MADHYA PRADESH BHOJ(OPEN) UNIVERSITY BHOPAL

P.G.DIPLOMA IN AI & DATA SCIENCE

Object Oriented Programming

OBJECT ORIENTED PROGRAMMING USING C++

मध्यप्रदेश भोज (मुक्त) विश्वविद्यालय – भोपाल

MADHYA PRADESH BHOJ (OPEN) UNIVERSITY – BHOPAL

COURSE WRITERS

Rohit Khurana, Faculty and Head, I.T.L. Education Solutions Ltd., New Delhi
Units (1.0-1.1, 1.2-1.2.1, 1.2.3-1.2.8, 1.3 -1.5, 1.7-1.13, 2.0, 2.1, 2.2- 2.2.1, 2.2.2, 2.3-2.4, 2.5-11, 3.0-3.4, 3.6-3.10, 4.0-4.2.6, 4.3-4.5, 4.7-4.12, 5.2, 5.3.3-5.3.6, 5.5-5.6.3, 5.7-5.12)

R. Subburaj, Former Professor and Consultant, S.R.M. University, Chennai
Units (1.2.2, 1.6, 3.5, 4.6, 5.0-5.1, 5.3-5.3.2, 5.4)

SYLLABI-BOOK MAPPING TABLE
Object Oriented Programming Using C++

Syllabi	Mapping in Book
UNIT - 1: **Object Oriented Paradigms and Metaphors:** Basic Concepts of Object-Oriented Programming, Objects, What is C++?, A Simple C++ Program, Initialization Input with C in Tokens, Control Statements, Decisions Nesting, Type Conversion. **Data Types:** Operators and Expressions, Tokens, Basic Data Types, Constants, User Defined Data Types, Derived Data Types, Declaration of Variables, Operations and Expressions, Operator and Function Overloading, Manipulation of Strings Using Operators, Polymorphism Streams.	**Unit-1:** Object Oriented Paradigms, Metaphors and Data Types **(Pages 3-88)**
UNIT - 2: **Function in C++:** The Main Function Passing Arguments to Function Returning Values From Functions Overload Functions Inline Functions Default Arguments. Class and Objects: The Concept of a Class, Classes Versus Objects.	**Unit-2:** Functions, Class and Objects in C++ **(Pages 89-122)**
UNIT - 3: **Constructor and Destructor:** Constructors, Destructors, Constructors of the String Class, String Class Assignment, String Access Operators and Method. Operator Overloading, Type Casting	**Unit-3:** Constructor and Destructor, Operator Overloading and Type Casting **(Pages 123-169)**
UNIT - 4: **Inheritance:** Derived Class, Relationships Superclass/Subclass, Multiple Inheritance, Constructors, Destructors and Inheritance, Hierarchical Inheritance Hybrid Inheritance, Virtual Base Classes. What Are Pointers? C++ Memory Map, Free Store, Pointers and Arrays, Reserving Dynamic Memory, Freeing Dynamic Memory, Polymorphism, Virtual Functions, Pure Virtual Functions, Early vs. Late Binding.	**Unit-4:** Inheritance and Pointers **(Pages 171-225)**
UNIT - 5: **Input-Output in C++:** Old vs. Modern C++I/O, C++ Streams, Creating Inserters, Creating Extractors, Creating Manipulator, Functions. **File Handling in C++:** Classes for File Stream Operations, Opening and Closing a File, Manipulations of File Pointers, Random Access, Command-Line Arguments, Standard Library Objects, The Container Classes, Theory of Operation, Vectors, Lists, Maps, Algorithms, The String Class.	**Unit-5:** Input-Output and File Handling in C++ **(Pages 227-317)**

CONTENTS

INTRODUCTION

Object Oriented Programming (OOP) has emerged as one of the most impressive programming paradigms in software development. Today, C++ is one of the most popular OOP languages which can be used to develop real-world applications. C++ is a programming language with a heritage extending from the ubiquitous C language by Bjarne Stroustrup in the 1980s. It treats data as a crucial element— not allowing it to move freely around the system. Therefore the main emphasis in C is on data and not on the procedure. You can design programs around the data being operated upon in C++. An object oriented language helps in combining data and functions that operate on data into a single unit known as object. The process of combining data and functions into a single unit is referred to as encapsulation.

C++ is used for developing different types of applications, such as real-time systems, simulation modelling, expert systems. It also provides flexibility to a user to introduce new types of objects in his programming on the basis of the requirement of the application. This feature is known as abstraction. Features like abstraction, object oriented programming and generic programming make C++ useful to those who wish to undertake computer programming.

This book, *Object Oriented Programming using C++* is divided into five units that follow the self-instruction mode with each unit beginning with an Introduction to the unit, followed by an outline of the Objectives. The detailed content is then presented in a simple but structured manner interspersed with Check Your Progress Questions to test the student's understanding of the topic. A Summary along with a list of Key Terms and a set of Self-Assessment Questions and Exercises is also provided at the end of each unit for recapitulation.

UNIT 1 OBJECT ORIENTED PARADIGMS, METAPHORS AND DATA TYPES

Structure

1.0 INTRODUCTION

The goal of programmers is to develop software that are correct, reliable and maintainable, and satisfy all the user requirements. Software development is not a static process. The software needs to be modified or redesigned according to change in user requirements, business rules and strategies. In addition, the complexity of the software also increases. To cope with the dynamic nature and complexity of the software, different approaches of programming have been developed since the invention of the computer. These approaches are known as programming paradigms. To understand the concept of object-oriented programming, it is necessary to know the fundamental terms and concepts of this approach. These include objects, classes, data abstraction, encapsulation, inheritance, polymorphism, and message passing.

A data type determines the type and the operations that can be performed on the data. C++ provides various data types and each data type is represented differently within the computer's memory. Data types that are derived from the built-in data types are known as derived data types. The various user-defined data types provided by C++ are structures, unions, enumerations and classes. A token is defined as the smallest unit of a program. When a program is compiled, the compiler scans the source code and parses it into tokens to find the syntax errors. C++ tokens are broadly classified into keywords, identifiers, constants, operators and punctuators.

C++ has an interesting feature called function overloading. By this feature, we can build a number of functions with the same name. But the argument lists of such functions have to be different and unique. In C++, the stream refers to the stream of characters that are transferred between the program thread and I/O (Input/Output). Stream classes in C++ are used to input and output operations on files and I/O devices. These classes include specific features for handling input and output of the program. The iostream.h library holds all the stream classes in the C++ programming language.

In this unit, you will study about the object oriented paradigms and metaphors, basic concepts of objected –oriented programming, C++ and simple C++ program, initialization input with C in tokens, control statements, decisions nesting and type conversion, operators and expressions, tokens and basic data types, constants and user define data, data types derived and declaration of variables, operator and function overloading, manipulation of strings using operators, polymorphism and streams in C++.

1.1 OBJECTIVES

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

- Understand the basics of object oriented paradigms and metaphors
- Explain the basic concept of object-oriented programming
- Explain the initialization input with C in tokens
- Describe the role of control statements in C++
- Analyse the decisions nesting and type conversion
- Discuss the meaning of operators and expressions
- Elaborate on the tokens and basic data types
- Discuss constants and user defined data
- Explain the operator and function overloading
- Describe the manipulation of strings using operators
- State the concept of polymorphism and streams in C++

1.2 OBJECT ORIENTED PARADIGMS AND METAPHORS

Initially, when the computers were invented, binary language was used to write the programs. However, as the programs grew in size, it became difficult to write programs using binary language. Then the assembly language was invented to write large programs, however, it was also not user-friendly. With the change in the user requirements, the size and the complexity of the programs continued to grow, which led to the development of high-level languages, such as BASIC and FORTRAN. However, these languages provided an unstructured way of writing programs. In unstructured programming paradigm, all the instructions of a program were written one after the other in a single function and hence, suitable for writing only small and simple programs. For large and complex programs, it became difficult to trace and debug errors.

To overcome the limitations of unstructured programming paradigm, other programming paradigms, namely, procedural and object-oriented programming paradigms were developed, which help the programmers to develop the programs in structured way.

Features of Object-Oriented Programming

Object-Oriented Programming (OOP) paradigm has revolutionized the process of software development. It not only includes the best features of structured programming, but also introduced some new and advanced features that the procedural programming lacked. The most important feature is that, unlike procedural programming in which the program is divided into a number of functions, OOP divides the program into a number of objects. An object is a unit of structural and behavioral modularity that contains a set of properties (or data) as well as the associated functions. In addition, programmers can create relationships between one object and another.

The functions of the object (also known as member functions) provide the only way to access the object's data. If the user wants to read or manipulate any data item, then it is possible only if the member function to do the same is available in the object. Therefore, the data is hidden from the outside world, and hence safe from accidental modifications. The basic idea behind OOP is shown in Figure 1.1.
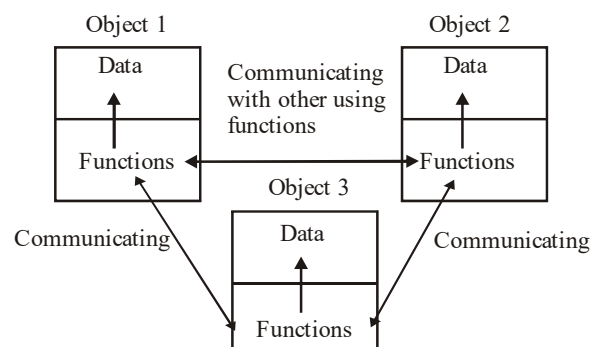
Fig. 1.1 *Data and Functions in OOP*

Some of the other features of OOP are as follows:

- OOP emphasises on data rather than the functions or the procedures.
- OOP models the real world very well by binding the data and associated functions together under a single unit and thus, prevents the free movement of data from one function to another.
- The data of one object can be accessed by the associated functions of that object only. Other functions are not allowed to access that data. In other words, data is hidden from the outside world. However, the functions of one object can access the functions of other object.
- The objects of the entire system can interact with each other by sending messages to each other.
- The programs written in OOP are easy to maintain and extend because new objects can be easily added to the existing system whenever required without modifying the other objects.
- OOP follows the bottom-up approach for designing the programs. That is, first objects are designed and then these objects are combined to form the entire program.

These new OOP features have tremendously helped in the development of well-designed high-quality software.

1.2.1 Basic Concepts of Object-Oriented Programming

To understand the concept of object-oriented programming, it is necessary to know the fundamental terms and concepts of this approach. These include objects, classes, data abstraction, encapsulation, inheritance, polymorphism, and message passing.

Objects

Objects are the small, self-contained and modular units with a well-defined boundary. An object consists of a state and behavior. The **state** of an object is one of the possible conditions that an object can exist in and is represented by its characteristics or attributes or data. The behavior of an object determines how an object acts or behaves and is represented by the operations that it can perform. In OOP, the attributes of an object are represented by the variables and the operations are represented by the functions.

For example, an object Biscuit may consist of data product code P001, product name *Britania Biscuits*, price ₹ 20 and quantity in hand 50. These data values specify the attributes or features of the object. Similarly, consider another object Maggi with product code P002, product name *Maggi Noodles*, price ₹ 10, and quantity in hand 20 (Refer Figure 1.2). In addition, the data in the object can be used by the functions, such as `check_qty()` and `display_ product()`. These functions specify the actions that can be performed on data.

Objects are what actually runs in the computer and thus, are the basic run-time entities in object-oriented systems. They are the building blocks of object-oriented programming. Although, two or more objects can have same attributes,

still they are separate and independent objects with their own identity. In other words, all the objects in a system take a separate space in the memory independent of each other. It must be noted that the main objective of breaking down complex software projects into objects is that changes made to one part of software should not adversely affect the other parts.

Classes

A class is defined as a user-defined data type which contains the entire set of similar data and the functions that the objects possess. In other words, a class in OOP represents a group of similar objects. As stated earlier, in the real world millions of objects exist and each of them has its own identity. However, each of them can be categorized under different groups depending on the common properties they possess and the functions they perform. For example, cars, scooters, motorbikes, buses, etc., all can be grouped under the category vehicles. Similarly, dogs, cats, horses, etc., can be grouped under the category animals. Thus, vehicles and animals can be considered as the classes.

A class serves as a blueprint or template for its objects. That is, once a class has been defined, any number of objects belonging to that class can be created. The objects of a class are also known as the instances or the variables of that class and the process of creating objects from a class is known as instantiation. Note that a class does not represent an object; rather it represents the data and functions that an object will have.

For example, a class Product consists of data such as `p_code, p_name, p_price` and `qty_in_hand`, which specify the attributes or features of the objects of the Product class. In addition, it consists of functions, such as `display_product()` and `check_qty()` that specify the actions that can be performed on data. (Refer Figure 1.2).
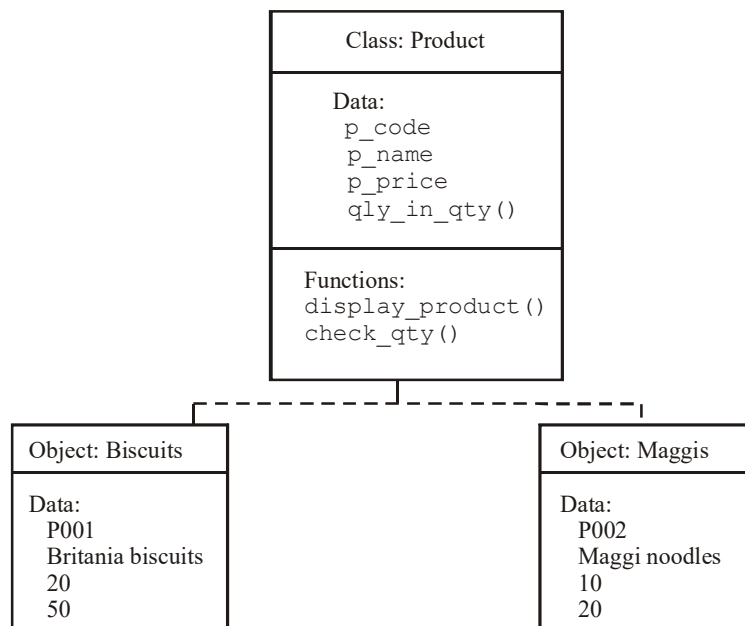
Fig. 1.2 *Class and its Objects*

Note that the data belonging to a particular class is known as its data members and the functions of the class are known as the member functions and both collectively are known as the members of the class.

Abstraction

Abstraction is a mechanism to hide irrelevant details and represent only the essential features, so that one can focus on important things at a time. It allows managing complex systems by concentrating on the essential features only. For example, while driving a car, a driver only knows the essential features to drive a car, such as how to use clutch, brake, accelerator, gears, steering, etc., and least bothers about the internal details of the car like motor, engine, wiring, etc.

Abstraction can be of two types, namely, data abstraction and control abstraction. Data abstraction (also known as data hiding) means hiding the details about the data and control abstraction means hiding the implementation details. In object-oriented programming, one can abstract both data and functions. However, generally, the classes in OOP are defined in such a way that the data is hidden from the outside world and the functions form the public interface. That is, the functions of the class can be directly accessed by other functions outside the class, and the hidden data can be accessed indirectly with the help of these functions.

Note that the values of the hidden data members cannot be passed to the outside world unless the functions are written to pass that information outside the class. Since the internal details of the class are hidden from the outside world, the data abstraction ensures security of data by preventing it from accidental changes or manipulations by other parts of the program.

Note: Classes in the object-oriented programming are also known as Abstract Data Types (ADT) as they use the concept of abstraction.

Encapsulation

Encapsulation is the technique of binding or keeping the data and functions (that operate on them) together in a single unit called a class. Encapsulation is the way to implement data abstraction. A well-encapsulated object acts as a 'Black Box' for other parts of the program. That is, it provides services to the external functions or other objects that interact with it. However, these external functions or the objects do not need to know its internal details. For example, in Figure 1.2 the data `p_code`, `p_name`, `p_price` and `qty_in_hand` and the functions `display_product()` and `check_qty` are encapsulated in a class Product.

Inheritance

Inheritance can be defined as the process whereby an object of a class acquires characteristics from the object of another class. As stated earlier, all the objects of a similar kind are grouped together to form a class. However, sometimes a situation arises when different objects cannot be combined together under a single group as they share only some common characteristics. In this situation, the classes are defined in such a way that the common features are combined to form a generalized class and the specific features are combined to form a specialized class. The specialized class is defined in such a way that in addition to the individual

characteristics and functions, it also inherits all the properties and the functions of its generalized class.

For example, in the real world, all the vehicles cannot be automobiles—some of them are pulled-vehicles as well. Thus, car and scooter both are vehicles that come under the category of automobiles. Similarly, rickshaw and bicycle are the vehicles that come under the category of pulled-vehicles. Thus, automobiles and pulled-vehicles inherit the common properties of the vehicle class and also have some other properties that are not common and differentiate them. Thus, the vehicles class is the generalization of automobiles and pulled-vehicles class, and automobiles and pulled-vehicles classes are the specialized versions of the vehicles class. It must be noted that while inheriting the vehicle class, the automobiles and pulled-vehicles do not modify the properties of the vehicle class, however, can add new properties that are exclusive for them (Refer Figure 1.3).
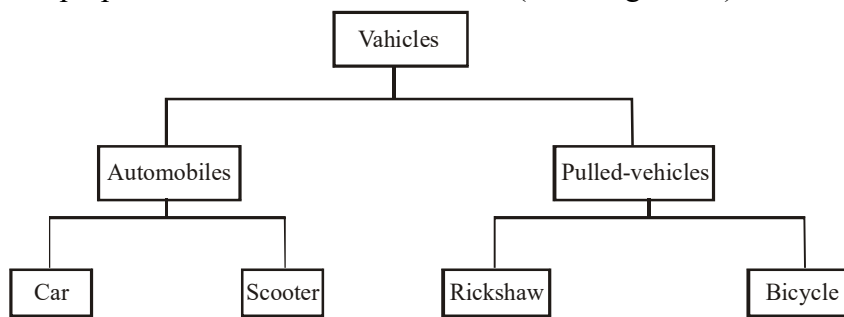
Fig. 1.3 *Inheritance*

In the same way, OOP allows one class to inherit the properties of another class or classes. The class, which is inherited by the other classes, is known as superclass or base class or parent class and the class, which inherits the properties of the base class, is called sub class or derived class or child class. The subclass can further be inherited to form other derived classes. For example, in Figure 1.3, car and scooter are the derived classes of automobiles and rickshaw and bicycle are the derived classes of pulled-vehicles.

Inheritance can be of two types, viz., single inheritance and multiple inheritance. If a class acquires properties from a single class, it is termed as single inheritance and if it acquires characteristics from two or more classes, it is known as multiple inheritance. The main advantage of inheritance is reusability. The existing classes can be simply reused in new software instead of writing a new code. Moreover, new features can be added without altering or modifying the features of the existing class.

Reusability and Extensibility

Inheritance allows code reusability, that is, it facilitates classes to reuse the existing code. It is useful when several classes having similar features are to be created. In such a case, one class is created having common features of all the classes, which is used as the base class. Whenever a new class is to be generated, it inherits this base class and only the unique features of new class are added, thereby avoiding repetition of code. The new class acquires the members of the old class that are already tested and debugged.

The base classes having common features can also be stored in reservoir so that they can be used by any programmer. These classes stored in reservoir form part of general-purpose programming tools and new classes generated on the basis of these classes become their specialized versions. Hence, inheritance allows extending and reusing already existing classes, thereby, saving time as well as increasing the reliability. For example, a common class `employee` can be created having some basic features, which can be used by any program requiring classes (like, `clerk,   manager,` `part_time_employee,` `full-time_employee,` etc.) to be generated having similar features. These features of inheritance play an important role in the program development.

Abstract Classes and Concrete Classes

While inheriting a base class, a derived class not only inherits the data and functions of its base class, but can also provide a different implementation (definition) for the functions of the base class. In such a case, the base class may or may not provide an implementation for its function. It only provides the interface for the functions.

A class which provides only the interface of one or more functions and not their implementations is known as an abstract class. An abstract class only specifies what the function does, what all it requires, etc., but it does not specify how the function works. Implementations of such functions are provided in the classes that inherit the abstract class. Note that the instances (objects) of an abstract class cannot be created. This is because it does not provide the implementation of the functions. The class that provides an implementation for all its functions is known as a concrete class. The concrete classes can have one or more objects. Note that derived classes that provide implementation of all the functions that have not been implemented in the abstract class are also considered as concrete classes.

Polymorphism

Polymorphism (a Greek word meaning having multiple forms) is the ability of an entity such as a function or a message to be processed in more than one form. It can also be defined as the property of an object belonging to a same or different class to respond to the same message or function in a different way. For example, if a message change_gear is passed to all the vehicles then the automobiles will respond to the message appropriately; however, the pulled vehicles will not respond. The concept of polymorphism plays an important role in OOP as it allows an entity to be represented in various forms.

In C++, polymorphism can be achieved either at compile-time or at run-time. At compile-time, polymorphism is implemented using operator overloading and function overloading. However, at run-time, it is implemented using virtual functions (Refer Figure 1.4).
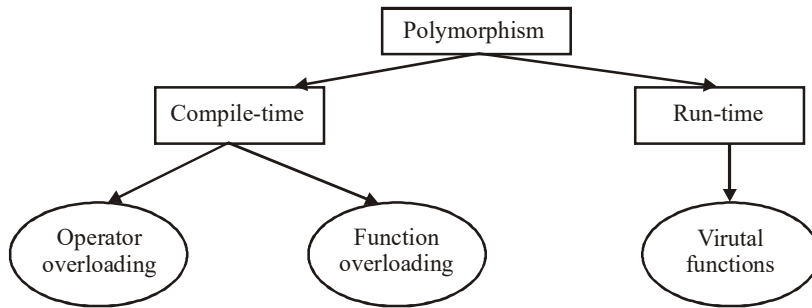
Fig. 1.4 *Different Ways of Implementing Polymorphism*

Operator overloading is the process that enables an operator to exhibit different behavior, depending on the data provided. For example, when the '+' operator is used with two numbers, it adds the two numbers and produces the sum. However, if it is used with two strings, it concatenates the two strings and produces the third concatenated string (Refer Figure 1.5).
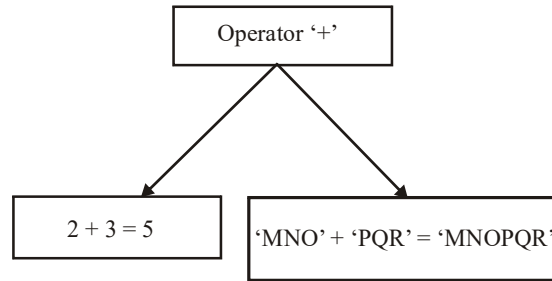
Fig. 1.5 *Operator Overloading*

Similarly, a single function can behave differently depending on the type of data provided. For example, in Figure 1.6, the function 'Add' can be used to add two integers and two float-point numbers. This form of polymorphism is known as function overloading.

Fig. 1.6 *Function Overloading*

Note: Compile-time polymorphism is also known as static binding as the linking of function call to the actual code of the function is done at compile-time itself.

Consider another example in which three different classes, square, rectangle and `circle` are derived from the base class `geometrical_shapes`. The function `area()` of the base class is implemented in different ways in all its derived classes and a call to a particular function is determined at run-time. This form of polymorphism is called run-time polymorphism (Refer Figure 1.7).
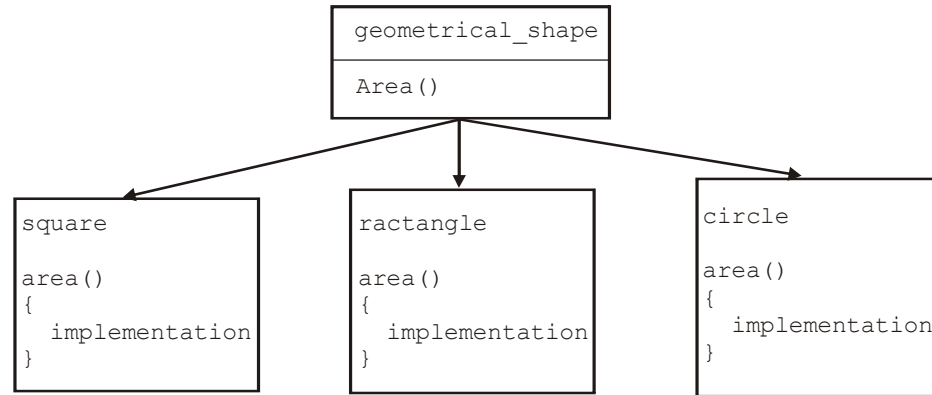
Fig. 1.7 *Run-Time Polymorphism*

Message Passing

Message passing is a process of interacting between different objects in a program. As discussed earlier, a program following the object-oriented paradigm comprises a set of objects each with a set of data and functions. When the program is executed, these objects interact or communicate with each other by sending and receiving messages. The messages are exchanged by calling the member functions of the classes.

Any object of a class that wants to communicate with the object of another class requests the object to invoke the required member function of its class. This function call is different from the normal function call as in this case the sending object is sending a request for the execution of the function. However, the receiving object may or may not accept the request depending on whether the function forms the public interface or it is hidden from the outside world. Thus, this form of communication is called message sending and not an ordinary function call.

For example, consider two classes *Product* and Order. The object of the Product class can communicate with the object of the Order class by sending a request for placing order (Refer Figure 1.8).
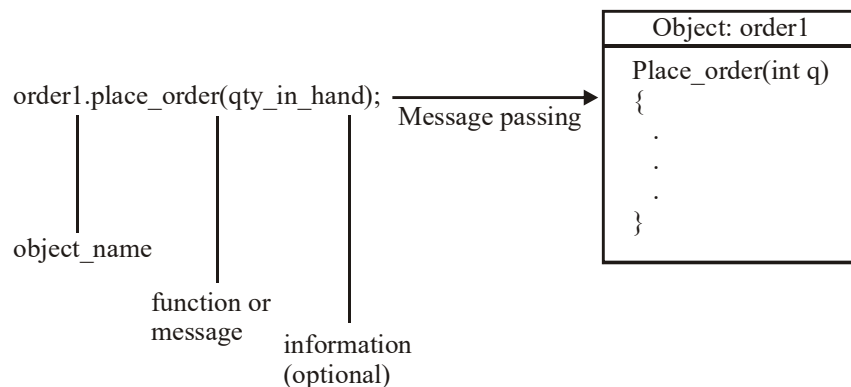
Fig. 1.8 *Message Passing*

1.2.2 Objects

In our day-to-day life, we come across a number of objects. Some examples are: PC, television set, radio receiver, telephone, table and car. An object will have the following two characteristics:

- State or Attributes
- Behaviour or operations

The 'State or Attributes' refers to the built-in characteristics of an object. They remain the same unless disturbed or modified. For example, a color television set has the following attributes:

- Color Receiver
- 64 Channels
- Volume and Picture Controls
- Remote Control unit

The 'Behaviour or Operations' of an object refers to its action. It can also be explicitly defined. The object television set can behave in any of the following manner at a given point of time:

- Switched on
- Switched off
- Displays picture and sound from
    - a TV transmitter
    - a cable TV connection
    - a VCR

Software objects can be visualized in a similar manner. They also possess one or more states and a particular behaviour at a given point of time.

Software Objects

One of the most interesting features of OOP is that software objects correspond to real objects. Software objects are made of data and functions, tied together. Let us represent an object pictorially. (Refer Fig. 1.9)
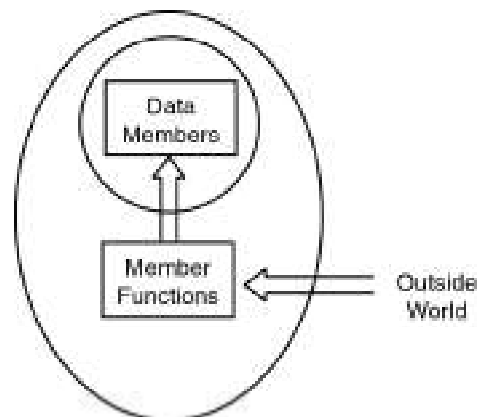
Fig. 1.9 Basic Representation of an Object

An object definition consists of the following:

- Data Members
- Member Functions

The data members establish the state or attributes for the object. The behaviour of the object is determined by the member functions. The state of the object can be changed through the member functions. The data members contain data. Thus, data is the nucleus of the object. The diagram illustrates the concept of objects clearly. The nucleus i.e., data can only be accessed through the member functions which are nothing but interfaces to access or manipulate data. The data is thus well-protected and inadvertent manipulation thereof can be prevented. The interface or function is the window to the outside world for manipulating the data. OOP is all about creating useful programs with the help of objects.

1.2.3 What is C++

C is a procedure-based language. Once you write a program in C, you must run it through a C compiler to turn your program into one that a computer can run (execute). C allows the input and output control in which a user can input the value to get the desired result. This language makes hardware devices easy to access and is used to manipulate individual bits in hardware registers. C++ is an object-oriented language that uses various concepts in handling of programs, such as virtual functions, multiple inheritance, exception handling, and polymorphism. Its object-oriented structure allows the code that can be reused and hence it cuts down the development time. C++ provides comprehensive coverage of abstract classes as interfaces, regular error handling, standard strings, I/O streams, etc. Table 1.1 explains the difference between C and C++ computer languages:

Table 1.1 *Differences between C and C++*

C	C++
C is procedure and function-oriented language and gives importance to procedure (functions) rather than data.	C++ is object-oriented language and gives importance to data.
C provides `scanf()` function to input the values and `printf()` function to display the result.	C++ provides `cin` object of class `istream` to input the values and `cout` object to display the result.
C does not follow the class and object concept.	C++ supports object and class for data encapsulation, data abstraction and polymorphism concept.
In C, macros are used.	In C++, an `inline` function is used instead of macros.
C supports pointers that basically refer to record and track the memory address or location of function.	This language supports GUI programming feature on a computer.
It is used especially in game programming and faster than C++.	It is well-suited to platform dependent applications.
It is structured language and use extensively pointers for memory, array, structures and functions.	It is complex for very large high level programs and difficult to debug the web applications.
It has no runtime checking and is case sensitive language.	It is also case sensitive language.

C does not support `new` and `delete` keywords. The memory operations to free or allocate memory in C are carried out by `malloc()` and `free()`functions. In C, `malloc()` function is used to allocate the memory. You can use this function to allocate memory in the following way:	C++ supports `new` and `delete` keywords for memory management. In C++, memory allocation for arrays is different for single objects. Following code is used to `new` keyword:
`int *x = malloc( sizeof(int) );` `int *x_array = malloc( sizeof(int) * 10 );`	`int *x = new int;` `int *x_array = new int[10];`
Following code is used to release the memory with the help of `free()` function:	Following code is used to use `delete` keyword:
`free( x );` `free( x_array );`	`delete x;` `delete[] x;`

1.2.4   A Simple C++ Program

Programs are a sequence of instructions or statements. These statements form the structure of a C++ program. C++ program structure is divided into various sections, namely *headers, class definition, member functions definitions* and `main()` *function* (Refer Figure 1.10).

C++Headers
Class definition
Member functions definitions
Main function
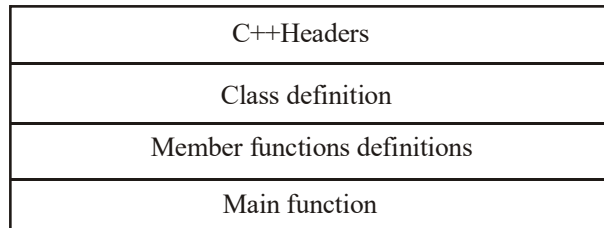
Fig. 1.10 *Structure of a C++ Program*

Note that C++ provides the flexibility of writing a program with or without a class and its member functions definitions. A simple C++ program (without a class) includes comments, headers, namespace, `main()` and input/output statements as shown in Figure 1.11.

| // A simple C++ program | ⟶ This is a comment and is ignored by the compiler

#include<iostream> ⟶ Header
using namespace std; ⟶ This tells the compiler to use std namespace

int main() ⟶ main Function
{
| Count<< "First C++Program";
  return 0; | ⟶ This is the body of main(). It contains the executable code
}
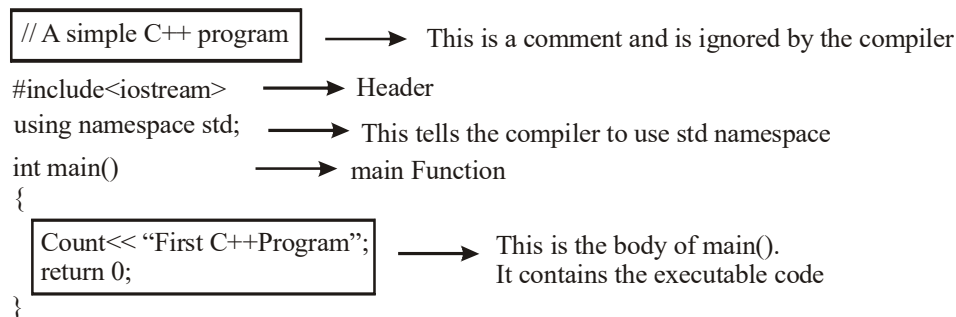
Fig. 1.11 *A Simple C++ Program Without a Class*

Comments

Comments are a vital element of a program that is used to increase the readability of a program and to describe its functioning. Comments are not executable statements and hence do not increase the size of a file.

C++ supports two comment styles: single line comment and multiline comment. Single line comments are used to define line-by-line descriptions. Double slash (//) is used to represent single line comments.

To understand the concept of single line comment, consider this statement:

```
    // An example to demonstrate single line
comment
```

It can also be written as follows:

```
    // An example to demonstrate
    // single line comment
```

Multiline comments are used to define multiple lines descriptions and are represented as /* ... */. For example, consider this statement.

```
    /* An example to demonstrate
        multiline comment */
```

Note: Multiline comment cannot contain another multiline comment. However, it may contain single line comment. This implies that /*……/*…….*/……*/ is not valid whereas /*….//…….*/ is valid.

Generally, multiline comments are not used in C++, as they require more space on the line. However, they are useful within the program statements where single line comments cannot be used. For example, consider this statement:

```
    for(int i=0; i<10; //loop runs 10 times i++)
```

A compiler ignores everything written after the single line comment and, hence, an error occurs. Therefore, in this case, multiline comments are used. For example, consider this statement.

```
    for(int i=0; i<10; /*loop runs 10 times */
i++)
```

Headers

Generally, a program includes various programming elements, such as built-in functions, classes, keywords, constants, operators, etc., that are already defined in the standard C++ library. In order to use such pre-defined elements in a program, an appropriate header must be included in the program. The standard *headers* contain information, such as prototype, definition and return type of library functions, data type of constants, etc. As a result, programmers do not need to explicitly declare (or define) the pre-defined programming elements. Standard headers are specified in a program through the preprocessor directive #include.

iostream Header File

When a compiler processes the instruction #include<iostream>, it includes the contents of iostream in the program. This enables a programmer to use

standard input, *output* and *error* facilities that are provided only through the standard streams defined in <iostream>. These standard streams process data as a stream of characters, that is, data is read and displayed in a continuous flow. The standard streams defined in <iostream> are listed here.

- **cin:** Pronounced 'see in' and is the standard input stream that is associated with the standard input device (keyboard) and is used to take the input from users.

- **cout:** Pronounced 'see out' and is the standard output stream that is associated with the standard output device (monitor) and is used to display the output to users.

- **cerr:** Pronounced 'see err' and is the standard error stream that is associated with the standard error device (monitor) and is used to report errors to the users. The cerr object does not have a buffer (temporary storage area) and hence immediately reports errors to users.

- **clog:** Pronounced 'see log' and is the buffered error stream that is associated with the standard error device (computer screen) and is used to report errors to users. Unlike cerr, clog reports errors to users only when the buffer is full.

Note: The letter c in cin, cout, cerr and clog stands for 'Console'.

For many years, C++ applied C-style headers, that is, .h extension in the headers. However, the standard C++ library introduced new style headers that include only header name. Hence, most modern compilers do not require any extension, though they support the older .h extension. Some of the C-style headers and their equivalent C++ style headers are listed in Table 1.2.

Table 1.2 *C Style and C++Style Headers*

C Style Header	C++ Style Header
<assert.h>	<cassert>
<ctype.h>	<cctype>
<float.h>	<cfloat>
<fstream.h>	<fstream>
<iomanip.h>	<iomanip>
<iostream.h>	<iostream>
<limits.h>	<climits>
<math.h>	<cmath>
<stdio.h>	<cstdio>
<stdlib.h>	<cstdlib>
<string.h>	<cstring>
<time.h>	<ctime>

Namespace

Since its creation, C++ has gone through many changes by the C++ Standards Committee. One of the new features added to this language is namespace. A *namespace* permits grouping of various entities, such as classes, objects, functions and various C++ tokens, etc., under a single name. Different users can create separate namespaces and, thus, can use similar names of the entities. This avoids compile-time error that may exist due to identical name conflicts.

The C++ Standards Committee has rearranged the entities of the standard library under a namespace called `std`. The statement `using namespace std` informs the compiler to include all the entities present in the namespace `std`. The entities of a namespace can be accessed in different ways which are listed here:

- By specifying the `using` directive:

  ```
  using namespace std;
  cout<<"Hello World";
  ```

- By specifying the full member name:

  ```
  std::cout<<"Hello World";
  ```

- By specifying the `using` declaration:

  ```
  using std::cout;
  cout<<"Hello World";
  ```

Note: All the entities of a namespace are public.

As soon as the new-style header is included, its contents are included in the `std` namespace. Thus, all modern C++ compilers support these statements:

```
#include<iostream>
using namespace std;
```

However, some old compilers may not support these statements. In that case, the statements are replaced by this single statement:

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

`main()` Function

The `main()` is a startup function that starts the execution of a C++ program. All C++ statements that need to be executed are written within `main()`. The compiler executes all the instructions written within the opening and closing curly braces '`{}`' that enclose the body of the `main()`. Once all the instructions in the `main()` are executed, the control passes out of the `main()` terminating the entire program and returning a value to the operating system.

By default, the `main()` in C++ returns an `int` value to the operating system. Therefore, the `main()` should end with the `return 0` statement. A return value zero indicates success and a non-zero value indicates failure or error.

Following are the features of `main()` function:

- The `main()` function is the mandatory function for all C++ programs.

- All functions require a return type whereas `main()` function requires either an `int` or `void` return type.

- The `main()` function also has a variety of input variable formats that are placed place in the () parentheses.

- The curly braces {and} represent the opening and closing of the scope in the `main()` function.

Note: Every C++ program must have one and only one `main()` function, and it is automatically called by the compiler.

1.2.5 Initialization Input with C in Tokens

C is a procedure-based language. Once you write a program in C, you must run it through a C compiler to turn your program into one that a computer can run (execute). C allows the input and output control in which a user can input the value to get the desired result. This language makes hardware devices easy to access and is used to manipulate individual bits in hardware registers. C++ is an object-oriented language that uses various concepts in handling of programs, such as virtual functions, multiple inheritance, exception handling, and polymorphism.

In C programming language, the 'Tokens' are considered as the most significant concept typically used for developing a C program. Fundamentally, the tokens in C language are referred as the building block of C programming language. The C programming supports the following 6 types of tokens:

1. **Keywords:** In C programming language, the 'Keywords' are predefined or reserved keywords. The C language has 32 keywords and each keyword has its special function.

2. **Identifiers:** In C programming language, the 'Identifier' is typically used for naming functions, variables, structures, unions, arrays, etc. The identifier is user-defined words and can be uppercase letters, lowercase letters, digits, and underscore. Remember that identifiers are not used for keywords.

3. **Strings:** In C programming language, the 'Strings' are considered as an array of characters having null character '\0' at the end of the string. Strings in C language are enclosed in double quotes (" ") and Characters are enclosed in single quotes(' ').

4. **Operators:** In C programming language, the 'Operators' are used to perform special operations on data. There are following two types of operators:

   Unary Operator: Applied with a single operand.

   Binary Operator: Applied between 2 operands.

   The types of operators include Arithmetic Operators, Relational Operators, Shift Operators, Logical Operators, Bitwise Operators, Conditional Operators, and Assignment Operator.

5. **Constants:** In C programming language, the 'Constant' is used for the value fixed, i.e., the constant value cannot be changed once declared.

6. **Special Symbols:** In C programming language, the following 'Special Symbols' are used.

Square Brackets [ ]: The square brackets are used for single and multi-dimensional arrays.

Simple Brackets ( ): The simple brackets are used for function declaration.

Curly Braces { }: The curly braces are used for opening and closing the code.

Comma (,): The comma is used to separate the variables.

Hash/Pre-Processor (#): The hash/pre-processor is used for the header file.

Asterisk (*): The asterisk is used to specify the Pointers.

Tilde (~): The tilde is used for destructing the memory.

Period (.): The period is used for accessing union members.

C Program Examples to Implement Tokens in C

Example 1.1: C Program for Keywords.

Program Code

```
#include <stdio.h>//Add all the basic C language libraries
int main()
{
//declare integer variable
int i=121;
//declare float variable
float f=11.11;
//declare character variable
char c='C';
//declare String variable in 2 ways
char s1[20]="VIKAS";
char s3[]="VIKAS";
//declare constant variable
const constant=3.14;
//declare short variable
short s=10;
//declare double variable
double d=12.12;
//displaying output of all the above keywords
printf("INT: %d\n", i);
printf("SHORT: %d\n", s);
printf("FLOAT: %f\n", f);
printf("DOUBLE: %f\n", d);
printf("CHAR: %c\n", c);
printf("STRING 1: %s\n", s1);
```

```
printf("STRING 3: %s\n", s3);
printf("CONSTANT: %d\n", constant);
return 0;
}
```

The output of the program

INT: 121

SHORT: 10

FLOAT: 11.110000

DOUBLE: 12.120000

CHAR: C

STRING 1: VIKAS

STRING 3: VIKAS

CONSTANT: 3

Example 1.2: C Program for Switch.

Program Code

```
#include <stdio.h>//Add all the basic C language
libraries#include
//main method used for running the application
int main()
{
//decalre variable
int n;
//asking enter any choice between 1 to 4
printf("Enter Any Choice Between 1 to 4=>");
scanf("%d",&n);
//switch case, based on choice it will gives us output
//if we did not take break each case then where ever it is
true that value and rest are printf
//none are true then default value will be print
switch (n)
{
case 1:
printf("I am Rajesh");
break;
case 2:
printf("I am Amanpreet");
break;
case 3:
printf("I am Vishal");
break;
case 4:
```

```
printf("I am Ashutosh");
break;
default:
printf("Opps! I Am Default");
}
return 0;
}
```

The output of the program

Enter Any Choice Between 1 to 4=>

Enter Any Choice Between 1 to 4=> 1

I am Rajesh

Enter Any Choice Between 1 to 4=>

Enter Any Choice Between 1 to 4=> 4

I am Ashutosh

Enter Any Choice Between 1 to 4=>

Enter Any Choice Between 1 to 4=> 6

Opps! I Am Default

Example 1.3: C Program for Functions.

Program Code
```
#include <stdio.h>//Add all the basic C language
libraries#include
int input(void);//declaring method
int getSquareArea(int side);//declaring method
int getCube(int cube);//declaring method
//main method used for running the application
int main()
{
int i=input();
int sArea= getSquareArea(i);
int cube=getCicrcleArea(i);
//displaying output
printf("Square Area is = %d\n",sArea);
printf("Cube of the Number is = %d\n",cube);
return 0;
}
//method definition
//this for asking the user input
int input(void)
{
int n;
//asking the user to input
```

```
printf("Enter Any Number=> ");
scanf("%d",&n);
return n;
}
//method definition
//this for getting square area
int getSquareArea(int input)
{
return input*input;
}
//method definition
//this for getting cube of the number
int getCicrcleArea(int cube)
{
return cube*cube*cube;
}
```

The output of the program

Enter Any Number => 25

Square Area is = 625

Cube of the Number is = 15625

Tokens in C programming language are considered to be the building block of the application and it gives complete structure to the C language code.

1.2.6 Control Statements

A statement is an instruction given to the computer to perform a specific action. In C++, a statement can be either a single statement or a compound statement. A single statement specifies a single action and is always terminated by a semicolon ';'. A compound statement, also known as a block, is a set of statements that are grouped as a compound statement and are always enclosed within curly braces '{}'.

By default, the statements are executed in the same order in which they appear in the program and each statement is executed only once. However, the serial execution of statements makes a program inflexible and unsuitable for most of the practical applications. To make a program more flexible, control statements are used to alter the flow of control of the program.

In C++, the control statements are broadly classified into three categories, namely, conditional statements, iteration statements and jump statements. All these control statements are commonly used with the logical tests or test conditions to alter the flow of control conditionally or unconditionally. To alter the flow conditionally, a particular condition is evaluated to control the flow of execution. On the other hand, to alter the flow unconditionally, no such condition is evaluated.

Conditional Statements

Conditional statements, also known as selection statements, are used to make decisions based on a given condition. If the condition evaluates to `True`, a set of statements is executed, otherwise another set of statements is executed.

The `if` Statement

The `if` statement selects and executes the statement(s) based on a given condition. If the condition evaluates to `True`, then a given set of statement(s) is executed. However, if the condition evaluates to `False`, then the given set of statements is skipped and the program control passes to the statement following the `if` statement.

The syntax of the `if` statement is:
```
if(condition)
{
statement 1;
statement 2;
}
statement 3;
```

The `if-else` statement

The `if-else` statement causes one of the two possible statement(s) to execute depending upon the outcome of condition.

The syntax of the `if-else` statements is:
```
if(condition)  //if part
{
statement1;
statement2;
}
else         //else part
statement3;
```

Here, the `if-else` statement comprises two parts, namely, `if` and `else`. If the condition is `True`, the `if` part is executed. However, if the condition is `False`, the `else` part is executed.

Example 1.4: A code segment to determine the largest of three numbers
```
.
.
if(a>b)
{
if(a>c)
cout<<"a is largest";
}
else      //nested if-else statement within  else
{
if (b>c)
cout<<"b is largest";
```

```
else
cout<<"c is largest";
}
.
.
.
```

The `if-else-if` ladder

The `if-else-if` ladder, also known as the `if-else-if` staircase, has an `if-else` statement within the outermost `else` statement. The inner `else` statement can further have other `if-else` statements.

The syntax of the `if-else-if` ladders is:
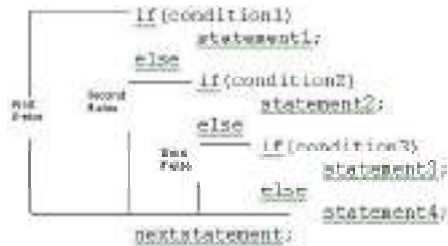
Example 1.5: A program to check whether a character is in lower-case or upper case

```
#include<iostream>
using namespace std;
int main()
{
char ch;
cout<<"Enter an alphabet: ";
cin>>ch;
if((ch>='A') && (ch<='Z'))
cout <<"The alphabet is in upper case";
else
      if((ch>='a') && (ch<='z'))
          cout<<"The alphabet is in lower case";
      else
          cout<<"It is not an alphabet";
   return 0;
}
```

The output of this program

```
Enter an alphabet: $
It is not an alphabet
```

Conditional Operator as an Alternative

The conditional operator '? :' selects one of the two values or expressions based on a given condition. Due to this decision-making nature of the conditional

operator, it is sometimes used as an alternative to `if-else` statements. It must be noted that the conditional operator selects one of the two values or expressions and not the statements, as in the case of an `if-else` statement. In addition, it cannot select more than one value at a time, whereas `if-else` statement can select and execute more than one statement at a time. For example, consider the statement:

```
max = (x>y ? x : y)
```

This statement assigns maximum of x and y to `max`.

The `switch` Statement

The `switch` statement selects a set of statements from the available sets of statements. The `switch` statement tests the value of an expression in a sequence and compares it with the list of integers or character constants. When a match is found, all the statements associated with that constant are executed.

The syntax of the `switch` statement is:

```
switch(expression)
{
case <constant1>: statement1;
  [break;]
case <constant2>: statement2;
  [break;]
case <constant3>: statement3;
[default]    : statement4;
   [break;]
}
statement5;
```

The C++ keywords `case` and `default` provide the list of alternatives. It must be noted that it is not necessary that every `case` label should specify a unique set of statements. The same set of statements can be shared by multiple `case` labels.

The keyword `default` specifies the set of statements to be executed in case no match is found. It must be noted that there can be multiple `case` labels, but there can be only one `default` label.

The `break` statements in the `switch` block are optional. However, it is used in the `switch` block to prevent a fall through. Fall through is a situation that causes the execution of the remaining cases even after a match has been found. In order to prevent this, `break` statements are used at the end of statements specified by each `case` and `default`. This causes the control to immediately break out of the `switch` block and execute the next statement.

Example 1.6: A code segment to demonstrate the use of switch statement

```
 .
cin>>x;
int x;
switch(x)
```

```
{
case 1:cout<<"Option 1 is selected";
 break;
case 2:cout<<"Option 2 is selected";
 break;
case 3:cout<<"Option 3 is selected";
 break;
case 4:cout<<"Option 4 is selected;
 break;
default:cout<<"Invalid option!";
}
.
.
```

In Example 1.6, depending upon the input, an appropriate message is displayed. That is, if 2 is entered, then the message Option 2 is selected is displayed. In case, 5 is entered, then the message Invalid option! is displayed.

Similar to if and if-else statements, switch statements can also be nested within one another. A nested switch statement contains one or more switch statements within its case label or default label (if any).

Note: Switch statement cannot be used for testing floating-point values or string values.

Iteration Statements or Loops

The statements that cause a set of statements to be executed repeatedly either for a specific number of times or until some condition is satisfied are known as iteration statements. That is, as long as the condition evaluates to True, the set of statement(s) is executed. The various iteration statements used in C++ are for loop, while loop and do-while loop.

The **for** loop

The for loop is one of the most widely used loops in C++. The for loop is a deterministic loop in nature, that is, the number of times the body of the loop is executed is known in advance.

The syntax of the for loop is:

```
for(initialise; condition; update)
{
//body of the for loop
}
```

Note that initialise, condition and update are optional expressions and are always specified in parentheses. All the three expressions are separated by semicolons. The semicolons are mandatory and hence cannot be excluded even if all the three expressions are omitted.

Example 1.7: A program to display a count down using `for` loop

```
#include<iostream>
using namespace std;
int main ()
{
int n;
for(n=1; n<=10; n++)
cout<< n <<" ";  // body of the loop
cout<<"\nThis is an example of for loop!!!";
              //next statement in sequence
return 0;
}
```

The output of this program

```
1 2 3 4 5 6 7 8 9 10
This is an example of for loop!!!
```

`for` *loop using comma operator*

`for` loop allows multiple variables to control the loop using comma operator . That is, two or more variables can be used in the `initialise` and the `update` part of the loop. For example, consider the statement:

```
for(i=1,j=50;i<10;i++, j—)
```

This statement initialises two variables, namely `i` and `j` and updates them. It must be noted that `for` loop can have only one condition.

The **while** loop

The `while` loop is used to perform looping operations in situations, where the number of iterations is not known in advance. That is, unlike the `for` loop, the `while` loop is non-deterministic in nature.

The syntax of the `while` loop is:

```
while(condition)
{
// body of while loop
}
```

The following points should be noted about the `while` loop:

- Unlike `for` loops where explicit `initialise` and `update` expressions are specified, `while` loops do not specify any explicit `initialise` and `update` expressions. This implies that the control variable must be declared and initialised before the `while` loop and needs to be updated within the body of the `while` loop.

- The `while` loop executes as long as `condition` evaluates to `True`. If `condition` evaluates to `False` in the first iteration, then the body of `while` loop never executes.

- `while` loop can have more than one expression in its condition. However, such multiple expressions must be separated by commas and are executed in the order of their appearance.

The `do-while` loop

As discussed earlier, in a `while` loop, the condition is evaluated at the beginning of the loop and if the condition evaluates to `False`, the body of the loop is not executed even once. However, if the body of the loop is to be executed at least once, irrespective of whether the initial state of the condition is `True` or `False`, the `do-while` loop is used. This loop places the condition to be evaluated at the end of the loop.

The syntax of the `do-while` loop is:

```
do
{
//body of do while loop
}while(condition);
```

Example 1.8: A program to calculate the sum of an Arithmetic Progression (AP)

```
#include<iostream>
using namespace std;
int main()
{
int a,d,n,sum,term=0;   /*a is the first term,d is
the common difference, n is the   number of terms to be
summed*/
cout<<"Enter the first term,common difference,"
<<"and the number of terms to be summed "
<<"respectively:\n";
cin>>a>>d>>n;
sum=0;
int i=1;
cout<<"\nThe terms are ";
do                      //do-while loop
{
term= a+(i-1)*d;
sum+=term;  //Adding each term to 'sum'
cout<<term<<" ";
    ++i;
}while(i<=n);
cout<<"\nThe sum of A.P. is "<<sum;
return 0;
}
```

The output of the program

```
Enter the first term, common difference, and the number
of terms to be summed respectively:
```

```
3
6
5
The terms are 3 9 15 21 27
The sum of A.P. is 75
```

It must be noted that all the three loops (`for`, `while` and `do-while`) can be nested within the body of another loop.

Note: C++ allows declaration of variable within the conditional expression of an `if` or `switch` or `while` or within the initialisation of a `for` loo,p which is not permitted in C.

Jump Statements

Jump statements are used to alter the flow of control unconditionally. That is, jump statements transfer the program control within a function unconditionally. The jump statements defined in C++ are `break`, `continue`, `goto` and `return`. In addition to these jump statements, a standard library function `exit()` is used to jump out of an entire program.

The **break** Statement

The `break` statement is extensively used in loops and `switch` statements. A `break` statement immediately terminates the loop or the `switch` statement, bypassing the remaining statements. The control then passes to the statement that immediately follows the loop or the `switch` statement. A `break` statement can be used in any of the three C++ loops.

Note that a `break` statement, used in a nested loop, affects only the inner loop in which it is used and not any of the outer loops. Similarly, a `break` statement used in a `switch` statement breaks out of that `switch` statement and not out of any loop that contains the `switch` statement.

The **continue** Statement

The `continue` statement is used to 'continue' the loop with its next iteration. In other words, `continue` statement skips any remaining statements in the current iteration and immediately passes the control to the next iteration. The `continue` statement does not terminate the loop (as in the case of `break` statements) rather it only terminates the current iteration of the loop. Like a `break` statement, a `continue` statement can be used in any of the three loops.

Example 1.9: A program to add the factors of a number

```
#include<iostream>
using namespace std;
int main()
{
int x=0,y,sum=0;
cout<<"Enter a number: ";
cin>>y;
while(1)
```

```
{
x++;
if(x>y)
break;
if(y%x!=0)
continue;
sum=sum+x;
}
cout<<"\nSum of factors: "<<sum;
return 0;
}
```

The output of the program

```
Enter a number: 8
Sum of factors: 15
```

The **goto** Statement

The goto statement can be used anywhere within a function or a loop. As the name suggests, goto statements transfer the control from one part to another part in a program which is specified by a label. Labels are user-defined identifiers followed by a colon that are prefixed to a statement to specify the destination of a goto statement.

Example 1.10: A program to demonstrate the use of goto statement

```
#include<iostream>
using namespace std;
int main()
{
int x=10;
loop: cout<<x<<", ";   //loop is a label
x—;
if (x>0)
goto loop;
cout << "\n Here is the example of goto !";
return 0;
}
```

The output of the program

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
Here is the example of goto !
```

The **exit()** Function

The exit() function is a standard library function that terminates the entire program immediately and passes the control to the operating system. This function takes a single parameter that is, exit status of the program and returns same status to the operating system upon termination. The status can be either zero or non-zero

value; where zero shows successful termination and non-zero shows unsuccessful termination of the program.

Example 1.11: A program to demonstrate the use of `exit( )`

```cpp
#include<iostream>
#include<cstdlib>     //for exit()function
using namespace std;
int main()
{
int a;
cout<<"Enter the value for a: ";
while(cin>>a)
{
if(a<0)
{
cout <<"This program is going "
<<"to terminate!";
exit(0);
}
cout<<"Enter another value for a: ";
}
return 0;
}
```

The output of the program

```
Enter the value for a: 7
Enter another value for a: 8
Enter another value for a: -4
This program is going to terminate!
```

1.2.7   Decisions Nesting

Nested Structures

We can create a structure within another. The program below illustrates nested structures.

Program 1.1

```cpp
/*Program to demonstrate nested structures*/
#include<iostream>
using namespace std;
int main(){
   struct account {
      unsigned number;
      string name;
      int balance;
   };
```

```
    struct deposit {
       account ac;
       unsigned amount;
       int years ;
    } d2;
    struct deposit d1 = {001, "VASU", 1000, 50000, 3};
    d2=d1; /*structure copy*/
    cout<<"Ac No.= "<<d2.ac.number<<" name="<<d2.ac.name
    <<" balance= " <<d2.ac.balance<<" deposit= "<<d2.amount
    <<" term= "<<d2.years ;
  }
```

The output of the program

```
  Ac No.= 1 name=VASU balance= 1000 deposit= 50000 term= 3
```

Analysis of the Program

Here, we wanted to include struct account as a member of struct deposit. Therefore, we have declared struct account before declaring struct deposit. Note that struct account has been declared as the first member of struct deposit. We created an object d2 of struct deposit as soon as declared. Then, we assigned values to another object of the same struct deposit. Thereafter we copied d1 to d2 to illustrate structure copy. Note how members of the structures are addressed in the cout statement. When we refer to a member years of deposit, we simply address d2.years. But, when we refer to number we cannot declare the same way because it is not a member of deposit. Only object account ac is the member of deposit. The number is a member of account ac. Therefore, we have to address number as d2.ac.number.

Passing Structures to Functions

Passing each member of the structure to a function is a tedious job. Doing so, for the entire structure can be easier. An example is given below to clarify.

Program 1.2

```
/*To demostrate passing entire structure to function*/
#include<iostream>
using namespace std;
struct account {
      unsigned number;
      string name;
      int balance;
    };
int main(){
    account a1= {001, "Vasu", 1000};
    account credit(account x);//function
    a1=credit(a1);
    cout<<("A/c No Name Balance\n");
```

```
        cout<<a1.number<<"\t"<< a1.name<<"\t"<<a1. balance;
}
account credit(account y){
    int x;
    cout<<("enter deposit made \n");
    cin>>x;
    y.balance+=x;
    return y;
}
```

If we want to pass a structure, it has to be declared before main function. Therefore structure `account` has been declared as a global structure. The function `credit` is declared with return data type structure as follows:

```
    account credit( account x);
```

Thus we are passing and returning `account`. Then `credit` is called by simply passing structure `a1`. In the called program, deposit is added to the balance and updated. This is returned to the `main()` where the updated record is printed.

The output of the program

```
enter deposit made
2400
A/c No Name Balance
1 Vasu 3400
```

Structure Pointers

We know how to declare pointers to various data types and arrays. Similarly pointers to structures can also be declared. An example is given below:

```
account a1 = { 1, "Vasu", 1000 };
account * sp;
sp = & a1;
```

Now sp is a pointer to a structure. Therefore if we assume the latter as another data type, declaring it as an array or as a pointer follows the same rule. Structure is, in fact, a user defined data type.

The access to individual elements of a structure defined in the form of a pointer, is similar but instead of dot we put an arrow pointer ->. Arrow pointer is formed by typing minus followed by greater than sign. An example will clarify the point.

Program 1.3

```
/*To demonstrate structure pointers */
#include<iostream>
using namespace std;
int main(){
struct account {
    unsigned number;
    string name;
    int balance;
```

```
    }a5;
account a1= {001, "VASU", 1000};
account *sp;
sp=&a1;
cout<<"A/c No\t Name\t Balance\n";
cout<<sp->number<<"\t"<<sp->name<<"\t"<<sp->balance<<"\n";
a5=*sp;
cout<<a5.number<<"\t"<< a5.name<<"\t"<< a5.balance << "\n";
}
```

The output of the program

```
A/c No Name Balance
1 VASU 1000
1 VASU 1000
```

In this program sp is a pointer to structure account*sp is assigned the address of structure a1. Then the contents of structure *sp are printed. Then the elements of *sp are copied to a5 and then printed (to demonstrate copying of structures). Note the difference between the notations when accessing elements of a structure and a structure pointer.

Passing Structure by Reference

Structures can also be passed by reference. But remember that the structure should be defined as a global variable. The following example illustrates passing structures by reference.

Program 1.4

```
/*structure pointers & functions*/
#include<iostream>
using namespace std;
struct account {
      unsigned number;
      string name;
      int balance;
   };
int main(){
    account a1= {001, "VASU", 1000};
    account debit(account *, int);
    int deb;
    cout<<"Enter amount to be withdrawn \n";
    cin>>deb;
    debit(&a1, deb);
    cout<<"A/c No \t Name \t Balance \n";
    cout<<a1.number<<"\t"<<   a1.name<<"\t"<<
a1.balance<<"\n";
}
struct account debit(struct account *x, int y) {
    x->balance-=y;
```

```
        return *x;
}
```

The output of the program

```
Enter amount to be withdrawn
200
A/c No Name  Balance
b1        VASU  800
```

Note that the address of a1 is passed. The prototype declares passing structure by reference and the amount of debit by value. In the called program the debit is adjusted. Note the pointer notation in subtracting the debit amount from the balance.

Structure and Vector

A structure can be declared as a data type for a vector. A program declaring a structure as vector is given below:

Program 1.5

```
/*Using vector */
#include<iostream>
#include<vector>
using namespace std;
struct account{
    unsigned number;
    string name;
    int balance;
};
int main(){
    vector<account>a1(3);
    int i, num, deb;
    cout<<"Enter number, name and balance for 3 accounts
\n";
    for(i=0; i<3; i++){
    cin>>a1[i].number>>a1[i].name>>a1[i].balance;
    }
    cout<<"Enter account number &amount to be withdrawn
\n";
    cin>>num>>deb;
    for(i=0; i<3; i++){
        if(a1[i].number==num)
        a1[i].balance-=deb;
    }
    cout<<"statement after transaction \n";
    for(i=0; i<3; i++){
cout<<a1[i].number<<"\t"<<a1[i].name<<"\t"<<a1[i].
balance<<"\n";
```

```
        }
    }
```

We have to declare structure as a global variable in such programs, otherwise we will get compilation error. A structure account is declared before main function. In the main function, we have declared a vector as given below:

```
vector<account>a1(3);
```

The above declares a vector array of size 3 with structure account as the data type. After this, we create three records of accounts. Then, we carry out a debit operation in the same function. We look for the account number, which matches with the typed account number and stored as num. When they are equal, we debit the amount typed by the user and stored as deb. The last part of the main function brings out a statement of the accounts after the transaction. The result of the example is given below:

The output of the program

```
Enter number, name and balance for 3 accounts
001 Lakshmi  100000
002 Vishnu   200000
003 Parvathi 300000
Enter account number &amount to be withdrawn
003 50000
statement after transaction
1   Lakshmi  100000
2   Vishnu   200000
3   Parvathi 250000
```

1.2.8   Type Conversion

An expression may involve variables and constants either of same data type or of different data types. If an expression consists of mixed data types, then they must be converted to the same type while evaluation to avoid compatibility issues. This is accomplished by the **type conversion,** which is defined as the process of converting one data type to another. This section discusses conversions between objects and basic data types and advanced type casting.

Conversions between Objects and Basic Types

The compiler cannot handle the type conversions in the expression involving user-defined data types, such as classes. This is because the operators are overloaded explicitly by the user to handle user-defined data types. Thus, the user has to provide the conversion mechanisms to handle the type conversions for the user-defined data types. This can be achieved by using either a constructor or an appropriate conversion function.

Note that as long as the expression involves the objects of the same class type, the operations are carried out smoothly without any problem. However, if the expression involves incompatible data types, then only the conversions are to be carried out explicitly. There can be three types of type conversions in an expression involving user-defined data types, which are as follows:

- Basic Type to Class Type
- Class Type to Basic Type
- One class Type to Another Class Type

Basic Type to Class Type

The conversion of the basic data type to a class type is accomplished by defining a constructor of the class that accepts an argument of any basic type, which is to be converted to class type. The object of the class type that calls the constructor is passed implicitly. Hence, the operand on the left-hand side of the type conversion using a constructor should always be an object of the class.

Example 1.12: A program to demonstrate the concept of conversion from the basic type to a class type.

```
#include <iostream>
using namespace std;
class weight
{
int kg;
int gm;
public:
weight()
{
kg = 0;
gm = 0;
}
weight(int i)   //constructor for type conversion
{
kg = i/1000; //converts int to data members kg and gm
gm = i%1000;
}
void display()
{ cout<<"\nkilogram= "<<kg<<" and gram= "<<gm; }
};
int main()
{
weight w1;
int t = 4500;
w1 = t;   //implicit call to the constructor for conversion

cout<<"Weight is";
w1.display();
return 0;
}
```

The output of the program

```
Weight is
Kilogram = 4 and gram = 500
```

In this example, a constructor `weight` accepting an argument of type `int` is defined. It converts the integer value to the class type `weight`. Note that the statement `w1 = t` implicitly calls the constructor for the conversion. The constructor can also be called explicitly using the following statement:

```
w1 = weight(t); //explicit call to the constructor
```

Class Type to Basic Type

Sometimes a situation may arise when a class type needs to be converted to a basic type. In this case, the constructor cannot be used, as it requires a class object on the left-hand side of the type conversion. To handle such conversions, an overloaded casting operator function (also known as conversion function) must be defined. A **conversion function** is a special member function that specifies the implicit conversion of a class object to another type. The syntax for defining the conversion function inside the class is as follows:

```
operator data_type()
{ //function body }
```

Example 1.13: A program to demonstrate the concept of conversion function

```
#include<iostream>
using namespace std;
class weight
{
int kg;
int gm;
public:
weight(int i,int k)
{
kg = i;
gm = k;
}
operator int()  //conversion function for int
{
int i;
i =kg*1000 + gm;
return i;
}
operator float()     //conversion function for float
{
float i;
i = (float)gm/1000;
i += kg;
return i;
```

```
}
};
int main()
{
weight w1(2,500);
int total1 = w1;            //implicit call to int()
float total2 = w1;          //implicit call to float()
cout<<"Total weight in grams: "<<total1<<endl;
cout<<"Total weight in Kilograms: "<<total2<<endl;
return 0;
}
```

The output of the program
```
Total weight in grams: 2500
Total weight in Kilograms: 2.5
```

In this example, two conversion functions `operator int()` and `operator float()` are defined. The `operator int()` converts the object `w1` of class `weight` to `int` type and `operator float()` converts `w1` to `float`. Note that the statements `total1 = w1` and `total2 = w1` implicitly call the conversion functions. However, these functions can also be called explicitly using the following statements:

```
int total1 = int(w1);       //explicit call to operator
int()
float total2 = float(w1);   //explicit call to operator
float()
```

The following points must always be kept in mind while defining a conversion function:

- The conversion function must be a non-static member function of the class.
- The conversion function cannot have an argument list or a return type.
- Like constructor, the conversion function can also be called explicitly.
- The compiler implicitly invokes the conversion function whenever a class object present on the right-hand side of the assignment statement does not match with the data type of the variable present on the left-hand side of the assignment statement.

One Class Type to Another Class Type

When one class type is to be converted into another class type, the class type (object) that appears on the right-hand side is known as **source class** and the class type (object) that appears on the left-hand side is known as the **destination class**. The conversion of one class type to another can be handled by using either the constructor or the conversion function (casting operator function). The compiler treats both of them in the same way. However, if the constructor is used for conversion, it must be defined in the destination class and if the casting operator function is used, it must be defined in the source class.

Example 1.14: A program to demonstrate the conversion between one class type to another using constructor

```cpp
#include<iostream>
using namespace std;
class weight_kg  //source class
{
int kg;
public:
weight_kg(int k)
{
kg = k;
}
int getweight() {return kg;}
void display()
{ cout<<"Weight in Kg: "<<kg<<endl;}
};
class weight_gm //destination class
{
int gm;
public:
weight_gm()
{
gm = 0;
}
weight_gm(weight_kg w)      //constructor for conversion
{
gm = w.getweight()*1000;    //converting kg to gm
}
void display()
{ cout<<"Weight in gm: "<<gm<<endl;}
};
int main()
{
weight_kg wkg(1);
weight_gm wgm;
wgm = wkg;   //implicit call to constructor

//wgm=weight_gm(wkg); //explicit call to constructor

wkg.display();
wgm.display();
return 0;
}
```

The output of the program

```
Weight in Kg: 1
Weight in gm: 1000
```

In this example, two classes `weight_kg` and `weight_gm` are defined. A constructor to convert weight in kilograms to grams is defined in the destination class, `weight_gm`. Note that the statement `wgm = wkg` calls the constructor implicitly. However, `wgm = weight_gm(wkg)` calls the constructor explicitly. Since the data members of the source class are private, they are accessed indirectly using the public member function `getweight()` in the constructor.

Example 1.15: A program to demonstrate the conversion between one class type to another using conversion function.

```cpp
#include<iostream>
using namespace std;
class weight_gm //destination class
{
int gm;
public:
weight_gm(){gm=0;}    //default constructor
void putweight(int g){gm=g;}
void display()
{cout<<"Weight in gm: "<<gm<<endl;}
};
class weight_kg  //source class
{
int kg;
public:
weight_kg(int k){kg=k;} //parameterized constructor
operator weight_gm()  //conversion  function  in  source
class
{
weight_gm w;
int g = kg*1000;
w.putweight(g);
return w;
}
void display()
{cout<<"Weight in Kg: "<<kg<<endl;}
};
int main()
{
weight_kg wkg(5);
weight_gm wgm;
wgm = wkg;   //implicit call to conversion function
```

```
//wgm=weight_gm(wkg); //explicit call to conversion
function

wkg.display();
wgm.display();
return 0;
}
```

The output of the program
```
Weight in Kg: 5
Weight in gm: 5000
```

In this example, a conversion function operator weight_gm() is defined in the source class weight_kg to convert weight in kilograms to grams. Note that the statement wgm = wkg calls the conversion function implicitly. However, the statement wgm = weight_gm(wkg) calls the conversion function explicitly.

Advanced Type Casting

ANSI C++ has also introduced four new casting operators, namely, static_cast, const_cast, reinterpret_cast and dynamic_cast. Although, C++ allows explicit conversion from one data type to another, this traditional typecasting works correctly if performed on simple variables of built-in data types. However, it can generate a run-time error or produce an unexpected result if used to convert one pointer variable to another, independent of the types they are pointing to.

Example 1.16: A program to demonstrate the limitation of traditional typecasting.
```
#include<iostream>
using namespace std;

int main()
{
int a=5;
float f=10.9;
int *p=&a;
float *fp=&f;
cout<<"*p= "<<*p<<"\t*fp= "<<*fp<<endl;
p=(int*)(fp);   //explicit conversion of float pointer
//to int
cout<<"*p= "<<*p;
return 0;
}
```

The output of the program
```
*p = 5    *fp = 10.9
*p = 1093559910
```

In this example, since a `float` pointer is explicitly converted to an `int` pointer, it displays a garbage value. Note that these types of conversions do not generate any compile-time error.

In order to control these types of conversions between pointer variables, ANSI C++ has added a new casting operator, namely, `static_cast` operator. In addition to `static_cast`, other cast operators, namely, `const_cast`, `reinterpret_cast` and `dynamic_cast` are also introduced. The conversions performed using these casting operators are checked at compile-time (except `dynamic_cast` operator).

`static_cast` Operator

The `static_cast` operator is used to perform safe and portable conversions between fundamental data types. The syntax for using the `static_cast` operator is as follows:

```
static_cast<data_type>(variable)
```

where,

      `static_cast`=C++ keyword
      `data_type`=target data type
      `variable`=an already declared variable that needs to be converted into target data type

To understand the use of static_cast operator, consider the following statements:

```
int a;
float f;
a=static_cast<int>(f);      //equivalent to a=int(f);
```

Here, the `static_cast` operator is used to convert the `float` variable into `int`.

As stated earlier, the static_cast operator is also used to control conversions between different types of pointer variables. For example, the statement converting the `float` pointer to the `int` pointer in Example 1.16 can be rewritten using the `static_cast` operator.

```
p=static_cast<int*>(fp); //using static_cast operator to
                          //convert float* to int*
```

Now, this statement results in compile-time error '`Cannot convert from float * to int *`'. Thus, the use of `static_cast` operator prevents the conversion of one pointer type to another at compile-time.

The static_cast operator is also used to prevent conversions from const char* to non-const char *. For example, consider the following statements:

```
const char *str="Hello";
unsigned char *p;       //non-const
p=(unsigned char*)(str); //traditional typecasting,no error
p=static_cast<unsigned char*>(str);  //error
```

The `static_cast` operator is also used to convert a derived class pointer to a base class pointer. For example, consider the following statements:

```
base *b1;
derived *d1;
b1=static_cast<base*>(d1);  //converting derived* to base*
```

`const_cast` Operator

The `const_cast` operator is used to change the `const`-ness of a variable either of built-in or user-defined data type. It is generally used to remove the `const` qualification of the variable. The syntax for using the `const_cast` operator is as follows:

```
const_cast<data_type>(variable)
```

Since the `const_cast` operator is used to change the const-ness of the variable only and not its type, that is, the source and the target data type must be the same.

Example 1.17: A program to demonstrate the concept of `const_cast` operator.

```
#include<iostream>
using namespace std;
int main()
{
const char *str="Hello";
char *p;
p=const_cast<char*>(str);  //no error
return 0;
}
```

In this example, `str` (declared as `const`) is assigned to `p` after removing its const-ness using the `const_cast` operator. Note that the source and the target data types are same (`char *`). Moreover, no compile-time error is generated.

`reinterpret_cast` Operator

The `reinterpret_cast` operator is used to perform conversions between fundamentally different data types. For example, conversion of the `int` pointer to the `int` variable or vice versa, the `float` pointer to the `float` variable or vice versa, etc. It also enables to convert one pointer type to another. The syntax for using the `reinterpret_cast` operator is as follows:

```
reinterpret_cast<data_type>(variable)
```

Unlike `static_cast`, `reinterpret_cast` performs unsafe and non-portable conversions. Note that while using the `reinterpret_cast`, the programmer, rather than the compiler, is responsible for the results.

Example 1.18: A program to demonstrate the use of `reinterpret_cast` operator

```
#include<iostream>
using namespace std;
```

```
int main()
{
int f=100;
int *p;
p=&f;
cout<<"Original value of p= "<<p<<endl;
cout<<"Original Value of f= "<<f<<endl;

p=reinterpret_cast<int *>(f);     //int to int*
cout<<"After converting f to p ";
cout<<"\np= "<<p<<endl;
f=reinterpret_cast<int>(p);   //int* to int
cout<<"After converting p to f ";
cout<<"\nf="<<f<<endl;
return 0;
}
```

The output of the program

```
Original value of p = 0x0012ff88
Original Value of f = 100
After converting f to p
P = 0x00000064
After converting p to f
F =100
```

In this example, firstly, the integer variable f is converted to the integer pointer p using the reinterpret_cast operator and the value of p is displayed. Note that the value of f (that is, 100) is converted to the hexadecimal form as p is a pointer variable and it can hold only the address values. The pointer variable p is converted back to the integer variable using the reinterpret_cast operator and hence, the original value 100 is displayed.

dynamic_cast **Operator**

C++ also provides another casting operator known as dynamic_cast operator. This operator differs from all other three casting operators as it allows conversions of objects at run-time and thus, is used only with pointers and references to the objects. The dynamic_cast operator is used when the conversion must access the **Run-Time Type Information (RTTI)** of an object rather than its static type. The syntax for using the dynamic_cast operator is as follows:

```
dynamic_cast<data_type>(object)
```

It allows the program to attempt conversion of a polymorphic base class object into its derived class object in a safe manner. Polymorphic objects are the objects of the base class that contains virtual

functions. The dynamic_cast operator always **performs** a valid conversion by special checking during run-time. If the conversion is invalid, it returns a NULL pointer to indicate the failure.

Note: To use dynamic_cast operator , the <typeinfo> header is included, which provides the RTTI of an object.

Example 1.19: A program to demonstrate the concept of dynamic_cast operator.

```cpp
#include<iostream>
#include<typeinfo>
using namespace std;

class base
{
public:
virtual void display(){cout<<"\nInside the base class";}
};
class derived:public base
{
public:
void display(){cout<<"\nInside the derived class";}
};
int main()
{
base *b1,b;
derived *d1,d;
d1=dynamic_cast<derived *>(&d);
if (d1==NULL)
cout<<"NULL pointer";
else
{
cout<<"Successful casting from derived* to derived*";
d1->display();
}

b1=dynamic_cast<base *>(&d);
if (d1==NULL)
cout<<"NULL pointer";
else
{
cout<<"\nSuccessful casting from derived* to base*";
b1->display();
}
```

```
b1=dynamic_cast<base *>(&b);
if (d1==NULL)
cout<<"NULL pointer";
else
{
cout<<"\nSuccessful casting from base* to base*";
b1->display();
}

d1=dynamic_cast<derived *>(&b);
if (d1==NULL)
cout<<"\nNULL pointer";
else
{
cout<<"\nSuccessful casting from base* to derived*";
d1->display();
}
return 0;
}
```

The output of the program

```
Successful casting from derived* to derived*
Inside the derived class
Successful casting from derived* to base*
Inside the derived class
Successful casting from base* to base*
Inside the base class
NULL pointer
```

In this example, four dynamic casts between pointer objects of type base* (b1) and pointer object of type derived* (d1) are performed. The first three conversions, namely, derived* to derived*, derived* to base* and base* to base* are successful. The last conversion, that is, base* to derived* results in NULL pointer as it is invalid to cast a base class object to a derived class object.

Check Your Progress

1. What is the most important feature of OOP paradigm?
2. Define the term class.
3. What do you mean by the term superclass?
4. List the two ways of achieving polymorphism in C++.
5. What is C?
6. What do you understand by programs?
7. Define single statement and compound statement.
8. What is an iteration statement?

1.3 OPERATORS AND EXPRESSIONS

In addition to the operators, such as arithmetic operators, relational operators, logical operators, conditional operators and assignment operators that most of the languages support, C++ provides some additional operators which are listed in Table 1.3.

Table 1.3 *Some Additional C++ Operators*

Operators	Description
::	Scope resolution operator
::*	Pointer-to-member declarator
->*	Pointer-to-member operator
.*	Pointer-to-member operator
new	Memory allocation operator
delete	Memory release operator

Here, only scope resolution, and new and delete operators are discussed.

Scope Resolution Operator (::)

In C++, variables in different blocks or functions can be declared with the same name, that is, the variables in different scope can have the same name. However, a local variable overrides the variables having same name in the outer block or the variable with global scope. Hence, a global variable or variable in the outer block cannot be accessed inside the inner block. This problem is solved by introducing new operator scope resolution operator :: introduced in C++.

The scope resolution operator :: is a special operator that allows to access a global variable that is hidden by a local variable with the same name.

Example 1.20:

A program to demonstrate the use of scope resolution operator:

```
#include<iostream>
using namespace std;
int x = 5;                    //global variable
int main()
{              //outer block
    int x = 3;                //local variable
    cout<<"The local variable of outer block is: "<<x;
    cout<<"\nThe global variable is: "<<::x;
    {             //inner block
 int x = 10;        //local variable
  cout<<"\nThe local variable of inner block is: "<<x;
  cout<<"\nThe global variable is: "<<::x;
 }
 return 0;
    }
```

The output of the program

```
The local variable of outer block is: 3
The global variable is: 5
The local variable of the inner block is: 10
The global variable is: 5
```

In this example, three variables with same name `x` are declared. The `x` declared outside `main()` has global scope which is hidden by both the variables `x` declared in the outer block and the inner block inside `main()`. The variable `x` inside the inner block overrides both the variables (global variable and the variable declared in the outer block). Thus, the global version of variable `x` is accessed using `::x` in the inner and the outer block.

Note that the main application of the scope resolution operator is in the classes that will be discussed when classes are introduced.

new and delete

C++ provides two dynamic allocation operators `new` and `delete` to allocate and de-allocate memory at run-time respectively. The `new` operator is a unary operator that allocates memory and returns a pointer to the starting of the allocated space. The syntax of allocating memory using the `new` operator is

```
p_var= new data_type;
```

where,

> `p_var`=the name of a pointer variable
>
> `new`=C++ keyword
>
> `data_type`=any valid data type of C++

To dynamically allocate memory to a variable of type `int` at run-time, for example, a pointer to type `int` is defined first and then the memory is allocated at run-time using the `new` operator as shown here.

```
int *iptr;        //pointer declaration
iptr=new int;  //allocating memory to an int variable
```

In these statements, the `new` operator returns the address of the memory allocated for an `int` variable from the free store and this address is stored in the pointer `iptr`. The pointer declaration and allocation of the memory can also be performed using a single statement as given here.

```
int *iptr=new int;
```

Moreover, the allocated memory using `new` operator can also be initialized. The syntax to initialize the memory using `new` operator is

```
p_var =new datatype(value);
```

where, `value` = initial_value.

For example, consider this statement.

```
float *fptr=new float(20.45);//initializing the newly
                    // created variable
```

Like built-in data types, the memory can also be allocated dynamically to derived and user-defined data types, such as arrays, structures and classes. These examples illustrate this concept.

- To allocate memory to a one-dimensional array, consider these statements:
```
int *arr_iptr=new int[20];
//allocating memory space to an array of // 20 integers
float *arr_fptr=new float[10];
//allocating memory space to an array of //10 float numbers
```

- To allocate memory to a multi-dimensional array, consider these statements:
```
int (*arr_iptr)[3] = new int[3][3];
        //allocating memory to a 2-D array
int (*arr_iptr)[5][5] = new int[3][5][5];
//allocating memory to a 3-D array
```

Note that all the dimensions must be specified while creating multi-dimensional arrays with the new operator. However, the first dimension can be a variable whose value is provided at run-time. For example, consider these statements.

```
int (*arr_iptr)[3][5] = new int[][5][5];    //invalid
int (*arr_iptr)[3][5] = new int[x][5][5];   //valid
int (*arr_iptr)[3][5] = new int[3][5][];    //invalid
```

The lifetime of a variable created at run-time using the new operator is not restricted till the execution of the program. Rather, it remains in the memory until it is explicitly deleted using the delete operator. When a dynamically allocated variable is no longer required, it must be destroyed using the delete operator to ensure safe and efficient use of the memory.

The syntax for using the delete operator is
```
delete p_var;
```

For example, to delete the pointers iptr and fptr of types int and float respectively, these statements are used.
```
delete iptr;
delete fptr;
```

Similarly, to delete the array pointers arr_iptr and arr_fptr of types int and float respectively, these statements are used.
```
delete [] arr_iptr;
delete [] arr_fptr;
```

Note: It is necessary to put square brackets ([]) after the delete operator to delete dynamically created arrays. The square brackets indicate that it is an array and, hence, avoids unpredictable results.

The function performed by the new and delete operator is similar to the malloc() and free() (library functions used in C). However, new and delete operators have several advantages over malloc() and free(), which are listed here.

- new automatically returns a pointer to the appropriate data type. There is no need to explicitly typecast the pointer as required in malloc().

- It automatically allocates enough memory to accommodate the object without using the `sizeof` operator.

- An object can be initialized while allocating space using the `new` operator.

- Both `new` and `delete` operators can be overloaded.

1.3.1 Operators Precedence

Generally, an expression consists of more than one operator, hence, a compiler needs to know which operator is to be evaluated first. For this, it is important to determine the precedence and associativity of operators. The order in which different operators in an expression are evaluated is determined by the precedence of operators. The operators with a higher precedence are evaluated before the operators with a lower precedence. However, the order in which operators of the same precedence are evaluated is determined by the associativity of the operators. The associativity of an operator can be either from the left to the right or from the right to the left. The operators with the left to right associativity are evaluated from the left-hand side while the operators with the right to left associativity are evaluated from the right-hand side.

The precedence and associativity of the C++ operators are listed in Table 1.4. Note that the precedence of operators decreases from the top to bottom.

Table 1.4 *Precedence and Associativity of C++ Operators*

Operator	Description	Associativity
::	Global scope resolution	Right to left
::	Class scope resolution	Left to right
( )	Function call	Left to right
[ ]	Array subscript	Left to right
->	Indirect member selector	Left to right
.	Direct member selector	Left to right
++	Post increment	Left to right
--	Post decrement	Left to right
!	Logical negation	Right to left
~	Bitwise complement	Right to left
+	Unary plus	Right to left
−	Unary minus	Right to left
++	Pre increment	Right to left
--	Pre decrement	Right to left
&	Address of	Right to left
*	Dereference	Right to left
(type)	Cast	Right to left
sizeof	Size in bytes	Right to left
new	Allocate memory	Right to left
delete	Deallocate memory	Right to left
.*	Direct pointer to class member selection	Left to right
->*	Indirect pointer to class member	Left to right
*	Multiplication	Left to right
/	Division	Left to right
%	Modulus	Left to right
+	Addition	Left to right
−	Subtraction	Left to right
<<	Bitwise shift left	Left to right
>>	Bitwise shift right	Left to right

<	Less than	Left to right
<=	Less than or equal to	Left to right
>	Greater than	Left to right
>=	Greater than or equal to	Left to right
= =	Equal to	Left to right
!=	Not equal to	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise inclusive OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
?:	Conditional operator	Left to right
=, *=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=	Assignment operator	Right to left
,	Comma	Left to right

Note: In C++, there is no operator for exponentiation. It is implemented using a standard library function `pow()`.

1.3.2 Expressions

A combination of variables, constants and operators that represents a computation forms an *expression*. Depending upon the type of operands involved in an expression or the result obtained after evaluating expression, there are different categories of an expression. These categories of an expression are discussed here.

- **Constant Expressions:** The expressions that comprise only constant values are called *constant expressions*. Some examples of constant expressions are `20`, `'a'` and `2/5+30`.

- **Integral Expressions :** The expressions that produce an integer value as an output after performing all the type of conversions are called *integral expressions.* As for example, `x`, `6*x-y` and `10+int(5.0)` are integral expressions. Here, `x` and `y` are variables of type integer.

- **Float Expressions:** The expressions that produce floating-point value as output after performing all type of conversions are called *float expressions,* for example, `9.25`, `x-y` and `9+float(7)` are float expressions. Here, `x` and `y` are variables of type `float`.

- **Relational or Boolean Expressions:** The expressions that produce a bool type value, that is, either true or false are called *relational or boolean expressions,* for example, `x+y<100`, `m+n==a-b` and `a>= b+c` are relational expressions.

- **Logical Expressions:** The expressions that produce a bool type value after combining two or more relational expressions are called *logical expressions,* for example, `x==5 && m==5` and `y>x||m<=n` are logical expressions.

- **Bitwise Expressions:** The expressions that manipulate data at bit level are called *bitwise expressions,* for example, `a>>4` and `b<<2` are bitwise expressions.

- **Pointer Expressions:** The expressions that give address values as output are called *pointer expressions*, for example, `&x`, `ptr and —ptr` are pointer expressions. Here, `x` is a variable of any type and `ptr` is a pointer.

Note: An expression may be formed by combining various combination of the expressions discussed earlier. Such expression is known as compound expression.

- **Special Assignment Expressions:** An expression can be categorized further depending upon the way the values are assigned to the variables.

- Chained assignment: *Chained assignment* is an assignment expression in which same value is assigned to more than one variable using a single statement, for example, consider these statements.

  ```
  a=(b=20); or a=b=20;
  ```

  In these statements, the value 20 is assigned to variable `b` and then to variable `a`. Note that, the variables cannot be initialized at the time of declaration using chained assignment. For example, consider these statements.

  ```
  int a=b=30;              //illegal
  int a=30, int b=30;         //valid
  ```

- **Embedded Assignment:** Embedded assignment is an assignment expression that is enclosed within other assignment expression, for example, consider this statement.

  ```
  a=20+(b=30); //equivalent to b=30; a=20+30;
  ```

  In this statement, the value ~~30~~ is assigned to variable ~~b~~ and then the result (20 + 30), that is, 50 is assigned to variable ~~a~~. Note that the expression (b = 30) is an embedded assignment.

- **Compound Assignment:** *Compound assignment* is an assignment expression that uses a compound assignment operator that is a combination of the assignment operator with a binary arithmetic operator. Consider this statement, for example:

  ```
  a+=20;          //equivalent to a=a+20;
  ```

In this statement, the operator += is a compound assignment operator, also known as a short-hand assignment operator.

Type Conversion

An expression may involve variables and constants either of same data type or of different data types. However, when an expression consists of mixed data types then they are converted to the same type during evaluation to avoid compatibility issues. This is accomplished by the *type conversion*, which is defined as the process of converting one predefined data type into another. Type conversions are of two types, namely (i) *implicit conversions* and (ii) *explicit conversions* also known as *typecasting*.

Implicit Conversions

Implicit conversion, also known as *automatic type conversion*, refers to the type conversion that is automatically performed by a compiler. Whenever the

compiler confronts a mixed-type expression, first of all `char` and `short int` values are converted to `int`. This conversion is known as *integral promotion*. After applying this conversion, all the other operands are converted to the type of the largest operand and the result is of a type of the largest operand. Table 1.5 illustrates the implicit conversion of data types starting from the smallest to largest data type. In expression `5 + 4.25`, for example, the compiler converts the `int` into `float` as float is larger than int and then performs the addition.

Table 1.5 Order of Data Types

Data Types	
char short int	smallest
int	
unsigned	
long int	
unsigned long int	
float	
double	
long double	largest

Note: If one operand is `long` and the other is `unsigned int`, and if the value of the `unsigned int` cannot be represented by a `long`, both operands are converted to `unsigned long`.

Typecasting

Typecasting refers to the type conversion that is performed explicitly using type cast operator. In C++, typecasting can be performed by using two different forms which are given here.

```
data_type(expression)  //expression in parentheses
     (data_type)expression  //data type in parentheses
```

where,

data_type = data type (also known as *cast operator*) to which the expression is to be converted.

Note: The cast operator is considered as a unary operator and thus has the same precedence as other unary operators.

To understand typecasting, consider this example.

```
float(num)+ 3.5 ;  //num is of int type
```

In this example, `float()` acts as a conversion function that converts `int` to `float`. However, this form of conversion cannot be used in some situations. Consider this statement, for example:

```
ptr=int*(x);
```

In such cases, conversion can be done using the second form of typecasting (which is basically C-style typecasting) as shown here.

```
ptr=(int*)x;
```

In addition, C++ introduces four new cast operators, namely (i) `const_cast`, (ii) `static_cast`, (iii) `dynamic_cast` and (iv) `reinterpret_cast`.

1.4 TOKENS

A *token* is defined as the smallest unit of a program. When a program is compiled, the compiler scans the source code and parses it into tokens to find the syntax errors. C++ tokens are broadly classified into *keywords*, *identifiers*, *constants*, *operators* and *punctuators*.

Keywords

Keywords are the predefined words that have special significance in any language. Every keyword is reserved for a specific purpose and hence must not be used as user-defined names (identifiers). All the keywords of C++ are listed in Table 1.6.

Table 1.6 *C++ Keywords*

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

Identifiers

Identifiers are the names given to uniquely identify various programming elements, such as *variables*, *arrays*, *functions*, *classes*, *structures*, *namespaces*, and so on. While defining identifiers in C++, programmers must follow the rules listed here.

- An identifier must be unique in a program.
- An identifier must contain only upper case and lower case letters, underscore character (_) or digits 0 to 9.
- An identifier must start with a letter or an underscore.
- An identifier in an upper case is different from that in a lower case.
- An identifier must be different from a keyword. In addition, identifiers that start with a double underscore '—' or an underscore followed by an upper-case letter must be avoided, as these names are reserved by the Standard C++ Library.
- An identifier must not contain other characters such as '*', ';' and whitespace characters (tabs, space and newline).

Some valid and invalid identifiers in C++ are given here.

```
Pol78_ddm     //valid
 _78hhvt4     //valid
902gt1     //invalid as it starts with a digit
Tyy;ui8       //invalid as it contains the  ';' character
```

```
for       //invalid as it is a C++ keyword
Fg026 neo     //invalid as it contains spaces
```

Note: Unlike C, there is no limit to the length of an identifier in C++. Thus, all the characters are significant in C++, however, in C, the first 32 characters are significant.

1.4.1 Constants

Constants, also known as *literals*, are the values that a program cannot alter during its execution, for example, '391', 'Byron', '51.072' and 'p' are all constants. Based on the type of value (data), C++ constants are broadly classified into three categories, namely (i) *numeric constants*, (ii) *character constants* and (iii) *string constants*.

Numeric Constants

Numeric constants refer to the numbers consisting of a sequence of digits (with or without decimal point) that can be either positive or negative. However, by default, numeric constants are positive. Numeric constants can be further classified as *integer constants* and *floating-point constants*, which are listed in Table 1.7.

***Table 1.7** Type of Numeric Constants*

Type	Description	Example
Integer constants	Integer constants refer to integer-valued numbers. Integer constants can be represented by three different number systems, namely *decimal* (base 10), *octal* (base 8) and *hexadecimal* numbers (base 16). The octal constants are preceded by a `0` (zero) and hexadecimal constants are preceded by a `0x` or `0X`.	`54, -646,` `01612, 0x38A`
Floating-point constants	Floating-point constants refer to the real numbers, that is, the numbers with a decimal point. Floating-point constants are also written in the *floating-point notation* in which the constant is divided into a *mantissa* and an *exponent.*	`64.23, -74.32,` `537E-9`

Note: Use of special characters, such as comma ',', semicolon ';'and question mark '?' are not permitted in numeric constants.

Character Constants

Character constants refer to a single character enclosed in single quotes (' '). The examples of character constants are 'f', 'M', '8', '&', '7', etc. All character constants are internally stored as integer value.

Character constants can represent either the printable characters or the non-printable characters. The examples of printable character constants are 'a', '5', '#', ';', etc. However, there are a few character constants that cannot be included in a program directly through a keyboard, such as backspace, newline, and so on. These character constants are known as non-printable constants and

are included in a program using the *escape sequences*. An escape sequence
refers to a character preceded by the backslash character (\). Some of the escape
sequences used in C++ are listed in Table 1.8.

Table 1.8 Escape Sequences

Escape Sequences	Character Constants
\a	Alert (bell)
\b	Backspace
\f	Form feed
\n	Newline (Linefeed)
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\0	Null
\'	Single quote
\"	Double quote
\\	Backslash
\?	Question mark
\C	Octal constant (C is a three-digit octal constant)
\xC	Hexadecimal constant (C is a three-digit hexadecimal constant)

In addition to character constant, C++ supports another character literal known
as *wide character literal*. This wide character literal uses two bytes of memory
and is specified by preceding character with an L. The examples of wide character
literals are L 'b', L 'mn', and so on.

String Constants

String constants refer to a sequence of any number of characters enclosed in
double quotes (" "). The examples of string constants are "hello", "name",
"color", "date", etc. Note that string constants are always terminated by
the null ('\0') character.

The presence of a backslash character in a string constant indicates an
escape sequence. The string constant, for example, "welcome\"home" is
displayed as welcome" home. Note that the double quote next to the backslash
is an escape sequence and not a delimiter for the string constant.

Operators

Operators are the symbols that represent various computations (such as addition,
subtraction, etc.) performed on various data-items. These data-items on which
operators act are known as operands.

Depending on the number of operands and functions performed, the C++
operators can be classified into various categories. This includes *arithmetic
operators*, *relational operators*, *logical operators*, *the conditional operator*

assignment operators, bitwise operators and *other operators*. These categories are further classified into *unary operators*, *binary operators* and *ternary operators,* as shown in Figure 1.12.
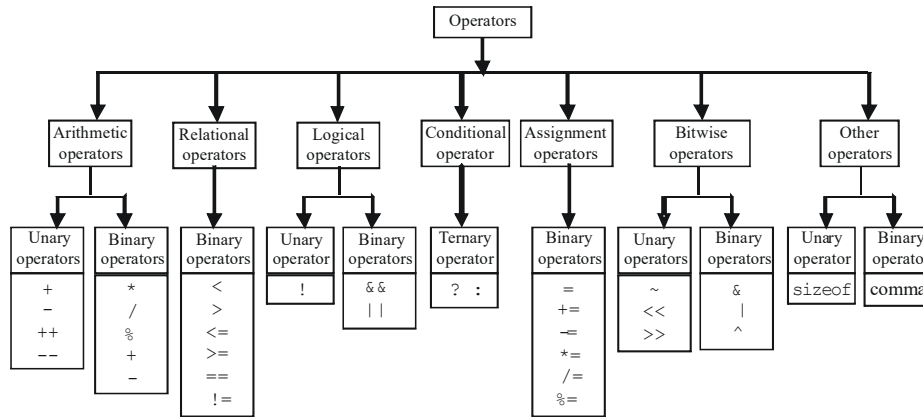
Fig. 1.12 *Types of Operators*

In addition to these operators (basically C operators), C++ also provides some new operators which you will learn later in this unit.

Punctuators

Punctuators, also known as *separators*, are tokens that serve different purposes based on the context in which they are used. Some punctuators are used as operators, some are used to demarcate a portion of the program, and so on. The various punctuators defined in C++ are asterisk '`*`', braces '`{ }`', brackets '`[ ]`', colon '`:`', comma '`,`', ellipsis '`…`', equal to '`=`', semicolon '`;`', parentheses '`( )`' and pound (hash) '`#`'.

1.5 BASIC DATA TYPES

A data type determines the type and the operations that can be performed on the data. C++ provides various data types and each data type is represented differently within the computer's memory. The various data types provided by C++ are (i) *built-in data types*, (ii) *derived data types* and (iii) *user-defined data types,* as shown in Figure 1.13.
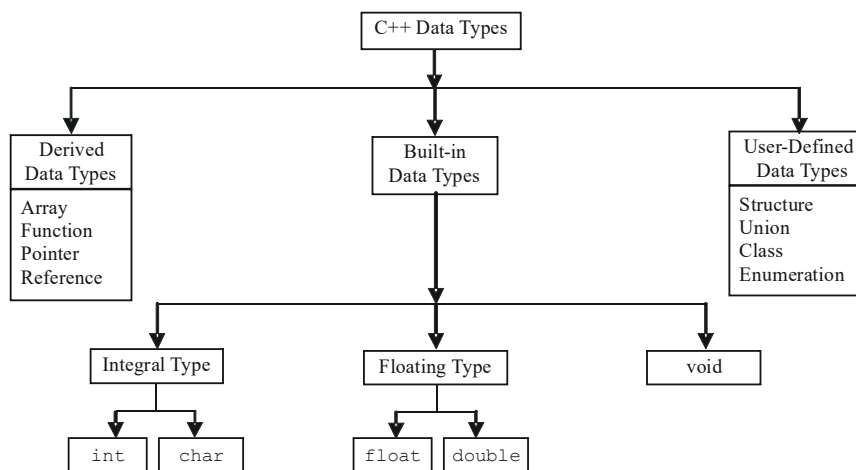
Fig. 1.13 *Various Data Types in C++*

Built-in Data Types

The basic (fundamental) data types provided by C++ are *integral*, *floating point* and `void` data type. Among these data types, the integral and floating-point data types can be preceded by several type modifiers. These modifiers (also known as type qualifiers) are the keywords that alter either the size or range or both of the data types. The various modifiers are `short`, `long`, `signed` and `unsigned`. By default the modifier is signed.

Note: The size and range of built-in data types vary from compiler to compiler and are specified in the header `climits`.

In addition to these basic data types, ANSI C++ has introduced two more data types namely, `bool` and `wchar_t`.

Integral Data Type

The integral data type is used to store integers and includes `char` (character) and `int` (integer) data types.

char

Characters refer to the alphabet, numbers and other characters (such as {, @, #, etc.) defined in the ASCII character set. In C++, the `char` data type is also treated as an integer data type as the characters are internally stored as integers that range in value from –128 to 127. The `char` data type occupies 1 byte of memory (that is, it holds only one character at a time).

The modifiers that can precede `char` are `signed` and `unsigned`. The various character data types with their size and range are listed in Table 1.9.

Table 1.9 Character Data Types

Type	Size (in Bytes)	Range
char	1	–128 to 127
signed char	1	–128 to 127
unsigned char	1	0 to 255

int

Numbers without the fractional part represent integer data. In C++, the `int` data type is used to store integers, such as 4, 42, 5233, –32, –745. Thus, it cannot store numbers, such as 4.28, –62.533. The various integer data types with their size and range are listed in Table 1.10.

Table 1.10 Integer Data Types

Type	Size (in Bytes)	Range
int	2	–32,768 to 32,767
signed int	2	–32,768 to 32,767
unsigned int	2	0 to 65,535
short int	2	–32,768 to 32,767
signed short int	2	–32,768 to 32,767
unsigned short int	2	0 to 65,535
long int	4	–2,147,483,648 to 2,147,483,647
signed long int	4	–2,147,483,648 to 2,147,483,647
Unsigned long int	4	0 to 4,294,967,295

Floating-Point Data Type

A floating-point data type is used to store real numbers, such as 3.28, 64.755765, 8.01, -24.53. This data type includes `float` and `double` data types. The various floating-point data types with their size and range are listed in Table 1.11.

Table 1.11 Floating-Point Data Types

Type	Size (in Bytes)	Range	Digits of Precision
float	4	$3.4*10^{-38}$ to $3.4*10^{38}$	7
double	8	$1.7 * 10^{-308}$ to $1.7*10^{308}$	15
long double	10	$3.4 * 10^{-4932}$ to $1.1*10^{4932}$	18

void

The `void` data type is used for specifying an empty parameter list to a function and return type for a function. When `void` is used to specify an empty parameter list, it indicates that the function does not take any arguments, and when it is used as a return type for a function, it indicates that the function does not return any value. For `void`, no memory is allocated and, hence, it cannot store anything. As a result, `void` cannot be used to declare simple variables, however, it can be used to declare generic pointers.

bool and wchar_t

The `bool` data type can hold only boolean values that is either `true` or `false`, where `true` represents `1` and `false` represents `0`. It requires only one bit of storage, however, it is stored as an integer in the memory. Thus, it is also considered as an integral data type. The bool data type is most commonly used for expressing the results of logical operations performed on the data. It is also used as a return type of a function indicating the success or the failure of the function.

In addition to `char` data type, C++ provides another data type `wchar_t`,which is used to store 16- bit wide characters. Wide characters are used to hold large character sets associated with some non-English languages.

Note: In C++, `wchar_t` is a built-in data type, while in C, it is defined in standard header file `stddef.h`.

1.5.1   User Defined Data Types

The various user-defined data types provided by C++ are *structures*, *unions*, *enumerations* and *classes*.

Structure, Union and Class

Structure and union are the significant features of C language. They provide a way to group similar or dissimilar data types referred to by a single name. However, C++ has extended the concept of structure and union by incorporating some new features in these data types to support object-oriented programming.

C++ offers a new user-defined data type known as class, which forms the basis of object-oriented programming. A *class* acts as a template that defines the

data and functions that are included in an object of a class. Classes are declared using the keyword `class`. Once a class has been declared, its object can be easily created.

Enumeration

An *enumeration* is a set of named integer constants that specify all the permissible values that can be assigned to enumeration variables. These set of permissible values are known as *enumerators*. For example, consider this statement.

```
enum country {US, UN, India, China} ;//declaring an
                                // enum type
```

In this statement, an enumeration data-type `country` (`country` is a tag name, consisting of enumerators `US, UN,` and so on, is declared. Note that these enumerators represent integer values, so any arithmetic operation can be performed on them.

By default, the first enumerator in the enumeration data type is assigned the value zero. The value of subsequent enumerators is one greater than the value of previous enumerator. Hence, the value of `US` is 0, value of `UN` is 1, and so on. However, these default integer values can be overridden by assigning values explicitly to the enumerators as shown here.

```
enum country {US, UN=3, India, china} ;
```

In this declaration, the value of `US` is 0 by default, the value of `UN` is 3, `India` is 4, and so on.

Once an enum type is declared, its variables can be declared using this statement. `country  country1,country2;`

These variables `country1, country2` can be assigned any of the values specified in `enum` declaration only. For example, consider these statements.

```
country1 = India;     //valid
country2 = Japan;     //invalid
```

Though the enumerations are treated as integers internally in C++, the compiler issues a warning, if an `int` value is assigned to an `enum` type. Consider these statements, for example:

```
country1 = 3;              //warning
country1= UN;             //valid
country1= (country)3;   //valid
```

Note: In C++, the enum type variables can be declared without using `enum` keyword while in C, it is necessary to use `enum` keyword.

C++ also allows to create anonymous enums, that is, enums without using tag name as shown in this statement.

```
enum {US, UN=3, India, China} ;
```

The enumerators of an anonymous enum can be used directly in the program as shown here.

```
int count = US;
```

Note: In C++, an `enum` declared within a structure or class is visible to that structure or class only.

1.5.2 Derived Data Types

Data types that are derived from the built-in data types are known as *derived data types*. The various derived data types provided by C++ are *arrays*, *functions*, *references* and *pointers*.

Array

An *array* is a set of elements of the same data type that are referred to by a same name. All the elements in an array are stored at contiguous (one after another) memory locations and each element is accessed by a unique index or subscript value. The subscript value indicates the position of an element in an array.

Function

A *function* is a self-contained program segment that carries out a specific well-defined task. In C++, every program contains one or more functions that can be invoked from other parts of a program, if required.

Reference

A *reference* is an alternative name for a variable, that is, a reference is an alias for a variable in a program. A variable and its reference can be used interchangeably in a program as both refer to the same memory location. Hence, changes made on any of them (say, a variable) are reflected in the other (on a reference).

Pointer

A *pointer* is a variable that can store the memory address of another variable. Pointers allow to use the memory dynamically, that is, with the help of pointers, the memory can be allocated or de-allocated to the variables at run-time, thus, making a program more efficient.

1.5.3 Declaration of Variables

Variables must be declared in a program before they are used. The declaration of a variable informs the compiler, the specific data type to which a variable is associated and allocates sufficient memory for it. The syntax for declaring a variable is as follows:

```
data_type variable_name;
```

For example, a variable `a` of type `int` can be declared using this statement.

```
int a;
```

At the time of the variable declaration, more than one variable of the same data type can be declared in a single statement, for example consider this statement.

```
int x, y, z;
```

Note that in C++, it is not necessary to declare the variables in the beginning of a program as required in C. It can be declared at the place where it is used first.

1.6 OPERATORS AND FUNCTION OVERLOADING

Operators are one of the five tokens of the language. C++ supports many built-in operators. The operators in C++ language can be classified into various categories as:

- Arithmetic Operators
- Relational Operators
- Boolean Logical Operators
- Bitwise Operators

The operators use one or more operands and perform the desired operations. The operands can be literals (constants), variables or expressions. C++ also supports three types of operations based on the number of operands used. They are:

- Binary (involving two operands)
- Unary (involving single operand only)
- Ternary (involving three operands)

Arithmetic Operators

The basic arithmetic operators are:

Addition (+) e.g. c = a + b

Subtraction (–) e.g. c = a – b

Multiplication (*) e.g. c = a * b

Division e.g. c = a/b

Modulus e.g. c = a % b

To get the value of the remainder, we use c = a % b.

The symbol `%` is also popularly called as modulus operator. Modulus operator can be used only with integers.

Therefore, c = 100/6; will produce c = 16.

d = 100 % 6; will produce d = 4

Let us execute a program to understand the arithmetic operators.

Program 1.6
```
/*to demonstrate use of arithmetic operators*/
#include<iostream>
using namespace std;
int main(){
    int var1=4, var2=17;
    float var4=2.5f, var5=12.0f;
    cout<<"\n addition of floats ="<< (var4+var5);
    cout<<"\n multiplication of integers ="<< (var1*var2);
```

```
    cout<<"\n modulus of integers ="<< (var2%var1);
    cout<<"\n division of floats ="<< (var5/var4);
    cout<<"\n subtraction of integers ="<< (var2-var1);
}
```

Here `var1` and `var2` are declared as integers and are assigned initial values of 4 and 17 respectively; `var4` and `var5` are declared as floats and assigned initial values of 2.5f and 12.0f respectively. (Note: if we don't assign the suffix 'f', the numbers will be treated as doubles, needing double the amount of storage space.) Various operations are carried out as part of the print (`cout`) statement itself. Addition of floats, subtraction of integers and multiplication of integers are simple. Let us calculate modulus of integers.

Integer

17%4=1 – which is the remainder.

Let us now look at the division. In case of float, the quotient will be found to the defined precision and hence 12/2.5 gives quotient of 4.8.

Now look at the result.

The output of the program

```
addition of floats =14.5
multiplication of integers =68
modulus of integers =1
division of floats =4.8
subtraction of integers =13
```

Expressions

We can form expressions by combining the following:

i. Data types (variables and constants)

ii. Operators (Arithmetic operators, relational operators, logical operators)

For instance, a = 100 + 2/4;

What is the right answer?

Is it 100 + 0.5 = 100.5 or, 102/4 = 25.5

Precedence of Operators

To avoid ambiguity, there are precedence rules for operators in C++. Precedence denotes which operator has to be evaluated first. The operator precedence is given in Annexure 3. In the above example, since "/" has precedence over "+", the expression will be evaluated as 100 + 0.5 = 100.5.

Let us confirm the operator precedence through a program. It is given in the example below:

Program 1.7

```
/*to demonstrate operator precedence*/
#include<iostream>
int main(){
```

```
std::cout<<"\n first calculation: "<< 24/6*4;
std::cout<<"\n second calculation: "<< (24/2-8/4+3);
}
```

Let us evaluate the second expression.

$24/2 - 8/4 + 3 = 12 - 8/4 + 3 = 12 - 2 + 3 = 13$

Look at the result of the program to confirm our discussion.

The output of the program

```
first calculation: 16
second calculation: 13
```

Whenever you are in doubt about the outcome of an expression, it is better to use parentheses to avoid ambiguity. Use of parentheses does not cause any overhead to the program and hence can be used liberally.

Arithmetic Assignment Operators

The assignment statements can be written in a shorthand notation when the variable on the Right Hand Side (RHS) of the expression repeats on the Left Hand Side (LHS). The following example will make it clear.

The general form is exp1 = exp1 + exp2.

This can also be written as exp1 += exp2.

Example

Simple form	Shorthand form
a = a+1;	a + = 1;
a= a - b;	a - = b;
a = a*(b+c);	a* = b+c;
a = a/b;	a / = b;
d = d - (a+b);	d - = a+b;
x = x%y;	x% = y;

An example involving shorthand assignment operators is given below for illustration.

Program 1.8

```
/*to demonstrate shorthand assignment operators*/
#include<iostream>
using namespace std;
int main(){
    int var1=15, var2=4;
    float var3=15.0f, var4=5.5f;
    cout<<"\n var1+=5 is " <<(var1+=5);
    cout<<"\n var3-=2.4 is " <<(var3-=2.4f);
    cout<<"\n var1/=4 is " <<(var1/=4);
    cout<<"\n var1%=var2 is " <<(var1%=var2);
    cout<<"\n var3*=var4 is " <<(var3*=var4);
}
```

The output of the program

```
var1 += 5 is 20
var3 -= 2.4 is 12.6
var1/ = 4 is 5
var1% = var2 is 1
var3* = var4 is 69.3
```

The program demonstrates the use of all the five shorthand assignment operators.

We may think there is something wrong with the result. After the execution of the first print statement, var1 gets the new value of 20 and after execution of second print var3 gets the new value of 12.6. After the third print var1 gets the value of 5. Now you will understand why we got unexpected result.

Binary Operators

We need two operands for any of the basic arithmetic operation such as addition, subtraction, division and multiplication. Since these operators need two operands, the operators are called binary operators. Don't confuse this with binary numbers.

Unary Operators

C++ has special operators for a single operand. These are called unary operators. These operators can be used for adding 1 (increment) to an integer variable or subtracting 1 (decrement) from an integer variable. The increment operator '++' and decrement operator '– –' are the unary operators.

Both the unary operators can be used either as prefix or as postfix. In both the cases, the operands will be incremented or decremented, but there is a difference. Suppose we write y = x++; then y will be assigned the value of x before it is incremented.

```
x = 5;
y = x++
```

In this case, y will become 5 and x will become 6 after execution of the second statement.

On the other hand if we write

```
x = 5:
y = ++ x;
```

y will become 6 since x will be incremented before y is assigned the value. Now look at the example given below for confirming the above concept.

Program 1.9

```
/*to demonstrate usage of unary operators as prefix and
suffix*/
#include<iostream>
int main(){
    int var1=9, var2;
    var2=++var1;
    std::cout<<"\n var2= "<< var2 << " var1= "<<var1;
    var2=var1–;
```

```
        std::cout<<"\n var2= "<< var2 << " var1= "<<var1;
}
```

The program starts with declaration and assignment as follows:

```
var1=9;
var2=++var1; /*var2 is assigned value of var1 after
increment */
```

Therefore, value of both `var1` and `var2` is 10.

Now

```
var1=10;
```

```
var2=var1--;      /*var2 is assigned value of var1 before
```
decrement*/

The output of the program

```
    var2= 10 var1= 10
    var2= 10 var1= 9
```

In some cases prefixing or suffixing may not cause any difference, but in some it will cause a difference as illustrated in the example above.

Relational Operators

The relational operators are used to check the relationship between two numeric operands or expressions. The relational operators of C++ are given below:

i. Greater than (>) e.g. x > y means, Is x greater than y?

ii. Less than (<) e.g. x < y means, Is x less than y?

iii. Greater than or equal (> =) e.g. x > = y means, Is x greater than or equal to y?

iv. Less than or equal (< =) e.g. x < = y means, Is x less than or equal to y?

v. Equal (==) e.g. x==y means, Is x equal to y?

vi. Not Equal (!=) e.g. x!=y means, Is x not equal to y?

The outcome of a relational expression involving integers, characters or floating point numbers, is either true or false. These operators help in finding out the relationship between two operands or expressions.

Assignment operators are written as follows:
```
identifier = expression;
```

Example

```
vari = 3;
int A = 3;
```

For instance, in the statement,

int vari = 3;

`int vari` is the declaration of the variable;

`vari = 3` is an assignment statement;

`=` is the assignment operator.

C++ allows multiple assignment in the following form:

```
identifier 1 = identifier 2 = (...)= expression
```

Example

```
va = vb = vz = 25;
```

But one should know the difference between assignment operator and equality operator. In other languages both are represented by =. But in C++ as well as in C, equality operator is expressed as == (pronounced as equal to equal to) and assignment is represented by =.

The relational operators are evaluated to check whether they are true or false. Let us look at an example to illustrate the concept of relational operator.

Program 1.10

```
/*to demonstrate relational operators*/
#include<iostream>
int main(){
    int var1=5;
    bool b1, b2;
    b1= (var1>3);
    std::cout<<"\n expression1 is "<<b1;
    b2= (var1>5);
    std::cout<<"\n expression2 is "<<b2;
}
```

Here the expression1 turns out to be true. Hence, b1 will be true and will be equal to 1. Since the expression2 is false, b2 will be false and equal to 0. Result of program confirms this.

The output of the program

```
expression1 is 1
expression2 is 0
```

Logical Operators

The relational operators are useful for checking a single condition. Of course they compare two numeric operands or expressions and result can be true or false. When we want to combine multiple conditions, we need logical operators. Examples involving multiple conditions are given below:

- If (A >B) AND if (A>C) print A is larger.
- If year = 50 OR if year =1950, print 1950

The logical operators such as AND, OR are used to combine multiple conditions as above. Thus, the logical operators are used with relational operators, which operate on numeric operands. Table 1.12 gives a list of logical operators defined in C++.

Table 1.12 *Logical Operators*

Operator	Symbol	Example	Remarks
Logical AND	&&	C=A&&B	C is true if both A&B are true
Logical OR	||	C=A||B	C is false if both A & B are false
Logical NOT	!	C=!A	C is true if A is false

Bitwise Operators

Bit wise operators access the internal representation of the numbers in bits, viz. bit 0 and 1. These operators apply only to the integer family operands including char. There are six operators for bit wise operation or manipulation. The operators and their symbols are given below:

    & bitwise AND

    | bitwise OR

    ^ bitwise exclusive OR

  << left shift

  >> right shift

   ~ bitwise unary NOT

Hexadecimal and Octal Representation

In C++, we can represent numbers as decimal, octal and hexadecimal numbers. Any problem involving bit wise operation involves conversion of the integers into binary numbers. After the operation, the system will convert the binary number into decimal and give the result.

In software, we organize the digits byte wise. But the hardware handles it bit wise. Therefore, the bit wise operators are very useful for directly interacting with the hardware. But in programs we give decimal numbers. The bit wise operator recognizes the number, carries out bit wise operation and gives the result in decimal numbers. To check whether the operation is correct or not, we have to convert the operand and the result to bits and then see whether it is correct. This is done only in the beginning stage, till we gain confidence. Later on, we don't have to do it. Although bit wise operators manipulate the bits, they understand the decimal, octal and hexadecimal numbers and carry out the operation at one go.

If we use decimal numbers, we have to convert them to binary, which is easy for the computer, but we will take time. Let us use hexadecimal, or octal numbers for discussing the use of bit wise operator since it will be easy for us to convert them to binary and vice versa. But, unless otherwise specified, the program will accept decimal numbers for bit wise operation and give the result in decimal form. For ease of discussions, we will specify the operand in the hex or octal number system and the result in the same radix.

Let us take a decimal number 6666. Let us convert it into binary, octal and hexadecimal numbers.

16-bit binary number equal to 6666-0001101000001010

equivalent hexadecimal number-1a0a

equivalent octal number-015012

What do you notice? The hexadecimal number groups 4 bits each from right(LSB) in the 16-bit binary number and gives equivalent hexadecimal of the 4 bits. Thus, we have 4 digits of hexadecimal number for the 16-bit binary number.

Similarly the 16-bit binary number is grouped into 3 bits each from the LSB. Therefore, 6 octal numbers will arise for a 16-bit binary number. The last group contains only the MSB. It can either be 0 or 1. In this case, it is zero. The other numbers could be from 0 to 7. Therefore, it is easy to convert binary numbers to octal numbers. Similarly, it is easy to convert an octal number to binary; convert each octal digit to its equivalent binary digit.

Now using examples, let us understand the operation of bit-wise operators.

Bitwise NOT Operator

The NOT operator inverts all bits i.e. it converts 1 to 0 and 0 to 1.

If x is the number x = 1100

~x = 0011.

This can be used to encrypt the information at the sending end and decrypt it at the receiving end.

It changes 1 to 0 and 0 to 1

The usage is as follows

Let us declare I, J as integers ;

I = 6666 ;

J = ~ I ;

I = 015012 -octal

J = 762765 -octal

Note the symbol for bit-wise NOT is different from logical NOT.

Bitwise AND Operator

It compares two bits and if both are 1, the output is 1, otherwise zero. This can be used to mark some sets of bits. Suppose we want to check only the 16th bit in 16-bit word of a number say A, we can carry out AND of A and another word whose 16th bit is 1 and all other 15 bits are 0. When you AND it, the 15 bits will be 0 and 16th bit will be a 0 or 1 depending on A.

Thus, it operates on 2 operands.

Given A = 015012 octal

B = 177777

C = A & B will provide an output of C = 015012 octal. This can be verified by converting into bits.

A = 1A0A hexadecimal

B = 0000

C = A & B will produce C = 0000 hexadecimal because B is all zeros.

OR Operator

This is also a binary operator.

Let A = 015012 octal

B = 000000

C = A | B ; will produce C = 015012 octal

Let   A =   015012 octal

X =   177777 octal

Y =   A | X ; will produce Y = 177777 octal.

This is because X is all ones and hence A | X will automatically produce all 1s even without looking at A.

When B is zero, the output will be 1 wherever A is 1. Therefore, the output will be same as A.

Exclusive OR operator

Only when one of the 2 operands is 1 we get the output of exclusive OR as 1; otherwise output will be 0.

A   =   1A0A Hex

=   0001101000001010

Let B   =   1111111111111111   =   FFFF hex

A ^ B   =   1110010111110101   =   E5F5 hex

Let C = 0000000000000000 = 0000 hex

A^ C = 0001101000001010 = IA0A hex

B^C =FFFF hex

Let us take a break and confirm the use of above 4 bit-wise operators.

Program 1.11

```
/*to demonstrate use of bit-wise operators*/
#include<iostream>
using namespace std;
int main(){
    int C, A=6666, B=0;
    C=A&B;
    cout<<"\n Result of AND " << C;
    C=A|B;
    cout<<"\n Result of OR " << C;
    C=A^B;
    cout<<"\n Result of Exclusive OR " << C;
    C=~A;
    cout<<"\n Result of NOT " << C;
}
```

We have declared A as 6666 and B as 0. We get the following results:

The output of the program

```
Result of AND 0
Result of OR 6666
Result of Exclusive OR 6666
Result of NOT -6667
```

You can easily verify the correctness of results in respect of AND, OR and Exclusive OR. You may find it difficult in the case of NOT. It is explained below. The binary equivalent of 6666 = 0001 1010 0000 1010

B=~A= 1110 0101 1111 0101

This is a negative number since MSB =1 and therefore the number is in 2's complement notation. To find the value of the number, we have to invert the bits and add 1. When we invert, we get the original number 6666 and when we add 1, we get 6667 and since it is a negative number the result –6667 is correct.

Right Shift Operator

```
int A, B;
B = A >> 2;
```

Here the bits in A will be shifted 2 places to the right. The resulting number will be stored in B. We have to answer two questions that may arise during this operation.

- What happens to the bits shifted out?
- What is stored in the vacuum created in the number?

Assume that we are shifting the number 1101 by 2 bits to the right. Then what happens to 01 on the right. Obviously they will be lost. Since we have shifted the number by 2 bits, 11 on the left would have moved to the rightmost positions. Then what will be stored in the two leftmost positions? The bit corresponding to MSB i.e. leftmost bit will fill these. Had it been zero, indicating that it is a positive number, the positions will be filled with 0. In this case, since MSB was 1, the positions will be filled with 11. Thus the result of shifting 1101 by 2 bits to the right will be 1111. This helps in preserving the sign bit and hence this function is called sign extension.

Now A = 6666. In hex, it will be 1A0A= 0001 1010 0000 1010

Let B = A >> 4 ;

The bits will be shifted by 4 places. Because sign bit is 0, the vacuum will be filled with 0s.

Therefore, B = 01A0

Let us take another case.

Let K = – 25 and let it be represented as a byte.

25 = 00011001

K in 2's complement representation will be 1110 0111

Now, let L=K >> 4

Then, L = 1111 1110

Left Shift Operator

$j = i \ll p$

will shift i by p bits and store the result in j.

What happens to the rightmost position from where the bits were shifted? Zeros will be placed whenever a bit is shifted.

$i = 015012$; / * octal number * /

$j = i \ll 3$;

The result will be 150120. Let us take one more example.

$A = 1A0A$; / * hexadecimal number * /

$B = A \ll 4$;

the shifted number will be,

A0A0

Shifting again by 4 bits will give 0A00

Check by shifting the octal number 150120 to the left by 3 bits. The result will be 101200. Let us confirm the use these three operators with a program.

Program 1.12

```
/*to demonstrate use of bit-wise shift operators*/
#include<iostream>
int main(){
    long B, A=6666;
    B=A>>4;
    std::cout<<"\n Result of right shift " <<B;
    B=A<<4;
    std::cout<<"\n Result of left shift " <<B;
}
```

We have declared A as 6666 and execute both types of shifts.

The output of the program

```
Result of right shift 416
Result of left shift 106656
```

The result of left shift needs needs explanation. We have,

$A == 6666$, it's hexadecimal equivalent will be 1A0A = 0001 1010 0000 1010

We now shift A to the left by 4 bits. Since A and B are long integers with at least 32 bits wide, the 4 bits shifted will not be lost. B will be actually 0001 1010 0000 1010 0000 which is equal to 106656 in decimal.

Bit Field

The bit wise operators don't really need bytes for their operation. However, the minimum size that is available is a character of 8-bit width. Some variables may require only one or two bits. In such cases, we will declare the variable as a char and we will not use the balance of the bits. However in some applications, we may

have to conserve memory space. In such cases, we can declare the size of the variable in terms of bits, for instance,

```
bool flag:1;
```

The above statement declares a variable flag of type bool. The size of the variable flag is 1 bit as indicated by the number following variable's name and preceded by a colon. Had it been 2, the size of flag will be 2 bits. In this way, we can put multiple variables on a single byte. For this purpose, such variables are bundled together as fields in a structure. An example is given below:

```
struct name {
    bool flag:1 ;
    int var:3 ;
    int:4 ;
} ;
```

The above declares three binary variables i.e. variables using binary digits or bits. In the last statement, we have not given any variable name and so the corresponding four bits will be unused. A member of the structure is defined to be a bit field by specifying the number of bits it will occupy, after a colon.

Bit field is not attractive because it increases the programming overheads. It is also more difficult to implement. Hence it is not advisable to use bit fields. On the contrary, dynamic allocation of memory can be used to overcome memory problems.

Function Overloading

C++ has an interesting feature called function overloading. By this feature, we can build a number of functions with the same name. But the argument lists of such functions have to be different and unique. However, the return data type of such functions need not be unique. Or in other words, function overloading is achieved through multiple functions with same names, but with unique argument list. An example of such overloaded functions are given below:

```
string add(string str1, string str2){
double add(double var1,double var2){
int add(int var1,int var2){
```

In the above example, there are three different functions with the same name add, but with different list of arguments. Since all are functions for add, this helps in readability of the code. Then how does it work? The compiler resolves the specific function to be called by looking at the number and type of arguments passed at the time of the call to a function. For instance, if we call add (4.5, 3.5), the function whose arguments match with the actual arguments, will be called. In this case, the function with arguments declared as double will be called. The example below illustrates function overloading.

Program 1.13

```
//To demonstrate function overloading
#include<iostream>
using namespace std;
```

```
inline string add(string str1, string str2){
    str1+=str2;
    return str1;
}
inline double add(double var1,double var2){
    var1+=var2;
    return var1;
}
inline int add(int var1,int var2){
    var1+=var2;
    return var1;
}
int main(){
    cout<<add(10, 5)<<"\n";
    cout<<add(56.10, 5.5)<<"\n";
    cout<<add("function ","overloading")<<"\n";
}
```

In the above example, the first print statement calls the function returning integer. Similarly, the next print statement calls the function returning double and the last statement calls the function returning string because both the arguments are strings.

The output of the program

```
15
61.6
function overloading
```

Rules for Overloading

Overloading is preferred when similar functions are to be carried out on different data types or when the number of arguments needed by the functions vary, although they may be of the same type. The arguments listed in the function header are called formal parameters or formal arguments. The arguments supplied along with the call to the functions are called actual parameters or actual arguments. While selecting the appropriate functions, a number of situations may arise as given below:

1. The arguments match exactly–in this case selecting the function is trivial.

2. Sometimes, the formal arguments and arguments passed will match with promotions. For instance,

    char to int

    short to int

    bool to int

    float to double

3. We can also match using standard conversion of the arguments as given below:

    long to unsigned long, int to double etc.

Thus, when there is no exact match for the arguments, promotions and conversions of data type are permissible.

Symbolic Constants and Macros

Both symbolic constants and macros are preprocessor directives. Both are defined using #define directive. We will see the difference between them later. But first let us look at symbolic constants. A symbolic constant is declared on top of the program. For instance,

```
#define PRINT cout
```

The above definition is a preprocessor directive. This means that the compiler shall substitute PRINT in the program with cout throughout, before compilation. The above definition improves readability. The symbolic constant in the above instance is 'PRINT'. Symbolic constants are given in upper case letters to differentiate them from other identifiers such as variables, function names etc. Some more examples of symbolic constants are given below:

```
#define SIZE 64
#define PI 3.14
#define NULL '\0'
#define FALSE 0
#define TRUE 1
#define PLUS +
#define MINUS -
```

Note the syntax carefully. There is no semicolon or comma. This is a directive to the compiler to substitute the symbolic constants with the specified tokens everywhere in the file except in strings and character constants.

Macro is a variation of symbolic constant. Macro is like an inline function. Therefore, it is defined with arguments. Macros can also carry out small functions as the following example illustrates.

Program 1.14

```
//To demonstrate macros
#include<iostream>
using namespace std;
#define ADD(a, b) (a+b)
int main(){
    cout<<ADD(10, 5)<<"\n";
    cout<<ADD(56.10, 5.5)<<"\n";
}
```

In the above example, the macro ADD(a, b) will be substituted with (a+b) before compilation. Using this we have added both integers and real numbers in the above program. Function overloading has also been achieved in the above program. Don't try to concatenate strings with macros. The compiler will flag an error.

The output of the program

```
15
61.6
```

The macro definition is reproduced below for further discussion.

```
#define ADD(a, b) (a+b)
```

From the above, we can generalize the macro definition as follows:

```
#define Name(arguments) to_be_substituted_with
```

Let us look at some more examples of macros:

```
#define MAX(a, b) ((a)>(b) ? (a):(b))

#define MIN(a, b) ((a)>(b) ? (b):(a))
```

Note the parentheses carefully. Thus macros carry out operations unlike symbolic constants.

Macros are popular in C. But they are rarely used in C++, since some programming tools are unable to handle them properly. For instance, some compiler implementations may not check the syntax after substitution. Since, function prototype is not applied to macros, checking of argument types may also not take place. It is also difficult to diagnose macros in case of run time errors. Thus, use of inline functions is preferred in C++.

1.7   MANIPULATION OF STRING USING OPERATORS

In C++, built-in operators cannot be used directly with string variables. However, some built-in functions, such as `strcmp`, `strcpy`, `strcat`, etc., are used to compare two strings, copy one string to another and concatenate two strings, respectively. For example, if `str1`, `str2`, `str3` are three string variables, then this statement generates a compile-time error.

```
        str3=str1+str2; //invalid
```

Note that other operators, such as <, ==, <=, etc., do not generate any compile-time error when used with strings however, do not give accurate results. Thus, these operators have to be overloaded for string manipulations.

Example 1.21

A program to demonstrate the concept of operator overloading for string manipulations.

```
#include<iostream>
#include<cstring>
using namespace std;
class  string_class
{
     char *str;
     int size;
public:
     string_class()
     {
          str = " ";
          size = 0;
     }
```

```
        string_class(char  *p)
        {
                strcpy(str,p);
                size = strlen(str);
        }
      void display()
      {cout<<str;}
      int operator<(string_class s);  //overloading
                                           < operator
      int operator==(string_class s); //overloading
                                          == operator
      string_class operator+(string_class s); //
      overloading + //operator
};
int  string_class::operator<(string_class  s)
{
      if (size == s.size)
      {
            if (strcmp(str,s.str)<0)
                  return 1;
            else
                  return 0;
      }
      else
            if (size<s.size)
                  return 1;
            else
                  return 0;
}
int  string_class::operator==(string_class  s)
{
      if (strcmp(str,s.str) == 0)
            return 1;
      else
            return 0;
}
string_class  string_class::operator+(string_class  s)
{
      string_class  s3;
      strcpy(s3.str,str);
      strcat(s3.str,s.str);
      return s3;
}
int main()
{
      string_class  str1("Hello"),str2("World"),str3;
      cout<<"First string is: ";
      str1.display();
      cout<<"\nSecond string is: ";
      str2.display();
```

```
          if (str1 == str2)    //calling operator==
          function
               cout<<"\nStrings are equal";
          else
               if (str1 < str2) //calling operator<
               function
                    cout<<"\nString 1 is less than string
               2";
               else
                    cout<<"\nString 2 is less than string
               1";
     str3 = str1 + str2;  //calling operator+
     function
     cout<<"\n\nConcatenated string is: ";
     str3.display();
     return 0;
}
```

The output of the program

```
First string is: Hello
Second string is: World
String 1 is less than string 2
Concatenated string is: HelloWorld
```

In this example, a class `string_class` containing a pointer `str` to an array of type `char` and a variable `size` of type `int`, is defined. Three operators `<`, `==` and `+` are overloaded to compare, to check the equality and to concatenate two objects of `string_class`, respectively.

Overloading using Friend Functions

In addition to member functions, an operator function can be defined as a friend function of the class for which it is being overloaded. The operator function defined as a friend function of the class is known as the *friend operator function.* Like other friend functions of the class, the friend operator function is declared inside the class, however, defined outside the class definition. The syntax to declare the friend operator function inside the class is as follows:

```
friend return_type operator op(parameter_list);
```

The syntax to define the friend operator function outside the class is as follows:

```
return_type operator op(parameter_list)
{
     //function body
}
```

Since the operator function is defined as a friend function of the class, it is invoked like an ordinary function, that is, without using the object name and the dot operator.

1.8 POLYMORPHISM AND STREAMS IN C++

C++ is a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or 'C with Classes', i.e., C++

introduces Object-Oriented Programming (OOP) features to C. It offers classes, which provide the four features commonly present in OOP and also in some non-OOP languages, namely abstraction, encapsulation, inheritance, and polymorphism.

1.8.1 Polymorphism

In C++ programming, polymorphism refers to the fact that the same entity (object or function) acts differently in different conditions. In Object-Oriented Programming (OOP), polymorphism is an essential and significant concept. The term 'Polymorphism' is made of two terms 'Poly' and 'Morphs', which means 'Multiple Types'.

Polymorphism enables one common interface for many implementations, and for objects to act differently under different circumstances. C++ supports several kinds of 'Static' (resolved at compile-time) and 'Dynamic' (resolved at run-time) polymorphisms. Compile-time polymorphism does not allow for certain run-time decisions, while runtime polymorphism typically incurs a performance penalty.

Static Polymorphism

Function overloading allows programs to declare multiple functions having the same name but with different arguments (i.e., ad hoc polymorphism). The functions are distinguished by the number or types of their formal parameters. Thus, the same function name can refer to different functions depending on the context in which it is used. The type returned by the function is not used to distinguish overloaded functions and differing return types would result in a compile-time error message.

When declaring a function, a programmer can specify one or more parameters as the default value. Consequently, the parameters with default values can optionally be omitted when the function is called, in this condition the default arguments will be used. When a function is called with fewer arguments then there are declared parameters, explicit arguments are matched to parameters in left-to-right order, with any unmatched parameters at the end of the parameter list being assigned their default arguments. In most of the cases, specifying default arguments in a single function declaration is desirable to provide overloaded function definitions with different numbers of parameters.

Templates in C++ provide a sophisticated mechanism for writing generic, polymorphic code (i.e., parametric polymorphism). Particularly, through the curiously recurring template pattern, it is possible to implement a form of static polymorphism that closely copycats the syntax for overriding virtual functions.

Dynamic Polymorphism

Variable pointers and references to a base class type in C++ can also refer to objects of any derived classes of that type. This permits arrays and other kinds of containers to hold pointers to objects of differing types, remember that the references cannot be directly held in containers. This enables dynamic (run-time) polymorphism, where the referred objects can behave differently, depending on their (actual, derived) types.

C++ also provides the dynamic_cast operator, which allows code to safely attempt conversion of an object, via a base reference/pointer, to a more derived type, the downcasting. The user must know which derived type is referenced. Upcasting, conversion to a more general type, can always be checked/performed at compile-time by means of static_cast, as ancestral classes are specified in the derived class's interface, visible to all. The dynamic_cast relies on Run-Time Type Information (RTTI), metadata in the program that enables differentiating types and their relationships. If a dynamic_cast to a pointer fails, the result is the nullptr constant, whereas if the destination is a reference, which cannot be null, the cast throws an exception. Objects known to be of a certain derived type can be cast to that with static_cast, bypassing RTTI and the safe runtime type-checking of dynamic_cast.

Therefore in C++, polymorphism means having many forms and usually polymorphism occurs when there is a hierarchy of the classes and they are related by the inheritance.

1.8.2 Streams

In C++, the stream refers to the stream of characters that are transferred between the program thread and I/O (Input/Output).

Stream classes in C++ are used to input and output operations on files and I/O devices. These classes include specific features for handling input and output of the program. The **iostream.h** library holds all the stream classes in the C++ programming language.

Basically, in C++ programming language, the 'Stream' is a flow of data into or out of a program, such as the data written to cout or read from cin. For this there are different classes, such as **istream** is a general purpose input stream, **cin** is an example of an **istream**, and **ostream** is a general purpose output stream.

Stream Class

The stream class in C++ has the following features:

- A C++ *class* is a collection of data and the methods essential for controlling and maintaining that data, it is done using **stream** classes.

- A C++ *object* is a specific variable having a class as its data type, the **cin** and **cout** are special pre-specified objects with different classes as their data types.

- A C++ **stream** is a flow of data into or out of a program, such as the data written to **cout** or read from **cin**.

- The **stream** class includes the following four different classes:

    o **istream** is a general purpose input stream, the **cin** is an example of an **istream**.

    o **ostream** is a general purpose output stream, **cout** and **cerr** both are the examples of **ostream**.

o **ifstream** is an input file stream. It is a special type of an **istream** that reads in data from a data file.

o **ofstream** is an output file stream. It is a special type of **ostream** that writes data out to a data file.

Object Oriented Programming (OOP), for example the C++ uses the concept *inheritance*. Inheritance is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar implementation. Also defined as deriving new classes (sub classes) from existing ones, such as super class or base class and then forming them into a hierarchy of classes. In inheritance, typically the classes *inherit* the properties of previously written classes. The descendant classes then add on additional properties, therefore making them *specializations* of their parent class.

Input/Output (I/O) Stream in C++

C++ Input/Output (I/O) streams are primarily defined by **iostream**, a header file that is part of the C++ standard library (the name stands for Input/Output Stream). In C++ and its predecessor, the C programming language, there is no special syntax for streaming data input or output. Instead, these are combined as a library of functions. Like the **cstdio** header inherited from C's **stdio.h**, **iostream** provides basic input and output services for C++ programs. **iostream** uses the objects **cin**, **cout**, **cerr**, and **clog** for sending data to and from the standard streams input, output, error (unbuffered), and log (buffered), respectively. As part of the C++ standard library, these objects are a part of the **std** namespace. A namespace is a set of signs (names) that are used to identify and refer to objects of various kinds. A namespace ensures that all of a given set of objects have unique names so that they can be easily identified.

The **cout** object is of type **ostream**, which overloads the left bit-shift operator to make it perform an operation completely unrelated to bitwise operations, and notably evaluate to the value of the left argument, allowing multiple operations on the same **ostream** object, essentially as a different syntax for method cascading, exposing a fluent interface. The **cerr** and **clog** objects are also of type **ostream**, so they overload that operator as well. The **cin** object is of type **istream**, which overloads the right bit-shift operator. The directions of the bit-shift operators make it seem as though data is flowing towards the output stream or flowing away from the input stream.

Check Your Progress

9. What is type conversion?
10. What are integral expressions?
11. Define the term tokens.
12. What is derived data types?
13. What do you mean by binary operators?
14. Define the term friend operator function.
15. What is stream in C++?

1.9 ANSWERS TO 'CHECK YOUR PROGRESS'

1. The most important feature is that, unlike procedural programming in which the program is divided into a number of functions, OOP divides the program into a number of objects.

2. A class is defined as a user-defined data type which contains the entire set of similar data and the functions that the objects possess. In other words, a class in OOP represents a group of similar objects. As stated earlier, in the real world millions of objects exist and each of them has its own identity. However, each of them can be categorized under different groups depending on the common properties they possess and the functions they perform.

3. The class, which is inherited by the other classes, is known as superclass or base class or parent class.

4. In C++, polymorphism can be achieved either at compile-time or at run-time. At compile-time, polymorphism is implemented using operator overloading and function overloading. However, at run-time, it is implemented using virtual functions.

5. C is a procedure-based language. Once you write a program in C, you must run it through a C compiler to turn your program into one that a computer can run (execute). C allows the input and output control in which a user can input the value to get the desired result.

6. Programs are a sequence of instructions or statements. These statements form the structure of a C++ program.

7. A single statement specifies a single action and is always terminated by a semicolon ';'. A compound statement, also known as a block, is a set of statements that are grouped as a compound statement and are always enclosed within curly braces '{}'.

8. The statements that cause a set of statements to be executed repeatedly either for a specific number of times or until some condition is satisfied are known as iteration statement.

9. An expression may involve variables and constants either of same data type or of different data types. If an expression consists of mixed data types, then they must be converted to the same type while evaluation to avoid compatibility issues. This is accomplished is called type conversion.

10. The expressions that produce an integer value as an output after performing all the type of conversions are called integral expressions.

11. A token is defined as the smallest unit of a program. When a program is compiled, the compiler scans the source code and parses it into tokens to find the syntax errors.

12. Data types that are derived from the built in data types are knows as derived data types.

13. We need two operands for any of the basic arithmetic operation, such as addition, subtraction, division and multiplication. Since these operators need two operands, the operators are called binary operators.

14. An operator function can be defined as a friend function of the class for which it is being overloaded. The operator function defined as a friend function of the class is known as the friend operator function.

15. In C++, the stream refers to the stream of characters that are transferred between the program thread and I/O (Input/Output). Stream classes in C++ are used to input and output operations on files and I/O devices. These classes include specific features for handling input and output of the program. The `iostream.h` library holds all the stream classes in the C++ programming language.

1.10 SUMMARY

- The most important feature is that, unlike procedural programming in which the program is divided into a number of functions, OOP divides the program into a number of objects.

- Objects are the small, self-contained and modular units with a well-defined boundary.

- A class is defined as a user-defined data type which contains the entire set of similar data and the functions that the objects possess. In other words, a class in OOP represents a group of similar objects.

- Abstraction is a mechanism to hide irrelevant details and represent only the essential features, so that one can focus on important things at a time.

- Inheritance can be defined as the process whereby an object of a class acquires characteristics from the object of another class. As stated earlier, all the objects of a similar kind are grouped together to form a class.

- Inheritance allows code reusability, that is, it facilitates classes to reuse the existing code. It is useful when several classes having similar features are to be created.

- The class, which is inherited by the other classes, is known as superclass or base class or parent class.

- In C++, polymorphism can be achieved either at compile-time or at run-time. At compile-time, polymorphism is implemented using operator overloading and function overloading. However, at run-time, it is implemented using virtual functions.

- Programs are a sequence of instructions or statements. These statements from the structure of a C++ program.

- Standard headers are specified in a program through the preprocessor directive #include.

- In C programming language, the 'Tokens' are considered as the most significant concept typically used for developing a C program. Fundamentally,

the tokens in C language are referred as the building block of C programming language.

- A single statement specifies a single action and is always terminated by a semicolon ';'. A compound statement, also known as a block, is a set of statements that are grouped as a compound statement and are always enclosed within curly braces '{}'.

- The statements that cause a set of statements to be executed repeatedly either for a specific number of times or until some condition is satisfied are known as iteration statements.

- An expression may involve variables and constants either of same data type or of different data types. If an expression consists of mixed data types, then they must be converted to the same type while evaluation to avoid compatibility issues. This is accomplished is called type conversion.

- The expressions that produce an integer value as an output after performing all the type of conversions are called integral expressions.

- A token is defined as the smallest unit of a program. When a program is compiled, the compiler scans the source code and parses it into tokens to find the syntax errors.

- A data type determines the type and the operations that can be performed on the data. C++ provides various data types and each data type is represented differently within the computer's memory.

- Data types that are derived from the built in data types are knows as derived data types.

- We need two operands for any of the basic arithmetic operation such as addition, subtraction, division and multiplication. Since these operators need two operands, the operators are called binary operators.

- An operator function can be defined as a friend function of the class for which it is being overloaded. The operator function defined as a friend function of the class is known as the friend operator function.

1.11 KEY TERMS

- **Unstructured programming paradigm:** In this type of programming, all the instructions of a program are written one after the other in a single function and hence, suitable for writing only small and simple programs.

- **Object:** A unit of structural and behavioral modularity that contains a set of properties (or data) as well as the associated functions.

- **State:** State of an object is one of the possible conditions that an object can exist in and is represented by its characteristics or attributes or data.

- **Class:** It is defined as a user-defined data type which contains the entire set of similar data and the functions that the objects possess.

- **Inheritance:** It can be defined as the process whereby an object of a class acquires characteristics from the object of another class.

- **Abstract class:** It refers to a class which provides only the interface of one or more functions and not their implementations.

- **Statement:** It refers to an instruction given to the computer to perform a specific action.

- **Type conversion:** It is the process of converting one data type to another.

- **Expression:** It is a combination of variables, constants and operators that represents a computation.

- **Token:** The smallest unit of a program, C++ tokens are broadly classified into keywords, identifiers, constants, operators and punctuators.

- **Arithmetic operators:** Arithmetic operators refer to addition, subtraction, multiplication, division, modulus.

- **Relational operators:** The relational operators are used to check the relationship between two numeric operands or expressions. These are greater than, less than, greater than or equal, equal, not equal.

- **Bit wise operators:** Bit wise operators access the internal representation of the numbers in bits, viz., bit 0 and 1.

1.12 SELF ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. What are the various features of OOP?
2. Differentiate between a class and an object.
3. Define the term concrete class.
4. What do you understand by function overloading?
5. How can the entities of a namespace be accessed?
6. Mention the types of tokens in C programming.
7. Write in brief about the control statements.
8. How to passing nesting structure to function.
9. What do you mean by advanced type casting?
10. State the three types of special assignment expressions.
11. What is the use of scope resolution operator (::) in C++?
12. What do you understand by the term built-in data types?
13. What is declaration of variables?
14. Define the term unary operators.
15. Write in brief about the manipulation of strings using operators.

Long-Answer Questions

1. Explain inheritance in detail explaining its property of reusability and extensibility.

2. Explain polymorphism with the help of an example and differentiate between compile-time and run-time polymorphism.

3. Describe the objects with the help of diagram.

4. Differentiate between C and C++ with the help of example.

5. Discuss the standard streams defined in the iostream header.

6. Discuss initialization input with C in tokens with the help of example.

7. Describe the categories of control statements with appropriate examples.

8. Describe the decisions nesting with the help of examples.

9. Discuss the type conversion and its types with the help of example.

10. Explain the operators and expressions with the help of example.

11. Describe tokens and its types.

12. Illustrate various data types in C++ with the help of diagram.

13. Explain the operators and its types with the help of example.

14. Write a program to demonstrate the concept of operator overloading for sting manipulations.

15. Explain the polymorphism and streams in C++.

1.13 FURTHER READING

Jeyapoovan, T. 2006. *Computer Programming: Theory and Practice* (with CD). New Delhi: Vikas Publishing House.

Khurana, Rohit. 2008. *Object Oriented Programming with C++*. New Delhi: Vikas Publishing House.

Saxena, Sanjay. 2009. *Introduction to Information Technology.* New Delhi: Vikas Publishing House.

Rumbaugh, James, Fedrick Blaha, William Premerlani, and Federick Eddy.1990. *Object- Oriented Modelling and Design.* New Jersey: Prentice Hall.

Balaguruswamy, E. 1998. *Object-Oriented Programming*. New Delhi: Tata McGraw-Hill.

UNIT 2 FUNCTIONS, CLASS AND OBJECTS IN C++

Structure

2.0 INTRODUCTION

A function is a program with a group of statements performing specific operations. The behavior of objects is implemented through functions. The dynamic properties of objects are facilitated only through functions. Thus, functions provide interfaces to communicate with the objects. Whenever a function is invoked, a set of operations is performed which includes passing the control from the calling function to the called function, managing stack for arguments and return values, managing registers, etc. All these operations take much of compiler time and slow down the execution process. This overhead can be avoided by making function calls execute faster and to perform type checking to make the function inline.

When a function is invoked, the control passes from the calling function to the called function. Consequently, the called function body is executed and the control returns to the calling function either when a return statement is encountered or the end of the calling function is reached. If a return statement is encountered, a function returns either a value or a reference to the calling function. You will also learn about function overloading. In some cases when a similar action is to be performed on different types of data, different functions having different names are to be defined for all types of data. This approach makes the program very complex as the programmer must keep a track of the names of all the functions defined in the program. To prevent such situations, C++ allows the functions to be overloaded. Class is a definition of an object. All class members are private. A class is a type and an object of the class is a variable. In C++, the data and functions (procedures to manipulate the data) are worked together as a self-contained unit called an object.

In this unit, you will study about the main function, passing arguments to function, returning value from functions, overload functions, inline functions, default arguments, class and objects, concept of a class, and classes versus objects.

2.1   OBJECTIVES

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

- Understand the basics of main function
- Discuss the passing arguments to function
- Explain the returning values from functions
- Describe the overload functions, inline functions and default arguments
- Discuss the basics of class and objects

2.2   MAIN FUNCTION

Functions mean operations. A **function** is a program with a group of statements performing specific operations. The function main is special and important, not only in C++, but also in C, Java and C #(Sharp). The behavior of objects is implemented through functions. The dynamic properties of objects are facilitated only through functions. Thus, functions provide interfaces to communicate with the objects.

User-Defined vs Library Functions

As the name indicates, user-defined functions are coded by the user for his specific requirement. The standard library provides a number of library functions. The C functions such as printf () and scanf() are available for C++ programs. In addition to library functions, the user can develop additional functions. In this unit, user-defined functions will only be discussed.

Function Declaration—Prototype

A function has to be declared before using it in a manner similar to variables and constants. A function may be declared either in the main function or in a class. The function declaration has to be done in the standard format known as function prototype. The general format of a function prototype is given below:

```
    return_data_type function_name (type of argument 1,
type of argument 2,…. );
```

Note the semicolon at the end of the declaration is similar to declaration of other data types. A function, after execution, may return a value to the function that called it. It may return an integer, character or float value. It may also not return a value at all, but may perform some operations. For instance, if it returns a float value, you may declare a function as:

```
    float func1(float arg 1, int arg 2);
```

If it does not return any value at all, you may declare the function as:

```
  void func2(float arg1, int arg2) ;         /*void
means nothing*/.
```

Even if no arguments are passed into a function, empty parentheses must follow the function name as illustrated below:

```
  char func4();
```

While defining arguments, the data type of the argument and the name of the variable – for instance, `float arg1` is given, where `float` is the data type and `arg1` is the name of the variable. The name given is just a dummy and it does not serve any useful purpose except for better readability. The compiler simply ignores the names of arguments in the prototype. You can even omit it and write as:

```
  char func3(float, int);
```

The programmer can use either method.

2.2.1 Passing Arguments to Function

The objects of a class can be passed as arguments to member functions as well as non-member functions either by value or by reference. When an object is passed by value, a copy of the actual object is created inside the function. This copy of object is destroyed when the function terminates. Moreover, any changes made to the copy of the object inside the function are not reflected in the actual object. On the other hand, in pass by reference, only a reference to that object (not the entire object) is passed to the function. Thus, the changes made to the object within the function are also reflected in the actual object.

Whenever an object of a class is passed to a member function of the same class, its data members can be accessed inside the function using the object name and the dot operator. However, the data members of the calling object can be directly accessed inside the function without using the object name and the dot operator.

Note: The non-member functions can access only the public members of the class with the help of objects passed as arguments and the private data members are not accessible to them.

Program 2.1: A program to demonstrate passing objects by value to a member function of the same class

```
#include<iostream>
using namespace std;
class weight
{
    int kilogram;
    int gram;
public:
    void getdata();
    void putdata();
```

```
                            void sum_weight(weight,weight);
};
void weight :: getdata()      //input weight from user
{
    cout<<"\nKilograms: ";
    cin>>kilogram;
    cout<<"Grams: ";
    cin>>gram;
}
void weight :: putdata()      //display weight
{
    cout<<kilogram<<" Kgs. and "<<gram<<" gms.\n";
}
//passing objects by value
void weight :: sum_weight(weight w1,weight w2)
{
    gram = w1.gram + w2.gram;
    kilogram=gram/1000;
    gram=gram%1000;
    kilogram+=w1.kilogram+w2.kilogram;
}
int main()
{
    weight w1,w2,w3;
    cout<<"Enter weight in kilograms and grams\n";
    cout<<"\nEnter weight #1";
    w1.getdata();                 //input weight1
    cout<<"\nEnter weight #2";
    w2.getdata();                 //input weight2
    w3.sum_weight(w1,w2);        //add two weights
    cout<<"\nWeight #1 = ";
    w1.putdata();                 //display weight1
    cout<<"Weight #2 = ";
    w2.putdata();                 //display weight2
    cout<<"Total Weight = ";
    w3.putdata();                 //display total weight
    return 0;
}
```

The output of the program

```
Enter weight in kilograms and grams
  Enter weight #1
  Kilograms: 12
  Grams: 560
```

```
Enter weight #2
Kilograms: 24
Grams: 850

Weight #1 = 12 Kgs. and 560 gms.
Weight #2 = 24 Kgs. and 850 gms.
Total Weight = 37 Kgs. and 410 gms.
```

In this program, the `sum_weight()` function has direct access to the data members of calling object (w3 in this case). However, the members of the objects passed as arguments (w1 and w2) can be accessed within function using the object name and the dot operator. Note that the objects w1 and w2 are passed by value; however, they can be passed by reference also. For example, to pass w1 and w2 by reference to the function `sum_weight()` (defined in **Program 2.1**)**,** the function will be declared and defined as follows:

```
//function prototype inside the class
void sum_weight(weight &,weight &); //pass by reference
//function definition outside the class
void weight :: sum_weight(weight &w1,weight &w2)
{
.    //body of function
.    //as defined in Example 2.1
}
```

Figure 2.1 shows accessing of member variables inside the sum_weight() function.
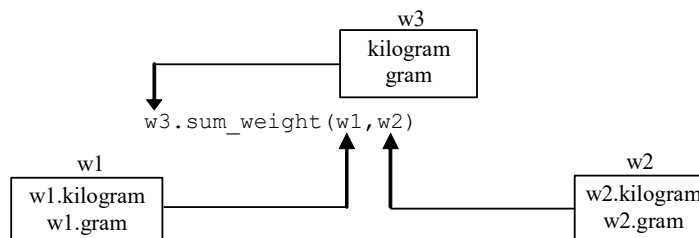
Fig. 2.1 *Accessing Data Members within Called Member Function*

2.2.2 Returning Value from Functions

When a function is invoked, the control passes from the calling function to the called function. Consequently, the called function body is executed and the control returns to the calling function either when a return statement is encountered or the end of the calling function is reached. If a return statement is encountered, a function returns either a value or a reference to the calling function.

The `return` statement is a jump statement that unconditionally passes the control out of a called function to its calling function. The syntax of a `return` statement is as follows:

```
return value;
```

where,

> `return` = A C++ keyword.
>
> `value` = A variable, constant, expression or a reference which may or may not be provided with `return`.

Return by Values

If the return type of a function is `void`, the called function terminates and returns the control when it encounters the closing curly brace (`}`) or a `return` statement with no arguments. However, if the return type is not `void`, the called function terminates when it encounters the first `return` statement with an argument. The function with non-`void` as its return type must have at least one `return` statement otherwise a compilation error is raised. If no specific value is returned through the `return` statement, a garbage value is returned.

Example 2.1

A program to demonstrate the concept of functions returning values.

```
 #include<iostream>
    using namespace std;
int average(int,int,int);
int main()
{
int a,b,c,r;
cout<<"Enter three numbers :\n";
cin>>a>>b>>c;
r=average(a,b,c);
cout<<"The average of three numbers is: "<<r;
return 0;
}
int average(int x,int y,int z)
{
int avg;
avg=(x+y+z)/3;
return avg;
}
```

The output of the program

```
Enter three numbers
78 36 42
```

```
The average of three numbers is: 52
```

In this example, the three variables `a`, `b` and `c` are passed as arguments to `average()` which calculates the average and returns its value to `main()` with the help of the statement `return avg.` This returned value is then assigned to the variable `r` in the calling function.

Note: Calls to functions with a `void` return type cannot be used directly in `cout` statements or as operands in an expression as they do not explicitly return a value.

Return by Reference

In addition to returning values, a function can return references as well. Returning references is useful in those situations when the function call is to be used as an `Lvalue`. An `Lvalue` is an expression that can appear on the left-hand side of an equals-to sign. In other words, a function returning a reference can be placed on the left-hand side of the assignment statement. To return a reference, the reference operator (`&`) is suffixed to the return type in the function prototype and definition.

Example 2.2

A program to demonstrate the concept of returning a reference.

```cpp
#include<iostream>
using namespace std;
char &change(int i);        // return a reference
char arr[50] = "Hello World";
int main()
{
 change(5);     //simple call
 cout<<arr<<endl;
 change(5) = 'X';// using function call as an Lvalue
 cout << arr;
 return 0;
}
char &change(int i)
{
    return arr[i];
}
```

The output of the program

```
Hello World
HelloXWorld
```

In this example, the first function call is a simple call. However, the second function call appears on the left-hand side of an assignment statement and '`X`' is assigned to it by specifying it on the right-hand side.

Returning Values from main()

Unlike C, C++ enables to return values from `main()`. However, it can only return a value of type `int`. After specifying the return type, the function header of `main()` can be written as

```cpp
int main()
```

It can also be written as

```cpp
int main(int argc, char **argv)
```

When the execution of the program gets over, it returns either a zero or non-zero value. This value is passed to the operating system to indicate either the success of the program or the error. Generally, a return value of zero indicates success and a non-zero value indicates failure or error.

Note: Most C++ compilers display a warning 'Function should return a value' if `return` statement is not specified in `main()`.

Recursion

Generally, a function definition makes a call to other functions, however, a C++ function can call itself. When a function definition includes a call to itself, it is referred to as a **recursive function** and the process is known as **recursion** or **circular definition**.

When a recursive function is called for the first time, a space is set aside in the memory to execute this call and the function body is executed. Then a second call to a function is made; again a space is set for this call and so on. In other words, memory spaces for each function call are arranged in a stack. Each time a function is called, its memory area is placed on the top of the stack and then is removed when the execution of the call is completed (Refer Figure 2.2).

(a) Memory allocation

(b) Memory allocation
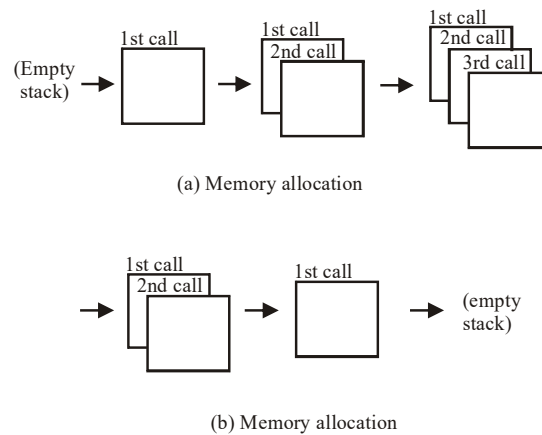
Fig. 2.2 *Calling Recursive Functions*

Note: A stack is a 'Last-In-First-Out structure.

Example 2.3

A program to demonstrate the concept of recursive function.

```
#include<iostream>
   using namespace std;
void reverse();
   int main()
   {
reverse();
cout<<" is the reverse of the entered characters";
return 0;
}
void reverse()
{
char ch;
   cout<<"Enter a character ('/' to end program) : ";
   cin>>ch;
if (ch != '/')
```

```
      {
      reverse();
      cout<<ch;
      }
  }
```

The output of the program

```
Enter a character ('/' to end program) : h
Enter a character ('/' to end program) : i
Enter a character ('/' to end program) : /
ih is the reverse of the entered characters
```

In this example, function `reverse()` is called to accept a character from the user. The function `reverse()` calls itself again and again until the user enters '/' and prints the reverse of the characters entered.

Note: A recursive function must include a condition or a statement to terminate the function.

Note that the recursive functions can also be defined iteratively using `for`, `while` and `do...while` loops. This is because recursion makes the program execution slower due to its extra stack manipulation and more memory utilization. In addition, recursion sometimes results in stack overflow as for each function call new memory space is allocated to local variables and function parameters on the stack. However, in some cases, recursive functions are preferred over their iterative counterparts as they make code simpler and easier to understand. For example, it is easier to implement Quicksort algorithm using recursion.

Note: Recursive function can be declared as inline where the number of calls is known. The compiler cannot generate the inline code at compile time if the number of recursion is unknown till runtime.

2.3 OVERLOAD FUNCTIONS

In some cases when a similar action is to be performed on different types of data, different functions having different names are to be defined for all types of data. This makes the program very complex as the programmer must keep a track of the names of all the functions defined in the program. To prevent such situations, C++ allows the functions to be overloaded.

Overloading affirms the role of a single entity for multiple tasks. Function overloading is a way to implement compile-time polymorphism that allows multiple functions to share the same name with different parameters. The compiler identifies the function either on the basis of the number of parameters, the data type of the parameters or the order of the data type of the parameters passed to the function. Moreover, functions with different return type but with similar function signature are not considered as overloaded functions.

Note: Constructors are the most commonly overloaded functions and `main()` is the only function that cannot be overloaded.

To understand the concept of function overloading, consider these function declarations.

```
void func(int);
void func(int,int);
```

In these statements, two functions named, `func()` are different as the number of arguments passed are different. Now, consider these statements.

```
void func(int);
void func(char);
```

In these statements, two functions named, `func()` are different as the data type of arguments passed is different. Now, consider these statements.

```
void func(int,float);
void func(float,int);
```

In these statements, two functions named, `func()` are different as the order of the data type of the arguments passed is different.

The compiler follows these steps to perform comparison between the actual and the formal arguments to find the best match (the most appropriate overloaded function).

- **Exact Match:** If the number and type of the arguments exactly match the number and type of parameters of any one of the overloaded functions then that function is called.

- **Match through Type Promotions:** If an exact match is not found, the compiler tries to promote the type of the argument to the type of the parameter of an overloaded function. For example, if the argument is of type `char`, the compiler promotes it to type `int` or to the equivalent type `unsigned int`.

- **Match through Standard Conversions:** If a match is also not found after performing type promotions, the compiler tries to convert type of the argument to type of the parameter through standard conversion rules. For example, if the argument is of type `int`, the compiler can convert it to type `float`, `double` or `long double`.

- **Match through User Defined Conversions**: If a match is also not found after performing standard type conversions, the compiler tries to convert type of the argument to type of the parameter through the user defined conversions.

- **Match through Ellipsis:** If a match is not found after performing the previous steps, the compiler tries to find an ellipsis in the definition (or declaration) of any of the overloaded functions.

Example 2.4

A program to demonstrate the concept of function overloading.

```
#include<iostream>
#include<math>
using namespace std;
int area(int); //function prototype 1
float area(float, float);   //function prototype 2
float area(float, float, float); //function prototype 3
```

```
int main()
{
int side;
float length, width, a, b, c;
cout<<"Enter side of a square : ";
cin>>side;
cout<<"Area of a square is : "<<area(side)<<endl;
cout<<"Enter length and width of a rectangle : ";
cin>>length>>width;
cout<<"Area of a rectangle is :
"<<area(length,width)<<endl;
cout<<"Enter three sides of a triangle : ";
cin>>a>>b>>c;
cout<<"Area of a triangle is : "<<area(a, b,
c)<<endl;
return 0;
}
int area(int s)
{
return (s*s);
}
float area(float len, float wid)
{
return (len*wid);
}
float area(float a, float b, float c)
{
    float s,area;
    s=(a+b+c)/2;
    area=sqrt(s*(s-a)*(s-b)*(s-c));
    return area;
}
```

The output of the program

```
Enter side of a square : 12
  Area of a square is : 144
  Enter length and width of a rectangle : 34.5 67.7
  Area of a rectangle is : 2335.65
  Enter three sides of a triangle : 12.5 20.5 25.5
  Area of a triangle is : 126.791
```

In this example, three functions are declared with the same name `area()`, however, accepting different arguments. The compiler examines the number, type of arguments or order of data type in the function call and calls the appropriate function.

In some situations, when only number of arguments is different then instead of function overloading, default arguments can be used. This is because default arguments combine different operations into one function and hence, reduce the overhead of writing multiple functions.

Example 2.5

A program to demonstrate the use of default arguments as an alternative to function overloading.

```cpp
#include<iostream>
void disp(char='*',int=4);
int main()
{
      disp();
      disp('^');
      disp('@',5);
      return 0;
}
void disp(char ch, int n)
{
      for(int i=0; i<=n; i++)
      {
            for(int j=0; j<i; j++)
            {
                  cout<<ch;
                  cout<<" ";
            }
            cout<<endl;
      }
}
```

The output of the program

```
*
*  *
*  *  *
*  *  *  *
^
^  ^
^  ^  ^
^  ^  ^  ^
@
@  @
@  @  @
@  @  @  @
@  @  @  @  @
```

In this example, three calls with different number of arguments are made to the same function void disp(char='*',int=4). In this way, default arguments are used to reduce number of functions and hence, serve as an alternative to function overloading.

Note: Overloading of unrelated functions should be avoided.

Example 2.6: A program to demonstrate function overriding

```
#include<iostream>
using namespace std;
const int MAX=5;

class queue
{
   public :
int q[MAX];
int rear;
    int front;
 public:
queue()
     {
rear=-1 ; front=-1;
}
    void qinsert(int qu)
     {
q[++rear]=qu;
    if(front=-1)
front++;
     }
    void display()
     {
       for (int i=front;i<=rear;i++)
          cout<<q[i]<<endl;
     }
    void qdelete()
     {
cout<<"Element deleted :"<<q[front++]<<endl;
}
};
class queue2:public queue
{
   public :
void qinsert(int var)
{
      if (rear==MAX-1)
         cout<<"\nQUEUE FULL\n\n";
      else
         queue::qinsert(var);
}
void qdelete()
{
```

```
                if(front==MAX)
                {
                    cout<<"\nQUEUE EMPTY";
                }
        else
                queue::qdelete();
}
};
int main()
{
    queue2 q1;
    q1.qinsert(10);
q1.qinsert(11);
    q1.qinsert(17);
    q1.qinsert(23);
    q1.qinsert(25);
    q1.display();
    q1.qinsert(34);    // Array is Full element cannot be
added

q1.qdelete();
    q1.qdelete();
    q1.qdelete();
    q1.qdelete();
    q1.qdelete();

q1.qdelete(); // No elements present
return 0;
}
```

The output of the program is

```
10
11
17
23
25
QUEUE FULL
Element deleted :10
Element deleted :11
Element deleted :17
Element deleted :23
Element deleted :25

QUEUE EMPTY
```

In Example 2.6, the derived class queue2 inherits the base class `queue`. The derived class `queue2` redefines the member functions `qinsert()` and `qdelete()` of the base class `queue`. The base class `queue` member functions insert and delete the elements of the array. However, the base class does not check the maximum number elements that an array can contain, hence, undesired result can appear. Therefore, the derived class overrides the member function `qinsert()` by first performing the check and then calling the base class `qinsert()` (`queue::qinsert()`) to insert the elements in the list. Similarly, `qdelete()` member function of derived class overrides the base class function `qdelete()`.

Check Your Progress

1. What is a function?

2. How can the objects of a class be passed?

3. What is Lvalue?

4. What is circular definition?

5. Define the term function overloading.

6. Write one restriction about main() function.

2.4 INLINE FUNCTIONS

Whenever a function is invoked, a set of operations is performed which includes passing the control from the calling function to the called function, managing stack for arguments and return values, managing registers, etc. All these operations take much of compiler time and slow down the execution process. This overhead can be avoided by using macros in a program. However, macros are not considered as true functions, as they do not perform type checking. Another way to make function calls execute faster and also perform type checking is to make the function inline.

An **inline function** is a function whose code is copied in place of each function call. In other words, each call to inline function is replaced by its code. Inline functions can be declared by prefixing the keyword inline to the return type in the function prototype. An inline function 'Requests' the compiler to replace its each and every call by the code in its body. That is, specifying a function as inline is just a request to the compiler and not a command. So, it does not change the behaviour of a function. Moreover, the compiler may or may not choose to replace each call by the body. In case, it does, the function becomes 'in line' with the rest of the source code.

The syntax for inline function declaration is as follows:

```
inline return_type function_name(parameter_list);
```

A difference between a normal function call and inline function call is shown in Figure 2.3.

```
#include<iostream>
using namespace std;
void prod()int, int);
```

Calling Function ──────────▶ `int main()`
```
{
   int a, b;
   a = 10;
   b = 20;
     prod(a, b);
}
```
Transfer control to body

Called Function ──────────▶ `void prod(int x, int y)`
```
{
     cout<< "The product of number is"<<(x*y)'
}
```

(a) Normal Function Call

```
#include<iostream>
using namespace std;
void prod()int, int);
```

Calling Function ──────────▶ `int main()`
```
{
   int a, b;
   a = 10;
   b = 20;
     prod(a, b);
}
```
Replaces call with body

Called Function ──────────▶ `void prod(int x, int y)`
```
{
     cout<< "The product of number is"<<(x*y)'
}
```
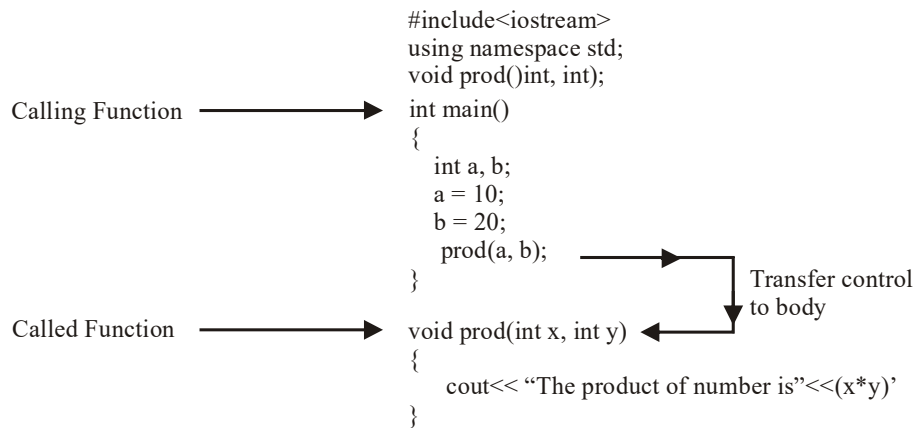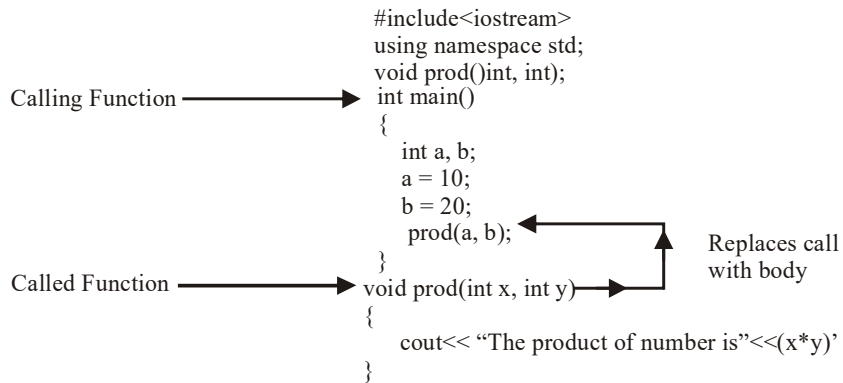
(b) Inline Function Call

Fig. 2.3 *Difference between a Normal and Inline Function Call*

In Figure 2.3(a), when the compiler reads the statement `prod(a,b)`, it transfers the control to the `prod()` function. However, in Figure 2.3(b), the `prod()` function is declared as an inline function. As a result, when the compiler reads the statement `prod(a,b)`, it replaces the function call with the definition of the `prod()` function.

Inline functions are ideal for functions that are small in size and frequently used by the programs. This is because inline functions reduce the time consumption and overhead involved in function calls. However, they also significantly increase the size of the program which in turn may adversely affect the readability of the program. Also, these functions restrict the portability of the program.

Note: The functions containing static variables, loops or switch statements cannot be inlined.

Example 2.7

A program to demonstrate how inline function works.
```
#include <iostream>
using namespace std;
int multiply (int);
void main( )
{
```

```
     int x;
     cout << "\n Enter the Input Value: ";
     cin>>x;
     cout << "\n The Output is: " << multiply (x);
}
inline int multiply(int x1)
{
     return 5*x1;
}
```

The output of the program

```
Enter the Input Value: 10
The Output is: 50
```

Program 2.2

```
//To demonstrate inline functions
#include<iostream>
using namespace std;
inline string conc(string str1, string str2){
    str1+=str2;
    return str1;
}
int main(){
    string s1="Programming ";
    string s2="in C++";
    cout<<conc(s1, s2)<<"\n";
    cout<<conc("ya ", "it works");
}
```

Result of Program 2.2

```
Programming in C++
ya it works
```

The function 'conc' in the above example concatenates two strings.

2.5 DEFAULT ARGUMENTS

Whenever a function is called, the calling function must provide all the arguments specified in the function's declaration. If the calling function does not provide the required arguments, the compiler raises an error. However, C++ allows a function to be called without specifying all of its arguments. This can be achieved by assigning a default value to the argument. Default value is specified in the function declaration and is used when the value of that argument is not passed while calling the function. The syntax for providing a default value to an argument is as follows:

```
return_type function_name(type param1=value1,…… type
paramN=valueN);
```

Note: Default arguments appear only in function declarations and not in function definitions.

Default arguments are used in situations where the value of an argument is same in most of the function calls. Default arguments provide a lot of flexibility during function calls. If a function call does not specify an argument, the default value is passed as an argument to the function. In case a function call specifies an argument, the default value is overridden and the specified value is passed to the function.

Example 2.8

A program to demonstrate the concept of default arguments.

```
#include<iostream>
using namespace std;
 float calc(float tax, float originalval, float
disc=25.50);
int main()
{
int price;
price=calc(4.25,1250.50); //default argument missing
cout<<"Calculated price with default discount value: ";
cout<<price<<endl;
price=calc(4.25,1250.50,15.25);
//default argument overridden
cout<<"Calculated price with assigned value : "<<price;
return 0;
}
float calc(float tax, float originalval, float disc)
{
float discvalue=originalval*disc/100;
float taxvalue=tax/100;
float price=originalval-discvalue+taxvalue;
return price;
}
```

The output of the program

```
Calculated price with default discount value : 931.665
Calculated price with assigned value : 1059.84
```

In this example, the function declaration of `calc()` specifies a default value for the third argument `disc`. In the first call to `calc()`, only two arguments (`4.25`, `1250.50`) are passed. As a result, the third argument `disc` is assigned the default value `25.50`. In the second call to `calc()`, the third argument is passed explicitly, and hence, it overrides the default value.

Note that all the default arguments must be specified at the end of the parameter list. If any default value of an argument is to be omitted, it should be omitted from the end and not in-between the argument list. For example, consider the declaration of the function `average()`.

```
average(int, int, int = 5,int = 10,int = 15,int = 20);
```

Some of the invalid function calls to `average()` are given here.

- `average(1,2, ,4, ,6);`
- `average(1,2,3,4, ,6);`

Since default arguments are missing from in-between the list, an error is raised during the compilation process.

Some of the valid function calls to `average()` are given here.

- `average(1,2,3,4,5);`
- `average(1,2,3,4);`
- `average(1,2);`

Since default arguments are missing from the end of the list, no error is raised during the compilation process.

Structure As Arguments

Like ordinary variables, structure variables can also be passed by value to a function.

Example 2.9

A program to demonstrate the concept of structure variables as arguments.

```
#include<iostream>
using namespace std;
struct details
{
int pcode;
 float qty;
 float amt;
 float total;
};
void calc(details);
int main()
{
 details d1={1,15,100,0}; //initializing structure
 calc(d1);
 cout<<"Total amount in main(): "<<d1.total;
 return 0;
}
void calc(details dd)
{
    dd.total=dd.amt*dd.qty;
  cout<<"Total  amount  in  called  function  :
"<<dd.total<<endl;
}
```

The output of the program

```
Total amount in called function : 1500
Total amount in main(): 0
```

In this example, the structure variable `d1` of structure details is passed by value to function `calc()`. The function `calc()` calculates the total amount and displays it. Since the structure variable is passed by value, the new value of the data member total of structure details is not reflected in `main()`.

2.6 OBJECT AND CLASSES

Class

Class in C++ is similarly a framework for user defined data type. Class and structure provide convenient tools to the programmer to build their own type. These types are convenient to represent real entities. They are useful because built in types cannot be easily used to represent real entities. A structure or class is an aggregate of built in types. Thus, they are quite handy to represent various real entities like bank account, student record, payroll, etc.

Class Definition

A class definition is similar to a structure. A structure can also be built with data elements and functions as in the above example. However, rarely are structures built with functions. A class will have declaration of data elements as well as functions. A class has a name or a tag. It contains variables and functions as illustrated in Figure 2.4.
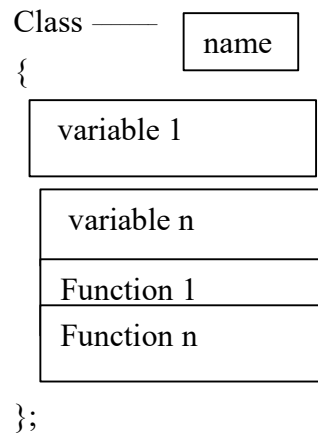
Fig. 2.4 *Structure of Class*

Note that similar to structures, a class definition also ends with a semi colon. A class definition may consist of one or more variable declarations and additional function declaration(s). The class is a keyword and template just like struct.

A class can be defined as a user-defined data type with a name tag and declaration of member variables and member functions.

A Simple Class

Let us look at an example of a class without any function. It is given below:

```
class Account {
int number;
double balance;
};
```

Every class is declared by class keyword. The class name or tag follows it. In the above example, class name is `Account`. Then there is an opening brace. Just after the opening brace, the variables are to be declared. In class Account, we have declared two variables, i.e., `number` as an integer and `balance` as a double.

Let us now add a function to the class as given below:

```
class Account {
int number;
double balance;
void display () {
cout<<balance;
};
```

We have added a function called `display` to the class Account. It returns void or nothing. It does not receive any parameter. The function header is similar to 'C' function headers. It follows the same method as the function prototype of "C" language. Here, we have given the complete function as part of the class.

Example 2.10: Class `CRectangle`:

```
// classes example
#include <iostream>
using namespace std;

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

The output of the program is as follows:

```
area: 12
```

The most important new thing in this code is the operator of scope (`::`, two colons) included in the definition of `set_values()`. It is used to define a member of a class from outside the class definition itself.

Notice that the definition of the member function `area()` has been included directly within the definition of the `CRectangle` class given its extreme simplicity, whereas `set_values()` has only its prototype declared within the class, but its definition is outside it. In this outside declaration, you must use the operator of scope (`::`) to specify that you are defining a function that is a member of the class `CRectangle` and not a regular global function.

The scope operator (`::`) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, in the function `set_values()` of the previous code, variables x and y are used, which are `private` members of class `CRectangle`, which means they are only accessible from other members of their class.

The only difference between defining a class member function completely within its class or to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second it will be a normal (not-inline) class member function, which in fact supposes no difference in behaviour.

Members x and y have private access (remember that if nothing else is said, all members of a class defined with keyword class have private access). By declaring them private access is denied to them from anywhere outside the class. This makes sense, since a member function has already been defined to set values for those members within the object: the member function `set_values()`. Therefore, the rest of the program does not need to have direct access to them. Perhaps in a simple example as this, it is difficult to see the utility in protecting those two variables, but in greater projects it may be very important that values cannot be modified in an unexpected way (unexpected from the point of view of the object).

One of the greater advantages of a class is that, as any other type, you can declare several objects of it. For example, following with the previous example of class `CRectangle`, you could have declared the object `rectb` in addition to the object `rect`:

Example 2.11: one class, two objects

```cpp
#include <iostream>
using namespace std;

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area () {return (x*y);}
};
void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}
```

```
int main () {
  CRectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

The output of the program is as follows:

```
rect area: 12
rectb area: 30
```

In this concrete case, the class (type of the objects) is CRectangle, of which there are two instances or objects: rect and rectb. Each one of them has its own member variables and member functions.

Notice that the call to rect.area() does not give the same result as the call to rectb.area(). This is because each object of class CRectangle has its own variables x and y, as they, in some way, have also their own function members set_value() and area() that each uses its object's own variables to operate.

That is the basic concept of object-oriented programming: Data and functions are both members of the object. The sets of global variables that passed from one function to another as parameters are no longer used, but instead the objects that have their own data and functions embedded as members are handled. Notice that no parameters are required in the calls to rect.area or rectb.area. Those member functions directly used the data members of their respective objects rect and rectb.

Example 2.12: How to declare an object:

```
class abc
{
private:
   int a, b, c;
public:
   void add()
{
c= a+b;
}
};
void main()
{
```

```
abc obj_abc; // obj_abc is the object of class abc
```

```
}
```

Next is an example program on how to access class members through object:

```
#include<iostream.h>
#include<conio.h>
class area
{
    private:
        float area;
    public:
        float area_square(float x)
    {
        area= x * x;
        return area;
    }
};
void main()
{
    area obj_area;
    cout<<obj_area . area_square(5.0); // calling a member
function
    // of a class with its object using dot operator
}
```

Output:

```
25.0
```

In the above example, `area` is the class which has one private data member, i.e., `float area`, and one publicly defined member function, i.e., `area_square()`. In the `main()`, we have created one object of class `area`. The created object is `obj_area`. Using the dot operator, we have accessed the publicly defined member function and finally got the output (i.e., 25.0).

Definition of an Object

Object is an instance of a class or in other words object is a replica of the class. When analysed in the context of structure, an object can be considered to be a variable of type class, similar to structure variables. A class provides a blueprint for the object. In the above example, `Account` is a class. Every person's account is an object. This means that each account holder has a number and balance. Any number of objects can be created for a class just as any number of accounts can be opened in a bank, but all accounts use the same template. This is the relationship between object and a class. The relationship is illustrated pictorially in Figure 2.5.
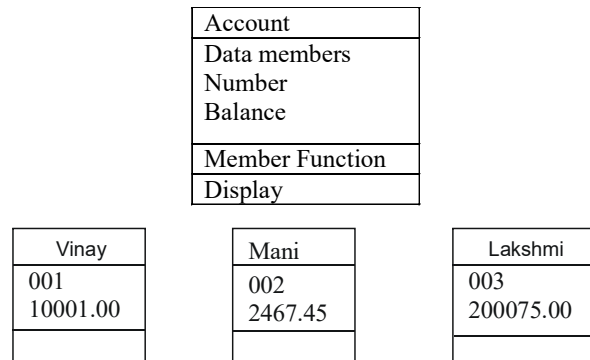
Fig. 2.5 *Objects or Instances of Class Account*

While class holds generic data structure, objects hold specific and unique data as illustrated in Figure 2.5

Objects are exact replica of the class. They each have a name as illustrated. They have their own data. The data members of the class provide for declarations for the type of variables in a class which in turn the objects will use. The functions may contain code. In C++, the variables and the functions declared within the class are called members of the class. The objects, which are instances of the class, will have their own copy of the blueprint for the variables. Although the objects can have their own copy of the member functions, it may not be necessary. They can share one copy of the member functions.

An object is nothing but a variable of type class. It is a self-contained computing entity with its own data and functions. This means that an object will have its own copy of the variables. However, the functions being common to all the objects, do not need to be kept in each object, one copy may suffice.

A class can give rise to a number of objects, but is not an object on its own. The objects thus created have a common structure but with different characteristics. Thus, the two objects of a class with different names will have same variable names but with different values, i.e., they have the same data type but different data. In some special cases, the two objects can also have same data. A class is a framework for proper encapsulation of objects. The data members of the objects can be accessed only through the interface available in public.

2.6.1 Concepts of a Class

Class is a definition of an object. All class members are private. A class is a type and an object of the class is a variable. In C++, the data and functions (procedures to manipulate the data) are worked together as a self-contained unit called an object. A class is an extended concept similar to that of structure in C programming language; this class describes the data properties alone. In C++ programming language, class describes both the properties (data) and behaviors (functions) of objects. Classes are not objects but they are used to instantiate objects. Classes contain data known as members and member functions. As a unit, the collection of members and member functions is an object. Therefore, this unit of objects makes up a class. In C programming language, a structure is specified with a name. The C++ programming language extends this concept. A class is specified with a name after the `class` keyword. Access specifiers are used to identify access rights

for the data and member functions of the class. The three main types of access specifiers in C++ programming language are `private, public` and `protected`. Generally, in class, all members (data) will be declared as `private` and the member functions would be declared as `public`. Private is the default access level. If no access specifiers are identified for members of a class, the members are defaulted to private access. Once the class is created, one or more objects can be created from the class as objects are instance of the class. The concept of specifying a class and defining member function are discussed in subsequent sections:

Specifying a Class

As you must be aware, a *class* is a user defined data type that binds data and the functions that operate on the data together in a single unit. Like other user defined data types, it also needs to be defined before using its objects in the program. A class definition specifies a new data type that can be treated as a built-in data type.

The syntax for defining a C++ class is as follows:

```
class class_name
{
        private:
            variables;
        functions;
        public:
        variables;
            functions;
    protected:
            variables;
            functions;
};
```

where,

```
class, private, public, protected =  C++ keywords
class_name=  The name of the class
variables=  Variables (data) of the class
functions=  Functions of the class
```

Note: The semicolon used immediately after the closing curly brace in the class definition is mandatory.

The variables and functions declared within the curly braces are collectively known as *members* of the class. The variables declared in the class are known as *data members* while the functions declared in the class are known as *member functions*.

Note: Members of a class cannot be declared with the `auto, extern` and `register` keywords. In addition, the data members cannot be initialized at the time of their declaration in the class.

The keywords `private, public` and `protected` are known as *access specifiers* (also known as *visibility mode*). Each member of a class is associated with an access specifier. The access specifier of a member controls its

accessibility as well as determines the part of the program that can directly access the member of the class. When a member is declared private, it can be accessed only inside the class while a public member is accessible both inside and outside the class. Protected members are accessible both inside the class in which they are declared as well as inside the derived classes of the same class. Once an access specifier has been used, it remains in effect until another access specifier is encountered or the end of the class declaration is reached. An access specifier is provided by writing the appropriate keyword (private, public or protected) followed by a colon ' : '. Note that the default access specifier of the members of a class is `private`, that is, if no access specifier is provided in the class definition, the access specifier is considered to be `private`.

Note: The `private`, `public` and `protected` access specifiers can appear in any order in the class definition. In addition, a particular access specifier can appear more than once in the class definition.

Example 2.13

A simple class definition.

```
class book
{                 //private by default
 char title[30];    //variables declaration
 float price;
 public:
 void getdata(char [],float); //function declaration
  void putdata();
};
```

In this example, a class named book with two data members title and price and two member functions getdata() and putdata() is created. As no access specifier is provided for data members, they are private by default whereas the member functions are declared as public. It implies that the data members are accessible only through the member functions while the member functions can be accessed anywhere in the program.

Generally, data members are declared as private and member functions are declared as `public`. Declaring the data members as private hides them from the rest of the program. This safeguards the data members and prevents any accidental changes to them by other parts of the program, thereby, implementing the concept of data hiding of object oriented programming. Similarly, specifying the member functions as public provides an interface that is visible and accessible to the other parts of the program.

Note that the member functions can be declared as private, however, it is useful if the member functions are to be accessed only within other member functions of the same class and not outside. In addition, all the members (data as well as functions) of a class can be declared as private. However, such a class prevents its access from the outside world and does not serve any purpose.

Note: The only difference between a C++ structure and a class is that the data and functions in a structure are by default `public` whereas the data and functions in a class are by default `private`.

Defining Member Functions

Member functions of a class can be defined either outside the class definition or inside the class definition. In both the cases, the function body remains the same; however, the function header is different.

Outside the Class

Defining a member function outside a class requires the function declaration (function prototype) to be provided inside the class definition. The member function is declared inside the class like a normal function. This declaration informs the compiler that the function is a member of the class and that it has been defined outside the class. After a member function is declared inside the class, it must be defined (outside the class) in the program.

The definition of member function outside the class differs from the normal function definition, as the function name in the function header is preceded by the class name and the scope resolution operator (: :). The scope resolution operator informs the compiler what class the member belongs to. The syntax for defining a member function outside the class is:

```
return_type class_name :: function_name(parameter_list)
{
  //body of the member function
}
```

Example 2.14

Definition of member function outside the class.

```
    class book
  {
     // body of the class as in Example 2.13
  };
                  void book :: getdata(char a[],float
b)
  {  //defining member function outside the class
   strcpy(title,a);
   price = b;
      }
  void book :: putdata()
  {
  cout<<"\nTitle of Book: "<<title;
  cout<<"\nPrice of Book: "<<price;
  }
```

Note that the member functions of the class can directly access all the data members and other member functions of the same class (`private`, `public` or `protected`) by using their names. In addition, different classes can use the same function name.

Inside the Class

A member function of a class can also be defined inside the class. However, when a member function is defined inside the class, the class name and the scope resolution operator are not specified in the function header. Moreover, the member functions defined inside a class definition are by default inline functions.

Note: Member functions that are small in size and frequently used are best suited to be defined inside the class.

Example 2.15

Definition of a member function inside a class.

```
        class book
         {
           char title[30];
           float price;
        public:
         void getdata(char [],float); // declaration
         void putdata()  //definition inside the class
           {
           cout<<"\nTitle of Book: "<<title;
           cout<<"\nPrice of Book: "<<price;
           }
        };
```

In this example, the member function `putdata()` is defined inside the class `book`. Hence, `putdata()` is by default an inline function.

Note that the functions defined outside the class can be explicitly made inline by prefixing the keyword `inline` before the return type of the function in the function header. For example, consider the definition of the function `getdata()`.

```
  inline void book :: getdata(char a[],float b)
  {
      //body of the function
  }
```

2.6.2 Classes versus Objects

In C++, a **class** is a user-defined type or data structure declared with keyword **class** that has data and functions, also called member variables and member functions, as its members whose access is governed by the three access specifiers **private**, **protected** or **public**. By default access to members of a C++ class is **private**. The **private** members are not accessible outside the class; they can be accessed only through methods of the class. The **public** members form an interface to the class and are accessible outside the class.

Instances of a class data type are known as **objects** and can contain member variables, constants, member functions, and overloaded operators defined by the programmer.

The **'Class'** and **'Object'** are the basic building blocks in Object Oriented Programming (OOP) languages. A **class** is written by a programmer in a defined structure to create an **object** in an OOP language. It defines a set of properties and methods that are common to all objects of one type.

Characteristically,

Classes: The **class** is the definitions for the data format and available procedures for a given type or class of object; may also contain data and procedures, known as class methods, themselves, i.e., classes contain the data members and member functions.

Objects: The **objects** are defined as the instances of classes.

Objects sometimes correspond to things found in the real world. For example, a graphics program may have objects, such as 'Circle', 'Square', 'Triangle', etc.

Each object is said to be an instance of a particular class, for example an object with its name field set to 'Vikas' might be an instance of class Employee. Procedures in Object Oriented Programming (OOP) are known as methods; variables are also known as fields, members, attributes, or properties. This leads to the following terms:

Class Variables: It belongs to the class as a whole; there is only one copy of each one.

Instance Variables or Attributes: The data that belongs to individual objects; every object has its own copy of each one.

Member Variables: It refers to both the class and instance variables that are defined by a particular class.

Class Methods: It belongs to the class as a whole and have access to only class variables and inputs from the procedure call.

Instance Methods: It belongs to individual objects, and have access to instance variables for the specific object they are called on, inputs, and class variables.

Fundamentally, an object can be a variable, a data structure, a function, or a method, and as such, is a value in memory referenced by an identifier. In the OOP paradigm, object can be a combination of variables, functions, and data structures; in particular in class-based variations of the paradigm it refers to a particular instance of a class.

Check Your Progress

7. What do you mean by the term inline function?
8. State benefit of inline functions.
9. What is the use of default arguments?
10. Define the term class.
11. What are access specifiers?

2.7 ANSWERS TO 'CHECK YOUR PROGRESS'

1. A function is a program with a group of statements performing specific operations. The behavior of objects is implemented through functions. The dynamic properties of objects are facilitated only through functions. Thus, functions provide interfaces to communicate with the objects.

2. The objects of a class can be passed as arguments to member functions as well as non-member functions either by value or by reference. When an object is passed by value, a copy of the actual object is created inside the function.

3. An Lvalue is an expression that can appear on the left-hand side of an equalsto sign.

4. When a function definition includes a call to itself, it is referred to as a recursive function and the process is known as recursion or circular definition.

5. Function overloading is a way to implement compile-time polymorphism that allows multiple functions to share the same name with different parameters.

6. The `main()` is the only function that cannot be overloaded.

7. An inline function is a function whose code is copied in place of each function call. In other words, each call to inline function is replaced by its code.

8. Inline functions are ideal for functions that are small in size and frequently used by the programs. This is because inline functions reduce the time consumption and overhead involved in function calls.

9. Default arguments are used in situations where the value of an argument is same in most of the function calls. Default arguments provide a lot of flexibility during function calls. If a function call does not specify an argument, the default value is passed as an argument to the function. In case a function call specifies an argument, the default value is overridden and the specified value is passed to the function.

10. A class definition is similar to a structure. A structure can also be built with data elements and functions. However, rarely are structures built with functions. A class will have declaration of data elements as well as functions. A class has a name or a tag.

11. The keywords private, public and protected are known as access specifiers (also known as visibility mode).

2.8 SUMMARY

- A function is a program with a group of statements performing specific operations. The behavior of objects is implemented through functions. The dynamic properties of objects are facilitated only through functions. Thus, functions provide interfaces to communicate with the objects.

- The objects of a class can be passed as arguments to member functions as well as non-member functions either by value or by reference. When an

object is passed by value, a copy of the actual object is created inside the function.

- If the return type of a function is void, the called function terminates and returns the control when it encounters the closing curly brace (}) or a return statement with no arguments.

- An Lvalue is an expression that can appear on the left-hand side of an equalsto sign.

- When a function definition includes a call to itself, it is referred to as a recursive function and the process is known as recursion or circular definition.

- When a recursive function is called for the first time, a space is set aside in the memory to execute this call and the function body is executed. Then a second call to a function is made; again a space is set for this call and so on.

- Function overloading is a way to implement compile-time polymorphism that allows multiple functions to share the same name with different parameters.

- An inline function is a function whose code is copied in place of each function call. In other words, each call to inline function is replaced by its code.

- Inline functions can be declared by prefixing the keyword inline to the return type in the function prototype.

- An inline function 'Requests' the compiler to replace its each and every call by the code in its body. That is, specifying a function as inline is just a request to the compiler and not a command. So, it does not change the behaviour of a function.

- Inline functions are ideal for functions that are small in size and frequently used by the programs. This is because inline functions reduce the time consumption and overhead involved in function calls.

- Whenever a function is called, the calling function must provide all the arguments specified in the function's declaration. If the calling function does not provide the required arguments, the compiler raises an error.

- Default arguments are used in situations where the value of an argument is same in most of the function calls. Default arguments provide a lot of flexibility during function calls.

- A class definition is similar to a structure. A structure can also be built with data elements and functions. However, rarely are structures built with functions. A class will have declaration of data elements as well as functions. A class has a name or a tag.

- Object is an instance of a class or in other words object is a replica of the class. When analysed in the context of structure, an object can be considered to be a variable of type class, similar to structure variables. A class provides a blueprint for the object.

- Class is a definition of an object. All class members are private. A class is a type and an object of the class is a variable. In C++, the data and functions (procedures to manipulate the data) are worked together as a self-contained unit called an object.

- The keywords private, public and protected are known as access specifiers (also known as visibility mode).

2.9 KEY TERMS

- **Recursion function:** When a function definition includes a call to itself, it is referred to as a recursive function.
- **Function overloading:** It is a way to implement compile-time polymorphism that allows multiple functions to share the same name with different parameters.
- **Default arguments:** Default arguments are used in situations where the value of an argument is same most of the function calls.
- **Class:** It refers to a user defined data type that binds data and the functions to operate on the data together in a single unit.
- **Objects:** It refers to the physical entities through which data and functions can be used in a program.

2.10 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. What is the difference between user-defined and library functions?
2. What are term non-member functions?
3. What is recursive function?
4. State the function of overloading.
5. Define the term inline function.
6. What do you understand by default arguments?
7. What are classes and objects in C++?
8. How is a member function declared?
9. How is a class specified?

Long-Answer Questions

1. Write a program to demonstrate passing objects by value to a member function of the same class.
2. Differentiate between return by values and return by reference.
3. Write a program to demonstrate the concept of recursive function.
4. Illustrate the program to demonstrate the use of default arguments as an alternative to function overloading.
5. Differentiate between normal and inline function call with the help of diagram.
6. Write a program to demonstrate the concept of structure variables as arguments.

7. Explain the concept of specifying a class and defining member functions with the help of C++ statements.

8. Discuss the concept of member functions outside and inside the class with the help of examples.

2.11 FURTHER READING

Jeyapoovan, T. 2006. *Computer Programming: Theory and Practice* (with CD). New Delhi: Vikas Publishing House.

Khurana, Rohit. 2008. *Object Oriented Programming with C++*. New Delhi: Vikas Publishing House.

Saxena, Sanjay. 2009. *Introduction to Information Technology.* New Delhi: Vikas Publishing House.

Rumbaugh, James, Fedrick Blaha, William Premerlani, and Federick Eddy.1990. *Object- Oriented Modelling and Design.* New Jersey: Prentice Hall.

Balaguruswamy, E. 1998. *Object-Oriented Programming*. New Delhi: Tata McGraw-Hill.

UNIT 3 CONSTRUCTOR AND DESTRUCTOR, OPERATOR OVERLOADING AND TYPE CASTING

Structure

3.0 INTRODUCTION

A constructor is a special member function that constructs storage area for the data members of an object by allocating and initialising memory for them. Hence, it makes the object functional by converting an object with the unused (uninitialised) memory into a usable (initialised) object.

Once an object is declared, memory space is allocated to the data members. In addition, during the execution of the program, the object may use other resources like files and so on. These resources and the memory allocated to objects must be released when an object is destroyed. This is accomplished by another special member function called destructor that is automatically invoked to release all the resources and memory that an object acquires during its lifetime.

One of the key features of C++ is that the objects of a class can be treated as variables of built-in data types. That is, C++ permits to perform all the arithmetic and logical operations on the objects of a class in the same way as these are performed on simple variables. To perform operations on simple variables, some built-in operators, such as '+', '–', '*', '/', '<', '>', '==', etc. are provided. To use these operators with the objects of a class, C++ provides a way by which an additional meaning can be given to these operators, which is known as operator overloading. It is one of the most important concepts of object-oriented programming and is a method of implementing compile-time polymorphism. Sometimes we need to carry out conversion from one data type to another. This can be achieved explicitly through what is known as type casting.

In this unit, you will study about the constructor and destructors, constructors of the string class, string class assignment, string access operators and method, operator overloading, type casting.

3.1 OBJECTIVES

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

- Explain the meaning of constructor and destructors
- Explain constructors of the string class and the string class assignment
- Describe the string access operators and operator overloading
- State the meaning of type casting

3.2 CONSTRUCTOR AND DESTRUCTORS

A constructor is a special member function that constructs storage area for the data members of an object by allocating and initialising memory for them. Hence, it makes the object functional by converting an object with the unused (uninitialised) memory into a usable (initialised) object. It is special as it has the same name as that of the class and is automatically invoked whenever an object of the class is created.

Unlike other member functions, a constructor does not have any return type (not even `void`). This is because the constructor is invoked automatically by the system and hence, no program has been defined for it to return anything to. The name and the absence of a return type help the compiler to distinguish a constructor from the other member functions of the class. Note that a constructor must be declared as public otherwise the objects of the class cannot be instantiated.

Like other member functions of the class, a constructor can also be defined either inside or outside the class definition. The syntax to define a constructor (inside the class) is:

```
class class_name
{
     .
.
   public:
class_name(parameter_list) //header of the constructor
{
//body of the constructor
}
};
```

where,

`parameter_list` is optional.

Note: Inside the class, a constructor is treated as an inline function.

The syntax to define the constructor outside the class is:

```
class class_name
{
.
.
.
  public:
class_name(parameter_list); //constructor prototype
.
.
};
class_name::class_name(parameter_list)  //constructor
definition
{
//body of the constructor
}
```

where,

`paramter_list` is optional.

Example 3.1: Implementing CRectangle including a constructor:

```
// example: class constructor
#include <iostream>
using namespace std;
class CRectangle {
    int width, height;
  public:
    CRectangle (int,int);
    int area () {return (width*height);}
};
CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}
int main () {
  CRectangle rect (3,4);
  CRectangle rectb (5,6);
  cout << 'rect area: ' << rect.area() << endl;
  cout << 'rectb area: ' << rectb.area() << endl;
  return 0;
}
```

The output of the program is :

```
rect area: 12
rectb area: 30
```

Example 3.2: on constructors and destructors

```cpp
#include <iostream>
using namespace std;
class CRectangle {
    int *width, *height;
  public:
    CRectangle (int,int);
    ~CRectangle ();
    int area () {return (*width * *height);}
};
CRectangle::CRectangle (int a, int b) {
  width = new int;
  height = new int;
  *width = a;
  *height = b;
}
CRectangle::~CRectangle () {
  delete width;
  delete height;
}
int main () {
  CRectangle rect (3,4), rectb (5,6);
  cout << 'rect area: ' << rect.area() << endl;
  cout << 'rectb area: ' << rectb.area() << endl;
  return 0;
}
```

When executed we get the following output of the program:

```
rect area: 12
rectb area: 30
```

Example 3.3: A program to demonstrate the order in which constructors and destructors are called in inheritance

```cpp
#include<iostream>
#include<cstring>
using namespace std;
class wood
{
protected:
    char type[10];
public:
    wood()
    {
        strcpy(type, "Teak");
        cout<<"\nBase class constructor wood called ";
    }
```

```
    ~wood()
    {cout<<"\nBase class destructor called";}
};
class table:public wood
{
    char dimension[5];
public:
    table()
    {
       strcpy(dimension, "2X4");
       cout<<"\nDerived class constructor table called";
    }
    ~table()
    {cout<<"\nDerived class destructor called";}
};
int main()
{
    table t1;
    return 0;
}
```

The output of the program is

```
Base class constructor wood called
Derived class constructor table called
Derived class destructor called
Base class destructor called
```

In this example, the derived class table inherits the base class wood. When the object t1 of the class table is declared in main(), constructor of wood is called first and then the constructor of table is called. However, when t1 is destroyed (when main() terminates), the destructor of table is called first and then the destructor for wood.

Note: The public and the protected members of the base class can be directly initialized using assignment statements in the body of the derived class constructor.

In case of multiple inheritance, the base class constructors are called in the order in which the base classes are specified in the derived class definition, that is, from left to right. However, when an object of the derived class is destroyed, the derived class destructor is called before any of the base class destructors is called. The base class destructors are called in the reverse order of calling the base class constructors, that is, from right to left.

Calling Constructors

Once a constructor is defined, it can be called implicitly as well as explicitly. If the name of the constructor is not used in the object declaration, the call is known as an implicit call to the constructor. On the other hand, if the name of the constructor

is used in the object declaration, the call is known as an explicit call to the constructor.

Example 3.4: A code segment to demonstrate the calling of constructor implicitly and explicitly

```
class display
{
int x, y;
   public:
    display()                          //constructor
    {
        x=1;
        y=2;
        cout<<"Value of x and y is : "<<x<<", "<<y<<endl;

}
};
int main()
{
    display disp;                      //implicit call
    display disp1 = display();   //explicit call
}
```

                                      name of the
                                      constructor

In Example 3.4, two objects, namely, `disp` and `disp1` of the class `display` are declared and initialised. The object `disp` is initialised by implicitly calling the constructor, whereas `disp1` is initialised by explicitly calling the constructor, that is, by using the constructor name, `display()`.

Note: A constructor and the other member functions of the same class can call each other.

Types of Constructor

Constructors are classified into three types, namely, default constructor, parameterised constructor and copy constructor. In this section, only default constructor and parameterised constructor are discussed.

Default Constructor

A default constructor is a constructor that has an empty parameter list and is used to initialise all the objects of a class with the same values. There can be only one default constructor in a class. If it is not defined explicitly, the compiler automatically provides a default constructor to construct the objects of a class. However, the default constructor provided by the compiler initialises all the data members with garbage values.

Note: A default constructor for a class X has a form of `X::X()`.

To understand the concept of default constructor, consider Example 3.2. In this example, the class `display` defines a default constructor named

display() to initialise the data members x and y of the class with values 1 and 2, respectively. Both the objects disp and disp1 are initialised with same values using the default constructor display(). Note that to invoke the default constructor, no parentheses (())are required in the implicit call to the constructor at the time of declaring an object. For example, consider the statements:

```
display disp;        //valid
display disp();      //invalid
```

A default constructor can also be defined without a body. In such situations, it can be defined as shown in the statement:

```
display(){}
```

Such default constructors are 'Do-Nothing' functions as they do not perform any task, however, they are invoked at an appropriate time by the compiler. In addition, they initialise data members with garbage values.

Parameterised Constructor

When different objects need to be initialised with different values, a parameterised constructor can be defined. A parameterised constructor is a constructor that accepts one or more parameters at the time of declaration of objects and initialises the data members of the objects with these parameters.

Example 3.5: A program to demonstrate the concept of parameterised constructor

```
#include<iostream>
using namespace std;
class library
{
int roll;
char name[30];
int b_code;
   public:
library(int r, char n[], int code)
//parameterised constructor
{
roll=r;
strcpy(name, n);
b_code=code;
}
void show()
{
cout<<"Roll no.: "<<roll<<endl;
cout<<"Student Name: "<<name<<endl;
cout<<"Code of Book Issued: "<<b_code<<endl<<endl;
}
};
int main()
{
```

```
library lib1(1, "Kanika", 101);  //implicit call to
//parameterised constructor
lib1.show();
library lib2=library(2, "Mansi", 102); //explicit call to
//parameterised constructor
lib2.show();
return 0;
}
```

The output of the program

```
Roll no.: 1
Student Name: Kanika
Code of Book Issued: 101

Roll no.: 2
Student Name: Mansi
Code of Book Issued: 102
```

In Example 3.5, the class `library` has a parameterised constructor that accepts three parameters, `r, n[]` and `b`. The objects `lib1` and `lib2` are declared and initialised with different values. The object `lib1` calls the parameterised constructor implicitly whereas the object `lib2` calls the parameterised constructor explicitly.

Note that an explicit call to this constructor can also be made by using the statement `library(2,"Mansi",102).show()` which allows to create a temporary instance of a class. A temporary instance or a temporary object remains in the memory as long as the statement is being executed. Once the execution of the statement terminates, the temporary instance is destroyed.

Note: The temporary instances do not have any name and hence, cannot be referred to further in the program. They are automatically deleted when no longer required.

A special case of parameterised constructors is one-parameter constructor that provides another way to initialise the objects of a class. In one-parameter constructor, the objects of a class are initialised using the assignment operator '='. Such constructor automatically converts the parameter of any type into class type to which it is assigned.

Initialiser List

So far, the data members of the class are initialised inside the body of a constructor using assignment statements in which the data members are first created and then the assignment operation takes place. There is an efficient way to initialise data members of a class using an initialiser list (also known as member-initialisation list). An initialiser list is provided at the end of the header and before the body of constructor. Using initialiser list, data members are initialised when they are created, that is, even before the execution of constructor. Note that an initialiser list can be defined for both default constructor and parameterised constructor. The syntax to define a default constructor with an initialiser list is:

```
class class_name
{
    .
    .
  public:
class_name():datamember1(val1),datamember2(val2),……,
datamemberN(valN)
    {
        //body of the constructor
    }
    .
    .
};
```

For example, the constructor in Example 3.5 can be alternatively defined using an initialiser list as shown in the code segment:

```
display(): x(1),y(2)    //initialiser list
{
cout<<"Value of x and y is : "<<x<<", "<<y<<endl;
}
```

In this code segment, when the constructor is called for the object disp, the initialiser list initialises x and y with the values 1 and 2, respectively. The syntax to define a parameterised constructor with an initialiser list is:

```
class class_name
{
    .
    .
  public:
class_name(type1 param1, type2 param2,…typeN paramN)
:datamember1(param1),datamember2(param2),
      ……,datamemberN(paramN)
    {
        //body of the constructor
    }
    .
    .
};
```

For example, the constructor in Example 3.5 can be alternatively defined using an initialiser list, as shown in the code segment:

```
count(int x):counter(x)
{}
```

In this code segment, when the constructor is called for the object c, the initialiser list initialises counter with the value of x.

Constructor Overloading

A class with a single constructor has been discussed so far. However, C++ allows defining multiple constructors with different number, data type or order of parameters in a single class and it is known as constructor overloading. Constructor overloading enables multiple constructors to initialise different objects of a class differently. These objects can be initialised with the same values or different values or with the existing objects of the same class. In C++, a class can simultaneously have a default constructor, a parameterised constructor and a copy constructor.

Note: When an object is initialised with an overloaded constructor, the compiler determines which constructor is to be called based on the number, data type and order of the parameters.

Example 3.6: overloading class constructors

```
#include <iostream>
using namespace std;
class CRectangle {
    int width, height;
  public:
    CRectangle ();
    CRectangle (int,int);
    int area (void) {return (width*height);}
};
CRectangle::CRectangle () {
  width = 5;
  height = 5;
}
CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}
int main () {
  CRectangle rect (3,4);
  CRectangle rectb;
  cout << 'rect area: ' << rect.area() << endl;
  cout << 'rectb area: ' << rectb.area() << endl;
  return 0;
}
```

The output of the program is as follows:

```
rect area: 12
rectb area: 25
```

In this case, rectb was declared without any arguments, so it has been initialized with the constructor that has no parameters, which initializes both width and height with a value of 5.

It is very important that if a new object is declared and its default constructor has to be used (the one without parameters), parentheses () are not included:

```
CRectangle rectb;   // right

CRectangle rectb(); // wrong!
```

Example 3.7: A program to demonstrate the concept of overloaded constructors

```
#include<iostream>
  using namespace std;
  class weight
  {
  int kg;
  int gm;
     public:
  weight(){}                        //default constructor
  weight(int kilogram, int gram) //parameterised constructor
  {
  kg=kilogram;
  gm=gram;
  }
  void show()
  {
  cout<<kg<<" Kgs and "<<gm<<" gms\n";
  }
  weight sum_weight(weight w2)
  {
  weight w;
     w.gm = gm + w2.gm;
     w.kg=w.gm/1000;
     w.gm=w.gm%1000;
     w.kg+=kg+w2.kg;
     return(w);
  }
  };
  int main()
  {
  //call to parameterised constructor
  weight w1(10,100);
  weight w2(20,200);
  weight w3;            //call to default constructor
  w3=w1.sum_weight(w2);
  cout<< "Weight1 = ";
  w1.show();
  cout<< "Weight2 = ";
  w2.show();
  cout<< "Weight3 = ";
```

```
w3.show();

return 0;

}
```

The output of the program

```
Weight1 = 10 Kgs and 100 gms

Weight2 = 20 Kgs and 200 gms

Weight3 = 30 Kgs and 300 gms
```

In Example 3.7, default and parameterised constructor are defined. The default constructor is invoked by the statement `weight  w3` and parameterised constructor is invoked by the statements `weight w1(10,100)` and `weight w2(20,200)`. Note that the default constructor `weight(){}` does nothing, however, if it is not specified in the class, the compiler generates an error on the statement `weight w3`. This is because in case of multiple constructors in a class, the compiler does not provide the default constructor implicitly. Hence, a default constructor must be defined explicitly in the class with multiple constructors if any of its objects is to be created without arguments.

Constructors with Default Arguments

Like other member functions, parameterised constructors can also have default arguments. These arguments are used when no corresponding value is passed at the time of declaration of the object. When a parameterised constructor is defined inside the class, the values for the default arguments are provided in the definition of constructor. However, when parameterised constructor is defined outside the class, the values for the default arguments are provided only in its prototype and not in its definition. For example, the parameterised constructor `weight()` (as defined in Example 3.5) with default arguments can be written as shown:

```
weight(int kilogram, int gram=100);
```

In this statement, the default value of `gram` is `100` and `weight()` can be invoked by the statement `weight w1(l0)`. However, if the value of `gram` is specified in the statement, it overrides the default value of `gram`. It should be noted that either all the arguments or only the trailing arguments can be omitted in the parameter list of the constructor.

The parameterised constructor that has all default arguments can be considered as a default constructor since it can be invoked without arguments. This implies that if an object is declared without parentheses, all the data members are initialised with the default values. For example, a default argument constructor of the form `weight::weight(int = 0)` can be called either with one or no argument. If no argument is specified, it can be considered as a default constructor of the form `weight::weight()`. A problem arises if both the forms of constructor are used in the same program. The statement `weight w` creates ambiguity whether to call the constructor `weight::weight(int = 0)` or `weight::weight()`. Thus, in case of constructors with all default arguments, the use of default constructor must be avoided.

Dynamic Initialisation of Object

Parameterised constructors can also be used to dynamically initialise the objects of a class. That is, the parameterised constructor can initialise the object with the values provided by the user at run-time. Moreover, by using multiple constructors in a class, different formats of initialisation can be provided.

Example 3.8: A program to demonstrate dynamic initialisation of objects

```cpp
#include<iostream>
using namespace std;
class item
{
float price, discount, tax, total;
   public:
item(float p, float t, float d=0.25) //first constructor
{
price=p;
discount=p*d;
tax=t;
}
item(float p, float t, int d)       //second constructor
{
price=p;
discount=p*(float)d/100;
tax=t;
}
void show()
{
total=price-discount+tax;
cout<<"Total price: "<<total<<endl;
}
};
int main()
{
float p, t, d1;
int d2;
cout<<"Enter the price and tax of Item 1 (in decimal):";
cin>>p>>t;
item i1(p,t);         //dynamic initialisation, call to
                            //first constructor
i1.show();
cout<<"\nEnter the price, tax and discount of Item 2"
<<" (in decimal):";
cin>>p>>t>>d1;
```

```
item i2(p,t,d1);     //dynamic initialisation, call to
                                //first constructor
i2.show();
cout<<"\nEnter the price, tax and discount (in integer)"
     <<" of Item 3:";
cin>>p>>t>>d2;
item i3(p,t,d2);    //dynamic initialisation, call to
                            //second constructor
i3.show();
return 0;
}
```

The output of the program

```
Enter the price and tax of Item 1 (in decimal): 10000
0.12
Total price: 7500.12


Enter the price, tax and discount of Item 2 (in decimal):
10000 0.12 0.30
Total price: 7000.12


Enter the price, tax and discount (in integer) of Item 3:
10000 0.12 30
Total price: 7000.12
```

In Example 3.8, the values of p, t, d1 and d2 are provided by the user at run-time and thus, the data members of the class are initialised dynamically. Since the constructor item() is overloaded, it provides different formats of initialisation, that is, item i1(p, t), item i2(p, t, d1) and item i3(p,t, d2). Both the statements item i1(p, t) and item i2(p, t, d1) invoke first constructor and the statement item i3(p, t, d2) invokes second constructor.

Destructors

Once an object is declared, memory space is allocated to the data members. In addition, during the execution of the program, the object may use other resources like files and so on. These resources and the memory allocated to objects must be released when an object is destroyed. This is accomplished by another special member function called destructor that is automatically invoked to release all the resources and memory that an object acquires during its lifetime. A destructor releases the resources and memory at run-time to clean up the unused storage area.

Like a constructor, a destructor is also special as it has the same name as that of the class of which it is a member, but with a tilde (~) prefixed to its name. A tilde (~) is a C++ complement operator, which reminds that a destructor is a complement of the constructor (creation). A destructor neither accepts any parameter nor has a return type (not even void). A destructor cannot be

overloaded, that is, a class can have only one destructor. Hence, a single destructor is used to destroy all the objects of a class created either using default, parameterised or copy constructor.

Note: Like a constructor, the destructor can be called within any other member functions. Similarly, other member functions can be called within a destructor.

The syntax to define a destructor of a class is:

```
class class_name
{
   .
   .
  public:
~class_name()           //header of the destructor
{
//body of the destructor
}
.
.
};
```

Note: Destructor can also be defined either inside or outside the class. However, it should be defined in the public section of the class to avoid the compile-time error.

A destructor is invoked implicitly by the compiler when the object goes out of scope. However, if object is created dynamically, it is difficult for the compiler to know whether the pointer points to something possessed by the object and that also has to be deleted along with the object or something independent of the object. Hence, the destructor must be explicitly provided.

To understand the concept of destructor, consider Example 3.9. In this example, the destructor for the class library can be specified as shown:

```
class library
{
.
public:
.
~library()
{
    cout<<"\nObject destroyed...";
}
.
.
};
```

The output of the program

```
Roll no.: 1
Student Name: Kanika
```

```
Code of Book Issued: 101

Roll no.: 2
Student Name: Mansi
Code of Book Issued: 102

Object destroyed...
Object destroyed...
```

In Example 3.9, since two objects namely `lib1` and `lib2` were created, the destructor is called twice.

Order of Calling Constructors and Destructors

The constructor and the destructor of a class are automatically invoked when memory is allocated and de-allocated to an object, respectively. Moreover, when multiple objects of a class are created, the constructor for each object is called in the order in which the objects are declared. However, the destructor for each object is called in the reverse order of the constructors, that is, in the reverse order of the object creation.

Note that for the objects with local scope, the constructor is called when their declaration is encountered in the respective block or function, while the destructor is called when the block or function terminates. For objects with global scope, the constructor is called only once when their declaration is encountered (generally before `main()`) in the program and the destructor is called when the program terminates.

Example 3.10: A program to demonstrate the order of calling constructor and destructor

```
#include<iostream>
using namespace std;
int counter;  //to count the number of objects
class memory
{
   public:
memory()              //constructor
{
      counter++;
cout<<"\tAllocating memory to object "<<counter<<endl;
    }
~memory()             //destructor
{
cout<<"\tDe-allocating memory to object "<<counter<<endl;
counter—;
    }
};
```

```
int main()
{
cout<<"Memory allocation in Main()"<<endl;
memory m1, m2;
{
cout<<"\nMemory allocation in Block 1"<<endl;
memory m3;
{
        cout<<"\tMemory allocation and de-allocation in"
 <<" Block 2"<<endl;
memory m4;
        }
cout<<"\nMemory de-allocation in Block 1"<<endl;
}
cout<<"\nMemory de-allocation in Main()";
return 0;
}
```

The output of the program

```
Memory allocation in Main()
        Allocating memory to object 1
        Allocating memory to object 2

Memory allocation in Block 1
        Allocating memory to object 3
        Memory allocation and de-allocation in Block 2
        Allocating memory to object 4
        De-allocating memory to object 4

Memory de-allocation in Block 1
        De-allocating memory to object 3

Memory de-allocation in Main()
          De-allocating memory to object 2
          De-allocating memory to object 1
```

In Example 3.10, the objects m1 and m2 are declared within the main() and the objects m3 and m4 are declared within block 1 and block 2, respectively. The constructor is first called for m1, then for m2 and so on. When the inner block terminates, the destructor for each of the objects is called in the reverse order of the constructors. That is, the destructor is first called for m4, then for m3, then for m2 and finally for m1.

3.3 CONSTRUCTORS OF THE STRING CLASS

The constructor of the **string** Class is used to construct a string object and initializing its value depending on the constructor version used.

Even though the constructors are basically functions that are typically created as part of a class, but they are special functions. Constructor functions are automatically called or invoked whenever a new object is created or instantiated. The constructor function construct, build, or initialize the new object.

Prototype	Example	Comments
`string();`	`string s1;`	**Default Constructor:** Builds an empty string.
`string(const char* s);`	`string s2("Hello, World!");`	**Conversion Constructor:** Converts a C-string into a string.
`string(const string& s);`	`string s3(s2);`	**Copy Constructor:** Makes a new string by copying a string.

String Class Constructors: The string class includes many constructors but above mentioned **string** constructors are used in day-to-day programming.

Instances of the **string** class manage their own memory and are able to grow automatically as characters are added to them.

Declaration: Following is the declaration for **std::string::string.**

string();

The string class has the following basic functionalities:

1. **Constructor with No Arguments:** The constructor with no arguments allocates the storage for the string object in the heap and assigns the value as a NULL character.

2. **Constructor with Only One Argument:** The constructor with only one argument accepts a pointer to a character or it can pass an array of characters, accepts the pointer to the first character in the array then the **constructor** of the **String** class allocates the storage on the heap memory of the same size as of the passed array and copies the contents of the array to that allocated memory in heap. It copies the contents using the **strcpy()** function declared in **cstring** library.

   Before performing the above mentioned operation it checks that if the argument passed is a **NULL** pointer then it performs as a **constructor** with **no arguments**.

3. **Copy Constructor:** The copy constructor is called when any object is created of the same type from an already created object then it performs a **profound copy**. It allocates new space on the heap for the object that is to be created and copies the contents of the passed object (that is passed as a reference).

4. Move Constructor: The move constructor is typically called when an object is initialized (by direct-initialization or copy-initialization) from **rvalue** of the same type. It accepts a reference to an **rvalue** of an object of the type of custom **string** class.

Following C++ program illustrates the implementation of the above defined methods using the custom string class **Mystring**.

Example 3.11

```cpp
// C++ program for illustrating the
// above defined functionality
#include <cstring>
#include <iostream>
using namespace std;
// Custom string class
class Mystring {
    // Initialise the char array
    char* str;
public:
    // No arguments Constructor
    Mystring();
    // Constructor with 1 arguments
    Mystring(char* val);
    // Copy Constructor
    Mystring(const Mystring& source);
    // Move Constructor
    Mystring(Mystring&& source);
    // Destructor
    ~Mystring() { delete str; }
};
// Function to illustrate Constructor
// with no arguments
Mystring::Mystring()
    : str{ nullptr }
{
    str = new char[1];
    str[0] = '\0';
}
// Function to illustrate Constructor
// with one arguments
Mystring::Mystring(char* val)
{
    if (val == nullptr) {
        str = new char[1];
        str[0] = '\0';
    }
```

```cpp
    else {
        str = new char[strlen(val) + 1];
        // Copy character of val[]
        // using strcpy
        strcpy(str, val);
        cout << "The string passed is: "
             << str << endl;
    }
}
// Function to illustrate
// Copy Constructor
Mystring::Mystring(const Mystring& source)
{
    str = new char[strlen(source.str) + 1];
    strcpy(str, source.str);
}
// Function to illustrate
// Move Constructor
Mystring::Mystring(Mystring&& source)
{
    str = source.str;
    source.str = nullptr;
}
// Driver Code
int main()
{
    // Constructor with no arguments
    Mystring a;
    // Convert string literal to
    // char array
    char temp[] = "Hello";
    // Constructor with one argument
    Mystring b{ temp };
    // Copy constructor
    Mystring c{ a };
    char temp1[] = "World";
    // One arg constructor called,
    // then the move constructor
    Mystring d{ Mystring{ temp } };
     return 0;
}
```

The output of the program

```
The string passed is: Hello
The string passed is: Hello
```

3.3.1 String Class Assignment

The member function `assign()` is used for the assignments, it assigns a new value to the string, replacing its current contents.

Syntax 1: Assign the value of **string str**.

`string& string::assign (const string& str)`

`str` : This is the string to be assigned.

Returns: *this

Example 3.12

```
// C++ code to assign (const string& str)
#include <iostream>
#include <string>
using namespace std;

// Function to demonstrate assign
void assignDemo(string str1, string str2)
{
    // Assigns str2 to str1
    str1.assign(str2);
    cout << "After assign() : ";
    cout << str1;


}


// Driver code
int main()
{
    string str1("Hello World!");
    string str2("Vikas Delhi");

    cout << "Original String : " << str1 << endl;
    assignDemo(str1, str2);

    return 0;
}
```

The output of the program

```
Original String : Hello World!
After assign() : Vikas Delhi
```

Syntax 2: Assigns at most **str_num** characters of **str** starting with index **str_idx**. It throws **out_of _range** if **str_idx > str. size()**.

```
string& string::assign (const string& str,
size_type str_idx, size_type str_num)
```

str : This is the **string** to be assigned.

str_idx : This is the index number in **str**.

str_num : This is the number of characters picked from **str_idx** to assign.

Return : *this

Example 3.13

```cpp
// C++ code to illustrate
// assign(const string& str, size_type
// str_idx, size_type str_num)

#include <iostream>
#include <string>
using namespace std;

// Function to demonstrate assign
void assignDemo(string str1, string str2)
{
    // Assigns 8 characters from
    // 5th index of str2 to str1
    str1.assign(str2, 5, 8);
    cout << "After assign() : ";
    cout << str1;

}

// Driver code
int main()
{
    string str1("Hello World!");
    string str2("VikasforDelhi");

    cout << "Original String : " << str1 << endl;
    assignDemo(str1, str2);

    return 0;
}
```

The output of the program

```
Original String : Hello World!

After assign() : forDelhi
```

3.3.2 String Access Operator

As **string** class is a container class, therefore, we can iterate over all its characters using an iterator similar to other containers, such as vector, set and maps, but generally, we use a simple **for** loop for iterating over the characters and index them using **[]** operator, the square brackets.

Fundamentally, the characters in a string can be accessed by referring to its index number inside square brackets **[]**.

The following C++ example prints the **first character** in **myString**:

Example 3.14

```
#include <iostream>
#include <string>
using namespace std;
int main() {
   string myString = "Hello";
   cout << myString[0];
   return 0;
}
```

The output of the program

```
H
```

String indexes in the above example start with 0: [0] which is the first character and [1] is the second character, etc.

In the following example the second character in **myString** is printed:

Example 3.15

```
#include <iostream>
#include <string>
using namespace std;
int main() {
   string myString = "Hello";
   cout << myString[1];
   return 0;
}
```

The output of the program

```
e
```

Changing Characters in the String: To change the value of a specific character in a string, refer to the index number, and use single quotes as illustrated in the following program.

Example 3.16

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string myString = "Hello";
    myString[0] = 'Y';
    cout << myString;
    return 0;
}
```

The output of the program

```
Yello
```

Check Your Progress

1. What do you mean by the term constructor?
2. Define the term destructors.
3. State the use of string class constructors.

3.4 OPERATOR OVERLOADING

Operator overloading is the process that enables an operator to exhibit different behavior, depending on the data being provided. It enables to change the functionality of the existing operators so that they can be used with user-defined data types, such as classes in addition to with built-in data types. For example, consider the statements:

```cpp
int a, b, c;
c = a + b;       //'+' is used with simple variables
```

In these statements, two simple variables are added and the result is stored in the third variable. However, the '+'operator cannot be directly used (without overloading) to add two objects of a class. For example, if `ob1`, `ob2` and `ob3` are three objects of class `A`, then the following statement results in compile-time error, if the '+' operator is not overloaded:

```cpp
ob3 = ob1 + ob2;              //invalid
```

Thus, to make this statement valid in C++, the '+' operator needs to be overloaded. Almost all the unary, binary and some special operators except few can be overloaded in C++. The operators that can be overloaded are listed in Table 3.1.

Table 3.1 *List of Operators that can be Overloaded*

Type	Operator
Unary	! (logical negation), & (address-of), * (pointer dereference), + (unary plus), − (unary minus), ~ (one's complement), ++ (increment) and −− (decrement)
Binary	arithmetic (+,−,*,/,%), logical (&&,||), relational (<,>,<=,>=,!=,==), shorthand (+=,−=,%=,*=,/=)
Special	<<,>>,(),=,[],new, delete, new[],delete[]

Some operators that cannot be overloaded are listed as follows:

- dot operator (.)

- dereference pointer to class members (.*)

- scope resolution operator (::)

- conditional operator (?:)

- sizeof operator

- preprocessor symbol (#)

An operator is overloaded with the help of a special function called an *operator function*. It defines the operations that the overloaded operator will perform on the objects of the class for which it is redefined. An operator function can be defined either as a public member function of the class or as a friend function. In other words, an operator can be overloaded either using member functions or using friend functions. To overload an operator, the following steps are to be followed:

(i) Create the class for which an operator is to be overloaded.

(ii) Declare the operator function either as a public member function of the class or as a friend function of the same class.

(iii) Define the operator function either inside or outside the class definition, (if it is a member function) and outside the class (if it is a friend function).

Some rules must always be kept in mind while overloading operators. These rules are as follows:

- The implementation of the operator can be changed, however, not the syntax for using the operator.

- The precedence and the associativity of an operator cannot be changed. However, parentheses can be used to change the order of evaluation.

- The number of operands required (unary, binary, ternary) with the operator cannot be changed.

- New operators cannot be created; however, new definitions for the existing operators can be created.

- Overloaded operators except the function call operator '()' cannot have default arguments.

- All overloaded operators except the assignment operator '=' can be inherited by the derived classes.

- There is no automatic 'composition' of operators. That is, overloading '=' and '+' operators does not imply that the '+=' operator has been overloaded automatically. It has to be overloaded explicitly.

- The operators `()`, `[]`, `->` and `=` can be overloaded only as member functions and not as friend functions.

Overloading Using Member Functions

The operator function defined as a member function of the class is known as member operator function. Like other member functions of the class, the member operator function can be defined either inside or outside the class definition.

The syntax to define the member operator function inside the class is:

```
return_type operator op(parameter_list)
{
//function body
}
```

where,

`return_type` = data type of the value returned by the function

`operator` = C++ keyword

`op` = operator being overloaded

`parameter_list` = list of arguments

`operator op()` = name of the operator function

Note: An operator function is defined as a member function of the class must be a public non-static member function.

If the member operator function is defined outside the class, it has to be first declared inside the class.

The syntax to declare the operator function inside the class is:

```
return_type operator op(parameter_list);
```

The syntax to define the member operator function outside the class is:

```
return_type class_name::operator op(parameter_list)
{
//function body
}
```

Note: It is optional to provide a space between the keyword `operator` and the operator `op` in the operator function header.

Overloading Unary Operators

When unary operators are overloaded using member functions, the member operator function does not accept any argument. That is, the `parameter_list` has no arguments for unary operators (except the postfix forms of ++ and − operators). This is because the operand for the unary operator, that is, the object that invokes the member operator function, is passed implicitly to the function.

Example 3.17: A program to demonstrate the concept of overloading logical negation operator

```
#include<iostream>
using namespace std;
class logical
{
bool a;
bool b;
   public:
void getdata(bool x,bool y);
void putdata();
void operator!();      //overloading '!' operator
};
void logical::getdata(bool x,bool y)
{
a = x;
b = y;
}
void logical::putdata()
{
cout<<"\na= "<<a<<"\tb= "<<b;
}
void logical::operator!()   //operator!()defined outside
the class
{
a = !a;        //logically negating the data members
b = !b;
}
int main()
{
logical logic;
logic.getdata(true,false);
cout<<"Original values are:";
logic.putdata();              //displaying  the  original
values
!logic;        //calling the operator!() function
cout<<"\nNew values are:";
logic.putdata();              //displaying new values
return 0;
+
```

The output of the program

```
Original values are:
a= 1     b= 0
```

```
New values are:
a= 0    b= 1
```

In Example 3.17, a class `logical` containing the data members `a` and `b` of
`bool` type, is defined. The unary operator '`!`' is overloaded to logically negate
the values of `a` and `b`. Note that the function `operator!()` does not accept
any argument as the object invoking the function is passed implicitly and the operator
works on that object only. Moreover, the function `operator!()` does not
return any value. Thus, it is invalid to write the statement:

```
logic2=!logic1  //invalid as function does not return any
value
```

However, if the function is modified to return a value of type ~~logical~~, then this
statement becomes valid.

Example 3.18:

```
    #include<iostream>
using namespace std;
class locate
{
    Int a;
    Int b;
public:
    void getdata(int x,int y);
    void show(void);
    void operator-(); //overloading the - operator
};
void locate :: getdata(int x, int y)
{
    a=x;
    b=y;
}
void locate :: show(void)
{
    cout << a << " ";
    cout << b << " ";
}
void locate :: operator-()
{
    a=-a;
    b=-b;
}
int main()
{
    locate L;
    L.getdata(-7,9);
```

```
    cout << "L : ";
    L.display();
    -L;    //the function operator-() activated
    cout << "S : ";
    L.display();
    return 0;
}
```

The output of the program is as follows:

```
L : -7 9
L : 7 -9
```

The function operator-() just changes the sign of the data member of the object L. Since this function is a member function of the same class, it can directly access the members of the object which activated it. As the function operator-() does not return any value so if you write

```
L = -L;
```

It will not work.

Overloading Increment and Decrement Operators

In C++, increment and decrement operators exist in two forms, namely, *prefix* and *postfix.* C++ enables to overload both these forms. Note that the prefix forms of ++ and – operators are overloaded exactly the same way as any other unary operator. That is, the member operator functions of prefix forms of both the operators do not accept any argument. However, the postfix forms accept an additional argument of type `int`.

The compiler uses the `int` argument to distinguish between the member operator functions of both the forms. By default, the value of this argument is zero. However, if any value is passed at the time of function call, the operator function should be called explicitly using the object name and the dot operator.

The syntax to define the member operator functions of prefix ++ and – operators, inside the class is:

```
return_type operator++()           // p r e f i x
increment
{
//function body
}
return_type operator—()            // p r e f i x
decrement
{
//function body
}
```

The syntax to define the member operator functions of postfix ++ and – operators, inside the class is:

```
return_type operator++(int a)      // p o s t f i x
increment
{
```

```
//function body
}

return_type operator—(int a)          //postfix
decrement
{
//function body
}
```

The member operator function for both the forms returns an object of the class for which it is being overloaded. The prefix form returns the incremented or decremented value of the object, however, the postfix form returns the original value of the object before incrementing or decrementing. This is because when the prefix form is used in an assignment statement, the value is incremented or decremented first and then is assigned to another variable. However, when the postfix form is used in an assignment statement, the original value is first assigned to the variable and then it is incremented or decremented.

Note: If the overloaded functions for ++ and − operators have return type `void`, then they cannot be used in an expression or in an assignment statement.

Example 3.19: A program to demonstrate the concept of overloading increment and decrement operators

```
#include<iostream>
using namespace std;
class weight
{
int kg;
int gm;
   public:
weight(int k, int g)         //parameterised constructor
{
kg = k;
gm = g;
}
void display()
{
cout<<"kilogram= "<<kg<<" and gram= "<<gm<<endl;
}
weight operator++();     //overloading prefix ++ operator
weight operator++(int a); //overloading postfix ++ operator
weight operator—();        //overloading prefix − operator
weight operator—(int a); //overloading postfix − operator
};
weight weight::operator++()
{
++kg;
++gm;
```

```
return weight(kg,gm); //returning the incremented value
as
            //unnamed temporary object
}
weight weight::operator—()
{
—kg;
—gm;
return weight(kg,gm); //returning the decremented value
}
weight weight::operator++(int a)
{
int k = kg;
int g = gm;
if (a == 0)
{
kg++;     //if a is 0, increment by 1
gm++;
}
else
{
kg += a; //if a is not 0, increment by a
gm += a;
}
return weight(k,g); //returning the value before increment
}
weight weight::operator—(int a)
{
int k = kg;
int g = gm;
if (a == 0)
{
kg—;      //if a is 0, decrement by 1
gm—;
}
else
{
kg -= a; //if a is not 0, decrement by a
gm -= a;
}
return weight(k,g); //returning the value before decrement
}

int main()
```

```cpp
{
weight w2(0,0);
weight w1(12,500);
cout<<"Original Values...\n";
cout<<"w1: ";
w1.display();
cout<<"w2: ";
w2.display();
w2 = ++w1;
cout<<"\nAfter prefix increment...\n";
cout<<"w1: ";
w1.display();
cout<<"w2: ";
w2.display();
w2 = --w1;
cout<<"\nAfter prefix decrement...\n";
cout<<"w1: ";
w1.display();
cout<<"w2: ";
w2.display();
w2 = w1++;
cout<<"\nAfter postfix increment...\n";
cout<<"w1: ";
w1.display();
cout<<"w2: ";
w2.display();
w2 = w1--;
cout<<"\nAfter postfix decrement...\n";
cout<<"w1: ";
w1.display();
cout<<"w2: ";
w2.display();
//w2 = w1++(5); //invalid
w2 = w1.operator++(5);      //explicit call
cout<<"\nIncrement by 5...\n";
cout<<"w1: ";
w1.display();
cout<<"w2: ";
w2.display();
//w2 = w1--(2);  //invalid
w2 = w1.operator--(2);    //explicit call
cout<<"\nDecrement by 2...\n";
cout<<"w1: ";
w1.display();
```

```
    cout<<"w2: ";
    w2.display();
    return 0;
    }
```

The output of the program

```
    Original Values...
    w1: kilogram= 12 and gram= 500
    w2: kilogram= 0 and gram= 0

    After prefix increment...
    w1: kilogram= 13 and gram= 501
    w2: kilogram= 13 and gram= 501

    After prefix decrement...
    w1: kilogram= 12 and gram= 500
    w2: kilogram= 12 and gram= 500

    After postfix increment...
    w1: kilogram= 13 and gram= 501
    w2: kilogram= 12 and gram= 500

    After postfix decrement...
    w1: kilogram= 12 and gram= 500
    w2: kilogram= 13 and gram= 501

    Increment by 5...
    w1: kilogram= 17 and gram= 505
    w2: kilogram= 12 and gram= 500

    Decrement by 2...
    w1: kilogram= 15 and gram= 503
    w2: kilogram= 17 and gram= 505
```

In Example 3.19, increment (++) and decrement (−) operators are overloaded in both the prefix and postfix forms. When the statements w2 = ++w1 and w2 = −−w1 are executed, the prefix form of increment and decrement operator is invoked, and the incremented and decremented values of w1 are assigned to w2 respectively. However, when the statements w2 = w1++ and w2=w1− are executed, the postfix form of increment and decrement operator is invoked, and the original value of w1 is first assigned to w2 and then w1 is incremented and decremented respectively. The return statement in all the functions returns a temporary instance of the class created by explicitly calling the constructor of the class.

Note that if an argument is passed while calling the postfix increment or decrement operator, the operator function is invoked explicitly using the object name and the dot operator. Thus, the statements w2 = w1++(5) and w2 = w1−(2) are invalid and generate compile-time error.

Overloading Binary Operators

Since the binary operators operate on two operands, one operand, that is, the object of the class invoking the function is passed implicitly to the member operator function. The other operand is passed as an argument, which can be passed either by value or by reference.

Example 3.20: A program to demonstrate the concept of overloading binary operators

```cpp
#include<iostream>
using namespace std;
class weight
{
int kg;
int gm;
   public:
weight()  //default constructor
{
kg = 0;
gm = 0;
}
weight(int k, int g)        //parameterised constructor
{
kg = k;
gm = g;
}
void display()
{cout<<"\nkilogram= "<<kg<<" and gram= "<<gm;}

weight operator+(weight w);      //overloading    +
operator
weight operator-(weight w);      //overloading    -
operator
int operator==(weight w);        //overloading  ==
operator
};
weight weight::operator+(weight w)
{
weight temp;
temp.gm = gm + w.gm;
temp.kg = temp.gm/1000;
temp.gm = temp.gm%1000;
```

```
temp.kg += kg+w.kg;
return temp;
}
weight weight::operator-(weight w)
{
weight temp;
if(gm < w.gm)
{
gm += 1000;
kg—;
}
temp.gm = gm - w.gm;
temp.kg = kg - w.kg;
return temp;
}
int weight::operator==(weight w)
{
int total1,total2;
total1 = (kg*1000) + gm;
total2 = (w.kg*1000) + w.gm;
if(total1 == total2)
    return 1;
else return 0;
}
int main()
{
weight w1(13,400),w2(13,400),w3,w4;
cout<<"First weight is:";
w1.display();
cout<<"\n\nSecond weight is:";
w2.display();
if(w1 == w2) //calling operator==() function
cout<<"\n\nWeights are equal";
else
cout<<"\n\nWeights are not equal";
w3 = w1 + w2;              //calling  operator+()
function
w4 = w1 - w2;             //calling  operator-()
function
cout<<"\n\nTotal weight is:";
w3.display();
cout<<"\n\nDifference of weights is:";
w4.display();
return 0;
}
```

The output of the program

```
First weight is:
kilogram= 13 and gram= 400

Second weight is:
kilogram= 13 and gram= 400
Weights are equal

Total weight is:
kilogram= 26 and gram= 800

Difference of weights is:
kilogram= 0 and gram= 0
```

In Example 3.20, three binary operators +, − and == are overloaded to add, subtract and compare the two objects of the class weight respectively. Note that all these functions accept only one argument of type weight. The other operand, that is, the object invoking the function is passed implicitly. To understand this concept, consider the statements:

```
w3 = w1 + w2;  //implicit call to operator+()
w4 = w1 - w2; //implicit call to operator-()
```

In these statements, the first operand w1 that invokes operator+() and operator−() is passed implicitly to both the functions. Thus, the data members of w1 are accessed directly without using the dot operator, whereas the data members of w2 (passed as an argument) are accessed using the object name and the dot operator (Refer Figure 3.1). However, these two statements can be replaced by normal function calls. To understand this concept, consider the statements:

```
w3=w1.operator+(w2);  //calling the operator functions
using

w4=w1.operator-(w2);  //the  object  name  and  the  dot
operator
            //explicit call
```
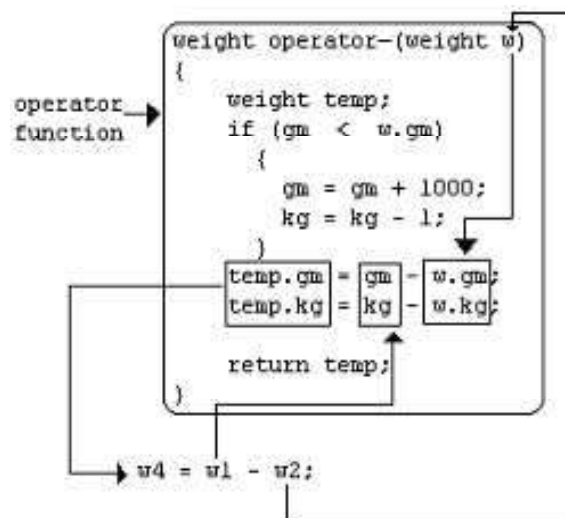

Fig. 3.1 *Overloading '−' Operator*

In Figure 3.1, `w.x` and `w.y` refer to the object `w2` and `x` and `y` refer to the calling object `w1` in `main()`. The local object `temp` is created to hold the results of addition of `w1` and `w2`. The object `temp` is returned and is assigned to `w4` in `main()`.

Note: While overloading binary operators through member functions, the operand that appears on the left-hand side of the overloaded binary operator must be an object of the class for which the operator is overloaded.

Overloading Shorthand Operators

If the data members of the class need to be incremented or decremented by some value other than 1, then instead of overloading the postfix forms of increment or decrement operators (as in Example 3.20), the shorthand operators (+= and −=) can be overloaded.

Example 3.21:

```
# include<iostream>
using namespace std;
 class sample
{
    float a;
float b;
public:
    sample(){}
    sample(float n1, float n2)
    {a=n1;b=n2;}
    sample operator+(sample);
    void display(void);
};
sample sample :: operator+(sample s)
{
    sample temp;
    temp.a=a+s.a;
    temp.b=b+s.b;
    return(temp);
}
void complex :: display(void)
{
cout << a << " +j" << b << "\n";
}
int main()
{
    sample S1,S2,S3;
S1=sample(1.8,5.2);
S2=sample(2.4,3.5);
S3=S1+S2;
cout << "S1= "; S1.display();
```

```
cout << "S2=";S2.display();
cout << "S3=";S3.display();
return 0;
}
```

The output of the program would be:

```
S1=1.8+j5.2;
S2=2.4+j3.5;
```

Here the function operator+() has the following features:

- It receives only one **sample** type argument explicitly.
- It returns a **sample** type value.
- It is a member function of **sample**.

Example 3.22:

```
For example: 3.20
class compute
{
private:
    int cvalue;
public:
    compute()
    {
    cvalue = 15;
    }
    void compute :: operator ++()
    {
    cvalue++;
    }
    void compute :: operatoe –()
    {
    cvalue –;
    }
    void showCompute()
{
      cout << endl
        << "Current compute value ="
        << cvalue
        << eldl;
    }
```

Here the increment and decrement operators have been overloaded. The system has no way to determine that whether these operators are overloaded for prefix or postfix operations. Hence, these operations must be overloaded in such a way that work both for prefix and postfix operations.

To make a distinction between the prefix and postfix notation of these operators, the following syntax is used.

```
operator ++ ()   // for prefix notation
operator ++ (int) //for postfix notation
```

In the above syntax, the use of *int* in parentheses is to indicate the compiler that this operator is being overloaded for postfix use.

Example 3.23: A program to demonstrate the concept of overloading shorthand operators

```cpp
#include<iostream>
using namespace std;
class weight
{
//data members as in Example 3.20
   public:
//constructor and display() function as in Example 3.20
void operator+=(weight w);       //overloading  +=
operator
void operator-=(weight w);       //overloading  -=
operator
};
void weight::operator+=(weight w)
{
gm += w.gm;
kg += w.kg + (gm/1000);
gm = gm % 1000;
}
void weight::operator-=(weight w)
{
if(gm < w.gm)
{
    gm += 1000;
kg—;
}
kg -= w.kg;
gm -= w.gm;
}
int main()
{
weight w1(12,500), w2(10,200);
cout<<"Original Values...\n";
w1.display();
w1 += w2;    //calling operator+= function
cout<<"\nAfter increment...\n";
w1.display();
```

```
w1 -= w2;    //calling operator-= function
cout<<"\nAfter decrement...\n";
w1.display();
return 0;
}
```

The output of the program

```
Original values
kilogram= 12 and gram= 500
After increment...
kilogram= 22 and gram= 700


After decrement...
kilogram= 12 and gram= 500
```

In Example 3.23, the shorthand operators += and -= are overloaded for the class weight. Note that the data members of the calling object (w1) can be accessed directly without using the dot operator. However, the data members of the object (w2), passed as an argument, are accessed with the help of object name and the dot operator.

Overloading Binary Operators for String Manipulations

In C++, built-in operators cannot be used directly with string variables. However, some built-in functions such as strcmp, strcpy, strcat, etc. are used to compare two strings, copy one string to another and concatenate two strings, respectively. For example, if str1, str2, str3 are three string variables, then the following statement generates a compile-time error:

```
str3=str1+str2; //invalid
```

Note that other operators such <, ==, <=, etc. do not generate any compile-time error when used with strings however, do not give accurate results. Thus, these operators have to be overloaded for string manipulations.

Example 3.24: A program to demonstrate the concept of operator overloading for string manipulations

```
#include<iostream>
#include<cstring>
using namespace std;
class string_class
{
char *str;
int size;
   public:
string_class()
{
str = " ";
size = 0;
}
```

```
string_class(char *p)
{
strcpy(str,p);
size = strlen(str);
}
void display()
{cout<<str;}

int operator<(string_class s);    //overloading    <
operator
int operator==(string_class s); //overloading == operator
string_class operator+(string_class s); //overloading +
//operator
};
int string_class::operator<(string_class s)
{
if (size == s.size)
{
if (strcmp(str,s.str)<0)
return 1;
else
return 0;
}
else
if (size<s.size)
return 1;
else
return 0;
}

int string_class::operator==(string_class s)
{
if (strcmp(str,s.str) == 0)
return 1;
else
return 0;
}
string_class string_class::operator+(string_class s)
{
string_class s3;
strcpy(s3.str,str);
strcat(s3.str,s.str);
return s3;
}
```

```
int main()
{
string_class str1("Hello"),str2("World"),str3;
cout<<"First string is: ";
str1.display();
cout<<"\nSecond string is: ";
str2.display();
if (str1 == str2)            //calling operator== function
cout<<"\nStrings are equal";
else
if (str1 < str2)      //calling operator< function
cout<<"\nString 1 is less than string 2";
else
cout<<"\nString 2 is less than string 1";

str3 = str1 + str2;          //calling operator+ function
cout<<"\n\nConcatenated string is: ";
str3.display();

return 0;
}
```

The output of the program

```
First string is: Hello
Second string is: World
String 1 is less than string 2
Concatenated string is: HelloWorld
```

In Example 3.24, a class `string_class` containing a pointer `str` to an array of type `char` and a variable `size` of type `int`, is defined. Three operators <, == and + are overloaded to compare, to check the equality and to concatenate two objects of `string_class` respectively.

3.5 TYPE CASTING

Sometimes we need to carry out conversion from one data type to another. This can be achieved explicitly through what is known as type casting. The programmer can force the compiler to convert one type into an appropriate one at the required places.

Examples

```
int x ;
float y = 2.5 ;
x = int (y) + 5 ;
```

Here int (y) will convert y into integer 2. Therefore, x will be equal to 7.

There is a difference between C and C++ with regard to the notation of casting as given below:

(data type) expression – for instance, (int) `var` [C notation]

data type (expression) – for instance, int (`var`) [C++ notation]

But either of the notations will work in C++.

Example

15.0/int (3.14) will be equal to 5.0 since int(3.14) will be equal to 3. Since the expression is in a mixed mode, result will be a float.

Example

```
Z = float (5/2 * 2);
```

You would expect Z to be 5.0

But it will be 4.0.

Since cast operator has a lower precedence, the expression within parentheses will be evaluated first in this order, $5/2 = 2 * 2 = 4$.

It will be type cast as 4.0.

But the type casting takes place in the statement where the type casting appears explicitly. The data type is not redefined permanently. It continues to be of the type as originally defined. Look at the example given below:

Program 3.1

```
/*to demonstrate type casting*/
#include<iostream>
using namespace std;
int main(){
    int varint;
    float varfloat=20.67f;
    varint=(int)varfloat;
    cout<<"\n value of varint " << varint;
    cout<<"\n value of varfloat remains "<< varfloat;
    varfloat=varint; //automatic conversion
    cout<< "\n value of varfloat " << varfloat;
    cout<<"\n value of varint remains " << varint;
}
```

The following declarations were made.

```
int varint;
float varfloat =20.67;
varint=(int) varfloat;
```

The fractional part will be truncated and we will get `varint=20`. On the contrary, if we try to assign varfloat to varint without type casting, compiler will flag an error. However, the following statement will be accepted:

```
varfloat=varint;
```

It is accepted because we are widening varint and now varfloat gets the value of 20.

The output of the program

```
value of varint 20
value of varfloat remains 20.67
value of varfloat 20
value of varint remains 20
```

Check Your Progress

4. What is operator overloading?
5. Define the term operator function.
6. On what do the binary operators operate?
7. What is type casting?

3.6 ANSWERS TO 'CHECK YOUR PROGRESS'

1. A constructor is a special member function that constructs storage area for the data members of an object by allocating and initialising memory for them. Hence, it makes the object functional by converting an object with the unused (uninitialised) memory into a usable (initialised) object. It is special as it has the same name as that of the class and is automatically invoked whenever an object of the class is created.

2. An object is declared, memory space is allocated to the data members. In addition, during the execution of the program, the object may use other resources like files and so on. These resources and the memory allocated to objects must be released when an object is destroyed. This is accomplished by another special member function called destructor that is automatically invoked to release all the resources and memory that an object acquires during its lifetime.

3. The constructor of the string class is used to construct a string object and initializing its value depending on the constructor version used. The string class includes many constructors but above mentioned string constructors are used in day-to-day programming.

4. Operator overloading is the process that enables an operator to exhibit different behavior, depending on the data being provided.

5. An operator is overloaded with the help of a special function called an operator function.

6. Binary operators operate on two operands, one operand, that is, the object of the class invoking the function is passed implicitly to the member operator function. The other operand is passed as an argument, which can be passed either by value or by reference.

7. Sometimes we need to carry out conversion from one data type to another. This can be achieved explicitly through what is known as type casting.

3.7 SUMMARY

- A constructor is a special member function that constructs storage area for the data members of an object by allocating and initialising memory for them. Hence, it makes the object functional by converting an object with the unused (uninitialised) memory into a usable (initialised) object.

- An object is declared, memory space is allocated to the data members. In addition, during the execution of the program, the object may use other resources like files and so on. These resources and the memory allocated to objects must be released when an object is destroyed.

- The constructor of the string Class is used to construct a string object and initializing its value depending on the constructor version used.

- The string class includes many constructors but above mentioned string constructors are used in day-to-day programming.

- The member function assign() is used for the assignments, it assigns a new value to the string, replacing its current contents.

- Operator overloading is the process that enables an operator to exhibit different behavior, depending on the data being provided.

- An operator is overloaded with the help of a special function called an operator function.

- The operator function defined as a member function of the class is known as member operator function.

- When unary operators are overloaded using member functions, the member operator function does not accept any argument.

- In C++, increment and decrement operators exist in two forms, namely, prefix and postfix.

- The member operator function for both the forms returns an object of the class for which it is being overloaded. The prefix form returns the incremented or decremented value of the object, however, the postfix form returns the original value of the object before incrementing or decrementing.

- Since the binary operators operate on two operands, one operand, that is, the object of the class invoking the function is passed implicitly to the member operator function. The other operand is passed as an argument, which can be passed either by value or by reference.

- If the data members of the class need to be incremented or decremented by some value other than 1, then instead of overloading the postfix forms of increment or decrement operators

- Sometimes we need to carry out conversion from one data type to another. This can be achieved explicitly through what is known as type casting.

3.8 KEY TERMS

- **Constructor:** Constructor is a special member function that constructs storage area for the data members of an object by allocating and initialising memory for them.

- **Default constructor:** It is a constructor that has an empty parameter list and is used to initialise all the objects of a class with the same values.

- **Destructor:** It releases the resources and memory at run-time to clean up the unused storage area.

- **Copy constructor:** The copy constructor makes a new string by copying a string.

- **Member operator function:** It refers to the operator function defined as a member function of the class.

3.9 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. What do you mean by the term calling constructors?

2. Differentiate between constructor and destructors.

3. Define the term copy constructor for string class.

4. What is string class assignment?

5. Write in brief about member operator function.

6. List the operators that can be overloaded in C++.

7. State the points that must be kept in mind while overloading operators.

8. What do you understand by overloading a binary and unary operator?

Long-Answer Questions

1. Explain the types of constructor with appropriate examples.

2. Differentiate between string class assignment and string access operator with the help of example.

3. Discuss the need of operator overloading.

4. Explain the concept of overloading unary and binary operators as member functions with the help of suitable examples.

5. Describe the type casting with appropriate examples.

3.10 FURTHER READING

Jeyapoovan, T. 2006. *Computer Programming: Theory and Practice* (with CD). New Delhi: Vikas Publishing House.

Khurana, Rohit. 2008. *Object Oriented Programming with C++*. New Delhi: Vikas Publishing House.

Saxena, Sanjay. 2009. *Introduction to Information Technology.* New Delhi: Vikas Publishing House.

Rumbaugh, James, Fedrick Blaha, William Premerlani, and Federick Eddy.1990. *Object- Oriented Modelling and Design.* New Jersey: Prentice Hall.

Balaguruswamy, E. 1998. *Object-Oriented Programming*. New Delhi: Tata McGraw-Hill.

UNIT 4 INHERITANCE AND POINTERS

Structure

4.0 INTRODUCTION

Inheritance property facilitates reusability of software components. The user-defined types, namely the classes, facilitate inheritance. Assume that we have developed a program, taking into consideration, the user's requirement. Usually, the client will require some additional features, at the time of delivery after seeing the product. Inheritance allows code reusability. This implies that it facilitates classes to reuse existing code. The new class acquires members of the old class that are already tested and debugged. Hence, inheritance saves time and also increases reliability.

A dynamic data structure is one in which the memory for elements is allocated dynamically at runtime. The successive elements of a dynamic data structure may not be stored in contiguous memory locations but they are still linked together by means of some linkages or references. However, arrays have certain problems associated with them. As array elements are stored in adjacent memory locations, a sufficient block of memory is allocated to an array at compile time. Once the memory space is allocated to an array, it cannot be expanded or contracted. That is why an array is called a static data structure.

The addresses of these elements can be accessed in the program through the use of pointers, which are the vital elements of C++ programming that provide the means for accessing and manipulating the memory locations of the variables directly thus, making the programs more effective and efficient. Moreover, certain features of C++, such as virtual functions and this pointer, also require the use of pointers. The concept of virtual functions further helps in implementing run-time polymorphism, thereby achieving dynamic binding.

In this unit, you will study about the inheritance, derived class, relationships superclass/subclass, multiple inheritance, constructors and destructors, hierarchical inheritance, hybrid inheritance, virtual base classes, C++ memory map and free store, pointers and arrays, reserving and freeing dynamic memory, polymorphism, virtual functions and pure virtual function, early vs. late binding.

4.1 OBJECTIVES

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

- Understand the basics of inheritance
- Explain the derived class
- Discuss the relationship superclass and its subclass
- Describe the concept of multiple inheritance
- Explain the constructors and destructors
- Analyse the hierarchical inheritance and hybrid inheritance
- Explain the meaning of virtual base classes
- Elaborate on the C++ memory map and free store
- Discuss the pointers and arrays
- Analyse the reserving and freeing dynamic memory and polymorphism
- Explain the virtual function and pure virtual functions
- Describe the difference between early and late binding

4.2 INHERITANCE

Inheritance property facilitates reusability of software components. The user-defined types, namely the classes, facilitate inheritance. Assume that we have developed a program, taking into consideration, the user's requirement. Usually, the client will require some additional features, at the time of delivery after seeing the product! After the completion of the project, adding a new feature in the conventional programming languages is not an easy job. It can lead to new errors. On the contrary, in OOP, adding a new feature, after a class has been developed, is rather easy. The class, which is already available (after thorough testing) is known as a base class in C++. Adding a new feature may require either adding a new data element or a new function. This can be achieved by extending the program in OOP. For this, a new class has to be defined as inheriting the base class. This new

class is called a **derived class** in C++. The derived class can inherit some or all of the properties of the base class as per requirements. Adding a new class does not alter the base class. The base class is called the super class and the derived class as the sub-class in some languages. Conceptually there is no difference.

The derivation of properties through inheritance is akin to that in human beings. The child inherits all or some of the properties of the parent. The child may add his own properties. Both the inherited property and the newly acquired property can be used simultaneously by the child. However, the parent is aware of only what he has lent. The child can, in turn become a parent and, lend his properties to his child in a similar manner. This can go on and in C++ there is no limitation to the number of either the levels of inheritance or the derived classes for a base class.

4.2.1 Derived Class

Inheritance is a process of deriving a new class from an already existing class in such a way that the new class inherits all the members of the already existing class. In inheritance, the class which is inherited by the new class is known as base class or superclass or parent. The class which inherits the members of the existing class is known as derived class or sub class or child class. For example, Figure 4.1 shows four classes, named, `animal, carnivore, herbivore and omnivore`. The class `animal` is a base class inherited by the derived classes `carnivore, herbivore` and `omnivore`. The derived classes `carnivore, herbivore` and `omnivore` inherit all the members (properties and functionality) of the base class `animal`.
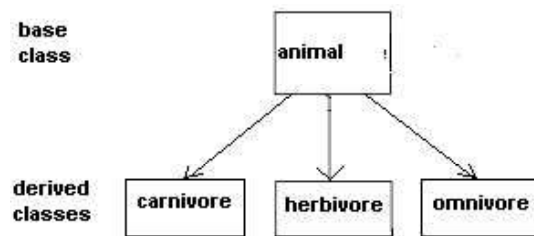
Fig. 4.1 *Inheritance*

In addition to the members of the base class, the derived class also includes its own members. That is, the derived class extends the base class, which is shown in Figure 4.2.
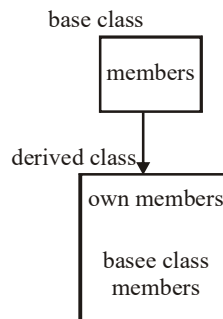
Fig. 4.2 *Derived Class Members*

Defining a Derived Class

Inheritance is implemented while defining the derived class. The name of the base class appears in the definition of derived class. Since the derived class is nonexistent, when base class is defined the inheritance is implemented only at the time of derived class definition.

The syntax to define a derived class is:

```
class derived_class : access_specifier base_class
{
//data members and member function of derived class
};
```

where,

`class` = C++ keyword

`derived_class` = the name of the derived class

`:` = depicts that `derived_class` is inherited from `base_class`

`access_specifier` = the access specifier of the base class—can be any of the three keywords, `private`, `public` or `protected`

`base_class` = the name of the base class

Example 4.1: A code segment defining a derived class

```
class polygon   // base class
{
   protected:
    int length, height;
   public:
    void setval(int a, int b);
};
class triangle: public polygon //Definition of derived
class
{

   public:
    int area();
void display();
};
```

In Example 4.1, two classes, namely, `polygon` and `triangle` are defined. Here, the class `triangle` inherits the class `polygon`. This implies, `triangle` is the derived class and `polygon` is the base class. The keyword `public` is the access specifier of the base class `polygon`.

Note: The friend functions of a base class are not inherited as they are not the members of the base class.

Accessibility of Base Class Members in Derived Class

As stated earlier, the derived class inherits all the members (public, protected and private) of the base class. However, only the public and protected members can be accessed. The private members are accessed indirectly using the protected and the public member functions of the base class.

Example 4.2: A program to demonstrate the accessibility of the members of the base class within a derived class

```
#include<iostream>
using namespace std;
class Shapes3D
{
   private:
    int length;
   protected:
    int breadth;
   public:
    int height;
    void setval(int, int, int );
};
void Shapes3D::setval(int a, int b, int c)
{length=a; breadth=b; height=c; }
class cube: public Shapes3D
{
   public:
    int volume(){return (length *breadth* height );}
    //error,length is private data member and is not
//accessible
void displayvol(){cout<<"Volume : "<<volume();}
};
```

In Example 4.2, the class `cube` inherits the class `Shapes3D`. In the member function `volume` of the class `cube`, the base class members, namely, `length`, `breadth` and `height` are accessed. Note that the statement `int volume(){return (length *breadth* height );}` generates a compile-time error as `length` is a private member of class `Shapes3D` and hence it is not accessible inside `cube`. However, the protected base class member `breadth` and the public base class member `height` can be directly accessed.

The accessibility of base class members by objects of the base class and inside a derived class is listed in Table 4.1.

Table 4.1 Accessibility of Base Class Members

Base Class Members	Accessibility by Objects of Base Class	Accessibility Inside the Derived Class
Public Members	Accessible	Accessible
Protected Members	Not Accessible	Accessible
Private Members	Not Accessible	Not Accessible

Note: The derived class members cannot be accessed inside its base class or by the objects of the base class.

Access Specifier of the Base Class

The access specifier of the base class in the derived class definition determines the way the derived class inherits the base class. It determines the access specifier of the base class members inside the derived class. The access specifier of a base class member in the derived class depends on the access specifier (visibility mode) provided while defining the derived class. Depending on the access specifiers `public`, `protected` or `private`, a base class can be publicly inherited, protectedly inherited or privately inherited, respectively.

Public Inheritance
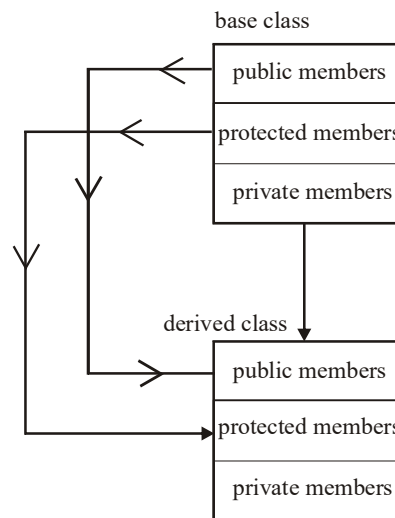
Fig. 4.3 *Publicly Inherited Base Class*

When the access specifier of the base class in the derived class definition is `public`, the base class is publicly inherited. The access specifier of the members of the base class remains the same in the derived class. This implies, the public members and protected members of the base class remain the public members and protected members of the derived class, respectively, as shown in Figure 4.3.

Note: Since private members of a base class are not accessed in the derived class the concept of data hiding is not violated.

When a base class is publicly inherited, the derived class object can access only the public members of the base class. The protected members of the base class are inaccessible by the objects of the derived class. However, they can be accessed by the member functions, friend classes and the friend functions of the derived class.

The syntax to define a derived class that publicly inherits its base class is:
```
class derived_class:public base_class
{
    .   //members of the derived_class
        ÷
};
```

Example 4.3: A program to demonstrate a publicly inherited base class

```cpp
#include<iostream>
#include<cstring> //for strcpy()
using namespace std;
class employee
{
   protected:
    int age;
    char name[20];
    char address[30];
   public:
    int ssn;
void getval(int, int, char [], char []);
};
void employee::getval(int sn, int ag, char str[], char
add[])
{
ssn=sn;
age=ag;
strcpy(name, str);
strcpy(address, add);
}
class engineer:public employee //Base class publicly
inherited
{
char engtype[15];
   public :
    void gettype(char type[]){strcpy(engtype, type);}
    void showeng(void);
};
void engineer::showeng()
{
cout<<"Serial no :"<<ssn<<endl;
cout<<"Age :"<<age<<endl;
cout<<"Name :"<<name<<endl;
cout<<"Address :"<<address<<endl;
cout<<"Engg Type :"<<engtype<<endl;
}
int main()
{
engineer e1;
e1.getval(20,63,"Sqn Ldr N Wesley","R.K Puram");
e1.gettype("Aeronautical Engineer");
e1.ssn=15;//public member of base class can be accessed
```

```
                        //when inherited publicily
        //e1.age=45;protected member age is inaccessible
    e1.showeng();
    return 0;
    }
```

The output of the program

```
    Serial no :15
    Age :63
    Name :Sqn Ldr N Wesley
    Address :R.K Puram
    Engg Type :Aeronautical Engineer
```

In Example 4.3, the class `employee` is publicly inherited by the class `engineer`. This implies that the public member `ssn` and the protected members `age`, `name`, `address` of the class `employee` become public and protected members of class `engineer`, respectively. In `main()`, the object `e1` of the class `engineer` is used to access the data members `ssn` and `age`. Note that the statement `e1.age=45` generates a compile-time error as `age` is a protected member of `engineer` and is not accessible by the objects of the derived class.

Protected Inheritance
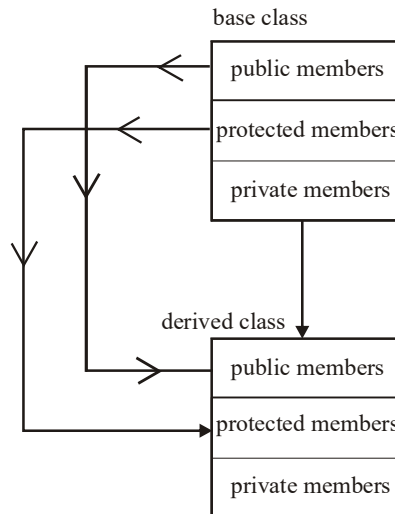
Fig. 4.4 *Protectedly Inherited Base Class*

When the access specifier of the base class in the derived class definition is `protected`, the base class is protectedly inherited. Both the public and the protected members of the base class become the protected members of the derived class.

When a base class is protectedly inherited, public and protected members of the base class are not accessible by the objects of the derived class. However, these base class members are accessible by the member functions, the friend classes and the friend functions of the derived class (Refer Figure 4.4).

The syntax to define a derived class that protectedly inherits its base class is:

```
class derived_class : protected base_class
{
    .   //members of the derived_class
    .
};
```

Example 4.4: A program to demonstrate a protectedly inherited base class

```
#include<iostream>
using namespace std;
class figure
{
   protected:
float area;
   public:
    float perimeter;
};
class  rectangle: protected  figure //class  inherited
protectedly
{
   public:
    float length,breadth;
    void calarea(){ area=(length*breadth);}
    void calperimeter(){perimeter=2*(length+breadth);}
};
int main()
{
    rectangle r;
    r.length=4.7;
    r.breadth=5.5;
    r.calarea();
    r.calperimeter();
    // cout<<"area is: "<<r.area;
       //area the protected member, object cannot access
    // cout<<"\nperimeter is: "<<r.perimeter;
       //perimeter the protected member in rectangle class
//object cannot access protected member perimeter
    return 0;
}
```

In Example 4.4, the class figure is protectedly inherited by the class rectangle. This implies that the protected member area and the public member perimeter of the base class figure become the protected members of rectangle. In main(), if the object r of the class rectangle tries to access the data members area and perimeter of the base class, a compile-

time error is generated as `area` and `perimeter` are now the protected members of `rectangle`.

Private inheritance
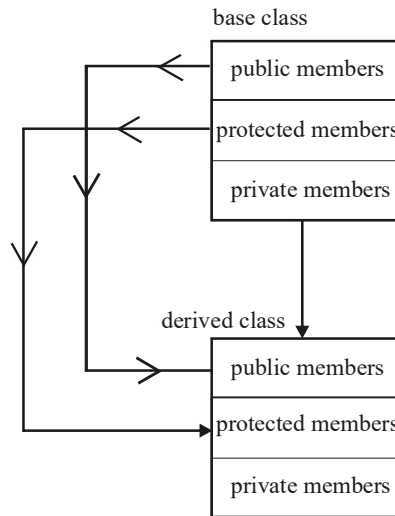
Fig. 4.5 Privately Inherited Base Class

When the access specifier of the base class in the derived class definition is `private`, the base class is privately inherited. In this case, both the public and the protected members of the base class become the private members of the derived class, as shown in Figure 4.5.

When a base class is privately inherited, the public and the protected members of the base class are not accessible by the objects of the derived class. However, these base class members are accessible by the member functions, friend classes and the friend functions of the derived class.

The syntax to define a derived class that privately inherits its base class is:

```
class derived_class : private base_class
{
    .   //members of the derived_class
    .
};
```

Note: By default, the access specifier of a base class in the derived class definition is `private`.

Restoring the Access Specifier of the Base Class Members

When a base class is privately or protectedly inherited, the access specifier of all its public and protected members changes in the derived class. Sometimes, the access specifier of a public or protected member of the base class may need to be retained in the derived class. This can be accomplished by declaring such member in the derived class explicitly with its original access specifier.

The syntax to restore the access specifier of a base class member in the derived class is:

```
class derived_class : access_specifier base_class
{
.
.
  access_specifier:
base_class::member_name;
.
};
```

where,

:: = the scope resolution operator—indicates that member_name is a member of base_class

member_name = the name member of the base class whose access specifier needs to be restored

Note that while declaring a base class member in the derived class, only name of the base class member is provided. The data type (for data members) or the parameter list and the return type (for member functions) need not be specified.

Example 4.5: A program to restore the access specifier of a base class member in the derived class

```
#include<iostream>
using namespace std;
class triangle
{
  public:
int s1,s2,s3,perimeter;
};
class righttriangle : protected triangle
{
  public:
   int angle1,angle2;
   triangle::perimeter;  //restoring access specifier
};
int main()
{
   righttriangle rt;
   rt.perimeter=25;  //valid as perimeter has become
//the public data member
   return 0;
}
```

In Example 4.5, the class triangle is protectedly inherited by the class righttriangle. Thus, the public members s1, s2, s3 and perimeter of triangle become protected members of righttriangle. However, in righttriangle, the access specifier of perimeter is restored by

declaring it with the `public` access specifier. As a result, `perimeter` now becomes a public member of `righttriangle`.

Some points must always be kept in mind while inheriting the classes. They are as follows:

- The base class should be inherited with the `public` visibility mode if the access specifier of the members of the base class is not required to be altered.

- The base class should be inherited with the `private` visibility mode if the members of the base class are not required to be further inherited.

- The base class should be inherited with the `protected` visibility mode if the members of the base class are to be hidden outside the class but can be further inherited.

Allocating Memory to the Objects of Base Class and Derived Class

Like ordinary variables, memory is also allocated to the objects of a class. The total number of bytes allocated to an object of a class is equal to the sum of the bytes allocated to the public, protected and private data members of the class. However, the size of an object of the derived class is equal to the sum of the size of all the data members of the base class as well as derived class.

Example 4.6: A program to demonstrate the memory space occupied by the base class and derived class objects

```
#include <iostream>
using namespace std;
class base
{
    char x[20]; //20 bytes
  protected:
    char y[30]; //30 bytes
  public:
    float z;    //4 bytes
};
class derived: public base
{
    float a;    //4 bytes
    char b[20]; //20 bytes
};
int main()
{
    base b;
    derived d;
    cout<<"Size of object of base class: "<<sizeof(b);
cout<<"\nSize of object of derived class: "<<sizeof(d);
    return 0;
}
```

The output of the program is

```
Size of object of base class: 54
Size of object of derived class: 78
```

In Example 4.6, the size of the object b of the base class base is sum of the sizes of its data members, that is, $20+30+4=54$. However, the size of the object d of the derived class derived is equal to the sum of the sizes of its data members and the size of the base class data members, that is, $4+20+54=78$.

4.2.2 Relationships Superclass/Subclass

Inheritance is a relationship between a superclass and its subclasses. It is a mechanism by which a new class be built from an existing class.

Superclass and Subclass

The class that is inherited by other classes is called a base class or superclass or parent class. The class that inherits the properties of the superclass is called a subclass or derived class or child class. A subclass inherits all the instance variables and methods defined by the superclass, at the same time it also contains its own members. For example, in Figure 4.6, Animal is the superclass which is inherited by three subclasses Carnivore, Herbivore and Omnivore. Hence, Carnivore, Herbivore and Omnivore inherit all members of the superclass Animal.
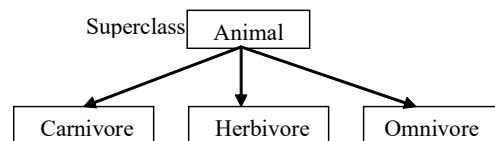
Fig. 4.6 *Superclass and Subclasses*

Defining a Subclass

Inheritance is implemented while defining a subclass. The name of the superclass is specified in the subclass definition. A subclass can be defined by using extends keyword.

The syntax to define a subclass is as follows:

```
class sub_class extends super_class
{
//variables and methods declaration
}
```

where,
sub_class is the name of the subclass that inherits the superclass.
super_class is the name of the superclass that is being inherited.
extends is the keyword that indicates that the super_class properties have been extended to the sub_class.

4.2.3 Multiple Inheritances

When a derived class inherits from more than one base class simultaneously, it is referred to as multiple inheritance. In multiple inheritance, the derived class inherits

the members of all its base classes and can directly access the public and the protected members of its base classes. For example, Figure 4.7 shows multiple inheritance in which the derived class *owner* is inherited from two base classes, namely, person and company.
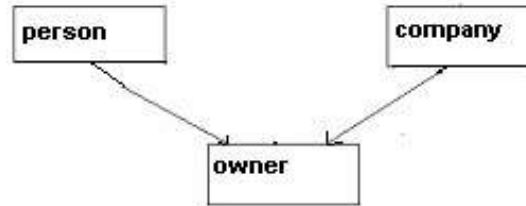
***Fig. 4.7** Multiple Inheritance*

The syntax to define the derived class that implements multiple inheritance is:

```
class derived_class: access_specifier1 base_class1,
access_specifier2 base_class2. . .,access_specifierk
base_classk
{
   .
   .
};
```

Example 4.7: A program to demonstrate multiple inheritance

```
#include<iostream>
#include<cstring>              //for strcpy()
using namespace std;
class person
{
int ssn, age;
char name[20], address[30];
   public :
    void getval(int sn, int ag, char st[], char add[])
    {
ssn=sn;   age=ag;
strcpy(name,st);
      strcpy(address,add);
}
    void show_person();
};
void person::show_person()
{
    cout<<"Serial No. :"<<ssn<<endl;
    cout<<"Name :"<<name<<endl;
cout<<"Age :"<<age<<endl;
    cout<<"Address :"<<address<<endl;
}
```

```
class company
{
    char cname[20], caddress[30];
  public:
   void setval(char cn[], char cadd[])
   {
strcpy(cname, cn);
strcpy(caddress, cadd);
}
   void show_com(void)
   {
cout<<"Company Name :"<<cname<<endl;
      cout<<"Company Address :"<<caddress<<endl;
   }
};
class owner : public person, public company
         //inheriting multiple base classes
{
   int licenceno;
  public :
   void getlicno(int lno)
{
licenceno=lno;
}
   void show_lno()
{
cout<<"Licence No :"<<licenceno<<endl;
}
};
int main()
{
owner o1;
   o1.getval(111,33,"N Suzana Wesley","R.K Puram");
   o1.setval("MS Vision","Cannaught Place");
   o1.getlicno(123456);
   o1.show_person();
   o1.show_com();
   o1.show_lno();
   return 0;
}
```

The output of the program
```
Serial No. :111
Name :N Suzana Wesley
```

```
Age :33
Address :R.K Puram
Company Name :MS Vision
Company Address :Cannaught Place
Licence No :123456
```

In Example 4.7, the class `owner` simultaneously inherits two classes, namely, `person` and `company`. This implies that `person` and `company` are the base classes of `owner`.

Ambiguity Resolution in Multiple Inheritance

In multiple inheritance, an ambiguity may arise when a member of two or more base classes has same name. In order to resolve such ambiguities, the member name is qualified with the base class name by using the scope resolution operator (`::`).

Example 4.8: A program to demonstrate the ambiguity in multiple inheritance

```cpp
#include<iostream>
#include<cstring>
using namespace std;
class bank
{
   protected :
    int code;
    char bname[15], baddress[25];
   public :
void getdetail(int c, char bn[], char badd[])
   {   code=c; strcpy(bname,bn);
       strcpy(baddress,badd);
}
};
class person
{
   protected:
    int code, age;
    char name[20], address[30];
   public :
    void getval(int sn, int ag, char st[], char add[])
    {   code=sn; age=ag; strcpy(name,st);
       strcpy(address,add);
}
};
class owner:public bank, public person
{
    int regno;
    public :
```

```
        void getregno(int reg){regno=reg;}
        void showreg();
    };
    void owner::showreg()
    {
        cout<<"Serial No. :"<<person::code<<endl;
                //code of person class
        cout<<"Age :"<<age<<endl;
        cout<<"Name :"<<name<<endl;
        cout<<"Address :"<<address<<endl;

        cout<<"Bank code :"<<bank::code<<endl;
                //code of bank class
        cout<<"Bank name :"<<bname<<endl;
        cout<<"Bank address :"<<baddress<<endl;

        cout<<"Registeration No: "<<regno<<endl;
    }
    int main()
    {
        owner o1;
        o1.getdetail(1345,"PNN Bank","Anand Vihar");
        o1.getval(112,27,"Himanshu Ahuja","Anand Vihar");
        o1.getregno(12345);
        o1.showreg();
        return 0;
    }
```

The output of the program

```
Serial No. :112
Age :27
Name :Himanshu Ahuja
Address :Anand Vihar
Bank code :1345
Bank name :PNN Bank
Bank address :Anand Vihar
Registeration No: 12345
```

In Example 4.8, the class owner simultaneously inherits the classes bank and person. The class bank and person have the same data member code which can lead to ambiguity. The ambiguity can be resolved using name of the class and the scope resolution operator (::) along with the code data member in the showreg() member function. However, if the scope resolution operator is removed, a compile-time error is generated.

4.2.4 Construction, Destructors in Inheritance

A constructor allows the object to be initialised with the valid values at the time of (its) object declaration. A constructor is required in inheritance to initialise the data members of the base class through derived class. The derived class defines its own constructor in order to initialise its new members. However, to initialise the data members inherited from the base class, constructor of the base class is called. After the constructors are called, the destructors are also called to destroy the objects. However, constructors and destructors are called in a particular order.

Note: A derived class does not inherit the constructors and the destructor of its base class. However, constructor and destructor of the base class are explicitly called inside the derived class.

Order of Calling of Constructor and Destructor

When an object of the derived class is declared, the base class constructor is called first and then the derived class constructor is called. The constructor of the base class is called first because the base class is unaware of the derived class. Moreover, the base class members need to be initialised first as the derived class initialisation depends on the base class initialisation. In other words the, derived class uses base class members for its initialisation.

However, when an object of the derived class is destroyed, the destructors are called in reverse order. The destructors are executed in reverse order as base class is a foundation for the derived class, therefore, the destruction of the base class object implies destruction of the derived class object. Hence, the derived class destructor is called before the base class destructor.

Example 4.9: A program to demonstrate the order in which constructors and destructors are called

```
#include<iostream>
#include<cstring>
using namespace std;
class wood
{
   protected :
char type[10];
   public:
    wood ()
    {  trcpy(type, "Teak");
       cout<<"Base class constructor wood called "<<endl;
}
    ~wood()
    {
       cout<<"Base class destructor called "<<endl;
    }
};
class table:public wood
{
```

```
     char dimension[5];
   public:
    table()
    {
       strcpy(dimension, "2X4");
cout<<"Derived class constructor table called\n";
    }
    ~table()
    {
       cout<<"Derived class destructor called "<<endl;
    }
};
int main()
{
table t1;
return 0;
}
```

The output of the program

```
Base class constructor wood called
Derived class constructor table called
Derived class destructor called
Base class destructor called
```

In Example 4.9, the derived class table inherits the base class wood. When the object t1 of the class table is declared in main(), constructor of wood is called first and then the constructor of table is called. However, when t1 is destroyed (when main() terminates), the destructor of table is called first and then the destructor for wood.

Note: The public and the protected members of the base class can be directly initialised using assignment statements in the body of the derived class constructor.

Constructors and Destructors in Multiple Inheritance

In multiple inheritance, the base class constructors are called in the order in which the base classes appear in the definition of derived class, that is, from left to right. However, when an object of the derived class is destroyed, the derived class destructor is called before any of the base class destructors is called. The base class destructors are called in the reverse order of calling the base class constructors that is, from right to left.

Constructors and Destructors in Multilevel Inheritance

In case of multilevel inheritance, constructors are called in the order of their inheritance and destructors in the reverse order of inheritance. The constructor of the indirect base class is called first, then the constructor of the direct base class and finally, the derived class constructor is called. However, in case of destructor, the derived class destructor is called first, then the destructor of direct base class is called, and finally, the destructor of indirect base class.

Constructors and Destructors of Virtual Base Classes

If a derived class inherits both a virtual base class and a non-virtual base class, then the constructor of the virtual base class is called before the constructor of the non-virtual base class. The destructors are called in the reverse order of calling of the constructors. This implies, the destructor of the non-virtual base class is called before the destructor of the virtual base class.

Note that if a derived class is inherited from multiple virtual base classes, then the constructors of the virtual base classes are called in the order in which they are specified in the derived class definition, while the destructors are called in the reverse order of calling of constructors.

Parameterised Constructor of Base Class

Parameterised constructors are used to dynamically initialise the object of the class. The parameterised constructors of the base class can be initialised by providing the values through the derived class. This is accomplished by explicitly calling the parameterised constructor of the base class in the header of the derived class constructor. The arguments of the parameterised constructor can be provided through the arguments of the derived class constructor.

The syntax to define a derived class constructor that explicitly calls one or more base class constructor is:

```
Derived_class::derived_class(parameter_list):base_class1(parameter_list1),
base_class2(parameter_list2),..,base_classk(
parameter_listk)
{
    .    //body of the constructor
    .
}
```

Here,   base_class1(parameter_list1), base_class2(parameter_list2) and so on are the explicit calls to the base class constructors.

Note: In the case of multiple inheritance, the parameterised constructor of the base classes are called in the order in which they appear in the definition of derived class and not in the order in which they are called in the header of the derived class constructor.

Example 4.10: A program to demonstrate the calling of a parameterised constructor of the base class

```
#include<iostream>
using namespace std;
class aaa
{
    int i;
  public:
    aaa(int x)
{
      i=x;
cout<<"Value of i in the base class aaa "<<i<<endl;
}
```

```
    ~aaa()
    {
    cout<<"Destructor aaa called\n";
    }
    };
    class bbb
    {
        int j;
      public:
       bbb(int y)
          {
          j=y;
          cout<<"Value of j in the base class bbb "<<j<<endl;
    }
    ~bbb()
    {
cout<<"Destructor bbb called\n";
}
};
class ccc:public bbb, public aaa
{
                     int k;
   public:
     ccc(int z, int m, int n):aaa(z), bbb(m)
                                                  // ex-
plicit call to the base class constructor
{
k=n;
cout<<"Value of k in the derived class ccc "<<k<<endl;
}
~ccc()
{
cout<<"Destructor ccc called\n";
}
};
int main()
{
ccc c1(1,2,3);
return 0;
}
```

The output of the program

```
  Value of j in the base class bbb 2
  Value of i in the base class aaa 1
  Value of k in the derived class ccc 3
  Destructor ccc called
  Destructor aaa called
  Destructor bbb called
```

In Example 4.10, when the object `c1` of the class `ccc` is declared, the constructor of `c1` explicitly calls the constructors of `aaa` and `bbb` and passes the appropriate arguments to them. Note that constructor of `bbb` is called before the constructor of `aaa` and then the constructor of `ccc` is called. When `c1` is destroyed, the destructors are called in the reverse order of constructors.

Note: If a base class has a parameterised constructor, it is mandatory for the derived class to define a constructor and pass the arguments for the base class constructor.

4.2.5 Hierarchical Inheritance

Hierarchical inheritance is a type of inheritance in which more than one class is derived from a single base class. In hierarchical inheritance, a base class provides members that are common to all of its derived classes. For example, Figure 4.8 shows hierarchical inheritance in which two classes graduate and undergraduate are derived from single base class student.
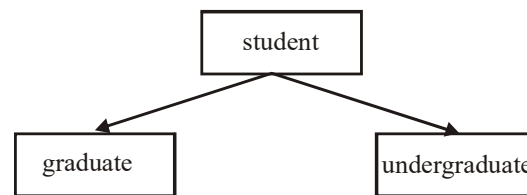
Fig. 4.8 *Hierarchical Inheritance*

In hierarchical inheritance, all the base class members are inherited by each of the derived class. In addition, the public and protected members of the base class are directly accessible by all the derived classes. However, members of any derived class cannot be accessed by another derived class.

The syntax to implement hierarchical inheritance (with two derived classes) is:

```
class base_class
{
    .    //members of base_class
    .
};
class derived_class1: access_specifer1 base_class
{
    .    //members of derived_class1
    .
};


class derived_class2: access_specfier2 base_class
{
   . //members of derived_class2
    .
};
```

Note: If multiple classes are derived from a single base class, each derived class maintains its own copy of the private, public and protected data members of the base class.

4.2.6 Hybrid Inheritance

According to the user requirement the various types of inheritance, can be combined. This type of inheritance is known as hybrid inheritance. For example, Figure 4.9 shows hybrid inheritance that involves multilevel and multiple inheritances.
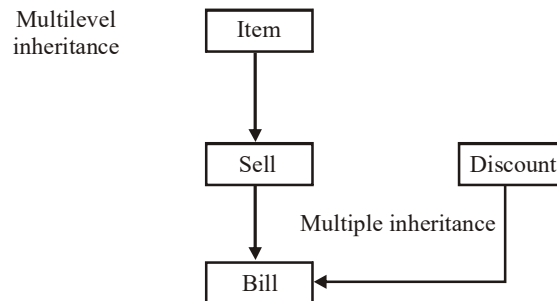
Fig. 4.9 *Hybrid Inheritance*

Example 4.11: A program to demonstrate hybrid inheritance

```cpp
#include<iostream>
#include<cstring>
using namespace std;
class item
{
   protected:
    char itemcode[5], itemname[15];
    int costp,qtyoh;
   public:
    void getitem(char code[], char itnm[], int cp, int
qt)
    {
       strcpy(itemcode,code); strcpy(itemname,itnm);
       costp=cp; qtyoh=qt;
    }
    void itemdis();
};
void item::itemdis()
{
cout<<"Item code :"<<itemcode<<endl;
    cout<<"Item name :"<<itemname<<endl;
    cout<<"Cost price :"<<costp<<endl;
    cout<<"Quantity on hand :"<<qtyoh<<endl;
}
class sell:public item
{
   protected:
    int sp ;
   public:
```

```
        void getval(int slprice)
        {
            sp=slprice;
        }
void showsp(){cout<<"Selling Price :"<<sp<<endl;}
};
class discount
{
   protected:
    int dis;
   public :
    void getdis(int d){dis=d;}
};
class bill:public sell,public discount
{
    int qty, totalprice;
   public :
    int calprice(int q)
    {
        qty=q; totalprice=(sp*qty)-dis;
        return totalprice;
    }
    void showbill();
};
void bill::showbill()
{
showsp();
cout<<"Qty to be purchased :"<<qty<<endl;
    cout<<"Discount :"<<dis<<endl;
    cout<<"Total price :"<<totalprice<<endl;
}
int main()
{
    bill b1;
    b1.getitem("co01","Computer",20000,15);
    b1.getval(30000);
    b1.getdis(2000);
    b1.calprice(2);
    b1.itemdis();
    b1.showbill();
    return 0;
}
```

The output of the program

```
Item code :co01
Item name :Computer
Cost price :20000
Quantity on hand :15
Selling Price :30000
Qty to be purchased :2
Discount :2000
Total price :58000
```

In Example 4.11, the derived class `bill` inherits the base class `sell`, which in turn is derived from the base class `item`. Hence, multilevel inheritance is formed. The class `bill` inherits two base classes `discount` and `sell` forming the multiple inheritance. The multiple and multilevel inheritance in this example together form hybrid inheritance.

Ambiguity Resolution in Hybrid Inheritance

In some hybrid inheritance ambiguity arises when the derived class directly accesses a member of its indirect base class. For example, as shown in Figure 4.10, the class `apprentice` is inherited from two base classes `employee` and `student`, which in turn are inherited from the class `person`. Therefore, an indirect base class `person` is inherited twice by the `apprentice` class. Hence, the class `apprentice` has two copies of all the members of its indirect base class `person` twice. Thus, ambiguity arises while accessing these members.
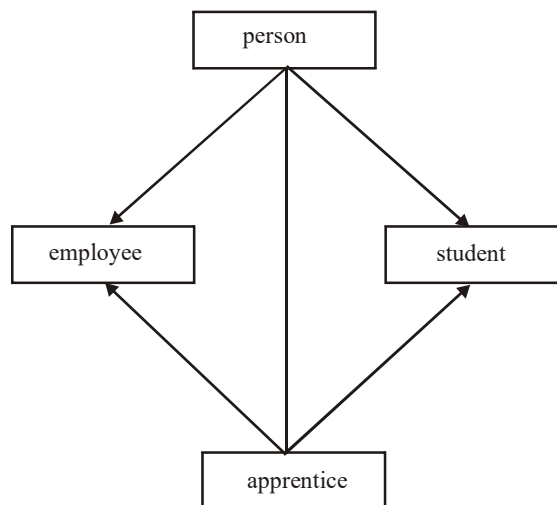
Fig. 4.10 *Hybrid Inheritance*

Example 4.12: A program to demonstrate the ambiguity when multiple, hierarchical and multilevel inheritance are combined

```cpp
#include<iostream>
#include<cstring>
using namespace std;
class person
{
```

```
                              protected:
                          int ssn,age;
                          char name[20],address[30],gender[2];
                              public:
                          void getper(int sn, int ag, char nm[],
                          char add[], char gen[])
                          {
                          ssn=sn; age=ag; strcpy(name,nm);
                          strcpy(address,add); strcpy(gender,gen);
                          }
                          void showper();
                          };
                          void person::showper()
                          {
                          cout<<"Security Serial Number :"<<ssn<<endl;
                          cout<<"Name :"<<name<<endl;
                          cout<<"Age :"<<age<<endl;
                          cout<<"Address :"<<address<<endl;
                              cout<<"Gender :"<<gender<<endl;
                          }
                          class employee:public person
                          {
                              protected:
                          int salary;
                              public:
                          void getsal(int sal){salary=sal;}
                          void showsal(){cout<<"Salary :"<<salary<<endl;}
                          };
                          class student:public person
                          {
                              protected:
                          char majordept[15];
                              public:
                          void getmaj(char maj[]){strcpy(majordept,maj);}
                          void showmaj(){cout<<"Major :"<<majordept<<endl;}
                          };
                          class apprentice:public employee, public student
                          {
                              protected:
                          int duration;
                              public:
                          void getdur(int d){duration=d;}
                          void showdur()
                          {cout<<"Duration : "<<duration<<" years"<<endl;}
```

```
};
int main()
{
apprentice a1;
a1.getper(111,29,"N Suzana Srivastava","R.K Puram","F");
//error
a1.getsal(20000);
a1.getmaj("History");
a1.getdur(4);
a1.showper();//ambiguity
a1.showsal();
a1.showmaj();
a1.showdur();
}
```

In Example 4.12, two derived classes `employee` and `student` are derived from the base class `person`. The classes `employee` and `student` become base classes to `apprentice`. Hence, the derived class `apprentice` has all the members of the direct base classes (`employee` and `student`) as well as its indirect base class `person`. Note that class `apprentice` inherits two copies of members of class `person` through class `employee` and class `student`. This leads to ambiguity. Hence, a compile-time error is generated when the object `a1` uses the member function `getper()` and `showper()` as two copies of base class members are present in object `a1`.

The ambiguity discussed in Example 4.13 can be resolved by qualifying the member with the name of either of the direct base classes using the scope resolution operator.

To resolve the ambiguity, the program in Example 4.12 can be rewritten as shown in this example.

Example 4.13: A program to resolve the ambiguity

```
#include<iostream.h>
class person
{
//body same as in the Example 4.12

};
class employee:public person
{
    //body same as in the Example 4.12
};
class student:public person
{
    //body same as in the Example 4.12
};
```

```
class apprentice:public employee, public student
{
    //body same as in the Example 4.12
};
int main()
{
    a1.student::getper(111,29,"Krishna  Masih","R.K
Puram","F");
a1.student::showper();//ambiguity resolved
}
```

In this example, the class name student and scope resolution operator is used to resolve the ambiguity.

The output of the program

```
Security Serial Number :111
Name :Krishna Masih
Age :29
Address :R.K Puram
Gender :F
Salary :20000
Major :History
Duration :4 years
```

4.3 VIRTUAL BASE CLASSES

In Example 4.13, the scope resolution operator enables to refer to specific class for accessing its members; however, it does not prevent the duplication of indirect base class members. The duplication of inherited members due to multiple paths can be prevented by making common base class as virtual base class at the time of declaration of intermediate base class. Virtual base class is an indirect base class declared using the keyword virtual in order to prevent its duplication. Specifying a base class as virtual ascertains that only one copy of the base class members exists for its derived classes. In other words, the virtual base class can be derived several times without its duplication. (Refer Figure 4.10)

The syntax to specify a virtual base class is:

```
class derived_class: virtual access_specifier base_class
{
    .   //member function of derived_class
    .
};
```

Note: The position of the virtual keyword and the access specifier of the base class can be interchanged. This implies, the syntax class derived_class : access_specifier virtual base_class {..}; is also valid in C++.

Example 4.14: A program to demonstrate the concept of virtual base class

```
#include<iostream>
#include<cstring>
```

```
using namespace std;
class person
{
    //body same as Example 4.12
};
class employee:public virtual person
{
    //body same as Example 4.12
};
class student:public virtual person
{
//body same as Example 4.12
};
class apprentice:public employee, public student
{
//body same as Example 4.12
};
int main()
{
apprentice a1;
a1.getper(111,29,"N Suzana Srivastava","R.K Puram","F");
a1.getsal(20000);
a1.getmaj("History");
a1.getdur(4);
a1.showper();   // no ambiguity
a1.showsal();
a1.showmaj();
a1.showdur();
}
```

In Example 4.14, the class `person` is specified as a virtual base class while defining `employee` and `student`. This implies that only one copy of data members and member functions exists for both `employee` and `student`. As a result, when object `a1` accesses the members of indirect base class `person`, no ambiguity arises.

4.4 C++ MEMORY MAP FREE STORE

A dynamic data structure is one in which the memory for elements is allocated dynamically at runtime. The successive elements of a dynamic data structure may not be stored in contiguous memory locations but they are still linked together by means of some linkages or references. Whenever a new element is to be inserted, memory is allocated for it dynamically and just linked to the data structure. The elements can be inserted as far as memory is available; thus, there is no upper limit on the number of elements in the data structure. Similarly, whenever an element is

deleted from the data structure, memory is de-allocated so that it can be reused in future. An example of a dynamic data structure is a linked list.

Static and Dynamic Memory Allocation

Allocation of memory is a unique and special task performed by the compiler when any program is compiled and run. The specifications for static and dynamic type of allocation are defined in the program module. Both static and dynamic memory allocation are discussed as follows:

Static Memory Allocation

In static memory allocation, memory is allocated at compilation time. For example, if we initialize as int arr[] {1 2 3}, then the compiler will automatically allocate the required memory space for declared variables, in this case 3 integers, at the time of compilation. The address of operator is used to acquire the reserved address which may be assigned to a pointer variable. As most of the declared variables contain static memory address, hence assigning pointer value to a pointer variable is termed as static memory allocation. Static memory is pre-allocated at the time of mapping process into the main memory.

This technique's application includes a program module (e.g., function or subroutine), which declares a static data locally. This is done in a manner in which these data are inaccessible in other modules till it receives references as parameters or returns them. A single copy of static data is held back and is accessible through many calls to the function in which it is declared. Static memory allocation, therefore, has the advantage of modularizing data within a program design in the situation where these data must be retained through the runtime of the program.

The use of static variables within a class in object-oriented programming enables a single copy of such data to be shared between all the objects of that class.

Dynamic Memory Allocation

While in dynamic memory allocation, memory is assigned at run time, malloc( ) compiler will merely view it as a function and an argument.

It makes use of functions such as malloc( ) or calloc( ) to get memory dynamically. In case these functions are made use of to get memory dynamically and the values returned by these functions are assigned to pointer variables, then they are called dynamic memory allocation. Memory is assigned during run time.

Dynamic memory is allocated on the heap space of the process map. The pointer reference helps in the visibility throughout the process.

In computer science, dynamic memory allocation (also called heap-based memory allocation) is the allocation of memory storage that can be used in a computer program during the runtime of that program. It can also be viewed as a method of distributing ownership of limited memory resources among several pieces of data and code.

Dynamically allocated memory exists till it is release either explicitly by the programmer, or by the garbage collector. On the other hand, static memory

allocation has a fixed duration. It is said that an object so allocated has a *dynamic lifetime*.

Implementations

- **Fixed-Size-Blocks Allocation:** Fixed-size-blocks allocation, also known as memory pool allocation, makes use of a free list of fixed-size blocks of memory (often all of the same size). This is known to work well for simple embedded systems.

- **Buddy Blocks:** In this system, memory is allocated from a large block in memory that is a power of two in size. If the block is more than twice of the desirable size, it is split into two. One of the halves is selected, and the process repeats (checking the size again and splitting if needed) until the block is just large enough.

    All the blocks of a particular size are kept in a sorted linked list or tree. When a block is freed, it is compared with its buddy. In case they are both free, they are combined and placed in the next largest size buddy-block list. (When a block is allocated, the allocator will start with the smallest sufficiently large block avoiding needlessly breaking blocks.)

Static and Dynamic Variables

A static variable upholds the same data all through the execution of a program while a dynamic variable can have different values during the course of a program.

Dynamic variables are allocated on the stack, so they are by default 'thread-local,' assuming a thread-safe implementation. By specifying them in a `set` statement in the thread's initial function, it makes them available (via `use` statements) to all the functions called in the same thread, as if they were global variables.

Theoretically, the `set` statement creates a set of dynamic variables and pushes them on the global stack of such sets. Reaching the end of the `set` statement pops the stack. The `use` statement searches the sets from the top of the stack down for each identifier listed in `use` statement and, if the identifier is found, stores the address of the dynamic variable in a local variable of the same name. Within the `use` statement, values of the dynamic variables are referenced indirectly. The `set` statement can be implemented with no allocation overhead, as all of the allocations can be done at compile time as local variables. The stack of sets is a list of structures defined by the following pseudo-C code, one for each dynamic variable.

```
struct dVariable {
struct dVariable *link;
const char *name;
Type *type;
void *address;
}
```

`dVariable` instances are connected via the `link` field. The `name` field points towards the name of the variable, the `type` field points towards the type descriptor sufficient for testing the subtype relation and the `address` field contains the address of the variable.

Dynamic variables are those that can have the space allocated to them at some point during the execution of a program or procedure and that can also be disposed of and have their space given back to the system by the program.

Dynamic and Static Variables in C

- Variable declarations can be done outside all functions or inside a function.
- Declarations made outside the functions are global and in fixed memory locations.
  - o The static declaration declares a variable outside a function to be a 'File Global' (cannot be referenced by code in other source files).
- Declarations within a block statement {} (function body or block statement nested within a function body) have the following features:
  - o They are dynamically allocated, unless declared static.
  - o They are allocated in memory when program execution enters the block.
  - o They occur when memory is released and when the execution exits the block.
  - o They occur if a function calls itself (directly or indirectly) and it gets a new set of dynamic variables (called a stack frame).
  - o They are handled like any other call to the function.

Static Memory Use of Static Variable

When a program begins to execute, there must be some specific blocks of memory set aside for use that cannot be trespassed upon by any other program or even by the system for instance, the memory containing the program's own code. While it is possible (in machine language) to write a program that can modify its own code, it is very dangerous practice and must never be done.

Moreover, any variables named in the declaration section must have a specific memory set aside for their contents, and this action can not be controlled or changed in any way by the programmer, except by declaring more or fewer variables in the first place. The memory in question can not itself be relocated to some other place or expanded or contracted.

Static variables are those that are created in the declaration section of a program and continue to exist (whether visible or not) and require space until its conclusion. Their space is allocated at the beginning of the program run.

The only change that can take place in static variable is their content and this change is done by assignment statements.

Items of all the data types, that have been taken into account till now, are of the static kind–once declared, they will be at a fixed location and consume a specific amount of memory during the running of the program. Both the size and location (relative to the start of the code) are predetermined at the time the program is compiled. Its location is relative and not absolute as there is no way for the compiler to determine ahead of time how many programs will be running and what memory will already be in use when the new program is loaded. However, with respect to program starting address, it is fixed and cannot be changed by the program.

Figure 4.11 illustrates a popular method of allocating memory. A block is a memory set aside for the program's use, and within this, the code is placed first (at the lowest address) and this is followed by the static variable space.
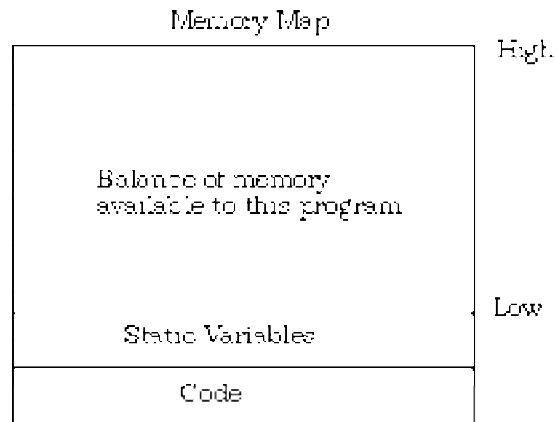
Fig. 4.11 *Allocation of Memory for Static Variable*

The area of memory into which the procedure activation records are dynamically and automatically placed is called stack and the marker that delimits the top end of the currently allocated stack is termed as the stack pointer.

Pointers

A pointer is a programming language data type whose value points directly to (or 'Points to') another value stored elsewhere in the computer memory by using its address. For high-level programming languages, pointers effectively take the place of general-purpose registers in low-level languages, such as assembly language or machine code—in contrast, such languages occupy a certain part of the available memory. A pointer refers to a location in memory, and by obtaining the value at the location it is known as dereferencing the pointer.

The pointer can be considered as a simple, less abstracted implementation of the highly abstracted reference data type. Several languages support some type of pointer. However, every language imposes restrictions on pointers which make them unique. Generally, copying and dereferencing pointers is much cheaper in time and space as compared to copying and accessing the data to which the pointers point

Pointers to data extensively improve performance for repetitive operations such as navigating strings, lookup tables, control tables and tree structures. In addition, in procedural programming, pointers are used to hold the addresses of entry points of subroutines as well as for run-time linking to Dynamic Link Libraries (DLLs). Also, in object-oriented programming, the pointers to functions often use virtual method tables for binding methods.

The term 'Pointer' is generally used to refer to references. Additionally, it is also used by data structures whose interface evidently allows the manipulation of pointers (by using pointer arithmetic*)* as a memory address. This is because pointers allow both protected as well as unprotected access to memory addresses. However, there are potential risks involved while using them primarily unprotected access to memory addresses.

When setting up, it is necessary to have pointers help manage the creation, implementation and control of data structures such as lists, queues and trees. Start pointers, end pointers and stack pointers are classic examples of pointers that are either absolute (the actual physical address or a virtual address in virtual memory) or relative (an offset from an absolute start address). These pointers actually use less bits rather than a full address. However, they usually require an additional arithmetic operation in order to be resolved.

Pointers and Dynamic Memory Allocation

Dynamically allocated blocks of memory are used to store data objects or arrays of objects. These blocks make use of pointers to store and manage their addresses. The dynamic allocation of objects is done in a heap or free store; which refers to an area of memory that is provided by most of the structured and object-oriented languages

The memory of your computer can be considered to be a succession of memory cells, each one of the minimal size—one byte—that your computer manages. These single-byte memory cells are numbered in a successive way. This is done so that within any block of memory, every cell has the same number as the previous one plus one.

Reference Operator (&)

As soon as a variable is declared, the amount of memory needed is assigned at a specific location in memory (its memory address). Generally, you do not actively decide the precise location of the variable within the panel of cells of the memory. Actually, during runtime, this task is automatically performed by the operating system. However, in some cases, you may want to know the address of where the variable is being stored during runtime, to operate with the relative positions of it.

Reference to a variable refers to the address that locates a variable within memory. This reference to a variable can be gained by preceding the identifier of a variable with an ampersand sign (&), known as a reference operator. This can be literally translated as the 'Address of'.

Dereference Operator (*)

As known, a variable which stores the reference of another variable is called a pointer. So, pointers are said to 'point to' the variable whose reference they store.

Using a pointer, you can directly access the value stored in the variable points to. To do this, you have to precede the pointer's identifier with an asterisk (*) only. This sign acts as dereference operator and that can be literally translated as 'Value Pointed by'.

Declaring Variables of Pointer Types

Owing due to the ability of a pointer to directly refer to the value that it points to, it becomes necessary to specify in its declaration which data type a pointer is going to point to. However, it is not the same thing as pointing a `char` as to point to an `int` or a `float`.

The declaration of pointers follows the following format:

```
type * name;
```

where `type` is the data type of the value that the pointer points to. This `type` is not the type of the pointer itself, but the type of the data the pointer points to. For example:

```
1 int * number;
2 char * character;
3 float * greatnumber;
```

These are declarations of pointers. Each one is intended to point to a different data type; In fact all of them are pointers and all of them occupy the same amount of space in memory (the size in memory of a pointer depends on the platform where the code is executed). Nevertheless, the data to which they point to do not occupy the same amount of space nor are of the same type: the first one points to an int, the second one to a char and the last one to a float. Therefore, although these example are variables, all of them are pointers which occupy the same size in memory, but have different types: int*, char* and float* respectively, depending on the type they point to.

4.4.1 Pointers and Arrays

The concept of array is very much bound to the pointer. In fact, the identifier of an array is equal to the address of its first element, as a pointer is. For example, assuming these two declarations:

```
1 int numbers [20];
2 int * p;
```

The following assignment operation would be valid:

```
p = numbers;
```

After that, `p` and `numbers` would be equivalent and will have the same properties. The only difference is that you could change the value of pointer `p` by another one, whereas `numbers` will always point to the first of the 20 elements of type `int` with which they were defined. Therefore, unlike `p`, which is an ordinary pointer, `numbers` is an array, and an array can be considered a constant pointer. Therefore, the following allocation will not be valid:

```
numbers = p;
```

As `numbers` is an array, so it operates as a constant pointer, and you cannot assign values to constants.

Null Pointer

A null pointer is a regular pointer of any type which has a special value that indicates that it is not pointing to any valid reference or memory address. This value is the outcome of type-casting the integer value zero to any pointer type.

```
1 int * p;
2 p = 0;     // p has a null pointer value
```

Null pointers are different from void pointers. A null pointer is a value that any pointer may take to represent that it is pointing to 'Nowhere,' whereas a void pointer is a special type of pointer which can point somewhere without a specific type. One indicates towards the value stored in the pointer itself and the other towards the type of data it points to.

4.4.2 Memory Representation in Free Store

To maintain a linked list in the memory, two parallel arrays of equal size are used. One array (for example INFO) is used for the info field and another array (say, NEXT), for the next field of the nodes of the list. The values in the arrays are stored such that the ith locations in arrays INFO and NEXT, contain the info and next fields of a node of the list, respectively. In addition, a pointer variable Start is maintained in the memory that stores the starting address of the list. Figure 4.12 shows the memory representation of a linked list where each node contains an integer.
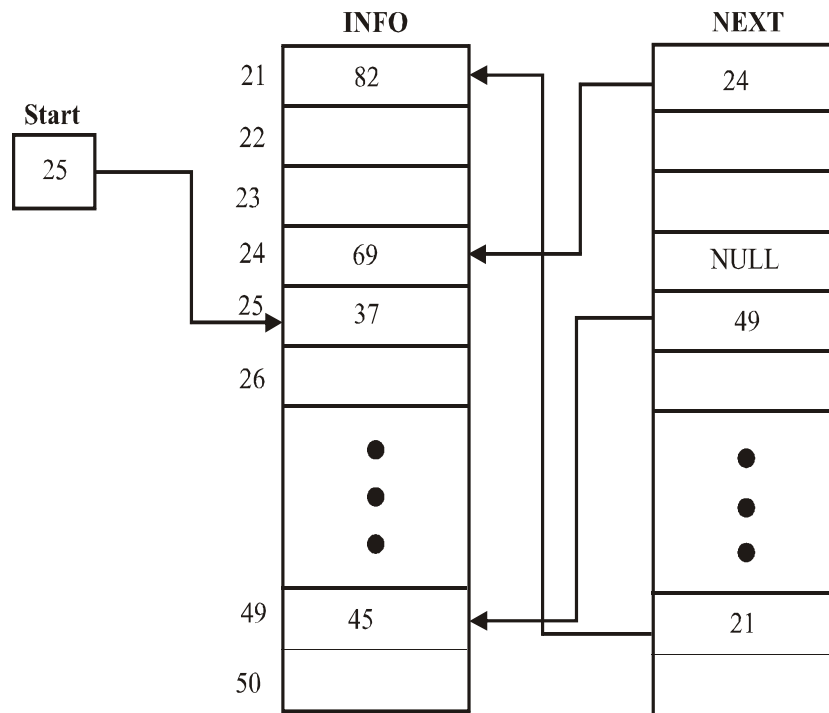
Fig. 4.12 *Memory Representation of a Linked List*

In Figure 4.12, the pointer variable Start contains 25, that is, the address of the first node of the list. This node stores the value 37 in the array INFO, and its corresponding element in the array NEXT stores 49. 49 the address of the next node in the list. Similarly it is for other nodes. Finally, the node at address 24 stores value 69 in the array INFO and NULL in the array NEXT, thus, it is the last node of the list. Note that the values in the array INFO are stored randomly and the array NEXT is used to keep track of the values in the list.

Memory Allocation

As memory is allocated dynamically to a linked list, a new node can be inserted anytime in the list. For this, the memory manager maintains a special linked list

known as **free storage list** or **memory bank** or **free pool** that consists of unused memory cells. This list keeps track of the free space available in the memory and a pointer to this list is stored in a pointer variable `Avail` (Refer Figure 4.13). Note that the end of a free storage list is also denoted by storing `NULL` in the last available block of memory.
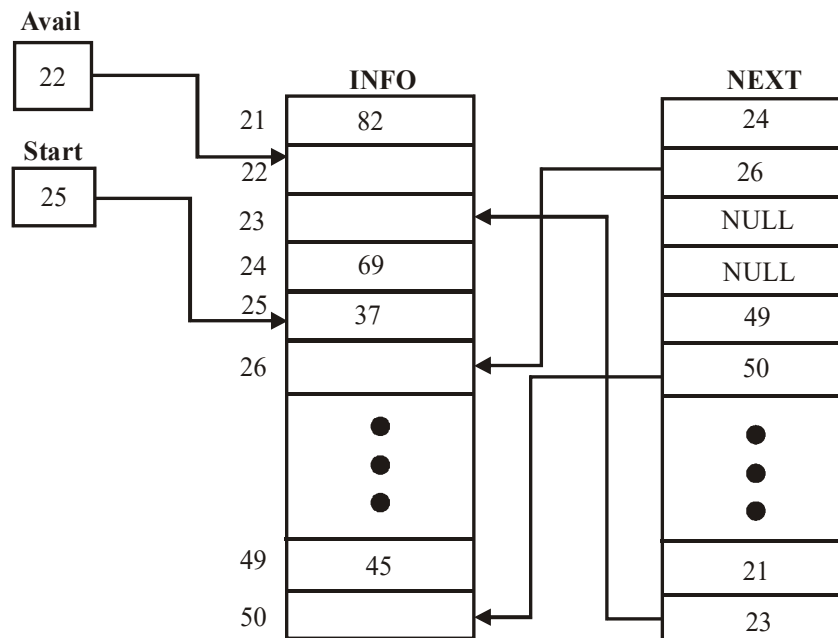
Fig. 4.13 *Free Storage List*

In Figure 4.13, `Avail` contains 22, hence, `INFO[22]` is the starting point of the free storage list. Since `NEXT[22]` contains 26, `INFO[26]` is the next free memory location. Similarly, other free spaces can also be accessed and `NULL` in `NEXT[23]` indicates the end of the free storage list.

While creating a linked list or inserting an element into a linked list, whenever a request for the new node arrives, the memory manager searches through the free storage list for the block of desired size. If the block of desired size is found, it returns a pointer to that block. However, sometimes there is no space available, that is, the free storage list is empty; this situation is termed as **overflow**. In this situation, the memory manager replies accordingly.

Check Your Progress

1. Define the term base class.

2. What are the different ways in which the base class can be inherited?

3. What do you mean by subclass?

4. Name the two base classes in multiple inheritance.

5. What is hierarchical inheritance?

6. Define the term hybrid inheritance.

7. What is a dynamic data structure?

8. What is pointer?

4.5 RESERVING AND FREEING DYNAMIC MEMORY

The dimension or array size declaration of an array is an important subject. Array dimension is to be declared before compiling. For instance, some valid declarations are:

```
char name[25] ;
int mark[40] ;
```

You have not come across any problem since you were initializing the arrays and hence, the array size was known. If you were to get the array elements at runtime, sometimes you could give either a lesser number of elements or more elements. In the former case garbage values will be stored in the empty spaces in the array misleading the user. In the latter case, the elements will be lost. To avoid this problem, you may think that you can specify marks [n] and give the value of n later at runtime. However, this will not work and the compiler will force you to give the actual dimension. Hence, dynamic memory allocation is useful. malloc and **calloc** serves to specify the actual dimension at runtime and hence, enable memory allocation dynamically. Next time when you execute the program, you can specify any value for n. It will definitely work.

The functions malloc () and calloc () allocate memory dynamically. The specifications are as follows:

The statement

```
void * malloc (n*size n );
```

returns the pointer to n bytes of memory or NULL if allotment is not possible. Since it returns a pointer, you have to specify the array as a pointer variable as shown:

```
int *b ; / * array declared * /
b = (int *) malloc (x * 2) ;  / * x is the array size * /
```

Similarly, the calloc is defined as void * calloc (sizen, size_size). Here, the number of arguments are two. The first one is the array size and the second one is the bytes occupied per element, i.e., the storage space required for the datatype. The function returns a pointer allocating space for an array of size sizen, of datatype size but the array contents will be initialized to zero. In malloc, the initial contents will be garbage values. We can use it as follows:

```
int * b;
b = (int *) calloc (x, 4);
```

Here, 4 indicates the array of type float and space is allocated to store x floats. When you use calloc or malloc, you must include alloc.h.

Program using Dynamic Memory Allocation

A program to demonstrate malloc and calloc is given below:

Example 4.15: To do string concatenation by using dynamic allocation of memory.

```
#include <stdio.h>
#include <alloc.h>
```

```
#include <string.h>
main ()
{
    char *newstrcat(char *dest, char *src);
    char *name1, *name2;
    int n1, n2;
    printf("Enter size of 2 names:\n");
    scanf("%d%d", &n1, &n2);
    name1 = (char *) calloc(n1, 1);
    name2 = (char *) calloc(n2, 1);
    printf("Enter 2 names\n");
    scanf("%s%s", name1, name2);
    printf("New name:%s\n",newstrcat(name1,name2));
}
char *newstrcat(char *dest, char *src)
{
    char *w;
    int i,len,len1,cnt=0;
    len=strlen(dest);
    len1=strlen(src);
    w=(char *)malloc(len+len1+1);
    for(i=0;i<len;i++)
        w[cnt++]=dest[i];
    for(i=0;i<len1;i++)
        w[cnt++]=src[i];
        w[cnt]=0;
    return(w);
}
```

The output of the program

```
Enter size of 2 names:
4 4
Enter 2 names
Rama samy
New name:Ramasamy
```

calloc is used to allot memory to name1 and name2. The function newstrcat is called while printing. In the called function malloc is used to allot space equal to the size of name1, name2 +1. The additional space is for placing NULL. The string is concatenated by copying one character at a time using two for statements. Finally, the concatenated string is returned to the main function where it is printed.

The memory allocated using malloc, calloc can also be freed, when it is no longer necessary by using the function free (). For instance, in the main () of the above example, after the last statement we can add the following statements;

```
free(name1);
free(name2);
```

These statements will deallocate the memory space allocated to them after the job of printing the concatenated string is over. Thus, dynamic allocation of memory as per the exact need and freeing it after use is quite useful for conserving memory. This concept is used by professional programmers when they develop commercial software products. Memory saved is more than money saved in such product development.

4.6 POLYMORPHISM

When a class has several different function declarations which are specified by single name in the same scope, the functions are said to be overloaded. When the function is called, the correct function is selected by comparing the types of the actual arguments with the types of the formal arguments. This is called *function overloading*. Similarly, operator overloading can be carried out. Therefore, one operator executes different operations depending upon the type of arguments. Operator overloading is possible only with the existing operators. In fact, C++ allows declaration of overloaded operators as a function. The basic difference between an overloaded operator and an overloaded function is that: the number of arguments for a given operator are predefined and the overloaded operator may appear in the natural form rather than the function call form. The following code is an example of operator overloading.
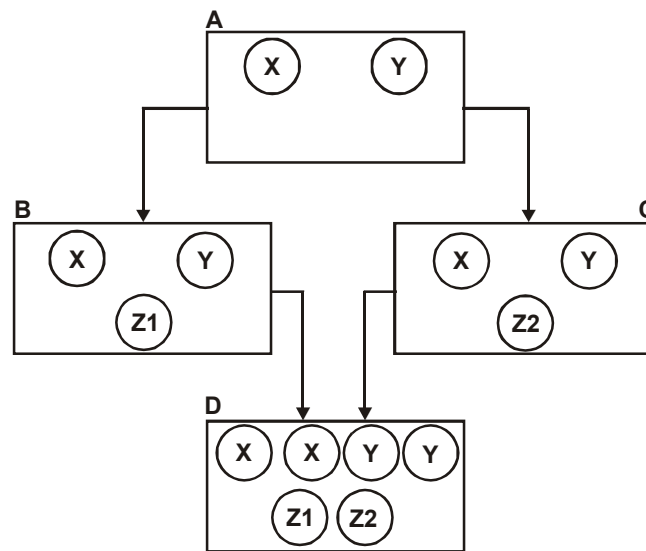
Fig. 4.14 *Class D with Duplicate Copies of Properties of A.*

```
#include<iostream.h>
class complex
{
private:
int real, imag; public:
complex() {} complex(int re,int im) {
```

```
real = re; imag = im; } complex operator+(const complex
&x)
{
return complex(real + x.real,imag+x.imag); }
void value() {
cout << "real: "<< real <<" Imaginary: "<<imag; } };
void main() {
complex a(5,3),b(2,4),c;
c = a+b;
c.value(); }
```

The operator can be overloaded either as a member function or as a friend function. In the case of a friend function, the `friend` keyword must be added before the function name, and the friend function has one more argument in case of operator overloading. For example, the above overloaded operator with the friend option is defined as:

```
friend complex operator+(const complex &x,const complex
&y) {
return complex(x.real+y.real,x.imag+y.imag); }
```

The friend keyword allows programmers to designate either specific functions or the classes whose functions can access not only public members but protected and private members. Operator overloading member functions need only one argument for the binary operators, and no argument for the unary operator, as one of the arguments would be the object from which the function is called. This argument is passed through the `this` pointer, which passes its own address of the object. Notice that, friend functions are not a member of the class.

Runtime polymorphism can be obtained by using virtual functions. A virtual function is declared in a base class by using the keyword `virtual` in front of its declaration. The virtual function may then be overloaded in derived classes. A virtual function overloaded in a derived class is treated differently, as the base class function is taken by default if the derived class does not have any implementation of that function. Otherwise, depending on the address of the object the pointer holds, the respective function will be called. The following program illustrates this.

Program 4.1

```
#include<iostream.h>
class A
{
int a ; public:
A() { a=10; } virtual void value()
cout « "a = "<<a<<endl; } };
class B:public A {
int b; public:
B() { b=20; }
void value()
{
```

```
cout « "b = "«b«endl;
} >;
void main() C
A aobj,*aptr;
B bobj;
aobj.value();
bobj.value();
aptr = &aobj;
aptr->value();
aptr = &bobj;
aptr->value(); }
```

The output of the program

```
The output of this program is: a = 10 b = 20 a = 10
b = 20
```

Notice that if a pointer to an object of the base class is declared, then pointers to objects of the derived class may be assigned to this base class pointer.

Templates and Exception Handling

A template is used to create a parameterized class in which various types of members are specified using formal parameters when the class is declared. The template class syntax requires the prefix

```
template < class T >
```

before the definition of the template class. It specifies that a template is being declared and that a type name *T* will be used in the declaration. The following sample program shows the usage of class template.

Program 4.2

```
#include<iostream.h> template <class T> class sample {
T i; public:
sample(T a) {i=a;}
void value()
{
cout << i<< endl;
> };
void main() {
sample<int> si (10);
sample<float> s2(20.5);
si.value();
s2.value(); }
```

The output of the program

The output of this program is:
```
10
20.5
```

Since *sample* is the template class, where ever the name is used, the template argument should be given. A function template can also be used in a similar manner. The member function *value* of the above program is defined as a template function which is given below.

```
template<class T> void sample<T>::value() { ,   cout <<
i<< endl; }
```

The concept of exception handling is used when the programmer wishes to display or do something on his own if any error occurs while running the program. The keywords used here are `try`, `throw` and `catch`. Before taking up further explanation, consider the following program.

Program 4.3

```
#include <iostream.h> class error_handler
{
int size; public:
error_handler(int x) {
size = x; }
class ranged; //exception class void check(int y) {
if (y>size)
throw range ( ) ; cout << "Given range is accepted"<<
endl; } };
void main() {
error_handler eh(10);
try
{
eh.check(8);
eh.check(15);
eh.check(6); }
catch(error_handler::range) {
cout << "Given range is rejected"<< endl; }
```

The output of the program

The output of this program is:

Given range is accepted

Given range is rejected

In this program the argument value of the function `check` is verified with `size`. If it is lesser than or equal to the value of `size`, the value will be accepted, Otherwise, it will throw out the corresponding exception. Once the thrown exception is caught, the required operation is carried out. Here, we are simply rejecting the value. In order to throw out an exception, we must have an exception class such as `range`, in this program.

4.7 VIRTUAL FUNCTIONS

Virtual functions are one of the attributes of C++ that support run-time polymorphism. A virtual function is a member function that is declared inside the

base class and its functionality can be overridden in the derived classes. When the base class containing virtual function is inherited, the derived classes may implement their own versions of that function. The entire function of the base class can be replaced by a set of new implementation in the derived class. This implies that the base class provides a common interface and this interface can be implemented in different ways in different derived classes. A member function can be made virtual by prefixing its declaration with the keyword `virtual` in the base class. For example, consider the base class definition:

```
class base
{
    public:
virtual void display()    //virtual function
{
cout<<"Base class";
}
};
```

In this class definition, the function `display()` of the class `base` is declared `virtual`. Thus, the `display()` function can be redefined in the derived classes of the class `base`.

Note: If the virtual function is declared inside and defined outside the base class, then the keyword `virtual` does not require to be specified in the definition

Whenever a virtual function is inherited, its virtual nature also gets inherited into all its derived classes and it is not required to use the keyword `virtual` in the subsequent derived classes. For example, consider the derived class definition:

```
class derived:public base
{
    public:
void display()    //overriding virtual function
{
cout<<"Derived class";
}
};
```

In addition, when a derived class that inherits a virtual function and is used as a base class, the virtual function can still be overridden. This implies that a virtual function always remains virtual irrespective of the number of times it is inherited.

A virtual function must be accessed using a pointer of base class type and not of derived class type. However, if the function is not declared `virtual` and the pointer of base class is made to point to the derived class, the function of the base class is always executed. That is, the compiler selects a function on the basis of type of the pointer instead of its contents. However, if the function is declared `virtual` in the base class, the compiler selects which version of the function is to be executed depending upon the contents of the pointer. Moreover, this decision is made at the run-time, thereby, employing dynamic binding. Hence, declaring a member function `virtual` informs the compiler that the function call is resolved at run-time. For example, consider the statements:

```
base b;                //object of base class
derived d;             //object of derived class
base *bptr;            //pointer of type base
bptr=&b;               //pointing to base class
bptr->display();       //display() of base class
bptr=&d;               //pointing to derived class
bptr->display();       //display() of derived class
```

If a virtual function is not redefined in the derived class, a call to that function uses the function implementation defined in the base class. Moreover, like a non-virtual member function, a virtual member function is directly accessible in the derived class with the help of the scope resolution operator (: :). However, in this case, the mechanism of virtual function will not work and the base class implementation of the function is called.

Memory Management

For each class containing at least one virtual function, a v-table (pronounced as virtual table) is constructed in the memory by the compiler. The v-table stores the base addresses of all the virtual functions defined in the class. Each object of the class (containing one or more virtual functions) contains a vptr (pronounced as virtual pointer) in the beginning of the object in the memory that points to v-table in the memory (Refer Figure 4.15). Note that if a virtual function is not implemented in the derived class, the v-table of derived class stores the address of the function defined in the base class.
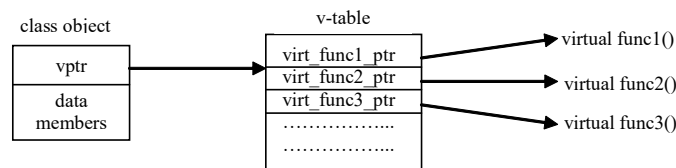
Fig. 4.15 Virtual Table in Memory

When a virtual function is called by an object, the vptr of that object provides the base address of the v-table for that class, which in turn provides the address of the function, called by the object. This is how a virtual function call is resolved at run-time and dynamic binding is achieved.

Note: There exists only one v-table for each class in spite of the number of virtual functions contained in it.

Example 4.16: A program to demonstrate virtual function

```
#include<iostream>
using namespace std;
class person
{
   protected:
char name[20];
   public:
virtual void getdata()      //virtual function
```

```
{
cout<<"Name: ";
cin>>name;
}
virtual void display(){ }    //empty virtual function
};
class emp : public person
{
int emp_no;
   public:
void getdata()           //virtual function overriding
{
cout<<"Employee no.: ";
cin>>emp_no;
person::getdata();      //getdata() of person
}
void display()           //virtual function overriding
{
cout<<"\nEmployee's Details\n";
cout<<"Employee no.: "<<emp_no<<"\n";
cout<<"Name: "<<name<<"\n";
}
};
class stu : public person
{
int roll_no;
   public:
void getdata()           //virtual function overriding
{
cout<<"Roll no.: ";
cin>>roll_no;
person::getdata();       //getdata() of person
}
void display()          //virtual function overriding
{
cout<<"\nStudent's Details\n";
cout<<"Roll no: "<<roll_no<<"\n";
cout<<"Name: "<<name<<"\n";
}
};
int main()
{
person* ptr[5];         //array of pointers to person
int i=0;  char ch;
```

```
      do{
      cout<<"Enter data for employee or student(e/s): ";
      cin>>ch;
      if(ch=='e')
      {
      ptr[i]=new emp;    //pointing to object of emp
      cout<<"Enter details of employee\n";
      }
      else
      {
      ptr[i]=new stu;    //pointing to object of stu
      cout<<"Enter details of student\n";
      }
      ptr[i++]->getdata();   //getdata() of ith object
      cout<<"Want to enter another (y/n): ";
      cin>>ch;
      }while(ch=='y' || ch=='Y');
      for(int j=0;j<i;j++)
      ptr[j]->display();     //display() of jth object
      return 0;
      }
```

The output of the program

```
Enter data for employee or student(e/s): e
Enter details of employee
Employee no.: 103
Name: Smith
Want to enter another (y/n): y
Enter data for employee or student(e/s): s
Enter details of student
Roll no.: 12
Name: Robert
Want to enter another (y/n): n
Employee's Details
Employee no.: 103
Name: Smith
Student's Details
Roll no: 12
Name: Robert
```

In Example 4.16, the functions getdata() and display() of the base class person are declared as virtual and are overridden in the derived classes emp and stu. An array of pointers ptr to person is declared in main() and depending on the user's choice, the elements of ptr point to the objects of either emp or stu class. Note that whenever ptr points to the object of emp class, the functions getdata() and display() of the emp

class are called. Similarly, the functions `getdata()` and `display()` of the `stu` class are called when `ptr` points to the object of the `stu` class.

Note: A class declaring or inheriting a virtual function is known as polymorphic class and the objects of such a class are known as polymorphic objects.

The base class virtual function and its redefined versions in the derived classes must have the same prototype. However, if the prototype differs, the compiler considers these functions as overloaded functions and the virtual function mechanism is ignored. Some of the points regarding virtual functions are as follows:

- A virtual function cannot be global or static, however, it can be declared as a friend function of another class.
- The constructors of a base class cannot be made virtual since at the time the constructor is invoked, the virtual table is not available in the memory. However, the destructors can be made virtual.
- On incrementing or decrementing the pointer of base type, it will always point to the next or the previous object, respectively of its base class type irrespective of the contents of the pointer.

4.7.1 Pure Virtual Functions

As stated earlier, if a derived class does not redefine a virtual function, the base class implementation of virtual function is invoked. However, in some cases, either no meaningful definition of the virtual function exists in the base class or each derived class is required to define its own version of the virtual function. To handle such cases, C++ provides pure virtual functions.

A virtual function having no definition within the base class is called pure virtual function. A virtual function can be made pure by appending the pure specifier "=0" to its declaration in the base class.

The syntax for declaring a pure virtual function is:

```
virtual return_type function_name(parameter_list)=0;
```

To understand the concept of pure virtual functions, consider Example 4.17. In this example, the virtual function `display()` of the class `person` is defined without a body. Moreover, this function has never been called within the program. Thus, `display()` can be made a pure virtual function as shown in this example.

Example 4.17: A code segment to demonstrate pure virtual function

```
class person
{
...   //as defined in Example 4.17
...
virtual void display()=0;    //pure virtual function
};
```

Note that if a virtual function is declared as pure, it must be redefined in all of its derived classes otherwise a program error occurs. In addition, a virtual function cannot have both the pure specifier and the definition.

Abstract Classes

A class that contains at least one pure virtual function is known as an abstract class or abstract base class. An abstract class is different from a polymorphic class. An abstract class cannot be instantiated, as at least one of its members (that is pure

virtual function) lacks implementation. This implies that abstract classes can only be used to act as a base class for other classes and not for instantiation. In addition, an abstract class cannot be used as an argument or the return type of a function. However, a pointer to an abstract class can be made. Moreover, an abstract class can be derived from a non-abstract class by overriding its non-pure virtual function with a pure virtual function.

For example, the class person (as defined in Example 4.17**)** is an abstract class as it contains a pure virtual function display(). Thus, the objects of the class person cannot be created within the program. If an attempt is made to create the objects of the class person, the compiler reports an error.

Note: Any class inherited from an abstract class remains abstract until each of its pure virtual function is overridden.

Virtual Destructors

A destructor of the derived class is called before the destructor of the base class. However, an exception arises in destructors when the pointer of the base class is made to point to the object of derived class and memory is allocated to the object with the use of new operator. Now, if the memory is de-allocated using the delete operator, only the base class destructor is called and the derived class destructor is not called as the pointer is of base class type. This results in what is known as memory leak, which is defined as the loss of memory access due to the wrong destructor being invoked.

However, by making the base class destructor virtual, both the destructors will be called in the right order. A destructor can be made virtual by prefixing its declaration using the keyword virtual in the base class. Note that the virtual member function and a virtual destructor are different in the sense that if the derived class redefines the function then in the former case, only the derived class version of the function is called whereas in the latter case, both the derived and base class versions of destructor are called.

Note: A destructor can also be declared as pure virtual destructor. However, the body of a pure virtual destructor must be defined otherwise a run-time error is generated.

Example 4.18: A program to demonstrate virtual destructors

```
#include<iostream>
using namespace std;
class base
{
   public:
virtual ~base()     //virtual destructor
{
cout<<"base class destructor called\n";
}
};
class derived : public base
{
   public:
~derived()
```

```
{
cout<<"derived class destructor called\n";
}
};
int main()
{
base *ptr;
ptr=new derived; //memory allocation to derived object
delete ptr;      //memory de-allocation
return 0;
}
```

The output of the program

```
derived class destructor called
base class destructor called
```

In Example 4.18, the destructor of the class base is declared virtual. When the memory allocated to the derived class object is de-allocated using the pointer ptr of base type and the delete operator, the derived class destructor is called before the base class destructor.

4.7.2 Early vs. Late Binding

C++ polymorphism can be achieved either at compile-time or at run-time. At compile-time, polymorphism is implemented using operator overloading and function overloading. However, at run-time, it is implemented using virtual functions.

Function overloading is a way to implement compile-time polymorphism that allows multiple functions to share the same name with different parameters. The compiler identifies the function on the basis of its signature. Operator overloading is the process that enables an operator to exhibit different behaviour, depending on the data provided. These types of compile-time polymorphism are also known as early binding or static binding as the linking of function call to the actual code of the function is done at compile-time itself.

A virtual function is a member function that is declared inside the base class and its functionality can be overridden in the derived classes. A virtual function is accessed using a pointer of base class type and not of derived class type. The compiler selects which version of the function is to be executed depending upon the contents of the pointer. Moreover, this decision is made at the run-time, thereby, employing dynamic binding. Hence, declaring a member function virtual informs the compiler that the function call is resolved at run-time. This type of runtime resolution is known as late binding.

Check Your Progress

9. Define the term free storage list.
10. What is calloc?
11. What are virtual functions?
12. Define the term pure virtual functions.

4.8 ANSWERS TO 'CHECK YOUR PROGRESS'

1. Inheritance is a process of deriving a new class from an already existing class in such a way that the new class inherits all the members of the already existing class. In inheritance, the class which is inherited by the new class is known as base class or superclass or parent.

2. Depending on the access specifiers `public,` `protected` or `private`, a base class can be publicly inherited, protectedly inherited or privately inherited, respectively.

3. Inheritance is implemented while defining a subclass. The name of the superclass is specified in the subclass definition. A subclass can be defined by using `extends` keywords.

4. The derived class owner is inherited from two base classes, namely, person and company.

5. Hierarchical inheritance is a type of inheritance in which more than one class is derived from a single base class. In hierarchical inheritance, a base class provides members that are common to all of its derived classes.

6. According to the user requirement the various types of inheritance, can be combined. This type of inheritance is known as hybrid inheritance.

7. A dynamic data structure is one in which the memory for elements is allocated dynamically at runtime.

8. A pointer is a programming language data type whose value points directly to (or 'Points to') another value stored elsewhere in the computer memory by using its address.

9. As memory is allocated dynamically to a linked list, a new node can be inserted anytime in the list. For this, the memory manager maintains a special linked list known as free storage list or memory bank or free pool that consists of unused memory cells.

10. The `calloc` is defined as `void * calloc (sizen, size_size)`. Here, the number of arguments are two. The first one is the array size and the second one is the bytes occupied per element, i.e., the storage space required for the datatype. The function returns a pointer allocating space for an array of `size sizen`, of datatype `size` but the array contents will be initialized to zero.

11. Virtual function is a member function that is declared inside the base class and its functionality can be overridden in the derived classes. When the base class containing virtual function is inherited, the derived classes may implement their own versions of that function.

12. A virtual function having no definition within the base class is called pure virtual function.

4.9 SUMMARY

- Inheritance is a process of deriving a new class from an already existing class in such a way that the new class inherits all the members of the already existing class. In inheritance, the class which is inherited by the new class is known as base class or superclass or parent.

- Depending on the access specifiers `public`, `protected` or `private`, a base class can be publicly inherited, protectedly inherited or privately inherited, respectively.

- Inheritance is implemented while defining the derived class. The name of the base class appears in the definition of derived class. Since the derived class is nonexistent, when base class is defined the inheritance is implemented only at the time of derived class definition.

- Inheritance is implemented while defining a subclass. The name of the superclass is specified in the subclass definition. A subclass can be defined by using `extends` keywords.

- The derived class owner is inherited from two base classes, namely, person and company.

- When a derived class inherits from more than one base class simultaneously, it is referred to as multiple inheritance. In multiple inheritance, the derived class inherits the members of all its base classes and can directly access the public and the protected members of its base classes.

- Hierarchical inheritance is a type of inheritance in which more than one class is derived from a single base class. In hierarchical inheritance, a base class provides members that are common to all of its derived classes.

- According to the user requirement the various types of inheritance, can be combined. This type of inheritance is known as hybrid inheritance.

- In some hybrid inheritance ambiguity arises when the derived class directly accesses a member of its indirect base class.

- A dynamic data structure is one in which the memory for elements is allocated dynamically at runtime.

- A static variable upholds the same data all through the execution of a program while a dynamic variable can have different values during the course of a program.

- A pointer is a programming language data type whose value points directly to (or 'Points to') another value stored elsewhere in the computer memory by using its address.

- A null pointer is a regular pointer of any type which has a special value that indicates that it is not pointing to any valid reference or memory address. This value is the outcome of type-casting the integer value zero to any pointer type.

- As memory is allocated dynamically to a linked list, a new node can be inserted anytime in the list. For this, the memory manager maintains a special

linked list known as free storage list or memory bank or free pool that consists of unused memory cells.

- Dynamic memory allocation is useful. `malloc` and `calloc` serves to specify the actual dimension at runtime and hence, enable memory allocation dynamically.

- The `calloc` is defined as `void * calloc (sizen, size_size)`. Here, the number of arguments are two. The first one is the array size and the second one is the bytes occupied per element, i.e., the storage space required for the datatype. The function returns a pointer allocating space for an array of `size sizen`, of datatype `size` but the array contents will be initialized to zero.

- When a class has several different function declarations which are specified by single name in the same scope, the functions are said to be overloaded. When the function is called, the correct function is selected by comparing the types of the actual arguments with the types of the formal arguments. This is called function overloading.

- Virtual function is a member function that is declared inside the base class and its functionality can be overridden in the derived classes. When the base class containing virtual function is inherited, the derived classes may implement their own versions of that function.

- A virtual function having no definition within the base class is called pure virtual function.

- A class that contains at least one pure virtual function is known as an abstract class or abstract base class.

- Polymorphism can be achieved either at compile-time or at run-time. At compile-time, polymorphism is implemented using operator overloading and function overloading.

- A virtual function is a member function that is declared inside the base class and its functionality can be overridden in the derived classes. A virtual function is accessed using a pointer of base class type and not of derived class type.

4.10 KEY TERMS

- **Inheritance:** It refers to a process of deriving a new class from an already existing class in such a way that the new class inherits all the members of the already existing class.

- **Multiple inheritance:** When a derived class inherits from more than one base class simultaneously, it is referred to as multiple inheritance.

- **Hierarchical inheritance:** It is a type of inheritance in which more than one class is derived from a single base class.

- **Hybrid inheritance:** According to the user requirement various types of inheritance can be combined. This type of inheritance is known as hybrid inheritance.

- **Dynamic data structure:** It is a data structure in which the memory for elements is allocated dynamically at runtime.

- **Virtual function:** It refer to a member function that is declared inside the base class and its functionality can be overridden in the derived classes.

- **Pure virtual function:** It is a virtual function having no definition within the base class is called pure virtual function.

4.11 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. Defins the term derived class.
2. What do you mean by public inheritance?
3. Differentiate between superclass and subclass.
4. What do you understand by the term parameterised constructor of base class?
5. What is hybrid inheritance?
6. Define the term null pointer.
7. State the concept of dynamic memory allocation.
8. What is polymorphism?
9. Write in brief about abstract classes.
10. Define the term virtual destructors.
11. Differentiate between early and late binding.

Long-Answer Question

1. Explain how inheritance facilitates code reusability.
2. Discuss different types of access specifiers.
3. Describe how the access specifier of base class members can be restored.
4. Explain the multiple inheritances with the help of diagram.
5. Explain how the constructors and destructors are called in a multiple inheritance.
6. Differentiate between hierarchical and hybrid inheritance with the help of diagram.
7. Describe virtual base classes with the help of appropriate examples.
8. Discuss the dynamic data structure and its advantages over a static data structure.
9. Explain memory representation and memory allocation in free storage.
10. Write the program to do string concatenation by using dynamic allocation memory.

11. Explain how the compiler handles calls to virtual functions.

12. Differentiate between virtual and pure virtual functions with the help of example.

4.12 FURTHER READING

Jeyapoovan, T. 2006. *Computer Programming: Theory and Practice* (with CD). New Delhi: Vikas Publishing House.

Khurana, Rohit. 2008. *Object Oriented Programming with C++*. New Delhi: Vikas Publishing House.

Saxena, Sanjay. 2009. *Introduction to Information Technology.* New Delhi: Vikas Publishing House.

Rumbaugh, James, Fedrick Blaha, William Premerlani, and Federick Eddy.1990. *Object- Oriented Modelling and Design.* New Jersey: Prentice Hall.

Balaguruswamy, E. 1998. *Object-Oriented Programming*. New Delhi: Tata McGraw-Hill.

UNIT 5 INPUT-OUTPUT AND FILE HANDLING IN C++

Structure

5.0 INTRODUCTION

In C++, I/O either with the console or other devices, such as disk drive is visualised as an exchange of streams of bytes between the programs and I/O devices. Buffer can be visualized as a fast memory device, which can store bytes of data. The buffer provides for temporary storage of the data. For instance, if a program wants to output to a printer, the entire text is placed on the buffer. The buffer will in turn transfer the characters to the printer via the output stream. It is more important in the case of disc drives since we cannot read or write one character at a time which will cause a lot of overhead.

The formatting functions defined in ios_base are presented in header file `<ios>`. The formatting functions of `istream` and `ostream` are in their respective header files and through inheritance in `<iostream>`. The basic_ios is a virtual base class in the C++ standard library. It provides an interface to all the stream classes and thus provides general properties required of a stream.

The C language introduced library functions to make the job of a programmer easy and interesting. C++ has also incorporated many standard libraries. For instance, strings were treated as an array of characters in the earlier versions of

C++, as in C. Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. String manipulation is one of the most common task of any C++ program. A string is defined as a sequence of characters terminated by null character and can be represented as an array of char type. However, C++ also provides a better alternative for handling string.

In this unit, you will study about the old vs modern in C++, i/o C++ streams, creating inserters and extractors, manipulation functions, classes for file stream operations, opening and closing a file, manipulation of file pointers, random access, command-line arguments, standard library objects, container classes and vectors, list, map and algorithms, string class.

5.1 OBJECTIVES

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

- Understand the old vs modern in C++
- Explain the i/o C++ streams
- Discuss about the creating inserters and extractors, manipulation functions
- Define the classes for file stream operations
- Explain the opening and closing a file
- Understand the manipulations of file pointers
- Analyse the random access
- Define the command-line arguments
- Elaborate on the standard library objects
- Discuss about the container classes and vectors
- Analyse the lists, map and algorithms
- Explain the map in C++ Standard Template Library (STL)
- Define the string class

5.2 OLD VS. MODERN C++

C++ is a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or 'C with Classes'. The language has expanded significantly over time, and modern C++ now has object-oriented, generic, and functional features in addition to facilities for low-level memory manipulation. It is almost always implemented as a compiled language, and many vendors provide C++ compilers, including the Free Software Foundation, LLVM, Microsoft, Intel, Oracle, and IBM, so it is available on many platforms.

C++ was designed with an orientation toward system programming and embedded, resource-constrained software and large systems, with performance, efficiency, and flexibility of use as its design highlights. C++ has also been found useful in many other contexts, with key strengths being software infrastructure

and resource-constrained applications, including desktop applications, video games, servers (for example, e-commerce, web search, or databases), and performance-critical applications (for example, telephone switches or space probes).

Modern C++ is standardized by the International Organization for Standardization (ISO), with the latest standard version ratified and published by ISO in December 2020 as ISO/IEC 14882:2020 (informally known as C++20). The C++ programming language was initially standardized in 1998 as ISO/IEC 14882:1998, which was then amended by the C++03, C++11, C++14, and C++17 standards. The current C++20 standard supersedes these with new features and an enlarged standard library. Before the initial standardization in 1998, C++ was developed by Danish computer scientist Bjarne Stroustrup at Bell Labs since 1979 as an extension of the C language; he wanted an efficient and flexible language similar to C that also provided high-level features for program organization. Since 2012, C++ has been on a three-year release schedule with C++23 as the next planned standard.

After C++98, C++ evolved relatively slowly until, in 2011, the C++11 standard was released, adding numerous new features, enlarging the standard library further, and providing more facilities to C++ programmers. After a minor C++14 update released in December 2014, various new additions were introduced in C++17. On January 3, 2018, Stroustrup was announced as the 2018 winner of the Charles Stark Draper Prize for Engineering, "For conceptualizing and developing the C++ programming language".

After becoming finalized in February 2020, a draft of the C++20 standard was approved on 4 September 2020 and officially published on 15 December 2020.

As of 2021 C++ ranked fourth on the TIOBE index, a measure of the popularity of programming languages, after C, Java, and Python. The TIOBE programming community index is a measure of popularity of programming languages, created and maintained by TIOBE Software BV, based in Eindhoven, the Netherlands. TIOBE stands for The Importance of Being Earnest, the title of an 1895 comedy play by Oscar Wilde.

C++ Core Guidelines: The C++ Core Guidelines are an initiative led by Bjarne Stroustrup, the inventor of C++, and Herb Sutter, the convener and chair of the C++ ISO Working Group, to help programmers write **"Modern C++"** by using best practices for the language standards C++14 and newer, and to help developers of compilers and static checking tools to create rules for catching bad programming practices.

The main aim is to efficiently and consistently write type and resource safe C++.

The Core Guidelines were announced in the opening keynote at CPPC on 2015.

The Guidelines are accompanied by the Guideline Support Library (GSL), a header only library of types and functions to implement the Core Guidelines and static checker tools for enforcing Guideline rules.

5.3   C++ STREAMS

The purpose of a program is to enable communication between the user and a computer. The software developed is used for carrying out desired tasks. It takes input data from the user. The software performs specified tasks using the data supplied. The computer will communicate the results in the user-defined media. Visualize any process carried out using software to realize the importance of Input/ Output (I/O). For instance, in air traffic control system, the input data, such as relative position, speed etc is generated by the aircrafts approaching the station. The software should process the input and give directions for each flight to land at the specified time and location, taking into consideration the input. In this case the output in the form of directions to land may appear on the console. In a departmental stores, the sales person enters the code and number of items sold as the input data through the keyboard. The software makes a bill and displays it in the monitor and prints it out. The external interface is therefore essential for any Information Technology product.

The basic input for any computer system goes from the keyboard and the output is displayed in the monitor. The error messages will be displayed in the console by default. In this chapter, we will discuss about how the C++ programs communicate with the Input / Output devices, such as console (keyboard and video monitor). We will discuss about I/O with Disk drives-files later. The principles, philosophy and methods for I/O are common across devices. Thus C++ stands apart in giving a unique I/O methodology. It is so flexible to device a suitable methodology for an application using the facility provided in C++ standard libraries.

We did take input from keyboard by using `cin>>` and displayed output in monitor using `cout<<`. We could not discuss much of the theory behind them early in the book, since understanding them required advanced concepts, such as multiple inheritance, operator overloading, and virtual functions. Now that we understand these advanced concepts, it will be easy to understand the console input / output methodology. The library functions in C such as `printf, scanf` can still be used in C++ also. But in C++ we would love to use the new methodology since they are much simple to use and at the same time flexible and powerful. In C as well as in C++ we do not have keywords for input/output unlike programming languages such as BASIC or FORTRAN. It should not be considered as a weakness, but as strength. In C we use functions for input / output. In C++, we use classes, objects, functions and overloaded operators for implementing input / output.

C++ Streams I/O

In C++, I/O (Input/Output) either with console or other devices, such as Disk Drive is visualized as exchange of stream of bytes between the programs and I/O devices. The bytes can be digits or characters. When we get input to a program either from keyboard or from disk or from another program we extract stream of bytes. Similarly when a program gives an output, it inserts or sends out a stream of bytes to an output device, such as console monitor, printer, disk drive or another program.

A stream can be considered to be an intermediary for I/O, between the program and I/O devices. Therefore for input we need an intermediary called input stream, which acts as the interface between the program and input device as shown inFigure 5.1(a).

Fig. 5.1(a) *Use of Input Stream*

When we can visualize Input as given in Figure, this can be applied to input from any device such as keyboard, floppy disc drive, hard disc drive or even any other program. Similarly we can associate an output stream for output of a program as shown in Figure 5.1(b).
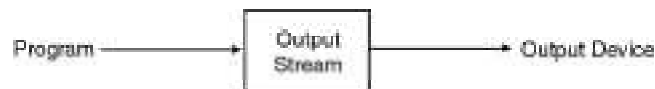

Fig. 5.1(b) *Use of Output Stream*

Here again the program can be any C++ program. The output device can be another program, a printer, disc drive or console monitor. The Input / Output (I/O) streams are implemented in the form of classes and are part of the standard libraries supplied with the C++ language system or IDE. This kind of I/O involves the following:

- Designating an appropriate stream for I/O for the program.
- Linking the I/O device to the stream through the software.

For instance, when the program encounters `cin<< var1`; the keystrokes are received at the input stream. From there it is transferred to the main memory by the program and stored as *var1*. Similarly, when the program encounters `cout<< var2`; the contents of var2 are placed at the output stream by the program and thereafter displayed in the monitor. The keyboard is the default or standard input device and the console monitor is the standard output device. A schematic diagram of I/O between C++ programs and standard I/O devices is given in Figure 5.2.
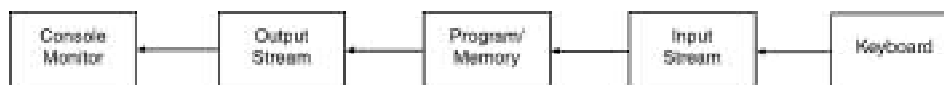
Fig. 5.2 *I/O with Keyboard and Monitor*

The above figure also indicates the conceptual realization of standard I/O with streams.

Buffer

Since the speed of operation of the various devices, such as main memory, keyboard, printer etc. are widely varying, there is a need to incorporate a buffer, another intermediary. Buffer can be visualized as a fast memory device, which can store bytes of data. The buffer provides for temporary storage of the data. For instance, if a program wants to output to a printer, the entire text is placed on the

buffer. The buffer will in turn transfer the characters to the printer via the output stream. It is more important in the case of disc drives since we cannot read or write one character at a time which will cause a lot of overhead. Therefore, the buffer comes handy. In this case, for an input from disc drive, the entire data is transferred to the buffer from the file. Then, depending on the requirements, the necessary bytes are transferred to the program via the input stream. When we want to send text to a disc drive, the data is put on the buffer. Then when the buffer is full or when the entire text has been transferred to the buffer, it is flushed or emptied and written on to the disc. Flushing is basically clearing or emptying the buffer. In C++, the input buffer is flushed when we hit a Return or Enter key. In the case of output buffer, flushing takes place when a new line character is encountered. When we transfer strings line by line to a disc drive, we append new line characters at the end of the line. This is a signal to the buffer to flush it and pass it on to the stream for writing to a file. Now, the buffer is ready to receive another line of text. Therefore, we can insert a buffer in between the stream and the I/O device on either side. Thus, the buffer makes reading and writing of devices of incompatible speeds much easier. So far we have been discussing the concept. Now we will look at the aspects of implementation.

`basic_streambuf` **CLASS**

To implement the buffers a class called `basic_streambuf` is available in the C++ standard library. It allocates memory (in the computer) for creating a buffer. It has also member functions for managing the buffer memory. Managing involves filling the buffer, flushing the buffer and accessing the contents of the buffer. Thus, this class is quite useful to set up a buffer for I/O in C++ programs. Similarly, the streams are implemented by classes.

5.3.1 Stream Classes

The `basic_ios` is a virtual base class in the C++ standard library. It provides an interface to all the stream classes and thus provides general properties required of a stream. The properties include whether it is an input stream or output stream i.e. whether the stream is opened for reading or if it is opened for writing. The basic_ios class also has a pointer to an object of `basic_streambuf` class. Thus it provides the link between the buffer and streams for proper coordination between them. We will discuss more about the stream classes later.

The `basic_ios` class contains many member functions for carrying out input and output. This class is used for input / output operations with all types of devices, such as Disk / file input / output, standard input / output using console etc.

Insertion Operator `<<` and `cout`

We have used the insertion << operator extensively. Let us now see its origin. Actually, it is an overloaded operator. This means there must be an overloaded operator function and should be in a class. Actually it is a member of `basic_ostream` class. The `basic_ostream` class is derived from `basic_ios` class. The `cout` is an object. It directs the output to the standard output stream, which points to the console monitor. The *cout* is an object derived from `basic_ostream` class. The relationship of the operator, object and classes are depicted in Figure 5.3.
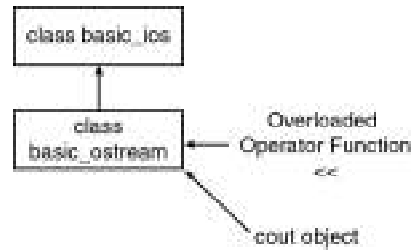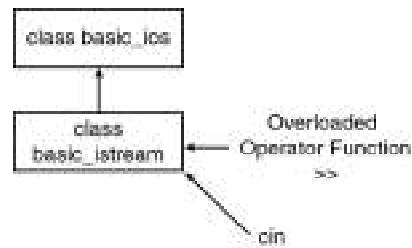
Fig. 5.3 *Origin of << and Cout*

Extraction Operator **>>** and `cin`

The >> operator is also overloaded and the `cin` is an object and they are similarly derived as given in Figure 5.4.

Fig. 5.4 *Origin of >> and Cin*

Thus `basic_ios` is the common base class for both `basic_istream` and `basic_ostream` classes.

The stream classes have a `basic` prefix. The classes are stored in header files in the standard C++ library without prefix as shown in Table 5.1.

Table 5.1 *Stream Classes and Header Files*

Class	Header file
■ basic_ios	■ <ios>
■ basic_istream	■ <istream>
■ basic_ostream	■ <ostream>

We will omit the `basic` prefix for the sake of convenience.

`istream` Class

The `istream` class in the standard library is derived from *ios*. It contains member functions to carry out formatted and unformatted input operations. It contains the overloaded extraction (>>) operator functions. Some of its member functions are:

```
get()
read()
getline()
```
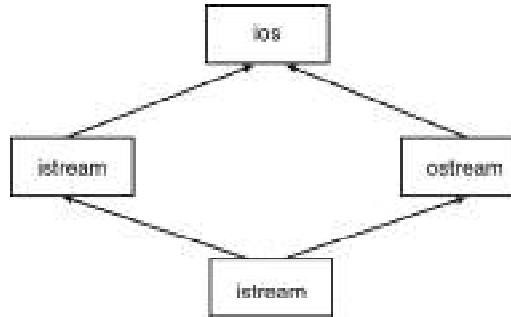
`ostream` Class

The `ostream` class in the standard library implements a mechanism for converting value of any type to a sequence of characters. The `ostream` class contains the overloaded insertion (<<) operator function and also the following member functions:

```
put()
write()
```

iostream Class

The class `iostream` in the standard library is inherited both from `istream` and `ostream` as indicated in Figure 5.5.

Fig. 5.5 *Iostream Class Derivation*

Note that the stream classes have a basic prefix, although they have been omitted in the Figure 5.5 for convenience. Hence the class `iostream` has also a basic prefix and stored in header file `<iostream>`.

Therefore, we could use `cin` and `cout` by including routinely `<iostream>` in our program file. It supports both `cin` and `cout`, assigning them to the standard input device namely the keyboard and standard output device namely the console monitor respectively.

The `basic_iostream` class inherits both from `basic_istream` and `basic_ostream` classes to provide a single interface both for input / output. Thus all the five functions mentioned above are available with it also, due to inheritance. The overloaded operator functions call separate functions in the stream classes for different types of data such as integer, float and character type. It is for this reason that we do not specify the format such as `%d`, `%f`, `%c` etc. as in C. The operators `<<` and `>>` are just symbols. We could cascade them as given below:

```
cout << x << y < "\n";
```

The actual output appears in the same order as specified. We can also cascade different data types as input as given below:

```
float f ;
int in ;
cin >> f >> in;
```

Since depending on data types, separate function in the class will be called, there is no problem when mixing data types as in the above example.

Functions **get** and **put**

The `get` function receives one character at a time. There are two prototypes available in C++ for `get` as given below:

```
get (char *)
get ()
```

Their usage will be clear from the example below:
```
char ch ;
cin.get (ch);
```

In the above, a single character typed on the keyboard will be received and stored in the character variable `ch`.

Let us now implement the `get` function using the other prototype:
```
char ch ;
ch = cin.get();
```

This is the difference in usage of the two prototypes of `get` functions. What is the difference between the `>>` operator and `get` function? We could have written cin >> ch; In such case, the extraction operator will ignore the white spaces and new line characters. But the `get` function will take note of them.

Remember that the `get` function belongs to the `istream` class.

The complement of `get` function for output is the `put` function of the `ostream` class. It also has two forms as given below:
```
 cout.put (var);
```

Here the value of the variable *var* will be displayed in the console monitor. We can also display a specific character directly as given below:
```
cout.put ('a');
```

Here the program will display the given character on the console monitor.

The program below illustrates the use of `get` and `put` function.

Program 5.1
```
#include<iostream>
using namespace std;
int main(){
    char ch;
    int i;
    cout<<"Enter 10 characters with out giving space\n";
    for (i=0; i<10; i++){
        cin.get(ch);
        cout.put(ch);
    }
    cout<<"\n Enter 10 characters with space\n";
    for (i=0; i<10; i++){
        cin.get(ch);
        cout.put(ch);
    }
}
```

In the above program, in the first loop you are asked to enter ten characters without giving space. Therefore after entering the characters and pressing Enter key, the program will reproduce the characters. But in the next loop we enter characters with space. The `get` function recognizes white spaces. Hence after reading the fifth character and the following white space, ten characters (including

white spaces) would have been received. The `for` loop will terminate since it has executed ten times, five time receiving white space and five times the actual character. Hence we can receive only five characters interleaved with five white spaces. The characters entered after that will be ignored. The put function will therefore display five characters with spaces as the result below indicates.

The output of the Program
```
Enter 10 characters with out giving space
aeiouaeiou
aeiouaeiou
Enter 10 characters with space
a e i o u a e i o u
a e i o u
```

Functions `getline` and `write`

C++ supports functions to read and write a line at one go. The `getline()` function will read one line at a time. The end of the line is recognized by a new line character, which is generated by pressing the **Enter** key. We can also specify the size of the line. The prototype of the `getline` function is given below:
```
cin.getline (var, size);
```

When we invoke the above, the system will read a line of characters contained in variable `var` one at a time. The reading will stop when it encounters a new line character or when the required number (size-1) of characters have been read, whichever occurs earlier. The new line character will be received when we enter a line of size less than specified and press the **Enter** key. The **Enter** key or **Return** key generates a new line character. This character will be read by the function but converted into a **NULL** character and appended to the line of characters. Then what is the difference between `getline` and `cin`.

In the case of `cin`, it will treat the white space as the end of the string. Therefore it can read only one word and not a string consisting of more than one word with white spaces in between the words. The program below would help to understand `getline` function.

Program 5.2
```
#include<iostream>
using namespace std;
int main(){
    char ch[15];
    cin.getline(ch,10);
    cout<<"\n"<<ch<<"\n";
}
```

The output of the Program
```
John Joseph
John Jose
```

In the above example, we have declared a C style string *ch* of width 15. We first get the string using the `getline` function. But note that we want to read only

(10-1) 9 characters. Then we display it using `cout`. Therefore `ch` receives 9 characters and NULL is appended as the 10th character. This is confirmed by the result of the program where we display *ch* using *cout*. The name is truncated.

Similarly, the *write* function displays a line of given size. The prototype of the write function is given below:

```
write (var, size) ;
```

where `var` is the name of the string and `size` is an integer.

Program 5.3

```
#include<iostream>
using namespace std;
int main(){
    char ch[15];
    cin.getline(ch, 15);
    cout<<"\n";
    cout.write(ch, 10);
}
```

The output of the Program

```
Tom Peters John
Tom Peters
```

In the example, we receive 15 characters using `getline` function, but write a line of 10 characters. We input 15 characters including (space). The `write` results in truncation since we have specified a line size of 10.

String Stream

The cout is an `ostream` object. The cin is an `istream` object. Thus, we were attaching the streams to the console monitor and keyboard. Similarly, we created objects of `ifstream` and `ofstream` and attached to disk drive for file I/0. C++ supports handling strings just like files. The counterparts of the input/output streams corresponding to strings are as given below:

- **istringstream** for input

- **ostringstream** for output

The equivalent of `fstream` class for string streams is stringstream. It is presented in <`sstream`>. The stringstream inherits both from *istringstream* and *ostringstream*. It may be a bit difficult to visualize attaching strings to stream. The *istringstream* by default is opened for reading. Similarly the *ostringstream* is, by default, opened for writing. The usage of the string streams will be clear from the following example.

Output String Stream

The program below implements declaring and initializing an output string object.

Program 5.4

```
//to demonstrate output stream
#include<sstream>
using namespace std;
```

```
#include<iostream>
int main(){
    //creating ostream object and writing to it
    ostringstream ostr("This is a nice way of creating a
string stream \n");
    cout<<ostr.str();
}
```

Look at how the object is declared and initialized to the constructor. The `ostr` is the object of a class `ostringstream`. This class will be available because we have #included `sstream`. We can initialize the object with any length of characters. The reason is that a string object will expand as needed. Notice that the initial value of a string stream is assigned through the constructor. Now, the `ostringstream` object `ostr` has been created. We would always doubt whether it has been really created. To confirm this, we can read the contents of the object by calling a function `str()` as given in the last statement in the program. In the last statement, we get the contents of `ostr` and assign it to cout object. The result of the program confirms that the program is working fine.

The output of the Program

```
This is a nice way of creating a string stream
```

Input String Stream

The counterpart of output stream is input stream. The corresponding class is `istringstream`. The purpose of the input stream object is to read from a string object. The following program implements input stream.

Program 5.5

```
//To demonstrate input stream
#include<sstream>
using namespace std;
#include<iostream>
int main(){
    string st="creating istream object and reading from
it";
    istringstream istr(st);
    cout<<"We will print the contents of the string stream
at one go\n";
    cout<<istr.str();
 }
```

In the above program, we have declared and assigned values to a string object `st`. Then, we create an object `istr` of `istringstream`. The constructor of the class receives a string object `st`. Thus, the `istringstream` object `istr` is created and initial value assigned in one statement as given below:

```
istringstream istr(st);
```

Now, `istr` will hold the entire string, which was in the string `st`. The contents of `istr` can also be brought out, by calling the function `str()` as given in the last statement of the program.

The output of the Program

```
We will print the contents of the string stream at one go
creating istream object and reading from it
```

Thus, devising string streams is similar to that of file streams. This can be exploited for using the formatting facilities provided in the *ios* class.

The "C" type strings are available in the header file *strstream.h*. For input of C style strings, we can use the *istrstream* class and for output the *ostrstream* class. These can be used to define streams and read and write array of characters in the C style. However, the solution provided in the C++ standard library is more attractive.

Formatted and Unformatted Console

The `get` function receives one character at a time. There are two prototypes available in C++ for `get` as given below:

```
get (char *)
get ()
```

Their usage will be clear from the example below:

```
 char ch ;
 cin.get (ch);
```

In the above, a single character typed on the keyboard will be received and stored in the character variable `ch`.

Let us now implement the `get` function using the other prototype:

```
char ch ;
ch = cin.get();
```

This is the difference in usage of the two prototypes of `get` functions. What is the difference between the `>>` operator and `get` function? We could have written cin >> ch; In such case, the extraction operator will ignore the white spaces and new line characters. But the `get` function will take note of them.

Remember that the `get` function belongs to the `istream` class.

The complement of `get` function for output is the `put` function of the `ostream` class. It also has two forms as given below:

```
cout.put (var);
```

Here the value of the variable *var* will be displayed in the console monitor. We can also display a specific character directly as given below:

```
cout.put ('a');
```

Here the program will display the given character on the console monitor.

The program below illustrates the use of `get` and `put` function.

Program 5.6

```
#include<iostream>
using namespace std;
int main(){
    char ch;
    int i;
```

```
cout<<"Enter 10 characters with out giving space\n";
for (i=0; i<10; i++){
    cin.get(ch);
    cout.put(ch);
}
cout<<"\n Enter 10 characters with space\n";
for (i=0; i<10; i++){
    cin.get(ch);
    cout.put(ch);
}
}
```

In the above program, in the first loop you are asked to enter ten characters without giving space. Therefore after entering the characters and pressing Enter key, the program will reproduce the characters. But in the next loop we enter characters with space. The get function recognizes white spaces. Hence after reading the fifth character and the following white space, ten characters (including white spaces) would have been received. The for loop will terminate since it has executed ten times, five time receiving white space and five times the actual character. Hence we can receive only five characters interleaved with five white spaces. The characters entered after that will be ignored. The put function will therefore display five characters with spaces as the result below indicates.

The output of the Program

```
Enter 10 characters with out giving space
aeiouaeiou
aeiouaeiou
Enter 10 characters with space
a e i o u a e i o u
a e i o u
```

In the above program we have used both get and put functions. Now let us see what happens when we use the operators instead of the functions, for the sake of comparison. Look at the Example below:

Program 5.7

```
#include<iostream>
using namespace std;
int main(){
    char ch;
    int i;
    cout<<"Enter 10 characters with space\n";
    for (i=0; i<10; i++){
        cin>>ch;
        cout.put(ch);
    }
    cout<<"\n Enter 10 characters with space\n";
    for (i=0; i<10; i++){
```

```
        cin.get(ch);
        cout<<ch;
    }
}
```

Here in the first loop we use the >> operator for input. Since the operator will ignore white spaces, only characters will be received without spaces. So the loop will receive all the ten characters in contrast with the previous example. The next put function will display all the ten characters typed, but without space. Note the difference between get function and >> operator. Had we used the get function, we would have received five characters interleaved with five spaces since it cannot ignore white spaces and the loop would have been executed ten times getting five characters and five spaces.

In the next loop, since we use get function, although we entered ten characters, because of the space in between, only five characters and spaces would have been received. This will be displayed as the result of the Example indicates.

The output of the Program

```
Enter 10 characters with space
a e i o u a e i o u
aeiouaeiou
Enter 10 characters with space
a e i o u a e i o u
a e i o u
```

Formatted Console Input/Output

So far we have been printing out without any specific format. C++ does support formatted input and output, which will be discussed briefly in the following paragraphs. There is one more stream class on top of the hierarchy called ios_base. We can say that the class basic_ios is derived from ios_base as indicated in Figure 5.6.
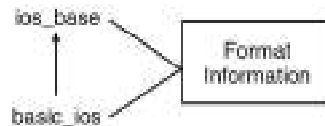
Fig. 5.6 *Relationship between the Top Stream Classes*

Both the classes, ios_base and basic_ios contain functions for formatting. Since classes istream and ostream are derived from class ios, they along with class iostream (derived from them) derive the formatting functions.

Width

The ios class contains a function width(). This is used to define the width of a display. Therefore, it is used in conjunction with the object *cout* as given below:

```
cout << width (10);
```

When such a statement is given, the display following this statement will have a total width of ten characters. Suppose the print statement following it occupies more than the specified number of characters, then C++ will not truncate the display but will accommodate the required width. However, whenever the next statement has to display a variable of size less than specified, then leading spaces will be given. For instance, with the above statement, if the display occupies only seven spaces, then three leading spaces will be left as shown in Figure 5.7.

***Fig. 5.7** Operation of Width Function*

The following example would confirm this.

Program 5.8

```
//To demonstrate width()
#include<iostream>
using namespace std;
int main(){
    int fvar=123141;
    cout.width(8);
    cout<<fvar<<"\n";
    cout.width(5);
    cout<<fvar<<"\n";
}
```

The output of the Program

```
123141
123141
```

In the latter case, since the width required exceeds the specified five characters, the print statement ignores the set width and starts from column 1.

Precision

As we know, the double numbers are represented using double precision and float numbers are represented with single precision. This means, the float numbers will have six digits after the decimal point. The programmer can dictate the number of digits after the decimal point at his will as given below:

```
cout .precision(4);
```

In this case, the display will have only four digits after the decimal point. This is achieved through the function `precision()` of class `ios`. In the case of `width()`, the specification applies only to the statement following it. However, in the case of `precision`, it remains valid for all statements following it, if not reset. We can also combine `width` with `precision`. Then the output will satisfy both the specifications.

5.3.2 Managing Output with Manipulators

The formatting functions defined in `ios_base` are presented in header file `<ios>`. The formatting functions of `istream` and `ostream` are in their respective header

files and through inheritance in `<iostream>`. Additional manipulators are available in the header file called `<iomanip>`. To use these functions, we must include `iomanip` in the program file specifically. For instance, setting the width can be achieved using manipulator as follows:

```
cout << setw (5) << var1;
```

This statement will call the function `setw(int width)`. The `var1` will be displayed with the width of five digits. Of course, if the number of digits is short, there will be leading spaces meaning that the output will be right justified.

Similarly, for setting the precision there is a function in `iomanip` as given below:

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

Using this, we can set the precision of the display that follows as given below:

```
cout << setprecision (5) << var2 ;
```

The `var2` will be printed with a precision of 5 i.e. there will be five digits after the decimal points. Of course, if the trailing digits are zero, they will be skipped.

There is another interesting manipulator called `endl`. This is equivalent of `'\n'`, which is a new line character. Whenever `endl` is encountered, the display will jump to the next line. Let us confirm the concepts learnt through a program.

Program 5.9

```
//To demonstrate setw()
#include<iostream>
using namespace std;
#include<iomanip>
int main(){
    float fvar=256.141;
    cout<<setw(8)<<fvar<<endl;
    cout<<setw(5)<<fvar<<endl;
    cout.setf(ios::fixed, ios::floatfield);
    cout<<setprecision(2)<<fvar<<endl;
}
```

The output of the Program

```
256.141
256.141
256.14
```

Note that there are only two digits after decimal point in the last line since we had set the precision to two. In the above example we had used a function `setf` with two arguments. We will discuss about it in the next section.

The `width()` and `setw()` are identical in operation. Similarly `precision()` and `setprecision()` are also identical. However, the header file `iomanip` is to be included if we want to use `setw()` and `setprecision()`..

Set Flag

In addition, there is another function of class `ios_base` used to specify what is known as flags, which can control the display. The function is `setf()` which stands for set flag.

It is used in conjunction with the `cout` object as given below:

```
cout.setf (argument1, argument2);
```

The argument1 is called `flag` and `argument2` is known as **bit field**. The bit field indicates the group to which the flag belongs.

Using this a number of tasks can be carried out. The tasks are grouped into three categories as given below:

- adjustfield // deals with left / right justification etc.
- floatfield // deals with floating point numbers
- basefield // deals with various types of number representation such as octal, hex etc.

One of these arguments is to be given as `argument2`. Naturally, if we are interested in left justification of the following display, then the `argument2` will be adjustfield. If it is about conversion to octal, then the `argument2` will be basefield.

The `setf` function needs one more argument namely the argument1. If we are interested in left justification, then the `argument1` for that will be `ios::left`. For instance, to display a variable `var3` in the left justified manner and width of 5, we will declare as follows:

```
cout.setf (ios::left, ios::adjustfield);
cout.width (5) ;
cout << var3 ;
```

This is how we use the flags for calling *setf* function. Some of the interesting flags used for formatting are given in Table 5.2.

Table 5.2 Use of Setf Function

Purpose	Argument1	Argument2
Left justified	`ios::left`	`ios::adjustfield`
Right justified	`ios::right`	`ios::adjustfield`
Scientific notation	`ios::scientific`	`ios::floatfield`
Fractional notation	`ios::fixed`	`ios::floatfield`
Octal base	`ios::oct`	`ios::basefield`
Hexadecimal base	`ios::hex`	`ios::basefield`

Let us execute a program to familiarize with the `setf` function.

Program 5.10

```
//To demonstrate flag
#include<iostream>
#include<iomanip>
using namespace std;
int main(){
    int fvar=25614;
    cout<<setw(10)<<fvar<<endl;
```

```
        cout.setf(ios::left, ios::adjustfield);
        cout<<fvar<<endl; //left justified
        cout.setf(ios::right, ios::adjustfield);
        cout<<fvar<<endl; //right justified
        cout.setf(ios::oct, ios::basefield);
        cout<<fvar<<endl; //octal
        cout.setf(ios::hex, ios::basefield);
        cout<<fvar<<endl; //hex
}
```

Take out a paper and write the predicted results before looking at the result of the program given below:

The output of the Program

```
25614
25614
25614
62016
640e
```

Filler Character

We observed that leading white spaces were to be left when the number of characters to be displayed is less than the width. For instance,

```
cout.width (6);
cout << 456;
```

In the above case, three leading spaces will be left. The space can also be filled with a character of our choice as given below:

```
cout.width (6) ;
cout.fill (' * ');
cout << 456;
```

In this case, the leading spaces on the left will be filled with * and the output will appear as follows:

```
*** 456
```

The character used for filling is called **filler** or **paddling character**. The default filler is space.

Bjarne Stroustrup has designed the `width` function to avoid truncation of digits. If the width of the number exceeds the specified width then the complete digits will be printed irrespective of the specified width.

There are some flags with no bit field or second argument. They are given below:

`ios::showbase`	–	base or radix indicator such as octal, hex, decimal on output
`ios::dec`	–	make the conversion to base 10
`ios::showpoint`	–	display trailing decimal point and zeros

ios::uppercase	–	use upper case letters for hexadecimal numbers
ios::skipws	–	skip white spaces while reading input
ios::showpos	–	+ sign will precede if the number is positive

The flags are set using setf function. We can clear the flags by using the function unsetf. We saw that function width (d) of ios class is equivalent of setw (d) of iomanip. Similarly, precision (d) of ios class and setprecision (d) are equivalent. Some more equivalent functions are listed in Table 5.3.

Table 5.3 Equivalent Functions

`<ios> manipulators`	`<iomanip>`
`fill (x)`	`setfill (int x)`
`setf (f)`	`setiosflags (long f)`
`unsetf(f) – clear the flag`	`resetiosflags (long f)`

Example given below would demonstrate some additional functions discussed above.

Program 5.11

```
//To demonstrate additional functions
#include<iostream>
#include<iomanip>
using namespace std;
int main(){
    int var=256;
    cout.fill('$');
    cout<<setw(13)<<var<<endl;
    //converting to oct and displaying base
    cout.setf(ios::oct, ios::basefield);
    cout.setf(ios::showbase);
    cout<<129<<endl;
    //converting to hex and displaying in upper case
    cout.setf(ios::hex, ios::basefield);
    cout.setf(ios::uppercase);
    cout<<7199<<endl;
}
```

The output of the Program

```
$$$$$$$$$$256
0201
0X1C1F
```

Notice that the base of octal number is shown by the prefix of 0 before the number. This will be clear if we compare the result of the previous program where the octal number was displayed without the prefix since we did not ask for the base. The base of hexadecimal number is indicated by 0X and upper case can be realized by the appearance of C and F.

Flexibility of >> and << Operators

In C++ the extraction and insertion operators have been designed to handle every data type. It makes the job of input / output effortless. The features of operators can be summarized as given below:

- Convenient, since there is no need to give the format as in C or other languages

- Efficient, since it converts any data type into characters automatically

- Safe to use, since it does not lead to any exceptional conditions

- Flexible, since a user can overload it to carry out input / output of more complex data types

Now, we will overload both the insertion and extraction operators to carry out input / output of objects. The program below would implement overloading of the operators.

Program 5.12

```
//Overloading of >> and << operators
#include<iostream>
using namespace std;
class fps{
    public:
    int foot;
    int inch;
    friend ostream& operator <<(ostream&, fps& );
    friend istream& operator >>(ostream&, fps& );
};
ostream& operator <<(ostream& A, fps& B){
    A<<"foot = "<< B.foot<<"inch = "<< B.inch;
    return A;
}
istream& operator >>(istream &C, fps& D) {
    cout<<"Enter the length in foot and inch \n";
    C>>D.foot>>D.inch;
    return C;
}
int main()
{
    fps F;
    cin>>F;
    cout<<F;
}
```

In the above program, the operators << and >> are redefined using operator functions. In the main function, we get input to object F of type `fps` as if we would have received a built-in data type. Similarly, we have also used the overloaded operator << to display the object F at one go. The result of the program

confirms that it is possible to overload the insertion and extraction operators to carry out the respective functions on objects as easily and conveniently as they are used for built-in data types.

The output of the Program

```
Enter the length in foot and inch
56 78
foot = 56inch = 78
```

5.3.3 Classes for File Stream Operations

A **file stream** refers to the flow of data between a program and files. Depending on the flow of data from a file or to a file, a stream can be classified into two types (as shown in Figure 5.8).

- **Input Stream**: It reads the data from the file and supplies it to the program.
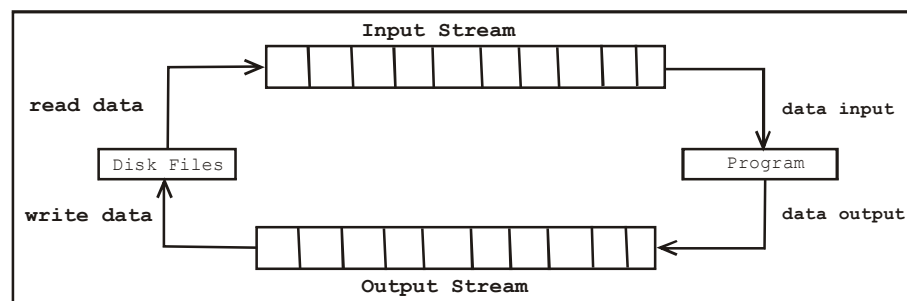- **Output Stream**: It receives data from the program and writes it to the file.

Fig. 5.8 *File Input and Output Streams*

There are two modes by which data can be transferred to and from streams: **text mode** and **binary mode**. A file, which is opened in the text mode, is known as **text file,** whereas a file opened in the binary mode is known as **binary file**. The main aspects where a text file and a binary file differ from one another are as follows:

- **Storage of Numbers:** A text file stores data as a sequence of characters while the binary file stores data as a sequence of bytes. For example, if a number, say 12345, is stored in the text format, it occupies five bytes of memory. However, if it is stored in the binary format, it takes two bytes of memory (as shown in Figure 5.9). Hence, binary files occupy less memory. Moreover, since data is stored in the same format as in the internal memory, saving and accessing the data from binary files is faster than text files.
- **Handling Newline Character:** In text files, some character translations take place while data is being read from or written to the file. For example, newline character (\n) is expanded into carriage return/line feed combination before being written to the disk. Hence, there is a possibility that the number of characters written (or read) may not be the same as that in the file. While in binary files, no such character translation takes place. Thus, the number of bytes read (or written) is same as that in the file.

- **Representation of End of File:** In text files, a special character whose ASCII value is 26 represents the end of file. While in binary files, there is no such special character to detect the end of file. The end of file is determined by keeping track of the number of characters present in the file.
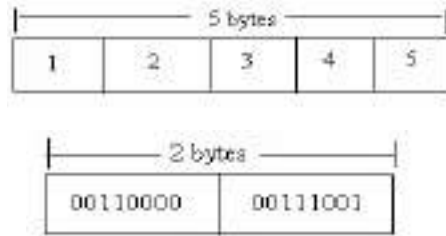
Fig. 5.9 *Representation of a Number in Character and Binary Formats*

Classes for File Stream

C++ provides different streams to represent different kinds of data flow. Each stream is associated with a particular class, which contains the member functions for reading from or writing to the devices. For example, the istream class, which is derived from ios, provides members like extraction operator (>>), get(), etc. on console for performing the input-specific operations. The classes used for the console I/O operations are declared in the header <iostream>.

However, the classes specific to the disk file I/O operations are known as **file stream classes** and declared in the header <fstream>. The header <fstream> defines several classes, including ifstream, ofstream and fstream, which are used for working with files. These classes are derived from istream, ostream and iostream, respectively, which in turn are derived from ios class (Refer Figure 5.10). Thus, the file-specific classes can have access to all the member functions of ios. In addition, the file-specific classes are also derived from fstreambase class. The fstreambase class contains the object of class filebuf, which also inherits member functions of the class streambuf.

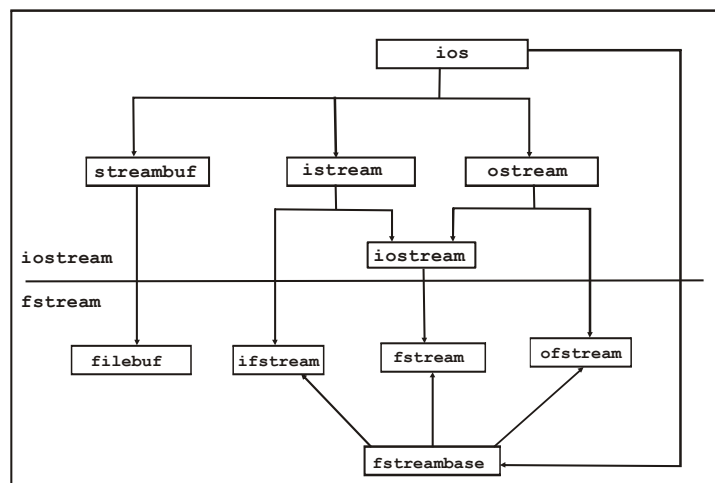***Fig. 5.10*** *C++ Stream Class Hierarchy*

The description of all file-specific classes is as follows:

- `ifstream`**:** The `ifstream` is an input file stream class, which provides functions for performing the reading operations only. It contains functions, such as `get()`, `getline()`, `read()`, `seekg()` and `tellg()`, which are derived from the `istream` class.

- `ofstream`**:** The `ofstream` is an output file stream class, which provides functions for performing the writing operations only. It contains functions, such as `put()`, `write()`, `seekp()` and `tellp()`, which are derived from the `ostream` class.

- `fstream`**:** The `fstream` is an I/O file stream class and hence, provides functions for performing the input as well as output operations. It contains all the functions of `istream` and `ostream` classes, which are inherited through the `iostream` class.

- `filebuf`**:** The `filebuf` class is used to manage the buffered I/O of file stream. It contains `open()` and `close()` functions.

- `Fstreambase`**:** The `fstreambase` class serves as a base class for `ifstream`, `ofstream` and `fstream` classes. It contains `open()` and `close()` functions and also other operations, which are common to all file streams.

5.3.4 Opening and Closing a File

To perform any operation on a file, it needs to be opened first. A file is opened by linking it to a stream. Thus, for opening a file, an object of the particular stream class is created first and then associated with the file. The stream class to be used for creating the stream object depends upon the type of operations to be performed on the file. For example, to read data from a file, an object of `ifstream` is required, to write data to a file, an object of `ofstream` is required and to perform reading and writing to a file, an object of `fstream` is required. A file can be opened in two ways, that is either by using the constructor or the member function `open()` of the stream class.

Note: In C++, each file is an object of a particular stream class.

Opening a File Using a Constructor

When a file is opened using the constructor of a stream class, the file name is passed to it as an argument. Thus, the constructor initialises the stream object with the file name passed to it.

The syntax for opening a file using the constructor is:

```
stream_class stream_object("file_name");
```

where,

`stream_class` = the name of the stream class whose object is to be created

`stream_object` = the object name which can be any valid identifier

`file_name` = the name of the file to be opened

For example, a file named "`marks`" can be opened for input using constructor as shown in the statement:

```
ifstream infile("marks"); //for input only
```

In this statement, an object `infile` of the `ifstream` class (input file stream) is created and initialised with the file `marks`. Thus, the object `infile` can be used only to read data from the file `marks`.

Similarly, a file named "`result`" can be opened for output using constructor as shown:

```
ifstream outfile("result"); //for output only
```

A single file can also be opened for both reading and writing with the help of the statement:

```
fstream iofile("student");  //for both input and output
```

Note: When a file is opened as an output file, the operating system either creates a new file (if the file does not exist) or overwrites the contents of the file (if the file already exists).

The constructor of a stream class can link only a single file to a stream. Thus, in a program using multiple files, a separate stream for each file is required. For example, consider the statements:

```
ofstream out1("file1");
ofstream out1("file2");
//error: Multiple declarations for 'out1'
ofstream out2("file2");
```

Hence, opening file using constructor is useful when only one file is to be used with the stream. However, to manage multiple files using one stream (as required in case of sequential processing of files), the `open()` method is used.

Opening a File Using the `open()` Function

The function `open()` which is a member function of the `fstreambase` class has the same parameter as the constructor of stream class. However, unlike constructors, the creation of stream object and its linking with a particular file using `open()` function are performed in separate statements.

The syntax for opening a file using function `open()` is:

```
stream_class stream_object;
stream_object.open("file_name");
```

For example, consider the statements:

```
ifstream infile;      //create input stream
infile.open("marks"); //connect stream to file marks
ofstream outfile;      //create output stream
outfile.open("result"); //connect stream to file result

fstream iofile;        //create input/output stream
iofile.open("student"); //connect stream to file student
```

A stream object, created once using the `open()` function, can be linked repeatedly with different files. Hence, it helps in managing multiple files without the overhead of creating a new stream object each time.

Sometimes, opening a file (either using constructor or `open()` method) fails due to some reason. In that case, the value of the associated stream evaluates to false.

Thus, before performing any operation on a file, it is necessary to know whether the file is successfully opened or not. This can be accomplished by using the function `is_open()`, which is a member function of the classes `fstream`, `ifstream` and `ofstream`. The prototype of `is_open()` function is:

```
bool is_open();
```

This function checks the value of the associated stream and returns true if the stream is linked to open file, otherwise it returns false.

File Modes

The file has been opened so far by specifying only one argument (that is, file name) in both the constructor and open()function. However, another argument can also be included in both the functions that specifies the file mode. The file mode describes how a file is to be used, that is, to read from it, to write to it, to append to it and so on.

The general form of `open()` function with two arguments is:

```
stream_object.open("file_name", file_mode);
```

where,

`file_mode` = an enumeration defined in the `ios` class that specifies how the file is opened.

The various modes that a file can have are listed in Table 5.4.

Table 5.4 File Modes for Opening a File

Mode	Description
`ios::in`	opens a file for reading
`ios::out`	opens a file for writing
`ios::trunk`	deletes the contents of the file if it already exists, that is, truncates the file to zero length
`ios::app`	appends the data at the end of the file only
`ios::ate`	moves to the end of file on opening, however permits addition as well as modification of data anywhere in the file
`ios::binary`	opens the file in binary mode
`ios::nocreate`	open fails if the file does not exist—does not create a new file
`ios::noreplace`	open fails if the file already exists—does not replace the existing file with a new one

Two or more file modes can be combined using the bitwise OR operator (`|`) between them as shown:

```
outfile.open("example", ios::app | ios::nocreate)
```

where,

`outfile` = an object of `ofstream` class.

The prototype of both the constructor and `open()` function contain default mode for each type of stream. The default mode for the `ifstream` class is `ios::in`, for `ofstream` class is `ios::out`, and for `fstream` class `ios::in|ios::out`. Thus, in the absence of this argument, the default value is provided. For example, consider the statements:

Equivalent statements
```
infile.open("marks");
infile.open("marks", ios::in);
```

where,

`infile` = an object of `ifstream` class.

Note: Depending on the compiler, the `open()` function may or may not provide the default mode `ios::in|ios::out` for `fstream` class. Hence, it needs to be specified explicitly.

Some points must always be kept in mind while opening a file, which are as follows:

- A file name can include a path specifier. However, if the path is not specified, the compiler assumes the file to be present in the current directory.

- Opening a file in the `ios::out` mode also opens it in the `ios::trunc` mode. Hence, an object of the `ofstream` class is directly opened in the `ios::out|ios::trunc` mode by default.

- In both `ios::app` and `ios::ate` modes, a file is created by the specified name, if it does not exist.

- The mode `ios::app` can be used with the files capable for output only.

- By default, all the files are opened in text mode. Hence, to open a file in binary mode, the file mode `ios::binary` must be specified as shown:
  ```
  ofile.open("account", ios::app|ios::binary);
  ```

Closing a File

When a stream object goes out of scope, the destructor of the object is invoked implicitly, which closes the associated file automatically. However, a file can also be closed explicitly by using the function `close(`, which is a member function of the `fstreambase` class. The `close()` function takes no parameter and returns no value.

The syntax for explicitly closing a file is:
```
stream_object.close();
```

For example, a stream object, say `iofile`, can be closed with the help of the statement:
```
iofile.close();  //close connection
```

Note that closing a file only disconnects the file from the stream that is linked to it. However, the stream object still exists and can be used again for associating with the same or another file. For example, consider the statements:
```
ifstream ifile;  //create input stream
ifile.open("student_detail.txt");
//connect stream to student_detail.txt
ifile.close(); //disconnect    stream    from
student_detail.txt
ifile.open("result.dat");
//connect same stream to result.dat
ifile.close();  //disconnect stream from result.dat
```
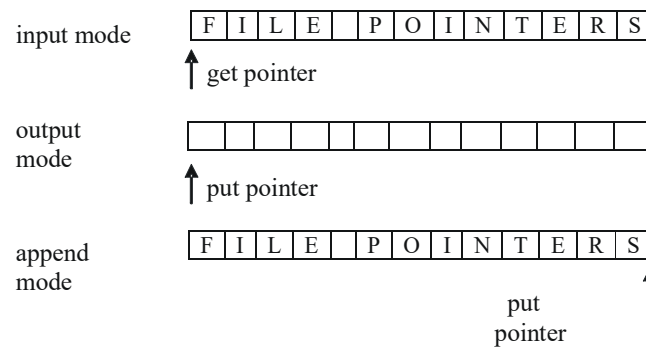
Note: If a single stream is used with multiple files using `open()` function, the file needs to be closed explicitly.

5.3.5 Manipulations of File Pointers

In C++, every file is associated with two file pointers, namely, get pointer (input pointer) and put pointer (output pointer). These file pointers are not the usual C++ pointers as the name implies. They simply represent integer values that specify the current position (byte number) in the file from where writing and reading will start.

The get pointer specifies the position from where the next read operation will start and the put pointer specifies the position from where the next write operation will start. When the file is opened in input mode, the get pointer is placed at the beginning of the file. Similarly, when the file is opened in the output mode, the existing data is deleted and the put pointer is placed at the beginning of the file. However, if an existing file is opened in append mode using `ios::app` mode specifier, the put pointer is moved to the end of the file so that writing can start after the end of existing data. The default position of the file pointers is shown in the Figure 5.11.

input mode | F | I | L | E | | P | O | I | N | T | E | R | S |
↑ get pointer

output
mode
↑ put pointer

append
mode | F | I | L | E | | P | O | I | N | T | E | R | S |
                       put
                      pointer

Fig. 5.11 *Default Position of File Pointers*

Note that each time a read or write operation is performed, the appropriate pointer is automatically advanced sequentially in the file. However, in some cases like adding a record between existing records or modifying the existing records, a file is required to be read from or written to any other position rather than its default position. In such cases, the file pointers must be moved to the desired location explicitly by the programmer. This section discusses how file pointers are manipulated to move to the desired position in a file.

Specifying the Position

The manipulation of the pointers can be accomplished through the functions namely, `tellg()`, `tellp()`, `seekg()` and `seekp()` provided by the file stream classes. The functions `tellg()` and `seekg()` belong to the `ifstream` class whereas the functions `tellp()` and `seekp()` belong to the `ofstream` class. The description of these functions is given in Table 5.5.

Table 5.5 *Functions to Manipulate File Pointers*

Functions	Prototype	Purpose
`tellg()`	`pos_type tellg()`	returns the current position of the get pointer
`tellp()`	`pos_type tellp()`	returns the current position of the put pointer
`seekg()`	`ifstream& seekg(pos_type pos)`	moves the get pointer to the specified location `pos` from the beginning of the file where `pos` is a positive integer value
`seekp()`	`ofstream &seekp(pos_type pos)`	moves the put pointer to the specified location `pos` from the beginning of the file where `pos` is a positive integer value

where,

`pos_type` = a type defined by `ios`, which is capable of holding the largest value that can be returned by any of these functions.

To understand the concept of the function `tellp()`, consider the statements:

```
ofstream outfile("myfile",ios::app);
int current_pos = outfile.tellp();
```

Here, the first statement moves the put pointer to the end of the file as `myfile` is opened in append mode. The second statement returns the number of bytes in the file.

To understand the concept of the function `seekg()`, consider the: statements:

```
ifstream infile("myfile");
infile.seekg(5);
```

The second statement sets the get pointer at the byte number 5, that is, the 6th byte in the file. Note that the bytes in a file are numbered from zero as shown in Figure 5.12.
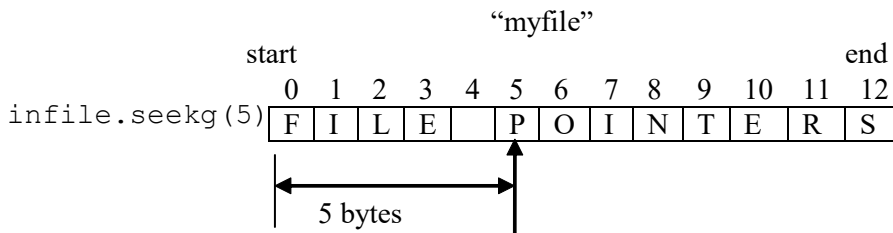
Fig. 5.12 *Use of* `seekg( )`

Specifying the Offset

As discussed earlier, the get pointer can be moved to the desired position by specifying the positive value of `pos_type` in the function `seekg()` as an argument. This positive value represents the absolute position in the file. However, the get pointer can also be moved to the desired position using an offset value (can be positive or negative) relative to one of the three specific positions in the file. These three positions which are defined in the `ios` class are mentioned in Table 5.6.

Table 5.6 *Relative Positions*

Value	Description
`ios::beg`	offset is relative to the first character in the file or the beginning of the file
`ios::cur`	offset is relative to the current position of file pointer
`ios::end`	offset is relative to the last character in the file or end of the file

The offset value relative to the positions (mentioned in Table 5.6) can be set by using the overloaded version of `seekg()` or `seekp()` that accepts two arguments as shown in the prototypes:

```
ifstream &seekg(off_type offset, seek_dir refposition);
ofstream &seekp(off_type offset, seek_dir refposition);
```

where,

`off_type` = integer type defined by `ios`

`seek_dir` = enumeration defined by `ios` that determines how seek takes place

Thus, the functions `seekg()` and `seekp()` move the get pointer and put pointer respectively by `offset` number of bytes from the position specified by the parameter `refposition`. The `refposition` must be one of the following positions as mentioned in Table 5.6.

Various examples of the calls to the overloaded forms of the seekg()function with an object `iofile` of class `fstream` are listed in Table 5.7.

Table 5.7 *Examples of Seek Calls Using Offset*

seek calls	Output
`iofile.seekg(n,ios::beg);`	moves the get pointer to (n+1)th byte in the file
`iofile.seekg(0,ios::beg);`	sets the get pointer at the beginning of the file
`iofile.seekg(2,ios::cur);`	moves the get pointer forward by 2 bytes from the current position
`iofile.seekg(0,ios::cur);`	get pointer remains at current position
`iofile.seekg(0,ios::end);`	sets the get pointer at the end of the file
`iofile.seekg(-5,ios::end);`	moves the get pointer backward by 5 bytes from the end of file
`iofile.seekg(3,ios::cur);`	moves the get pointer forward by 3 bytes from the current position.

Note that the function `seekg()` returns a reference to the file stream object. Thus, it can be combined with an input operation by using an extraction operator as shown:

```
file.seekg(5,ios::beg)>>num;
```

This statement moves the get pointer to byte number 5 from the beginning of the file, and reads data from that position into `num` where, `num` is a variable of any data type depending upon the contents of the file.

Similarly, the function `seekp()` can be used to move the put pointer where the next write operation is to be performed. However, the write operation will overwrite the data starting from that position.

Note: The offset value in `seekg()` and `seekp()` must be positive relative to `ios::beg` and negative relative to `ios::end`.

5.3.6 Random Access

Randomly accessing a file means directly reaching the desired record/object in the file. To access an object in a file directly, the put pointer is placed at the beginning of the object by skipping `(object_no-1)*object_size` number of bytes from the beginning of the file. Once the put pointer is placed at the appropriate object number, the object can be modified. Note that an object can be accessed directly only if the file consists of similar objects, that is, objects of equal size. The size of each object can be determined using the `sizeof` operator shown as follows:.

```
int  object_size=sizeof(object);
```

Once, the size of an object is determined, the position of the `nth` object can be obtained using the following statement:

```
int position = (n-1)* object_size;
```

where, `position` provides the byte number of the first byte of the nth object and the file pointer can be moved to this position (byte number) by using functions `seekg()` or `seekp()` as discussed earlier.

The total number of objects in a file can also be obtained by using the following statement:

```
int num_objects = file_size/object_size;
```

where, `file_size` can be determined by using the functions `tellg()` or `tellp()` when the file pointer is located at the end of the file.

Note: Random-access I/O should be performed only on the files opened for the binary operations to get the correct output.

Example 5.1: A menu-driven program to demonstrate the update of a file Employee.Dat containing name, id and salary of employees using random access

```
#include<fstream>
#include<conio>
using namespace std;
class Employee
{
int Emp_No;
char Name[30];
float salary;
public:
void getdata();
void putdata();
int getEno()
{
```

```
            return Emp_No;
}
};
void Employee :: getdata()
{
cout<<"Emp_No: ";
cin>>Emp_No;
cout<<"E_Name: ";
cin>>Name;
cout<<"Salary: ";
cin>>salary;
}
void Employee :: putdata()
{
cout<<Emp_No;
cout<<"\t"<<Name;
cout<<"\t\t "<<salary<<endl;
}
void Add();
void Show();
void Modify();
int main()
{
int choice;
do
{
clrscr();
cout<<"\nMain Menu";
cout<<"\n1. Add new record";
cout<<"\n2. Show records";
cout<<"\n3. Modify a record";
cout<<"\n4. Exit";
cout<<"\nEnter your choice . . . ";
cin>>choice;
switch(choice)
{
case 1: Add();
break;
case 2: Show();
break;
case 3: Modify();
break;
case 4: cout<<"\nTerminating the program.";
break;
```

```
default:cout<<"\nEnter No. from 1 to 4";
}
getch();
}while(choice!=4);
return 0;
}
void Add()
{
ofstream iofile;
iofile.open("Employee.DAT",ios::app|ios::binary );
Employee Emp;
cout<<"\nEnter the data to be appended : "<<"\n";
    Emp.getdata();
    iofile.write((char *)&Emp, sizeof(Emp));
    cout<<"\nThe record is added in the file";
}


void Show()
{
    ifstream iofile;
    iofile.open("Employee.DAT", ios::in|ios::binary );
    Employee Emp;
    ofile.seekg(0);
    out<<"Records Of Employees.......\n";
    while(iofile.read((char *)&Emp, sizeof(Emp)))
    {
       Emp.putdata();
    }
}
void Modify()
{
fstream iofile;
iofile.open("Employee.DAT",
ios::in|ios::out|ios::ate|ios::binary);
Employee Emp;
int total_bytes = iofile.tellg();
int num_records = total_bytes/sizeof(Emp);
cout<<" Total number of records = "
<<num_records<<endl<<endl;
cout<<"Enter record no. to be updated: " ;
int recordno;
cin>>recordno;
long pos=(recordno-1)*sizeof(Emp);
iofile.seekp(pos);
```

```
cout<<"Enter new values for this record\n";
Emp.getdata();
iofile.write((char*) &Emp, sizeof(Emp));
cout<<"Record is modified\n";
}
```

The output of the program

```
Main Menu
1. Add new record
2. Show records
3. Modify a record
4. Exit
Enter your choice . . . 1
Enter the data to be appended:
Emp_No: 230
E_Name: John
Salary: 34000
The record is added in the file

Main Menu
1. Add new record
2. Show records
3. Modify a record
4. Exit
Enter your choice . . . 1
Enter the data to be appended:
Emp_No: 123
E_Name: Smith
Salary: 34568
The record is added in the file

Main Menu
1. Add new record
2. Show records
3. Modify a record
4. Exit
Enter your choice . . . 1
Enter the data to be appended:
Emp_No: 345
E_Name: Mary
Salary: 35678
```

```
The record is added in the file

Main Menu
1. Add new record
2. Show records
3. Modify a record
4. Exit
Enter your choice . . . 2
Records Of Employees.......
230      John            34000
123      Smith           34568
345      Mary            35678

Main Menu
1. Add new record
2. Show records
3. Modify a record
4. Exit
Enter your choice . . . 3
Total number of records = 3
Enter record no. to be updated
3
Enter new values for this record
Emp_No: 348
E_Name: Peter
Salary: 5678
Record is modified

Main Menu
1. Add new record
2. Show records
3. Modify a record
4. Exit
Enter your choice . . . 2
Records Of Employees.......
230      John            34000
123      Smith           34568
348      Peter           5678
```

In this example, a new object of type employee can be added, an existing object can be modified or detail of all the employees can be displayed depending on the choice entered by the user.

To add a new object, the file `Employee.DAT` is opened using `ios::app` mode, which sets the file pointer to the end of the file.

To display the existing objects, the pointer is set to the beginning of the file using the statement `seekg(0)`. The total number of objects is also determined using the statement `num_records = total_bytes/sizeof(Emp)` and displayed.

To modify the data of an existing object, the file `Employee.DAT` is opened using `ios::ate` mode, which allows to write data anywhere in the file. The file pointer is set to the first byte of record to be modified using statements `long pos=(recordno-1)*sizeof(Emp)` and `iofile.seekp(pos)`.

Check Your Progress

1. Define the term streams.
2. What is istream class?
3. State about the formatting function.
4. What is set flag?
5. Name the various modes by which data can be transferred to and from streams.


5.4 STANDARD LIBRARY OBJECTS

The language C amply demonstrated that no program is written just using the 24 keywords of the programming language. Always C programs used some library functions, such as `<stdio.h>`. The C library functions are available for C++ programs in the global namespace with dot h suffix.

The C++ standard library classes, objects and functions are grouped under the namespace called `std`. There are interesting classes in the C++ standard library under the `namespace std`. Some of the special classes are:

- string
- stack
- vector
- list
- map

They are part of every standard C++ implementation.

The standard library of C++ is contained in a separate namespace called `std`. The user-developed header files will be in their respective namespaces. The C standard headers are in the global namespace. Any other code, which has not been specifically grouped under a namespace, will also be in the global namespace as depicted in Figure 5.13.
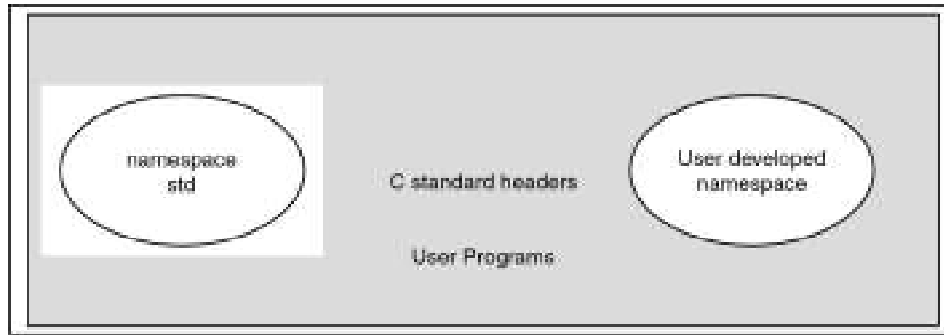
Fig. 5.13 *Contents of Global Namespace in C++*

To illustrate the ease of use of standard library in programs, the following standard libraries will be discussed:

- string
- vector

Difference between Character Array and C++ String

C treats an array of characters as a string and is thus declared as:

```
char array[3] ={'g', 'o', 'd'};
```

When you declare a string as a character array, you may also write it as follows:

```
char array[3] ="god";
```

The system appends a NULL at the end of the string. Still the size of this array is 3. But in C++, the programmer has to allocate the space for NULL. Hence the above string requires 4 spaces for storing an array of 3 characters. Therefore in C++, character arrays can only accept string literals. That means they cannot be modified. For instance, you may have the following type of declarations:

```
char str[]= "Swamy";
char * str= "Swamy";
```

The string on the RHS will be treated as a string literal incapable of any modification due to the difference in handling NULL in C and C++. This restriction has been imposed to enable portability of the old C code to C++. A string in C++ is very easy to be defined and used with the help of the standard library. You do not even have to specify the length of the string. But when you declare a string as a character array, you need to keep in mind its treatment in C++.

Strings

The class called string is available in the standard library in std namespace. A class is a user-defined type. Therefore, in order to use the special functions pertaining to string, you may declare "using namespace std;" on top of the programs. This will also allow you to use the i/o objects such as cout and cin. In case the statement "using namespace std;" is not added, the objects and functions of the library have to be prefixed with std as:

```
std::cout
std::cin
```

Either of the two methods is permissible.

Program 5.13 illustrates a string from the keyboard and display it on the console monitor.

Program 5.13

```
//use of strings
#include<iostream>
using namespace std;
int main(){
    string str1;
    cout<<"Enter your name\n";
    cin>>str1;
    cout<<"Your name is: "<<str1;
}
```

Here string `str1` has been declared in a simple manner. The string is a type defined in the standard library, and `str1` is a variable of the type string. In OOP terminology, `str1` is an object of class string. But for the availability of the standard library, you should have declared string as an array of characters. This declaration does not put any restriction on the number of characters in the string. The `str1` is connected to the standard input device, namely, the keyboard in a simple manner as given below:

```
cin>>str1;
```

The output of the Program

```
Enter your name
Om Venkata chala pathaye namaha
Your name is: Om
```

You will find that the full name typed has not been reproduced. You may try another way of assigning a long string and displaying it on the console as given in Program 5.14.

Program 5.14

```
//use of strings
#include<iostream>
using namespace std;
int main(){
    string str1;
    str1="Om Venkatachala Pathaye Namaha";
    cout<<"Your name is: "<<str1;
}
```

In this example, a string has been assigned to `str1` in the program itself.

The output of the Program

```
Your name is: Om Venkatachala Pathaye Namaha
```

This program displays the complete name. Then, why was it not possible to reproduce the full name when it was received from the console? What has happened

is that the system assumed the white space after "Om" as end of input. Therefore, it did not look for the rest of the characters. To avoid this happening, you may get the full line as shown in program 5.15.

Program 5.15

```
//use of strings
#include<iostream>
using namespace std;
int main(){
    string str1;
    cout<<"Enter your name\n";
    getline (cin,str1);
    cout<<"Your name is: "<<str1;
}
```

The above example has accomplished the task of getting one line at a time through the statement `getline (cin,str1);`

The `getline` function has two arguments. The first one is the object `cin` and the second is the string `str1`.

The output of the Program

```
Enter your name
Om Venkatachala Pathaye namaha
Your name is: Om Venkatachala Pathaye namaha
```

Concatenation of Strings

With the help of the standard library, you may concatenate or add strings very easily by simply using the + sign as the Program 5.16 indicates.

Program 5.16

```
//concatenation of strings
#include<iostream>
using namespace std;
int main(){
    string str1, str2, str3;
    str1="Vetri ";
    str2="Vel";
    str3=str1+str2;
    cout<<"concatenated string is: "<<str3;
}
```

Of the three strings declared in the beginning of the main function, the first two, namely, `str1` and `str2` are initialized with strings. The following statement, using the + sign, concatenates the 2 strings.

```
    str3=str1+str2;
```

The output of the Program

```
concatenated string is: Vetri Vel
```

Empty string

You may check whether a string is empty or not by using a built-in boolean function called `empty()`. Program 5.17 checks whether a string is empty and, if not, finds out the length of the string.

Program 5.17

```
//finding string empty and length
#include<iostream>
using namespace std;
int main(){
    string str1="Ganesh";
    if (str1.empty())
    cout<<"empty \n";
    else
    cout<<"length of string is: "<<str1.length();
}
```

The functions `empty()` and `length()` are to be used in conjunction with a string object which is nothing but a string variable. In Program 5.17, you may have declared a string variable "`str1`". Then you may check whether it is empty. If it is so, you may print "`empty`", If otherwise the length of the string is found and displayed by the last statement in the program.

The output of the Program

```
length of string is: 6
```

String comparison

Just as numbers can be compared, you can compare strings by using the sign "==". Program 5.18 compares strings.

Program 5.18

```
//String Comparison
#include<iostream>
using namespace std;
int main(){
    string str1="Ganesh";
    string str2="Vinayaga";
    string str3="Ganesh";
    if (str1==str2)
    cout<< "strings 1 and 2 are same";
    else
    if(str1==str3)
    cout<<"strings 1 and 3 are same";
}
```

In Program 5.18, three string variables have been declared and values assigned to them. You compare the strings `str1` and `str2` first and then `str1` and `str3`.

The output of the Program

```
strings 1 and 3 are same
```

5.4.1 Vector of Operations

A class or a structure may be created only to hold data. Such classes or structures are called containers. A container object holds other objects or data. C++ provides this facility. A vector is one such popular container. A vector is a class used to represent multi-dimensional arrays easily.

It is important to understand the fundamental uses of the vector class provided in the header file <vector> in the standard C++ library and housed in the namespace std. For this purpose, you can now demonstrate usage of vector as a one-dimensional array. You need to make the following declaration of the library-supplied vector as a one-dimensional array.

```
vector <double>array(4);
```

Here vector is the name of the class in the standard library, double is the data type and array is an object of a vector class. The number enclosed within parentheses indicates the size of the array. This creates a vector object with four elements. Each element in this case is of double data type . On the contrary, if square brackets, as given below are used, it will refer to four empty vector objects.

```
vector <double>array[4];
```

This will mean that four vectors objects are created. Since your purpose is to create one vector object with four elements, the first declaration with parentheses is appropriate.

In order to use the vector class, you should use "#include <vector>" in the program. Now look at the Program 5.19.

Program 5.19

```
//To demonstrate vectors
#include<iostream>
#include<vector>
using namespace std;
int main(){
    vector <double>array(4);
    double sum=0.0;
    cout<<"Enter 4 real numbers \n";
    for(int i=0; i<4; i++){
    cin>>array[i];
    sum+=array[i];
    }
    double average=sum/4;
    cout<<"Average of the 4 numbers typed="<<average;
}
```

In Program 5.19, an object "array" of vector has been declared. A general form of declaration of a vector object is given below:

```
vector <datatype>name;
```

In the main function, you can get four real numbers through the following statement.

```
cin>>array[i];
```

The individual elements of the vector object are given within square brackets. Add all the four elements, find the average and print the same.

The output of the Program

```
Enter 4 real numbers
10.0 20.0 30.0 40.0
Average of the 4 numbers typed=25
```

In Program 5.19, a new container of type vector has been declared. This vector holds a one-dimensional array. Vector is more convenient than the arrays because if the array size is not declared correctly, then you may get into problems with arrays. However, the vector of the standard library takes care of such problems.

Storing Strings in Vector Object: The following example string is used as the type of element. Since you are familiar with the usage of strings, you can modify the Program 5.19 to declare a vector object of type string. In this program, you may also sort the strings using bubble sort and display the sorted strings. Look at Program 5.20.

Program 5.20

```
//To demonstrate sorting strings using vectors
#include<iostream>
#include<vector>
using namespace std;
int main(){
    int i=0, j=0;
    vector <string>array(4);
    cout<<"Enter 4 strings \n";
    for(i=0; i<4; i++)
        cin>>array[i];
    string temp;
    for(i=0; i<3; i++)
    for(j=i+1; j<4; j++)    {
        int k=array[i].compare( array[j]);
        if(k>0){
            temp=array[i];
            array[i]=array[j];
            array[j]=temp;
        }
    }
    cout<<"Sorted names are given below \n";
    for(i=0; i<4; i++)
```

```
    cout<<array[i]<<"\n";
}
```

In Program 5.20, the vector object of type string is declared in the same manner as in Program 5.19.

```
    vector <string>array(4);
```

The first part of the main function gets four strings from the keyboard. The next part sorts the strings. The function compare of C++ library has been used as given below:

```
    int k=array[i].compare( array[j]);
```

The compare function returns 0 if the strings are identical. If string1 is less than string2, then it will return a negative number. If string1 is greater than string2 then it will return a positive number. The returned integer is assigned to integer k. If k is greater than 0, then string1 is greater than string2. Since the sorting needs to be done in ascending order, you may interchange both the strings.

Finally, you may print out the sorted strings.

The output of the Program

```
Enter 4 strings
karthik
joseph
kanna
ganesh
Sorted names are given below
ganesh
joseph
kanna
karthik
```

Using Vectors with Two Elements: The convenience of vectors will become apparent when they are need to define objects with two fields or two elements or two coordinates. An aggregate of two elements has to be created using a structure. You may create a structure called `Marks` with two data types as given below:

```
    struct Marks{
        string name;
        unsigned short mark;
    };
```

The above declares a structure named `Marks`. It has two data items namely, 'name' of type string and 'mark' of type unsigned short `int`. A vector with the structure as above is thus declared.

```
    vector <Marks>Array(4);
```

This vector object named `Array` contains four elements. Each element has a structure of Marks. In the previous example, data type was double. In this example the data type is a structure called Marks. Program 5.21 initializes one of the elements of Array and then displays the same.

Program 5.21

```
//To demonstrate structure using vectors
#include<iostream>
#include<vector>
using namespace std;
struct Marks{
    string name;
    unsigned short mark;
};
int main(){
    vector <Marks>Array(4);
    Array[2].name="Radha";
    Array[2].mark=99;
    cout<<(Array[2]). name<<"\t"<<Array[2].mark;
}
```

Program 5.21 creates a vector of `Marks` type of size four. Then, the individual items of the third element, i.e., with index 2 are initialized as given below:

```
Array[2].name="Radha";
Array[2].mark=99;
```

Since the second element of the Array object has been referred, square bracket are used as in the case of simple arrays. Only while declaring a vector array, parentheses are used. Note also how the contents of each member of the vector element is printed.

The output of the Program

```
Radha 99
```

Sorting Vector Elements Based on a Key – Bubble Sort: You can sort a vector elements as done with array elements. However, in the vector, you have two data items in each element. You can sort the vector based on one of the elements, which you may call the key. The example given in Program 5.20 is repeated in Program 5.22.

Program 5.22

```
//To demonstrate sorting vector elements
#include<iostream>
#include<vector>
using namespace std;
struct Marks{
    string name;
    unsigned short mark;
    };
int main(){
    vector <Marks>array(4);
    int i=0, j=0;
    cout<<"Enter 4 names followed by marks \n";
    for(i=0; i<4; i++)
```

```
        cin>>array[i].name>>array[i].mark;
    Marks temp;
    for(i=0; i<3; i++)
    for(j=i+1; j<4; j++){
        int k=(array[i].name). compare( array[j].name);
        if(k>0){
            temp=array[i];
            array[i]=array[j];
            array[j]=temp;
        }
    }
    cout<<"Sorted vector is given below \n";
    for(i=0; i<4; i++){
        cout<<array[i].name<<"\t"<<array[i].mark<<"\n";
    }
}
```

The structure `Marks` is declared as a global type here as well as in the previous program. Then it is possible declare a vector object of type Marks with four elements. The user has to input the names and marks of the four sets using the `cin` object. Then, you may sort the array based on one of the elements, namely, 'name'. The interesting part is in the block, where the data is interchanged when `k` is greater than 0. When the name of the preceding element is greater than the succeeding element, you can interchange not only the `names` but also the `marks`, i.e., the entire element. This is the popular bubble sort. What is new is that a vector of more than one element can be sorted based on one of the elements called key. Other sorting methods can also be implemented with vectors. After the sorting is completed, the contents of vector can be printed.

The output of the Program

```
John 96
Joseph 93
Anand 87
Xavier 78
Sorted vector is given below
Anand 87
John 96
Joseph 93
Xavier 78
```

Thus, the vector can be used for handling more complex arrays easily.

5.5   CONTAINER CLASSES

Containership is also referred to as nesting in which a class has an object of another class as its member. Containership (or containment or aggregation or composition), like inheritance, enables the implementation of a logical relationship between classes.

The class (enclosing class) that contains an object of another class has access only to the public members of that class. The private and protected members of the contained class are not accessible to the enclosing class.

Example 5.2: A program to demonstrate the containership of classes

```cpp
#include<iostream>
#include<stdio>
using namespace std;
class edudetail
{
protected:
    char school[20];
public:
    char degree[20];
    void getdetail()
    {
        cout<<"\nEnter the school education :";
        gets(school);
        cout<<"\nEnter the degree education :";
        gets(degree);
    }
    void showedu()
    {
        cout<<"School education :"<<school<<"\n";
        cout<<"Degree education :"<<degree;
    }
};
class employee
{
    char name[20], empno[6];
public :
    edudetail edu1;
    void getemp()
    {
        cout<<"\nEnter the employee code :";
        gets(empno);
        cout<<"\nEnter the name :";
        gets(name);
        edu1.getdetail();
    }
    void show();
};
void employee::show ()
{
    cout<<"\n\nEmployee Qualification\n";
```

```
    cout<<"Employee Code :"<<empno<<"\n";

    cout<<"Employee Name :"<<name<<"\n";

    //  cout<<edu1.school; protected member inaccessible

    edu1.showedu();

}

int main()

{

    employee e1;

    e1.getemp();

    e1.show();

    return 0;

}
```

The output of the program

```
Enter the employee code: em10

Enter the name: Mini Suzana Wesley

Enter the school education: 10 + 2

Enter the degree education: M.Sc IT

Employee Qualification

Employee Code: em10

Employee Name: Mini Suzana Wesley

School education: 10 + 2

Degree education: M.Sc IT
```

In this example, the object `edu1` of the class `edudetail` is declared in the class `employee`. The public data member `degree` of the class `edudetail` can be accessed with the help of object `edu1`. However, the protected data member `school` is inaccessible.

Containership vs Inheritance

Containership and inheritance facilitate the implementation of different real-world relationships through various classes. However, both differ in the type of relationship that they implement. Containership implements 'has-a' relationship, whereas inheritance implements the 'is-a-kind-of' relationship. Containership is appropriate in a situation where one real-world object has or contains another real-world object, whereas inheritance is appropriate in a situation where one real-world object is a special kind of some other real-world object.
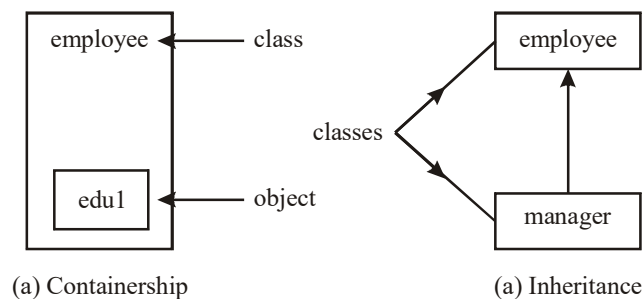

(a) Containership   (a) Inheritance

Fig. 5.14 *Containership and Inheritance*

Figure 5.14(a) is an example of containership that depicts 'has-a' kind of relationship. Here, the class `employee` 'has-a' object `edu1` of the class `edudetail`. Figure 5.14(b) is an example of inheritance that depicts 'is-a-kind-of' relationship. That is, `manager` 'is-a-kind-of' `employee`. The differences between containership and inheritance are listed in Table 5.8.

Table 5.8 Differences between Containership and Inheritance

Containership	Inheritance
The object of one class is used as a member in another class.	The object of one class inherits the property of another class.
Does not support the concept of reusability.	Supports the concept of reusability.
Does not provide additional features to an existing class.	Provides additional features to an existing class.
Represents 'has-a' relationship.	Represents 'is-a-kind-of' relationship.
The private and protected members of the inner class are not accessible to the enclosing class.	The derived class can access the protected members of the base class.

Check Your Progress

6. In which header are the disk file I/O operations declared?
7. Write the types of file pointers.
8. State about the command-line arguments.
9. Write the C++ standard library function.
10. Explain the term container class.

5.6 LISTS, MAP AND ALGORITHMS

A list is a sequence of elements of homogeneous type. For example, list of names, list of marks, list of addresses, list of employees and so on. Lists are once created and then modified during their lifetime. We need to add a new element, search the existence of a given element, delete an existing element from the list, etc. In addition, we can combine two or more lists to create a single list or split a list in two or more lists as per the requirements.

5.6.1 Map in C++ Standard Template Library (STL)

Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have same key values.

Following are some basic functions associated with Map:

begin() – Returns an iterator to the first element in the map.

end() – Returns an iterator to the theoretical element that follows last element in the map.

size() – Returns the number of elements in the map.

max_size() – Returns the maximum number of elements that the map can hold.

empty() – Returns whether the map is empty.

pair insert(keyvalue, mapvalue) – Adds a new element to the map.

erase(iterator position) – Removes the element at the position pointed by the iterator.

erase(const g) – Removes the key value '**g**' from the map.

clear() – Removes all the elements from the map.

Program 5.23

```cpp
#include <iostream>
#include <iterator>
#include <map>
using namespace std;
int main()
{
    // empty map container
    map<int, int> gquiz1;

    // insert elements in random order
    gquiz1.insert(pair<int, int>(1, 40));
    gquiz1.insert(pair<int, int>(2, 30));
    gquiz1.insert(pair<int, int>(3, 60));
    gquiz1.insert(pair<int, int>(4, 20));
    gquiz1.insert(pair<int, int>(5, 50));
    gquiz1.insert(pair<int, int>(6, 50));
    gquiz1.insert(pair<int, int>(7, 10));
    // printing map gquiz1
    map<int, int>::iterator itr;
    cout << "\nThe map gquiz1 is : \n";
    cout << "\tKEY\tELEMENT\n";
    for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr)
    {
        cout << '\t' << itr->first
            << '\t' << itr->second << '\n';
    }
    cout << endl;
    // assigning the elements from gquiz1 to gquiz2
    map<int, int> gquiz2(gquiz1.begin(), gquiz1.end());

    // print all elements of the map gquiz2
    cout << "\nThe map gquiz2 after"
        << " assign from gquiz1 is : \n";
```

```
cout << "\tKEY\tELEMENT\n";
for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
{
    cout << '\t' << itr->first
         << '\t' << itr->second << '\n';
}
cout << endl;
// remove all elements up to
// element with key=3 in gquiz2
cout << "\ngquiz2 after removal of"
        " elements less than key=3 : \n";
cout << "\tKEY\tELEMENT\n";
gquiz2.erase(gquiz2.begin(), gquiz2.find(3));
for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
{
    cout << '\t' << itr->first
         << '\t' << itr->second << '\n';
}
// remove all elements with key = 4
int num;
num = gquiz2.erase(4);
cout << "\ngquiz2.erase(4) : ";
cout << num << " removed \n";
cout << "\tKEY\tELEMENT\n";
for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
{
    cout << '\t' << itr->first
         << '\t' << itr->second << '\n';
}
cout << endl;
// lower bound and upper bound for map gquiz1 key = 5
cout << "gquiz1.lower_bound(5) : "
     << "\tKEY = ";
cout << gquiz1.lower_bound(5)->first << '\t';
cout << "\tELEMENT = "
     << gquiz1.lower_bound(5)->second << endl;
cout << "gquiz1.upper_bound(5) : "
     << "\tKEY = ";
cout << gquiz1.upper_bound(5)->first << '\t';
cout << "\tELEMENT = "
     << gquiz1.upper_bound(5)->second << endl;

    return 0;
}
```

The output of the program

tput and
File Handling in C++*

```
The  map  gquiz1  is  5.23:
        KEY       ELEMENT
        1       40
        2       30
        3       60
        4       20
        5       50
        6       50
        7       10
The  map  gquiz2  after  assign  from  gquiz1  is
5.23:
        KEY       ELEMENT
        1       40
        2       30
        3       60
        4       20
        5       50
        6       50
        7       10
    gquiz2  after  removal  of  elements  less  than
key=3  :
        KEY       ELEMENT
        3       60
        4       20
        5       50
        6       50
        7       10
    gquiz2.erase(4)  5.23:  1  removed
        KEY       ELEMENT
        3       60
        5       50
        6       50
        7       10
    gquiz1.lower_bound(5)  :        KEY  =  5
ELEMENT  =  50
    gquiz1.upper_bound(5)  :        KEY  =  6
ELEMENT  =  50
```

NOTES

*Self - Learning
Material* **277**

List of the Functions used in Map

Following is the list of all the functions that are used in Map:

- **map insert() in C++ STL** – Insert elements with a particular key in the map container.

- **map count() function in C++ STL** – Returns the number of matches to element with key value '**g**' in the map.

- **map equal_range() in C++ STL** – Returns an iterator of pairs. The pair refers to the bounds of a range that includes all the elements in the container which have a key equivalent to '**k**'.

- **map erase() function in C++ STL** – Used to erase element from the container.

- **map rend() function in C++ STL** – Returns a reverse iterator pointing to the theoretical element right before the first key-value pair in the map (which is considered its reverse end).

- **map rbegin() function in C++ STL** – Returns a reverse iterator which points to the last element of the map.

- **map find() function in C++ STL** – Returns an iterator to the element with key value '**g**' in the map if found, else returns the iterator to end.

- **map crbegin() and map crend() function in C++ STL** – The **map crbegin()** returns a constant reverse iterator referring to the last element in the map container. The **map crend()** returns a constant reverse iterator pointing to the theoretical element before the first element in the map.

- **map cbegin() and map cend() function in C++ STL** – The **map cbegin()** returns a constant iterator referring to the first element in the map container. The **map cend()** returns a constant iterator pointing to the theoretical element that follows last element in the multimap.

- **map emplace() in C++ STL** – Inserts the key and its element in the map container.

- **map max_size() in C++ STL** – Returns the maximum number of elements a map container can hold.

- **map upper_bound() function in C++ STL** – Returns an iterator to the first element that is equivalent to mapped value with key value '**g**' or definitely will go after the element with key value '**g**' in the map.

- **map operator= in C++ STL** – Assigns contents of a container to a different container, replacing its current content.

- **map lower_bound() function in C++ STL** – Returns an iterator to the first element that is equivalent to mapped value with key value '**g**' or definitely will not go before the element with key value '**g**' in the map.

- **map emplace_hint() function in C++ STL** – Inserts the key and its element in the map container with a given hint.

- **map value_comp() in C++ STL** – Returns the object that determines how the elements in the map are ordered ('**<**' by default).

- **map key_comp() function in C++ STL** – Returns the object that determines how the elements in the map are ordered ('**<**' by default).

- **map::size() in C++ STL** – Returns the number of elements in the map.

- **map::empty() in C++ STL** – Returns whether the map is empty.

- **map::begin() and map::end() in C++ STL** –The **map::begin()** returns an iterator to the first element in the map. The **map::end()** returns an iterator to the theoretical element that follows last element in the map.

- **map::operator[] in C++ STL** – This operator is used to reference the element present at position given inside the operator.

- **map::clear() in C++ STL** – Removes all the elements from the map.

- **map::at() and map::swap() in C++ STL** –The **map::at()** function is used to return the reference to the element associated with the key '**k**'. The **map::swap()** function is used to exchange the contents of two maps but the maps must be of same type, although sizes may differ.

5.6.2 Abstract Data Types (ADTs)

Generally, handling small problems is much easier than handling comparatively larger problems. The same rule is applied on the programming also. Therefore, a large program is decomposed into small logical units or modules, each of which does a well-defined subtask of the whole program. The size of each module is kept as small as possible and if required, other modules are invoked from it. This modular design provides several advantages. First, several people can be employed to work on a single program, which increases the speed of completing the given task. Second, a well-designed modular program has modules, independent of each others implementation, which makes the program easily modifiable.

An **Abstract Data Type (ADT)** is an extension of modular design in a way that the set of operations of an ADT are defined at a formal, logical level and nowhere in ADT's definition is mentioned how these operations are implemented. The data type integer is an example of abstract data type. We frequently perform operations on integers that are associated with them like addition, subtraction, division, multiplication, modulus, etc. However, we do not know how these operations are actually performed on integers. We only know the syntax of how to perform these operations in some programming language. For example, C language defines +, −, /, *, % to perform some basic arithmetic operations on integers.

The basic idea of ADT is that the implementation of the set of operations are written once in the program and the part of program which needs to perform an operation on ADT accomplishes this by invoking the required operation. If there is a need to change the implementation details of an ADT, the change will be completely transparent to the programs using it. The data structures, namely, *lists*, *stacks* and *queues* are some examples of ADTs.

The List ADT

Mathematically, a list is a sequence of zero or more elements where each element is of type T. It is generally represented by a comma-separated sequence of elements as shown here.

$$a_1, \ a_2, \ a_3, \ ...., \ a_n$$

In the above list, n denotes the size of list and a_1 and a_n are the first and last element of the list, respectively. In case n=0, the list is called an **empty list**—a list having no elements.

One important property associated with lists is that its elements can be linearly ordered according to their positions in the list. For any list of size n, we can say that the element a_i is the successor of a_{i-1} for 1<i<=n and the predecessor of a_{i+1} for 1<=i<n. The position of any element a_i in the list is i. To form the list an abstract data type, a set of operations must be defined that can be performed on the objects of list type. Some common operations that can be applied on lists include traversing the list, inserting a new element into the list and deleting some specific element from the list. All these operations are discussed in this chapter.

There are various ways to implement the lists. In this chapter, we will discuss array, linked list and cursor implementation of lists.

Array-Based Implementation of Lists

One way to implement lists is to use an array. The elements of lists are stored in contiguous locations in the array. Thus, each element of the list can be directly accessed using its position in the array and the whole list can be traversed very easily. Figure 5.15 shows an example of array implementation of lists.
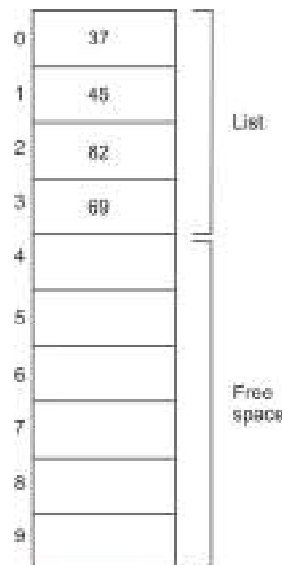

Fig. 5.15 *An Example of Array Implementation of Lists*

The array implementation of list is quite simple, however, there are certain problems associated with it. First, a sufficient block of memory needs to be allocated to array at compile-time and once the memory is allocated it cannot be expanded or contracted. This may lead to significant wastage of memory or short of memory if the number of elements in the list increases or decreases significantly at run-time. Another problem is that the insertion (except the insertion at the end) and deletion of an element from an array are expensive operations, since they may require a number of elements to be shifted to either make space for new elements or cover the gap created by deleted elements. Because of these problems, arrays are not generally used to implement the lists.

5.6.3 Linked List Implementation

In this implementation, the successive elements of list are linked by means of pointers or links; therefore they are named as linked list. This implementation frees the programmer from the constraint of using contiguous memory to store a list. Also, the insertion or deletion of elements does not require shifting of the existing elements as in the case of array implementation; elements can be inserted or deleted merely by adjusting the pointers. However, one disadvantage associated with this implementation is the extra space required for storing the pointers.

Formally, a **linked list** is defined as a linear collection of homogeneous elements called **nodes**. Each node of the linked lists stores an element of the list as well as a pointer to the node containing the next element of the list. For example, in the linked list representing the list a_1, a_2, a_3, ..., a_n, the node containing a_1 stores a pointer to the node containing a_2 that in turn stores a pointer to the node containing a_3 and so on.

Depending on the number of pointers in a node of the linked list or the purpose for which the pointers are maintained, a linked list can be classified into various types, such as singly linked list, circular linked list and doubly linked list. In this section, we will discuss only singly and doubly linked lists.

Singly Linked Lists

In a singly linked list (also called **linear linked list**), each node consists of two fields: `info` and `next` (Refer Figure 5.16). The `info` field contains the data and the `next` field contains the address of memory location where the next node is stored. The last node of the singly linked list contains `NULL` in its `next` field that indicates the end of list.
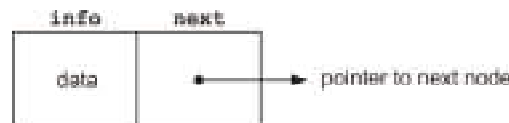
Fig. 5.16 Node of a Singly Linked List

Note: *The data stored in info field may be a single data item of any data type or a complete record representing a student or an employee or any other entity. In this chapter, however, we assume that the info field contains an integer data.*

A linked list contains a list pointer variable `Start` that stores the address of the first node of the list. In case, the `Start` contains `NULL`, the list is called an

empty list or a **null list**. Since, each node of the list contains only a single pointer pointing to the next node (not to previous node) thereby allowing traversing in only one direction, it is also referred to as **one-way list**. Figure 5.17 shows a singly linked list with four nodes.
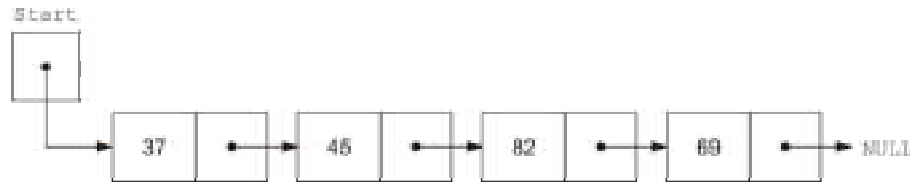

Fig. 5.17 *A Singly Linked List with Four Nodes*

A number of operations can be performed on the singly linked lists. These operations include traversing, searching, inserting and deleting nodes, reversing, sorting and merging linked lists. Before implementing these operations, first, we need to understand how a node of a linked list is created.

Creating a Node

Creating a node means defining its structure, allocating memory to it and its initialization. As discussed earlier, the node of a linked list consists of data and a pointer to next node. To define a node containing an integer data and a pointer to next node in C language, we can use a self-referential structure whose definition is shown here.

```
typedef  struct  node
{
     int  info;        /*to  store  integer  type  data*/
     struct  node  *next;
                      /*to  store  a  pointer  to  next
node*/
}Node;

Node  *nptr;          /*nptr  is  a  pointer  to  node*/
```

After declaring a pointer `nptr` to the new node, the memory needs to be allocated dynamically to it. If the memory is allocated successfully (that is, no overflow), the node is initialized. The `info` field is initialized with a valid value and the `next` field is initialized with NULL.

Algorithm 5.1 Creation of Node

```
  create_node(

1. Allocate memory for nptr
                        //nptr is a pointer to new
   node
2. If nptr = NULL
       Print "Overflow: Memory not allocated!" and go
   to step 7
   End If
```

```
3. Read item            //item is the value to be
   inserted                           //in the
   new node
4. Set nptr->info = item
5. Set nptr->next = NULL
6. Return nptr           //returning pointer nptr
7. End
```

Now, the linked list can be formed by creating several nodes of type `Node` and inserting them either in the beginning or at the end or at a specified position in the list.

Traversing

Traversing a list means accessing its elements one by one to process all or some of the elements. For example, if we need to display values of the nodes, count the number of nodes or search a particular item in the list, then traversing is required. We can traverse the list by using a temporary pointer variable (say, `temp`), which points to the node currently being processed. Initially, we make `temp` to point to the first node, process that element, then move `temp` to point to the next node using statement `temp=temp->next`, process that element and move so on as long as the last node is not reached, that is, until `temp` becomes `NULL`.

Algorithm 5.2 Traversing a List

```
display(Start)

1. If Start = NULL       //Start points to the
   first                              //node of
   list
       Print "List is empty!!" and go to step 4
   End If
2. Set temp = Start      //initializing temp with
   Start
3. While temp != NULL
       Print temp->info      //displaying value
   of eachnode
       Set temp = temp->next
                         //moving temp to point to
   next node
   End While
4. End
```

Another example of traversing a linked list is counting number of nodes in the linked list, which is given here.

Algorithm 5.3 Counting Number of Nodes

```
count_node(Start)

1. Set count = 0
2. Set temp = Start      //initializing temp with
   Start
3. While temp != NULL    //traversing the list
       Set count = count + 1

                         //incrementing count

       Set temp = temp->next
   End While
4. Return count          //returning total number
   of nodes                              //in the
   list
5. End
```

Insertion

To insert a node in the linked list, a new node is created (as explained in Algorithm 5.1) and then placed at the desired position by adjusting the pointers. Nodes can be inserted either in the beginning or at the end or at any specified position in the list as discussed in this section.

Insertion in Beginning

To insert a node in the beginning of list, the next field of new node (pointed by nptr) is made to point to the existing first node and the Start pointer is modified to point to the new node (Refer Figure 5.18).

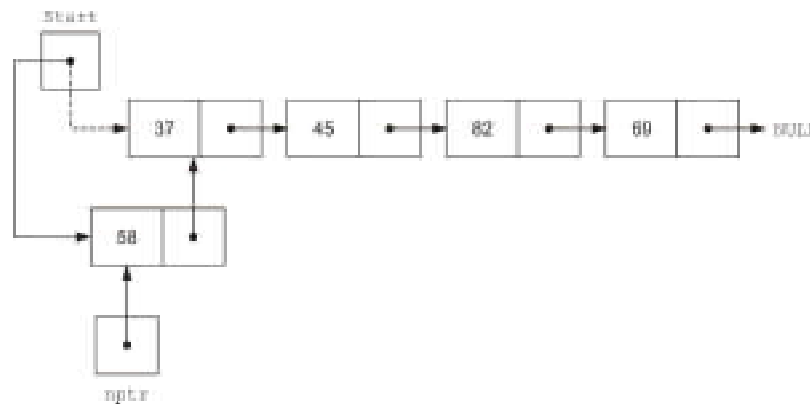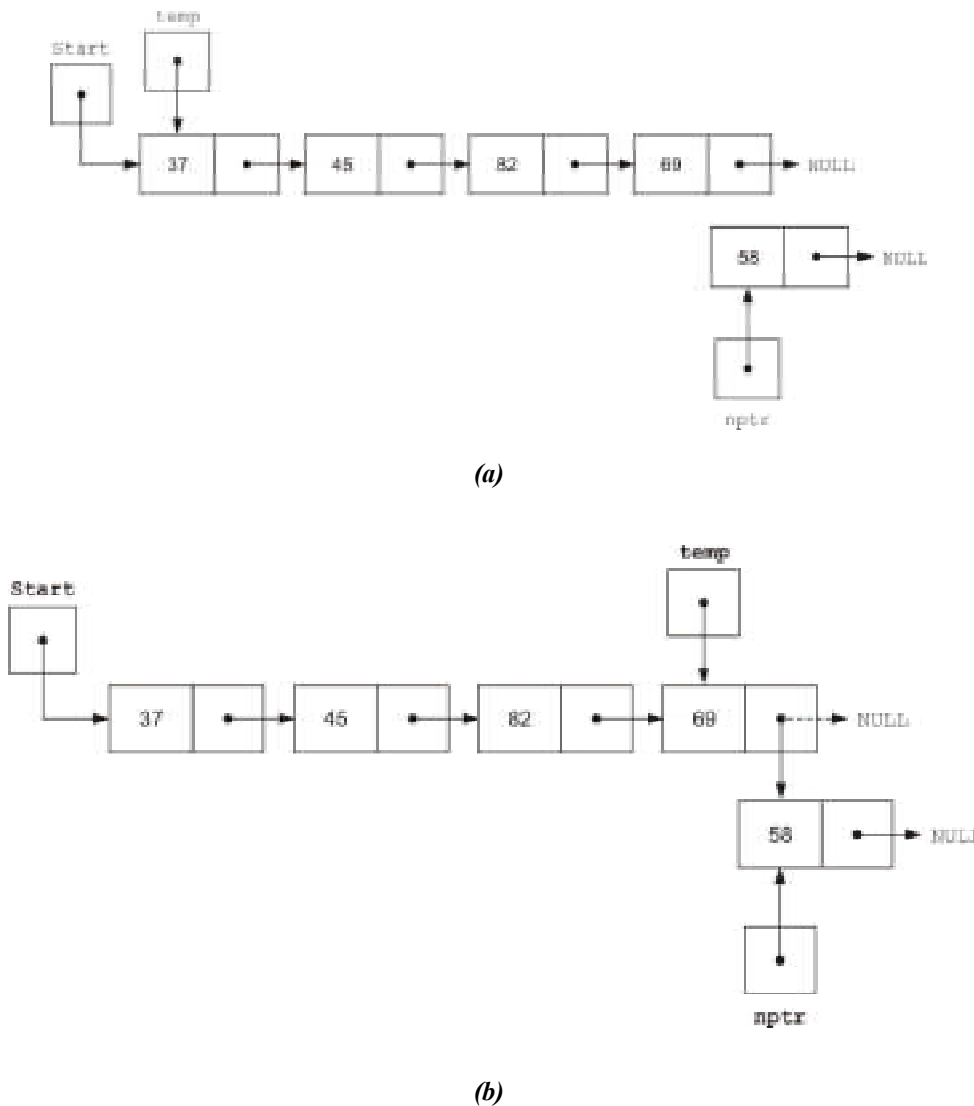**Fig. 5.18** *Insertion in the Beginning of a Linked List*

Algorithm 5.4 Insertion in Beginning

```
insert_beg(Start)
1. Call create_node()

              //creating a new node pointed by
   nptr
```

```
2. Set nptr->next = Start

3. Set Start = nptr

        //Start pointing to new node

4. End
```

Insertion at End

To insert a node at the end of a linked list, the list is traversed up to the last node and the `next` field of this node is modified to point to the new node. However, if the linked list is initially empty, then the new node becomes the first node and `Start` points to it. Figure 5.19(a) shows a linked list with a pointer variable `temp` pointing to its first node and Figure 5.19(b) shows `temp` pointing to the last node and the `next` field of last node pointing to the new node.

(a)

(b)

Fig. 5.19 *Insertion at the End of a Linked List*

Algorithm 5.5 Insertion at End

```
insert_end(Start)

1. Call create_node()
                //creating a new node pointed by
   nptr
2. If Start = NULL
                //checking for empty list
     Set Start = nptr
                //inserting new node as the first
   node
   Else
     Set temp = Start
     While temp->next != NULL
                //traversing up to the last node
          Set temp = temp->next
     End While
     Set temp->next = nptr
                //appending new node at the end
   End If
3. End
```

Insertion at a Specified Position

To insert a node at a position (say, pos) specified by the user, the list is traversed up to pos-1 position. Then the next field of new node is made to point to the node that is already at pos position and the next field of the node at pos-1 position is made to point to the new node. Figure 5.20 shows the insertion of the new node pointed by nptr at third position.
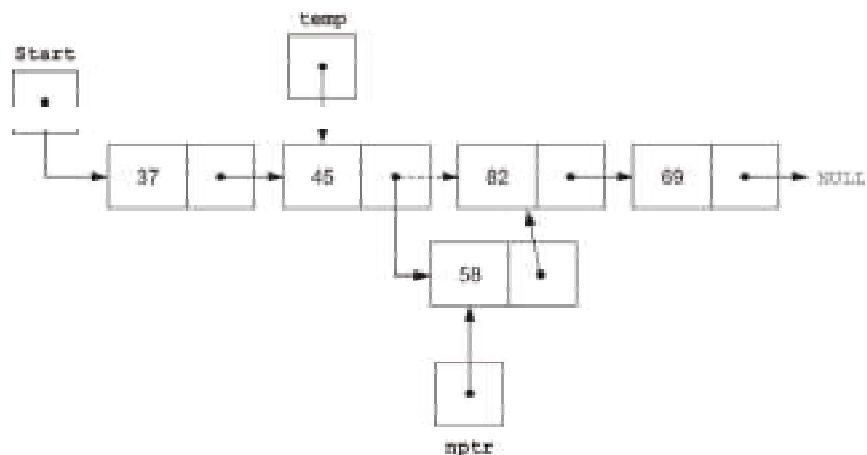

Fig. 5.20 *Insertion at a Specified Position in a Linked List*

Algorithm 5.6 Insertion at a Specified Position

```
insert_pos(Start)
1. Call create_node()
                //creating a new node pointed by
   nptr
2. Set temp = Start
3. Read pos
                //position at which the new node
                is to be
                //inserted
4. Call count_node(temp)
                //counting total number of nodes
   in count                     //variable
5. If (pos > count + 1 OR pos = 0)
     Print "Invalid position!" and go to step 7
   End If
6. If pos = 1
     Set nptr->next = Start
     Set Start = nptr  //inserting new node as
   the first node
   Else
     Set i = 1
     While i < pos - 1
               //traversing up to the node at pos-
   1 position
         Set temp = temp->next
         Set i = i + 1
     End While
     Set nptr->next = temp->next
               //inserting new node at pos
   position
     Set temp->next = nptr
   End If
7. End
```

Deletion

Like insertion, nodes can be deleted from the linked list at any point of time and from any position. Whenever a node is deleted, the memory occupied by the node is deallocated. Note that while performing deletions, we need to keep track of the node that is the immediate predecessor of the node to be deleted. Thus, two

temporary pointer variables are used (except in case of deletion from beginning) while traversing the list.

Note: *A situation where the user tries to delete a node from an empty linked list is termed as Underflow.*

Deletion from Beginning

To delete a node from the beginning of a linked list, the address of the first node is stored in a temporary pointer variable `temp` and `Start` is modified to point to the second node in the linked list. After that the memory occupied by the node pointed by `temp` is deallocated. Figure 5.21 shows the deletion of node from the beginning of a linked list.
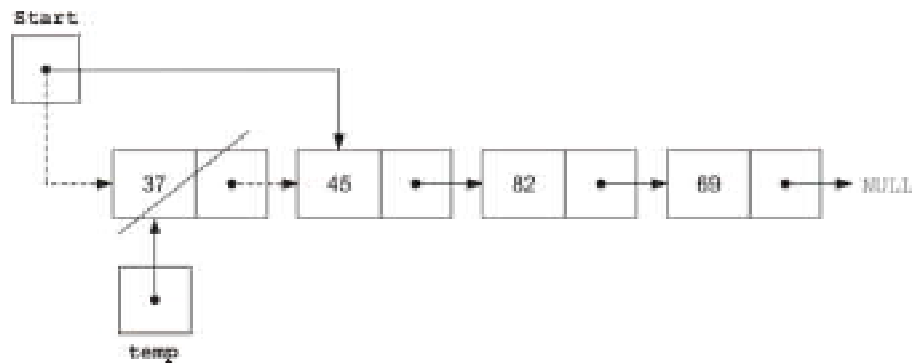

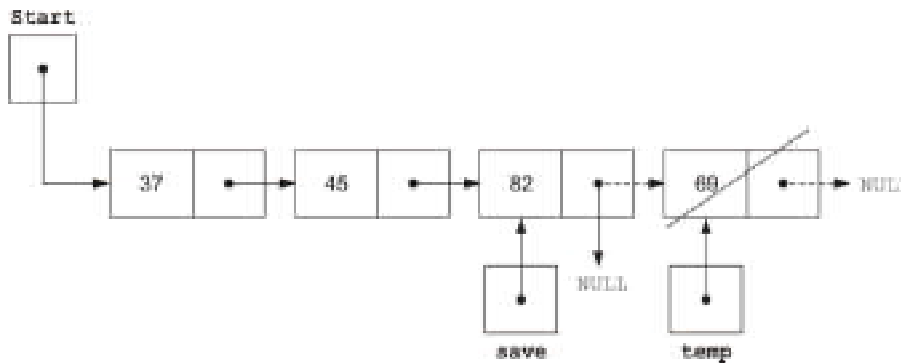***Fig. 5.21*** *Deletion from the Beginning of a Linked List*

Algorithm 5.7 Deletion from Beginning

```
delete_beg(Start)

1. If Start = NULL       //checking for underflow

      Print "Underflow: List is empty!" and go to
   step 5

   End If

2. Set temp = Start      //temp pointing to the
   first node

3. Set Start = Start->next    //moving Start to
   point to the                           /  /
   second node

4. Deallocate temp       //deallocating memory

5. End
```

Deletion from End

To delete a node from the end of a linked list, the list is traversed upto the last node. Two pointer variables `save` and `temp` are used to traverse the list, where `save` points to the node previously pointed by `temp`. At the end of traversing, `temp` points to the last node and `save` points to the second last node. Then the `next` field of the node pointed by `save` is made to point

to NULL and the memory occupied by the node pointed by `temp` is deallocated. Figure 5.22 shows the deletion of node from the end of a linked list.

***Fig. 5.22** Deletion from the End of a Linked List*

Algorithm 5.8 Deletion from the End

```
delete_end(Start)

1. If Start = NULL     //checking for underflow
       Print "Underflow: List is empty!" and go to
   step 6
   End If
2. Set temp = Start    //temp pointing to the first
   node
3. If temp->next = NULL //deleting the only node
   of the list
       Set Start = NULL
   Else
       While (temp->next) != NULL
                    //traversing up to the last
   node
           Set save = temp
                    //save pointing to node
   previously
                    //pointed by temp
           Set temp = temp->next
                    //moving temp to point to
   next node
       End While
   End If
4. Set save->next = NULL
                //making new last node to point to
   NULL
```

```
5. Deallocate temp     //deallocating memory
6. End
```

Deletion from a Specified Position

To delete a node from the position (say, `pos`) specified by the user, the list is traversed upto the `pos` position using pointer variables `temp` and `save`. At the end of traversing, `temp` points to the node at `pos` position and `save` points to the node at `pos-1` position. Then the `next` field of the node pointed by `save` is made to point to the node at `pos+1` position and the memory occupied by the node pointed by `temp` is deallocated. Figure 5.23 shows the deletion of node at third position.
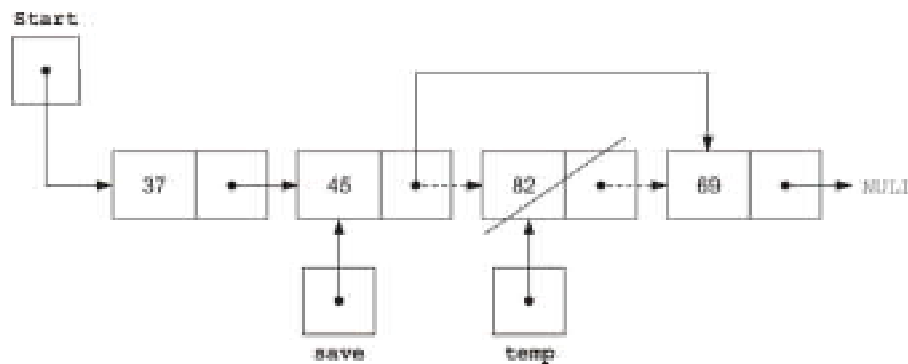

Fig. 5.23 *Deletion from a Specified in a Linked List*

Algorithm 5.9 Deletion from a Specified Position

```
delete_pos(Start)
1. If Start = NULL    //checking for underflow
      Print "Underflow: List is empty!" and go to
   step 8
   End If
2. Set temp = Start
3. Read pos            //position of the node to be
   deleted
4. Call count_node(Start)
                       //counting total number of
   nodes in                    //count variable
5. If pos > count OR pos = 0
      Print "Invalid position!" and go to step 8
   End If
6. If pos = 1
      Set Start = temp->next
                       //deleting the first node
   Else
```

```
      Set i = 1

      While i < pos   //traversing up to the node
                          //at position pos

            Set save = temp

            Set temp = temp->next

            Set i = i + 1

      End While

      Set save->next = temp->next

                        //deleting the node at
    position pos

    End If
7. Deallocate temp    //deallocating memory
8. End
```

Doubly Linked Lists

In a singly linked list, each node contains a pointer to the next node and it has no information about its previous node. Thus, we can traverse only in one direction, that is, from beginning to end. However, sometimes it is required to traverse in the backward direction, that is, from end to beginning. This can be implemented by maintaining an additional pointer in each node of the list that points to the previous node. Such type of linked list is called **doubly linked list**.

Each node of a doubly linked list consists of three fields: `prev`, `info` and `next` (Refer Figure 5.24). The `info` field contains the data, the `prev` field contains the address of the previous node and the `next` field contains the address of the next node.

Fig. 5.24 *Node of a Doubly Linked List*

Since, a doubly linked list allows traversing in both forward and backward directions, it is also referred to as a **two-way list**. Figure 5.25 shows an example of a doubly linked list having four nodes. Note that the `prev` field of the first node and the `next` field of the last node in a doubly linked list points to NULL.
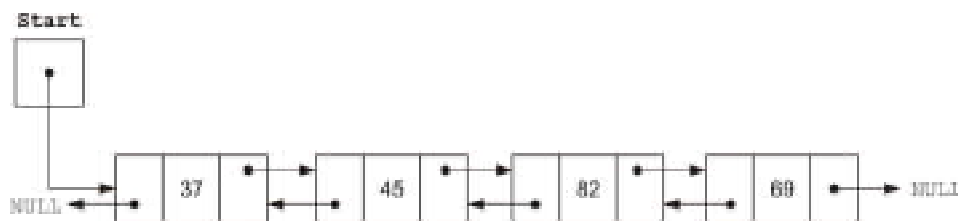

Fig. 5.25 *An Example of a Doubly Linked List with Four Nodes*

To define the node of a doubly linked list in C language, the structure used to represent the node of singly linked list is extended to have an extra pointer, which points to previous node. The structure of a node of doubly linked list is shown here.

```
typedef struct node
{
     int info;              /*to store integer type
data*/
     struct node *next;     /*to store a pointer to next
                              node*/
     struct node *prev;     /*to store a pointer to
previous                             node*/
}Node;
Node *nptr;                  /*nptr is a pointer to node*/
```

When memory is allocated successfully to a node, that is, no overflow, the node is initialized. The info field is initialized with a valid value and the prev and next fields are initialized with NULL.

Algorithm 5.10 Creating a Node of Doubly Linked List

```
 create_node()

1. Allocate memory for nptr
                         //nptr is a pointer to new
   node
2. If nptr = NULL
       Print "Overflow: Memory not allocated!" and go
                                         to step 8
3. Read item           //item is the value stored
   in the node
4. Set nptr->info = item
5. Set nptr->next = NULL
6. Set nptr->prev = NULL
7. Return nptr
8. End
```

Note that all the operations that are performed on singly linked lists can also be performed on doubly linked lists. In this section, we will discuss only insertion and deletion operations on doubly linked lists.

Insertion

To insert a new node in the beginning of a doubly linked list, a pointer (say, nptr) to new node is created. The next field of new node is made to point to the existing first node and prev field of the existing first node (that has become the second node now) is made to point to the new node. After that, Start is modified to point to new node. Figure 5.26 shows the insertion of node in the beginning of a doubly linked list.
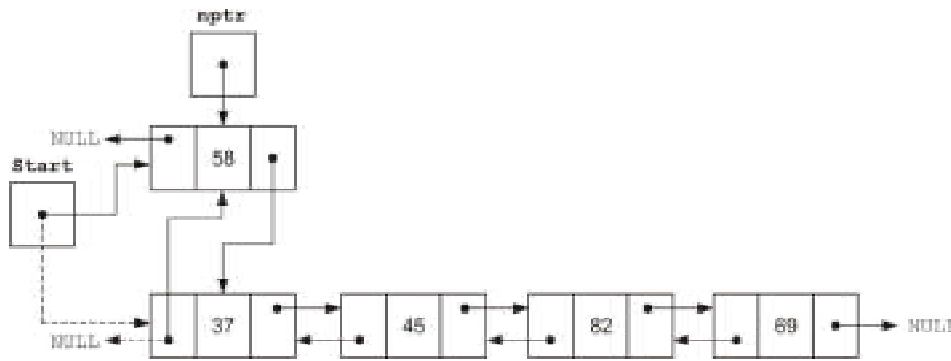
Fig. 5.26 *Insertion in the Beginning*

Algorithm 5.11 Insertion in the Beginning

```
 insert_beg(Start)

1. Call create_node()  //creating a new node pointed
   by                      //nptr

2. If Start != NULL

     Set nptr->next = Start

                     //inserting node in the
   beginning

     Set Start->prev = nptr

   End If

3. Set Start = nptr    //making Start to point to
   new node

4. End
```

To insert a new node at the end of a doubly linked list, the list is traversed upto the last node using some pointer variable (say, `temp`). At the end of traversing, `temp` points to the last node. Then, the `next` field of the last node (pointed by `temp`) is made to point to the new node and the `prev` field of the new node is made to point to the node pointed by `temp`. However, if the list is empty, the new node is inserted as the first node in the list. Figure 5.27 shows the insertion of new node at the end of a doubly linked list.
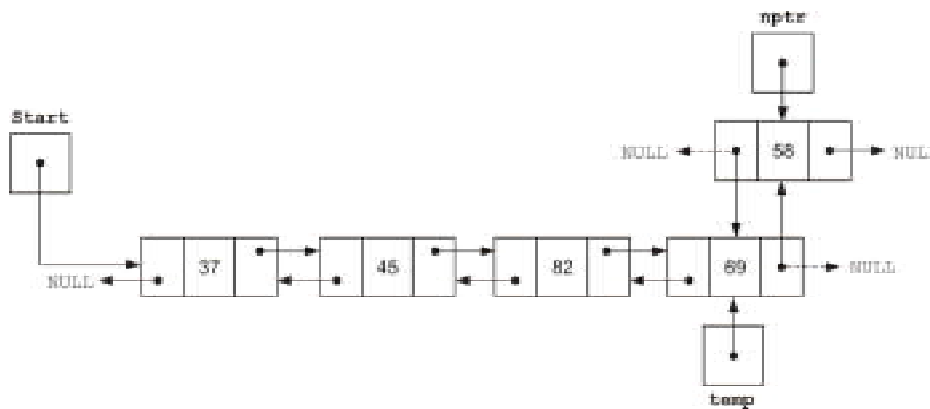
Fig. 5.27 *Insertion at the End*

Algorithm 5.12 Insertion at the End

```
insert_end(Start)

1. Call create_node()   //creating a new node pointed
   by nptr

2. If Start = NULL

     Set Start = nptr  //inserting new node as
   the first node

   Else

     Set temp = Start  //pointer temp used for
   traversing

     While temp->next != NULL

          Set temp = temp->next

     End While

     Set temp->next = nptr

     Set nptr->prev = temp

   End If

3. End
```

To insert a new node (pointed by nptr) at a specified position (say, pos) in a doubly linked list, the list is traversed upto pos-1 position. At the end of traversing, temp points to the node at pos-1 position. For simplicity, we use another pointer variable (say, ptr) to point to the node that is already at pos position. Then, the prev field of the node pointed by ptr is made to point to the new node and the next field of the new node is made to point to the node pointed by ptr. Also, the prev field of the new node is made to point to the node pointed by temp and the next field of the node pointed by temp is made to point to the new node. Figure 5.28 shows the insertion of new node at the third position in a doubly linked list.
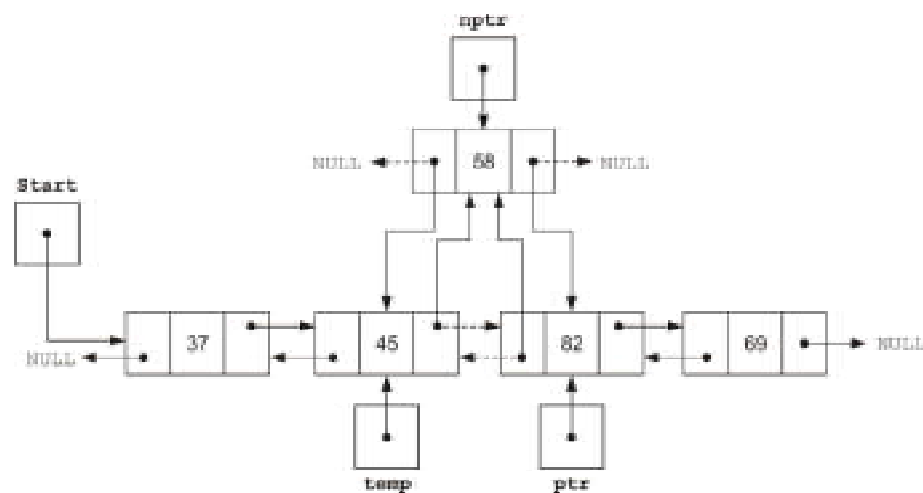

Fig. 5.28 *Insertion at a Specified Position*

Algorithm 5.13 Insertion at a Specified Position

```
insert_pos(Start)
```

1. Call create_node()
                //creating a new node pointed by
   nptr

2. Set temp = Start

3. Read pos

4. Call count_node(temp)
                //counting number of nodes in count

5. If pos = 0 OR pos > count + 1

     Print "Invalid position!" and go to step 7

   End If

6. If pos = 1

     Set nptr->next = Start

                //inserting node at the beginning

     Set Start = nptr

                //Start pointing to new node

   Else

     Set i = 1

     While i < pos-1

                //traversing up to the node at pos-
   1 postion

            Set temp = temp->next

            Set i = i + 1

     End While

     Set ptr = temp->next

     Set ptr->prev = nptr

     Set nptr->next = ptr

     Set nptr->prev = temp

     Set temp->next = nptr

   End If

7. End

Deletion

To delete a node from the beginning of a doubly linked list, a pointer variable (say, temp) is used to point to the first node. Then Start is modified to point to the next node and the prev field of this node is made to point to NULL. After that the memory occupied by the node pointed by temp is deallocated. Figure 5.29 shows the deletion of a node from the beginning of a doubly linked list.
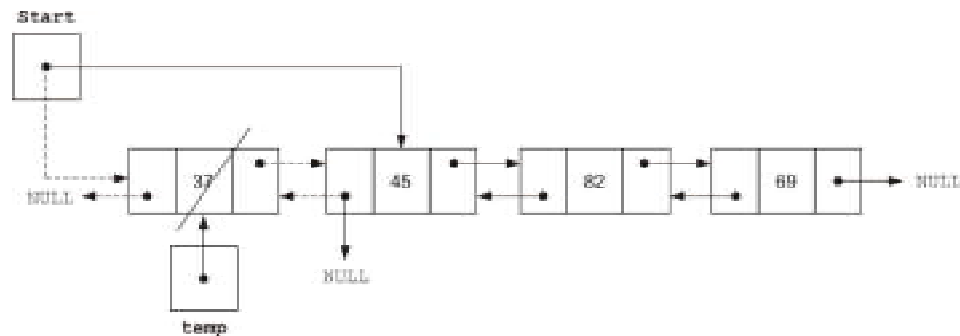

Fig. 5.29 *Deletion from the Beginning*

Algorithm 5.14 Deletion from Beginning

```
delete_beg(Start)

1. If Start = NULL

      Print "Underflow: List is empty!" and go to
   step 6

   End If

2. Set temp = Start            //temp points to the
   node to                                        //be
   deleted

3. Set Start = Start->next      //making Start
   to point to
   //next node

4. Set Start->prev = NULL

5. Deallocate temp            //deallocating
   memory

6. End
```

Note: The process of deleting node from the end of a doubly linked list is same as that of singly linked list.

To delete a node from a position (say, pos) specified by the user, the list is traversed upto the pos position using pointer variables temp and save. At the end of traversing, temp points to the node at pos position and save points to the node at pos−1 position. Here, for simplicity, we use another pointer variable ptr to point to the node at pos+1 position. Then the next field of the node at pos−1 position (pointed by save) is made to point to the node at pos+1 position (pointed by ptr). In addition, the prev field of the node at pos+1

position (pointed by `ptr`) is made to point to the node at `pos-1` position (pointed by `save`). After that the memory occupied by the node pointed by `temp` is deallocated. Figure 5.30 shows the deletion of node at third position from a doubly linked list.
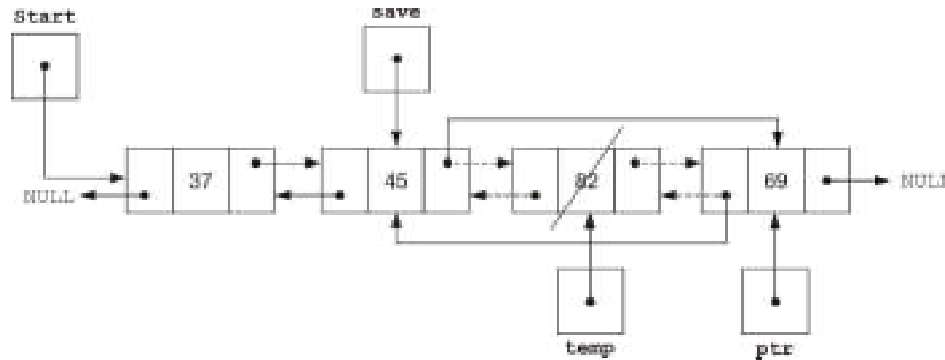
Fig. 5.30 *Deletion from a Specified Position*

Algorithm 5.15 Deletion from a Specified Position

```
delete_pos(Start)

1. If Start = NULL

      Print "Underflow: List is empty!" and go to
   step 8

   End If

2. Set temp = Start

3. Read pos

4. Call count_node(temp)

        //counting total number of nodes in count
   variable

5. If pos > count OR pos = 0

      Print "Invalid position!" and go to step 6

   End If

6. If pos = 1

      Set Start = Start->next     //deleting the
   first node

      Start->prev = NULL

   Else

      Set i = 1

      While i < pos

           //traversing up to the node at pos
   position

         Set save = temp
```

```
                      //save pointing to the node at pos-1
    position
            Set temp = temp->next
               //making temp to point to next node
            Set i = i + 1
        End While
        Set ptr = temp->next
        Set save->next = ptr
        Set ptr->prev = save
    End If
7. Deallocate temp            //deallocating
    memory
8. End
```

5.7 STRING CLASS

String manipulation is one of the most common task of any C++ program. A string is defined as a sequence of characters terminated by null character and can be represented as an array of `char` type. However, C++ also provides a better alternative for handling string. It comes with the built-in `string` class defined in `string` header, which facilitates convenient handling of the strings and the various operations related to it.

String Class

The `string` class is the most important class of C++ which is used for manipulation of strings. The `string` class automatically manages memory requirements for strings. One can work with the objects of type `string` in the same way as one works with variables of the built-in data type.

The need to include `string` class in standard library is due to the fact that the standard C++ operators can be used to perform some basic operations on the objects that are of `string` type, while null-terminated strings cannot be manipulated by any of the standard C++ operators. Another reason is the safety that it provides while copying a string in another string. In case of copying the character array, if the size of the target array is less than that of the source array then a program error or system crash may occur. However, in case of the objects of `string` type such problems are automatically handled.

Therefore, the advantages of `string` class can be summarized as:

- **Consistency:** a new data type of `string` type is created
- **Convenience:** the standard C++ operators can be used to operate on strings
- **Safety:** array boundaries are automatically taken care of

Some of the simple operations that can be performed on the objects of `string` class by using various operators, listed in Table 5.9.

Table 5.9 *Operators Applicable on Objects of* `string` *Type*

Operator	Meaning
=	Assignment
==	Equality
!=	Inequality
+	Concatenation
+=	Concatenation assignment
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
[ ]	Subscripting
<<	Output
>>	Input

However, to perform more complex operations, several member functions of the `string` class can be used, which are listed in Table 5.10.

Table 5.10 *Member Functions of* `string` *Class*

Function	Purpose
`append()`	appends a part of string or entire string to the end of another string
`assign()`	assigns a part or entire string to another string
`at()`	accesses an individual character of string from a specified location
`begin()`	returns an iterator to the first character of a string
`compare()`	compares two strings and gives appropriate result
`empty()`	returns 1 if string is empty and 0 if string is not empty
`end()`	returns an iterator to the end of the string
`erase()`	removes the substring from a string
`insert()`	inserts characters in the string at the specified position
`length(), size()`	returns the number of characters present in a string
`replace()`	replaces specified substring with new substring

Creating String Objects

The `string` class consists of constructors and member functions with multiple overloaded forms to create and manipulate `string` objects. The prototypes of three of its most commonly used constructors are given here.

```
string();                    //for creating empty
   string object
string(const char *str);     //for   creating
   string object from
   //the null-terminated string str
string(const string &str);   //for   creating
   string from another
      //string
```

An object of the `string` type represents a sequence of characters of the `char` type.

The syntax for creating an object of `string` class is

```
string string_object;
```

where,

`string` = C++ keyword

`string_object` = name of a variable used to store string

For example, consider these statements.

```
string str1;            //creates an empty string
string str2("Hello"); //creates a string with
   initial value
string str4 = str3;   //creating string from
   already                        //existing
   string
```

Program 5.24: A program to demonstrate the basic functionality of object of `string` class type

```
#include<iostream>
#include<string>                    //header for
   string class
using namespace std;
int main()
{
     string str1("Computer");
     string str2;
     cout<<"Enter the second string : ";
     cin>>str2;
     cout<<"Length  of  first  string  :
   "<<str1.length();
     cout<<"\nFirst string : "<<str1;
     cout<<"\nLength of second string :
```

```
"<<str2.length();
    cout<<"\nSecond string : "<<str2;
    return 0;
}
```

The output of the program

```
Enter the second string : Science
Length of first string : 8
First string : Computer
Length of second string : 7
Second string : Science
```

In this example, two objects `str1` and `str2` of `string` type are created and initialized with the values `Computer` and `Science`, respectively. The length of the strings is then displayed using the `length()` function of `string` class. Note that the length of a string can also be displayed using the `size()` function.

> **Note:** *If the size of the strings is not specified anywhere, `string` objects are automatically sized to hold the string that are assigned to them.*

String Operations

Simple string operations can be performed by using string operators and the advanced operations can be carried out by using the member functions of `string` class. Some of the common string operations using member functions of `string` class are discussed here.

Concatenating Strings

Concatenation of one string at the end of another string can be accomplished by using the '+' operator. Concatenation of a string variable with string literal can also be accomplished with the help of the '+' operator. The string literal can appear on the either side of '+' operator. For example, consider these statements.

```
char str[] = "How are you";
str3 = str1 + str2;
str3 = "Hello " + str2;
str3 = str1 + str;
```

where,

`str1` and `str2` are objects of `string` type and `str` is a character array.

Note: Two string literals cannot be concatenated using the '+' operator. One of the string operands to be concatenated must be an object of the string type.

The program given here demonstrates the working of the concatenation operator '+' over strings.

Program 5.25: A program to demonstrate string concatenation

```cpp
#include<iostream>
#include<string>
using namespace std;
int main()
{
        //Declaring objects of string type
        string Fname, Sname, Name, Str1, Str2;
        cout<<"Enter your first name : ";
        cin>>Fname;                  //reading string
        cout<<"Enter your last name : ";
        cin>>Sname;                  //reading string
        //Concatenating two objects of string type
        //and one string literal
        Name = Fname + " " + Sname;
        //Concatenating one string literal
        //and one object of string type
        Str1 = "Hello! " + Name;
        //Concatenating one object of string type
        //and one string literal
        Str2 = Str1 + ", How are you?";
        cout<<"String after concatenation : "<<Str2;
        return 0;
}
```

The output of the program

```
Enter your first name : John
Enter your second name : Smith
String after concatenation : Hello! John Smith,
   How are you?
```

In this example, the objects `Fname`, `Sname`, `Name`, `Str1` and `Str2` of `string` type are declared. The values of `Fname` and `Sname` read from the user are concatenated and stored into the third `string` object `Name`. After this, the string literals are concatenated with `string` object `Name` and the resultant concatenated string is displayed using the standard output operator <<.

Comparing and Swapping Strings

The contents of two strings can be compared by using any of the relational operators (==, !=, <, <=, >=, >). The objects of `string` type are compared character by character until either the characters are different or the end of either or both the

strings are reached. When non-matching characters are encountered then their ASCII codes determines which of the strings has the lesser value. If all the characters are same and the strings are of different lengths, the smaller string is 'less than' the longer string. Two strings are said to be equal if they consist the same number of characters as well as all corresponding characters are matching. Since comparison is done on the basis of ASCII codes, the comparison is case-sensitive.

Program 5.26: A program to demonstrate the comparison of strings

```
#include<iostream>
#include<string>
using namespace std;
int main()
{
       //Initiallizing the list of names
       string Names[] = { "John", "Scott", "Julia",
                    "Jeniffer", "Martin" };
       string Str1, Str2;
       int i, flag=0;
       cout<<"List of names : \n";
       for(i=0; i<5; i++)
            cout<<Names[i]<<"\n";
       cout<<"Enter the name to be replaced from
    list: ";
       cin>>Str1;
       for(i=0; i<5; i++)
       {
       //Comparing two strings using relational
    operator '=='
          if(Str1 == Names[i])
          {
             cout<<"Enter the new name to replaced
    with : ";
             cin>>Str2;
             Names[i].swap(Str2); //swapping two
    strings
             flag = 1;
          }
       }
       if(flag)
```

```
    {
        cout<<"List of names after replacement :
";
        for(i=0; i<5; i++)
        {
            cout<<"\n"<<Names[i];
        }
    }
    else
        cout<<"\nEntered name does not exist
in list.";
    return 0;
}
```

The output of the program

```
List of names :
John
Scott
Julia
Jeniffer
Martin
Enter the name to be replaced from list: Julia
Enter the new name to replaced with : Miller
List of names after replacement :
John
Scott
Miller
Jeniffer
Martin
```

In this example, two strings are compared for equality using the relational operator '=='. If the string entered by the user matches with any of the names in the list then that name is swapped with new string using the swap() function of string class.

The two strings can also be compared by using the compare() function of string class. The compare() function returns 0 if two strings are equal, returns a negative value (<0) if first string is less than the second string and returns a positive value (>0) if first string is greater than the second string. For example, consider this code segment.

```
int main()
{
```

```
        //Same as in Program 5.26

        //Comparing two strings using compare() member
    function

        if(Names[i].compare(Str1)==0)

        {

            cout<<"Enter the new name to replaced with
    : ";

            cin>>Str2;

            Names[i].swap(Str2);    //swapping two
    strings

            flag = 1;

        }

}
```

Accessing Characters and Substrings

The individual characters of a string object can be accessed either by using an array notation or by using at() function of the string class. In addition, the substring from a string can be extracted using the substr() function. For example, consider these statements.

```
Name[i];        //accesses ith element of a string
    Name
Name.at(i);     //accesses ith element of a string
    Name
Str1 = Name.substr(4, 6);    //substring of six
    characters from
    //fourth position of string
                        //Name is assigned to
    string Str1
```

   The program given here demonstrates the different ways of accessing individual character of a string. It also shows how the substring from a main string can be extracted and stored into another string.

Program 5.27: A program to demonstrate the accessing of characters of a string

```
#include<iostream>
#include<string>
#include<cctype>
using namespace std;
int main()
{
        string Message = "gIVE hAPPINESS aND iT bEGETS

    hAPPINESS";
```

```
string Sub;
int i;
cout<<"The original string is : "<<Message;
for(i=0; i<Message.length(); i++)
{
        //Accessing individual characters of
a string
        if(islower(Message.at(i)))
            Message.at(i)                =
toupper(Message.at(i));
        else if(isupper(Message.at(i)))
            Message.at(i)                =
tolower(Message.at(i));
}
cout<<"\nString after changes  : "<<Message;
//Retrieving substring from main string
Sub = Message.substr(0, 14);
cout<<"\nSubstring extracted from main
string is :"        <<Sub;
return 0;
}
```

The output of the program

```
The original string is : gIVE hAPPINESS aND iT bEGETS
   hAPPINESS
String after changes  : Give Happiness And It Begets
   Happiness
Substring extracted from main string is : Give
   Happiness
```

In this example, the `at()` function is used to access individual characters of a string and the `substr()` function is used to extract a substring of fourteen characters starting from the zero position of the string message. The extracted substring is stored in another string named `Sub`.

Searching a String

The `string` class provides various functions that can be used for performing search operations on a string. These functions return the position of a substring or of a character to be searched in a string. The substring to be searched can be another object of `string` type or a single character or a null terminated string. To accomplish different search requirements, various forms of the `find()` function are available that are listed in Table 5.11.

Table 5.11 *Different Forms of* `find()` *Member Function*

Function	Purpose
`find()`	returns a position from left where substring is located in main string
`find_first_of()`	returns a position of first occurrence of character in main string
`find_last_of()`	returns a position of last occurrence of character in main string
`rfind()`	searches a string from right

The program given here demonstrates the working of various forms of `find()` function for different searching needs.

Program 5.28: A program to demonstrate the working of different forms of `find()` function

```
#include<iostream>
#include<string>
using namespace std;
int main()
{
      string Message = "Give Happiness And It Begets
                            Happiness";
      string Sub;
      char ch;
      int pos;
      cout<<"The main string is : "<<Message;
      cout<<"\nSize  of  main  string  :
   "<<Message.size();
      cout<<"\n\nEnter a character to be searched
   : ";
      cin>>ch;
      cout<<"Searching using find() function . .
   .";
      pos = Message.find(ch);
      cout<<"\nCharacter "<<ch<<" is present at "
          <<pos<<" position.";

   cout<<"\nSearching using find_first_of() function..
   .";
      pos = Message.find_first_of(ch);
      cout<<"\nCharacter "<<ch<<" is present at "
          <<pos<<" position.";
```

```
cout<<"\nSearching using find_last_of() function ..
.";
   pos = Message.find_last_of(ch);
   cout<<"\nCharacter "<<ch<<" is present at "
      <<pos<<" position.";
   cout<<"\n\nEnter the substring to be searched
: ";
   cin>>Sub;
   cout<<"Searching using find() function . .
.";
   pos = Message.find(Sub);
   cout<<"\nSubstring "<<Sub<<" is present at "
      <<pos<<" position from left.";
   cout<<"\nSearching using rfind() function
. . .";
   pos = Message.rfind(Sub);
   cout<<"\nSubstring "<<Sub<<" is present at
"
      <<pos<<" position from right.";
   return 0;
}
```

The output of the program

```
The main string is : Give Happiness And It Begets
   Happiness
Size of main string : 38
Enter a character to be searched : n
Searching using find() function . . .
Character n is present at 10 position.
Searching using find_first_of() function . . .
Character n is present at 10 position.
Searching using find_last_of() function . . .
Character n is present at 34 position.
Enter the substring to be searched : Happiness
Searching using find() function . . .
Substring Happiness is present at 5 position from
   left.
```

```
Searching using rfind() function . . .
Substring Happiness is present at 29 position from
    right.
```

Modifying a String

After performing various search operations, it might be required to modify a string. The modifications can be in the form of appending, inserting, replacing a substring in a string or deleting characters or substring from a string. These tasks are accomplished by using the functions, `append()`, `insert()`, `replace()` and `erase()`, respectively.

The program given here demonstrates various ways of making changes in a given string.

Program 5.29: A program to demonstrate the working of different functions for modifying the string

```
#include<iostream>
#include<string>
using namespace std;
int main()
{
        string MainStr = "Give Happiness And ";
        cout<<"Original string        : "<<MainStr;
        MainStr.append("And Begets Sadness");
        cout<<"\nUsing append function : "<<MainStr;
        MainStr.erase(18,4);
        cout<<"\nUsing erase function  : "<<MainStr;
        MainStr.insert(19, "It ");
        cout<<"\nUsing insert function : "<<MainStr;
        MainStr.replace(29,3,"Happi");
        cout<<"\nUsing replace function : "<<MainStr;
        return 0;
}
```

The output of the program

```
Original string        : Give Happiness And
Using append function : Give Happiness And And Begets
    Sadness
Using erase function  : Give Happiness And Begets
    Sadness
```

Using insert function : Give Happiness And It Begets
    Sadness

Using replace function : Give Happiness And It Begets
    Happiness

5.7.1 Command-Line Arguments

As we know that C++ supports command-line arguments by which the arguments can be passed to the program while invoking it from the command prompt. They are usually used to pass the name of data file to an application. For example, consider the statement:

```
cc sample.cpp
```

where,

`sample.cpp` = the command-line argument that follows the name of the program `cc` to be executed.

Example 5.3: A program to create a file comparison utility which takes the name of files as arguments from the command prompt

```
#include<fstream>
using namespace std;
int main(int argc, char *argv[])
{
int i;
char string1[100], string2[100];
if (argc!=3)
{
cout<<"Usage: comparefile<file1><file2>"<<endl;
          //comparefile is the name of program
return 1;
}
ifstream infile1(argv[1],ios::in|ios::binary);
if(!infile1)
{
cout<<"Cannot open file "<<argv[1]<<endl;
return 1;
}
ifstream infile2(argv[2],ios::in|ios::binary);
if(!infile2)
{
cout<<"Cannot open file "<<argv[2]<<endl;
return 1;
}
cout<<"Comparing files....." <<endl;
do
{
```

```
infile1.read((char *)string1, sizeof(string1));
    cout<<"\nThe  contents  of  "<<argv[1]<<"  :
"<<endl<<string1;
infile2.read((char *)string2, sizeof(string2));
cout<<"\nThe contents of "<<argv[2]<<" : "<<endl<<string2;
if (infile1.gcount()!=infile2.gcount())
            // comparing size
{
    cout<<"\n\nFiles are of different size"<<endl;
    infile1.close();
    infile2.close();
    return 0;
    }
   else
   {
    for(i=0;i<infile1.gcount();i++) // comparing contents
    if(string1[i]!=string2[i])
    {
       cout<<"\n\nFiles are of same size but ";
       cout<<"Contents are different";
       infile1.close();
       infile2.close();
       return 0;
    }
    }
}while(!infile1.eof() && !infile2.eof());
cout<<"\nFiles are same.";
infile1.close();
infile2.close();
return 0;
}
```

The output of the program

First run

```
C:\>comparefile sample
Usage: comparefile<file1><file2>
```

Second run

```
C:\>comparefile sample sample1
Comparing files.....

The contents of sample :
This is an example of command_line
The contents of sample1 :
This is an example of command line
Files are of same size but Contents are different
```

In Example 5.2, two files namely `sample` and `sample1` are passed as arguments from the command prompt which are accessed using the appropriate pointer `argv[1]` and `argv[2]` respectively. Firstly, the program checks if the number of command line arguments is correct or not using `argc` which represents the total number of command line arguments. If the number of command line arguments does not match, message is displayed. Further, the program checks whether the files can be opened or not.

If the files are opened, the number of characters in both the files is determined using the function `gcount()`. If the size of both files is different, the message is displayed accordingly. Otherwise, the characters are compared, and the message is displayed depending upon whether the contents are same or not.

Check Your Progress

11. What is the basic idea of ADT?

12. Define the term underflow.

13. What are Maps in C++?

14. What is string class?

15. What do you mean by the term string operations?

5.8 ANSWERS TO 'CHECK YOUR PROGRESS'

1. In C++, I/O either with console or other devices, such as Disk Drive is visualized as exchange of stream of bytes between the programs and I/O devices.

2. The `istream class` in the standard library is derived from `ios`. It contains member functions to carry out formatted and unformatted input operations. It contains the overloaded extraction (>>) operator functions.

3. The formatting function defined in `ios_base` are presented in header file `<ios>`.

4. There is another function of class `ios_base` used to specify is known as flags.

5. There are two modes by which data can be transferred to and from streams: text mode and binary mode.

6. In header <fstream>, the disk file I/O operations are declared.

7. In C++, every file is associated with two file pointers, namely, get pointer (input pointer) and put pointer (output pointer).

8. C++ supports command-line arguments by which the arguments can be passed to the program while invoking it from the command prompt. They are usually used to pass the name of data file to an application.

9. The C++ standard library classes, objects and functions are grouped under the namespace called std. There are interesting classes in the C++ standard library under the namespace std. Some of the special classes are:

- String
- Stack
- Vector
- List
- Map

10. Containership is also referred to as nesting in which a class has an object of another class as its member. Containership (or containment or aggregation or composition), like inheritance, enables the implementation of a logical relationship between classes. The class (enclosing class) that contains an object of another class has access only to the public members of that class. The private and protected members of the contained class are not accessible to the enclosing class

11. The basic idea of ADT is that the implementation of the set of operations are written once in the program and the part of program which needs to perform an operation on ADT accomplishes this by invoking the required operation.

12. A situation where the user tries to delete a node from an empty linked list is termed as underflow.

13. Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have same key values.

14. The string class is the most important class of C++ which is used for manipulation of strings. The string class automatically manages memory requirements for strings. One can work with the objects of type string in the same way as one works with variables of the built-in data type.

15. Simple string operations can be performed by using string operators and the advanced operations can be carried out by using the member functions of string class.

5.9   SUMMARY

- In C++, I/O either with console or other devices, such as Disk Drive is visualized as exchange of stream of bytes between the programs and I/O devices.

- Buffer can be visualized as a fast memory device, which can store bytes of data. The buffer provides for temporary storage of the data. For instance, if a program wants to output to a printer, the entire text is placed on the buffer.

- The buffer will in turn transfer the characters to the printer via the output stream. It is more important in the case of disc drives since we cannot read or write one character at a time which will cause a lot of overhead.

- C++ is a general-purpose programming language created by Bjarne Stroustrup as an extension of the C programming language, or 'C with Classes'.

- The `istream` class in the standard library is derived from `ios`. It contains member functions to carry out formatted and unformatted input operations. It contains the overloaded extraction (>>) operator functions.

- The `ostream` class in the standard library implements a mechanism for converting value of any type to a sequence of characters. The `ostream` class contains the overloaded insertion (<<) operator function.

- The `cout` is an ostream object. The `cin` is an istream object. Thus, we were attaching the streams to the console monitor and keyboard. Similarly, we created objects of `ifstream` and `ofstream` and attached to disk drive for file I/O.

- There is another function of class `ios_base` used to specify is known as flags.

- The formatting functions defined in `ios_base` are presented in header file `<ios>`.

- The formatting functions of `istream` and ostream are in their respective header files and through inheritance in `<iostream>`.

- A file stream refers to the flow of data between a program and files.

- There are two modes by which data can be transferred to and from streams: text mode and binary mode.

- In C++, every file is associated with two file pointers, namely, get pointer (input pointer) and put pointer (output pointer).

- Randomly accessing a file means directly reaching the desired record/object in the file. To access an object in a file directly, the put pointer is placed at the beginning of the object by skipping (object_no-1)*object_size number of bytes from the beginning of the file.

- C++ supports command-line arguments by which the arguments can be passed to the program while invoking it from the command prompt. They are usually used to pass the name of data file to an application.

- The language C amply demonstrated that no program is written just using the 24 keywords of the programming language. Always C programs used some library functions, such as <stdio.h>. The C library functions are available for C++ programs in the global namespace with dot h suffix.

- Containership is also referred to as nesting in which a class has an object of another class as its member.

- Containership (or containment or aggregation or composition), like inheritance, enables the implementation of a logical relationship between classes. The class (enclosing class) that contains an object of another class has access only to the public members of that class. The private and protected members of the contained class are not accessible to the enclosing class.

- The basic idea of ADT is that the implementation of the set of operations are written once in the program and the part of program which needs to perform an operation on ADT accomplishes this by invoking the required operation.

- A situation where the user tries to delete a node from an empty linked list is termed as underflow.

- In a singly linked list, each node contains a pointer to the next node and it has no information about its previous node. Thus, we can traverse only in one direction, that is, from beginning to end.

- The string class defined in the <string> facilitates in convenient handling of string objects.

- The string class automatically manages memory requirements for strings.

- The string class consists of constructors and member functions with multiple overloaded forms to create and manipulate string objects.

- Concatenation of a string variable with string literal or string variable can be accomplished with the help of '+' operator.

- The individual characters of a string object can be accessed either by using an array notation or by using at() function of the string class.

- Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have same key values.

- The string class meets all the basic requirements necessary to be a container and hence, it supports common container functions, such as begin(), end() and size().

- Appending, inserting, replacing a substring in a string or deleting characters or substring from a string can be accomplished by using the functions, append(), insert(), replace() and erase(), respectively.

5.10 KEY TERMS

- **Buffer:** It refer to a fast memory device that can store bytes of data.
- **File stream:** It is the flow of data between a program and files.
- **Input stream**: It reads the data from the file and supplies it to the program.
- **Output stream**: It receives data from the program and writes it to the file.
- **Text file:** It refer to a file which is opened in the text mode.
- **Binary file:** It is a file which is opened in the binary mode.

5.11 SELF-ASSESSMENT QUESTIONS AND EXERCISES

Short-Answer Questions

1. What is buffer?
2. Differentiate between `iostream` and `ostream` class.

3. What do you mean by the term precision?

4. How is the end of file represented in text and binary files?

5. Differentiate between ios::ate and ios::app mode.

6. Differentiate between character array and C++ string.

7. Define the term vector.

8. Differentiate between containment and inheritance.

9. What is a linked list?

10. Write in brief about the structure of the node of a singly linked list.

11. Define doubly linked list.

12. State the need for including string class in C++.

13. Differentiate between old and modern C++.

Long-Answer Questions

1. Explain the C++ streams with the help of diagram.

2. Discuss the insertions and extractions function with the help of diagram.

3. Explain how to manage output with manipulators.

4. Describe the modes by which data can be transferred to and from streams.

5. Explain the stream classes commonly used for disk I/O operations with the help of diagram.

6. Differentiate between opening a file using constructor function and `open()` function. What method is preferred to handle multiple files using one stream and why?

7. Describe the manipulations of file pointer with appropriate examples.

8. Write a program to create a file comparison utility which takes the name of files as arguments from the command prompt.

9. Describe how different standard library functions pertaining to strings are declared.

10. Discuss the concept of containership and inheritance with the help of diagram.

11. Differentiate between the linked list and array implementation.

12. Consider a linked list containing integer values and write an algorithm to find the average MEAN of the values.

13. Write an algorithm to create a linked list storing the names, age and salaries of ten employees. Arrange the list in the descending order of salaries.

14. Explain the different ways in which concatenation of two or more string objects can be done with the help of examples.

15. Discuss the functions used in map with appropriate examples.

5.12 FURTHER READING

Jeyapoovan, T. 2006. *Computer Programming: Theory and Practice* (with CD). New Delhi: Vikas Publishing House.

Khurana, Rohit. 2008. *Object Oriented Programming with C++*. New Delhi: Vikas Publishing House.

Saxena, Sanjay. 2009. *Introduction to Information Technology.* New Delhi: Vikas Publishing House.

Rumbaugh, James, Fedrick Blaha, William Premerlani, and Federick Eddy.1990. *Object- Oriented Modelling and Design.* New Jersey: Prentice Hall.

Balaguruswamy, E. 1998. *Object-Oriented Programming*. New Delhi: Tata McGraw-Hill.