5, 2) creates a nameless object of class X and assigns its address to base class pointer b. Here, note that the object b is pointer object of base class and it is initialized with address of derived class object. Again, pointer b invokes the function show( ) of base class and not the function of class X (derived class.). To invoke the function of derived class X, following statement is used:

```
((X*)b)->show( );
```

In the above statement, **typecasting (upcasting)** is used. The upcasting forces the objects of class W to behave as if they were objects of class X. This time the function show( ) of class X (derived class) is invoked. Obtaining the address of a derived class object, and treating it as the address of the base class object is known as **upcasting**.

The statement X x(3, 4) creates object x of class X. The statement X \*d=&x declares pointer object d of derived class X and assigns address of x to it. The pointer object d invokes the derived class function show( ). Figure 12.3 shows pictorial representation of the above explanation.

Here, pointer of class W can point to its own class as well as its derived class. The pointer of derived class X can point to its own class but cannot to its base class.



**Fig.12.3** Type-compatibility of base and derived class pointers

## 12.4 VIRTUAL FUNCTIONS

Virtual functions of base classes must be redefined in the derived classes. The programmer can define a virtual function in a base class and can use the same function name in any derived class, even if the number and type of arguments are matching. The matching function overrides the base class function of the same name. Virtual functions can only be member functions. We can also declare the functions as given below:

`int Base::get(int)` and `int Derived::get(int)` even when they are not virtual.

The base class version is available to derived class objects via scope override. If they are virtual, only the function associated with the actual type of objects is available. With virtual functions we cannot alter the function type. If two functions with the same name have different arguments, C++ compiler considers them as different, and the virtual function mechanism is dropped.

## 12.5 RULES FOR VIRTUAL FUNCTIONS

- (1) The virtual functions should not be static and must be member of a class.
- (2) A virtual function may be declared as `friend` for another class. Object pointer can access virtual functions.
- (3) Constructors cannot be declared as virtual, but destructors can be declared as virtual.
- (4) The virtual function must be defined in public section of the class. It is also possible to define the virtual function outside the class. In such a case, the

declaration is done inside the class and definition is done outside the class. The virtual keyword is used in the declaration and not in function declarator. (5) It is also possible to return a value from virtual function like other functions.

(6) The prototype of virtual functions in base and derived classes should be exactly the same. In case of mismatch, the compiler neglects the virtual function mechanism and treats them as overloaded functions.

(7) Arithmetic operation cannot be used with base class pointer.

(8) If a base class contains virtual function and if the same function is not redefined in the derived classes in that case the base class function is invoked.

(9) The `operator` keyword used for operator overloading also supports virtual mechanism.

#### **12.4 Write a program to declare virtual function and execute the same function defined in base and derived classes.**

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

class first
{
    int b;
public :
    first( ) { b=10; }

    virtual void display( ) { cout <<"\n b = " <<b; }

};

class second : public first
{
    int d ;
public :
    second( ) { d=20; }

    void display( ) { cout <<"\n d = " <<d; }

};

int main( )
{
    clrscr( );
    first f,*p;
    second s;
    p=&f;
    p->display( );
```

```
p=&s;  
p->display( );  
  
return 0;  
}
```

## OUTPUT

**b = 10**

**d = 20**

**Explanation:** The above program is the same as the previous one. The only difference is that the `virtual` keyword precedes the `display()` function of the base class as per the statement `virtual void display() { cout <<"\n b = " <<b; }`. The `virtual` keyword does the runtime binding. In first call, the `display()` function of base class is executed and in second call after assigning address of derived class to pointer `p`, `display()` function of derived class is executed.

**12.5 Write a program to use pointer for both base and derived classes and call the member function. Use `virtual` keyword.**

## VIRTUAL FUNCTIONS

```
# include <iostream.h>  
# include <conio.h>  
  
class super  
{  
    public :  
  
    virtual void display () {cout <<"\n In function display () class  
super";}  
    virtual void show () { cout <<"\n In function show() class  
super";}  
};  
  
class sub : public super  
{  
    public :  
    void display () { cout <<"\n In function display() class sub ";}  
    void show () { cout <<"\n In function show() class sub ";}  
};  
  
int main()  
{  
    clrscr();
```

```

super S;
sub A;
super *point;
cout <<"\n Pointer point points to class super\n";
point=&S;
point->display( );
point->show( );
cout<<"\n\n Now Pointer point points to derived class sub\n";
point=&A;
point->display( );
point->show( );
return 0;
}

```

## **OUTPUT**

**Pointer point points to class super**

**In function display( ) class super**

**In function show( ) class super**

**Now Pointer point points to derived class sub**

**In function display( ) class sub**

**In function show( ) class sub**

**Explanation:** In the above program, the base class super and the derived class sub have the member functions with similar name. They are `display()` and `show()`. In function `main()`, the variable `S` is an object of class `super` and the variable `A` is an object of derived class `sub`. The pointer variable `point` is pointer to the base class. The address of object `S` is assigned to the pointer `S`. The pointer calls both the member functions. Similarly, the variable `A` is an object of the derived class `sub`. The address of `A` is assigned to the pointer `point` and again the pointer calls the member functions.

The member functions of base class are preceded by the keyword `virtual`. If the `virtual` keyword is removed, both the function calls execute the member functions of base class. The member functions for derived classes are not executed though the pointer has the address of the derived class.

If the `virtual` keyword is not removed, the first function calls, executes the member function of base class and later derived class.

## **12.6 ARRAY OF POINTERS**

Polymorphism refers to late or dynamic binding i.e., selection of an entity is decided at run-time. In class hierarchy, same method names can be defined that perform different tasks, and then the selection of appropriate method is done using dynamic binding. Dynamic binding is associated with object pointers. Thus, address of different objects can be stored in an array to invoke function dynamically. The following program explains this concept.

## 12.6 Write a program to create an array of pointers. Invoke functions using array objects.

```
# include <iostream.h>
# include <constream.h>
class A
{
public:
    virtual void show( )
    {
        cout <<"A\n";
    }
};

class B : public A
{
public :
    void show( )
{ cout <<"B\n"; }
};

class C : public A
{
public :
    void show( )
    {
        cout <<"C\n";
    }
};

class D : public A
{
public:
    void show( )
{
    cout <<"D\n" ;
}
};

class E : public A {
public :
```

```

void show( )
{
    cout <<"E";
}

};

void main( )
{
    clrscr( );
    A a;
    B b;
    C c;
    D d;
    E e;

    A *pa[] = {&a,&b,&c,&d,&e };

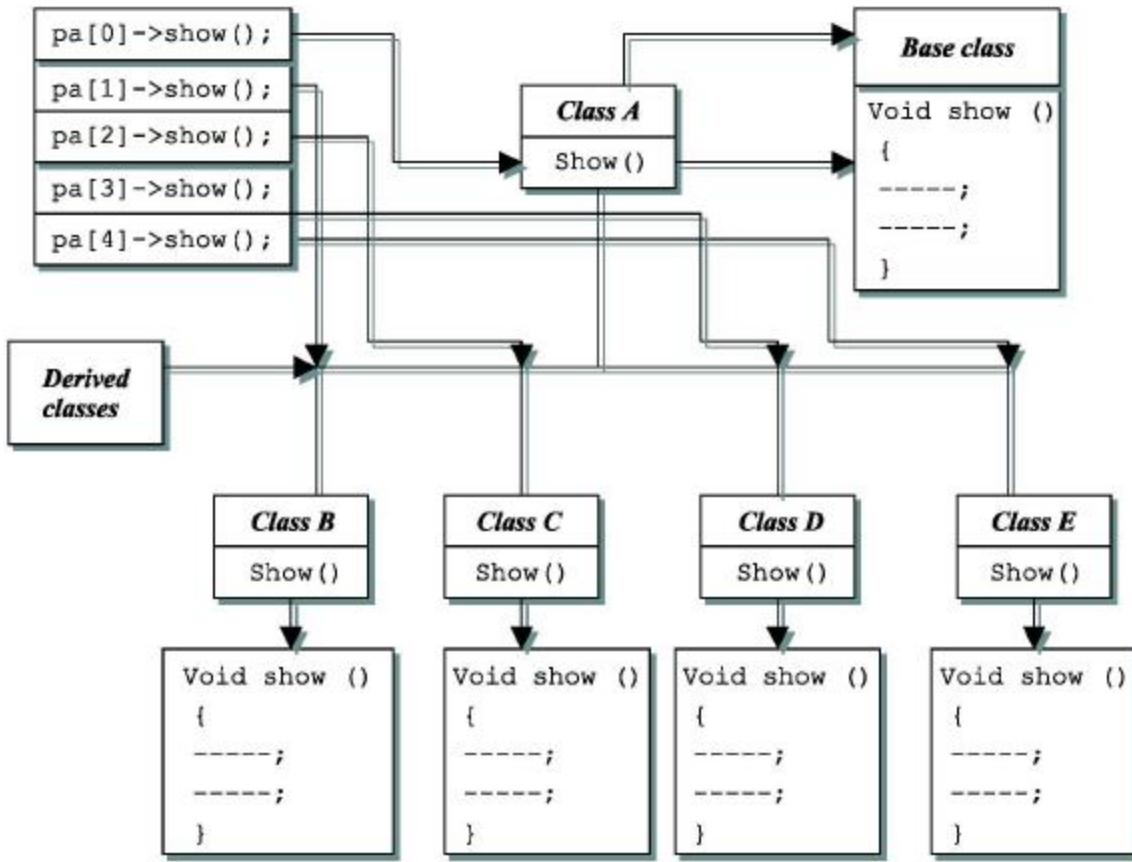
    for ( int j=0;j<5;j++)
        pa[j]->show( );
}

```

## **OUTPUT**

**A**  
**B**  
**C**  
**D**  
**E**

**Explanation:** In the above program, class A is a base class. The classes B, C, D, and E are derived from class A. All these classes have a similar function show( ). In function main(), a, b, c, d and e are objects of classes A, B, C, D and E respectively. The function show( ) of base class is declared virtual. An array of pointer \*pa is declared and it is initialized with addresses of base and derived class objects i.e., a, b, c, d and e. Using for loop and array, each object invokes function show( ). If the base class function show( ) is non-virtual then every time function show( ) of base class is executed. Figure 12.4 illustrates this concept more clearly.



**Fig.12.4** Early and late binding of functions

## 12.7 PURE VIRTUAL FUNCTIONS

In practical applications, the member functions of base classes is rarely used for doing any operation; such functions are called as **do-nothing functions**, **dummy functions**, or **pure virtual functions**. The do-nothing function or pure functions are always virtual functions. Normally pure virtual functions are defined with null-body. This is so because derived classes should be able to override them. Any normal function cannot be declared as pure function. After declaration of pure function in a class, the class becomes abstract class. It cannot be used to declare any object. Any attempt to declare an object will result in an error “cannot create instance of abstract class”.

The pure function can be declared as follows:

### DECLARATION OF PURE VIRTUAL FUNCTION

```
virtual void display ( ) =0; // pure function
```

In the above declaration, the function `display( )` is pure virtual function. The assignment operator is not used to assign zero to this function. It is used

just to instruct the compiler that the function is pure virtual function and it will not have a definition.

A pure virtual function declared in base class cannot be used for any operation. The class containing pure virtual function cannot be used to declare objects. Such classes are known as **abstract classes or pure abstract classes**. Anyone who attempts to declare object from abstract class would be reported an error message by the compiler. In addition, the compiler will display the name of the virtual function present in the base class. The classes derived from pure abstract classes are required to redeclare the pure virtual function. All other derived classes without pure virtual functions are called as **concrete classes**. The concrete classes can be used to create objects. A pure virtual function is like unfilled container, the derived class made-up to fill.

## 12.7 Write a program to declare pure virtual functions.

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

class first
{
protected:
    int b;
public :
    first( ) { b=10; }

    virtual void display ( ) =0; // pure function
};

class second : public first
{
    int d ;
public :
    second( ) { d=20; }

    void display ( ) { cout <<"\n b= "<<b <<" d = "<<d; }
};

int main( )
{
    clrscr( );
    first *p;
// p->display // abnormal program termination
    second s;
    p=&s;
    p->display( );
```

```
    return 0;
}
```

## OUTPUT

**b= 10 d = 20**

**Explanation:** In the above program, the `display()` function of the base class is declared as pure function. The pointer object `*p` holds address of objects of derived classes and invokes the function `display()` of the derived class. Here, the function `display()` of the base class does nothing. If we try to invoke the pure function using the statement `p-> display()` as per the given remark of the above program, the program is terminated with an error “abnormal program termination.”

## 12.8 ABSTRACT CLASSES

Abstract classes are like skeleton upon which new classes are designed to assemble well-designed class hierarchy. The set of well-tested abstract classes can be used and the programmer only extends them. Abstract classes containing virtual function can be used as help in program debugging. When various programmers work on the same project, it is necessary to create a common abstract base class for them. The programmers are restricted to create new base classes.

The development of such software can be demonstrated by creating a header file. The file `abstract.h` is an example of file containing abstract base class used for debugging purpose. Contents of file `abstract.h` is as follows:

Contents of `abstract.h` header file

```
# include <iostream.h>
struct debug
{
    virtual void show ( )
    {
        cout<<"\n No function show ( ) defined for this class";
    }
};
```

## 12.8 Write a program to use abstract class for program debugging.

```
# include "abstract.h"
# include <constream.h>

class A : public debug
{
    int a;
```

```

public:

A(int j=0) { a=j;
}

void show ( ) { cout <<"\nIn class A a="<<a ; }

};

class B : public debug
{
    int b;
public:
    B (int k ) { b=k; }
};

void main( ) {
    clrscr( );
    A a(1);
    B b(5);
    a. show( );
    b. show( );
}

```

## **OUTPUT**

**In class A a=1**

**No function show ( ) defined for this class**

**Explanation:** Observe the contents of file abstract.h. The structure debug contains virtual function show ( ). This file is # include in the above program. Classes A and B are derived from class debug which is defined in header file abstract.h. In function, a and b are objects of class A and B respectively. The statement a. show( ) invokes the function show( ) and value of a is displayed. The object b also invokes the function show( ). However, the class B does not have function show( ). Hence, function show( ) of abstract base class is executed which displays a warning message.

The traditional languages do not provide such facility. An abstract class develops into a dominant and powerful interface when software system undergoes various changes. It is essential to confirm that the debugging interface is accurately constructed. If changes are made in actual project, it is compulsory to add appropriate methods to abstract base classes. In case the programmer needs to define a function warn( ) to the class debug, then the header file can be updated. The contents of file would be

Contents of abstract.h header file

```

# include <iostream.h>
struct debug
{
    virtual void show( )
    {
        cout<<"\n No function show ( ) defined for this class";
    }
};

    virtual void warn( )
    {
        cout<<"\n No function warn ( ) defined for this class";
    }
};

```

While defining such abstract classes, following points must be followed:

- (1) Do not declare objects of abstract class type.
- (2) An abstract class can be used as a base class.
- (3) The derived class should not have pure virtual functions. Objects of derived classes can be declared.

## 12.9 WORKING OF VIRTUAL FUNCTIONS

Before learning mechanisms of virtual functions let's revise few points related to virtual functions.

- (1) Binding means link between a function call and the real function that is executed when function is called.
- (2) When compiler knows which function to call before execution, it is known as early binding.
- (3) Dynamic binding means that the actual function invoked at run time depends on the address stored in the pointer. In this binding, link between function call and actual function is made during program execution.
- (4) The keyword `virtual` prevents compiler to perform early binding. Binding is postponed until program execution.

The programs which follow next illustrate the working of virtual functions step by step.

### 12.9 Write a program to define virtual, non-virtual functions and determine size of the objects.

```

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

class A
{
    private :

```

```

int j;

public :
virtual void show ( ) { cout << endl << "In A class"; }
};

class B
{
private :
int j;
public :
void show( ) { cout << endl << "in B class"; }
};

class C
{
public :
void show( ) { cout << endl << "In C class"; }
};

void main( )
{
clrscr( );
A X;
B Y;
C z;

cout << endl << "Size of x = "<< sizeof (x);
cout << endl << "Size of y = "<< sizeof (y);

cout << endl << "Size of z = "<< sizeof (z);
}

```

## OUTPUT

**Size of x = 4**

**Size of y = 2**

**Size of z = 1**

**Explanation:** In the above program, the class A has only one data element of integer type but the size of the object displayed is 4. The size of object of class B that contains one integer data element is two and finally the size of object of class C is displayed one even if class C has no data element.

The function `show( )` of class A is prefixed by `virtual` keyword. Without `virtual` keyword, the size of objects would be 2,2 and 1. The size of object x with a virtual function in class A is addition of data member `int` (2 bytes) and void pointer (2 bytes). When a function is declared as `virtual`, compiler inserts

void pointer. In class C, though it has no object, the size of object displayed is 1 and this is because the compiler attempts the size of object z not to be zero since every object should have an individual address. Hence, minimum size one is considered. The minimum non-zero positive integer is one.

To perform late binding, the compiler establishes VTABLE (virtual table) for every class and its derived classes having virtual functions. The VTABLE contains addresses of the virtual functions. The compiler puts the address of the virtual functions in the VTABLE. If no function is redefined in the derived class that is defined as virtual in the base class, the compiler takes address of base class function.

When objects of base or derived classes are created, a void pointer is inserted in the VTABLE called VPTR (vpointer). The VPTR points to VTABLE. When a virtual function is invoked using base class pointer, the compiler speedily puts code to obtain the VPTR and searches for the address of function in VTABLE. In this way, appropriate function is invoked and dynamic binding takes place. The VPTR should be initialized with beginning address of the apposite VTABLE. When the VPTR is initialized with apposite VTABLE, the type of object can be determined by itself. However, it is useless if it is applied at the point when a virtual function is invoked.

Assume that a base class pointer object points to the object of derived class object. If function is invoked using the base class pointer, the compiler makes different code to accomplish the function call. The compiler begins from base class pointer. The base class pointer holds address of derived class object. With the aid of this address, the VPTR of derived class is obtained. Via VPTR, the VTABLE of derived class is obtained. In the VTABLE, the address of the function being invoked is acquired and function is called. All the above process is handled by the compiler and the user need not worry about it. The program given next makes the concept clearer.

## **12.10 Write a program to define virtual member functions in derived classes.**

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

class shop
{
    public :

    virtual void area ( ) { cout << endl << "In area of shop"; }

    virtual void rent ( ) { cout << endl << "In rent of shop"; }
```

```
void period ( ) { cout <<endl<<"In period of shop"; }

};

class shopA : public shop
{
public :

void area( ) { cout <<endl<<"In area of shopA"; }

void rent( ) { cout <<endl<<"In rent of shopA"; }

};

class shopB : public shop
{
public :

void area( ) { cout <<endl<<"In area of shopB"; }
void rent( ) { cout <<endl<<"In rent of shopB"; }
void period( ) { cout <<endl<<"In period of shopB"; }

};

class shopC: public shop

{
void area ( ) { cout <<endl<<"In area of shopC"; }

};

void main( )
{
clrscr( );
shop *p;
shop s;
p=&s;
p->area( );
p->rent( );
p->period( );

shop *sa,*sb,*sc;

shopA a;
shopB b;
shopC c;

sa=&a;
sb=&b;
```

```

sc=&c;

sa->area( );
sa->rent( );

sb->area( );
sb->rent( );

sc->area( );
sc->rent( );

sa->period( );
sb->period( );

shop d;
d.area( );

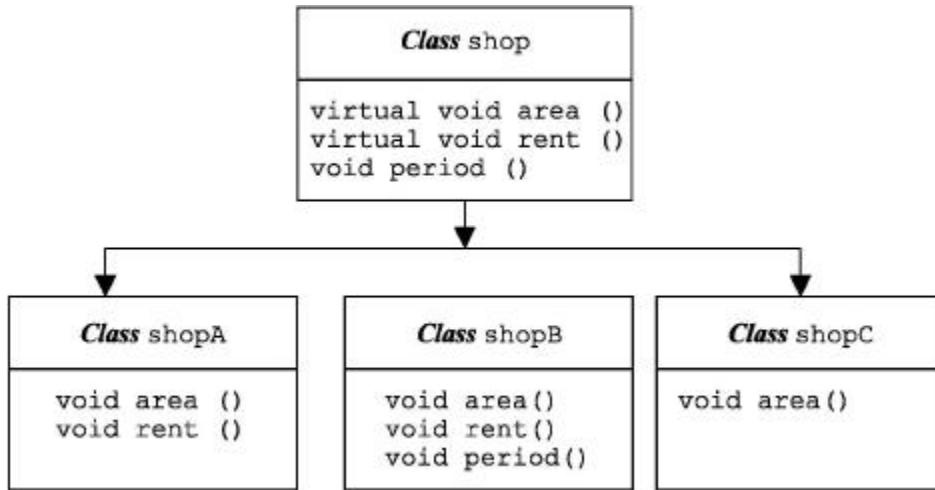
shopA e;
e. area( );
shopC f;
f..rent( );
}

```

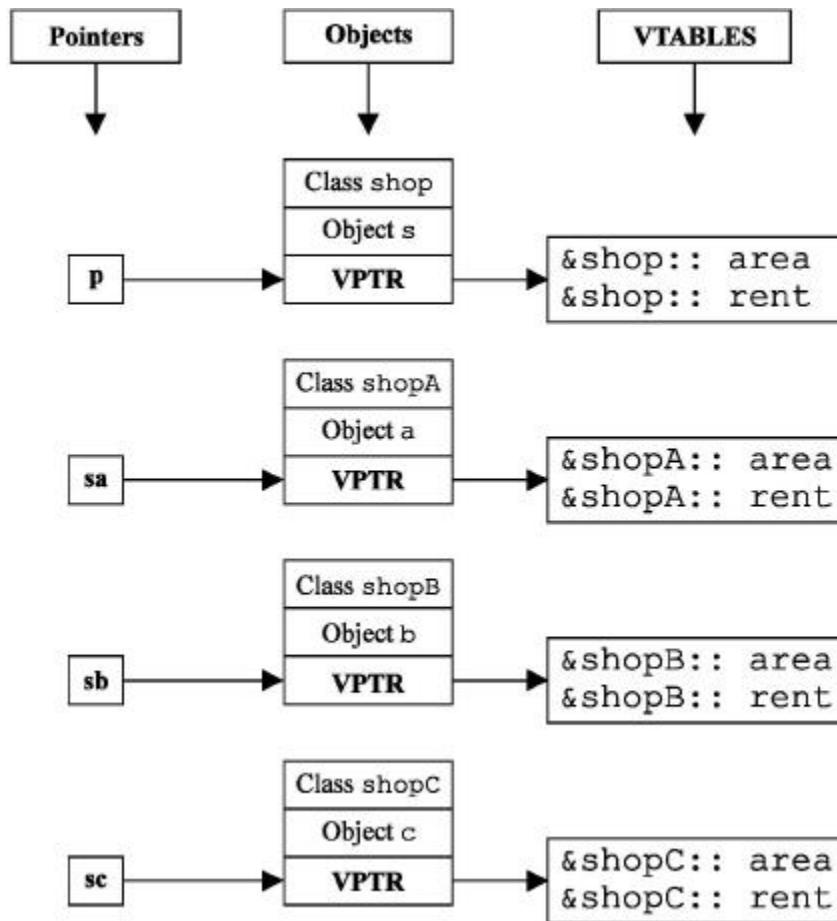
## **OUTPUT**

**In area of shop**  
**In rent of shop**  
**In period of shop**  
**In area of shopA**  
**In rent of shopA**  
**In area of shopB**  
**In rent of shopB**  
**In area of shopC**  
**In rent of shop**  
**In period of shop**  
**In period of shop**  
**In area of shop**  
**In area of shopA**  
**In rent of shop**

***Explanation:*** In the above program, four classes are declared as shown in Figure 12.5.



**Fig.12.5** Base and derived classes



**Fig.12.6** VPTR and VTABLES

As discussed before, a VTABLE is formed for each class having virtual function and for the derived class of the same class. The VTABLE is formed for the

classes: shop, shopA, shopB, and shopC. These VTABLES hold address of virtual function. Also, the compiler would place a VPTR that points to the particular VTABLE as shown in [Figure 12.6](#).

The class shopC is without function rent( ). Therefore, the VTABLE holds address of base class rent( ) function. Consider the following statements:

```
shop *p; // Base class object pointer  
shop s; // object of base class  
p=&s; // Address of s is assigned to p
```

The pointer p contains address of object s. Now consider the following statements:

```
p->area(); // Invokes function area() of base class  
p->rent(); // Invokes function rent() of base class  
p->period(); // Invokes function period() of base class
```

From the functions in the previous page, first two functions, area( ) and rent( ) are virtual and period( ) is non-virtual function. Though address of base class object or derived class object is stored in the base class pointer, the function of base class is invoked because period( ) is non-virtual function. The member function area( ) is declared as virtual in the base class. All the three derived classes shopA, shopB and shopC contain function area( ). The execution of these functions depends on address stored in the base class pointer. For example, p contains address of object a, the functions of class shopA are invoked. Similarly, storing addresses of object b and c functions of classB and classC can be invoked.

```
p=&s; // Address of s is assigned to p
```

In the above statement, p contains address of object s (base class). Therefore, when function area( ) is invoked by the pointer p, VPTR is created from the object s. With the help of VPTR, VTABLE of the class shop is obtained and address of function area( ) for class shop is accessed. With the aid of address, shop::area( ) is finally invoked. Consider the following statements:

```
shop *sa; // object pointer declaration  
shopA a; // object of derived class  
sa=&a; // assigns address of derived class object to base  
class pointer  
sa->area(); // Invokes function area()  
sa->rent(); // Invokes function rent()
```

In the statement sa=&a; address of derived class object is stored in the base class pointer. When function area( ) is invoked, the VPTR of object a is used to obtain the VTABLE of class shopA. From this VTABLE address

of shopA:: area( ) and shopA :: rent are obtained and finally invoked. Likewise, for objects b and c binding is achieved.

Consider the following statements:

```
sa->period();
sb->period();
```

Function period( ) is non-virtual function. The VTABLE is not used to call the function shop :: period( ). In addition, the function period( ) is not redefined in the derived classes. Now concentrate on following statements:

```
shop d;           //Base class object
d.area();

shopA e;         //Derived class object
e.area();

shopC f;         //Derived class object
f..rent();
```

In the above statements, dynamic binding is not performed and hence VPTR and VTABLES are not created. The functions are called as usual.

C++ places addresses of the virtual member functions in the virtual table. When call to these member functions is made, the accurate address is obtained from the virtual table. This entire procedure takes time. Therefore, the virtual function makes program execution a bit slow. Conversely, they provide better flexibility.

## 12.10 VIRTUAL FUNCTIONS IN DERIVED CLASSES

We know that when functions are declared virtual in base classes, it is mandatory to redefine virtual functions in the derived class. The compiler creates VTABLES for the derived classes and stores address of function in it. In case the virtual function is not redefined in the derived class, the VTABLE of derived class contains address of base class virtual function. Thus, the VTABLE contains address of all functions. Thus, to some extent it is not possible that a function existing has its address not present in the VTABLE. Consider the following program. It shows a possibility when a function exists but address is not located in the VTABLE. Figure 12.7 shows VTABLE for base and derived classes.

**12.11 Write a program to redefine a virtual base class function in the derived class. Also, add new member in the derived class. Observe the VTABLE.**

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

```

class A
{
    public :
        virtual void joy( ) { cout << endl << "In joy of class A"; }
};

class B : public A
{
    public :
        void joy( ) { cout << endl << "In joy of class B"; }
        void virtual joy2( ) { cout << endl << "In joy2 of class B"; }
};

void main( )
{
    clrscr( );
    A *a1,*a2;
    A a3;
    B b;
    a1=&a3;
    a2=&b;
    a1->joy( );
    a2->joy( );
    // a2->joy2( ); // joy2 is not member of class A
}

```

## **OUTPUT**

**In joy of class A**

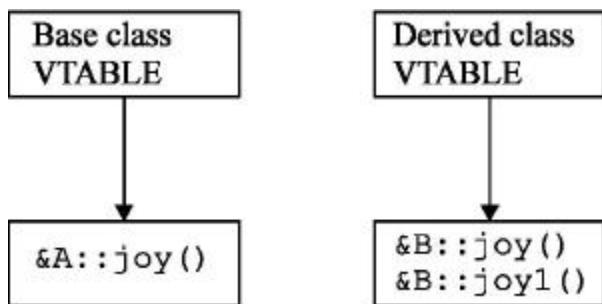
**In joy of class B**

**Explanation:** In the above program, the base class A contains one virtual function joy( ). In the derived class B, the function joy( ) is redefined and defines a new virtual function joy2( ). Figure 12.7 shows VTABLES created for base and derived classes.

a2->joy2( );

The above statement will generate an error message. In the above statement, pointer a2 is treated as only pointer to base class object. The function joy2( ) is not a member of base class A. Hence it is not allowed to invoke the function joy2( ) using the pointer object a2. However, an address of derived class is assigned to base class pointer, and the compiler in no way can determine that we are working with derived class objects. The compiler avoids calls to virtual functions present only in derived classes. In hierarchical class organization of various levels, if it is essential to invoke a function at any

level by using base class pointer, then the function should be declared virtual in the base class.



**Fig.12.7** VTABLES for base and derived classes

## 12.11 OBJECT SLICING

Virtual functions permit us to manipulate both base and derived objects using same member functions with no modifications. Virtual function can be invoked using pointer or reference. If we do so, object slicing takes place. The following program takes you to the real thing.

### 12.12 Write a program to declare reference to object and invoke function.

```
# include <constream.h>
# include <iostream.h>
class B
{
    int j;

public :
    B (int jj) { j=jj; }

virtual void joy( )

{
    cout <<" \n In class B ";
    cout <<endl<<" j= "<<j;
}

class D : public B
{
    int k;

public :
```

```

D (int jj, int kk) : B (jj)
{ k=kk; }

void joy( )
{
    B::joy( );
    cout <<" \n In class D";
cout <<endl<<" k= "<<k;
}
};

void main( )
{
    clrscr( );
    B b(3);
    D d (4,5 );
    B &r=d;
    cout <<" \n Using Object ";
    d.joy( );
    cout <<" \n Using Reference ";
    r.joy( );
}

```

## **OUTPUT**

### **Using Object**

**In class B**

**j= 4**

**In class D**

**k= 5**

### **Using Reference**

**In class B**

**j= 4**

**In class D**

**k= 5**

**Explanation:** In the above program, a reference object *r* is created to object *d* using the statement *B &r=d*. The member function *joy( )* is invoked using *r* and *d* objects. The output is same.

### **12.13 Write a program to demonstrate object slicing.**

```

# include <iostream.h>
# include <constream.h>

class A
{
    public :
        int a;

```

```

A ( ) { a=10; } ;

class B : public A
{
public :
int b;

B( ) {a=40; b=30; }

};

void main( )
{
clrscr( );

A x;
B y;
cout <<" a=" <<x.a <<" ";
x=y;
cout <<" Now a=" <<x.a;
}

```

## **OUTPUT**

**a=10 now a=40**

**Explanation:** In the above program, class A has only one data member a and derived class B has one member b. In function main ( ) object x and y of class A and B are declared. The object x has only one member i.e., a and the object y has two members a and b. The statement x=y i.e., the derived class object is assigned to base class object. In such assignment part of derived object is assigned to base class object. Thus, if an object of a derived class is assigned to a base class object, the compiler allows it. However, it copies only the base class members of the object and this process is known as object slicing.

## **12.12 CONSTRUCTORS AND VIRTUAL FUNCTIONS**

It is possible to invoke a virtual function using a constructor. Constructor makes the virtual mechanism illegal. When a virtual function is invoked through a constructor, the base class virtual function will not be called and instead of this the member function of the same class is invoked.

### **12.14 Write a program to call virtual function through constructor.**

```

# include <iostream.h>
# include <constream.h>

```

```

class B {
int k;

public :
B ( int l) { k=l; }

virtual void show ( ) { cout <<endl<<" k="<<k; }

};

class D: public B
{
    int h;

public :

D (int m, int n) : B (m)
{
    h=n;
    B *b;
    b=this;
    b->show( );
}

void show ( )
{
    cout <<endl<<" h="<<h;
}
};

```

```

void main( )
{
    clrscr( );
    B b(4);
    D d(5,2);
}

```

## **OUTPUT**

**h=2**

**Explanation:** In the above program, the base class B contains a virtual function. In the derived class D the same function is redefined. Both the base and derived classes contain constructors. In the derived class constructor, a base class pointer \*b is declared. We know that this pointer contains address of object calling the member function. Here, the this pointer holds address of

object d. The pointer object b invokes function show( ). The derived class show( ) function is invoked.

Here, though the object d is not fully constructed it still invokes the member function of the same class. This is possible because a virtual function call reaches ahead into inheritance.

### 12.13 VIRTUAL DESTRUCTORS

We know how virtual functions are declared. Likewise, destructors can be declared as virtual. The constructor cannot be virtual, since it requires information about the accurate type of the object in order to construct it properly. The virtual destructors are implemented in the way like virtual functions. In constructors and destructors pecking order (hierarchy) of base and derived classes is constructed. Destructors of derived and base classes are called when a derived class object addressed by the base class pointer is deleted.

For example, a derived class object is constructed using new operator. The base class pointer object holds the address of the derived object. When the base class pointer is destroyed using delete operator, the destructor of base and derived classes is executed. The following program explains this:

### 12.15 Write a program to define virtual destructors.

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

class B {
public :

B ( ) { cout << endl << "In constructor of class B"; }

virtual ~B ( ) { cout << endl << "In destructor of class B" ; }

};

class D : public B
{
public :

D( ) { cout << endl << "In constructor of class D"; }

~ D( ) { cout << endl << "In destructor of class D" ; }

};

void main ( )
```

```
{  
    clrscr( );  
    B *p;  
    p= new D;  
    delete p;  
}
```

## OUTPUT

**In constructor of class B**

**In constructor of class D**

**In destructor of class D**

**In destructor of class B**

**Explanation:** In the above program, the destructor of the base class B is declared as virtual. A dynamic object is created and the address of nameless object created is assigned to pointer p. The new operator allocates memory required for data members. When object goes out of scope it must be deleted and the same performed by the statement delete p. When derived class object is pointed by the base class pointer object, in order to invoke base class destructor, virtual destructors are useful.

## 12.14 DESTRUCTORS AND VIRTUAL FUNCTIONS

When a virtual function is invoked through a non-virtual member function, late binding is performed. When call to virtual function is made through the destructor, the redefined function of the same class is invoked. Consider the following program:

### 12.16 Write a program to call virtual function using destructors.

```
# include <iostream.h>  
# include <conio.h>  
  
class B  
{  
public :  
  
~ B ( ) { cout << endl << " in virtual destructor"; }  
virtual void joy( ) { cout << endl << "In joy of class B"; }  
};  
  
class D : public B  
{  
public :  
~ D ( )  
{  
    B *p;  
    p=this;
```

```

p->joy( );
}

void joy( ) { cout << endl << " in joy( ) of class D"; }

};

void main( ) {
    clrscr( );
    D X;
}

```

## OUTPUT

**in joy( ) of class D  
in virtual destructor**

**Explanation:** In the above program, in destructor of the derived class function joy( ) is invoked. The member function joy( ) of derived class is invoked followed by virtual destructor.

## SUMMARY

- (1) The word **poly** means **many** and **morphism** means **several** forms. Both the words are derived from Greek language. Thus, by combining these two words a whole new word is created called as polymorphism i.e., various forms.
- (2) The information pertaining to various overloaded member functions is to be given to the compiler while compiling. This is called as early binding or static binding. Deciding function call at run-time is called as run-time or late or dynamic binding. Dynamic binding permits to suspend the decision of choosing suitable member functions until run-time.
- (3) Pointers to objects of base classes are type compatible with pointers to objects of derived classes. Reverse is not possible.
- (4) Virtual functions of base class must be redefined in the derived classes. The programmer can define a virtual function in a base class, and then can use the same function name in any derived class.
- (5) Address of different objects can be stored in an array to invoke function dynamically.
- (6) In practical applications, the member function of the base class is rarely used for doing any operation; such functions are called as do-nothing functions, dummy functions, or pure virtual functions.
- (7) All other derived classes without pure virtual functions are called as **concrete classes**.

(8) Abstract classes are like skeleton upon which new classes are designed to assemble well-designed class hierarchy. They are not used for object declaration.

(9) Virtual function can be invoked using pointer or reference.

(10) If an object of a derived class is assigned to a base class object, the compiler allows it. However, it copies only the base class members of the object, this process is known as object slicing.

(11) It is possible to invoke a virtual function using a constructor. Constructor makes illegal the virtual mechanism.

(12) We learned how to declare virtual functions. Likewise, destructors can be declared as virtual. The constructor cannot be virtual. The virtual destructors are implemented in the same way like virtual functions. Destructors of derived and base classes are called when a derived class object addressed by the base class pointer is deleted.

### EXERCISES

**[A] Answer the following questions.**

- (1) What is polymorphism?
- (2) Explain compile time and run-time binding.
- (3) Explain the use of virtual keyword.
- (4) What are pure functions? How are they declared?
- (5) Is it possible to declare an object of the class that contains pure function?
- (6) What is the difference between virtual function and virtual classes?
- (7) What are virtual destructors?
- (8) How C++ compiler accomplishes dynamic binding?
- (9) Where do we use virtual functions? Give its applications.
- (10) What is early binding and late binding?
- (11) Explain object slicing.
- (12) Explain virtual destructors.
- (13) What are abstract classes? How can they be used for debugging a program?
- (14) What are VPTR and VTABLE? Explain in detail.
- (15) Describe rules for declaring virtual functions.
- (16) What is the difference between base class pointer and derived class pointer?

**[B] Answer the following by selecting the appropriate option.**

- (1) Consider the statement `virtual void display( ) = 0.` The `display( )` function is
  - (a) pure virtual function
  - (b) pure member function

- (c) normal function
  - (d) all of the above
- (2) The do-nothing function is nothing but
- (a) pure virtual function
  - (b) pure member function
  - (c) both (a) and (b)
  - (d) none of the above
- (3) Static binding is done at the time of
- (a) compilation of the program
  - (b) at run-time
  - (c) both (a) and (b)
  - (d) none of the above
- (4) Dynamic binding is done using the keyword
- (a) virtual
  - (b) inline
  - (c) static
  - (d) void
- (5) The virtual keyword solves the
- (a) ambiguity in base and derived classes
  - (b) ambiguity in derived classes
  - (c) ambiguity in base classes
  - (d) none of the above
- (6)  $B$  is a base class object and  $D$  is derived class object. The statement  $B=D$
- (a) copies all elements of object  $d$  to object  $b$
  - (b) copies only base portion of object  $d$  to  $b$
  - (c) copies only derived portion of object  $D$  to  $B$
  - (d) none of the above
- (7) When a base class is not used for object declaration it is called as
- (a) abstract class
  - (b) container class
  - (c) concrete class
  - (d) derived class
- (8) The derived class without pure virtual function is called as
- (a) concrete class
  - (b) abstract class
  - (c) container class
  - (d) derived class
- (9) A pointer to base class object can hold address of
- (a) only derived class object

- (b) only base class object
- (c) address of base class object and its derived class object
- (d) none of the above

**[C] Attempt the following programs.**

- (1) Write a program to declare a function `show()` in base and derived class. Display message through the function to know name of the class whose member function is executed. Use late binding concept using virtual keyword.
- (2) Write a program to define class A, B and C. The class C is derived from A and B. Define `count()` member function in all the classes as virtual. Count number of objects created.
- (3) Write a program to declare matrix class which has data member integer array as 3 X 3. Derive class matrix A from class matrix and matrix B from matrix A. All these classes should have a function `show()` to display the contents. Read and display the elements of all three matrices.
- (4) Write a program to declare reference object. Invoke member functions of the class using reference object.
- (5) Write a program to demonstrate object slicing.
- (6) Write a program to demonstrate use of abstract classes.
- (7) Write a program to redefine a virtual base class function in the derived class. Also, add new member in the derived class. Observe the VTABLE.
- (8) Write a program to define virtual, non-virtual functions and determine size of the objects.
- (9) Write a program to invoke member functions of base and derived classes using pointer of base class.
- (10) Write a program to access members of base and derived classes using pointer objects of both classes.

**[D] Find the bugs in the following programs.**

(1)

```
class B { virtual void display ()=0; };
void main () { B d; }
```

(2)

```
class B

{
public:
    virtual void display ()
{ cout <<"\n no function display defined in this class."; }
};

struct D : B { };
```

```
void main( )
{
    D d;
    d.display( );
}
```

(3)

```
class B
{
    int a, b, c;
public :
    B ( ) { a=10, b=20,c=40; }
};

class D : public B { };

void main ( )
{
B b;
D d;
d=b;
}
```

(4)

```
class B { };

class D : public B
{
    int i,j,k;
public:
    D ( ) { i=5; j=10; k=15; } };

void main( )
{
    B b;
    D d;
    b=d;
    cout <<b.i;
}
```

(5)

```
class B
{ public :
```

```
B ( ) { cout << endl << "In constructor of class B"; }
virtual ~B ( ) =0;
};

class D : public B
{   public :
    D ( ) { cout << endl << "In constructor of class D"; }
    ~D ( ) { cout << endl << "In destructor of class D"; }
};

void main( )
{
B *p;
p= new D;
delete p;
}
```

# 13

CHAPTER

## Applications with Files

C  
H  
A

—• 13.1 Introduction

—• 13.2 File Stream Classes

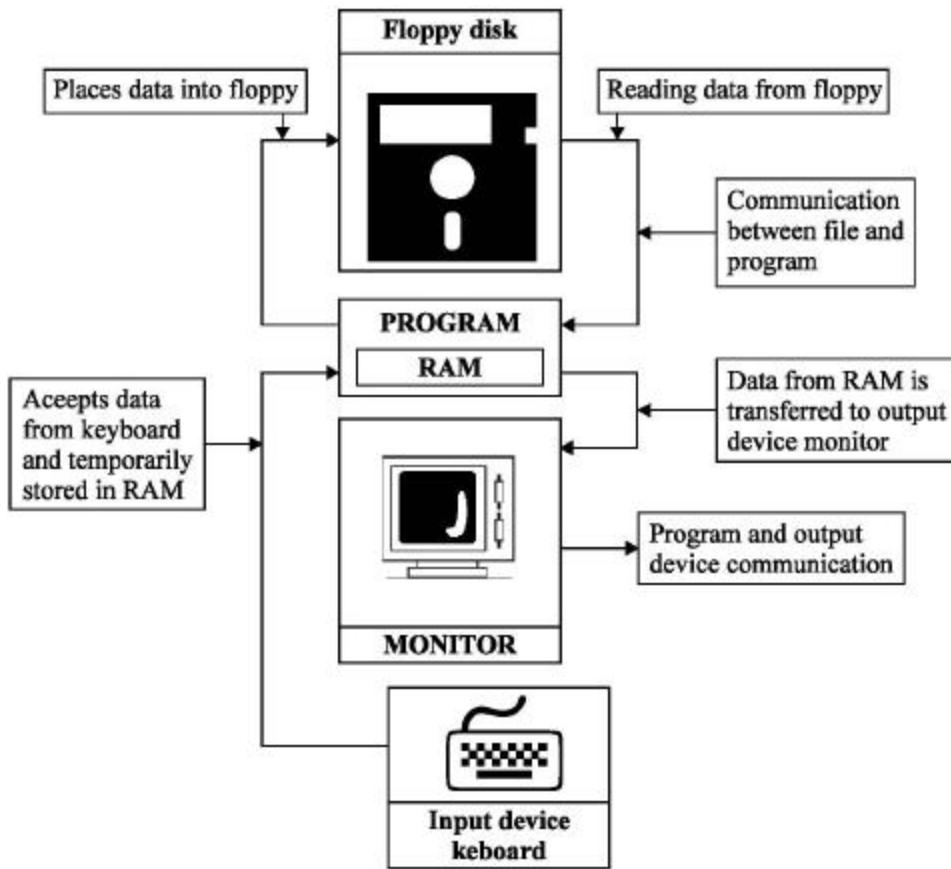
- [13.3 Steps of File Operations](#)
- [13.4 Checking for Errors](#)
- [13.5 Finding End of a File](#)
- [13.6 File Opening Modes](#)
- [13.7 File Pointers and Manipulators](#)
- [13.8 Manipulators with Arguments](#)
- [13.9 Sequential Read and Write Operations](#)
- [13.10 Binary and ASCII Files](#)
- [13.11 Random Access Operation](#)
- [13.12 Error Handling Functions](#)
- [13.13 Command Line Arguments](#)
- [13.14 Strstreams](#)
- [13.15 Sending Output to Devices](#)

## 13.1 INTRODUCTION

With the advancement of information technology a great amount of information is available on Internet. Huge amount of data is processed in computer networking. The information can be uploaded or downloaded from the desktop computer. The information transfer in computer networking in

day-to-day life is in the form of files. The files can be written, read, or updated depending upon the applications. The data is saved in the file on the disk. The file is an accumulation of data stored on the disk. The stored data can be retrieved or updated. The console I/O function reads data through input devices and displays on the screen. The data read through these functions are temporarily stored in variables or in arrays. The data stored vanishes when the program ends. In order to store the data permanently, we need to apply disk I/O function. The disk I/O functions are associated with disk files. These functions perform read and write operations with the file. Like other languages, C++ also provides function that allows a programmer to perform read/write operation with the file. The disk I/O functions in C++ are very different as compared to C, though the entire disk I/O functions of C can be used in C++. However, these functions are not appropriate with the object-oriented conditions.

Secondary storage devices such as floppy disks, hard disks etc. are used to store data in the form of file. The main memories of computer such as random access memory or read only memory are not used for storage of files. This is because the main memory of computer is limited and cannot hold large amount of data. Another reason is that the main memory is volatile i.e., when the computer is switched off the contents of RAM are vanished. The operating system is a set of system programs. The operating system provides programs for file management and controlling storage devices such as hard disks, floppy disks etc., as shown in Figure 13.1.

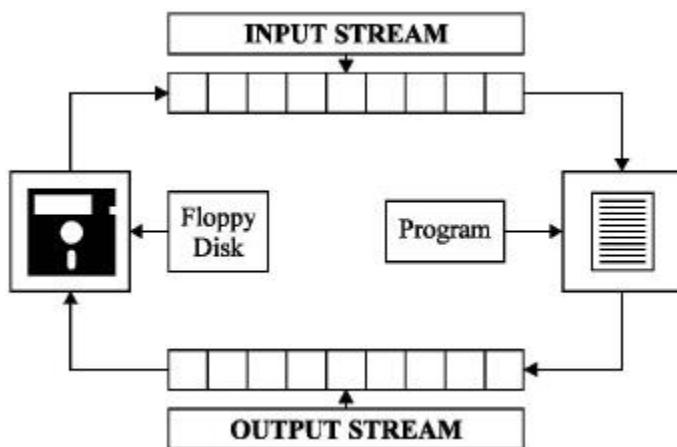


**Fig. 13.1** Communication between program, file and output device

As shown in [Figure 13.1](#), data read from keyboard are stored in variables. Variables are created in RAM (type of primary memory). On applying disk, I/O operations selected or all variables created in RAM can be stored to secondary storage devices such as hard disk or floppy disk. In [Figure 13.1](#) floppy disk is indicated. It is also possible to read data from secondary storage devices. When data is read from such devices it is placed in the RAM and then using console I/O operations it is transferred to screen. RAM is used to hold data temporarily. The variables are used to store data during program execution and the variables are created in RAM. Hence, secondary storage device RAM plays an important role.

The file is an accumulation of data stored on the disk created by the user. The programmer assigns file name. The file names are unique and are used to identify the file. No two files can have the same name in the same directory. There are various types of files such as text files, program files, data files, executable files etc. Data files contain a combination of numbers, alphabets, symbols etc. called as *data*.

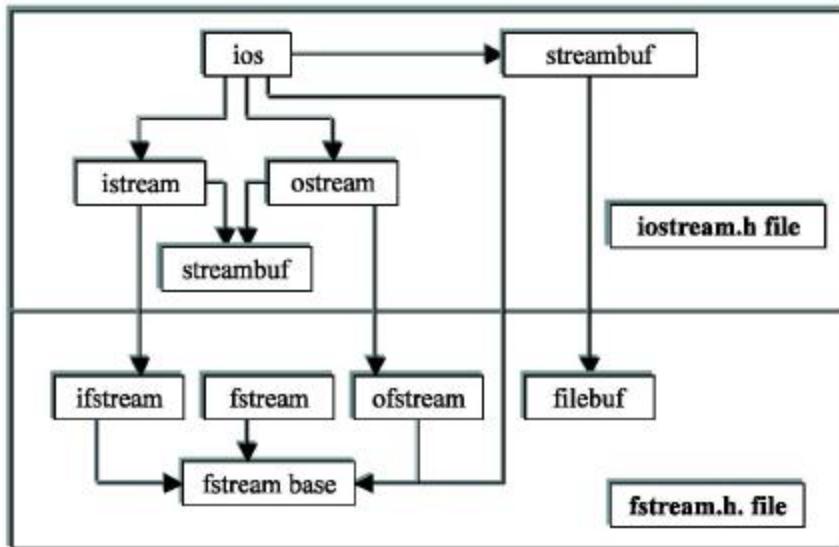
Data communication can be performed between program and output devices or files and program. File streams are used to carry the communication among above-mentioned devices. **The stream is nothing but flow of data in bytes in sequence.** If data were received from input devices in sequence then it is called as source stream and when the data were passed to output devices then it is called as destination stream. Figure 13.2 shows the input and output streams. The input stream brings data to the program and the output stream collects data from the program. In this way, input stream extracts data from the file and transfers it to the program while the output stream stores the data into the file provided by the program.



**Fig. 13.2** Input and output streams

### 13.2 FILE STREAM CLASSES

Stream is nothing but flow of data. In object-oriented programming the streams are controlled using the classes. The operations with the files are mainly of two types. They are read and write. C++ provides various classes as shown in Figure 13.3 to perform these operations. The `ios` class is the base class. All other classes are derived from the `ios` class. These classes contain several member functions that perform input and output operations. The `streambuf` class has low-level routines for controlling data buffer.



**Fig. 13.3** iostream class summary

The `istream` and `ostream` classes control input and output functions respectively. The `ios` is the base class of these two classes. The member functions of these classes handle formatted and unformatted operations (for more details about formatted and unformatted functions please refer to Chapter “**INPUT and OUTPUT in C++**”). The functions `get( )`, `getline( )`, `read( )` and overloaded extraction operators (`>>`) are defined in the `istream` class. The functions `put( )`, `write( )` and overloaded insertion operators (`<<`) are defined in the `ostream` class.

The `iostream` class is also a derived class. It is derived from `istream` and `ostream` classes. There are also another three useful derived classes. They are `istream_withassign`, `ostream_withassign`, and `iostream_withassign`. They are derived from `istream`, `ostream`, and `iostream` classes, respectively.

The classes `ifstream` and `ofstream` are derived from `istream` and `ostream` respectively. These classes handle input and output with the disk files. The header file `fstream.h` contains declaration of `ifstream`, `ofstream` and `fstream` classes including `iostream.h` file. This file should be included in the program while doing disk I/O operations. **THE FILEBUF** Filebuf accomplishes input and output operations with files. The `streambuf` class does not organize streams for input or output operations. The derived classes of `streambuf` perform these operations. It also arranges a space for keeping input data and for sending output. The I/O

functions of classes `istream` and `ostream` invoke the `filebuf` functions to perform the insertion or extraction on the streams. It holds constant `openprot` used in function `open( )` and `close( )` as a member.

**THE FSTREAMBASE** The `fstreambase` acts as a base class for `fstream`, `ifstream`, and `ofstream`. The functions such as `open( )` and `close( )` are defined in `fstreambase`.

**THE IFSTREAM** This class is derived from `fstreambase` and `istream` by multiple inheritance. It can access the member functions such as `get( )`, `getline( )`, `seekg( )`, `tellg( )` and `read( )`. It allows input operations and provides `open( )` function with default input mode.

**THE OFSTREAM** This class is derived from `fstreambase` and `ostream` classes. It can access the member functions such as `put( )`, `seekp( )`, `write( )` and `tellp( )`. It allows output operations and provides member function `open( )` with default output mode.

**THE FSTREAM** Allows simultaneous input and output operations on a `filebuf`. The member functions of base classes `istream` and `ostream` start the input and output. For example, `fstream` invokes the member function `istream::getline( )` to read characters from the file. It provides `open( )` function with default input mode.

### 13.3 STEPS OF FILE OPERATIONS

Before performing file operations, it is necessary to create a file. Operation of a file involves the following basic activities:

- File name
- Opening file
- Reading or writing the file (File processing)
- Detecting errors
- Closing the file

The file name can be a sequence of characters, called as a string. Strings are always declared with character array. Using file name a file is recognized. The length of file name depends on the operating system, for example, WINDOWS-98 supports long file names whereas MS-DOS supports only eight characters. A file name also contains extension of three characters. The file name should not be device name such as LPT1, CON etc. The list of device name is given in Table 13.6. You might have observed the `.cpp` extension to the C++ program

file name separated by dot (.). The extension is optional. The following file names are valid in MS-DOS and WINDOWS-98 operating systems.

```
data.dbf      // extension is .dbf  
tc.exe       // extension is .exe  
Prg.cpp      // extension is .cpp  
Prg.exe      // extension is .exe  
Prg.obj      // extension is .obj  
Marks        // without extension
```

The first step in the disk file I/O operation is creation of a file stream object and connecting it with the file name. The classes `ifstream`, `ofstream`, and `fstream` can be used for creating file stream defined in the header file `fstream.h`. The selection of the class is according to the operation that is to be carried out with the file. The operation may be read or write. There are two methods for opening of a file.

### **Constructor of the class**

**Member function** `open( )`

#### **(1) CONSTRUCTOR OF THE CLASS**

When objects are created, constructor is automatically executed and objects are initialized. In the same way, the file stream object is created using suitable class and it is initialized with the file name. The constructor itself uses the file name as the first argument and opens the file. The class `ofstream` creates output stream objects and `ifstream` creates input stream objects.

Consider the following examples:

- (i) `ofstream out ("text");`
- (ii) `ifstream in ("list");`

In statement (i), `out` is an object of the class `ofstream` and file name `text` is opened and data can be written to this file. The file name `text` is connected with the object `out`. Similarly, in statement (ii), `in` is an object of the class `ifstream`. The file `list` is opened for input and connected with the object `in`. It is also valid to use same name for input and output operations. It is possible to use these file objects in program statements like stream objects. Consider the following statements:

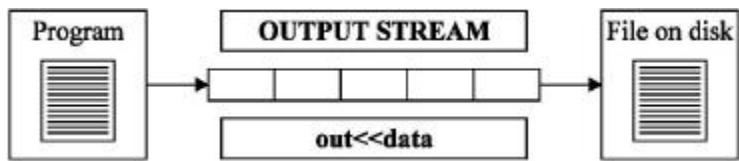
```
cout<<"One Two Three";
```

The above statement displays the given string on the screen.

```
out<<"One Two Three";
```

The above statement writes the specified string into the file pointed by the object `out` as shown in [Figure 13.4](#). The insertion operator `<<` has been

overloaded appropriately in the `ostream` class to write data to appropriate stream.



**Fig. 13.4** Interaction between `fstream` object and disk file

Consider the following statements:

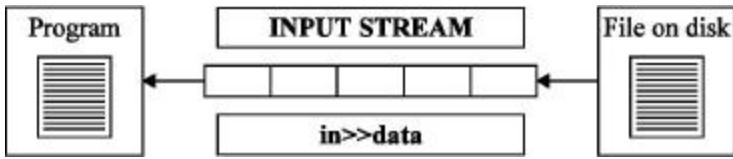
```
out<<"One Two Three"; // Write string to the file  
out<<age;           // Write contents of variable  
out<<5000;          // Write 5000 to file  
out<<'A';          // Write character 'A' to file  
out<<123.14;        // Write 123.14 to file  
out<<123.34<<endl<<5000<<'A'; //Multiple values in one statement
```

In the last statement we have separated each value using `endl` manipulator. It is essential because when we stored values such as 15 and 123.34 they are stored in the file as strings i.e., 123.34 would be stored as '1','2','3','.', '3','4'. The value 123.34 requires four bytes but when stored in a file occupies six bytes. When we need to store more numeric values more bytes are occupied and file size is increased. Every data item must be separated by delimiter. This is essential because during read operation, the extraction operator cannot determine where one number is ending and another is beginning. The same problem can be observed while reading strings. The above limitation can be overcome using binary file operation that is discussed later in the same chapter.

Similar are the following statements:

```
in>>string; // Reads string from the file where string is a character array  
in>>num;    // Reads number from the file where num is an integer variable
```

In the above statements, the object `in` reads data from the file associated with it as shown in [Figure 13.5](#). For reading data from a file we have to create an object of `ifstream` class. The new line character ( "`\n`" ) inserted at the end of every line helps the overloaded operator to separate the various items stored in the file. When numbers are read back from file they are converted into their binary format.



**Fig. 13.5** Interaction between `ifstream` object and disk file

### 13.1 Write a program to open an output file using `fstream` class.

```

# include <fstream.h>
# include <conio.h>

int main ( )
{
    clrscr( );
    ofstream out("text");
    out <<"One Two Three Four\n";
    out << "1    2    3    4";
    out <<"\n** The End ** ";
    return 0;
}
  
```

**Explanation:** In the above program, the statement `ofstream out ("text")` text is opened and connected with the object `out`.

#### Use of object `out`

```

out <<"One Two Three Four \n";
out << "1    2    3    4";
out <<"\n** The End ** ";
  
```

The above statements write data (enclosed in quotation marks) to the file pointed by the object `out` i.e., `text`. The operator `<<` is an insertion operator and writes the data in the file named `text`. The contents of the file can be seen by the user and it appears as follows:

#### Contents of the file `text`

```

One Two Three Four
1    2    3    4
** The End ** 
  
```

While opening a file for writing operations the user should give the new file name. If the given file already exists, its contents are erased and treated as a new file.

### 13.2 Write a program to read data from file using object of `ifstream` class.

```

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

```

int main ( )
{
    clrscr( );
    char *n;
    ifstream in("text");           // Opens a file in read mode
    in >>n;                      // Reads string from file
    cout<<n <<" ";               // Displays string to the console
    in >>n ;
    cout<<n <<" ";
    in >>n ;
    cout<<n <<" ";
    return 0;
}

```

## **OUTPUT**

### ***One Two Three***

**Explanation:** In the above program, the statement `ifstream in ("text")` is opened and connected with the object `in`. The statement `in >>n` reads data from file and assigns it to the variable followed by the (`>>`) extraction operator. Here, the variable `*n` is a character pointer. If numeric value is read using character type, data cannot be used for arithmetic operations. If character type data is read using numeric type variable, ASCII values of character read will be displayed. Hence, the variable type should match with data type. The statement `cout<<n <<" ";` displays the data stored in variable `n`.

In the above programs, the file associated with the objects, are automatically closed when the stream object goes out of scope. In order to explicitly close the file following statement is used:

### ***Closing of files***

```

out.close();
in.close();

```

Here, `out` is an object and `close()` is a member function that closes the file connected with the object `out`. Similarly, file associated with object `in` is closed by the member function `close()`.

### **13.3 Write a program to write and read text in a file.**

#### **Use `ofstream` and `ifstream` classes.**

```

#include <fstream.h>
#include <conio.h>

```

```

int main ( )
{
    clrscr( );

```

```

char name[15];
int age;
ofstream out("text");
cout <<"\n Name : ";
cin >>name;
cout <<"Age    : ";
cin>>age;
out<<name<<"\t";
out<<age <<"\n";
out.close( );      // File is closed

ifstream in ("text");
in>>name;
in>>age;
cout <<"\nName\t: "<<name<<"\n";
cout <<"Age      : "<<age;
in.close( );
return 0;
}

```

## **OUTPUT**

**Name : Sameer**

**Age : 24**

**Name : Sameer**

**Age : 24**

***Explanation:*** The above program, is a combination of the last two programs.

The `out` is an object of `ofstream` class and it is linked with the output file `text`. The data entered by the user is written in the file `text`.

The `close( )` function closes the file associated with the object `out`. Again, the same file is opened by `in` object of class `ifstream` for reading purpose. The data recently written is read and displayed on the screen. Thus, in a program both write and read operations are performed. The use of function `close( )` is essential to change the mode of the file. In case if a file is not closed and again an attempt is made to open it in another mode, no compile time error is generated. However, the result will not be satisfactory.

## **(2) THE OPEN( ) FUNCTION**

In the last few programs, we have studied how files could be opened for reading and writing using constructors. Now the second approach can be used with `open( )` function. The function `open( )` is used to open a file.

The `open( )` function uses the stream object. The `open( )` function has two arguments. First is file name and second is the mode. The mode specifies the purpose of opening a file i.e., read, write, append etc. The details are discussed

in section 13.5. In the following examples, default mode is considered. The default values for ifstream is (ios:: in), reading only and fstream is (ios : : out), writing only.

#### (A) Opening file for write operation

```
ofstream out;           // Creates stream object out  
out.open ("marks.dbf"); // Opens file and links with the  
object out  
out.close ( )          // Closes the file pointed by the  
object out  
out.open ("result.dbf") // Opens another file
```

#### (B) Opening file for read operation

```
ifstream in;           // Creates stream object in  
in.open ("marks.dbf"); // Opens file and links with the object in  
in.close( ) ;          // Closes the file pointed by object in  
in.open ("result.dbf"); // Opens another file
```

### 13.4 Write a program to open multiple files for writing and reading purposes. Use open( ) function.

```
# include <fstream.h>  
# include <conio.h>  
  
int main ( )  
{  
    clrscr( );  
    ofstream out;  
  
    // Writing data //  
  
    out.open ("months"); // Opens file  
    out<<"March\n";      // Writes string to the file  
  
    out<<"April\n";  
    out <<"June\n";  
    out.close( );         // Closes the file  
    out.open ("days");    // Opens another file  
    out <<"31\n";  
    out <<"30\n";  
    out <<"30\n";  
    out.close( );         // closes the file
```

```

// reading data //

# define T 20

char text[T];
ifstream in;
in.open ("months");    // opens file for reading
cout <<"\nMonth Names \n";

while (in)
{
    in.getline(text,T);
    cout<<text<<"\n";
}

in.close( );

in.open ("days");
cout <<"\nDays \n";
while (in)
{
    in.getline(text,T);
    cout<<text<<"\n";
}

in.close( );
return 0;
}

```

## **OUTPUT**

**Month Names**

**March**

**April**

**June**

**Days**

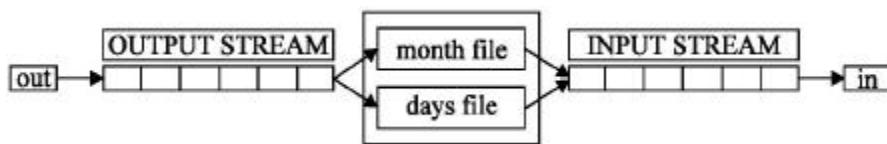
**31**

**30**

**30**

**Explanation:** In the above program, `out` is an object of `ofstream` class. The object `out` is used with `open ( )` function to open files for write operation.

Two files are opened i.e., months and days. Month names are written in file `months` and number of days is written in the file `days`. The data is written in the files one after another. The same object is used for both the files; hence, it is necessary to close previously linked file with the object before opening new file. In this program, `month` file is opened first and after closing it the `days` file is opened. The files are closed using the function `close( )`. The same procedure is implemented for opening the same files for reading purpose. When end of file is reached, the `while` loop terminates.



**Fig. 13.6** Stream objects with multiple files

As shown in [Figure 13.6](#) above the `out` object opens `months` and `days` file one after another for writing. The object `in` opens these files for reading. Before opening next file, the previously associated file with the object is closed. A single object cannot open multiple files simultaneously.

### 13.4 CHECKING FOR ERRORS

Until now, we carried out the file operations without thinking whether it is performed successfully or not. If any fault occurs during file operation, fault cannot be detected. Various errors can be made by the user while performing file operation. Such errors must be reported in the program to avoid further program failure. When the user attempts to read file that does not exist or opens a read-only file for writing purpose, in such situations operation fails. Such errors must be reported and proper actions have to be taken before further operations.

The `!` (Logical negation operator) overloaded operator is useful for detecting the errors. It is a unary operator and in short it is called as `not` operator. The `( ! )` not operator can be used with object of stream classes. This operator returns non-zero value if stream error is found during operation. Consider the following program.

### 13.5 Write a program to check whether the file is successfully opened or not.

```
# include <fstream.h>
```

```
# include <constream.h>

void main ( )
{
    clrscr( );
    ifstream in ("text");

    if (!in)    cerr <<"\n File is not opened "<<in;
    else        cerr <<"\n file is opened "<<in;
}
```

## OUTPUT

### **File is opened 0x8f15ffd2**

**Explanation:** In the above program, an existing file “text” is opened for reading. The file is opened successfully. Hence, no error is generated. The `if ( )` statement checks the contents in objects using the following statement `if (! in)`. The value of object is `0x8f15ffd2`. In case the operation fails the value of object `in` would be `0x8f160000`.

### **13.6 Write a program to detect error in file operation using ! operator.**

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

main ( )
{
    clrscr( );
    char c, f_name[10];
    cout <<"\n Enter file name : ";
    cin>>f_name;

    ifstream in(f_name);

    if (!in)
    {
        cerr<<" Error in opening file "<<f_name<<endl;
        return 1;
    }

    in>>resetiosflags(ios::skipws);

    while (in)
```

```
{  
    in>>c;  
    cout<<c;  
}  
    return 0;  
}
```

## OUTPUT

```
Enter file name : TEXT  
One Two Three Four  
1 2 3 4  
** The End **
```

**Explanation:** In the above program, using constructor of a class `ifstream` a file is opened. The file that is to be opened is entered by the user during program execution. If the file does not exist, the `ifstream` object (`in`) contains 0. The `if( )` statement checks the value of object `in` and if operation fails a message will be displayed and program is terminated. In case the file exists then using `while( )` loop, contents of file is read one character at a time and displayed on the screen. You can observe use of `not` operator with object `in` in `if( )` and `while( )` statement. If the statement `in>>resetiosflags (ios::skipws);` is removed the contents of file would be displayed without space in one line. The operator `>>` ignores white space character. Consider the following statement:

### With `not` operator

```
if (!in)  
{  
    statement1;  
    else  
    statement2;  
}
```

The `not` operator is used in association with the object. It is also possible to perform an operation without the use of `not` operator. The statement would be as follows and it works opposite as compared to the above statement.

### Without `not` operator

```
if (in)  
{  
    statement1;  
    else  
    statement2;  
}
```

## 13.5 FINDING END OF A FILE

While reading a data from a file, it is necessary to find where the file ends i.e., end of file. The programmer cannot predict the end of file. In a program while reading the file, if the program does not detect end of file, the program drops in an infinite loop. To avoid this, it is necessary to provide correct instruction to the program that detects the end of file. Thus, when end of file is detected, the process of reading data can be easily terminated. The `eof()` member function is used for this purpose. The `eof()` stands for end of file. It is an instruction given to the program by the operating system that end of file is reached. It checks the `ios::eofbit` in the `ios::state`. The `eof()` function returns non-zero value when end of file is detected, otherwise zero.

### **13.7 Write a program to read and display contents of file. Use `eof()` function.**

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

main ( )
{
    clrscr( );
    char c, f_name[10];
    cout <<"\n Enter file name : ";
    cin>>f_name;

    ifstream in(f_name);

    if (!in)
    {
        cerr<<" Error in opening file "<<f_name<<endl;
        return 1;
    }

    while (in.eof( )==0)
    {
        in.get(c);
        cout<<c;
    }

    return 0;
}
```

### **OUTPUT**

**Enter file name : text**

# Programming with ANSI and TurboC

**Explanation:** > The above program is same as the previous one. Here, the member function `eof( )` is used in the `if( )` statement. The program displays the contents of file “text.” While specifying a file name for reading purpose, be sure it must exist.

**13.8 Write a program to detect end of file using function `eof( )`; Display the values returned by the `eof( )` function.**

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

main ( )
{
    clrscr( );
    char c;
    ifstream in("text");

    while (in.eof( )==0)
    {
        in.get(c);
        cout <<in.eof( );
    }

    return 0;
}
```

## OUTPUT

**Explanation:** As explained earlier, the `eof( )` returns one when end of file is found, otherwise zero. Thus, until it returns `zero`, the program continues to read data from file and when end of file is detected, the reading routine is terminated and `one` is displayed. In the above program, the `while( )` loop checks return value of `eof( )` function. The codes executed when `eof( )` function is called is given next.

```
ios::operator void *()
{
    return fail() ? 0 : this;
}
```

The above function converts an ifstream object into a void pointer. It also invokes `ios::fail()` function, which is as follows:

```
int fail ()  
{  
    return & (failbit | badbit | hardfail);  
}
```

The above function returns non-zero file when end of file is not encountered. The function returns address of object using `this` pointer. Zero is returned when end of file is reached. The `while()` or `if()` statement checks whether the value is zero or non-zero (address of object). Here, zero means false and non-zero means true. The details of the above functions are discussed later in the same chapter.

## 13.6 FILE OPENING MODES

In previous examples we have learnt how to open files using constructor and `open()` function by using the objects of `ifstream` and `ofstream` classes. The opening of file also involves several modes depending upon operation to be carried out with the file. The `open()` function has two arguments as given below:

### Syntax of `open()` function

```
object.open ("file_name", mode);
```

Here `object` is a stream object, followed by `open()` function. The bracket of `open` function contains two parameters. The first parameter is name of the file and second is mode in which file is to be opened. In the absence of mode parameter default parameter is considered. The file mode parameters are given in [Table 13.1](#).

**Table 13.1** File modes

| Mode parameter            | Operation                                               |
|---------------------------|---------------------------------------------------------|
| <code>ios::app</code>     | Adds data at the end of file                            |
| <code>ios::ate</code>     | After opening character pointer goes to the end of file |
| <code>ios:: binary</code> | Binary file                                             |

|                |                                                 |
|----------------|-------------------------------------------------|
| ios::in        | Opens file for reading operation                |
| ios::nocreate  | Open unsuccessful if the file does not exist    |
| ios::noreplace | Open files if its already present               |
| ios::out       | Open files for writing operation                |
| ios::trunc     | Erases the file contents if the file is present |

(1) The mode `ios::out` and `ios::trunc` are the same. When `ios::out` is used, the contents of specified file (if present) will be deleted (truncated). The file is treated as a new file.

(2) When file is opened using `ios::app` and `ios::ate` modes, the character pointer is set to end of the file. The `ios:: app` lets the user to add data at the end of file whereas `ios::ate` allows user to add or update data anywhere in the file. If the given file does not exist, new file is created. The mode `ios::app` is applicable to the output file only.

(3) The `ifstream` creates input stream and `ofstream` creates output stream. Hence, it is not compulsory to give mode parameters.

(4) While creating an object of `fstream` class, the programmer should provide the mode parameter. The `fstream` class does not have default mode.

(5) The file can be opened with one or more mode parameters. When more than one parameters are necessary, bitwise or operator separates them. The following statement opens a file for appending. It does not create a new file if the specified file is not present.

## FILE OPENING WITH MULTIPLE ATTRIBUTES

```
out.open ("file1", ios::app | ios:: nocreate)
```

### 13.9 Write a program to open a file for writing and store float numbers in it.

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

void main ( )
{
    float a=784.52, b=99.45, c =12.125;
```

```

    ofstream out ("float.txt",ios::trunc);
    out<<setw(10)<<a<<endl;
    out<<setw(10)<<b<<endl;
    out<<setw(10)<<c<<endl;
}

```

**Explanation:** In the above program, the file “float.txt” is opened. If the file already exists, its contents are truncated. The three float numbers are written in the file.

### 13.10 Write a program to open a file in binary mode. Write and read the data.

```

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

int main ( )
{
    clrscr( );
    ofstream out;
    char data[32];

    out.open ("text",ios::out | ios::binary);
    cout <<"\n Enter text   "<<endl;
    cin.getline(data,32);
    out <<data;
    out.close( );
    ifstream in;
    in.open("text", ios::in | ios::binary);
    cout <<endl<<"Contents of the file \n";
    char ch;
    while (in.eof( )==0)
    {
        ch= in.get( );
        cout<<ch;
    }
    return 0;
}

```

### OUTPUT

**Programming In ANSI and TURBO-C**

**Contents of the file**

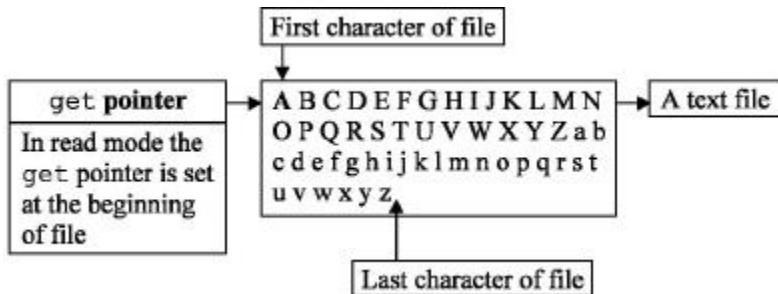
**Programming In ANSI and TURBO-C**

**Explanation:** The above program is the same as the last one. The only difference is that files are opened here in binary mode.

## 13.7 FILE POINTERS AND MANIPULATORS

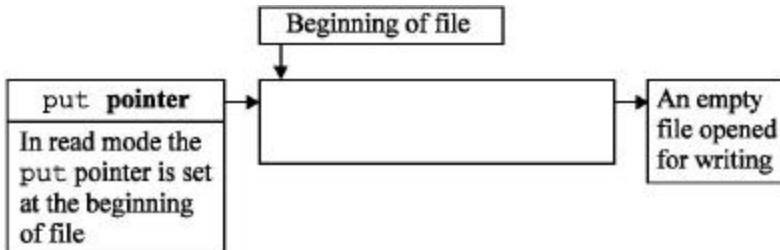
All file objects hold two file pointers associated with the file. These two file pointers provide two integer values. These integer values indicate exact position of file pointers in number of bytes in the file. The read or write operations are carried out at the location pointed by these file pointers .One of them is called as get pointer (input pointer) and the second one is called as put pointer (output pointer). During reading and writing operations with files, these file pointers are shifted from one location to another location in the file. The (input) get pointer helps in reading the file from the given location and the output pointer helps for writing data in the file at specified location. When read and write operations are carried out, respective pointer is moved. While file is opened for reading or writing operation, the respective file pointer, input or output is by default set at the beginning of the file. This makes possible for performing read or write operation from the beginning of file. The programmer need not explicitly set the file pointers at the beginning of files. To explicitly set the file pointer at the specified position, the file stream classes provide following functions:

**Read Mode** When a file is opened in read mode, the get pointer is set at the beginning of the file as shown in [Figure 13.7](#). Hence, it is possible to read the file from the first character of the file.



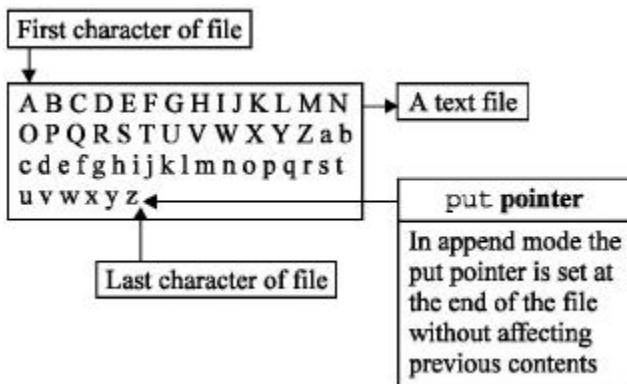
**Fig. 13.7** Status of get pointer in read mode

**Write Mode** When a file is opened in write mode, the put pointer is set at the beginning of the file as shown in [Figure 13.8](#). Thus, it allows write operation from beginning of the file. In case the specified file already exists, its contents will be deleted.



**Fig. 13.8** Status of put pointer in write mode

**Append Mode** This mode allows to add data at the end of file. When file is opened in append mode, the output pointer is set at the end of file as shown in [Figure 13.7](#). Hence, it is possible to write data at the end of file. In case the specified file already exists, new file is created, and the output is set at the beginning of the file. When a pre-existing file is successfully opened in append mode, its contents remain safe and new data is appended at the end of file.



**Fig. 13.9** Status of put pointer in append mode

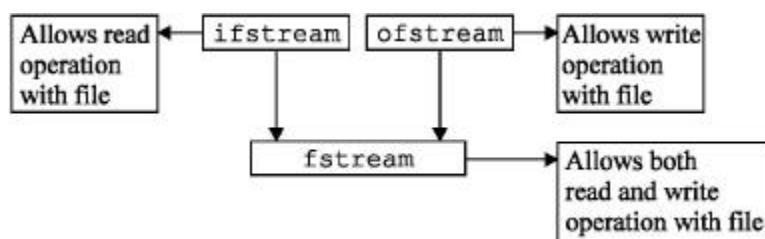
C++ has four functions for setting of pointers during file operation. The position of cursor in file can be changed using these functions. These functions are described in [Table 13.2](#).

**Table 13.2** File pointer handling functions

| Function | Uses                                            | Remark                          |
|----------|-------------------------------------------------|---------------------------------|
| seekg( ) | Shifts input (get) pointer to a given location. | Member of <code>ifstream</code> |

|          |                                                      |                                 |
|----------|------------------------------------------------------|---------------------------------|
| seekp( ) | Shifts output (put) pointer to a given location.     | Member of <code>ofstream</code> |
| tellg( ) | Provides the present position of the input pointer.  | Member of <code>ifstream</code> |
| tellp( ) | Provides the present position of the output pointer. | Member of <code>ofstream</code> |

As given in [Table 13.2](#) the `seekg( )` and `tellg( )` are member functions of `ifstream` class. The functions `seekp( )` and `tellp( )` are member functions of `ofstream` class. All the above four functions are present in the class `fstream`. The class `fstream` is derived from `ifstream` and `ofstream` classes. Hence, this class supports both input and output modes as shown in [Figure 13.10](#). The `seekp( )` and `tellp( )` works with put pointer and `tellg( )` and `seekg( )` works with get pointer.



**Fig. 13.10** Derivation of `fstream` class

Now consider the following examples.

### 13.11 Write a program to append a file.

```

# include <fstream.h>
# include <conio.h>

int main ( )
{
    clrscr( );
    ofstream out;
    char data[25];

    out.open ("text",ios::out);
    cout <<"\n Enter text  "<<endl;

    cin.getline(data,25);
  
```

```

out <<data;
out.close( );

out.open ("text", ios::app );
cout <<"\n Again Enter text "<<endl;
cin.getline (data,25);
out<<data;
out.close( );

ifstream in;
in.open("text", ios::in);
cout <<endl<<"Contents of the file \n";

while (in.eof( )==0)
{
    in>>data;
    cout<<data;
}
return 0;
}

```

## OUTPUT

Enter text

C-PLUS-

Again Enter text

PLUS

Contents of the file

C-PLUS-PLUS

**Explanation:** In the above program, the file `text` is opened for writing i.e., output. The text read through the keyboard is written in the file. The `close()` function closes the file. Once more, the same file is opened in append mode and data entered through the keyboard is appended at the end of file i.e., after previous text. The append mode allows the programmer to write data at the end of file. The `close()` function closes the file. The same file is opened using the object of `ifstream` class for reading purpose. The `while` loop is executed until the end of file is detected. The statements within the `while` loop reads text from file and displays it on the screen.

### 13.12 Write a program to read contents of the file. Display the position of the get pointer.

```
# include <fstream.h>
```

```

# include <conio.h>

int main ( )
{
    clrscr( );
    ofstream out;
    char data[32];

    out.open ("text",ios::out);
    cout <<"\n Enter text " << endl;
    cin.getline(data,32);
    out << data;
    out.close( );

    ifstream in;
    in.open("text", ios::in);
    cout << endl << "Contents of the file \n";
    int r;

    while (in.eof( )==0)
    {
        in>>data;
        cout<<data;
        r=in.tellg( );
        cout << " (" <<r <<") ";
    }
    return 0;
}

```

## **OUTPUT**

**Enter text**

**Programming In ANSI and TURBO-C**

**Contents of the file**

**Programming (11)In (14)ANSI (19)and (23)TURBO-C (31)**

***Explanation:*** The above program is same as the previous one. In addition here, the function tellg ( ) is used. This function returns current file pointer position in number of bytes from beginning of the file. The number shown in brackets in output specifies position of file pointer from the beginning of file. The same program is illustrated below using binary mode.

## **13.8 MANIPULATORS WITH ARGUMENTS**

The seekp ( ) and seekg ( ) functions can be used with two arguments.

Their formats with two arguments follows below:

```

seekg ( offset, pre_position);
seekp ( offset, pre_position);

```

The first argument `offset` specifies the number of bytes the file pointer is to be shifted from the argument `pre_position` of the pointer.

The `offset` must be a positive or negative number. The positive number moves the pointer in forward direction whereas negative number moves the pointer in backward direction. The `pre_position` argument may have one of the following values.

- `ios::beg` **Beginning of the file**
- `ios::cur` **Current position of the file pointer**
- `ios::end` **End of the file**

Figure 13.11 shows status of `pre_position` arguments.



**Fig. 13.11** Status of `pre_position` arguments

In the above figure, status of `ios::beg` and `ios::end` is shown. The status `ios::cur` cannot be shown like `ios::beg` or `ios::end`.

The `ios::cur` means present position of file pointer.

The `ios::beg` and `ios::end` may be referred as `ios::cur`. Suppose the file pointer is in the middle of file and you want to read the file from the beginning, then you can set the file pointer at the beginning using `ios::beg`. However, if you want to read the file from current position, you can use the option `ios::cur`.

The `seekg( )` function shifts the associated file's input (`get`) file pointer.

The `seekp( )` function shifts the associated file's output (`put`) file pointer. Table 13.3 describes few pointer offsets and their working.

**Table 13.3** File pointer with its arguments

| Seek option                          | Working                      |
|--------------------------------------|------------------------------|
| <code>in.seekg (0,ios :: beg)</code> | Go to the beginning of file  |
| <code>in.seekg (0,ios :: cur)</code> | Rest at the current position |
| <code>in.seekg (0,ios ::end)</code>  | Go to the end of file        |

|                          |                                              |
|--------------------------|----------------------------------------------|
| in.seekg (n,ios :: beg)  | Shifts file pointer to n+1 byte in the file  |
| in.seekg (n,ios :: cur)  | Go front by n byte from current position     |
| in.seekg (-n,ios :: cur) | Go back by n bytes from the present position |
| in.seekg (-n,ios::end)   | Go back by n bytes from the end of file      |

In Table 13.3, in is an object of class `ifstream` class.

### 13.13 Write a program to write text in the file. Read the text from the file from end of file. Display the contents of file in reverse order.

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

int main ( )
{
    clrscr( );
    ofstream out;
    char data[25];
    out.open ("text",ios::out);
    cout <<"\n Enter text   "<<endl;
    cin.getline(data,25);
    out <<data;
    out.close( );

    ifstream in;
    in.open("text", ios::in);
    cout <<endl<<"Reverse Contents of the file \n";
    in.seekg(0,ios::end);
    int m=in.tellg( );
    char ch;

for (int i=1;i<=m;i++)
{
    in.seekg(-i,ios::end);
    in>>ch;
    cout<<ch;
}
return 0;
}
```

### OUTPUT

Enter text  
Visual\_C\_+\_+

## Reverse Contents of the file

+ + C\_lausiV

**Explanation:** In the program, `text` file is opened in output mode and string entered is written to the file. Again the same file is opened for reading purpose. The statement `in.seekg (0, ios::end);` moves the get pointer at the end of file. The `tellg( )` function returns the current position of the file pointer in the file. Hence, file pointer is set to the end of file. The `tellg( )` returns the number of last byte i.e., size of the file in bytes and it is stored in the integer variable `m`. The `for` loop executes from 1 to `m`. The statement `in.seekg (-i, ios::end)` reads `i`th byte from the end of file. The statement `in>>ch` reads the character from file indicated by the file pointer. The `cout` statement displays the read character on the screen. Thus, the contents of the file displayed is in reverse order.

**13.14 Write a program to enter a text and again enter a text and replace the first word of the first text with the second text. Display the contents of the file.**

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

int main ( )
{
    clrscr( );
    ofstream out;
    char data[25];

    out.open ("text",ios::out);
    cout <<"\n Enter text  "<<endl;
    cin.getline(data,25);
    out <<data;

    out.seekp(0,ios::beg);
    cout<<"\nEnter text to replace the first word of first text:";
    cin.getline(data,25);
    out<<data;
    out.close( );

    ifstream in;
    in.open("text", ios::in);
    cout <<endl<<"Contents of the file \n";
```

```
while (in.eof() !=1)
{ in>>data;
  cout<<data; }

return 0;
}
```

## OUTPUT

Enter text

Visual C++

Enter text to replace the first word of first text : Turbo-

### Contents of the file

Turbo-C++

**Explanation:** In the above program, `text` is entered and written in the file `text`. This process is explained in previous examples. Here again, the statement `out.seekp(0, ios::beg)` sets the file pointer (put pointer) at the beginning of file. Again, text is entered and written at the current file pointer position. The previous text is overwritten.

## 13.9 SEQUENTIAL READ AND WRITE OPERATIONS

C++ allows file manipulation command to access file sequentially or randomly. The data of sequential file must be accessed sequentially i.e., one character at a time. In order to access nth number of bytes, all previous characters are read and ignored. There are number of functions to perform read and write operations with the files. Some function read /write single character and some function read/write block of binary data. The `put( )` and `get( )` functions are used to read or write a single character whereas `write( )` and `read( )` are used to read or write block of binary data.

### THE PUT( ) AND GET( ) FUNCTIONS

The function `get( )` is a member function of the class `fstream`. This function reads a single character from the file pointed by the get pointer i.e., the character at current get pointer position is caught by the `get( )` function.

The `put( )` function writes a character to the specified file by the stream object. It is also a member of `fstream` class. The `put( )` function places a character in the file indicated by put pointer.

### **13.15 Write a program to write and read string to the file using put( ) and get( ) functions.**

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

int main( )
{
    clrscr( );

    char text[50];
    cout <<"\n Enter a Text : ";
    cin.getline(text,50);
    int l=0;
    fstream io;
    io.open("data", ios::in | ios::out);

    while (l[text]!='\0')
        io.put(text[l++]);

    io.seekg(0);
    char c;
    cout <<"\n Entered Text : ";
    while (io)
    {
        io.get(c);
        cout<<c;
    }
    return 0;
}
```

#### **OUTPUT**

**Enter a Text : PROGRAMMING WITH ANSI AND TURBO-C**

**Entered Text : PROGRAMMING WITH ANSI AND TURBO-C**

***Explanation:*** In the above program, the file data is opened in read and write mode at once. The `getline( )` function reads the string through the keyboard and stores in the array `text[50]`. The statement `io.put(text[l++])` in the first `while` loop reads one character from array and writes it to the file indicated by the stream object `io`. The first `while` loop terminates when the null character is found in the text.

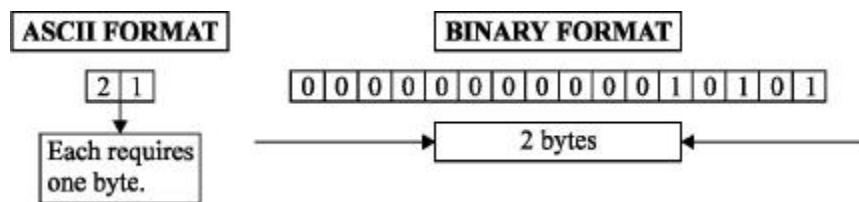
The statement `io.seekg(0)` sets the file pointer at the beginning of the file. In the second while loop, the statement `io.get(c)` reads one character at a time from the file and `cout( )` statement displays the same character on the screen. The `while` loop terminates when end of file is detected.

### 13.10 BINARY AND ASCII FILES

The insertion and extraction operators, known as stream operator, handles formatted data. The programmer needs to format data in order to represent it in a suitable fashion. The description of formatted and unformatted data is given in Chapter “**INPUT and OUTPUT IN C++**”. ASCII codes are used by the I/O devices to share or pass data to the computer system, but central processing unit (CPU) manipulates the data using binary numbers i.e., 0 and 1. For this reason, it is essential to convert the data while accepting data from input devices and displaying the data on output devices. Consider the following statements:

```
cout<<k; // Displays value of k on screen  
cin>>k; // Reads value for k from keyboard
```

Here, `k` is an integer variable. The operator `<<` converts value of integer variable `k` into stream of ASCII characters. In the same fashion, the `<<` operator converts the ASCII characters entered by the user to binary. The data is entered through the keyboard, a standard input device. For example, you enter 21. The stream operator `>>` gets ASCII codes of the individual digits of the entered number 21 i.e., 50 and 49. ASCII code of 2 and 1 are 50 and 49 respectively. The stream operator `>>` converts the ASCII value to equivalent binary format and assigns it to the variable `k`. The stream operator `<<` converts the value of `k` (21) that is stored in binary format into equivalent ASCII codes i.e., 50 and 49. Figure 13.9 shows representation of integer number in ASCII and binary format.



**Fig. 13.12** Representation in BINARY and ASCII formats

**13.16 Write a program to demonstrate that the data is read from the file using ASCII format.**

```

# include <fstream.h>
# include <constream.h>
int main( )
{
    clrscr( );
    char c;
    ifstream in("dat");

    if (!in)
    {
        cerr<<" Error in opening file. ";
        return 1;
    }

    while (in.eof( )==0)

    {
        cout<<(char)in.get( );
    }

    return 0;
}

```

## **OUTPUT**

### **PROGRAMMING WITH ANSI AND TURBO-C**

**Explanation:** In the above program, the data file is opened in read mode. The file already exists. Using get( ) member function of ifstream class, the contents of file is read and displayed. Consider the following statement:

```
cout<<(char)in.get( );
```

The get( ) function reads data from file in ASCII format. Hence, it is necessary to convert the ASCII number to equivalent character. The typecasting format (char) converts ASCII number to equivalent character. In case the conversion is not done, the output would be as follows:

**8082797182657777378713287738472326578837332657868328485  
82667967-1**

The output displayed above are ASCII numbers and -1 at the end indicates end of file.

After typecasting the original string will be as shown in the output.

### **(1) THE WRITE( ) AND READ( ) FUNCTIONS**

The data entered by the user are represented in ASCII format. However, the computer can understand only machine format i.e., 0 and 1. When data is

stored in text format numbers are stored as characters and occupy more memory space. The functions `put( )` and `get( )` read/ write a character. The data is stored in the file in character format. If large number of numeric data is stored in the file, it will occupy more space. Hence, using `put( )` and `get( )` creates disadvantages.

This limitation can be overcome using `write( )` and `read( )` functions. The `write( )` and `read( )` functions use binary format of data during operation. In binary format, the data representation is same in file and system. Figure 13.12 shows difference between ASCII and binary format. The bytes required to store an integer in text form depends upon its size whereas in binary format the size is fixed. The binary form is exact and allows quick read and write operation because no conversion takes place during operations. The formats of the `write( )` and `read( )` functions are given below.

```
in.read((char *) & P, sizeof(P));  
out.write((char *) & P, sizeof(P));
```

These functions have two parameters. The first parameter is the address of the variable `P`. The second is the size of the variable `P` in bytes. The address of the variable is converted to char type. Let us consider program 13.17.

### 13.17 Write a program to perform read and write operations using `write( )` and `read( )` functions.

```
# include <iostream.h>  
# include <conio.h>  
# include <string.h>  
  
int main( )  
{  
    clrscr( );  
    int num[]={100,105,110,120,155,250,255};  
    ofstream out;  
    out.open("01.bin");  
    out.write((char *) &num, sizeof(num));  
    out.close( );  
  
    for (int i=0;i<7;i++) num[i]=0;  
  
    ifstream in;  
    in.open("01.bin");  
    in.read((char *) & num, sizeof(num));
```

```
    for (i=0;i<7;i++)    cout<<num[i]<<"\t";
    return 0;
}
```

## OUTPUT

```
100 105 110 120 155 250 255
```

**Explanation:** In the above program, integer array is initialized with 7 integer numbers. The file 01.bin is opened. The statement `out.write((char *) & num, sizeof(num))` writes the integer array in the file.

The `&num` argument provides the base address of the array and the second argument provides total size of the array. The `close()` function closes the file. The same file is opened again for reading purpose. Before reading the contents of the file, the array is initialized to zero that is not necessary. The statement `in.read((char *) & num, sizeof(num))` reads data from the file and assigns it to the integer array. The second `for` loop displays the contents of the integer array. The size of file "01.bin" will be 14 bytes i.e., two bytes per integer. If the above data is stored without using `write()` command, the size of the file will be 21 bytes.

## (2) READING AND WRITING CLASS OBJECTS

The functions `read()` and `write()` perform write and read operations in binary format that is exactly same as internal representation of data in the computer. Due to these capabilities of the functions, large data can be stored in small amount of memory. Both these functions are also used to write and read class objects to and from file. During read and write operation only data members are written to the file and the member functions are ignored.

Consider the following program.

### 13.18 Write a program to perform read and write operations with objects using `write()` and `read()` functions.

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

```
class boys
{
    char name [20];
    int age;
    float height;
public :
    void get()
    {
```

```

        cout << "Name      : " ; cin>>name;
        cout << "Age       : " ; cin>>age;
        cout << "Height   : " ; cin>>height;
    }

void show ( )
{
    cout <<"\n"<<name<<"\t"<<age <<"\t"<<height;
}
};

int main( )
{
    clrscr( );
    boys b[3];
    fstream out;
    out.open ("boys.doc", ios::in | ios::out);
    cout <<"\n Enter following information :\n";

    for (int i=0;i<3;i++)
    {
        b[i].get( );
        out.write ((char*) & b[i],sizeof(b[i]));

    }
    out.seekg(0);
    cout <<"\n Entered information\n";
    cout <<"Name      Age      Height";

    for (i=0;i<3;i++)
    {
        out.read((char *) & b[i], sizeof(b[i]));
        b[i].show( );
    }
    out.close( );
    return 0;
}

```

## **OUTPUT**

**Enter following information:**

**Name : Kamal**

**Age : 24**

**Height : 5.4**

**Name : Manoj**

```
Age : 24  
Height : 5.5  
Name : Rohit  
Age : 21  
Height : 4.5
```

#### Entered information

| Name  | Age | Height |
|-------|-----|--------|
| Kamal | 24  | 5.4    |
| Manoj | 24  | 5.5    |
| Rohit | 21  | 4.5    |

**Explanation:** In the above program, the class `boys` contains data members' name, age, and height of `char`, `int`, and `float` type. The class also contains member functions `get()` and `show()` to read and display the data. In function `main()` an array of three objects is declared i.e., `b[3]`. The file "boys.doc" is opened in output and input mode to write and read data. The first `for` loop is used to call the member function `get()` and data read via `get()` function is written to the file by `write()` function. The same method is repeated while reading the data from file. While reading data from file, `read()` function is used and the member function `show()` displays the data on the screen.

### 13.11 RANDOM ACCESS OPERATION

Data files always contain large information and the information always changes. The changed information should be updated otherwise the data files are not useful. Thus to update data in the file we need to update the data files with latest information. To update a particular record of data file it may be stored anywhere in the file but it is necessary to obtain at which location (in terms of byte number) the data object is stored.

The `sizeof()` operator determines the size of object. Consider the following statements.

(a) `int size=sizeof(o);`

Where, `o` is an object and `size` is an integer variable. The `sizeof()` operator returns the size of object `o` in bytes and it is stored in the variable `size`. Here, one object is equal to one record.

The position of nth record or object can be obtained using the following statement.

(b) `int p=(n-1 * size);`

Here, p is the exact byte number of the object that is to be updated, n is the number of object, and size is the size in bytes of an individual object (record).

Suppose we want to update 5<sup>th</sup> record. The size of individual object is 26.

(c) p=(5-1\*26) i.e. p= 104

Thus, the fifth object is stored in a series of bytes from 105 to 130. Using functions seekg( ) and seekp( ) we can set the file pointer at that position.

**13.19 Write a program to create a text file. Add and modify records in the text file. The record should contain name, age, and height of a boy.**

```
# include <stdio.h>
# include <process.h>
# include <fstream.h>
# include <conio.h>

class boys
{
    char name [20];
    int age;
    float height;
public :

void input( )
{
    cout << "Name      : ";    cin>>name;
    cout << "Age      : ";      cin>>age;
    cout << "Height   : ";      cin>>height;
}

void show (int r)
{
    cout <<"\n"<<r<<"\t"<<name<<"\t"<<age <<"\t"<<height; }

};

boys b[3];
fstream out;

void main ( )
{
    clrscr( );
```

```

void menu (void);
out.open ("boys.doc", ios::in | ios::out | ios::noreplace);
menu( );
}

void menu(void)
{
    void get(void);
    void put(void);
    void update(void);
    int x;
    clrscr( );
    cout <<"\n Use UP arrow key for selection";
    char ch=' ';
    gotoxy(1,3);
    printf ("ADD ( )");
    gotoxy(1,4);
    printf ("ALTER( )");
    gotoxy(1,5);
    printf ("EXIT ( )");
    x=3;
    gotoxy(7,x);
    printf ("*");

    while (ch!=13)
    {
        ch=getch( );

        if (ch==72)
        {
            if (x>4)
            {
                gotoxy(7,x);
                printf (" ");
                x=2;
            }

            gotoxy(7,x);
            printf (" ");
            gotoxy(7,++x);
            printf ("*");
        }
    }
}

```

```

switch(x)
{
    case 3 :   get( );  put( ); getch( ); break;
    case 4 :   put( ); update( ); put( ); getch( ); break;
    default :  exit(1);
}
menu( );
}

void get ( )
{
    cout <<"\n\n\n Enter following information :\n";
    for (int i=0;i<3;i++)
    {
        b[i].input( );
        out.write ((char*) & b[i],sizeof(b[i]));
    }
}
void put ( )
{
    out.seekg(0,ios::beg);
    cout <<"\n\n\n Entered information \n";
    cout <<"Sr.no      Name      Age       Height";
    for (int i=0;i<3;i++)

    {
        out.read((char *) & b[i],
        sizeof(b[i]));
        b[i].show(i+1);
    }
}

void update( )
{
    int r, s=sizeof(b[0]);
    out.seekg(0,ios::beg);
    cout <<"\n" <<"Enter record no. to update : ";
    cin>>r;
    r=(r-1)*s;
    out.seekg(r,ios::beg);
    b[0].input( );
    out.write ((char*) & b[0],sizeof(b[0]));
    put( );
}

```

## **OUTPUT**

**Use UP arrow key for selection**

**ADD (\*)**  
**ALTER( )**  
**EXIT ( )**

**Enter following information :**

**Name : Sachin**

**Age : 28**

**Height : 5.4**

**Name : Rahul**

**Age : 28**

**Height : 5.5**

**Name : Saurav**

**Age : 29**

**Height : 5.4**

**Entered information**

| Sr.no | Name   | Age | Height |
|-------|--------|-----|--------|
| 1     | Sachin | 28  | 5.4    |
| 2     | Rahul  | 28  | 5.5    |
| 3     | Saurav | 29  | 5.4    |

**Explanation:** In the above program, the class `boys` contain data member name, age, and height. The class `boys` also contain member functions `input()` and `show()`. The `input()` function is used to read data and `show()` function displays data on the screen.

After class definition and before `main()` function, array of objects `b[3]` and `fstream` object `out` are declared. They are declared before `main()` for global access. The file `boys.doc` is opened in input and output modes to perform both read and write operation.

The `menu()` function displays menu on the screen. The menu items can be selected using up arrow key. To start the operation hit enter. There are three more user-defined functions. They are `get()`, `put()` and `update()`. The `get()` function calls the member function `input()` to read data through the keyboard. The `get()` function writes the data using `write()` function. The `put()` function calls the member function `show()`.

The `put()` function calls the member function `show()`. The `put()` function reads the data from the file using `read()` function.

The `update()` function is used to modify the previous record. The `seekg()` function sets the file pointer at the beginning of the file. The `sizeof()` operator determines the size of object and stores it in the variable `s`.

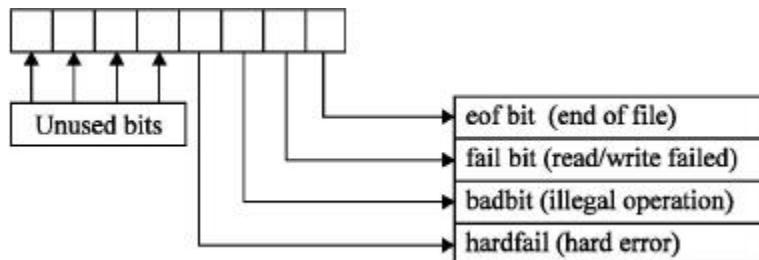
### 13.12 ERROR HANDLING FUNCTIONS

Until now, we have performed the file operation without any knowledge of failure or success of the function `open( )` that opens the file. There are many reasons and they may result in error during read/write operation of the program.

- (1) An attempt to read a file which does not exist.
- (2) The file name specified for opening a new file may already exist.
- (3) An attempt to read contents of file when file pointer is at the end of file.
- (4) Insufficient disk space.
- (5) Invalid file name specified by the programmer.
- (6) An effort to write data to the file that is opened in read only mode.
- (7) A file opened may already be opened by another program.
- (8) An attempt to open read only file for writing operation.
- (9) Device error.

The ‘stream state’ member from the class `ios` receives values from the status bit of active file. The class `ios` also contains many different member functions. These functions read the status bit of the file where error occurred during program execution are stored. These functions are given in [Table 13.3](#) and various status bits are described in [Table 13.4](#).

All streams such as `ofstream`, `ifstream`, and `fstream` contain state connected with it. Faults and illegal conditions are managed (controlled) by setting and checking the state properly. [Figure 13.13](#) describes it more clearly.



**Fig. 13.13** Status bits

**Table 13.4** Status bits

|         |                                               |
|---------|-----------------------------------------------|
| eofbit  | End of file encountered                       |
| failbit | Operation unsuccessful                        |
| badbit  | Illegal operation due to wrong size of buffer |

|          |                |
|----------|----------------|
| hardfail | Critical error |
|----------|----------------|

**Table 13.5** Error trapping functions

| Functions  | Working and return value                                                                                                                                                                                |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| fail( )    | Returns non-zero value if an operation is unsuccessful. This is carried out by reading bits <code>ios::failbit</code> , <code>ios::badbit</code> , and <code>ios::hardfail</code> of <code>ios</code> . |
| eof( )     | Returns non-zero value when end of file is detected otherwise returns zero. The <code>ios::eofbit</code> is checked.                                                                                    |
| bad( )     | Returns non-zero value when error is found in operation. The <code>ios::badbit</code> is checked.                                                                                                       |
| good( )    | Returns non-zero value of no error occurred during file operation i.e., no status also indicates that the above functions are false. When this function returns true, the file operation.               |
| rdstate( ) | Returns the stream state. It returns value of various bits of <code>ios::state</code> .                                                                                                                 |

The following examples illustrate techniques of error checking.

### (A) AN ATTEMPT TO OPEN A NON-EXISTING FILE FOR READING

```
ifstream in("data.txt");

if (!in)
{
    cout << "File not found";
}
```

In above format an attempt is made to open a file for reading. If the file already exists, it will be opened otherwise operation fails. Thus, by checking the value of object `in` we can confirm failure or success of the operation and accordingly further processing can be decided.

### (B) AN ATTEMPT TO OPEN A READ ONLY FILE FOR WRITING

```
ofstream out("data.txt");

if (!out)
    cout << "Unable to open file";
```

```
else
    cout<<"File opened";
```

Suppose that the `data.txt` file is protected (marked read-only) or used by another application in multitasking operating environment. If the same file is opened in write mode as shown above, the operation fails. By checking value of object `out` with `if( )` statement we can catch the error and transfer the program control to suitable sub-routine.

### (C) CHECKING END OF FILE

```
ifstream in("data.txt");

while (!in.eof( ))
{
// read data from file
// display on screen
}
```

We may seek to open an existing file and read its contents. After opening a file in read mode, it is necessary to read the characters from file using appropriate function (`read( )` or `get( )`). While reading a file, the get pointer is advanced to successive characters and the same process can be repeated using loops. The compiler cannot determine by itself the end of file. The `eof( )` function determines the end of file. Thus, by checking the value of `eof( )` function we can determine the end of file. In addition, by checking the value of object the end of file is determined. Such conditions must be placed in the `while` loop parenthesis. While reading file, use only `while` or `for` loop.

### (D) ILLEGAL FILE NAME

```
ifstream in("****");

while (!in.eof( ))
{
// read data from file
// display on screen
}
```

While performing file operation, it is the user's responsibility to specify the correct file name. If illegal file name is specified by the user, file operation fails. In the above format `"****"` is given as file name that is invalid.

### (E) OPERATION WITH UNOPENED FILE

```
ifstream in("DATA");
```

```

while (!in.eof( ))
{
// read data from file
// display on screen
}

```

Suppose “DATA” file does not exist and an attempt is made to open it for reading. Any operation applied with this file will be of no use. Hence, while performing file operation first we have to check whether the file is opened successfully or not. After the confirmation, we can proceed to the next step. Programs referred to above discussions are explained below.

### **13.20 Write a program to detect whether the file is opened successfully or not.**

```

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

ifstream in; // Global object

void main( )
{
    clrscr( );
    void show (void);
    in.open("dat") ;
    char c;

    if (in!=0)    show( );
    else          cout <<"\n File not found";
}

```

```

void show( )
{
    char c;
    cout <<"\n Contents of file : ";

```

```

    while (in)
    {
        in.get(c);
        cout<<c;
    }
}

```

### **OUTPUT**

## **File not found**

**Explanation:** In the above program, object in of `ifstream` class is declared globally. It can be accessed by any normal function. In function `main()` using the object in a file is opened. If the `open()` function fails to open file, it returns zero otherwise non-zero value. The if statement checks the value of object in and if it is non-zero `show()` function is invoked, otherwise “File not found” message is displayed. The `show()` function reads file and displays the contents on the screen. In the above program, the `open()` function tries to open “dat” file, which does not exist. Hence, the output is “File not found.” If the specified file exists, the contents of file will be displayed.

### **13.21 Write a program to display status of various error trapping functions.**

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

void main ( )
{
    clrscr( );
    ifstream in;
    in.open ("text.txt", ios::nocreate);
    if (!in)
        cout <<"\n File not found";
    else
        cout<<"\nFile ="<<in;
        cout <<"\nError state = "<<in.rdstate( );
        cout <<"\ngood ( )      = "<<in.good( );
        cout <<"\neof ( )       = "<<in.eof( );
        cout <<"\nfail( )       = "<<in.fail( );
        cout <<"\nbad ( )       = "<<in.bad( );
    in.close( );
}
```

## **OUTPUT**

**File not found**

**Error state = 4**

**good () = 0**

**eof () = 0**

**fail () = 4**

**bad () = 4**

**Explanation:** In the above program, an attempt is made to open a non-existent file. The if statement checks the value of object in. The specified file

does not exist, hence it displays the message “File not found.” The program also displays the values of various bits using the functions `good()`, `eof()`, `bad()`, `fail()` and `rdstate()` error trapping functions. For more information about these functions, please refer to [Table 13.4](#).

### 13.13 COMMAND LINE ARGUMENTS

An executable program that performs a specific task for operating system is called as command. The commands are issued from the command prompt of operating system. Some arguments are to be associated with the commands hence these arguments are called as command line arguments. These associated arguments are passed to program.

Like C, in C++ every program starts with a `main()` function and it marks the beginning of the program. We have not provided any arguments so far in the `main()` function. Here, we can make arguments in the `main()` like other functions. The `main()` function can receive two arguments and they are `argc` (**argument counter**) and `argv` (**argument vector**). The first argument contains the number of arguments and the second argument is an array of char pointers. The `*argv` points to the command line arguments. The size of array is equal to value counted by the `argc`. The information contained in the command line is passed on to the program through these arguments when the `main()` is called up by the system.

(1) Argument `argc` An argument `argc` counts total number of arguments passed from command prompt. It returns a value that is equal to total number of arguments passed through the `main()`.

(2) Argument `argv` It is a pointer to an array of character strings that contains names of arguments. Each word is an argument.

**Syntax** - `main ( int argc, char * argv [ ] );`

**Example** - `ren file1 file2.`

Here, `file1` and `file2` are arguments and `copy` is a command. The first argument is always an executable program followed by associated arguments. If argument is not the first program name itself becomes an argument but the program will not run properly and will flag an error. The contents of `argv[ ]` would be as per following:

```
argv [0] → ren  
argv [1] → file1  
argv [2] → file2
```

### 13.22 Write a program to simulate rename command using command line arguments.

```
# include <stdio.h>
# include <fstream.h>
# include <conio.h>
# include <process.h>

main(int argc, char *argv[])
{
    fstream out;
    ifstream in;

    if (argc<3 )
    {
        cout<<"Insufficient Arguments";
        exit(1);
    }

    in.open(argv[1],ios::in | ios::nocreate);

    if (in.fail( ))
    {
        cout <<"\nFile Not Found";
        exit(0);
    }
    in.close( );
    out.open(argv[2],ios::in | ios::nocreate);

    if (out.fail( ))
    {   rename(argv[1],argv[2]); }
    else
        cout <<"\nDuplicate file name or file is in use.";
    return 0;
}
```

**Explanation:** In the above program, the `main()` receives two file names. The existence of file can be checked by opening it in read mode. If the file does not exist the program terminates. On the other hand if the second file exists the rename operation cannot be done. The rename operation is carried out only when first file exists and the second file does not exist. Make exe file of this program and use it on the command prompt.

### 13.14 STRSTREAMS

## (1) OSTRSTREAM

The `strstream` class is derived from `istrstream` and `ostrstream` classes. The `strstream` class works with memory. Using objects of `ostrstream` class different type of data values can be stored in an array.

### 13.23 Write a program to demonstrate use of `ostrstream` objects.

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

main ( )
{
    clrscr( );
    char h='C';
    int j=451;
    float PI=3.14152;
    char txt[]="applications";
    char buff[70];

    ostrstream o (buff,70);
    o<<endl <<setw(9)<<"h="<<h<<endl <<setw(9) <<"j="<<oct<<j<<endl
    <<setw(10)<<"PI="<<setiosflags(ios::fixed)<<PI<<endl <<setw(11)
    <<"txt="<<txt <<ends;
    cout<<o.rdbuf( );
return 0;
}
```

### OUTPUT

**h=C  
j=703  
PI=3.14152  
txt= applications**

**Explanation:** The `strstream` deals with memory. If we want to pick characters from a `strstream` or we want to add characters into `strstream`, this can be done by creating `istrstream` and `ostrstream` objects. When object `o` is created the constructor of `ostrstream` is executed. Once an object of `ostrstream` is created we can assign any formatted text to array associated with it. The statement `cout<<o.rdbuf( )` displays the formatted information on the screen.

## (2) ISTRSTREAM

It is one of the base classes of the `strstream` class. Using objects of `istrstream` class, data can be extracted from an array. Suppose a character array contains numbers and characters. You can extract number from an array and assign it to an integer variable. Similarly, other values can be extracted from an array. The following program illustrates this.

### 13.24 Write a program to demonstrate the use of `istrstreams`.

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

main ( )
{
    clrscr( );
    char *book;
    int pages;
    float price;
    char *text="550 175.75 C++";

    istrstream o(text);
    o>>pages>>price>>book;
    cout <<endl <<pages <<endl <<price<<endl<<book;
    cout <<o.rdbuf( );
    return 0;
}
```

#### OUTPUT

550  
175.75  
C++

**Explanation:** The `istrstream` is opposite of the `strstream`. It picks different types of data from an array. In the above program, character pointer `text` contains data of `integer`, `float`, and `character` type. Using the object of `istrstream` class we can separate the contents and store them in appropriate variable. The `*book` pointer variable displays string. The remaining contents are displayed by the function `rdbuf()`.

## 13.15 SENDING OUTPUT TO DEVICES

It is also possible to send information of file directly to devices like printer or monitor. Table 13.6 describes various devices with their names and descriptions. The program given next illustrates the use of such devices.

**Table 13.6** Standard devices

| Device Name | Description              |
|-------------|--------------------------|
| CON         | Console (monitor screen) |
| COM1 or AUX | Serial port – I          |
| COM2        | Serial port – II         |
| LPT1 OR PRN | Parallel printer – I     |
| LPT2        | Parallel printer – II    |
| LPT3        | Parallel printer – III   |
| NUL         | Dummy device             |

### **13.25 Write a program to read a file and send data to the printer.**

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

#define eject out.put('\x0C');

void main ( )
{clrscr( );
 char h;
 char name[20];
 cout <<"Enter file name : ";
 cin>> name;

ifstream in (name);

if (!in)
{
    cerr <<endl<<"File opening error";
    _cexit( );
}

ofstream out ("LPT1");
```

```

if(!out)
{
    cerr << endl << "device opening error";
    _cexit( );
}

while (in.get(h) !=0)
out.put(h);
eject;

}

```

**Explanation:** In the above program, the file name is entered through the keyboard and it is opened for reading purpose. The `ifstream` object `in` opens the file. The `ofstream` object `out` activates the printer.

The `if` statement checks both the objects for detect operation status whether the operations have failed or are successful. The `while` loop reads data from the file and using `put ( )` statement passes it to the devices associated with the object `out`. In this program data read is passed to the printer. The macro `eject` defined at the beginning of the program advances page of printer. In case printer is not attached, the message displayed will be as under:

```

Error Message
System Message
Error accessing LPT1 device
>> Retry << Cancel

```

The user can select `retry` if the printer is attached otherwise by selecting `cancel`, the operation can be cancelled.

### SUMMARY

- (1) These days, huge amount of data is processed in computer networking. The information is uploaded or downloaded from the desktop computer. The information transfer in computer networking in day-to-day life is done in the form of files. The data is saved in the file on the disk. The file is an accumulation of data stored on the disk.
- (2) Stream is nothing but flow of data. In object-oriented programming the streams are controlled using the classes.
- (3) The `istream` and `ostream` classes control input and output functions respectively.
- (4) The `iostream` class is also a derived class. It is derived from `istream` and `ostream` classes. There are three more derived classes which are useful. They are `istream_withassign`,

`ostream_withassign`, and `iostream_withassign`. They are derived from `istream`, `ostream`, and `iostream` respectively.

(5) **Filebuf** accomplishes input and output operations with files.

The `fstreambase` acts as a base class for `fstream`, `ifstream`, and `ofstream`. The `ifstream` class is derived from `fstreambase` and `istream` by multiple inheritance. It can access member functions such as `get()`, `getline()`, `seekg()`, `tellg()` and `read()`. The `ofstream` class is derived from `fstreambase` and `ostream` classes. It can access the member functions such as `put()`, `seekp()`, `write()` and `tellp()`. The `fstream` class allows for simultaneous input and output on a `filebuf`. The member function of base classes `istream` and `ostream` starts the input and output.

(6) There are two methods (a) constructor of the class and (b) member function `open()` of the class for opening a file.

(7) The class `ofstream` creates output stream objects and `ifstream` creates input stream objects.

(8) The `close()` member function closes the file.

(9) The `open()` function uses the same stream objects. The `open()` function has two arguments. First is file name and second mode.

(10) When end of file is detected, the process of reading data can be easily terminated. The `eof()` function is used for this purpose. The `eof()` stands for end of file. It is an instruction given to the program by the operating system that end of file is reached. The `eof()` function returns 1 when end of file is detected.

(11) The mode `ios::out` and `ios::trunc` are near about same.

The `ios::app` lets the user to add data at the end of file whereas `ios::ate` allows user to add or update data anywhere in the file.

(12) The `seekg()` function shifts the associated file's input (get) file pointer. The `seekp()` function shifts the associated file's output (put) file pointer.

(13) `seekg()` — Shifts input (get) pointer to a given location.

(14) `seekp()` — Shifts output (put) pointer to a given location.

(15) `tellg()` — Provides the present position of the input pointer.

(16) `tellp()` — Provides the present position of the output pointer.

(17) The `put( )` and `get( )` functions are used to read or write a single character whereas `write( )` and `read( )` are used to read or write block of binary data.

(18) The `fail( )`, `eof( )`, `bad( )` and `good( )` are error trapping functions.

(19) An executable program that performs a specific task for operating system is called as command. The commands are issued from the command prompt of operating system. Some arguments are to be associated with the commands and hence these arguments are called as command line arguments.

(20) **Syntax** `main( int argc, char * argv[ ] )`

### EXERCISES

[A] Answer the following questions.

(1) What is a stream?

(2) What is a file?

(3) Describe the different derived classes from `ios` that control the disk I/O operations.

(4) Describe the two methods of opening of file.

(5) Explain detection of end of file with function `eof( )`.

(6) Describe the syntax of `open( )` function with its arguments.

(7) Which are the different types of file opening modes? List their names with meaning.

(8) Describe file manipulators with their syntaxes.

(9) Describe the various errors trapping functions.

(10) What are the possible reasons for failure of `open( )` function?

(11) What are command line arguments?

(12) Explain the use of `not` operator and `eof( )` function.

(13) Explain sequential and random file operations.

(14) How do you write a data in file in binary format?

(15) What are the limitations of using `put( )` and `get( )` functions?

(16) Explain the uses of `ostrstream` and `istrstream` class.

(17) Explain the difference between Binary and ASCII files.

[B] Answer the following by selecting the appropriate option.

(1) The `eof( )` stands for

(a) end of file

(b) error opening file

(c) error of file

(d) none of the above

(2) Command line arguments are used with function

- (a) main ( )
  - (b) member function
  - (c) with all functions
  - (d) none of the above

(3) The statement `in.seekg(0, ios::end)` sets the file pointer

- (a) at the end of file
  - (b) at the beginning of file
  - (c) in the middle of file
  - (d) none of the above

#### (4) The close( ) function

- (a) closes the file
  - (b) closes all files opened
  - (c) closes only read mode file
  - (d) none of the above

(5) The write( ) function writes

- (a) single character
  - (b) object
  - (c) string
  - (d) none of the above

(6) What will be the value of variable c after execution of the following program?

```
# include <fstream.h>

void main ( )
{
char text[5];
ofstream out ("data.txt");
out<<"Programming with ANSI and TurboC C";
out.close( );

ifstream in ("data.txt");
int c=0;

while (!in.eof( ))
{ in.getline(text,5);
c++;      }
}
a)    c=9
b)    c=5
```

(7) What will be the values of variable *j* and *k* after execution of the following program?

```
# include <fstream.h>
```

```
void main ( )
{
    ofstream j;
    ifstream k;
    cout <<"j="<<sizeof(j)<<endl;
    cout <<"k="<<sizeof (k);
}
```

- (a) j=78 k=80
  - (b) j=80 k=80
  - (c) j=80 k=78
  - (d) none of the above
- (8) The object of `fstream` class provides
- (a) both read and write operations
  - (b) only read operation
  - (c) only write operation
  - (d) none of the above
- (9) To add data at the end of file the file must be opened in
- (a) append mode
  - (b) read mode
  - (c) write mode
  - (d) both (a) and (c)
- (10) When a file is opened in read or write mode the file pointer is set
- (a) at the beginning of file
  - (b) at the end of file
  - (c) in the middle of file
  - (d) none of the above
- (11) While performing file operation this file must be included in
- (a) `fstream.h`
  - (b) `iostream.h`
  - (c) `constream.h`
  - (d) all of the above
- (11) The constructor of this class requires file name and mode for opening file
- (a) `ifstream`
  - (b) `ofstream`
  - (c) `fstream`
  - (d) all of the above

**[C] Attempt the following programs.**

(1) Write a program to open a file in output and input mode. Accept data and write to the file. Display the contents of the file.

(2) Write a program to write and read data in a file using object I/O functions `write( )` and `read( )`. Declare class with data members `name[20]`, `int billno`, `int amount_debited`, and `int received_amount` and `int balance`. Add 10 records and display the list of persons with balances. The user should have a facility to modify the existing records.

(3) Write a program to read and write a data file. While writing data follow the following instructions:

(a) Add one space between two successive words.

(b) Capitalize the first character of sentences.

(c) If numerical data is present put them in a bracket.

(4) Write a program to copy contents of one file to another file. Use command line arguments.

(5) Write a program to read a file. Display the total number of characters, words, lines, and blank spaces in the file.

(6) Write a program to exchange contents of two files. Use command line arguments.

(7) Write a program to compare two files in terms of number of bytes.

(8) Write a program to count characters and numericals present in a file.

(9) Write a program to print the contents of a file on printer.

(10) Write a program to check whether the given file exists or not. Display suitable message. User should enter the file name.

(11) Write a program to write contents of one file in reverse into another file.

(12) Write a program to find length of a file.

## [D] Find the bugs in the following programs.

(1)

```
ifstream in;
void main( )
{
    ifstream in;
    void show (void);
    in.open("data");
    if (in!=0)    show();
    else    cout <<"\n File not fount";
}

void show ( )
{   char c;
    cout <<"\n Contents of file : ";
```

```
while (in) { in.get(c); cout<<c; }
```

***Tip: Assume the file "data" exists.***

(2)

```
void main ( )
{
    char name[15];
    ofstream out("text");
    cin >>name;
    out<<name<<"\t";
    ifstream in("text"); in>>name;
    cout <<"\nName\t: "<<name<<"\n";
    in.close( );
}
```

(3)

```
void main ( )
{
    ifstream in("data");
    while (in) { cout<<(char)in.get( ); }
    return 0;
}
```

***Tip: Assume the file "data" exists.***

(4)

```
void main ( )
{
    fstream in("data");

    while (in.eof( )==0)
    {
        cout<<(char)in.get( );
    }
}
```

# 14

CHAPTER

## Generic Programming with Templates

C  
H  
A  
P  
T  
E  
R  
  
O  
U  
T  
L  
I  
N  
E

- 14.1 Introduction
- 14.2 Need of Template
- 14.3 Definition of Class Template
- 14.4 Normal Function Template
- 14.5 Working of Function Templates
- 14.6 Class Template with More Parameters
- 14.7 Function Templates with More Arguments
- 14.8 Overloading of Template Functions
- 14.9 Member Function Templates

- [14.10 Recursion with Template Function](#)
- [14.11 Class Template with Overloaded Operators](#)
- [14.12 Class Template Revisited](#)
- [14.13 Class Templates and Inheritance](#)
- [14.14 Bubble Sort Using Function Template](#)
- [14.15 Guidelines for Templates](#)
- [14.16 Difference Between Templates and Macros](#)
- [14.17 Linked List with Templates](#)

## 14.1 INTRODUCTION

Template is one of the most useful characteristics of C++. Few old compilers do not allow template mechanism. Templates are part of ANSI C++ standard. All major compilers support templates in their new versions. Instantiation is the activity of creating particular type using templates. The specific classes are known as instance of the template. The template provides generic programming by defining generic classes. In templates, generic data types are used as arguments and they can handle a variety of data types. **A function that works for all C++ data types is called as generic function.** Templates help the programmer to declare group of functions or classes. When used with functions they are called function templates. For example, we can create a template for function `square( )`. It helps us to calculate square of a number of any data type including `int`, `float`, `long`, and `double`. The templates associated with classes are called as class templates. A template is almost same as macro and it is type-secured. The difference between template and macro is illustrated in Section 14.16.

## 14.2 NEED OF TEMPLATE

Template is a technique that allows using a single function or class to work with different data types. Using template we can create a single function that can process any type of data i.e., the formal arguments of template functions are of template (generic) type. They can accept data of any type such as `int`, `float`, `long` etc. Thus, a single function can be used to accept values of different data types. Figures [14.1](#) and [14.2](#) clear the difference between non-template function and template function. Normally, we overload function when we need to handle different data types. This approach increases the program size. The disadvantages are that not only program length is increased but also more local variables are created in the memory. Template safely overcomes all the limitations occurring while overloading function and allows better flexibility to the program. The portability provided by a template mechanism can be extended to classes. The following sections describe use of template in different concepts of C++ programming. Let us understand the process of declaration of a template.

## 14.3 DEFINITION OF CLASS TEMPLATE

To declare a class of template type, following syntax is used.

### Template Declaration

```
template class <T>
class name_of_class
{
    // class data member and function
}
```

The first statement `template class <T>` tells the compiler that the following class declaration can use the template data type. `T` is a variable of template type that can be used in the class to define variable of template type. Both `template` and `class` are keywords. The `<>` (angle bracket) is used to declare variables of template type that can be used inside the class to define variables of template type. One or more variables can be declared separated by comma. Templates cannot be declared inside classes or functions. They must be global and should not be local.

```
T k;
```

Where, `k` is the variable of type template. Most of the authors use `T` for defining template; instead of `T`, we can use any alphabet.

**14.1 Write a program to show values of different data types using overloaded constructor.**

```

# include <iostream.h>
# include <conio.h>
class data

{
public :

data (char c)
{ cout <<"\n"<< " c = "<<c <<" Size in bytes :"<<sizeof(c); }

data (int c)
{ cout <<"\n"<< " c = "<<c <<" Size in bytes :"<<sizeof(c); }

data (double c)
{ cout <<"\n"<< " c = "<<c <<" Size in bytes :"<<sizeof(c); }
};

int main ( )
{ clrscr( );
  data h('A');    // passes character type data
  data i(100);   // passes integer type data
  data j(68.2);  // passes double type data
  return 0;
}

```

## **OUTPUT**

**c = A Size in bytes :1**  
**c = 100 Size in bytes :2**  
**c = 68.2 Size in bytes :8**

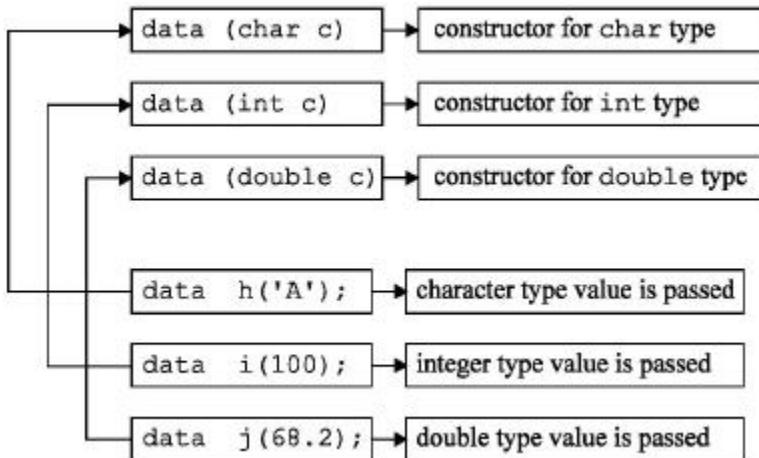
**Explanation:** In the above program, the class data contains three overloaded one argument constructor. The constructor is overloaded for `char`, `int`, and `double` type. In function `main()`, three objects `h`, `i` and `j` are created and values passed are of different types. The values passed are `char`, `int`, and `double` type. The compiler invokes different constructors for different data types. Here, in order to manipulate different data types we require overloading the constructor i.e., defining separate function for each non-compatible data type. This approach has the following disadvantages:

(1) Re-defining the functions separately for each data type increases the source code and requires more time.

(2) The program size is increased. Hence, occupies more disk space.

(3) If function contains bug, it should be corrected in every function.

Figure 14.1 shows the working of the program.



**Fig. 14.1** Working of non-template function

From the above program it is clear that for each data type we need to define separate constructor function. According to data type of argument passed respective constructor is invoked. C++ provides templates to overcome such problems and helps a programmer to develop generic program. The same program is illustrated with template as follows.

#### 14.2 Write a program to show values of different data types using constructor and template.

```

#include <iostream.h>
#include <conio.h>
template <class T>

class data
{
public :
    data (T c)
    {
        cout <<"\n" << " c = "<<c <<" Size in bytes :"<<sizeof(c);
    }
};

int main( )
{
    clrscr( );
    data <char> h('A');

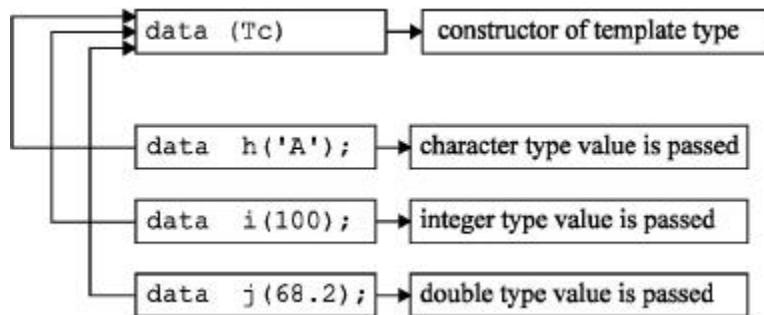
    data <int> i(100);
    data <float> j(3.12);
    return 0;
}

```

#### OUTPUT

```
c = A Size in bytes :1  
c = 100 Size in bytes :2  
c = 3.12 Size in bytes :4
```

**Explanation:** In the above program, the constructor contains variable of template T. The template class variable can hold values of any data type. While declaring an object the data type name is given before the object. The variable of template type can accept values of any data type. Thus, the constructor displays the actual values passed. The template variable c can hold values of any data type. The value and space in bytes required by these variables are displayed as the output. The size of data type changes according to the data types used in the program. [Figure 14.2](#) shows the working of the program.



**Fig. 14.2** Working of template function

In the above program, different values are passed using constructor, but for all data types same template function is used.

#### 14.4 NORMAL FUNCTION TEMPLATE

In the last section, we saw how to make a template class. In the same fashion a normal function (not a member function) can also use template arguments. The difference between normal and member function is that normal functions are defined outside the class. They are not members of any class and hence can be invoked directly without using object of dot operator. The member functions are the class members. They can be invoked using object of the class to which they belong. The declaration of template member function is described later in this chapter. In C++ normal functions are commonly used as in C. However, the user who wants to use C++ as better C, can utilize this concept. The declaration of template normal function can be done as follows.

##### Normal Template Function Declaration

```
template class <T>
```

```

Function_name ( )
{
    // code
}

The following program shows practical working of template function.

14.3 Write a program to define normal template function.

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

template <class T>

void show ( T x)
{ cout<<"\n x="<<x ;    }

void main( )
{
    clrscr( );
    char c='A';
    int i=65;
    double d=65.254;

    show(c);
    show(i);
    show(d);
}

```

### **OUTPUT**

```

x=A
x=65
x=65.254

```

**Explanation:** Before the body of the function `show()`, template argument `T` is declared. The function `show()` has one argument `x` of template type. As explained earlier, the template type variable can accept all types of data. Thus, the normal function `show()` can be used to display values of different data types. In `function()`, the `show()` function is invoked and `char`, `int` and `double` type of values are passed. The same is displayed in the output.

You are now familiar with utilities of templates. One more point to remember is that when we declare a class template, we can define class data member of template type as well as the member function of the class can also use the template member. For making member function of template type, no separate declaration is needed. The following program explains the above point.

### **14.4 Write a program to define data members of template type.**

```

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

template <class T>

class data
{
    T x;
public :

    data (T u) { x=u; }

    void show (T y)
    {
        cout<<" x="<<x;
        cout<<" y=" <<y<<"\n";
    }
};

int main( )
{
    clrscr( );
    data <char> c('B');
    data <int> i(100);
    data <double> d(48.25);

    c.show('A');
    i.show(65);
    d.show(68.25);
    return 0;
}

```

## **OUTPUT**

**x=B y=A  
x=100 y=65  
x=48.25 y=68.25**

**Explanation:** In this program, before declaration of class data, template <class T> is declared. This declaration allows entire class including member function and data member to use template type argument. We have declared data member x of template type. In addition, the one argument constructor and member function show( ) also have one formal argument of template type.

## **14.5 Write a program to create square( ) function using template.**

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

```

# include <conio.h>
template <class S>

class sqr
{
public :

sqr (S c)
{
    cout <<"\n" << " c = " <<c*c;
}
};

int main( )
{
clrscr( );
sqr <int> i(25);
sqr <float> f(15.2);

return 0;
}

```

## **OUTPUT**

**c = 625**  
**c = 231.039993**

**Explanation:** In the program in the previous page, the class `sqr` is declared. It contains a constructor with one argument of template type. In `main()` function, the object `i` indicates `int` type and `f` indicates `float` type. The object `i` and `f` invokes the constructor `sqr()` with values 15 and 15.2. The constructor displays the squares of these numbers.

## **14.5 WORKING OF FUNCTION TEMPLATES**

In the last few examples, we learned how to write a function template that works with all data types. After compilation, the compiler cannot guess with which type of data the template function will work. When the template function is called at that moment, from the type of argument passed to the template function, the compiler identifies the data type. Every argument of template type is then replaced with the identified data type and this process is called as *instantiating*. Thus, according to different data types respective versions of template functions are created. Though the template function is compatible for all data types, it will not save any memory. When template functions are used, four versions of functions are used. They are data

type int, char, float, and double. The programmer need not write separate functions for each data type.

## 14.6 CLASS TEMPLATE WITH MORE PARAMETERS

Like functions, classes can be declared to handle different data types. Such classes are known as **class templates**. These classes are generic type and member functions of these classes can operate on different data types. The class template may contain one or more parameters of generic data type. The arguments are separated by comma with template declaration. The declaration is as follows:

### Template with Multiple Parameters

```
template <class T1, class T2>

    class name_of_class
{
    // class declarations and definitions
}
```

## 14.6 Write a program to define a constructor with multiple template variables.

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

template <class T1, class T2>
class data
{ public :
    data (T1 a, T2 b)
    { cout <<"\n a = "<<a <<" b = "<<b; }
};

int main( )
{ clrscr( );
    data <int,float> h(2,2.5);
    data <int,char> i(15,'C');
    data <float,int> j(3.12,50);
    return 0; }
```

## OUTPUT

**a = 2 b = 2.5**

**a = 15 b = C**

**a = 3.12 b = 50**

**Explanation:** In the above program, the data constructor has two arguments of generic type. While creating objects, type of arguments is mentioned in the angle bracket. When arguments are more than one they are separated by comma. Consider the statement `data< int, float > h (2,2.5)`. It

tells the compiler that the first argument is of `integer` type and second argument is of `float` type. When objects are created, constructors are called and values are received by the template arguments. The output of the program is given above.

## 14.7 FUNCTION TEMPLATES WITH MORE ARGUMENTS

In previous examples, we defined constructors with arguments. In the same fashion, we can also define member functions with generic arguments. The format is given below:

### Function Template Declaration

```
template <class T>
return_data_type function_name (parameter of template type)
{
    statement1;
    statement2;
    statement3;
}
```

### 14.7 Write a program to display the elements of integer and float array using template variables with member function.

```
# include <iostream.h>
# include <conio.h>
template <class T1, class T2>

class data
{
public :

void show (T1 a, T2 b)
    { cout <<"\na = "<<a <<" b = "<<b;      }
};

int main( )
{
clrscr( );
int i[]={3,5,2,6,8,9,3};
float f[]={3.1,5.8,2.5,6.8,1.2,9.2,4.7};
data <int,float> h;

for (int m=0;m<7;m++)
h.show(i[m],f[m]);
return 0;
}
```

### OUTPUT

```
a = 3 b = 3.1
a = 5 b = 5.8
a = 2 b = 2.5
a = 6 b = 6.8
a = 8 b = 1.2
a = 9 b = 9.2
a = 3 b = 4.7
```

**Explanation:** In the above program, array elements of integer and float are passed to function `show( )`. The function `show( )` has two arguments of template type i.e., `a` and `b`. The `show( )` function receives integer and float values and displays them. The output of the program is shown above.

#### **14.8 Write a program to exchange values of two variables. Use template variables as function arguments.**

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

template <class E>

void exchange(E &a, E &b)
{
    E t=a;
    a=b;
    b=t;
}

int main( )
{
    clrscr( );
    int x=5,y=8;
    cout <<"\n Before exchange " <<x <<" y= " <<y;
    exchange (x,y);
    cout <<"\n After exchange " <<x <<" y= " <<y;
    return 0;
}
```

#### **OUTPUT**

**Before exchange x= 5 y= 8**

**After exchange x= 8 y= 5**

**Explanation:** In the above program, `exchange( )` is a user-defined function. It has two arguments of template types. In `main( )` the function `exchange( )` is invoked and two integers are passed. The

function exchange( ) receives these values by reference. The values of two variables are exchanged with the help of temporary variable t. In the output the values of variables x and y before and after execution of function exchange( ) are displayed.

#### **14.9 Write a program to define function template with multiple arguments.**

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

template <class A, class B, class C>

void show(A c,B i,C f)
{ cout <<"\n c = "<<c <<" i = "<<i <<" f = "<<f; }

int main( )
{
    clrscr( );
    show('A',8,50.25);
    show(7,'B',70.89);
    return 0;
}
```

#### **OUTPUT**

**C = A i = 8 f = 50.25**

**C = 7 i = B f = 70.89**

**Explanation:** In the above program, templates with A, B, and C classes are declared. The function show( ) has three arguments of type A, B and C respectively. In main( ), the function show( ) is invoked and three values of different data types are passed. The function show( ) displays the values on the screen.

#### **14.8 OVERLOADING OF TEMPLATE FUNCTIONS**

A template function also supports overloading mechanism. It can be overloaded by normal function or template function. While invoking these functions an error occurs if no accurate match is met. No implicit conversion is carried out in parameters of template functions. The compiler follows the following rules for choosing appropriate function when program contains overloaded functions.

- (1) Searches for accurate match of functions; if found it is invoked.
- (2) Searches for a template function through which a function that can be invoked with accurate match can be generated; if found it is invoked.

- (3) Attempts normal overloading declaration for the function.  
(4) In case no match is found, an error will be reported.

#### **14.10 Write a program to overload a template function.**

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

template <class A>

void show(A c)
{
    cout <<"\n Template variable c = "<<c;
}

void show (int f)
{
    cout <<"\n Integer variable f = "<<f;
}

int main( )
{
clrscr( );
show('C');
show(50);
show(50.25);

return 0;
}
```

#### **OUTPUT**

```
Template variable c = C
Integer variable f = 50
Template variable c = 50.25
```

**Explanation:** In the above program, the function `show( )` is overloaded. One version contains template arguments and other version contains integer variables. In `main( )`, the `show( )` function is invoked three times with `char`, `int` and `float` and values are passed. The first call executes the template version of function `show( )`, the second call executes integer version of function `show( )` and the third call again invokes the template version of function `show( )`. Thus, in the above fashion template functions are overloaded.

#### **14.9 MEMBER FUNCTION TEMPLATES**

In the previous example, the template function defined were inline i.e., they were defined inside the class. It is also possible to define them outside the class. While defining them outside, the function template should define the function and the template classes are parameterized by the type argument.

#### **14.11 Write a program to define definition of member function template outside the class and invoke the function.**

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

template <class T>
class data
{
    public :
        data (T c);

};

template <class T>
data<T>::data (T c)
{
    cout <<"\n" << " c = " <<c;
}

int main( )
{
    clrscr( );
    data <char> h('A');
    data <int> i(100);
    data <float> j(3.12);
    return 0;
}
```

#### **OUTPUT**

```
c = A
c = 100
c = 3.12
```

**Explanation:** In the above program, the constructor is defined outside the class. When we define outside the class, the member function should be preceded by the template name as given below:

#### **Statements of program 14.11**

```
template <class T>
data<T>::data (T c)
```

The first line defines template and second line indicates template class type T.

### **14.10 RECURSION WITH TEMPLATE FUNCTION**

Like normal function and member function, the template function also supports recursive execution of itself. The following program illustrates this:

#### **14.12 Write a program to invoke template function recursively.**

```
# include <iostream.h>
# include <conio.h>
# include <process.h>
# include <stdlib.h>
# include <assert.h>

template <class O>

void display (O d)
{
    cout<<(float) (rand( )%(d*d))<<"\n";

    // if (d==1) exit(1);
    assert(d!=1);
    display(--d);

}

void main ( )
{
    int x=10;
    clrscr( ),

    display(x);
}
```

#### **OUTPUT**

**46**

**49**

**38**

**12**

**28**

**17**

**11**

**7**

**0**

**0**

**Assertion failed: d!=1, file REC\_TEMP.CPP, line 14**

**Abnormal program termination**

**Explanation:** The template function generates random number and displays it each time when the function `display( )` is invoked. The function `rand( )`

) defined in stdlib.h is used. The function calls itself recursively until the value of d becomes 1. The assert( ) statement checks the value of d and terminates the program when condition is satisfied. The assert( ) is defined in assert.h. We can also use if statement followed by exit( ) statement as given in comment statement.

## 14.11 CLASS TEMPLATE WITH OVERLOADED OPERATORS

The template class permits to declare overloaded operators and member functions. The syntax for creating overloaded operator function is similar to class template members and functions. The following program explains the overloaded operator with template class.

### 14.13 Write a program to overload + operator for performing addition of two template based class objects.

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

template <class T>

class num
{
private :
T number;

public:

num ( ) { number=0; }
void input( )
{
    cout <<"\n Enter a number : ";
    cin>>number;
}
num operator +(num);
void show( ) { cout <<number; }
};

template <class T>
num <T> num <T> :: operator + (num <T> c)
{
    num <T> tmp;
    tmp.number=number+c.number;
    return (tmp);
}

void main( )
```

```
{  
    clrscr( );  
    num <int> n1,n2,n3;  
    n1.input( );  
  
    n2.input( );  
    n3=n1+n2;  
    cout <<"\n\t n3 = "  
    n3.show( );  
}
```

## OUTPUT

Enter a number : 8

Enter a number : 4

n3 = 12

**Explanation:** In the above program, class num is declared with template variable number. The input ( ) function is used to read number through the keyboard. The operator + ( ) function performs addition of elements of two template objects.

In function main ( ), n1, n2 and n3 are three objects of num class. The statement n3=n1+n2 invokes the overloaded operator and addition of two objects is performed. The show ( ) function displays the contents of the object.

## 14.12 CLASS TEMPLATE REVISITED

As explained earlier, the template mechanism can be used safely in every concept of C++ programming such as functions, classes etc. We also learned how to define class templates. Class templates are frequently used for storage of data and can be very powerfully implemented with data structures such as stacks, linked lists etc. So far, our conventional style is to create a stack or linked list that manipulates or stores only a single type of data, for example, int. If we want to store float data type we need to create a new class to handle float data type and so on can be repeated for various data types. However, template mechanism saves our work of re-defining codes by allowing a single template based class or function to work with different data types.

Reusability is the facility provided by inheritance in which a class can be reused. Template also provides reusability. Inheritance allows reuse of previously defined classes whereas using templates, repetitive definition of functions can be avoided and a single function can be used for multiple data

types. Hence, the template based function code is reused instead of defining multiple functions.

#### 14.14 Write a program to create template-based stack. Store integer and float in it.

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

template <class S>

class stack
{
    S num[5];
    int head;

public:
    stack ( ) { head=-1; }

    void push ( S d)
    {
        if (head==4) cout << endl << "Stack is Full";
        else
        {
            head++;
            num[head]=d;
        }
    }

    S pop ( )
    {
        if (head==-1) { cout << "\n Stack is empty";
                        return NULL;
        }
        else
        {
            S d=num[head];
            head--;
            return d;
        }
    }
};

void main ( )
{
    clrscr( );
    stack <int>s1;
    for (int j=10;j<=60;j+=10)
        s1.push(j);
```

```

for ( j=0;j<=5;j++)
cout<<endl<<s1.pop( );

stack <float>s2;
for ( j=1;j<=6;j++)
s2.push(.1+j );

for ( j=0;j<=5;j++)
cout<<endl<<s2.pop( );
}

```

## **OUTPUT**

**Stack is Full**

50  
40  
30  
20  
10

**Stack is empty**

0

**Stack is Full**

5.1  
4.1  
3.1  
2.1  
1.1

**Stack is empty**

0

**Explanation:** As shown in the output, the `stack` class created can store any type of data. In this program, it has stored `integer` and `float` type of data. The stack can store five elements. The member function `push( )` is used to add elements in the stack and `pop( )` function is used to remove elements from the stack. Both functions display messages when stack is either full or empty. When we perform the `push( )` operation beyond the limit i.e., pushing element more than stack capacity, the messages displayed will be “Stack is full.” When `pop( )` operation is performed beyond the limit i.e., an attempt to erase the element when no element is stored in the stack, the message will be “Stack is empty.” In function `main( )`, `s1` and `s2` are template type objects. Using these objects `integer` and `float` values are added and displayed respectively.

The order in which the elements are pushed in the stack is exactly opposite (reverse) to the order in which they are popped from the stack.

### 14.13 CLASS TEMPLATES AND INHERITANCE

The template class can also act as base class. When inheritance is applied with template class it helps to compose hierarchical data structure known as container class.

(1) Derive a template class and add new member to it. The base class must be of template type.

(2) Derive a class from non-template class. Add new template type member to derived class.

(3) It is also possible to derive classes from template base class and omit the template features of the derived classes.

As said earlier, the template characteristics of the base class can be omitted by declaring the type while deriving the class. All the template-based variables are substituted with basic data types. The following declaration illustrates derivation of a class using template featured base class.

```
Template <class TL,...>
    class XYZ
{
    // Template type data members and member functions
}

Template <class TL,...>
    class ABC : public XYZ <TL,...>
{
    // Template type data members and member functions
}
```

### 14.15 Write a program to derive a class using template base class.

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

template <class T>
class one
{
protected:
    T x,y;

    void display ( )
    {
        cout <<x;
    }
};
```

```

template <class S>

class two : public one <S>
{
    S z;
public :

    two (S a, S b, S c)
    {
        x=a;
        y=b;
        z=c;
    }
    void show ( )
{
cout <<"\n x="<<x <<" y="<< y<<" z="<<z;
}
};

void main ( )
{
    clrscr( );
    two <int> i(2,3,4) ;

    i.show( );
    two <float> f(1.1,2.2,3.3);
    f.show( );
}

```

## **OUTPUT**

**x=2 y=3 z=4**

**x=1.1 y=2.2 z=3.3**

**Explanation:** In the above program the class one is a base class and class two is a template-derived class. Consider the statement given below:

```

template <class S>
class two : public one <S>

```

Using the above statement derivation is carried out. The base class name is followed by template class name S. The function `show()` is a member of derived class `two()`. In function `main()`, I and f are objects of class `two`. The object I is used to pass integer elements and f is used to pass float numbers.

## **14.14 BUBBLE SORT USING FUNCTION TEMPLATE**

In application software, large amount of data is manipulated and sorting techniques are frequently used in such operations. The sorting mechanism helps to view the information in different ways. The template function can also be used in sorting operation. The following program illustrates the use of template function with bubble sorting method.

**14.16 Write a program to sort integer and float array elements using bubble sort method.**

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

template <class S>

void b_sort ( S u[], int k)
{
    for (int x=0;x<k-1;x++)
        for (int y=k-1;x<y;y--)

            if (u[y]<u[y-1]) // checks successive numbers
            {
                S p;
                p=u[y];           // assigns to temporary variable
                u[y]=u[y-1];      // exchange of elements
                u[y-1]=p;         // collects from temporary variable
            }
}

void main ( )
{
    clrscr( );
    int i[5]={4,3,2,1,-4};
    float f[5]={5.3,5.1,5.5,4.5,3.2};
    b_sort(i,5);          // call to function
    b_sort(f,5);          // call to function
    int x=0,y=0;
    cout <<"\nInteger array elements in ascending order:";

    while (x<5) cout <<i[x++] <<" ";    // display contents of integer
array

    cout <<"\nFloat array elements in ascending order : ";

    while ( y<5) cout <<f[y++] <<" ";   // displays contents of float
array
}
```

**OUTPUT**

**Integer array elements in ascending order:-4 1 2 3 4**

**Float array elements in ascending order : 3.2 4.5 5.1 5.3 5.5**

**Explanation:** In the above program, `b_sort( )` is a function with template and integer arguments. The two `for` loops are used to obtain two successive numbers of the array. The `if` statement checks the two successive numbers and if second number is greater than first, they are exchanged using assignment statements. In function `main( )`, integer and float arrays are declared and initialized. The base addresses (of both arrays) of arrays and number of elements are passed to the function `b_sort( )`. The function `b_sort( )` arranges contents of both the arrays in ascending order. The two `while` loops display the contents of arrays.

## 14.15 GUIDELINES FOR TEMPLATES

(1) Templates are applicable when we want to create type secure class that can handle different data types with same member functions.

(2) The template classes can also be involved in inheritance. For example

```
template <class T>
class data : public base<T>
```

Both data and base are template classes. The class `data` is derived from template class `base`.

(3) The template variables also allow us to assign default values. For example,

```
template <class T, int x=20>
class data
{
    t num[x];
}
```

(4) The name of the template class is written differently in different situations. While class declaration, it is declared as follows:

```
class data { };
```

For member function declaration is as follows:

```
void data <T> :: show (T d) { }
```

Where, `show( )` is a member function.

Finally, while declaring objects the class name and specific data is specified before object name.

```
data <int> i1 // object of class data supports integer
values
data <float> f1 // object of class data supports float
values
```

(5) All template arguments declared in template argument list should be used for definition of formal arguments . If one of the template arguments is not used , the template will be specious. Consider the example:

```

Template < class T>
T show (void)
{
    return x;
}

```

In the above example, the template argument T is not used as a parameter and the compiler will report an error.

```

template <class T>
void show (int y)
{   T tmp; }

```

In the above example, template type argument is not an argument. Possibly the system will crash.

```

Template <class T, class S>
void max ( T & k)

{
S p;
}

```

The template variable S is not used. Therefore, compile time error is generated.

## 14.16 DIFFERENCE BETWEEN TEMPLATES AND MACROS

(1) Macros are not type safe i.e., a macro defined for integer operation cannot accept float data. They are expanded with no type checking .

(2) It is difficult to find errors in macros.

(3) In case a variable is post-incremented or decremented , the operation is carried out twice.

```

#define max(a) a + ++a

void main ( )
{ int a=10, c;
c=max(a);
cout<<c;
}

```

The macro defined in the above macro definition is expanded twice. Hence, it is a serious limitation of macros. The limitation of this program can be removed using templates as shown below:

## 14.17 Write a program to perform incrementation operation using template.

```

#include <iostream.h>
#include <constream.h>

template <class T>

```

```

T max(T k)
{
    ++k;
    return k;
}

void main ( )
{
    clrscr( );
    int a=10, c;
    c=max(a);
    cout<<c;
}

```

## **OUTPUT**

**11**

**Explanation:** In the above program, template function `max ( )` is defined. An integer value is passed to this function. The function `max ( )` increments the value and returns it.

## **14.17 LINKED LIST WITH TEMPLATES**

The linked lists are one among popular data structures. The following program explains the working of linked list with templates.

### **14.18 Write a program to create linked list with template class.**

```

# include <iostream.h>
# include <constream.h>

template <class L>

class list
{
    L t;
    list *next;
public :

    list ( L t);

void add_num (list *node)
{
    node->next=this;

    next=NULL;
}

list *obtainnext( ) { return next;}

```

```

L getnum ( ) { return t; }

};

template <class L> list <L>::list (L y)
{
    t=y;
    next=0;
}

void main ( )
{
    clrscr();
    list <int> obj(0);
    list <int> *opt, *last;
    last=&obj;
    int j=0;

    while (j<15)
    {
        opt=new list <int> (j+1);
        opt->add_num (last);
        last=opt;
        j++;
    }
    opt=&obj;

    j=0;

    while (j<15)
    {
        cout<<opt->getnum() << " ";
        opt=opt->obtainnext();
        j++;
    }
}

```

## **OUTPUT**

**0 1 2 3 4 5 6 7 8 9 10 11 12 13 14**

***Explanation:*** In the above program, the class template is declared in the same way as in previous programs. The class list has two arguments. One is of template L type (t) other is of object of the class list (\*next). The pointer next points to the next element of the linked list. The add\_num ( ) function is used to add successive numbers in the linked list. The function obtainnext ( ) returns address of the next element. The function declarator of obtainnext ( ) is preceded by symbol '\*' because it

returns address of the element. In function `add_num()` the pointer `next` is initialized with zero. If we do not initialize the `next` with zero it will become wild pointer and may point to any location of the system. Using loop, successive numbers are added and displayed.

### SUMMARY

- (1) Template is one of the most useful characteristics of C++. It is newly added in C++. Instantiation is the activity of creating particular type using templates. The specific classes are known as instance of the template. The template provides generic programming by defining generic classes. In templates, generic data types are used as arguments and they can handle a variety of data types.
- (2) Declaration and definition of every template class starts with the keyword `template` followed by parameter list.
- (3) The class template may contain one or more parameters of generic data type. The arguments are separated by comma with template declaration.
- (4) Function template can be defined with one or more parameters.
- (5) A template function also supports overloading mechanism. It can be overloaded by normal function or template function.
- (6) It is also possible to define member function definition outside the class. While defining them outside, the function template should define the function and the template classes are parameterized by the type argument.

### EXERCISES

#### [A] Answer the following questions.

- (1) What are templates?
- (2) How can normal function be declared as template function?
- (3) How can member function be declared as template function?
- (4) How can templates be used for generic programming?
- (5) Explain need of templates.
- (6) What do you mean by overloading of template function?
- (7) Explain class template.
- (8) What are the differences between templates and macros?
- (9) Explain inheritance with class template.
- (10) Explain working of template function.
- (11) Describe guidelines for defining templates.

#### [B] Answer the following by selecting the appropriate option.

- (1) templates are suitable for
  - (a) any data type
  - (b) basic data type

- (c) derived data type  
(d) all of the above
- (2) template can be declared using the keyword  
(a) template  
(b) try  
(c) class  
(d) none of the above
- (3) template class is also called as  
(a) generic class  
(b) container class  
(c) virtual class  
(d) base class
- (4) Function template can accept  
(a) only one parameter  
(b) only two parameters  
(c) any number of parameters  
(d) none of the above
- (5) The statement temp <void> x where temp is class template  
(a) object x can be used to pass integer values  
(b) object x can be used to pass float values  
(c) cannot be used to pass values but can invoke member function  
(d) none of the above
- (6) Observe the following program:

```
void show (int x) { cout<<x; } // normal function

template <class T>
void show (T t) { cout<<"\t"<<t; } // template function

void main ( )
{
    show(5);
}
```

Which function will be invoked first?

- (a) normal function  
(b) template function  
(c) ambiguity is generated  
(d) none of the above
- (7) Select the correct template definition:  
(a) template <class T>  
(b) class < template T>

- (c) template <T>  
(d) template class <T>
- (8) Function template are normally defined  
(a) in function main( )  
(b) globally  
(c) anywhere  
(d) in a class
- (9) Select the true statement from sentences given below related with function template:  
(a) The variable names used as formal argument in the template function should be unique.  
(b) Every formal variable in the template definition should appear at least once in function signature.  
(c) All template definition function must start with the keyword template.  
(d) The code in the template varies every time the template function is instantiated.
- (10) Which of the following statement declares a function template which does not return any value and contains pointer of type T.  
(a) void sort (T \*a);  
(b) void sort (T \*a);  
(c) T sort (T \*a);  
(d) void sort (T a);

**[C] Attempt the following programs.**

- (1) Write a program to define template and display the absolute value for data type int, double, and long.
- (2) Write a program to define a template and display the following lines through it:  
(a) String “ABCDEFGHIJKLM”  
(b) Digits 1,2,3,4,5,6,7,8,9  
(c) Float 1.2,2.2,3.2,5.4,8.5,2.6,2.4
- (3) Write a program to create a template to find the maximum value stored in an array.
- (4) Write a program to find reverse of given number using template based function.
- (5) Write a program to convert a given decimal number to binary using template function. The user must be able to convert integer and decimal number to hexadecimal.

(6) Write a program to show values of different data types using constructors and templates.

(7) Write a program to define normal template function.

(8) Write a program to pass an object to template function. Display the contents of members of object.

(9) Write a program to display the reverse string using template function.

## [D] Find bugs in the following programs.

(1)

```
template <class T>

void show (T t) { cout <<t; }

class data
{
public:
    int x,y,z; data () { x=1; y=2; z=3; }
};

void main ( )
{ data X;
    show(X);
}
```

(2)

```
template <class T>
class temp
{

    T t;
public:
    temp () {t=0;}
    void show () { cout <<; } };

void main ( )
{ temp <void> x;
    x.show(); }
```

(3)

```
template <class T>
class one
{ protected:
    T x,y;
};

class two : public one <T>
{ S z;
public :
two (S a, S b, S c)
```

```
{      x=a;      y=b;      z=c;      }
void show ( )
{ cout <<"\n x=<<x <<" y=<< y<<" z=<<z;  }
};

void main ( )
{
    two <float> f(1.1,2.2,3.3);
    f.show( );
}

(4)
template <class T>

T main (T x)
{
    cout<<x;
}
```

15

CHAPTER

# Exception Handling

- 15.1 Introduction
- 15.2 Principles of Exception Handling
- 15.3 The Keywords try, throw and catch
- 15.4 Exception Handling Mechanism
- 15.5 Multiple Catch Statements
- 15.6 Catching Multiple Exceptions
- 15.7 Rethrowing Exception
- 15.8 Specifying Exception
- 15.9 Exceptions in Constructors and Destructors
- 15.10 Controlling Uncaught Exceptions
- 15.11 Exception and Operator Overloading
- 15.12 Exception and Inheritance
- 15.13 Class Template with Exception Handling

## —•15.14 Guidelines for Exception Handling

### 15.1 INTRODUCTION

While writing large programs, a programmer makes many mistakes. Due to this, bugs occur even in the released softwares. Developing an error free program is the objective and intention of the programmer. Programmers have to take care to prevent errors. Errors can be trapped using exception-handling features.

Few languages support exception-handling features. Without this feature, the programmers have to find out errors himself/ herself. The errors may be logical errors or syntactic mistakes (syntax mistakes). The logical errors remain in the program due to poor understanding of the program. The syntax mistakes are due to lack of understanding of the programming language. C++ provides exception-handling procedure to reduce the errors that a programmer makes.

The programmer always faces unusual errors while writing programs. An exception is an abnormal termination of a program, which is executed in a program at run-time or it may be called at runtime when an error occurs. The exception contains warning messages like invalid argument, insufficient memory, division by zero, and so on. ANSI C++ has built in language functions for trapping the errors and control the exceptions. All C++ compilers support this newly added facility.

### 15.2 PRINCIPLES OF EXCEPTION HANDLING

Like errors, exceptions are also of two types. They are as follows:

- (a) Synchronous exceptions
- (b) Asynchronous exceptions

C++ has a well-organized object-oriented method to control run-time errors that occur in the program. The goal of exception handling is to create a routine that detects and sends an exceptional condition in order to execute suitable action. The routine needs to carry following responsibilities:

- (1) Detect the problem
- (2) Warn that an error has come
- (3) Accept the error message
- (4) Perform accurate actions without troubling the user

An exception is an object. It is sent from the part of the program where an error occurs to that part of the program which is going to control the error.

Exception provides an explicit pathway that contains errors to the code that controls the errors.

### 15.3 THE KEYWORDS TRY, THROW AND CATCH

Exception handling technique passes control of program from a location of exception in a program to an exception handler routine linked with the **try block**. An exception handler routine can only be called by throw statement.

#### (1) THE KEYWORD TRY

The try keyword is followed by a series of statements enclosed in curly braces.

**Syntax of try statement**

```
try
{
    statement 1;
    statement 2;
}
```

#### (2) THE KEYWORD THROW

The function of throw statement is to send the exception found. The declaration of throw statement is as given below:

**Syntax of throw statement**

```
throw (excep);
throw excep;
throw // re-throwing of an exception
```

The argument excep is allowed in any type and it may be a constant.

The catch block associated with try block catches the exception thrown.

The control is transferred from try block to catch block.

The throw statement can be placed in function or in nested loop but it should be in try block. After throwing exception, control passes to the catch statement.

#### (3) THE KEYWORD CATCH

Like try block, catch block also contains a series of statements enclosed in curly braces. It also contains an argument of exception type in parenthesis.

**Syntax of catch statement**

```
try
{
    Statement 1;
    Statement 2;
}
catch ( argument)
```

```
{  
    statement 3; // Action to be taken  
}
```

When exception is found the `catch` block is executed. The `catch ( )` statement contains an argument of exception type and it is optional. When argument is declared it can be used in the `catch` block. After execution of the `catch` block, the statements inside the blocks are executed. In case no exception is caught, the `catch` block is ignored and if mismatch is found the program terminates.

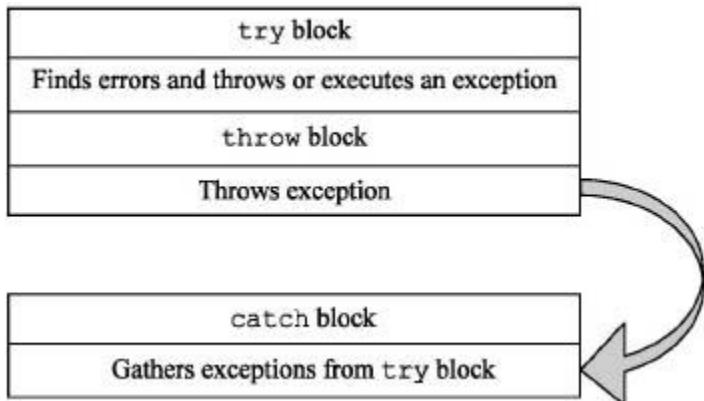
## 15.4 EXCEPTION HANDLING MECHANISM

C++ exception handling mechanism provides three keywords; `try`, `throw`, and `catch`. The keyword `try` is used at the starting of exception.

The `throw` block is present inside the `try` block. Immediately after `try` block, `catch ( )` block is present. Figure 15.1 shows `try`, `catch`, and `throw` statements.

As soon as an exception is found, the `throw` statement inside the `try` block throws an exception (a message) for `catch ( )` block that an error has occurred in the `try` block statement. Only errors occurring inside the `try` block are used to throw exception. The `catch` block receives the exception send by the `throw` block. The general form of the statement is shown in Figure 15.2.

When `try` block passes an exception using `throw` statement, the control of the program passes to the `catch` block. The data types used by `throw` and `catch` statements must be the same otherwise program gets aborted using `abort ( )` function, that is executed implicitly by the compiler. When no error is found and no exception is thrown, in such a situation `catch` block is disregarded and the statement after the `catch` block is executed.



**Fig. 15.1** Exception mechanism

```

Statement;
Statement;

try                                // Beginning of exception handling routine
{
    Statement;
    Statement;
    Statement;

    throw (object);                // Statements that
                                    // finds and forces to throw an exception

} catch (object) <-->           // Traps exception

{
    Statement;                   // Statement that controls exception
    Statement;
    Statement;
    Statement;
}
                                // End of exception handling routine

```

**Fig. 15.2** try, throw, and catch blocks

### 15.1 Write a program to throw exception when j=1 otherwise perform the subtraction of x and y.

```

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

int main( )
{
    int x,y;
    cout <<"\n Enter values of x and y\n";
    cin>>x>>y;
    int j;
    j=x>y ? 0 :1;

```

```

try
{
    if (j==0)
        cout <<"Subtraction (x-y)= "<<x-y<<"\n";
    else { throw(j); }
}

catch (int k)
{
cout <<"Exception caught : j = "<<j <<"\n";
}
return 0;
}

```

## OUTPUT

**Enter values of x and y**

**7 8**

**Exception caught: j = 1**

**Enter values of x and y**

**4 8**

**Exception caught: j = 1**

**Explanation:** In the above program, values of x and y are entered. The conditional operator tests the two numbers. If x is greater than y then zero is assigned to j, otherwise one. In the `try` block the `if( )` condition checks the value of j, the subtraction is carried out when j is zero otherwise the `else` block throws the exception. The `catch` statement catches the exception thrown. The output of the program is shown above.

## 15.5 MULTIPLE CATCH STATEMENTS

We can also define multiple catch blocks, in try blocks. Such program also contain multiple throw statements based on certain conditions. The format of multiple catch statement is given below:

```

try
{
    // try section
}
catch (object1)
{
    // catch section1
}
catch (object2)
    // catch section2

```

```

}
.....
.....
catch (type n object)
{
    // catch section-n
}

```

As soon as an exception is thrown, the compiler searches for appropriate by matching `catch ( )` block. The matching `catch ( )` block is executed and control passes to the successive statement after the last `catch ( )` block. In case no match is found, the program terminates. In multiple `catch ( )` statement, if objects of many catch statements are similar to type of an exception, in such a situation the first `catch ( )` block that matches is executed.

## **15.2 Write a program to throw multiple exceptions and define multiple catch statement.**

```

#include <iostream.h>
void num (int k)
{
    try
    {
        if (k==0) throw k;
        else
            if (k>0) throw 'P';
            else
                if (k<0) throw .0;
                cout <<"*** try block ***\n";
    }
    catch(char g)
    {
        cout <<"Caught a positive value \n";
    }

    catch (int j)
    {
        cout <<"caught a null value \n";
    }

    catch (double f)
    {
        cout <<"Caught a negative value \n";
    }

    cout <<"*** try catch ***\n \n";
}

```

```
int main( )
{
cout <<"Demo of multiple catches\n";
num(0);
num(5);
num(-1);

return 0;
}
```

## OUTPUT

**Demo of multiple catches**

**caught a null value**

**\*\*\* try catch \*\*\***

**Caught a positive value**

**\*\*\* try catch \*\*\***

**Caught a negative value**

**\*\*\* try catch \*\*\***

**Explanation:** In the above program, function num( ) contains try block with multiple catch blocks. In function main( ), the user-defined function num( ) is invoked with one argument. The if statement within the try block checks the number whether it is positive, negative or zero. According to this, exception is thrown and respective catch( ) block is executed. Here, in throw statement, objects of different data types such as int, char and double are used to avoid ambiguity.

## 15.6 CATCHING MULTIPLE EXCEPTIONS

It is also possible to define single or default catch( ) block from one or more exceptions of different types. In such a situation, a single catch block is used for catch exceptions thrown by multiple throw statements.

```
catch( )
{
    // statements for handling
    // all exceptions
}
```

## 15.3 Write a program to catch multiple exceptions.

```
# include <iostream.h>

void num (int k)
{
    try
```

```

{
    if (k==0) throw k;
    else
        if (k>0) throw 'P';
    else
        if (k<0) throw .0;
    cout <<"*** try block ***\n";
}

catch (...)
{
    cout <<"\n Caught an exception\n";
}

int main( )
{
    num(0);
    num(5);
    num(-1);

    return 0;
}

```

## **OUTPUT**

**Caught an exception**

**Caught an exception**

**Caught an exception**

**Explanation:** The above program is same as the previous one. Here, only one difference is observed and that is, instead of multiple catch blocks, single catch block is defined. For all the exceptions thrown, the same catch block is executed. It is a generic type of catch block. The statement `catch (...)` catches all the thrown exceptions.

## **15.7 RETHROWING EXCEPTION**

It is also possible to pass again the exception received to another exception handler i.e., an exception is thrown from `catch( )` block and this is known as rethrowing of exception. The following statement accomplishes this:  
`throw;`

The above `throw` statement is used without any argument. This statement throws the exception caught to the next `try catch` statement.

The following program illustrates this.

## **15.4 Write a program to rethrow an exception.**

```
# include <iostream.h>
void sub( int j, int k)
{
    cout <<"inside function sub ( )\n";
    try
    {
        if (j==0)
            throw j;
        else

            cout <<"Subtraction = "<<j-k <<"\n";
    }
    catch (int)
    {
        cout <<"Caught null value \n";
        throw;
    }
    cout << "*** End of sub ( ) ***\n\n";
}

int main( )
{
cout <<"\n inside function main ( )\n";
try
{
sub (8,5);
sub (0,8);
}
catch (int)
{
    cout <<"caught null inside main ( ) \n";
}
cout <<"end of function main ( ) \n";

return 0;

}
```

## **OUTPUT**

```
inside function main ( )
inside function sub ( )
Subtraction = 3
** End of sub ( ) ***
```

```
inside function sub ( )
Caught null value
caught null inside main ( )
end of function main ( )
```

**Explanation:** In the above program, two `try` blocks are defined. One is defined in function `sub( )` and other in function `main( )`. The `sub( )` function has two integer arguments. When `sub( )` function is invoked two integer values are sent to this function. The `if` statement in `try` block of `sub( )` function checks whether the value of first variable i.e., `j` is zero or non-zero. If it is non-zero, subtraction is carried out otherwise the `throw` statement throws an exception. The `catch( )` block inside the function `sub( )` collects this exception and again throws the exception using the statement `throw`. Here, `throw` statement is used without any argument. The `catch` block of function `main` catches the rethrown exception.

## 15.8 SPECIFYING EXCEPTION

The specified exceptions are used when we want to bind the function to throw only specified exception. Using a `throw` list condition can do this. The format for such exception is given below.

### Specifying Exceptions

```
data-type function_name (parameter list) throw (data type list)
{
    Statement 1;
    Statement 2; Function definition
    Statement 3;
}
```

The data type list indicates the type of exceptions to be thrown. If we want to deny a function from throwing exception, we can do this by declaring data type list `void` as per the following statement.

```
throw(); // void or vacant list
```

## 15.5 Write a program to restrict a function to throw only specified type of exceptions.

```
# include <iostream>
using namespace std;

void check (int k) throw (int,double)
{
    if (k==1) throw 'k';
    else
        if (k==2) throw k;
        else
```

```

        if (k== -2) throw 1.0;
        cout <<"\n End of function check ( )";
}
void main ( )
{
    try {
        cout <<"k==1\n";

        check(1);

        cout <<"k==2\n";
        check(2);

        cout <<"k== -2\n";
        check(-2);

        cout <<"k==3\n";
        check(3);
    }
    catch ( char g)
    {
        cout <<"Caught a character exception \n";
    }
    catch (int j)
    {
        cout <<"Caught a character exception \n";
    }
    catch (double s)
    {
        cout <<"Caught a double exception \n";
    }

    cout <<"\n End of main ( ) ";
}

```

## **OUTPUT**

**k==1**

**Caught a character exception**

**End of main ( ) Press any k**

**Explanation:** In the above program, `check ( )` function is defined. It is followed by exception, specifying statement. The `if` statements check the value of formal argument. According to the value, exception is thrown. In

function `main()`, the function `check()` is invoked with different values. The output of the program is above.

## 15.9 EXCEPTIONS IN CONSTRUCTORS AND DESTRUCTORS

Copy constructor is called in exception handling when an exception is thrown from the `try` block using `throw` statement. Copy constructor mechanism is applied to initialize the temporary object. In addition, destructors are also executed to destroy the object. If exception is thrown from constructor, destructors are called only completely constructed objects.

### 15.6 Write a program to use exception handling with constructor and destructor.

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

class number
{
    float x;
public :

    number (float);
    number( ) {};

    ~number ( )
    {
        cout<<"\n In destructor";
    }
    void operator ++ (int) // postfix notation
    { x++; }

    void operator -- ( ) // prefix notation
    {
        --x; }
    void show ( )
    {
        cout <<"\n x="<<x;
    }
};

number :: number ( float k)
{
    if (k==0)
        throw number( );
    else
```

```

        x=k;
    }

void main( )
{
    try
{

    number N(2.4);
    cout <<"\n Before incrimination: ";
    N.show( );
    cout <<"\n After incrimination: ";
    N++; // postfix increment
    N.show( );
    cout <<"\n After decrementation:" ;
    --N; // prefix decrement
    N.show( );
    number N1(0);
}

catch (number)
{
    cout <<"\n invalid number";
    exit(1);
}
}

```

## **OUTPUT**

**Before incrimination:**

**x=2.4**

**After incrimination:**

**x=3.4**

**After decrementation:**

**x=2.4**

**In destructor**

**In destructor**

**invalid number**

**Explanation:** In the above program, the operators ++ and -- are overloaded. The class `number( )` has two constructors and one destructor. The one argument constructor is used to initialize data member `x` with the value given by the user. The overloaded operators are used to increase and decrease the value of object. When the user specifies the zero value, an exception is thrown

from the constructor. The exception is caught by the `catch( )` statement. In function `main( )` two objects `N` and `N1` are declared. The exception is thrown when object `N1` is created. When exception is thrown control goes to `catch( )` block. The `catch( )` block terminates the program. The destructors are also invoked.

## 15.10 CONTROLLING UNCAUGHT EXCEPTIONS

C++ has the following functions to control uncaught exceptions.

### (1) THE TERMINATE( ) FUNCTION

In case exception handler is not defined when exception is thrown `terminate( )` function is invoked. Implicitly the `abort( )` function is invoked.

#### 15.7 Write a program to catch uncaught exceptions.

```
# include <iostream.h>
class one {};
class two {};

void main( )
{
    try
    {
        cout <<" An uncaught exception\n";
        throw two( );
    }
    catch (one)
    {
        cout <<" Exception for one ";
    }
}
```

**Explanation:** In the above program, an exception is thrown for class `two` using the statement `throw two( )`. The `catch( )` block is defined to handle exception for class `one` and it contains an argument of class `one` type. When an exception is thrown it does not find a match hence the program is terminated. For termination, `abort( )` function is called implicitly by the compiler.

### (2) THE SET\_TERMINATE( ) FUNCTION

The `set_terminate( )` function is used to transfer the control to specified error handling function. The `set_terminate( )` function requires only one argument that is nothing but a function name where control is transferred. It returns nothing. When no function is specified, the program is terminated by an implicit call to `abort( )` function.

### 15.8 Write a program to demonstrate the use of `set_terminate( )` function.

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

class one {};
class two {};

void skip()
{
    cout <<"\n function skip ( ) is invoked";
}

void main()
{
    set_terminate(skip);

    try
    {
        cout<<"\n Throwing exception : ";
        throw (two);
    }

    catch (one)
    {
        cout <<"\n An exception of one is caught";
    }
    // At this point function skip is invoked
}
```

#### OUTPUT

Throwing exception  
function skip ( ) is invoked

**Explanation:** This program is same as the last one. The only difference is that compiler does not find a matching `catch( )` statement for the exception thrown by the `throw` statement. The program is terminated. In this program at the beginning, a user-defined function `skip( )` is defined. The `skip( )` function is associated with the `set_terminate( )` function. When compiler does not find a matching exception instead of terminating program, it invokes the function associated with `set_terminate( )` function. Here, the exception is thrown for object of class two. The `catch( )` statement holds an object of class one. Hence, the thrown exception object does not match the `catch( )` object. Due to this mismatch, the function `skip( )` is invoked to handle such an uncertain situation.

## 15.11 EXCEPTION AND OPERATOR OVERLOADING

Exception handling can be used with operator-overloaded functions. The following program illustrates the same.

### 15.9 Write a program to throw exception from overloaded operator function.

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

class number
{
    int x;
public :
    number ( ) {};
    void operator -- ( );
    void show ( ) { cout <<"\n x=" <<x; }
    number ( int k) { x=k; }
};

void number :: operator -- ( ) // prefix notation
{
    if (x==0) throw number( );
    else --x;
}

void main ( )
{
    try
{
    number N(4);
    cout <<"\n Before decrementation: ";
```

```

N.show( );
while(1)
{
    cout << "\n After decrementation";
    --N;
    N.show();
}
}

catch (number)
{
    cout << "\n Reached to zero";
    exit(1);
}

}

```

## **OUTPUT**

**Before decrementation**

**x=4**

**After decrementation**

**x=3**

**After decrementation**

**x=2**

**After decrementation**

**x=1**

**After decrementation**

**x=0**

**After decrementation**

**Reached to zero**

**Explanation:** In this program, the operator -- is overloaded. This operator when used with class objects decreases the values of class members.

Using while( ) loop the value of object N is continuously decreased. The object N is initialized with four. The operator --( ) function checks the value of x (member of object N) and if value of x reaches to zero, an exception is thrown which is caught by the catch( ) statement.

## **15.12 EXCEPTION AND INHERITANCE**

In the last few examples, we learned how exception mechanism works with operator function and with constructors and destructors. The following program explains how exception handling can be done in inheritance.

**15.10 Write a program to throw an exception in derived class.**

```

# include <iostream.h>

class ABC
{
protected:
    char name[15];
    int age;
};

class abc : public ABC // public derivation
{
    float height;
    float weight;
public:
    void getdata( )
    {
        cout <<"\n Enter Name and Age : ";
        cin >>name>>age;

        if (age<=0)
        throw abc( );

        cout <<"\n Enter Height and Weight : ";
        cin >>height >>weight;
    }

    void show( )
    {
        cout <<"\n Name : "<<name <<"\n Age : "<<age<<" Years";
        cout <<"\n Height : "<<height <<" Feets"<<"\n Weight : " <<weight
        <<" Kg." ;
    }
};

void main( )
{
    try
    {

        abc x;
        x.getdata( ); // Reads data through keyboard
        x.show( ); // Displays data on the screen
    }
    catch (abc) { cout <<"\n Wrong age"; }
}

```

## OUTPUT

**Enter Name and Age: Amit 0**

**Wrong age**

**Explanation:** In the above program, the two classes ABC and abc are declared. The class ABC has two protected data members name and age. The class abc has two float data members height and weight with two-member functions getdata( ) and show( ). The class abc is derived from class ABC. The statement `class abc : public ABC` defines the derived class abc. In function `main( )`, x is an object of derived class abc. The object x invokes the member function `getdata( )` and `show( )`. This function reads and displays data respectively. In the function `getdata( )`, the `if` statement checks the age entered. If the age entered is zero or less than zero an exception is thrown. The `catch( )` block is executed and the message "WRONG AGE" is displayed.

### **15.13 CLASS TEMPLATE WITH EXCEPTION HANDLING**

The following program illustrates how exception handling can be implemented with class templates.

**15.11 Write a program to show exception handling with class template.**

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

class sq{};

template <class T>
class square
{
    T s;

public:
    square ( T x)
    {
        sizeof(x)==1 ? throw sq ( ) : s=x*x;
    }

    void show( )
    {
        cout <<"\n Square : "<<s;
    }
};
```

```

void main( )
{
    try
    {

        square <int> i(2);
        i.show( );
        square <char> c('C');
        c.show( );

    }

    catch (sq)
    {
        cout <<"\n Square of character cannot be calculated";
    }
}

```

## **OUTPUT**

**Square: 4**

**Square of character cannot be calculated**

**Explanation:** In the above program, the class square uses template type of argument. Using conditional operator, size of passed argument is checked. If it is equal to one an exception is thrown, otherwise square is calculated and stored in the class data member s. The show( ) function displays the square on the screen. When a character type data is entered, an exception is thrown because size of character is one and square cannot be calculated.

## **15.14 GUIDELINES FOR EXCEPTION HANDLING**

(1) It is not essential for throw statement to appear in the try block in order to throw an exception. The throw statement can be placed in any function, if the function is to be invoked through the try block.

|                                                                                                          |                                                             |
|----------------------------------------------------------------------------------------------------------|-------------------------------------------------------------|
| <pre> try {     statement;     show (variable)     statement; } catch (object) {     statement; } </pre> | <pre> show () {     statement;     throw (object); } </pre> |
|----------------------------------------------------------------------------------------------------------|-------------------------------------------------------------|

### **Fig. 15.3** throw statement out of try block

As shown in the [Figure 15.3](#) the throw statement is present in the `show()` function. The `show()` function is invoked inside the `try` block. Thus, exception handling can also be defined in the above style.

(2) When an exception not specified is thrown, it is known as unexpected exception.

(3) In case an exception is thrown before complete execution of constructor, destructor for that object will not be executed.

(4) As soon as an exception is thrown, the compiler searches nearby handlers (catch blocks). After finding a match, it will be executed.

(5) Overuse of exception handling increases the program size. So apply it whenever most necessary. Incorrect use of exception handling is not consistent and generates bugs in the program. Such bugs are hard to debug.

### **SUMMARY**

(1) There are a few languages that support exception handling feature. Without this feature, the programmer will have to detect bugs.

(2) The errors may be logical errors or syntactic mistakes (syntax mistakes). The logical errors remain in the program due to unsatisfactory understanding of the program. The syntax mistakes are due to lack of understanding of the programming language itself.

(3) ANSI C++ has built in language functions for trapping the errors and controlling the exceptions. All C++ compilers support this newly added facility.

(4) C++ provides a type-secure, integrated procedure for coping with the predictable but peculiar conditions that arise at run-time.

(5) The goal of exception handling is to create a routine that detects and sends an exceptional condition in order to execute suitable action.

(6) An exception is an object. It is sent from the part of the program where an error occurs to the part of the program which is going to control the error.

(7) C++ exception method provides three keywords, `try`, `throw`, and `catch`. The keyword `try` is used at the starting of exception. The entire exception statements are enclosed in the curly braces. It is known as `try` block.

(8) The `catch` block receives the exception send by the `throw` block in the `try` block.

(9) We can also define multiple `catch` blocks in `try` block, such a program also contains multiple `throw` statements based on certain conditions.

(10) It is also possible to define single or default `catch ( )` block from one or more exception of different types. In such a situation, a single `catch` block is used for catch exceptions thrown by the multiple `throw` statements.

(11) It is also possible to again pass the exception received to another exception handler i.e., an exception is thrown from `catch ( )` block and this is known as rethrowing of exception.

(12) The specified exceptions are used when we want to bind the function to throw only specified exceptions. Using a `throw` list condition can also do this.

### EXERCISES

**[A] Answer the following questions.**

- (1) What do you mean by exception handling?
- (2) How many different types of errors are encountered in a program?
- (3) Describe the role of keywords `try`, `catch`, and `throw` in exception handling?
- (4) What will happen if the programming language does not support the exception handling feature?
- (5) What do you mean by re-throwing of an exception?
- (6) Explain specifying exception with necessary steps.
- (7) How are multiple `catch` blocks defined?
- (8) What will happen if an exception is thrown for which no matching `catch ( )` block is defined?
- (9) Explain guidelines for exception handling.
- (10) Explain mechanism of exception handling.
- (11) Explain the use of function `set_terminate ( )`. How can we associate a function with this statement?

**[B] Answer the following by selecting the appropriate option.**

- (1) The statement catches the exception
  - (a) `catch`
  - (b) `try`
  - (c) `throw`
  - (d) `template`
- (2) In multiple `catch ( )` statement the number of `throw` statements are
  - (a) same as `catch ( )` statement
  - (b) twice than `catch ( )` statement
  - (c) only one `throw` statement
  - (d) none of the above
- (3) Exception is generated in

- (a) try block
  - (b) catch block
  - (c) throw block
  - (d) none of the above
- (4) In exception handling one of the following functions is implicitly invoked
- (a) abort( )
  - (b) exit( )
  - (c) assert( )
  - (d) none of the above
- (5) What will happen if throw( ) function has an empty exception and is placed after a function argument list
- (a) no exception will be thrown
  - (b) default exception will be invoked
  - (c) the nearest one exception is thrown
  - (d) unexpected actions by the program
- (6) If catch( ) statement is defined to catch exception for base class type object it can also catch all\_\_\_\_ derived class of the base class
- (a) errors
  - (b) parameters
  - (c) exceptions for objects
  - (d) none of the above

**[C] Attempt the following programs.**

- (1) Write a program to accept 10 integers in an array. Check all numbers in the array. When any negative number is found, throw an exception.
- (2) Write a program to accept a string. When any space is encountered, throw exception.
- (3) Write a program to explain the concept of multiple catch statements.
- (4) Write a program to use catch( ) statement.
- (5) Write a program to call a nested function. The function has an exception. Include essential exception handling statements.

# 16

## CHAPTER

### Working with Strings

C  
H  
A  
P  
T  
E  
R  
  
O  
U  
T  
L  
I  
N  
E

- [16.1 Introduction](#)
- [16.2 Moving from C String to C++ String](#)
- [16.3 Declaring and Initializing String Objects](#)
- [16.4 Relational Operators](#)
- [16.5 Handling String Objects](#)
- [16.6 String Attributes](#)
- [16.7 Accessing Elements of Strings](#)
- [16.8 Comparing and Exchanging](#)
- [16.9 Miscellaneous Functions](#)

## 16.1 INTRODUCTION

A string is nothing but a sequence of characters. String can contain small and capital letters, numbers and symbols. String is an array of character type. Each element of string occupies a byte in the memory. Every string is terminated by a null character. The last character of such a string is null ('\0') character and the compiler identifies the null character during execution of the program. In a few cases, the null character must be specified explicitly. The null character is represented by '\0', which is the last character of a string. It's ASCII and Hex values are zero. The string is stored in the memory as follows:

```
char country[6] = "INDIA" ; ; // Declaration of an array 'country'.
```

Here, text "INDIA" is assigned to array country[6]. The text is enclosed within double quotation mark.

| I  | N  | D  | I  | A  | \  |
|----|----|----|----|----|----|
| 73 | 78 | 68 | 73 | 65 | 00 |

Each character occupies a single byte in memory as shown above. At the end of the string, null character is inserted by the compiler. The first row shows elements of string and the second row shows their corresponding ASCII values. The compiler takes care of storing the ASCII numbers of characters in the memory. In addition, the ASCII value of NULL character is also stored in the memory.

The programmer can use the string in program for storing and controlling text. The text comprises of words, sentences, names etc. The various operations with strings such as copying, comparing, concatenation, or replacing requires lot of effort in 'C' programming. These strings are called as C-style string. In C using function declared in `string.h` header file, string manipulation is done. [Table 16.1](#) describes these functions.

**Table 16.1** String library functions

| Functions              | Description                                |
|------------------------|--------------------------------------------|
| <code>strlen( )</code> | Determines length of a string              |
| <code>strcpy( )</code> | Copies a string from source to destination |

|             |                                                                                                                  |
|-------------|------------------------------------------------------------------------------------------------------------------|
| strncpy( )  | Copies characters of a string to another string up to the specified length                                       |
| strcmp( )   | Compares characters of two strings ( <b>Function discriminates between small &amp; capital letters.</b> )        |
| stricmp( )  | Compares two strings ( <b>Function doesn't discriminate between small &amp; capital letters.</b> )               |
| strncmp( )  | Compares characters of two strings up to the specified length                                                    |
| strnicmp( ) | Compares characters of two strings up to the specified length. Ignores case                                      |
| strlwr( )   | Converts uppercase characters of a string to lowercase                                                           |
| strupr( )   | Converts lowercase characters of a string to uppercase                                                           |
| strdup( )   | Duplicates a string                                                                                              |
| strchr( )   | Determines first occurrence of a given character in a string                                                     |
| strrchr( )  | Determines last occurrence of a given character in a string                                                      |
| strstr( )   | Determines first occurrence of a given string in another string                                                  |
| strcat( )   | Appends source string to destination string                                                                      |
| strncat( )  | Appends source string to destination string up to specified length                                               |
| strrev( )   | Reversing all characters of a string                                                                             |
| strset( )   | Set all characters of string with a given argument or symbol                                                     |
| strnset( )  | Set specified numbers of characters of string with a given argument or symbol                                    |
| strspn( )   | Finds up to what length two strings are identical                                                                |
| strpbrk( )  | Searches the first occurrence of the character in a given string and then it string starting from that character |

Following program explains use of above functions.

## **16.1 Write a program to declare string (character array) . Read string through the keyboard and count the length of the string using string library function.**

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

int main( )
{
    char name[15];
    clrscr( );
    cout<<"\n Enter your name : ";
    cin>>name;
    cout <<"\n Length of name is : "<<strlen(name);
    return 0;
}
```

### **OUTPUT**

**Enter your name : Suraj**

**Length of name is : 5**

**Explanation:** In the above program, the string is declared using the statement `char name[15];` using the library function `strlen()` total number of characters entered is displayed.

## **16.2 Write a program to display reverse of entered string.**

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

int main( )
{
    char name[15];
    clrscr( );
    cout<<"\nEnter your name : ";
    cin>>name;
    cout <<"\n Reverse string is : "<<strrev(name);
    return 0;
}
```

### **OUTPUT**

**Enter your name : Akash**

**Reverse string is : hsakA**

**Explanation:** In the above program, a character array `name [ ]` is declared. String is entered through the keyboard. The `strrev( )` function displays reverse string.

### 16.3 Write a program to initialize a string using different formats.

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

int main( )
{
clrscr( );

char text[]="Welcome"; // Using double quote
char text1[]={‘W’,‘e’,‘l’,‘c’,‘o’,‘m’,‘e’,‘\0’}; // using single
quote
cout<<"\n First string : "<<text;
cout <<"\n Second string : "<<text1;
return 0;
}
```

#### OUTPUT

First string : Welcome

Second string : Welcome

**Explanation:** In this program, two methods of initialization of arrays are used. Here, both declaration and initialization are done in the same statement.

```
char text[]="Welcome";
```

In this statement array `text []` is declared and it is initialized with string “Welcome”. It is an easy way to initialize character array. It is not necessary to include null character. Here also, size of array is not mentioned. When declaration and initialization is done in the same statement, the compiler determines the size of the array. Hence, it is optional to mention the size of array in the subscript `[]` bracket.

```
char text1[]={‘W’,‘e’,‘l’,‘c’,‘o’,‘m’,‘e’,‘\0’}; //  
using single quote
```

A character array (string) can also be initialized as shown in the above statement. However, in this method each character must be included in the single quotation mark and must be separated by comma. The string should be terminated by null character. Hence, the second method is difficult as compared to the first. Here the programmer has to specify the null character at the end of string.

### 16.2 MOVING FROM C STRING TO C++ STRING

In the last few examples we observed that a string is nothing but a sequence of characters and it is declared as character array. However, manipulation of

string in the form of character array requires more effort. C uses library function defined in `string.h` to carry string manipulation.

To make the string manipulation easy the ANSI committee added a new class called `string`. It allows us to define objects of `string` type and they can be used as built in data type. The `string` class is considered as another container class and not a part of STL (Standard Template Library). The STL is described in the [next chapter](#). The programmer should include the `string` header file.

Instead of declaring character array an object of `string`, class is defined. Using member function of `string` class, string operations such as copying, comparison, concatenation etc. are carried out more easily as compared to C library functions.

The `string` class is very vast. It also contains several constructors, member functions, and operators. These constructors, member functions, and operators help us to perform the various operations with strings.

### 16.3 DECLARING AND INITIALIZING STRING OBJECTS

In C, we declare strings as given below:

```
char text[10];
```

whereas in C++ string is declared as an object. The string object declaration and initialization can be done at once using constructor in the `string` class. The constructors of the `string` class are described in [Table 16.2](#).

**Table 16.2** String constructors

| Constructors                                   | Meaning                                    |
|------------------------------------------------|--------------------------------------------|
| <code>String ( ) ;</code>                      | Produces an empty string                   |
| <code>String (const char *text);</code>        | Produces a string object from a null ended |
| <code>String (const string &amp; text);</code> | Produces a string object from other string |

We can declare and initialize string objects as follows:

Declaration of string objects

```
string text; // Using constructor without argument
```

```

string text ("C++");           // Using constructor with one argument

text1=text2                    // Assignment of two string objects

text = "c++"+ text1           // Concatenation of string objects

cin>> text                   // Reads string without space through the keyboard

getline(cin, text)            // Reads string with blank spaces

```

Two string objects can be concatenated using overloaded + operator. The overloaded += operator appends one string to the end of another string. The operators << and >> are overloaded operators and can be used for input and output operations.

|                     |                                        |
|---------------------|----------------------------------------|
| text1+=text         | is equivalent to text1=text1+text      |
| text1+="xyz"        | is equivalent to text1=text1+"xyz"     |
| cin >> text         | // Reads string without spaces         |
| cout << text        | // Displays the contents on the screen |
| getline (cin, text) | // Reads string with blank spaces      |

Here are some illustrations based on the above concepts.

#### **16.4 Write a program to declare string objects. Perform assignment and concatenation operations with the string objects.**

```

#include <iostream>
#include <string>

using namespace std;
int main( )
{
    string text;           // Vacant string object
    string text1(" C++"); // Constructor with one argument

```

```

string text2(" OOP");

cout <<"text1 : "<<text1 <<"\n";
cout <<"text2 : "<<text2 <<"\n";
text=text1;      // assignment operation
cout <<"text : "<<text<<"\n";
text=text1+" " + text2;
cout <<"Now text : "<<text;
    return 0;
}

```

## **OUTPUT**

**text1 : C++**

**text2 : OOP**

**text : C++**

**Now text : C++ OOP**

**Explanation:** In the above program, three string objects `text`, `text1`, and `text2` are defined. The object `text` is not initialized. The objects `text1` and `text2` are initialized with the strings “C++” and “OOP” respectively. The object `text` is initialized with the contents of object `text1` using ‘=’ operator as per the statement `text=text1`. Again joining of object `text1` and `text2` is done and resulting string is assigned to the object `text` as per the statement `text=text1+" " + text2`.

Table 16.3 describes various member functions of `string` class.

**Table 16.3** String manipulating functions

| <b>Function</b>          | <b>Use</b>                                               |
|--------------------------|----------------------------------------------------------|
| <code>append( )</code>   | Adds one string at the end of another string             |
| <code>assign( )</code>   | Assigns a specified part of string                       |
| <code>at( )</code>       | Access the characters located at given location          |
| <code>begin( )</code>    | Returns a reference to the beginning of a string         |
| <code>capacity( )</code> | Calculates the total elements that can be stored         |
| <code>compare( )</code>  | Compares two strings                                     |
| <code>empty( )</code>    | Returns false if the string is not empty, otherwise true |

|                          |                                                              |
|--------------------------|--------------------------------------------------------------|
| <code>end( )</code>      | Returns a reference to the termination of string             |
| <code>erase( )</code>    | Erases the specified character                               |
| <code>find( )</code>     | Finds the given sub string in the source string              |
| <code>insert( )</code>   | Inserts a character at a given location                      |
| <code>length( )</code>   | Calculates the total number of elements of string            |
| <code>max_size( )</code> | Calculates the maximum possible size of string in a given sy |
| <code>replace( )</code>  | Substitutes specified characters with the given string       |
| <code>resize( )</code>   | Modifies the size of the string as specified                 |
| <code>size( )</code>     | Provides the number of characters in the string              |
| <code>swap( )</code>     | Exchanges the given string with another string               |

The operators used with arithmetic or comparison operations can be used with string objects. Table 16.4 describes these operators.

**Table 16.4** String manipulating operators

| Operator           | Working                      |
|--------------------|------------------------------|
| <code>=</code>     | Assignment                   |
| <code>+</code>     | Joining two or more strings  |
| <code>+=</code>    | Concatenation and assignment |
| <code>==</code>    | Equality                     |
| <code>!=</code>    | Not equal to                 |
| <code>&lt;</code>  | Less than                    |
| <code>&lt;=</code> | Less than or equal           |
| <code>&gt;</code>  | Greater than                 |

|     |                                |
|-----|--------------------------------|
| >=  | Greater than or equal          |
| [ ] | Subscription (used with array) |
| <<  | Insertion operator             |
| >>  | Extraction operator            |

## 16.4 RELATIONAL OPERATORS

**Table 16.4** describes various relational operators. These operators can be used with string objects for assignment, comparison etc. The following program illustrates the use of relational operators with string objects.

### 16.5 Write a program to compare two strings using string objects and relational operators.

```
# include <iostream>
# include <string>
using namespace std;

int main( )
{
    string s1("OOP");
    string s2("OOP");
    if (s1==s2)
        cout <<"\n Both the strings are identical";
    else
        cout <<"\n Both the strings are different";
    return 0;
}
```

#### OUTPUT

**Both the strings are identical**

**Explanation:** In the above program, two string objects `s1` and `s2` are declared. Both the string objects are initialized with the string "OOP". The `if` statement checks whether the two strings are identical or different. Appropriate message will be displayed on comparison. Thus, in this program the two string objects contain the same string and hence, it displays the message "Both the strings are identical".

### 16.6 Write a program to compare two strings using standard function `compare()`.

```
# include <iostream>
# include <string>
```

```

using namespace std;
int main( )
{
    string s1("aaa");
    string s2("bbb");
    int d=s1.compare(s2);

    if (d==0)
        cout <<"\n Both the strings are identical";
    else if (d>0)
        cout <<"s1 is greater than s2";
    else
        cout <<"s2 is greater than s1";
    return 0;
}

```

## **OUTPUT**

**s2 is greater than s1**

**Explanation:** In the given program, two string objects s1 and s2 are declared and initialized with the strings "aaa" and "bbb". Both the string objects are compared using compare( ) function. The return value of function compare( ) is stored in the integer variable d. The function compare( ) returns zero if the two strings are same, otherwise it returns positive value. Using if..else condition appropriate messages are displayed.

## **16.5 HANDLING STRING OBJECTS**

The member functions insert( ), replace( ), erase( ) and append( ) are used to modify the string contents. The following program illustrates the use of these functions:

**insert( )**

This member function is used to insert specified strings into another string at a given location. It is used in the following form:

**s1.insert(3,s2);**

Here, s1 and s2 are string objects. The first argument is used as location number for calling string where the second string is to be inserted.

## **16.7 Write a program to insert one string into another string using insert( ) function.**

```

# include <iostream>
# include <string>

using namespace std;
int main( )

```

```

{
string s1("abchijk");
string s2("defg");

cout <<"\n s1= "<<s1;
cout <<"\n s2= "<<s2;
cout <<"\n after insertion";

s1.insert(3,s2);
cout <<"\n s1= "<<s1;
cout <<"\n s2= "<<s2;

return 0;
}

```

## **OUTPUT**

```

s1= abchijk
s2= defg
after insertion
s1= abcdefghijk
s2= defg

```

**Explanation:** In the above program, two string objects `s1` and `s2` are declared and initialized with strings “`abchijk`” and “`defg`”. The `insert( )` function inserts the string “`defg`” in the string “`abchijk`” at location 3. Now, the resulting string is “`abcdefghijk`”. In the statement `s1.insert(3, s2)`, the object `s1` invokes the member function `insert( )` and passes the argument 3 and `s2` explicitly.

`erase( )`

The `erase( )` member function is used to erase/ remove specified characters from specified location. It is used in the following form:

`s1.erase(3,7);`

Here `s1` and `s2` are string objects. The first argument is starting element number and second argument is last element number i.e., character elements from 3 to 7 are removed.

## **16.8 Write a program to remove specified characters from the string.**

```

# include <iostream>
# include <string>
using namespace std;

int main( )
{
string s1("abc12345defg");
cout <<"\n s1= "<<s1;
cout <<"\n after erase ( )";

```

```
s1.erase(3,5);
cout <<"\n s1= "<<s1;
return 0;
}
```

### OUTPUT

**s1= abc12345defg**  
**after erase ( )**  
**s1= abcdefg**

**Explanation:** In the above program, s1 is a string object declared and initialized with string "abc12345defg". The object s1 invokes the member function insert( ) with two integers 3 and 5. 3 indicates starting element number and 5 indicates number of characters to be erased.

The insert( ) function erases next five characters from 3<sup>rd</sup> character.

replace( )

This member function replaces given character in a string. It requires three arguments as per the following format:

```
s1.replace(2,5,s2);
```

Here s1 and s2 are string objects.

### 16.9 Write a program to replace a string with given strings.

```
# include <iostream>
# include <string>

using namespace std;

int main( )
{
    string s1("abcdefg");
    cout <<"\n s1= "<<s1;
    cout <<"\n after replace( )";
    s1.replace(1,3,"BCD");
    cout <<"\n s1= "<<s1;
    cout<<"\n";
    return 0;
}
```

### OUTPUT

**s1= abcdefg**  
**after replace( )**  
**s1= aBCDefg**

**Explanation:** In the above program, s1 is a string object initialized with "abcdefg". The object s1 invokes the member function replace( ) with three arguments. The first argument indicates the starting character

element, the second argument indicates the location of last character, and the third argument is a string that is to be replaced.

append( )

The above function is used to add a string at the end of another string. It is used in the following format:

s1.append(s2);

Here s1 and s2 are two objects. The contents of s2 are appended in the string s1.

### 16.10 Write a program to append one string at the end of another string

.Use append( ) function.

```
# include <iostream>
# include <string>

using namespace std;

int main( )
{
    string s1("abcdefg");
    string s2("hijklmn");

    cout <<"\n s1= "<<s1;
    cout <<"\n after append ( )";

    s1.append(s2);

    cout <<"\n s1= "<<s1;
    cout<<"\n";
    return 0;
}
```

#### OUTPUT

```
s1= abcdefg
after append ()
s1= abcdefghijklmn
```

**Explanation:** In the above program, two string objects s1 and s2 are declared and initialized with the strings “abcdefg” and “hijklmn”. The s1 object invokes the member function append( ) and s2 is passed as argument. The string s2 is added at the end of string s1.

## 16.6 STRING ATTRIBUTES

The various attributes of the string such as size, length, capacity etc. can be obtained using member functions. The size of the string indicates the total number of elements currently stored in the string. The capacity means the

total number of elements that can be stored in string. The maximum size means the largest valid size of the string supported by the system.

`s1.size( )`

The member function `s1.size( )` determines the size of the string object i.e., number of bytes occupied by the string object. It is used in the given format.

`s1.size( )`

Here `s1` is a string object and `s1.size( )` is a member function.

### **16.11 Write a program to display the size of the string object before and after initialization.**

```
# include <iostream>
# include <string>

using namespace std;
int main( )
{
    string s1;
    cout << "\nsize : " << s1.size( );
    s1="hello";
    cout << "\nNow size : " << s1.size( );
    return 0;
}
```

#### **OUTPUT**

**size : 0**

**Now size : 5**

**Explanation:** In the above program, `s1` is a string object. In the first `cout` statement `s1` invokes the member function `s1.size( )`, which returns the size of the string object `s1`. The `s1.size( )` object returns the size zero because the object `s1` is empty. The object `s1` is initialized with the string “hello.” In the second `cout` statement again the object `s1` invokes the member function `s1.size( )`. This time the `s1.size( )` member function returns the size 5. Thus, the `s1.size( )` function determines the size of the string object.

`s1.length( )`

The member function `s1.length( )` determines the length of the string object i.e. number of characters present in the string. It is used in the following format:

`s1.length( )`

Here `s1` is a string object and `s1.length( )` is a member function.

The member functions `s1.size( )` and `s1.length( )` provide the same result. Each character occupies one byte in the memory. The number of bytes and

total number of elements present in the string are always same, hence both these functions provide the same result.

### **16.12 Write a program to calculate the length of the string. Use member function length( ).**

```
# include <iostream>
# include <string>

using namespace std;

int main( )
{
string s1;

cout <<"\n Length : "<<s1.length( );
s1="hello";
cout <<"\n Now Length : "<<s1.length( );

return 0;
}
```

#### **OUTPUT**

**Length : 0**

**Now Length : 5**

**Explanation:** The above program is same as the previous one. Here, instead of size( ) function, length( ) member function is used. The length( ) member function displays the length of the string object before and after initialization.

capacity( )

The member function capacity( ) determines the capacity of the string object i.e. number of characters it can hold. It is used in the following format:  
s1.capacity( )

Here s1 is a string object and capacity( ) is a member function.

### **16.13 Write a program to display the capacity of the string object. Use member function capacity( ).**

```
# include <iostream>
# include <string>

using namespace std;
int main( )
{
string s1;

cout <<"\n Capacity : "<<s1.capacity( );
s1="hello";
```

```
cout <<"\n Capacity : "<<s1.capacity( );
s1="abcdefghijklmnopqrstuvwxyzabcdef";
cout <<"\n Capacity : "<<s1.capacity( );
return 0;
}
```

## OUTPUT

**Capacity : 0**  
**Capacity : 31**  
**Capacity : 63**

**Explanation:** In the above program, s1 is declared as a string object. The member function capacity( ) returns the capacity of the string object to hold the character elements. The maximum size of string in this system is 4294967293 and is obtained by the member function max\_size( ). In this program the capacity( ) function returns 31. We get different values in different situations. If the string object is empty the function returns 0. In case the string object contains string less than 32 characters, it returns 31. In this program, the string is initialized with 32 characters. The capacity( ) function returns the value 63 i.e. 32 + 31.

**max\_size( )**

The member function max\_size( ) determines the maximum size of the string object i.e. number of characters it can hold. It is used in the following format:

s1.max\_size( )

Here s1 is a string object and max\_size( ) is a member function.

## 16.14 Write a program to display the maximum size of the string object.

```
# include <iostream>
# include <string>

using namespace std;

int main( )
{
string s1;
cout <<"\n Maximum size : "<<s1.max_size( );
return 0;
}
```

## OUTPUT

**Maximum size : 4294967293**

**Explanation:** In the above program, s1 is declared as a string object.

The max\_size( ) function returns the maximum size of the string object i.e. 4294967293.

empty( )

The empty( ) function determines whether the string is empty or filled. If the string is empty, it returns 1, otherwise 0. It is used in the following format:  
s1.empty( )

Here s1 is a string object and empty( ) is a member function.

### 16.15 Write a program to determine whether the string object is initialized or not. Use empty( ) member function.

```
# include <iostream>
# include <string>

using namespace std;
int main( )
{
    string s1;

    cout <<"\n Empty : "<<(s1.empty ( )? "True" :"False");
    s1="abc";
    cout <<"\n Empty: "<<(s1.empty( ) ? "True" :"False");
    return 0;
}
```

#### OUTPUT

Empty : True

Empty : False

**Explanation:** In the above program, s1 is declared as a string object and it is not initialized. The member function empty( ) is called with conditional operator. It displays the message true(1) i.e. string is empty.

The string object s1 is initialized with string “abc” and again empty( ) function is invoked. This time it returns false( ) i.e. string is not empty.

## 16.7 ACCESSING ELEMENTS OF STRINGS

It is possible to access particular word or single character of string with the help of member functions of string class. The supporting functions are illustrated with suitable examples.

at( )

This function is used to access individual character. It requires one argument that indicates the element number. It is used in the given format.

s1.at(5);

Here s1 is a string object and 5 indicates 5th element that is to be accessed.

### 16.16 Write a program to display the string elements one by one. Use the member function at( ).

```
# include <iostream>
# include <string>

using namespace std;

int main( )
{
string s1("PROGRAMMING");

for (int j=0;j<s1.length( );j++)
cout<<s1.at(j);

return 0;
}
```

#### OUTPUT

#### PROGRAMMING

**Explanation:** In the above program, s1 is a string object initialized with the string “PROGRAMMING.” The for loop is used to represent successive character location. The at( ) function with one integer argument displays the characters. The variable j indicates the element number.

The statement cout<<s1.at(j) displays the character. We can also use the overloaded operator [ ] to display the string without the use of at( ) function. Thus, the statement would be cout<<s1. [j].

find( )

The find( ) member function finds the given sub-string in the main string. It is used in the following format:

s1.find(s2);

Here s1 and s2 are string objects. The find( ) function searches the sub-string s2 in the main string s1.

### 16.17 Write a program to find sub-string from the source string.

```
# include <iostream>
# include <string>

using namespace std;

int main( )
{
string s1("Bangalore is the capital of Karnataka");
```

```
int x=s1.find("capital");

cout<<"Capital is found at : "<<x;

return 0;
}
```

## OUTPUT

**Capital is found at : 17**

**Explanation:** In the above program, `s1` is a string object and it is initialized with the string. The `find( )` member function finds the given string in the source string. Here, in this program the `find( )` function searches the word “capital” in the string `s1`. The `find( )` function returns element number of the previous element from where the sub-string starts.

`substr( )`

The member function `substr( )` is used to find sub-string in the main string. It requires two-integer arguments. The first argument indicates the starting element of the string and the second argument indicates the last argument of the string. It is used in the following format:

`s1.substr(s, e);`

Here `s1` is a string object. The variables `s` and `e` are integer variables that indicate the starting and ending element number of the sub-string.

## 16.18 Write a program to retrieve the sub-string from the main string.

```
# include <iostream>
# include <string>

using namespace std;
int main( )
{
    string s1 ("C plus plus");

    cout <<s1.substr(2,4);
    cout<<"\n";

    return 0;
}
```

## OUTPUT

**Plus**

**Explanation:** In the above program, `s1` is declared as a string object. It is initialized with the string “C plus plus.” The member function `substr( )` requires two integer arguments, that indicate the starting and ending element of the sub-string. The sub-string is displayed on the screen.

`find_first_of( )`

This member function is used to find the first occurrence of the given character(s). It is used in the following format:

```
s1.find_first_of ('p');
```

Here s1 is a string object and 'p' is a character whose first occurrence is to be found.

### **16.19 Write a program to find the first occurrence of the given character.**

**Use member function** `find_first_of( )`.

```
# include <iostream>
# include <string>

using namespace std;

int main( )
{
    string s1 ("C plus plus");

    cout <<s1.find_first_of('p');
    cout<<"\n";
    return 0;
}
```

#### **OUTPUT**

**2**

**Explanation:** In the above program, s1 is declared as string object and it is initialized with the string "C plus plus". The member function `find_first_of( )` searches the character 'p' and when finds it, returns the element number of the previous character.

`find_last_of( )`

This member function is used to find the last occurrence of the given character(s). It is used in the format given next.

```
s1.find_last_of('p');
```

Here s1 is a string object and 'p' is a character whose last occurrence is to be found.

### **16.20 Write a program to find the last occurrence of the given character.**

**Use member function** `find_last_of( )`.

```
# include <iostream>
# include <string>

using namespace std;

int main( )
{
    string s1 ("C plus plus");
```

```
cout << s1.find_last_of('p');
cout << "\n";
    return 0;
}
```

## OUTPUT

7

**Explanation:** In the above program, `s1` is declared as string object and it is initialized with the string "C plus plus". The member function `find_last_of()` searches the character 'p' and when its last occurrence is found, returns the element number of the previous character.

## 16.8 COMPARING AND EXCHANGING

The string class contains functions, which allows the programmer to compare and exchange the strings. The related functions are illustrated below with suitable examples:

`compare( )`

The member function `compare( )` is used to compare two string or substrings. It returns 0 when the two strings are same, otherwise returns a non-zero value. It is used in the following formats:

`s1.compare(s2);`

Here `s1` and `s2` are two string objects.

`s1.compare(0,4,s2,0,4);`

Here `s1` and `s2` are two string objects. The integer number indicates the substring of both the strings that are to be compared.

**16.21 Write a program to compare two strings. Use `compare( )` function.**

```
# include <iostream>
# include <string>

using namespace std;

int main( )
{
    string s1 ("Take");
    string s2 ("Taken");

    int d=s1.compare(s2);
    cout << "\n d= "<<d;

    d=s1.compare (0,4,s2,0,4);
    cout << "\n d= "<<d;
    return 0;
}
```

```
}
```

**OUTPUT**

```
d= -1
d= 0
```

**Explanation:** In the above program, `s1` and `s2` are two string objects initialized with the strings “Take” and “Taken” respectively. The first `compare( )` statement checks the two string objects. It returns `-1` i.e. the strings are not identical. In the second `compare( )` statement, element numbers of both the strings that are to be compared are sent. Thus, passing element number sub-strings can be compared. In the above program, `0` to `4` elements of both the strings are compared. The function `compare( )` returns `0` i.e. the two sub-strings are identical.

```
swap( )
```

The `swap( )` member function is used to exchange the contents of two string objects. It is used in the following format:

```
s1.swap(s2);
```

Here `s1` and `s2` are two string objects. The contents of `s1` are assigned to `s2` and vice versa.

**16.22 Write a program to exchange the contents of two string objects.  
Use the member function `swap( )`.**

```
# include <iostream>
# include <string>

using namespace std;

int main( )
{
    string s1 ("Take");
    string s2 ("Took");

    cout <<"\n s1= "<<s1;
    cout <<"\n s2= "<<s2;

    cout <<"\n After swap ( ) ";

    s1.swap(s2);
    cout <<"\n s1= "<<s1;
    cout <<"\n s2= "<<s2;
    return 0;
}
```

**OUTPUT**

```
s1= Take
```

```
s2= Took  
After swap ( )  
s1= Took  
s2= Take
```

**Explanation:** In the above program, string objects `s1` and `s2` are declared and initialized with "Take" and "Took". The `swap( )` function exchanges the contents of `s1` and `s2` i.e. contents of `s1` is assigned to `s2` and vice versa. The output of the program is given above.

## 16.9 MISCELLANEOUS FUNCTIONS

`assign( )`

This function is used to assign a string wholly/ partly to other string object. It is used in the following format:

```
s2.assign(s1)
```

Here `s2` and `s1` are two string objects. The contents of string `s1` are assigned to `s2`.

```
s2.assign(s1,0,5);
```

In the above format, element from 0 to 5 are assigned to object `s2`.

### 16.23 Write a program to assign a sub-string from main string to another string object.

```
# include <iostream>  
# include <string> using namespace std;  
int main( )  
{  
    string s1("c plus plus");  
    string s2;  
    int x=0;  
    s2.assign(s1,0,6);  
    cout <<s2;  
    return 0;  
}
```

#### OUTPUT

C plus

**Explanation:** In the above program, `s1` and `s2` are two string objects. The object `s1` is initialized with the string "c plus plus". The object `s2` invokes the `assign( )` function with the integer argument 0,6, that indicates the sub-string. The sub-string is assigned to object `s2` i.e. "C plus".

```
begin( )
```

This member function returns the reference of the first character of the string. It is used in the following format:

```
x=s1.begin( );
```

Here `x` is a character pointer and `s1` is a string object.

### 16.24 Write a program to find starting character of a given string.

```
# include <iostream>
# include <string>
using namespace std;
int main ( )
{
    string s1("C plus plus");
    char *x;
    x=s1.begin( );
    cout <<*x;
    return 0;
}
```

#### OUTPUT

```
C
```

**Explanation:** In the above program, `s1` is a string object and it is initialized with string "C plus plus". The `x` is a character pointer. The object `s1` invokes the function `begin( )` and returns the starting address of the string `s1` to character pointer `x`. The contents of `x` printed are 'C' i.e. the first character of the string.

#### SUMMARY

(1) A string is nothing but a sequence of characters. They are declared as character arrays. Each element of string occupies a byte in the memory. Every string is terminated by a null character.

(2) To make the string manipulation easy the ANSI committee added a new class called `string`. It allows us to define objects of string type and they can be used as built-in data type. The string class is considered as another container class and not a part of STL (Standard Template Library).

(3) Two string objects can be concatenated using overloaded `+` operator. The overloaded `+=` operator appends one string to the end of another string. The operators `<<` and `>>` are overloaded operators and can be used for input and output operations.

(4) Table 16.4 describes various relational operators. These operators can be used with string objects for assignment, comparison etc.

(5) The member functions `insert( )`, `replace( )`, `erase( )` and `append( )` are used to modify the string contents.

(6) The various attributes of the strings such as size, length, capacity etc. can be obtained using member functions. The size of the string indicates the total number of elements currently stored in the string. The capacity means the total number of elements that can be stored in the string. The maximum size means the largest valid size of the string supported by the system.

(7) It is possible to access particular word or single character of string with the help of member functions of string class such as `at( )`, `find( )`, `substr( )`, `find_first_of( )` and `find_last_of( )`.

(8) The string class contains functions which allows the programmer to compare and exchange the strings. These functions are `swap( )` and `compare( )`.

### EXERCISES

**[A] Answer the following questions.**

- (1) What do you mean by string?
- (2) What is the difference between "a" and 'a'?
- (3) What is a string object?
- (4) What is the difference between string object and character pointer?
- (5) List any five string functions with their uses. Explain their syntaxes.

**[B] Answer the following by selecting the appropriate option.**

- (1) A and B are two string objects. `A="abc"` and `B="xyz"`. `A=A+B` will produce
  - (a) "abcxyz"
  - (b) "abc"
  - (c) "xyzabc"
  - (d) none of the above
- (2) The statement `string a("abc")`
  - (a) initializes string object
  - (b) replaces the string
  - (c) appends the string
  - (d) none of the above
- (3) A string is initialized using
  - (a) default constructor
  - (b) explicit constructor
  - (c) assignment operator
  - (d) both (a) and (c)
- (4) We can access string characters using
  - (a) `at( )` function
  - (b) subscript operator [ ]

- (c) ( ) operator
- (d) none of the above

**[C] Attempt the following programs.**

- (1) Write a program to read three strings "C," "PLUS" and "PLUS." Concatenate them in a single string.
- (2) Write a program to create an array of string objects. Read at least 10 names. Display the names alphabetically.
- (3) Write a program to read a string. Count the number of vowels and spaces in the string.
- (4) Write a program to read a string. Add the same string in the reverse order to the end of the same string.
- (5) Write a program to read a string. Remove all duplicate letters from the string.
- (6) Write a program to read a string. Change the first letter of every word capital.
- (7) Write a program to display the reverse string of the entered string.
- (8) Write a program to exchange the contents of two string objects. Use the member function `swap()`.
- (9) Write a program to display the string elements one by one. Use the member function `at()`.
- (10) Write a program to determine whether the string object is initialized or not. Use `empty()` member function.
- (11) Write a program to declare string objects. Perform assignment and concatenation with the string objects.

# 17

## CHAPTER

# Overview of Standard Template Library (STL)

CHAPTER  
OUTLINE

- [17.1 Introduction to STL](#)
- [17.2 STL Programming Model](#)
- [17.3 Containers](#)
- [17.4 Sequence Containers](#)
- [17.5 Associative Containers](#)
- [17.6 Algorithms](#)
- [17.7 Iterators](#)
- [17.8 Vector](#)
- [17.9 List](#)

—• [17.10 Maps](#)

—• [17.11 Function Objects](#)

## 17.1 INTRODUCTION TO STL

We have studied templates, which allow us to do generic programming. Generic functions or classes support all data types. The **Standard Template Library** is an advanced application of templates. It contains several in-built functions and operators that help the programmer to develop complex programs. The programmer only needs to include appropriate header file to use the function or operator from file like library functions. STL library helps to write programs free from bugs. In case, any bug is present, it is easy to debug. For example, if a programmer wants to create link list, he/she may require writing a program that may be of 40 to 50 lines. However, using STL functions (list algorithm), it can be done in a few minutes.

The *Standard Template Library (STL)* is a new feature of C++ language. All-famous compiler vendors provide the STL as a feature of their compilers. The STL is vast and heterogeneous collection of reusable container classes. It consists of vectors, lists, queues, and stacks. The STL is portable with various operating systems.

**Meng Lee and Alexander Stepanov** of *Hewlett-Packard* introduced the STL and it was accepted in 1994 as an addition to the customary C++ library. The STL provides well-coded and compiled data structures and functions that are helpful in generic programming. STL is reusable. STL topic is very vast, hence, its complete description is beyond the scope of this book. Only major features are explained. STL contents are defined in the namespace `std`. It is essential to write the statement using namespace `std` at the beginning of the program. In the [next chapter](#) namespaces are explained.

## 17.2 STL PROGRAMMING MODEL

STL is divided into three parts namely containers, algorithms, and iterators. All these three parts can be used for different programming problems. All these parts are closely associated with each other.

### (1) CONTAINERS

A container is an object that contains data or other objects. The standard C++ library has a number of container classes that allow a programmer to perform common tasks. These containers support generic programming and can be used for handling data of different data types. All STL container classes are declared in namespace std. There are two types of containers sequence, container and associative container as shown in [Figure 17.1](#). Sequence containers are created to allow sequential and random access to members. The associative container allows access to their elements through key.

## (2) ALGORITHM

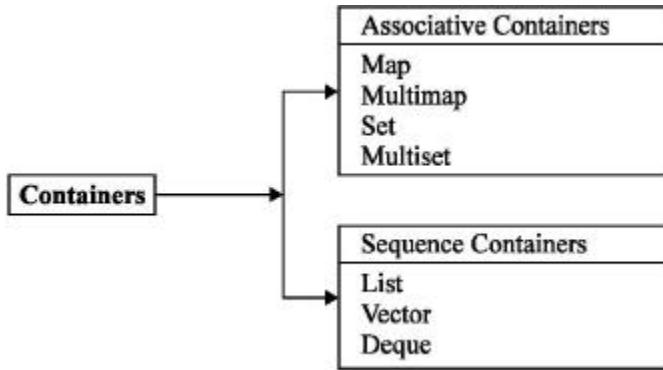
An algorithm is a technique that is useful to handle the data stored in containers. The STL comprises approximately 60 standard algorithms that give support to perform frequent and primary operations such as copying, finding, sorting, merging, and initializing. The standard algorithms are defined in the header file <algorithm>.

## (3) ITERATORS

An iterator is an object. It exactly behaves like pointers. It indicates or points to data element in a container. It is utilized to move between the data items of containers. Iterators can be incremented or decremented like pointers. Iterators link algorithms with containers and handle data stored in the containers.

### 17.3 CONTAINERS

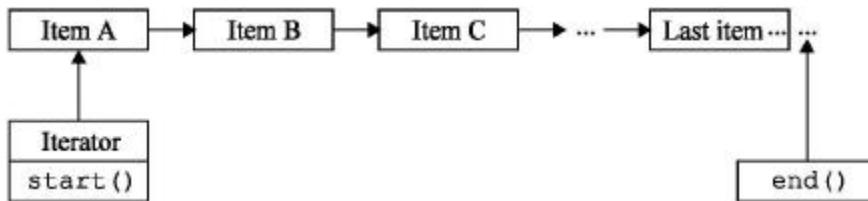
The standard C++ library has a sequence of container classes. The container classes are robust and support C++ programmers to control common programming operations. There are three types of STL container classes. They are sequence, associative and derived containers. Sequence containers are developed to allow sequential and random access to all the members. Associative containers are expanded to get their elements by values. Derived containers such as stack, queue, and priority queue can be created from various sequence containers. The STL has ten containers. They are divided into three types as shown in [Figure 17.1](#).



**Fig. 17.1** Classification of containers

## 17.4 SEQUENCE CONTAINERS

The STL sequence containers allow controlled sequential access to elements. Sequence containers hold data elements in a linear series as shown in [Figure 17.2](#). Every element is associated by its location within the series. All these elements enlarge themselves to let insertion of element. These elements also allow various operations like copying, finding etc.



**Fig. 17.2** Data items in sequence container

The STL has three types of sequence containers:

- (i) vector
- (ii) list
- (iii) deque

Iterators are used to get the elements in all these containers. All these containers have different speed.

### (1) VECTOR

We know that an array is used to store similar data elements. Elements of arrays are stored in successive memory locations and can be accessed, in order, from element number 0 onwards. The vector class exactly acts like an array. The vector container class is secured and efficient than arrays.

A vector container class is accelerated to allow quick access to its elements in sequence. It is defined in the header file `<vector>`. A vector is able to enlarge

itself. For example a vector declared for 5 elements, if assigned 6 elements, then the vector automatically grows its size so that it can hold the 6th element. The vector class is declared next.

### **Declaration of Vector class**

```
template<class TE>, class B =allocator<TE>class vector
```

Here the class TE is the type of element in vector and class B is an allocator class. M\_allocators are responsible for allocating memory and releasing them. By default, the new( ) and delete( ) operators are used for allocating memory and releasing. The default constructor of class TE is invoked to produce a new element. This allows entry of another parameter in order to define a default constructor for user-defined classes. The vector that holds integers and floats is declared below:

### **Vector declaration for int and float data**

```
vector<int> vi      // for integer elements  
vector <float> vf // for float elements
```

To declare a vector of 15 items a constructor can be declared as shown below:

```
vector<item> sale(15);
```

The compiler allocates memory to 15 elements. Here, the default constructor item :: item( ) is used.

## **(2) LIST**

The list container allows usual deletion and insertion of items. A list is also a sequence that can be accessed bi-directionally. It is defined in header file <list>. It acts as double linked list. Every node is connected to both back and front node in the list. The iterator is used to transverse the link. The list class supports all the member functions of vector class. The elements in the linked list are accessed using pointers. The list container has a technique known as iterator, which is used to access elements of list container. An iterator is same as pointer. The iterator can be referenced like pointers to access elements.

## **(3) DEQUE**

Deque is similar to a multiple ended vector. It has ability like vector class to perform sequential read and write operations. The deque class allows improved front-end and back-end operations. Deques are perfectly

appropriate for the operation that contains insertion and deletion at one or both ends and where sequential access is essential.

## 17.5 ASSOCIATIVE CONTAINERS

The associative container allows fast access to the objects in the container. These containers are useful for huge dynamic tables in which we can search an element randomly and sequentially. These containers use tree like structures of elements instead of linked lists. These structures allow quick random update and retrieval operations. Associative containers are non-sequential and allow direct access to elements. Associative containers are divided into four categories:

- (1) Set
- (2) Multiset
- (3) Map
- (4) Multimap

All these above listed containers hold data elements in a structure called *tree*. The tree provides quick finding, deletion, and insertion. These containers perform very slowly in random access operation and are inappropriate for sorting operation.

Container set and multiset store data elements. They also allow various operations for managing them. The variable name is used as key name. A set might contain objects of `player` class. It can be sorted in ascending order using the names as keys. The multiset may have multiple set of elements i.e. it permits duplicate data elements. The set does not allow duplicate elements. Containers, map and multimap, store both key name and value. The values are associated with key names. The values can be handled using key names. The values are also called as *mapped values*. Multimap allows the use of various keys. Map container permits single key.

## 17.6 ALGORITHMS

Algorithms are independent template functions. They are not members or friend functions. They allow the programmer in manipulating data stored in various containers. Algorithms carry out operations on containers by dereferencing iterators. Each container has its functions for performing common operations. Algorithm is nothing but a function template with arguments of iterator type. Algorithms receive iterators as arguments. The iterators inform the algorithm regarding objects of container on which operation is to be carried out. In addition, STL contains approximately sixty

standard algorithms. These functions allow quick operations in complicated and large operations. Including the <algorithm> header file we can use these functions. The programmer reuses all these container classes.

**Table 17.1** Non-mutating sequence operation

| Operators        | Use                                                     |
|------------------|---------------------------------------------------------|
| search_n( )      | Searches a sequence of a given number of same elements. |
| find_if( )       | Searches first equivalent of a predicate in a sequence. |
| search( )        | Searches sub-sequence in other sequence.                |
| find_first_of( ) | Searches a value from one sequence to another.          |
| find( )          | Searches first presence of a value in a sequence.       |
| find_end( )      | Searches last occurrence of a value in a sequence.      |
| adjacent_find( ) | Searches contiguous pair of objects that are identical. |
| mismatch( )      | Searches elements for which two sequences vary.         |
| count( )         | Calculates presence of a value in a sequence.           |
| count_if( )      | Calculates elements that are similar to a predicate.    |
| equal( )         | True if two series' are identical.                      |
| for_each( )      | Performs an operation with each element.                |

Non-mutating sequence algorithms as shown in [Table 17.1](#) allow operations, which do not alter the elements in a sequence. For example, the operators `for_each( )`, `find( )` `search( )`, `count( )` etc. The program given next explains how to use `for_each( )` algorithm.

### 17.1 Write a program to demonstrate use of operator `for_each( )`.

```
# include <iostream>
# include <vector>
# include <algorithm>
using namespace std;

template <class S>
class show
```

```

{
public:
    void operator( ) ( const S& s)
    {
        cout<<s;
    }

};

int main( )
{
    show<int> showvalue;
    vector<int> vi(4);

    for ( int j=0;j<4;++j)
        vi[j]=j;

    cout<<" for_each( ) \n";

    for_each(vi.begin( ),vi.end( ),showvalue);
    cout<<"\n";
    return 0;
}

```

## OUTPUT

**0123**

**Explanation:** The operator( ) is overloaded in the class show.

The showvalue is an object of the class show. A vector vi is created which can hold four elements. Using first for loop, 0 to 3 numbers are inserted in the vector. While reading numbers from vector, instead of using for loop the operator for\_each( ) is used. The first argument of for\_each( ) operator (vi.begin( )) gives the reference of first element. The second argument gives the reference of last element of the vector and the third argument (showvalue) is a function object. Thus, using for\_each( ) operator members of vector can be accessed.

**17.2 Write a program to demonstrate use of fill( ) algorithm.**

```

# include <iostream>
# include <vector>
# include <algorithm>

using namespace std;

template <class S>
class show
{
public:
    void operator( ) ( const S& s)
    {
        cout<<s;
    }
};

int main( )
{
    show<int> showvalue;
    vector<int> vi(8);

    fill(vi.begin( ),vi.begin( ) + 2,5);
    fill(vi.begin( )+2,vi.begin( ) + 4,6);
    fill(vi.begin( )+4,vi.end( ),7);

    cout<<" for_each( ) \n";
    for_each(vi.begin( ),vi.end( ),showvalue);
    cout<<"\n";
    return 0;
}

```

## **OUTPUT**

**55667777**

**Explanation:** This program is same as the previous one. Here, the algorithm `fill( )` is used. The `fill( )` is a mutating algorithm because it changes the elements of the vector.

The following statements are used to fill the elements in the vector:

`fill(vi.begin( ),vi.begin( ) + 2, 5);`

The above statement fills first 2 elements with value 5. The function `begin( )` gives beginning reference and `begin( ) + 2` gives reference for the second.

`fill(vi.begin( )+2,vi.begin( ) + 4, 6);`

The above statement fills value 6 at third and fourth location of the vector.  
Here also, `begin( )` function is used.

```
fill(vi.begin( )+4, vi.end( ), 7);
```

The above statement fills value 7 from location number 5 to end.

Table 17.2 describes algorithm operations related to sorting.

**Table 17.2** Sorting operations

| Operators                               | Use                                                       |
|-----------------------------------------|-----------------------------------------------------------|
| <code>binary_search( )</code>           | Performs a binary search on an indexed sequence           |
| <code>equal( )</code>                   | Searches whether two sequences are equal                  |
| <code>equal_range( )</code>             | Searches a sub-range of elements with a specified value   |
| <code>includes( )</code>                | Searches if a sequence is a sub-sequence of another       |
| <code>inplace_merge( )</code>           | Combines two successive sorted sequences                  |
| <code>lexicographical_compare( )</code> | Compares in ascending order one sequence with other       |
| <code>lower_bound( )</code>             | Searches the first occurrence of a given value            |
| <code>make_heap( )</code>               | Makes a heap from a sequence                              |
| <code>max( )</code>                     | Returns greatest of two values                            |
| <code>max_element( )</code>             | Returns the highest elements inside a sequence            |
| <code>merge( )</code>                   | Combines two sorted sequences                             |
| <code>min( )</code>                     | Returns smallest of two values                            |
| <code>min_element( )</code>             | Returns the smallest number of a sequence                 |
| <code>mismatch( )</code>                | Searches the mismatch among the elements of two sequences |
| <code>nth_elements( )</code>            | Places given elements in its appropriate places           |
| <code>partial_sort( )</code>            | Sorts a portion of a sequence                             |
| <code>partial_sort_copy( )</code>       | Sorts a portion of a sequence and copies                  |
| <code>pop_heap( )</code>                | Erases the uppermost elements                             |

|                                          |                                                              |
|------------------------------------------|--------------------------------------------------------------|
| <code>push_heap( )</code>                | Appends or adds an element to heap                           |
| <code>set_difference( )</code>           | Builds a sequence that is the variance of two ordered sets   |
| <code>set_intersection( )</code>         | Makes a sequence that holds the intersection of ordered sets |
| <code>set_symmetric_difference( )</code> | Yields a set that is the symmetric variance between two sets |
| <code>set_union( )</code>                | Yields sorted union of two ordered sets                      |
| <code>sort( )</code>                     | Sorts the sequence container                                 |
| <code>sort_heap( )</code>                | Sorts a heap                                                 |
| <code>stable_partition( )</code>         | Puts elements matching a predicate by first matching order   |
| <code>stable_sort( )</code>              | Sorts arranging order of similar elements                    |
| <code>upper_bound( )</code>              | Searches the last occurrence of a given value                |

Table 17.3 describes mutating sequence operations.

**Table 17.3** Mutating sequence operations

| <b>Operators</b>              | <b>Use</b>                                                   |
|-------------------------------|--------------------------------------------------------------|
| <code>swap_ranges( )</code>   | Exchanges two sequences                                      |
| <code>generate( )</code>      | Substitutes all elements with the result of an operation     |
| <code>copy_backward( )</code> | Duplicates (copies) a sequence from the ends                 |
| <code>fill( )</code>          | Fills up a series with a given value                         |
| <code>reverse( )</code>       | Opposites (reverses) the order of elements                   |
| <code>fill_n( )</code>        | Fills up first n elements with a given value                 |
| <code>copy( )</code>          | Duplicates (copies) a sequence                               |
| <code>unique( )</code>        | Erases similar contiguous elements                           |
| <code>generate_n( )</code>    | Substitutes first n elements with the result of an operation |

|                                 |                                                                                    |
|---------------------------------|------------------------------------------------------------------------------------|
| <code>iter_swap( )</code>       | Exchanges elements pointed to by iterator                                          |
| <code>random_shuffle( )</code>  | Inserts elements in random order                                                   |
| <code>remove( )</code>          | Erases elements of a given value                                                   |
| <code>replace( )</code>         | Substitutes elements with a specified value                                        |
| <code>replace_if( )</code>      | Substitutes elements matching a predicate                                          |
| <code>remove_copy_if( )</code>  | Duplicates (copies) a sequence subsequently removing elements matching a predicate |
| <code>unique_copy( )</code>     | Copies after removing similar contiguous elements                                  |
| <code>rotate( )</code>          | Rotates (turns) elements                                                           |
| <code>remove_if( )</code>       | Erases elements matching a predicate                                               |
| <code>remove_copy( )</code>     | Duplicates (copies) a sequence subsequently removing a given value                 |
| <code>replace_copy( )</code>    | Duplicates (copies) a sequence substituting elements with a specified value        |
| <code>transform( )</code>       | Applies an action to all elements                                                  |
| <code>replace_copy_if( )</code> | Duplicates (copies) a sequence substituting elements matching a predicate          |
| <code>rotate_copy( )</code>     | Issues (copies) a sequence into a rotated                                          |
| <code>swap( )</code>            | Exchanges two elements                                                             |
| <code>reverse_copy( )</code>    | Copies a sequence into opposite direction                                          |

Table 17.4 describes numeric operations.

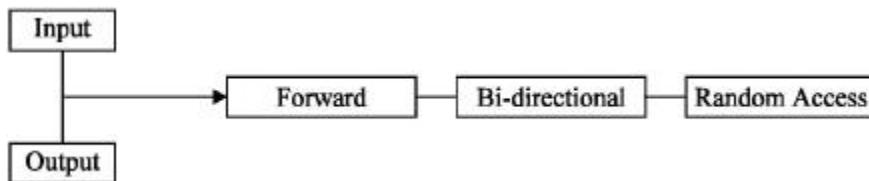
**Table 17.4** Numeric operations

| Operators                           | Use                                                      |
|-------------------------------------|----------------------------------------------------------|
| <code>inner_product( )</code>       | Collects the details of operations on a pair of sequence |
| <code>adjacent_difference( )</code> | Creates a sequence from another sequence                 |
| <code>accumulate( )</code>          | Collects the return values of operations on a sequence   |

|                             |                                                                    |
|-----------------------------|--------------------------------------------------------------------|
| <code>partial_sum( )</code> | Creates a sequence by applying an operation on a pair of elements. |
|-----------------------------|--------------------------------------------------------------------|

## 17.7 ITERATORS

Iterators act as pointers. They are used for accessing contents of container classes. They allow movement from one element to another element. It is called iterating. Each container type supports one kind of iterator according to its necessity. The types are input, output, forward, bi-directional and random access. The hierarchy of iterators is shown in [Figure 17.3](#) and described in [Table 17.5](#) and [17.6](#).



**Fig. 17.3** Iterator hierarchy

**Table 17.5** Operation of container classes

| Iterator       | Access mode | Way of action           | I/O ability        |
|----------------|-------------|-------------------------|--------------------|
| Forward        | Linear      | Can move forward only   | Can write and read |
| Bi-directional | Linear      | Can move front and back | Can write and read |
| Random         | Random      | Can move front and back | Can write and read |
| Input          | Linear      | Can move forward only   | Can read only      |
| Output         | Linear      | Can move forward only   | Can write only     |

**Table 17.6** Iterator status

| Iterator status | Meaning                                                                     |
|-----------------|-----------------------------------------------------------------------------|
| Singular        | The value of iterator is not dereferenced in any container.                 |
| Past-the-end    | The iterator addresses to the object past the last object in the container. |
| Dereferenceable | The iterator addresses the legal objects in the container.                  |

Iterators are used by algorithms to carry operations. Iterators are smart pointers. They are allowed to contain values that indicate one of the statuses of iterators as described in [Table 17.6](#). Each iterator type supports all the

attributes listed in [Table 17.6](#) and shown in [Figure 17.3](#). Iterators can be incremented, decremented, and their limits are up to the capacity of containers. Containers have member functions which return iterators.

## 17.8 VECTOR

The vector is a more useful container. Like arrays, it also stores elements in neighbouring memory locations. Each element can be directly accessed using the operator `[]`. A vector is able to enlarge its size if extra element is assigned to it. Class vector has various constructors that can be used to create vector objects. The function of vector class is shown in [Table 17.7](#).

**Table 17.7** Functions of vector class

| Function                  | Use                                                 |
|---------------------------|-----------------------------------------------------|
| <code>swap( )</code>      | Swaps the elements in the two given vectors         |
| <code>size( )</code>      | Provides the number of elements                     |
| <code>resize( )</code>    | Changes the size of the vector                      |
| <code>push_back( )</code> | Appends an element to the end                       |
| <code>pop_back( )</code>  | Erases the last element                             |
| <code>insert( )</code>    | Inserts items (elements) in the vector              |
| <code>erase( )</code>     | Erases given elements                               |
| <code>end( )</code>       | Provides reference to the end of the vector         |
| <code>empty( )</code>     | Determines whether the vector is empty              |
| <code>clear( )</code>     | Erases entire elements from the vector              |
| <code>capacity( )</code>  | Provides the present capacity (limit) of the vector |
| <code>begin( )</code>     | Provides reference to the starting element          |
| <code>back( )</code>      | Provides reference to the last element              |
| <code>at( )</code>        | Provides reference to an element                    |

### 17.3 Write a program to remove elements from vector object.

```
# include <iostream>
```

```

# include <vector>
# include <math.h>
using namespace std;

int main( )
{
    vector<float> e;
    cout.precision(2);
    cout <<"\n Original elements : ";
    for (int k=5;k<12;k++)
    {
        e.push_back(sqrt(k));
        cout <<sqrt(k)<<" ";
    }

    vector <float> ::iterator ir=e.begin( );
    e.erase (ir+3,ir+5);

    int s=e.size( );

    cout <<"\n The elements after erase( ) : ";
    for (k=0;k<s;k++)
    cout <<e[k]<<" ";
    cout <<"\n";
}

return 0;
}

```

## **OUTPUT**

**Original elements : 2.2 2.4 2.6 2.8 3 3.2 3.3**

**The elements after erase( ) : 2.2 2.4 2.6 3.2 3.3**

***Explanation:*** In the above program, `e` is an object of a vector capable of storing float values. The square roots of 5 to 12 numbers are stored in the object `e`. The `erase( )` function erases 4th and 5th element. The variable `ir` is an iterator and it gives reference to the element. The `erase( )` function can be used with one or more arguments. The last `for` loop displays the elements after deletion.

## **17.9 LIST**

The list is frequently used feature. It allows bi-directional linear list. Elements can be inserted or deleted at both the ends. The elements in the list can be accessed sequentially. Iterators are used to access individual elements of the list. **Table 17.8** shows the functions of list class.

**Table 17.8** Functions of list class

| Function      | Task                                                       |
|---------------|------------------------------------------------------------|
| clear( )      | Erases entire elements                                     |
| back( )       | Provides reference to the end elements                     |
| empty( )      | Determines if the list is vacant or not                    |
| begin( )      | Provides reference to the first element                    |
| erase( )      | Erases given elements                                      |
| end( )        | Provides reference to the end element of the list          |
| merge( )      | Combines two sorted lists                                  |
| pop_front( )  | Erases the first elements                                  |
| pop_back( )   | Erases the last elements                                   |
| remove( )     | Erases elements as specified                               |
| reverse( )    | Reverses the list elements                                 |
| insert( )     | Adds given elements                                        |
| push_back( )  | Appends an element to the end                              |
| push_front( ) | Appends an element to the front                            |
| size( )       | Provides the size of the list                              |
| unique( )     | Erases identical elements in the list                      |
| resize( )     | Changes the size of the list                               |
| swap( )       | Swaps the element of a list with those in the calling list |
| sort( )       | Sorts the list elements                                    |

#### **17.4 Write a program to add elements to both the ends of the list and display the numbers.**

```
# include <iostream>
# include <list>

using namespace std;

void show( list <int> &num)
{
    list<int> :: iterator n;
    for (n=num.begin( );n!=num.end( ); ++n)
        cout <<*n<<" ";
}

int main( )
{
list <int> list;
list.push_back(5);
list.push_back(10);
list.push_back(15);
list.push_back(20);

cout<<"\nNumbers are ";
show(list);
list.push_front(1);
list.push_back(25);
cout<<"\nAfter adding numbers are ";
show(list);
return 0;
}
```

#### **OUTPUT**

**Numbers are 5 10 15 20**

**After adding numbers are 1 5 10 15 20 25**

**Explanation:** The above program is same as the previous one. In addition, here two elements are added to both the ends of the list. The function `push.front()` adds elements to the front end of the list and `push_back()` function appends the element to the rear end of the list. The output shows all the elements of the list.

#### **17.5 Write a program to delete elements from both ends of the list.**

```
# include <iostream>
# include <list>
```

```

using namespace std;

void show( list <int> &num)
{
    list<int> :: iterator n;
    for (n=num.begin( );n!=num.end( ); ++n)
        cout <<*n<<" ";
}

int main( )
{
    list <int> list;
    list.push_back(5);
    list.push_back(10);
    list.push_back(15);
    list.push_back(20);

    cout<<"\nNumbers are ";
    show(list);
    list.pop_front( );
    list.pop_back( );
    cout<<"\nAfter deleting numbers are ";
    show(list);
    return 0;
}

```

## OUTPUT

**Numbers are 5 10 15 20**

**After deleting numbers are 10 15**

**Explanation:** In the above program, the member function `pop_front()` is used to delete the front element of the list and the `pop.back()` function is used to delete the back element of the list. The output shows the list of numbers.

## 17.6 Write a program to sort the list elements and display them.

```

# include <iostream>
# include <list>

using namespace std;
void show( list <int> &num)
{
    list<int> :: iterator n;

```

```

        for (n=num.begin( );n!=num.end( ); ++n)
            cout <<*n<<" ";
    }

int main( )
{
    list <int> list;
list.push_back(23);
list.push_back(19);
list.push_back(5);
list.push_back(15);
list.push_back(25);
list.push_back(20);

cout<<"\n Unsorted list :";
show(list);
cout<<"\n Sorted      list : ";
list.sort( );
show(list);
return 0;
}

```

## OUTPUT

**Unsorted list :23 19 5 15 25 20**

**Sorted list : 5 15 19 20 23 25**

**Explanation:** In the above program, the `push_back()` function is used to add elements to the `list` object. The function `sort()` is used to sort the list element in ascending order. The program shows the list of unsorted and sorted elements using `show()` function.

## 17.7 Write a program to merge two lists and display the merged list.

```

# include <iostream>
# include <list>

using namespace std;

void show( list <int> &num)
{
    list<int> :: iterator n;
    for (n=num.begin( );n!=num.end( ); ++n)
        cout <<*n<<" ";
}

```

```

int main( )
{
    list <int> listX,listY;
    listX.push_back(23);
    listX.push_back(19);
    listX.push_back(5);

    listY.push_back(15);
    listY.push_back(25);
    listY.push_back(20);

    cout<<"\n Elements of listX :";
    show(listX);
    cout<<"\n Elements of listY :";
    show(listY);

    listX.merge(listY);
    cout <<"\n Merged list : ";
    show(listX);
    return 0;
}

```

## **OUTPUT**

**Elements of listX :23 19 5**

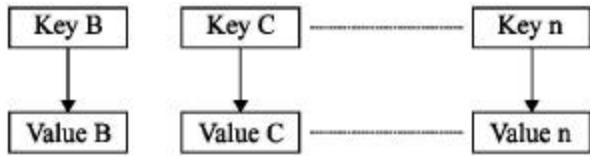
**Elements of listY :15 25 20**

**Merged list : 15 23 19 5 25 20**

***Explanation:*** In the above program, the `listX` and `listY` are initialized with three objects each with the function `push_back()`. The `merge()` function merges the elements of two list objects into one. The elements are merged in the calling object. Thus, in this program, `listX` calls the function and `listY` is sent as an argument. The `listX` contains its own elements and elements of `listY`. In the output elements of `listX` and `listY` are displayed.

## **17.10 MAPS**

A map is a series of pairs of key names and values associated with it as shown in [Figure 17.4](#). Access to data values depends upon the key and it is very quick. We must specify the key to get the corresponding value.



**Fig. 17.4** Keys and values in maps

### 17.8 Write a program to manipulate list of item names and their code numbers using maps.

```

#include <iostream>
#include <map>
#include <string>
using namespace std;

typedef map<string,int> item_map;

int main ( )
{
    int sz;
    string item_name;
    int codeno;
    item_map item;
    cout <<"Enter item name and code number for 2 items: \n";
    for (int i=0;i<2;i++)
    {
        cin>>item_name;
        cin>>codeno;
        item[item_name]=codeno;
    }
    item["PC"]=2510;

    item.insert(pair<string,int> ("printer",2211));
    sz=item.size( );
    cout<<"\n Size of map :"<<sz <<"\n\n";
    cout <<"List of item name and code numbers \n";
    item_map::iterator t;
    for ( t=item.begin( );t!=item.end( );t++)
        cout <<(*t).first <<" " <<(*t).second<<"\n";cout <<"\n";
    cout <<"Enter item name : ";
    cin>>item_name;
    codeno=item[item_name];
    cout <<"Code Number : "<<codeno <<"\n";
    return 0;
}

```

#### OUTPUT

**Enter item name and code for 2 items:**

**TV 51**

**CD 52**

**Size of map :4**

**List of item name and code numbers**

**CD 52**

**PC 2510**

**TV 51**

**printer 2211**

**Enter item name : PC**

**Code Number : 2510**

**Explanation:** In the above program, using `typedef` statement `item_map` variable is created. The first `for` loop and `cin` statement reads two records through the keyboard. Using member function `insert( )` and assignment two records are inserted. The `size( )` member function returns the number of records. The variable `t` is an iterator to the map. The second `for` loop and the iterator produces the list of item names and code numbers. At the end, the program displays code number associated with the entered item name. The `item_name` is a key. The list is automatically arranged in sorted order.

A map is also known as an associative array. The key is given with the help of subscript operator `[ ]` as given below:

```
item ["PC"] = 2510;
```

The above statement creates record for "PC" and links it with code number 2510. The `codeno` is an object. It is also possible to insert and remove pairs at any point in the map. The `insert( )` and `delete( )` functions perform these tasks. Frequently used functions are described in [Table 17.9](#).

### **17.9 Write a program to clear map using clear( ) function.**

```
# include <iostream>
# include <map>
# include <string>
using namespace std;
typedef map<string,int> item_map;

int main( )
```

```

{
    int sz;
    string item_name;
    int codeno;
    item_map item;

    item["PC"]=2510;

    item.insert(pair<string,int> ("printer",2211));
    sz=item.size( );
    cout<<"\n Size of map :"<<sz <<"\n\n";

    item.clear( );
    sz=item.size( );

    cout<<"\n after clear ( ) Size of map :"<<sz <<"\n\n";
    return 0;
}

```

## **OUTPUT**

**Size of map :2**

**after clear ( ) Size of map :0**

**Explanation:** In the above program, two records are added in the map item\_map . Using size( ) function, size (total number of records) are displayed i.e. the clear( ) function clears the contents of the map. After clear( ) function, the size of map becomes zero.

Table 17.9 describes important functions of map class.

**Table 17.9** Important functions of the map class

| <b>Function</b> | <b>Task</b>                                 |
|-----------------|---------------------------------------------|
| clear( )        | Removes all elements from the map           |
| begin( )        | Provides reference to the first element     |
| erase( )        | Removes the given elements                  |
| insert( )       | Inserts the elements as given               |
| empty( )        | Determines whether the map is vacant or not |

|                      |                                                                   |
|----------------------|-------------------------------------------------------------------|
| <code>end( )</code>  | Provides reference to the end of the map                          |
| <code>size( )</code> | Provides the size of the map                                      |
| <code>find( )</code> | Provides the location of the given elements                       |
| <code>swap( )</code> | Swaps the elements of the given map with those of the calling map |

## 17.11 FUNCTION OBJECTS

A function object is a function. It is embedded in a class and hence behaves as an object. The class has same characteristics as template and can be put to use with different data types. The class can hold only one member function and the overloaded ( ) operator and never accommodates data. Function objects are frequently used as parameters for algorithms and containers.

```
sort( num, num+3, greater <float>( ));
```

Here, `num[ ]` is an array. We can use object `greater<float>( )` to sort elements of the array in descending order. The standard template library also contains several pre-defined objects. These objects perform arithmetical and logical operations. The header file `<functional>` should be included while performing these operations. There are function objects matching to foremost C++ operators. The variables `a` and `b` are objects of the class. `K` is a parameter sent to the function object. All the function objects described in the Tables [17.10](#), [17.11](#) and [17.12](#) are defined in `functional` header file.

**Table 17.10** Relational function objects

| Function object                     | Narration              |
|-------------------------------------|------------------------|
| <code>not_equal_to&lt;K&gt;</code>  | <code>a != b</code>    |
| <code>equal_to&lt;K&gt;</code>      | <code>a == b</code>    |
| <code>greater&lt;K&gt;</code>       | <code>a &gt; b</code>  |
| <code>greater_equal&lt;K&gt;</code> | <code>a &gt;= b</code> |
| <code>less&lt;K&gt;</code>          | <code>a &lt; b</code>  |
| <code>less_equal&lt;K&gt;</code>    | <code>a &lt;= b</code> |

**Table 17.11** Logical function objects

| Function object | Narration |
|-----------------|-----------|
| logical_and<K>  | a && b    |
| logical_not<K>  | ! a       |
| logical_or<K>   | a    b    |

**Table 17.12** Arithmetic function objects

| Function object | Narration |
|-----------------|-----------|
| plus<K>         | a + b     |
| divides<K>      | a / b     |
| modulus<K>      | a % b     |
| multiplies<K>   | a * b     |
| minus<K>        | a - b     |
| negate<K>       | - a       |

### 17.10 Write a program to sort an array using function object.

```
# include <iostream>
# include <algorithm>
# include <functional>

using namespace std;

int main( )
{
    float a[]={1.5,6.5,2.5};
    float b[]={6.4,4.8,1.4};

    sort (a,a+3,greater<int>());
    sort (b,b+3);

    for (int j=0;j<3;j++) cout <<a[j]<<" ";

    cout <<"\n";
}
```

```

for (int k=0;k<3;k++)    cout <<b[k]<<" ";
cout <<"\n";

float s[6];
merge (a,a+3,b,b+3,s);

for (j=0;j<6;j++)  cout <<s[j] <<" ";
cout <<"\n";
return 0;
}

```

## **OUTPUT**

**6.5 2.5 1.5**

**1.4 4.8 6.4**

**1.4 4.8 6.4 6.5 2.5 1.5**

**Explanation:** In the above program, two float arrays `a[ ]` and `b[ ]` are declared and initialized. The function `sort( )` both the arrays. The array `a[ ]` is sorted with the help of function object `greater<float>` and the array `b[ ]` is sorted without function. At the end, the `merge( )` function combines elements of both the arrays and the resulting array is stored in the array `s[ ]`. The contents of the merged array is displayed.

## **SUMMARY**

- (1) The STL is a new expansion in C++ language. All-famous compiler vendors give STL as a feature of their compilers.
- (2) Meng Lee and Alexander Stepanov of Hewlett-Packard introduced STL.
- (3) STL is portative between different operating systems. STL contents are defined in the namespace `std`.

(4) The important subdivisions are (a) containers, (b) algorithms and (c) iterators.

(5) A container is an object that holds data or other objects. An algorithm is a process that is useful to handle the data stored in containers. The STL consists of about 60 standard algorithms that give support to perform frequent and primary operations such as copying, finding, sorting, merging, and initializing.  
 (6) An iterator is an object. It is treated as a pointer. It indicates or points to data elements in a container. It is utilized to move between the data of containers.

- (7) The standard C++ library has a sequence of container classes. The container classes are robust and support C++ programmers to control common programming operations.
- (8) Sequence containers hold data elements in a linear series.
- (9) We know that arrays are used to store similar data elements. The data elements are stored in continuous memory locations. The container class vector is same as the array.
- (10) A vector container class is accelerated to allow quick access to its elements in sequence.
- (11) The list container allows a programmer for usual deletion and insertion of items. It is defined in header file <list>. It acts as double linked list.
- (12) A deque is similar to a multiple ended vector. It has ability like vector class to perform sequential read and write operations. The deque class allows improved front-end and back-end operations.
- (13) The stack, queue, and priority\_queue are called as container adaptors. All these container adaptors can be produced from unlike sequence containers.
- (14) Algorithms are independent template functions. They are not member or friend functions. They allow the programmer to manipulate data stored in various containers. Each container has functions for performing common operations.
- (15) Iterators act as pointers. They are used for accessing contents of container classes.
- (16) The vector is a more useful container. Like arrays, it stores elements in neighbouring memory locations. Each vector can be directly accessed using the operator [ ]. A vector is able to enlarge its size if extra element is assigned to it.
- (17) The list is a frequently used feature. It allows bi-directional linear list. Elements can be inserted or deleted at both the ends.
- (18) A map is a series of pairs of key names and values associated with it.

### EXERCISES

**[A] Answer the following questions.**

- (1) What is STL? Explain in brief.
- (2) Describe different sub-divisions of STL with their definitions.
- (3) What are containers? Describe types of containers with their components.
- (4) What do you mean by algorithms?
- (5) What are iterators?
- (6) What is the difference between iterator and pointer?
- (7) Describe any four functions of a vector class with their operations.

- (8) Describe any four functions of a list class with their operations.
- (9) What is a map? How does it work?
- (10) What do you mean by function object ?
- (11) Describe any four functions of a map class with their operations.
- (12) What are function objects?

**[B] Answer the following by selecting the appropriate options.**

- (1) A container is an object that
  - a) stores data
  - b) holds data temporarily
  - (c) both (a) and (b)
  - (d) none of the above
- (2) An algorithm is a process that
  - (a) processes the data
  - (b) stores the data
  - (c) sorts the data
  - (d) none of the above
- (3) An iterator is similar to
  - (a) pointer
  - (b) array
  - (c) class
  - (d) none of the above
- (4) The function object is similar to
  - (a) object
  - (b) variable
  - (c) both (a) and (b)
  - (d) none of the above
- (5) An algorithm is a part of
  - (a) standard template library
  - (b) class template
  - (c) iostream
  - (d) none of the above

**[C] Attempt the following programs.**

- (1) Write a program to demonstrate use of operator `for_each( )`.
- (2) Write a program to create function object.
- (3) Write a program to manipulate list of item names and their code numbers using maps.
- (4) Write a program to copy elements of one list object to another object.  
Display the contents of both the objects.
- (5) Write a program to merge two lists and display the merged list.

- (6) Write a program to demonstrate use of `fill( )` algorithm.
- (7) Write a program to transverse list using iterators.
- (8) Write a program to add and display elements in the vector object.
- (9) Write a program to add elements to both ends of the list and display the numbers.
- (10) Write a program to insert an element in the vector object. Use member function and iterator.

# 18

CHAPTER

## Additional About ANSI and TURBO C++

—• 18.1 Introduction

- [18.2 Innovative Data Types](#)
- [18.3 New Typecasting Operators](#)
- [18.4 The Keyword explicit](#)
- [18.5 The Keyword mutable](#)
- [18.6 Namespace Scope](#)
- [18.7 Nested Namespaces](#)
- [18.8 Anonymous Namespaces](#)
- [18.9 The using Keyword](#)
- [18.10 Namespace Alias](#)
- [18.11 The Standard Namespace STD](#)
- [18.12 ANSI and TURBO C++ Keywords](#)
- [18.13 ANSI and TURBO C++ Header Files](#)
- [18.14 C++ Operator Keywords](#)

## 18.1 INTRODUCTION

This chapter deals with various concepts that are not discussed so far including new improvements suggested by ANSI committee in C++ language. In addition, this chapter also contains information regarding additional

keywords provided by Turbo C++. List of commonly useful header files is also given.

The aim behind improvement made by ANSI is to allow the programmer to develop real world application programs in a simple and easy way. The new characteristic added allows a programmer improved achievements in complicated conditions. It is also possible to make programs without using the new characteristics. However, with additional features, the source code will be reduced for complex problems and programs can be made more efficient and short.

## 18.2 INNOVATIVE DATA TYPES

The ANSI/ISO committee introduced new data types to make better scope of standard data types in C++. These new data types are `bool` and `wchar_t`.

### (1) THE BOOL DATA TYPE

The `bool` is a keyword. The `bool` data type is used to store Boolean values i.e. `true` and `false`. The `true` and `false` are also keywords. C++ treats `true` and `false` as values. The default numeric value of `false` is zero and `true` is one. All conditional expressions return values of this type. For example, the expression `5>4` is true and it returns one. The variables of these data types are declared below:

#### Declaration of `bool` variables

```
bool t1;           // t1 is a variable of bool data type
t1= true          // t1 contains true (1) as value
bool f1=false     // declaration and initialization is done in same
type
```

These `bool` data type variables can be used in mathematical expressions as given in the statement below.

```
int m = true+2*5+t1;
```

When `bool` data types are used in expressions like above, they are automatically evaluated to integers. It is also possible to change by nature the data type pointers, floating, and integer values to `bool` data type.

#### Initialization of `bool` variables

```
bool a= 1;         // Variable a contains true value
bool b=0           // Variable b contains false value
bool c=1.2         // Variable c contains true value
```

### 18.1 Write a program to declare `bool` variables and display the values.

```
# include <iostream>
using namespace std;
```

```

int main( )
{
    bool t=true;
    bool f=false;

    cout <<"\n t = "<<t;
    cout <<"\n f = "<<f;

    bool g=1;

    if (g)

        cout <<"\n true";
    else
        cout <<"\n false";
    return 0;
}

```

## **OUTPUT**

**t = 1**  
**f = 0**  
**true**

**Explanation:** In the above program, `t` and `f` are variables of `bool` data type. The variable `t` contains `true` and `f` contains `false`. Values of these variables are displayed using `cout` statement. The variable `g` is another variable of `bool` data type and contains value 1 i.e. `true`. The `if( )` statement checks the value of variable `g` and displays the appropriate message.

## **18.2 Write a program to declare variables of `bool` data type and use it in mathematical expressions.**

```

# include <iostream>

using namespace std;

int main ( )
{
    bool t=true;
    bool h,f=false;

    cout <<"\n t = "<<t;
    cout <<"\n f = "<<f;

    int x= 4*t+5*f;
    h=x;

    cout <<"\n x= "<<x;
    cout <<"\n h= "<<h;
    return 0;
}

```

```
}
```

**OUTPUT**

```
t = 1  
f = 0  
x= 4  
h= 1
```

**Explanation:** In the above program, t and h are variables of `bool` data type and contains values `true` and `false`. The variable x is an integer variable. The variables t and f are used in mathematical expression and result obtained is assigned to variable x. The value of integer variable x is assigned to Boolean variable h. The output of the program is given above.

In case, the variable of `bool` type is assigned a value more than 1, for example, `bool x=2;` then the value of x will be considered as true i.e. 1. The variable of `bool` type also supports increment `++` (postfix or pre fix) operation. However, it cannot support decrement `(--)` operation. When a `bool` type variable with value 0 (`false`) is incremented, its value becomes one. However, when a variable contains Boolean value one and if any attempt is made to increase it, its value remains the same.

## (2) THE `WCHAR_T` DATA TYPE

The `wchar_t` character data type stores 16 bit long characters. The 16 bit characters are manipulated to symbolize the character sets of languages which contain characters more than 255. ANSI C++ also defines a character literal called as `wide_character` literal. It occupies two bytes in memory. The `wide_character` literal starts with the letter L, as given below.

**L 'ab'**

It is known as `wide_character` literal.

The declaration of `wchar_t` in header file `stdlib.h` is as follows:

```
typedef char wchar_t;
```

The following program demonstrates the use of `wchar_t` data type.

### 18.3 Write a program to use `wchar_t` data type.

```
# include <iostream.h>  
# include <stdlib.h>  
# include <constream.h>  
  
void main ( )  
{  
  
    clrscr( );  
    wchar_t c='A';
```

```
    cout <<c;  
}
```

## OUTPUT

A

**Explanation:** In the above program, variable c is of type wchar\_t and it is initialized with 'A'. The cout statement displays the same on the screen.

## 18.3 NEW TYPECASTING OPERATORS

Typecasting operation is used to change a value from one type to another. This can be applied at a place where automatic conversion of data types is not done.

### Typecasting

```
double b= (double)j // C typecasting style  
double a =double (d) // C++ typecasting style
```

Both the statements given above are correct and nothing happens wrong when applied in practical applications. New typecasting operators introduced by the ANSI/ISO committee are as follows:

- (1) static\_cast
- (2) dynamic\_cast
- (3) reinterpret\_cast
- (4) constant\_cast

### (1) THE STATIC\_CAST OPERATOR

The static\_cast operator is used for the conversion of standard data types. Using this operator, base class pointer can be converted to derived class pointer. Its syntax is as given below.

```
static_cast <data-type> object
```

The data\_type indicates the target data type of the cast. The specified object is converted to new data type. The new syntax of typecasting can easily be placed in the code. Hence, the above keyword is frequently used by the programmer instead of old format.

## 18.4 Write a program to perform typecasting using static\_cast operator.

```
# include <iostream.h>  
int main( )  
{  
int k=65;  
cout <<"\n size of k=" <<sizeof(k);  
cout <<"\n value of k=" <<k;  
double d=static_cast<double>(k);
```

```

cout <<"\n size of d=<<sizeof(d);
cout <<"\n value of d=<<d;
char c=static_cast<char>(k);
cout <<"\n c = "<<c;
return 0;
}

```

## **OUTPUT**

**Size of k=4**

**Value of k=65**

**Size of d=8**

**Value of d=65**

**c = A**

**Explanation:** In the above program, the integer k is initialized with 65. The variable d is of double data type. The value of k is assigned to d using static\_cast operator. The value of k is assigned to char data type variable c. The output displays the contents of the variable with the number of bytes occupied by them.

## **(2) THE CONST\_CAST OPERATOR**

The const\_cast operator explicitly modifies the const or volatile of a variable. It is used in the following format:

```
const_cast <data-type> (object)
```

In this type of casting, our goal is to change const or volatile nature of the variable. Hence, target and source data types are identical. It is frequently applied for removing const attribute of a variable.

### **18.5 Write a program to convert constant to non-constant.**

```

#include <iostream.h>

void main( )
{
    const int x=0;
    int *p=(int *)&x;
    p=const_cast<int*>(&x);
}

class data
{
    private :
        int d;

public :
void joy( ) const
{
    (const_cast<data*>(this))->d=100;
}

```

```
}
```

```
};
```

**Explanation:** In the above program, address of constant variable is assigned to pointer of non-constant variable and this is done using the operator `const_cast`.

### (3) THE REINTERPRET\_CAST OPERATOR

The `reinterpret_cast` operator is useful when programmer wants to alter one type into dissimilar type. In practical applications, it is used to modify a pointer kind of object to integer kind or vice versa. It is used in the following format:

```
reinterpret_cast <data-type> (object )
```

#### 18.6 Write a program to convert pointers to integers using reinterpret\_cast operator.

```
# include <iostream.h>

void main( )
{
    int b=487;

    int *rp=reinterpret_cast<int*> (b) ;
    cout <<endl << "rp=" << rp;
    rp++;
    cout << endl << "rp=" << rp;

    b=reinterpret_cast<int>(rp);
    cout << endl << "b=" << b;
    b++;
    cout << endl << "b=" << b;
}
```

#### OUTPUT

**rp=0x000001E7**

**rp=0x000001EB**

**b=491**

**b=492**

**Explanation:** In the above program, `b` is an integer type of variable and initialized with 487. The variable `rp` is a pointer and is initialized with the address of variable. The `rp` is increased and the value is increased with four. The output shows the values of the variable.

#### 18.7 Write a program to convert void pointer to char pointer using reinterpret\_cast operator.

```

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

void *getadd ( )
{
    static char text[50];
    return text;
}

void main ( )
{
    char *p=reinterpret_cast<char*>(getadd( ));
    //char *p=getadd();
    strcpy (p," Well Come");
    cout<<p;
}

```

## **OUTPUT**

**Well Come**

**Explanation:** In the above program, the function `getadd()` returns base address of array `text [50]` as a void address. The return type is `void *`. In function `main()`, the obtained address is converted to character type using operator `reinterpret_cast` and assigned to character pointer `p`. The `strcpy()` function copies a string to pointer `p`. Finally the `cout` statement displays the text stored in pointer `p`.

## **(4) THE DYNAMIC\_CAST OPERATOR**

The `dynamic_cast` operator is used to change type of an object during program execution. It is frequently used to typecast on polymorphic objects i.e. when the base class has virtual function. This operator casts the base class pointer to derived class pointer. The operation with `dynamic_cast` is also known as type-secure downcast. It is used in the following format:

`dynamic_cast <data-type> (object)`

The object must be a base class object. Its type is tested and altered. This always does valid conversion. It verifies that the casting is allowable at execution time. It returns null if typecasting is defective.

## **(5) RTTI USING TYPEID OPERATOR**

The **runtime type information (RTTI)** is a new improvement made by ANSI/ISO committee. The `typeid()` operator is used to obtain exact type of object or variable during program execution. This operator returns a reference to an object maintained by the system. This object identifies the type of argument.

The typeid( ) operator is used to identify the type of object or variable during program execution. The syntax is as given below.

```
char *object class = typeid (object) . name ( );
```

### 18.8 Write a program using typeid( ) to identify the type of object.

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

class one
{
public:
    virtual void say( )
    {
    }
};

class two : public one
{};

class three : public one
{};

void main( )
{
one *o;
cout << endl << typeid(o) . name( );

two *t;
cout << endl << typeid(t) . name( );
}
```

#### OUTPUT

```
class one *
class two *
```

**Explanation:** In the above program, class one has one virtual function say( ). The class two is derived from class one. In function main( ), \*o and \*t are pointer objects of class one and two. The typeid( ) operator identifies the type of objects and displays the class name from which they are created.

### 18.9 Write a program to identify the type of variable using typeid( ).

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

void main ( )
{
```

```
int i;
cout << endl << typeid(i).name();

float f;
cout << endl << typeid(f).name();
}
```

## OUTPUT

```
int
float
```

**Explanation:** In the above program, two variables `int (i)` and `float (f)` are declared. Using `typeid( )` function their types are displayed. The output is `int`, and `float` i.e. type of variables (`i`) and (`f`).

## 18.4 THE KEYWORD EXPLICIT

The keyword `explicit` is used to declare constructors of a class `explicit`. When constructor is invoked with single argument, implicit conversion is carried out. In this operation automatic conversion is performed. The type of argument taken is modified to an object type. To avoid such automatic type conversion `explicit` keyword is used followed by constructor's name. The following program explains the use of keyword `explicit`.

### 18.10 Write a program to declare explicit constructor and avoid automatic conversion.

```
# include <iostream.h>

class XYZ
{

int k;
public :
    explicit XYZ (int j)
    {      k=j; }
    void show ( )
    {
        cout <<"\n k = "<<k;

    }
};

int main( )
{
    XYZ XY(545);
    // XYZ XY=450; invalid assignment
```

```
    XY.show( );
    return 0;
}
```

## OUTPUT

k=545

**Explanation:** In the above program, the class XYZ is declared with one integer argument. The class also has one argument constructor followed by explicit keyword and one member function show( ). In function main( ), the statement XYZ XY(545) creates object XY and initializes the member variable k with number 545. The statement XYZ XY=450 is invalid because the constructor is explicit and does not permit automatic conversion. This statement is valid only when the class constructor is not declared explicit.

## 18.5 THE KEYWORD MUTABLE

A member function or an object can be declared as constant using the keyword const. The constant data elements are not modified and the constant member functions can alter value of any variable. The keyword mutable is used when we need to declare constant objects partly i.e. a specific data variable is to be modified. Thus, the data variable that is to be modified is declared as mutable. The declaration is given below.

```
mutable int k;
```

Here, the variable k can be modified though its class or object is declared as constant.

### 18.11 Write a program to use mutable keyword and modify the value of variable.

```
# include <iostream>

using namespace std;

class XYZ
{
private :
    mutable int k;
public :

    explicit XYZ (int d) {           k=d; }

    void value( ) const { k+=10; }

    void show( ) const { cout <<"\n k= "<<k; }
```

```

) ;

int main( )
{
const XYZ x(75);
x.show( );
x.value( );
x.show( );

return 0;
}

```

## **OUTPUT**

**k= 75**

**k= 85**

**Explanation:** In the above program, the class XYZ has one integer mutable member. The class also has value( ) and show( ) constant member functions. In function main( ), x is a constant object of class XYZ. When x is declared, mutable member k is initialized with 75. When the value( ) function is invoked it is incremented with 10. The show( ) function displays the values of the variables.

## **18.6 NAMESPACE SCOPE**

C++ allows variables with different scopes such as local, global etc. with different blocks and classes. This can be done using keyword namespace introduced by ANSI C++. The C++ standard library is a better example of namespace. All classes, templates and functions are defined inside the namespace std. In previous programs we have used the statement using namespace std. This tells the compiler that the members of this namespace are to be used in the current program.

### **(1) NAMESPACE DECLARATION**

The namespace can be defined in the programs. The declaration of namespace is same as class declaration except that the namespaces are not terminated by semi-colon. It is declared in the following format:

#### **Declaration of namespace**

```

namespace namespace_identifier
{
    // Definitions of variable functions and classes, etc.
}

```

#### **Example**

```

namespace num
{

```

```

int n;
void show ( int k )           { cout <<k; }
}

```

In the above example, the variable `n` and the function `show( )` are within the namespace scope `num`. We cannot reach the variable `m` directly. To access the variable and initialize it with a value the statement would be as given below:

```
num :: n=50;
```

Here, `n` is initialized to 50. The scope access operator is used to access the variable. This method of accessing elements becomes embarrassing.

We can also access elements directly using the following declarations:

### **Accessing namespace**

```

using namespace namespace_identifier // directive statement
using namespace_identifier :: member // declaration method

```

The first statement allows access to elements without using scope access operator.

The second statement allows access to only given elements.

### **Accessing namespace**

```

using namespace num
n=10;           // valid statement
show(15);       // valid statement

using namespace :: n;
m=20;           // valid statement
show(14)        // invalid statement

```

## **18.7 NESTED NAMESPACES**

It is also possible to declare nested namespaces. When one namespace is nested in another namespace, it is known as nested namespace.

### **Nested namespaces declaration**

#### **namespace NUM**

```

{
    statement1;

namespace NUM1
{
int k=10;
}
statement2;
}

```

The variable `k` can be accessed using following statements:

### **Variable accessing statements**

```
cout <<NUM::NUM1::K
```

OR

```
using namespace NUM;
cout <<NUM1::k;
```

## 18.8 ANONYMOUS NAMESPACES

A namespace without name is known as an anonymous namespace. The members of anonymous namespaces can access globally in all scopes. Frequent use of such namespaces is to protect global members from same name classes among files. Each file possesses separate anonymous namespace. The following program explains the use of namespaces.

### 18.12 Write a program to create namespace, declare, and access elements.

```
# include <iostream>
using namespace std;

namespace num
{
    int n;
    void show ( ) { cout <<"\n n = "<<n; }
}

int main ( )
{
    num::n=100;
    num::show( );
    return 0;
}
```

#### OUTPUT

**n=100**

**Explanation:** In the above program, the namespace num contains one integer member n and one member function show( ). In function main( ), the variable n is initialized to 100. The members are accessed using scope access operator and the namespace name. The function show( ) displays the value of variable.

## 18.9 THE USING KEYWORD

### (1) USING DIRECTIVE

The using keyword can be used for using declarations as well as using directives. The using directive provides access to all variables declared within the namespace. When using directive method is used, we can directly access the variable without specifying the namespace name. The following example illustrates this:

### **18.13 Write a program to create namespace, declare, and access elements. Use using directive method.**

```
# include <iostream>
using namespace std;

namespace num
{
    int n;
    void show( ) { cout <<"\n n = "<<n; }
}

int main( )
{
    using namespace num;
    n=100;
    show( );

    return 0;
}
```

#### **OUTPUT**

**n=100**

**Explanation:** In the above program, the namespace name is created with one integer variable `n` and function `show( )`. The statement `using namespace num` allows direct access to members of the namespace. Hence, in function `main( )`, `n=100` initializes `n` with 100 and `show( )` displays the value of `n` on the screen.

## **(2) USING DECLARATIONS**

In this method, the `using` keyword is optional. It is also possible to allow permission to access few members of namespaces directly outside the namespace. The program given next explains the above points.

### **18.14 Write a program to use using declaration method of namespaces and access variables.**

```
# include <iostream>
using namespace std;
namespace num
{
    int n;
    void show( )
    { cout <<"\n n = "<<n; }
}
int main ( )
{
    //using namespace num;
```

```

    num::n=100;

    num::show( );
    return 0;
}

```

## OUTPUT

**n=100**

**Explanation:** The above program is same as the last one. Only difference is that the namespace num is not included here. In order to access elements of namespace num, we need to precede the variable name with scope access operator and namespace name. The statement num::n=100 accesses the variable n and initializes it with 100. In the statement num::show( ), the function show( ) is invoked using the same syntax.

### 18.15 Write a program to declare nested namespace and anonymous namespace.

```

# include <iostream>
using namespace std;

namespace num
{
    int j=200;
    namespace num1      {      int k=400;      }
}

namespace  {      int j=500;      }

void main ( )
{   cout  <<"j = "<<num::j <<"\n";
    cout  <<"k = "<<num::num1::k <<"\n";
    cout  <<"j = "<<j <<"\n";
}

```

## OUTPUT

**j = 200**

**k = 400**

**j = 500**

**Explanation:** In the above program, num and num1 are two namespaces. The num1 is declared inside the namespace num. The last namespace defined is unnamed namespace. In function main( ), the members of the namespaces are accessed using scope access operator. The variable j is used in two different scopes.

### **18.16 Write a program to declare functions in namespace. Access the function in main( ).**

```
# include <iostream>
using namespace std;
namespace fun
{
    int add (int a, int b) { return (a+b); }
    int mul (int a, int b); // prototype declaration
}

int fun :: mul (int a, int b) { return (a*b); }

int main( )
{
    using namespace fun;
    cout <<"\n Addition : " <<add(20,5);
    cout <<"\n Multiplication : " <<mul(20,5);
    return 0;
}
```

#### **OUTPUT**

**Addition : 25**  
**Multiplication : 100**

**Explanation:** In the above program, two functions `add()` and `mul()` are declared in namespace `fun`. In function `main()`, the statement `using namespace fun` allows us to access the elements of `fun` namespace directly. Two integer values are passed to functions `add()` and `mul()`. They return addition and multiplication of numbers respectively.

### **18.17 Write a program to declare class in the namespace. Access the member functions.**

```
# include <iostream>
using namespace std;

namespace clas_s
{
class num
{
private :
    int t;
public :
    num (int m) { t=m; }
    void show ( ) { cout <<"\n t = " <<t; }
};

void main( )
```

```

{
    // indirect access using scope access operator
    clas_s ::num n1(500);
    n1.show( );

    // direct access using directive
    using namespace clas_s;
        num n2(800);
        n2.show( );
}

```

## **OUTPUT**

**t = 500**

**t = 800**

**Explanation:** In the above program, class num is defined in the namespace clas\_s. The class has one integer variable t and a member function show(). The member function show() displays the contents of the variable. In function main(), n1 and n2 are objects of class num. The data members are initialized using constructors. The members of class num are accessed with and without scope access operator. Both methods of accessing elements are explained in previous programs.

## **18.10 NAMESPACE ALIAS**

Alias means another name. A namespace alias is designed to specify another name to the existing namespace. It is useful if previous name is long. We can specify a short name as alias and call the namespace members. The following program explains how alias is specified.

### **18.18 Write a program to specify alias to existing namespace.**

```

# include <iostream>
using namespace std;

namespace number

{
    int n;
    void show( )
    { cout <<"\n n = "<<n; }

}

int main ( )
{
    namespace num=number;
    num::n=200;
    number::show( );
    return 0;
}

```

```
}
```

**OUTPUT**

**n=200**

**Explanation:** In the above program, the namespace number is defined. In function main( ), its alias name is created as follows:

```
namespace num=number;
```

The num is another name given to the namespace number. The member of namespace number can be accessed using both the names i.e. num and number. The variable n is accessed using name num and the show( ) function is accessed using name number.

## 18.11 THE STANDARD NAMESPACE STD

We are frequently using the statement `using namespace std;`. This statement inserts complete library in the current program. This is same as including a header file and the contents of header file is available in the current program. We can access all classes, functions, and templates declared inside this namespace.

```
using namespace std;
```

As mentioned earlier, the above method of specifying a namespace is called `using` directive. The above declaration makes possible to access all members of namespace directly. All header files also use the namespace feature. If we include header files and namespace together in the same program, the variable declared in the header file will be global. Instead of the above declaration, following declaration must be followed in order to prevent serious bugs. Consider the following program.

## 18.19 Write a program to access members of namespace std applying using declaration method.

```
# include <iostream>

int main ( )
{
int age;
std::cout<<"\nEnter your age : ";
std::cin>age;
std::cout<<"\n Your age is : "<<age;
}
```

**OUTPUT**

**Enter your age : 24**  
**Your age is : 24**

**Explanation:** In the above program, the l member of namespace std cannot be accessed directly. Consider the following statement:

```
std::cout<<"\nEnter your age : ";
```

Here, the namespace name std is preceded before object cout. In the same way, cin object is accessed.

## (1) GUIDELINES FOR DEFINING NAMESPACE

- (1) The namespaces are not terminated by semi-colon like classes.
- (2) Declarations done outside the namespaces are considered as members of global namespace.
- (3) Members of namespaces are defined inside the scope of namespace.
- (4) The namespace definition cannot be defined inside any function. It must be global.

## 18.12 ANSI AND TURBO C++ KEYWORDS

ANSI C++ has introduced various new keywords according to new characteristics of the language. The following table describes keywords of both ANSI and Turbo C++.

**Table 18.1** Keywords supported by ANSI and Turbo C++

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

Additional keywords for Turbo-C++

|           |        |           |        |
|-----------|--------|-----------|--------|
| cdecl     | _ss_ss | _export   | far    |
| _fastcall | huge   | interrupt | _loads |
| near      | pascal | _saveregs | _seg   |

## (1) THE KEYWORD PASCAL

The `pascal` keyword is used to declare a variable or a function using Pascal-style naming convention

**Syntax:**

```
pascal <data definition>;  
pascal <function definition>;
```

In addition, `pascal` declares Pascal-style parameter-passing conventions when applied to a function header (first parameter pushed first; the called function cleans up the stack).

**Examples:**

```
int pascal x;  
void pascal show(int x, int y);
```

### 18.20 Write a program to use `pascal` keyword and change the calling convention of C++ to pascal style.

```
# include <iostream.h>  
# include <constream.h>  
  
void main ( )  
{  
  
    clrscr( );  
    void pascal show (int,int);  
    int x=2;  
    show (++x,x);  
  
}  
void pascal show ( int x, int y)  
{  
cout <<"\n x="<<x <<" y="<<y;  
}
```

#### OUTPUT

x=3 y=3

**Explanation:** We know that in C/C++, calling convention is from right to left i.e. right most variable is first pushed in the stack. In order to change this calling convention style with pascal style, the `pascal` keyword can be used. The `pascal` keyword is given in the prototype declaration and function declarator. When we pass argument to this function, the calling convention will be left to right. The calling conventions are described in chapter “C++ and Memory”.

## (2) THE KEYWORD CDECL

The keyword `cdecl` is used to declare a variable or a function in C-style naming convention and pushes arguments in stack in C-style parameter passing conventions.

```
cdecl <data definition> ;
cdecl <function definition> ;
```

**Example:**

```
int cdecl l x;
void cdecl show(int x, int y);
```

### 18.21 Write a program to demonstrate the use of `cdecl` keyword.

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

void main ( )
{
    clrscr( );
    void cdecl      show (int,int);
    int x=7;
    show (++x,x);

}
void cdecl show ( int x, int y)
{
cout <<"\n x="<<x <<" y="<<y;
```

#### OUTPUT

**x=8 y=7**

**Explanation:** The above program is same as the previous one. Here, `cdecl` keyword is used to apply calling convention to C.

## 18.13 ANSI AND TURBO C++ HEADER FILES

### (1) ANSI / ISO ("INCLUDE") FILES

ANSI/ISO committee has introduced a new style for header files. The header file is now without extension `.h`.

**Example**

```
# include <iostream>
# include <vector>
```

In previous examples we have frequently used the above header files. However, the conventional method `<iostream.h>` is also valid. Few header files are renamed, for

example, `<limit.h>` with `<climit>`, `math.h` with `<cmath>`, `<stdio.h>` with `<cstdio>` and so on. Only few names are listed in [Table 18.2](#).

**Table 18.2** ANSI/ISO header file names

| C header files               | C++ header files              |
|------------------------------|-------------------------------|
| <code>&lt;cstring&gt;</code> | <code>&lt;typeinfo&gt;</code> |
| <code>&lt;ctime&gt;</code>   | <code>&lt;string&gt;</code>   |
| <code>&lt;cstdlib&gt;</code> | <code>&lt;new&gt;</code>      |
| <code>&lt;cassert&gt;</code> | <code>&lt;defines&gt;</code>  |
| <code>&lt;cctype&gt;</code>  | <code>&lt;sstream&gt;</code>  |
| <code>&lt;cerrno&gt;</code>  | <code>&lt;ostream&gt;</code>  |
| <code>&lt;cfloat&gt;</code>  | <code>&lt;iostream&gt;</code> |
| <code>&lt;climits&gt;</code> | <code>&lt;ios&gt;</code>      |
| <code>&lt;cmath&gt;</code>   | <code>&lt;iomanip&gt;</code>  |

## (2) TURBO C++ HEADER ("INCLUDE") FILES

[Table 18.3](#) contains few names of frequently used header files of Turbo C++. The list is very exhaustive. You can view the complete list of header files through *help menu* (contents option) present in Turbo C++ editor.

**Table 18.3** Turbo C++ header files

|                          |                        |                         |
|--------------------------|------------------------|-------------------------|
| <code>iostream.h</code>  | <code>complex.h</code> | <code>strstrea.h</code> |
| <code>constream.h</code> | <code>fstream.h</code> | <code>bcd.h</code>      |
| <code>process.h</code>   | <code>generic.h</code> | <code>string.h</code>   |
| <code>stdlib.h</code>    | <code>iomanip.h</code> | <code>values.h</code>   |
| <code>math.h</code>      | <code>new.h</code>     | <code>float.h</code>    |

## 18.14 C++ OPERATOR KEYWORDS

The C++ committee suggested keywords for operators such as `&&`, `||` etc. The keywords can be used in expressions instead of operators. [Table 18.4](#) describes the C++ operator keywords.

**Table 18.4** C++ operator keywords

| Operator            | Keyword                 |
|---------------------|-------------------------|
| <code>xor_eq</code> | <code>^=</code>         |
| <code>xor</code>    | <code>^</code>          |
| <code>or_eq</code>  | <code> =</code>         |
| <code>or</code>     | <code>  </code>         |
| <code>not_eq</code> | <code>!=</code>         |
| <code>not</code>    | <code>!</code>          |
| <code>compl</code>  | <code>~</code>          |
| <code>bitor</code>  | <code> </code>          |
| <code>bitand</code> | <code>&amp;</code>      |
| <code>and_eq</code> | <code>&amp;=</code>     |
| <code>and</code>    | <code>&amp;&amp;</code> |

## SUMMARY

- (1) The ANSI C++ has added many new characteristics to the original C++. The new characteristics added allow improved achievements in complicated conditions.
- (2) The ANSI C++ has introduced new data types to make better scope of standard data types in C++. These new data types are `bool` and `wchart_t`.
- (3) ANSI C++ defined new cast operators, such as `static_cast`, `dynamic_cast`, `reinterpret_cast`, and `constant_cast`.
- (4) The `static_cast` operator is used for the conversion of standard data types.
- (5) The `const_cast` operator explicitly modifies the `const` or `volatile` of a variable.
- (6) The `reinterpret_cast` operator is useful when a programmer wants to alter one type into basically dissimilar type.

(7) The `dynamic_cast` operator is used to change type of an object during program execution. It is frequently used to typecast on polymorphic objects.

(8) The `typeid` operator is useful to know the types of strange objects.

During program execution we can obtain their class name.

(9) The keyword `explicit` is used to declare constructors of a class explicit.

(10) The keyword `mutable` is used when we need to declare constant object partly i.e. a specific data variable is to be modified.

(11) C++ allows variables with different scopes such as local, global etc. with different blocks and classes. ANSI C++ has added a new keyword `namespace` to declare a scope, which has global variables. The C++ standard library is the better example of namespace.

(12) The namespace can be defined in the programs. The declaration of namespace is same as class declaration except that the namespace is not terminated by semi-colon.

(13) It is also possible to declare nested namespace. When one namespace is nested in another namespace it is known as nested namespace.

(14) A namespace without name is known as an anonymous namespace. The members of anonymous namespaces can be accessed globally in all scopes.

### EXERCISES

**[A] Answer the following questions.**

(1) Describe the new data types introduced by ANSI C++ standard committee.

(2) How the `bool` data type is useful?

(3) Describe `const_cast` operator with its use.

(4) What do you mean by `dynamic_cast` and `static_cast`?

(5) What is namespace? How does it work?

(6) What is `typeid` operator?

(7) When are the `explicit` and `mutable` keywords useful?

(8) What are anonymous namespaces?

(9) Explain alias of namespace.

(10) List the additional keywords provided by Turbo C++.

**[B] Answer the following by selecting the appropriate option.**

(1) An unnamed namespace is one

(a) that does not have a name

(b) compiler gives the name automatically

(c) it is pre-defined

(d) none of the above

(2) The `explicit` keyword is used to declare

(a) constructors explicit

- (b) functions explicit  
(c) destructors explicit  
(d) none of the above
- (3) The `mutable` keyword is used to modify the  
(a) constant object  
(b) ordinary object  
(c) volatile object  
(d) none of the above
- (4) The `typeid` operator is used to know  
(a) type of the object and variable  
(b) type of function  
(c) return value of function  
(d) none of the above
- (5) An alias to existing namespace means  
(a) another name to the existing namespace  
(b) name of nested namespace  
(c) a name given by the compiler to unnamed namespace  
(d) none of the above
- (6) Which of the following keywords is not a part of ANSI C++?  
(a) `far`  
(b) `public`  
(c) `static`  
(d) none of the above
- (7) Consider the statement `bool x=1.` The value of variable `x` after increment operation will be  
(a) 1  
(b) 2  
(c) 0  
(d) -1
- [C] Attempt the following programs.**
- (1) Write a program to explain the use of `mutable` and `explicit` keywords.
  - (2) Write a program to declare namespace with variables and functions. Use member of the namespace in the function.
  - (3) Write a program to define constant member function and constant object. Modify the contents of member variables. Use `mutable` keyword.
  - (4) Write a program to demonstrate the use of `static_cast` operator.
  - (5) Write a program to declare `bool` variables and display the values.

- (6) Write a program to perform increment operation with variables of `bool` type and display the value.
- (7) Write a program using `typeid( )` to identify the type of objects and data types.
- (8) Write a program to access members of namespace `std` by applying `using` declaration method.

# 19

CHAPTER

## Marching Towards JAVA

C  
H  
A  
P  
T

- [19.1 Evolution of Java](#)
- [19.2 Java Technology](#)
- [19.3 Features of Java](#)

- [19.4 Overview of Java](#)
- [19.5 Differences Between C/C++ and Java](#)
- [19.6 Keywords](#)
- [19.7 Java Virtual Machine \(JVM\)](#)
- [19.8 Structure of a Java Program](#)
- [19.9 Implementing a Java Program](#)
- [19.10 Constants, Variables and Data Types](#)
- [19.11 Classes, Objects and Methods](#)
- [19.12 Overriding Methods](#)
- [19.13 Multithreading](#)
- [19.14 Life Cycle of Thread](#)
- [19.15 Packages](#)
- [19.16 Applets](#)

## 19.1 EVOLUTION OF JAVA

In January 1991, Bill Joy, James Gosling, Mike Sheradin, Patrick Naughton, and several other experts met in Aspen, Colorado for the first time to discuss about a programming language, which was presented by Bill Joy. They wish to invent a better programming tool in the direction of Sun's "Stealth Project."

The members of the Stealth Project were later recognized by another name “Green Project.” James Gosling's objectives were to modify and extend C++. This gave a better path for the progress of the project. Originally, the language was given the name “Oak,” which later on was renamed as “JAVA” in the year 1995. JAVA is a slang term for coffee. The first working version took about 18 months to develop.

Later many more helping hands were held together in the design and development of this project. Experts then speculated the name “JAVA” from some project members calls James Gosling, Arthur Van Hoff and Andy Bechtolsheim. At last, the day has come to implement the full version of Java, the vision of which was to develop “smart” consumer electronic devices that could be centrally controlled and programmed by a handheld remote control like device. It was expected to lead the humanity to a new era of modernization.

## **19.2 JAVA TECHNOLOGY**

### **(1) JAVA PLATFORM**

A platform is a hardware or software background in which a program runs. The computers, nowadays, are interconnected by different networks worldwide and operate on many platforms. Among them are Microsoft Windows, Macintosh, OS/2, UNIX and NetWare. Software must be compiled separately to run on each platform, because they are specific to the same. The binary file for an application that runs on one platform cannot run on another platform because the binary file is platform-specific.

The Java platform is a software platform that delivers and runs highly interactive, dynamic, and secure applets and applications on networked computer systems, executing byte codes, which are not specific to any physical machine, but are machine instructions for a virtual machine. Files can run on any operating system that is running the Java platform. This portability is possible because at the core of the Java platform is the Java virtual machine.

The Java platform has two components:

- The Java Virtual Machine (Java VM)
- The Java Application Programming Interface (Java API)

### **(2) JAVA AND INTERNET**

Hot Java is a web browser that runs applets on Internet. Hence, Java is strongly connected with the Internet. Now users can easily compile applet

programs in the Internet and run them locally by using a web browser like “Hot Java.” They can even use this Web browser to download an applet located in a remote computer anywhere in the Internet in the world and run the application in their computer using the browser. Another important application is that Internet users can also set up their Web sites containing java applets that could be used by other remote users of Internet. This ability of Java has made it popular in the world.

### **(3) JAVA AND WORLD WIDE WEB**

World Wide Web is used by the Internet's distributed environment. It contains information and controls about the Web pages. It is not like menu driven system. **WWW** is open through any direction. We can navigate any new part from any part. There is no restriction to follow the same path all the way every time. It is made possible with the help of a language called HTML. The HTML is Hypertext Markup Language. Web pages contain tags that enable us to find, retrieve, manipulate, and display documentation worldwide. Java is used in distributed environments such as Internet. Java is being easily imported into Web.

## **19.3 FEATURES OF JAVA**

Java has the following important features:

- (1) Compiled and interpreted
- (2) Platform independent and portable
- (3) Object-oriented
- (4) Robust and secured
- (5) Distributed
- (6) Familiar, simple and small
- (7) Multithreaded and interactive
- (8) High performance
- (9) Dynamic and extensible

### **(1) JAVA ENVIRONMENT**

It includes large development tools, hundreds of classes and methods. Development tools are included in Java Development Kit (JDK). Classes and methods are parts of Java Standard Library (JSL).

### **(2) JDK**

It has a collection of the following things which are used to develop and run Java programs:

- (A) Applet viewer—enables to run Java applets
- (B) Javac (compiler)—the Java compiler which translates Java source code to byte code so that interpreter can understand
- (C) Java interpreter—which runs applets and applications by reading and interpreting byte codes
- (D) Javap—Java disk assembler enables us to convert byte code file into discs
- (E) JavaH—produces header files for use with native methods
- (F) Javadoc—creates HTML format documentation from Java source code file
- (G) Java debugger—helps to find errors
- (H) Java disk assembler

### **(3) JSL**

JSL includes hundreds of classes and methods divided into 6 major packages:

- (A) Language support package—collection of classes and methods required for implementing basic facial appearance of Java
- (B) Utility package—provides utility functions such as date and time
- (C) Input / output package—required for input and output manipulations
- (D) Net package—a collection of classes for communicating with other computers on network
- (E) AWT package—abstract window toolkit contains classes that implement platform independent GUI
- (F) Applet package—includes a set of classes that enables creating Java applets

## **19.4 OVERVIEW OF JAVA**

Programmers can develop two types of Java programs:

### **(1) STAND ALONE APPLICATIONS**

They are the programs written in Java to carry out certain task on stand-alone local computers. There are two steps:

- (i) Compiling source code into byte code using Java compiler
- (ii) Executing byte code using Java interpreter

### **(2) WEB APPLETS**

Applets are small Java programs from which we can develop simple animated graphics to complex games and utilities. We can also use Web enabled Java compatible browser like Internet explorer for running applets; if applet viewer is not available.

## **19.5 DIFFERENCES BETWEEN C/C++ AND JAVA**

Java does differ from C/C++ in many ways. Java was modeled after these languages but it does not incorporate many features of them. We will discuss here few points of differences between Java and C/C++.

### **(1) JAVA AND C**

There is a major difference between Java and C. Java is an object-oriented language whereas C is procedural language. In an effort to build simple and safe language many complicated parts of C are discarded in Java which are given below:

- Java does not support the pointer type.
- Java does not hold the C data types like `struct`, `union`, and `enum`.
- Java does not have the C keywords like `goto`, `sizeof`, and `typedef`.
- Java does not have preprocessors like C, and hence we cannot use `#define`, `#include`, and `#ifdef` statements.
- Java does not support any mechanism for defining variable arguments to functions.
- Java has added some new operators such as `instanceof` and `unsigned right-shift (>>>)`, conditional logical, etc.
- Java adds labelled `break` and `continue` statements.
- Java adds all the features required for object-oriented programming.

### **(2) JAVA AND C++**

While discussing the differences between Java and C++ we will have to be conscious because both of them are object-oriented programming languages. Hence, both of them incorporate the features of OOP's. Now, Java is a true object-oriented language while C++ and C an object-oriented extension. The differences are enlisted below:

- Java does not support operator overloading.
- Java does not make use of pointers like C++.
- There are no header files in Java.
- Java has replaced the `destructor` function with `finalize( )` function.
- Java does not support generic programming.
- Java does not support multiple inheritance classes.
- A new feature is added called as interface.
- Java does not support global variables.

**Table 19.1** Differences between C++ and JAVA

| C++                                           | Java                                                                  |
|-----------------------------------------------|-----------------------------------------------------------------------|
| C++ uses functions                            | Java uses <i>methods</i>                                              |
| In C++, members are by default <i>private</i> | In Java, <i>methods</i> are by default <i>public</i>                  |
| In C++, program is only <i>compiled</i>       | In Java, program is first <i>compiled</i> and then <i>interpreted</i> |

## 19.6 KEYWORDS

Some reserved words are kept in Java while designing, which can be used for some specific tasks only. These are called as Java's keywords. Keywords can be used only for that task for which they are designed; they will give error if used for other purposes. We cannot use these keyword names for variables, classes, methods etc. Java language has reserved 60 words. All keywords are written in lowercase letters, as Java is a case sensitive language. The keywords are given in Table 19.2.

**Table 19.2** Java keywords

|            |            |         |              |            |
|------------|------------|---------|--------------|------------|
| abstract   | boolean    | break   | byte         | byvalue    |
| case       | cast       | catch   | char         | class      |
| const      | continue   | default | do           | double     |
| else       | extends    | false   | final        | finally    |
| float      | for        | future  | generic      | goto       |
| if         | implements | import  | inner        | instanceof |
| int        | interface  | long    | native       | new        |
| null       | operator   | outer   | package      | private    |
| protected  | public     | rest    | return       | short      |
| static     | super      | switch  | synchronized | this       |
| threadsafe | throw      | throws  | transient    | true       |
| try        | var        | void    | volatile     | while      |

## 19.7 JAVA VIRTUAL MACHINE (JVM)

All language compilers translate source code into machine code for specific computer. Java compiler produces an intermediate code known as byte code, for a machine that does not exist but is known as Java virtual machine. It only exists inside the computer memory. It is a simulated computer within the computer and does all the major functions of real computer. Virtual machine

code is not an actual machine code. The machine specific code is generated by Java interpreter by acting as an intermediary between virtual machine and real machine.

## **19.8 STRUCTURE OF A JAVA PROGRAM**

A java program contains one or more sections given below:

### **(1) DOCUMENTATION SECTION (ESSENTIAL)**

It comprises a set of comment lines giving the name of program, author, and other details. We can give comments by using // or /\* \*/ or /\*\* \*/ formats.

### **(2) PACKAGE STATEMENTS (OPTIONAL)**

It declares package name and informs the compiler that the classes defined in program belong to this package.

### **(3) IMPORT STATEMENTS (OPTIONAL)**

This is similar to # include in C or C++.

### **(4) INTERFACE STATEMENTS (OPTIONAL)**

An interface is like a class but includes a group of method declarations and used only when we are dealing with multiple inheritance concept in program implementation.

### **(5) CLASS DEFINITIONS (OPTIONAL)**

Java programs contain multiple class definitions.

### **(6) MAIN METHOD CLASS (ESSENTIAL)**

It is the starting point of every standalone program.

## **19.9 IMPLEMENTING A JAVA PROGRAM**

Implementation of Java application involves three steps:

### **(1) CREATING A PROGRAM**

At the command prompt give edit test.java and type the following code.

#### **19.1 Write a simple Java program.**

```
class test
{
public static void main(String args[])
{
```

```
System.out.println("Hello");
}
}
```

## OUTPUT

Hello

**Explanation:** In the above program, the statement `System.out.println()` displays the message “Hello” on the screen. We save this file as `test.java`. The class name and file name should be the same. The Java program file should have `extension.java`. Java is case sensitive so wherever there is appearance of capital letter for any keyword then we must use that only and specify the name of the file according to class name.

## (2) COMPILING THE PROGRAM

We must follow the given format to invoke Java compiler:

`C:\javac test.java`

After much complications, a class file is created. For example, after above compilation `java.class` file is created.

## (3) RUNNING THE PROGRAM

We need to invoke here Java interpreter to run the program if there are no errors as given below:

`C:\java test`

Java is a pure OOP language, so everything should be placed in a class. The `public static void main(String args[])` line defines method named `main`, which is the starting point for an interpreter to begin execution. In Java, there should be only one `main` function but can have multiple classes. The ‘`public`’ keyword is an access specifier, which declares `main` method as unprotected and therefore making it accessible to all other classes. The `static` keyword declares the method as one that belongs to entire class and not a part of any objects of class. The `void` keyword is a type modifier that states that `main` method will not return any value. The fifth line prints a message. The `println` is a member of `out` object that is static data member of `System` class.

## 19.10 CONSTANTS, VARIABLES AND DATA TYPES

### CONSTANTS

Constants in Java refer to fix value that does not change during the execution of a program. Java supports several types of constants:

- (a) integer
- (b) real
- (c) single character
- (d) string
- (e) back slash '\' constants

## VARIABLES

A variable is an identifier that denotes a storage location used to store a data value. Variable names may consist of alphabets, digits, '\_', '\$' characters.

## DATA TYPES

Data types are broadly classified as primitive types called as intrinsic or built-in types and non-primitive types also known as derived types or reference types. Primitive data types may be numeric or non-numeric. The numeric data types are classified as integer and floating points. Non-numeric data types are classified as char and boolean. Non-primitive data types are classified as classes, interfaces, and arrays.

The data types determine the value and the operation that can be performed on the variable. For example, `int var`, defines variable `var` of type `integer`.

Java is a strongly typed language. This means every variable, expression has a type, and argument passing to the function is to check for type compatibility. The automatic type conversion is not allowed in Java as in other languages like C++.

In Java, there are two types of data types: primitive and reference.

A primitive type variable always contains only one value and its format is fixed. Examples of primitive type is a number, a character etc.

The following table shows all the primitive data types supported by the Java language.

## INTEGER

In Java, there are four types of integer types. All the four integer data types will have signed, positive and negative values. But unsigned and positive-only integers are not supported by Java language. [Table 19.3](#) shows the size and format of all the four integer data types.

**Table 19.3** Table to show size and format of integer types.

| Keywords | Size | Format |
|----------|------|--------|
|----------|------|--------|

|              |        |                  |
|--------------|--------|------------------|
| <b>byte</b>  | 8 bit  | Two's complement |
| <b>short</b> | 16 bit | Two's complement |
| <b>int</b>   | 32 bit | Two's complement |
| <b>long</b>  | 64 bit | Two's complement |

**byte** The byte is smallest in all the integer data types. The size of byte type is 8-bit. So in byte the minimum value you can represent is -128 and the maximum value is 127. The variables are declared using the keyword byte. The declaration syntax for byte type variables is:

```
byte var1, var2,...,varn;
```

The following example declares byte variable:

```
byte a;
```

**short** It is a signed 16-bit quantity. So the range is from -32,768 to 32,767. The following example shows the declaration of short types:

```
short a;
```

**int** The int is the most commonly used integer data type. It is represented as a signed 32-bit integer type and has a range from -2,147,483,648 to 2,147,483,647. They are mostly used in control loops and in arrays to represent indexes. The following example shows the declaration of int type:

```
int a;
```

**long** The last integer type is long, which is a signed 64-bit type. The range is from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. It is used to hold very large values.

## REAL NUMBERS

Real numbers are also known as floating point numbers. They are mainly used to hold numbers containing fractional part. For example, in the number 245.54, .54 is a fractional part. The floating point types are two in number:

1. float
2. double

The float type are single-precision numbers whereas double type are double precision numbers. The floating point numbers are usually double precision numbers. If you want to represent a single precision number, you must append f or F to the number. For example,

2.3567f or 2.356F

Floating point types support one special value, which is known as Not-a-Number(NaN). It is used to represent the result of operations, where an actual

number is not produced. For example, if you divide zero by zero, it will not produce any value.

The floating point types are discussed below:

**float** This is a single precision type and requires 32-bits for storage. It is required when a fractional component is needed. But it is not suitable to hold either very large or very small value. The following example shows the declaration of float variable:

```
float a;
```

**double** It is a double precision type and requires 64-bits for storage. Most of the mathematical functions, such as `sin( )`, `cos( )` return double values. When you are evaluating very large numbers, double type is used. **Table 19.4** gives the size and format of floating point types.

**Table 19.4** Size and format of floating point types

| Keyword | Size   | Format   |
|---------|--------|----------|
| float   | 32 bit | IEEE 754 |
| double  | 64 bit | IEEE 754 |

The following program shows the use of double type.

## 19.2 Write a program to show the use of double type.

```
class doubleType
class doubleType
{
public static void main(String args[])
{
    double length=12.25;

    double breadth=25.24;

    double arearect=length* breadth;

    System.out.println("The value of length is :" +length+ "cm");

    System.out.println("The value of breadth is :" +breadth+ " cm");
```

```
        System.out.println("The area of rectangle is :" + arearect +  
cm^2);  
    }  
}  
}
```

## OUTPUT

```
The value of length is :12.25cm  
The value of breadth is :25.24 cm  
The area of rectangle is :309.19 cm^2
```

**Explanation:** The above program calculates the area of rectangle. The variables `length` and `breadth` are defined as `double`. The value of area of rectangle is calculated and is displayed as `double`. If we define the variable `area` as `int`, then the fractional part of it is discarded, because `int` does not contain fractional part.

## CHARACTER TYPE

In Java, to store character constants, character type called `char` is used. The `char` in Java requires two bytes for storage as compared to `char` in C/C++ which needs only one byte because Java uses unicode to represent characters. The range of `char` is from 0 to 65,536. There is no negative `char`. The following program shows the behaviour of character type of variables.

### 19.3 Write a program to demonstrate character type variables.

```
class CharacterType  
{  
  
public static void main(String args[])  
{  
    char alphabet = 'A';  
    System.out.print(alphabet);  
  
    for (int i=0; i<25; i++)  
    {  
        alphabet++;  
        System.out.print(" " + alphabet); // Display alphabets  
        if (i==10)  
  
            System.out.println();  
  
    }  
}
```

## OUTPUT

```
A B C D E F G H I J K L  
M N O P Q R S T U V W X Y Z
```

**Explanation:** In the

program CharacterType.java, variable alphabet is defined as `char` type and is initialized as A. Note that the character is represented within single quotes (''). In `for` loop, the value of `alphabet` is incremented using `++` operator. The `++` operator is a unary operator since it is operated on single variable. The following program displays the ASCII value of character.

### 19.4 Write a program to demonstrate ASCII value of character.

```
class CharValue  
{  
    public static void main(String args[])  
    {  
        char charvalue='A';  
  
        int value=(int) charvalue;  
  
        System.out.println("The ASCII value of "+charvalue+"is: "+value);  
        value++;  
        charvalue=(char)value;System.out.println("The ASCII value  
"+value+" will have character: "+ charvalue);  
  
    }  
}
```

## OUTPUT

```
The ASCII value of Ais: 65
```

```
The ASCII value 66 will have character: B
```

**Explanation:** In the above program, `CharValue` displays the ASCII value of character variable. For example, in the above program variable is initialized as character A. Then its ASCII value is stored in the integer variable. When you assign the incompatible types, use typecasting.

## BOOLEAN TYPE

Boolean type takes only one of two possible values; true or false. It is used to test a particular condition and for performing logical operations. The keyword used to represent Boolean type is Boolean. The two values true and false have been declared as keyword in Java language. It uses only one bit for storage. Table 19.5 shows the size and format of character and Boolean types.

**Table 19.5** Size and format of character and boolean types

| Keyword | Size   | Format     |
|---------|--------|------------|
| char    | 16 bit | unicode    |
| boolean | 1 bit  | true/false |

The following program **BooleanTest** explains the application of boolean type.

### 19.5 Write a program to demonstrate Boolean values.

```
class BooleanTest
{
    public static void main(String args[])
    {

        int a=3,b=5;
        System.out.println("The value of a="+a);

        System.out.println("The value of b="+b);

        if(a<b) // conditional statement

        System.out.println("The condition(a<b) is:"+ (a<b)); // print the result as true or false

        else

        if(a>b)

        System.out.println("The condition (a>b) is:"+ (a>b));
    }
}
```

```
    }  
}
```

## OUTPUT

The value of a=3

The value of b=5

The condition (a<b) is : true

**Explanation:** The above program explains the application of Boolean values. The variables a and b are declared and initialized as int type. In the if statement, the values of variables a and b are compared. The result of if statement is either true or false. If the condition is satisfied then the result is Boolean value true and if the given condition is false, then the output of if statement is Boolean value false. In the above example, a=3 and b=5. Then the condition statement (a < b) is true and hence the output is Boolean value true, otherwise the result is Boolean value false.

## 19.6 Write a program to declare and initialize variables and display their values.

```
class program  
class program  
{  
    public static void main(String args[])  
{  
int a,b;  
a=-2,147,483,648;  
b=2,147,483,648;  
System.out.println("Value in a="+a);  
System.out.println("\n Value in b="+b);  
}  
}
```

## OUTPUT:

Value of a=-2,147,483,648 Value of b=2,147,483,648

**Explanation:** In the above program, a and b are integer variables. They are initialized with integer values. Using system.out.println( ) they are displayed on the screen.

## 19.7 Write a program to declare and initialize integer variables. Also display the value of variable.

```
class num  
{  
    public static void main (String args[])
```

```

{
    int n;
    n=458;
    System.out.println("n="+n);

    n=n+5;
    System.out.println("Now n="+n);
}
}

```

## **OUTPUT**

**n=458**

**Now n=463**

**Explanation:** In the above program, integer n is declared and initialized with 458. The value of n is displayed on the screen using System.out.println( ) function. 5 is added to variable n and new value is displayed.

## **19.11 CLASSES, OBJECTS AND METHODS**

A class is a user-defined type which holds both data and methods. The internal data of a class are called as data members and the functions are called as member functions or methods. The variables of a class are called as objects.

Class can be declared as:

```

class classname [ extends superclass name ]
{
variable declarations;
method declarations;
}

```

Here, the keyword extends the properties of superclass to the classname class. This concept is known as inheritance.

**Example**

```

class area
{
    double r;
}

```

The class area contains one member variable of double type.

## **CREATING OBJECTS**

An object in Java is essentially a block of memory that contains space to store all instance variables. Objects in Java are created by using new operator.

```

classname objectname;
objectname=new classname( );

```

### **Example**

```
area a;  
a=new area( );
```

First statement declares a variable to hold the object reference and second actually assigns the object reference to the variable. Both the statements can also be combined in one sentence as

```
area a=new area(0);
```

### **19.8 Write a program to define class. Declare objects.**

```
class area  
{  
float r;  
}  
  
class classdemo  
{  
public static void main (String args[])  
{  
double ar;  
area a= new area( );  
a.r=4;  
ar=3.14*a.r*a.r;  
System.out.println("Area : "+ar);  
}  
}
```

### **OUTPUT**

**Area: 50.24**

**Explanation:** In the above program, class area is defined with member variable r. An object of class area is defined. Using object member variable, r is initialized to four and area is calculated and displayed.

Methods can be added as

```
type methodname(parameter list)  
{  
method body;  
}
```

### **19.9 Write a program to define method in a class and display the area.**

```
class area  
{  
float r;  
  
void show( )
```

```

        {
            System.out.print("Area : ");
            System.out.println(3.14*r*r);
        }
    }

class classdemo
{
    public static void main(String args[])
    {
        area a= new area();
        a.r=4;
        a.show();
    }
}

```

## **OUTPUT**

**Area: 50.24**

**Explanation:** This program is same as the last one. Here, method `show()` is defined which displays the area. The method `show()` is invoked using object `a`.

## **19.12 OVERRIDING METHODS**

There may be certain occasions when we want an object to respond to the same method but have different behaviour when that method is called. That means we should override methods in super class. This is possible by defining a method class that has same name, same argument and return type as a method of superclass. When that method is called, the method defined in subclass is invoked and executed instead of the one in superclass. This is known as overriding.

### **19.10 Write a program to overload a member function.**

```

class overload
{

    void show()
    {   System.out.println("Without parameters");   }
}

```

```

void show( int x)
{
    System.out.println("With one parameter");

    System.out.println(" x = "+x);
}

}

class over
{
    public static void main ( String args[])
    {

        overload o= new overload();
        o.show();
        o.show(9);

    }
}

}

```

## **OUTPUT**

**Without parameters With one parameter**

**x = 9**

**Explanation:** In the above program, the class `overload` contains two methods with same name. The `show( )` method is overloaded with and without parameter. When a method `show( )` is invoked without parameter, the function body without parameter is executed. When one integer is passed the function body with one parameter is executed.

## **19.13 MULTITHREADING**

Multithreading is a conceptual programming paradigm where a program process is divided into two or more subprograms (processes) which can be implemented at the same time in parallel. For example, one subprogram displays animation while other one builds another animation. This is similar to dividing task into subtasks and assigning them to different processes for execution independently and simultaneously. Thread is similar to a program

that has a single flow of control. It has a beginning, body, and an end, which execute commands sequentially. Our simple programs are single threaded programs. A program that contains multiple flows of control is known as multithreaded program.

Multithreading is a powerful programming tool and distinct from other programming languages. It enables a programmer multiple things at one time. Longer programs can be divided into threads and executed in parallel. For example, we can send task such as printing into the background and continue to perform some other task in foreground.

## 19.14 LIFE CYCLE OF THREAD

There are many states thread can enter during life time:

1. new thread state
2. runnable
3. running
4. blocked state (optional)
5. dead state

There are certain methods that are used to implement threads.

Threads are implemented in the form of objects that contain a method called as `run( )` which is heart and soul of any thread. Whenever we want to start a thread we can use `start( )` method. If we want to stop a thread from running further we may do so by calling `stop( )` method. A thread can also be temporarily suspended or blocked as :

1. `sleep( )` blocks for a specified time.
2. `suspend( )` blocks until further orders.
3. `wait( )` blocks until certain condition occurs.

## 19.11 Write a program to demonstrate multithreading.

```
Class A extends thread
{
public void run( )
{
for(i=1;i<=5;i++)
{
if(i==1)
yield( );
System.out.println("From thread A : i= "+i);
}
System.out.println("Exit from A");
}
Class B extends thread
{
```

```

public void run( )
{
for(j=1;j<=5;j++)
{
System.out.println("From thread A : j= "+j);
if(j==3)
stop();
}

System.out.println("Exit from B");
}
Class C extends thread
{
public void run( )
{
for(k=1;k<=5;k++)
{
System.out.println("From thread C : k= "+k);
if(k==1)
try
{
sleep(1000);
}
catch(Exception e)      {      }
}
System.out.println("Exit from C");
}
class threadmethods
{
public static void main(String args[])
{
A threadA=new A();
B threadB=new B();
C threadC=new C();
System.out.println("Start thread A");
threadA.start();
System.out.println("Start thread B");
threadB.start();
System.out.println("Start thread C");
threadC.start();
System.out.println("End of main thread");
}
}

```

## **OUTPUT:**

**Start thread A**

**Start thread B**

**Start thread C**

```
From thread B : j=1
From thread B : j=2
From thread A : i=1
From thread A : i=2
End of main thread
From thread C : k=1
From thread B : j=3
From thread A : i=3
From thread A : i=4
From thread A : i=5
Exit from A
From thread C : k=2
From thread C : k=3
From thread C : k=4
From thread C : k=5
Exit from C
```

## 19.15 PACKAGES

To use classes from other programs without physically copying them into the program under development, can be accomplished in Java by using packages similar to class libraries in other languages. Packages are Java's way of grouping a variety of classes and/or interfaces together. The grouping is usually done according to functionality. In fact packages act as containers for classes. Packages provide a way to hide the classes, thus preventing other programs or packages from accessing classes that are meant for internal use only. Java packages are of two types:

**1. Java system packages** - Java system provides a large number of classes grouped into different packages according to functionality as:

- (A) Java.lang – language support classes.
- (B) Java.util – language utility classes.
- (C) Java.io – input/output support classes.
- (D) Java.awt – set of classes used to implement GUI.
- (E) Java.net – classes for networking.
- (F) Java.applet – classes for creating & implementing applets.

**2. User defined packages**—These are defined by users for their use.

## 19.16 APPLETS

Applets are small Java programs that are primarily used in Internet computing. They can be transported over Internet from one computer to another computer and run using applet viewer or any Java compatible Internet browser. It can perform arithmetic operations, display graphics, play sound, accept user input, create animation, and play interactive games.

### **(1) LOCAL AND REMOTE APPLETS :**

We can embed applets into web pages in two ways:

**Local** We can write our own applet and embed them into web pages.

**Remote** We can download an applet from remote system and then embed it into web pages.

### **(2) PREPARING TO WRITE APPLETS :**

Before we try to write applet we make sure that Java is properly installed and also have applet viewer / Java enabled browser. The steps are given below.

1. Building an applet code (.java file)
2. Creating an executable file (.class file)
3. Designing a web page using HTML tags
4. Preparing applet tag
5. Incorporating applet tag into web page
6. Creating HTML tag
7. Testing applet code

### **(3) BUILDING APPLET CODE**

It is essential that our applet code uses services of class namely graphics and applets from Java class library. Applet class contained in java.applet class provides life and behavior to the applet through its methods such as `init()`, `start()`, `paint()`. The `paint` method displays result of applet code on the screen. Output may be text or sound. The `paint()` requires a graphic object as an argument defined as:

```
public void paint(Graphics g)
```

This requires the applet code import `java.awt` that contains `graphics` class. The general applet code format is given below.

```
import java.awt.*;
import java.applet.*;
public class appletclassname extends Applet
{
    public void paint(Graphics g)
    {
    }
}
```

the `appletclassname` through a main class for the applet.

## **19.12 Write a program to create an applet.**

```
//Hello Java ! applet program
import java.awt.*;
import java.applet.*;
public class hellojava extends Applet
{
public void paint(Graphics g)
{
g.drawString("Hello Java !",10,100);

}
```

### **OUTPUT**

The output will be Hello Java ! in the applet viewer.

We have to add applet tag for the execution of applet as :

```
<APPLET
    CODE=hellojava.class
    WIDTH=400
    HEIGHT=400>
</APPLET>
```

### **SUMMARY**

(1) A platform is a hardware or software environment in which a program runs. The computers, nowadays, are interconnected by different networks worldwide, and operate on many platforms, Microsoft Windows, Macintosh, OS/2, UNIX and NetWare.

(2) Hot Java is a web browser that runs applet on Internet. Hence, Java is strongly interconnected with the Internet.

(3) World Wide Web is designed for use by the Internet's distributed environment. It contains information and controls about the web pages. It is not like menu driven system. WWW is open through any direction.

(4) All language compilers translate source code into machine code for specific computers. Java compiler produces an intermediate code known as byte code for a machine that does not exist, and is known as Java virtual machine.

(5) Data types are broadly classified as: Primitive types called as intrinsic or built-in types and Non-primitive also known as derived types or reference types. Primitive data types may be numeric or non-numeric. The numeric data types are classified as integer and floating point. Non-numeric data types are classified as char and Boolean. Nonprimitive data types are classified as classes, interfaces, and arrays.

(6) Multithreading is a conceptual programming paradigm where a program process is divided into two or more subprograms (processes) which can be implemented at the same time in parallel.

(7) To use classes from other programs without physically copying them into the program under development, can be accomplished in Java by using packages similar to class libraries in other languages. Packages are Java's way of grouping a variety of classes and/or interfaces together.

(8) Applets are small Java programs that are primarily used in Internet computing. They can be transported over Internet from one computer to another computer and run using applet viewer or any Java compatible Internet browser.

### EXERCISES

**[A] Answer the following questions.**

- (1) Explain evolution of Java.
- (2) Explain Java technology.
- (3) Explain structure of Java program.
- (4) What are static members and functions?
- (5) What do you mean by overriding methods?
- (6) Explain multithreading programming in Java.
- (7) What are packages and applets?