OBJECT ORIENTED PROGRAMMING USING C++

BCA - 304

Dr. N.P. Singh
Dean

Directorate of Distance Education
ਪੰਜਾਬ ਟੈਕਨੀਕਲ ਯੂਨੀਵਰਸਿਟੀ ਜਲੰਧਰ
PTU Punjab Technical University
Estd. Under Punjab Technical University Act, 1996 (Punjab Act No. 1 of 1997)

Preface

The Punjab Government established Punjab Technical University (PTU) in 1997 by an act of State Legislative. The University was entrusted with the responsibility of developing the new generation of technical manpower that can spearhead the industrial development of the State. Punjab Technical University has been envisaged to be the grooming ground for the future Engineers, Managers and Researchers.

As of today, PTU affiliates more than 300 Engineering, Management, Pharmacy, Hotel Management and Architecture institutions in the State that are approved by All India Council of Technical Education (AICTE).

PTU understands that restricting technical education to its campuses will not serve its objective of effective spreading of knowledge in the society. It is firmly understood that latest technical education has to be spread to the masses in every corner of the nation. This is how the Distance Education Programme (DEP) of the Punjab Technical University was conceived.

The objectives of the programme are to impart affordable, relevant, skill-based & remunerative technical education to the masses in the different corner of the country.

Today, the University has more than 2000 Learning Centres spread across the country offering quality technical education in the fields of Information Technology and Management, Paramedical Technology, Fashion Technology, Hotel Management and Tourism, Media and Mass Communication and Journalism etc.

The main purpose of this book is to impart the student an insight into the subject, explaining the complexities involved, in a simplified manner and helping them to achieve their academic goals.

For an easier navigation and understanding, this book contains the complete PTU curriculum of this subject and the topics. The various topics are dividing into Chapters, Units & Sub-Units and sufficient space is provided for students to make their brief notes.

This book encompasses a global approach for providing the simplified study material to both working as well as non-working students and is certain to get benefitted from the efforts of the authors of this book.

[illegible]

Dr.N.P. Singh
Dean (Distance Education Programme)

"Propelling Punjab to a Prosperous Knowledge Society"

Punjab Technical University Jalandhar
Jalandhar-Kapurthala Highway, Near Science City, Kapurthala- 144601
Phones : 01822-662502 Fax : 01822-255532
Website : www.ptu.ac.in email : deandep@ptu.ac.in, singhnp59@gmail.com

OBJECT ORIENTED PROGRAMMING USING C++

BCA - 304

Reviewer

Dr. N. Ch. S.N. Iyengar	Senior Professor, School of Computing Sciences, VIT University, Vellore

Authors: R. Subburaj (Units: 1.0-1.3, 1.8-1.11, 2.0-2.4, 2.6-2.11, Unit 3, 4.0-4.3, 4.5-4.10, Unit 5) Copyright © Reserved, 2008
Maria Litvin, Gary Litvin & K S Easwarakumar (Units: 1.4-1.7, 2.5, 4.4) Copyright © Reserved, 2008

Reprint 2010

VIKAS® PUBLISHING HOUSE PVT LTD
E-28, Sector-8, Noida - 201301 (UP)
Phone: 0120-4078900 • Fax: 0120-4078999
Regd. Office: 576, Masjid Road, Jangpura, New Delhi 110 014
• Website: www.vikaspublishing.com • Email: helpline@vikaspublishing.com

CAREER OPPORTUNITIES

Object oriented programming in C++ language is currently the most widely used programming language. This is due to the language's flexible features which enable programmers to write for numerical, commercial and graphical applications with equal ease. Further, the C++ language is preferred by engineers to write programs for lower level accessing along with the assembly language. Students who master this language will find attractive career opportunities in the IT sector, where the demand for programmers even now is increasing.

PTU DEP SYLLABI-BOOK MAPPING TABLE

Object Oriented Programming Using C++

Syllabi	Mapping in Book
SECTION I **Introduction:** Object Oriented Programming, Characteristics of Object Orientated Languages, Classes, C++ Basics: Program Statements, Variables and Constants, Loops and Decisions.	**Unit 1:** Introduction to Object Oriented Programming **(Pages 3-83)**
Functions: Defining a Function, Function Arguments & Passing by Value, Arrays & Pointers, Function & Strings, Functions & Structures.	**Unit 2:** Functions **(Pages 85-111)**
SECTION II **Classes & Objects:** Defining Class, Class Constructors and Destructors, Operator Overloading.	**Unit 3:** Classes and Objects **(Pages 113-159)**
Class Inheritance: Derived Class & Base Class; Virtual, Friends and Static Functions; Multiple Inheritance, Polymorphism.	**Unit 4:** Class Inheritance **(Pages 161-214)**
SECTION III **Input/Output Files:** Streams, Buffers & iostreams, Header Files, Redirection, File Input and Output.	**Unit 5:** Input/Output Files **(Pages 215-266)**

CONTENTS

INTRODUCTION

NOTES

Object Oriented Programming Systems (OOPS) are widely used in software development projects. C++ is one of the early OOPS. Using OOPS for program development reduces programming errors and improves their quality. Representing objects of the real world as software objects in OOPS is quite a natural way for program development. The introductory section brings out the characteristics of OOPS and, in particular, the three basic characteristics, which are:

- Encapsulation
- Inheritance
- Polymorphism

Since C++ is an extension of C language, a programmer who is knowledgeable in C will find it easy to understand the basic concepts such as variables, constants, loops and decision constructs such as `if..else` discussed in Unit 1.

Unit 2 deals with functions, arrays, pointers as well as functions with strings and structures. These have been discussed in detail and complemented with examples.

Unit 3 introduces classes and objects, the basic building blocks of OOPS. Operator overloading enables the carrying out of complex tasks by overriding basic operators such as +, -, * and /. This is an interesting concept in C++ wherein the reader of the program readily understands the basic purpose of the function.

Inheritance is another important feature of OOPS which enables reusability of programs that are already developed and tested, thus ensuring high quality of the programs. Virtual functions allow programmers to declare functions in the base class and redefine them to suit specific needs in each of the derived classes with the same name. Friend functions provide the members of one class access to another. However, this may go against the principle of OOP; namely data security. This aspect is discussed in Unit 4. Another interesting characteristic of OOPS is polymorphism by which functions with the same names can be used to handle different data types.

Unit 5 discusses the input and output functions with files. The C++ constructs for input/output of data are flexible and easy to program.

The entire text and programs are based on the latest ANSI/ISO standard C++. The text has been prepared with the objective of enabling learning by programming. Executing each program will enable the programmer to master the concepts of C++ in a step-by-step manner. This makes the book suitable for students of distance learning programmes.

UNIT 1 INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

NOTES

Structure

1.0 INTRODUCTION

In this unit, you will learn the principles of object oriented programming using C++ and the basics of data structures. C++ is different from other high-level languages such as FORTRAN, Pascal, BASIC, C and COBOL, because object oriented programming combines both data and the associated functions into a single unit called a class, whereas in other high-level languages the programs execute a sequence of instructions or procedures. Bjarne Stroustrup of AT&T Laboratories is the architect of the C++ language, which was first known as 'C with classes'. The name C++ (pronounced as C plus plus) was coined by Rick Mascitti in 1983. The class concept was derived from Simula 67.

In this unit you will also learn that in programming an object means data, hence data is a predominant factor in object oriented programming. Further, in the object oriented paradigm, accessibility of data is restricted and, therefore, accidental corruption of data is minimized or eliminated. The nucleus, i.e., data can only be accessed through member functions, which are nothing but interfaces to access or manipulate data. The data is thus well protected and its inadvertent manipulation can be easily prevented. The interface or function is the window to the outside world for manipulating the data and ensuring its integrity. You will learn the advantages of object oriented programming, i.e., data abstraction, reusability, maintainability, natural and modular concept, extensibility and data integrity and its characteristics, i.e., encapsulation, inheritance and polymorphism. C++ is therefore one of the few languages which supports both function oriented programming and object oriented programming.

1.1 UNIT OBJECTIVES

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

- Understand the concept of object oriented programming
- Explain the characteristics of object oriented languages

- Define classes
- Understand the basics of C++ programming
- Identify program statements
- Able to write a simple C++ program
- Differentiate between variables and constants
- Describe different types of loop structures in a program
- Explain the decision making statements in a program

NOTES

1.2 OBJECT ORIENTED PROGRAMMING

High-level programming languages such as FORTRAN, Pascal, BASIC, C and COBOL are known as function oriented or procedure oriented languages. They have been used for developing important and mission critical applications. In these languages, computer programs are organized as a sequence of instructions. In function oriented programming, the emphasis is on procedures or instructions and data integrity does not get due importance. In large software projects, a number of programmers will be involved in programming. Each one will develop programs for some portion of the product. An inexperienced programmer might spoil the integrity of some data items inadvertently due to poor understanding. This is due to the inherent design of the function oriented programming languages where data and function are not tied to each other closely. They are independent of each other causing major problems while the software is put to use.

The principle of object oriented programming is to combine both data and the associated functions into a single unit called a class. An object in programming means data. Data is therefore predominant in object oriented programming. The concept will become clear with examples. However, in the object oriented paradigm, accessibility of data is restricted and therefore, accidental corruption of data is minimized or eliminated. Thus, OOP ensures integrity of data.

Evolution of C++

Bjarne Stroustrup of AT&T Laboratories is the architect of C++ language. His book, *The C++ Programming Language* details the history of C++. In the 1980s, the language was known as 'C with classes'. Bjarne Stroustrup originally developed the extended language to write some event-driven simulations. 'C with classes' was used in major simulation projects. The name C++ (pronounced as C plus plus) was coined by Rick Mascitti in the year 1983. The author's main objective in designing this new language was to write successfull and good programs easier than in assemblers, C, or other high-level languages. According to Bjarne Stroustrup, C++ owes most to C. C is retained as a subset of C++. C in turn owes much to its predecessor BPCL. The class concept was derived from Simula 67.

The other objectives in the design of C++ by Bjarne Stroustrup were:

- Better C
- Support object oriented programming
- Support data abstraction

Further C++ is one of the few languages which support both function oriented programming and object oriented programming.

ANSI/ISO standard for C++

C++ has evolved through the continuous participation of users. After more than a decade of usage, Bjarne Stroustrup formulated the C++ reference manual, which led the American National Standards Institute (ANSI) to initiate development of a standard for C++. The international standard for the C++ programming language called ISO / IEC 14882 was released in the year 1998. (ISO stands for International Organization for Standardization and IEC stands for International Electro-technical Commission.) This marked a remarkable milestone. Since 1991, both ANSI and ISO were working together in the standardization of C++. The evolution of Stroustrup's C++ language as an international standard has given a new dimension to the language itself. Standard C++ is a far more powerful and polished language than C++ introduced by him.

Features added by the standard

The new features in standard C++ (whenever the language is referred to as standard C++, it means the C++ language with specifications in accordance with ISO/IEC standard 14882) include the following:

- Namespaces
- Exceptions
- Templates
- Runtime type identification
- Standard library including Standard Template Library (STL)

The standard has led to compatibility amongst C++ compilers and enhanced portability of C++ programs across platforms since every compiler has a common reference in the standard. Bjarne Stroustrup chose C as the base language for C++ because of the following reasons:

- Versatility and suitability to interact with hardware
- Adequacy for most system programming tasks
- Portability across platforms
- Adaptability to Unix programming environment

Furthermore, millions of lines of codes of library functions and utility software written in C can be used in C++ programs. C programmers need to learn only the new features of C++, instead of starting from the basics. Thus, C++ closely follows ANSI C.

Definition of 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
- Behavior 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

NOTES

- 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 as given below. (see Fig. 1.1.)

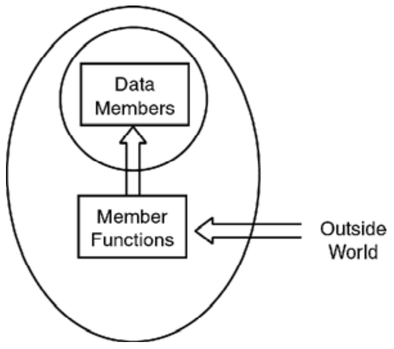


Figure 1.1 *Basic representation of an object*

An object definition consists of the following:

- Data members
- Member functions

The data members establish the state or attributes for the object. The behavior 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. Object Oriented Programming (OOP) is all about creating useful programs with the help of objects.

Abstractions

When we look at a television set, we identify it as an object with a specific state and a well-defined behaviour. We recognize it as one whole object. We do not attempt to see smaller objects inside it, although it may contain a number of components or objects. In a similar manner, we look at a building as one whole object. This is known as abstraction. The advantage of abstraction is that we are able to understand an object's overall characteristics without looking at other diversionary details.

The object in turn may have a number of distinct objects with unique states and behavior. A television set has a picture tube and speakers. Each one is in turn an object, with corresponding state and behaviour. The speaker can also be divided into smaller components or objects. The advantage of abstraction is that one can use an object without knowing all its details. For instance, a normal user can switch on the TV set or switch from one channel to another, without knowing its complete technical details. An experienced person can start doing these operations without much learning effort. Of course, a technician can make a schematic diagram of a TV receiver. Each block can thus further be divided. This is known as hierarchical classification or hierarchical abstraction. This helps the user to use an object with minimum information, i.e., just enough information about how to interact through the external interface. Object oriented programming is based on the concept of hierarchical abstraction. The complete program can be considered as one object. This object in turn consists of a number of small objects. Each small object consists of smaller objects. Such division can go on till we reach a basic or fundamental object, which cannot be further subdivided. This makes life easy. For instance, a person driving a car need not know all the parts in it, in order to drive. He can easily drive by using the external interfaces, such as steering wheel, accelerator, brake and clutch.

Data abstraction

In object oriented programming, each object will have external interfaces through which it can be made use of. There is no need to look into its inner details. The object itself may be made of many smaller objects again with proper interfaces. The user needs to know the external interfaces only to make use of an object. The internal details of the objects are hidden which makes them abstract. The technique of hiding internal details in an object is called data abstraction.

Reusability

In a TV set, if we want to change the speaker, the requirements or specification for interfacing the speaker with the TV receiver should be known in order to replace the existing speaker. Similarly, the removed speaker can be used in some other set. Thus, the concept of object orientation facilitates reusability, just by knowing interface specifications, without much further testing. Object oriented programs provide similar facility. Tested objects developed for one program can be used elsewhere without much difficulty. Thus, reusability is an important feature of OOP.

Sample program

The tool for learning programming in C++ is an Integrated Development Environment (IDE) that implements ANSI/ISO C++. If we use any one of them, we will find that everything needed for program development such as text editor, compiler, linker and run time system are available in one package. Learn from the system administrator about how to enter the program, compile and run it.

Let us now write the first program in C++. The program has to be entered in text editor. Let us type in the following program in the text editor available in our system.

Program 1.1

```
// Example E1x1. CPP
#include<iostream>
class E1x1
{
};
int main(){
    std::cout<<‘Om Vinayaga’;
}
```

After typing in the program, we have to save it in a file name. Let us choose the file name the same as the name of the class although there is no such requirement in C++. In this example, the name of the class is E1x1. Therefore, the file may be saved as E1x1.cpp. The file in which we have stored the above program is called source file. The program is called source code.

The next two steps are compilation and execution.

NOTES

Compilation To compile a program we have to call the compiler in the IDE or the system we are using. Then the application can be compiled. The C++ compiler checks for any error in the program. If there are errors, it will give a message listing the type of error and the line number in the program. We can edit the program, correct the errors and recompile using the same command, till there are no error messages. If we have typed the program E1x1.cpp correctly, we will not get any error. The resultant file after compilation will be saved as an object file. The object file consists of the above program converted into machine code. The system itself will save the object code in a file named E1x1.o.

Execution Now, build an executable version of the program by following the instructions of the IDE. The executable code will be saved in the file E1x1.exe. Even at this stage we may get errors. If so, they should be checked and corrected.

Note: The process of writing the program, compilation and execution is similar to C programs. The program file may be saved with .cpp extension or .cc extension as required in the IDE we are using. We can execute it by typing in the following in the DOS Prompt.

```
E1x1
```

Now we will get the result of execution as given below:

Result of Program 1.1

```
Om Vinayaga
```

To summarize, the code typed in and saved as E1x1.cpp is called source code. The file after compilation will be saved as object code in file E1x1.o. This will be in machine code. The executable code is saved in file E1x1.exe.

Analysis of the sample program Let us now take a closer look at the program E1x1.cpp. The first line is as given below :

```
//Example E1x1.cpp
```

The two backslashes in the beginning of the line indicates that it is a comment statement. The compiler will ignore them at the time of compilation. They are used for explaining the logic of the program and other relevant information. At a later point of time, when the developer reviews the program, he may not be able to recall why a particular statement was included. Also a programmer other than the developer might maintain the program. Therefore, it is a good programming practice to include adequate comment statements to explain the program. In this case, whatever is written after the two backslashes will be treated as a comment. The comment, in this case, can start anywhere in the line but should end in the end of the line. There is also another way of writing multiple comment lines. It is illustrated below.

```
/*We start programming by praying to Lord Vinayaga*/
```

Whatever is written between slash start (`/*`) till the corresponding closing `*/` (even if it is not in the same line) will be treated as comments. Now we will discuss the next statement.

```
#include<iostream>
```

The `#include` is a preprocessor directive. It is a direction to the compiler to include the program contained in the file name surrounded by angle brackets in the current program file, before compiling it. Angle bracket is an indication to the compiler that `iostream` is a header file. Some header files have a `.h` notation. Therefore, by this statement we are going to include contents of another file in the program.

We will see what is a namespace later. The std is a namespace in the C++ standard library. The declaration `std::cout`, refers to the object `cout` in the `std namespace` of standard library of C++. The reader will be comfortable with these words a little later. We include `<iostream>` because it contains definitions for the object `cout` and << operator. In other words, if we don't include the header file, the compiler will not recognize the operator << and the identifier `cout`. Therefore, in any program involving input and output with console we must include `<iostream>` so that we can carry out input and output easily. The file `<iostream>` is also in the std namespace.

The header file iostream is similar to `stdio.h`. In fact, we could use the latter in C++ programs also and in which case we have to use `printf()` instead of `cout`. Since `cout` is more convenient and appropriate, we will use it in C++ programs. Now let us look at the next statement or line.

```
    class E1x1
```

The class is a user-defined data type. Although C++ programs can be written procedure oriented without classes, this program has been written in an object oriented manner. In this program, the `class E1x1` does not contain any data or function and hence does not serve any useful purpose. We can even omit it without any harm. But then why is it included? Only to illustrate how a typical C++ program will look like. Note the semicolon at the end of the class definition. The class name generally reflects what is proposed to be done in the program. If a class is written for Bank Account it can be written as follows :

```
    class BankAccount
```

What we notice is that there is neither a space between the two words nor an underscore. The upper case letter highlights the second word. This style is common in Java and C# programs. Remember that `class` is a keyword and hence will be written in lower case letters.

Keywords are reserved words of the C++ language. The keywords are discussed in the in a later section in this unit. A class while defining implementation of an object may declare variables and functions. All these will be written between opening and closing braces. The semicolon after the closing brace indicates the end of the class definition. The next statement is the place from which program execution starts.

```
    int main()
```

Every C++ program will contain a `main` function from where the execution of every C++ program starts.

```
    int
```

Functions in C++ are similar to functions in C. The functions may return a data type or nothing. If it returns an integer at the end of execution, we may specify the return data type as `int`. Thus, the main function returns an integer in this program.

```
    main()
```

It is not a C++ keyword, but a C++ function to be present in all programs. It is the starting point for program execution. C++ compiler compiles all classes and stores them in object files. But if an application does not contain main function, then the program cannot start execution. Since empty parentheses is specified the main function does not receive any values or arguments.

The main function contains only one statement, which is discussed below:

```
    std::cout<<'Om Vinayaga';
```

`std::cout` is a built-in object provided by standard C++. It is used for outputing to the standard output device. This object is created from the iostream class of standard C++. The iostream class is used to create objects for performing input and output operations. The language provides each program with several built-in iostream objects for input and output with the console. Note the semicolon at the end of the statement. All statements except for function definition will terminate with a semicolon.

The above statement results in printing of the string given within quotes namely 'Om Vinayaga'. The identifier `cout` (pronounced as c out) denotes an object. It points to the standard output device namely the console monitor. The operator << is called insertion operator. It directs the string on its right to the object on its left. Here it is `cout`. Therefore, the object `cout` directs the string to the standard output device namely the console monitor. The symbol `(::)` is called **scope resolution operator**. The std refers to the namespace in the standard library of C++. The prefix of std using scope resolution operator refers to the `cout` in the namespace std. This is similar to referring to this book as Subburaj::C++.

The task of the insertion operator is to direct the strings and other variables on its right to the object on its left. In this case, we are passing only a string literal or constant. We can also pass variables. In such cases it will display on the monitor, the current value of the variable.

Note that `class E1x1` and main function are enclosed within a pair of braces.

Effect of standard C++

The programs written before 1998 are to be modified suitably to execute them in the compilers implementing standard C++, since there are minor differences. The header files contain definition of library classes and functions. By including the header files in the program, we can call and use the library classes and functions. In the earlier versions of C++, the header files such as iostream were to be included in the program file with dot h extension. But now the standard library of C++ contains C++ library files. They are placed in `namespace std` in header files without dot h suffix. The new compilers therefore will not accept dot h extension for the header files such as iostream in the C++ standard library. Other header files such as those in C library functions place their declarations in the global namespace. The latter headers are to be used with .h suffix. Readers are advised to check the convention for header files in the compiler they are using. In a truly compatible standard C++, the header files in the standard library have to be included without suffix as given below.

```
<iostream>
```

Similarly, the objects `cin` and `cout` were used without reference to any standard library. Now, it is necessary that we call `cin` and `cout` as given below:

```
std::cin

std::cout
```

We can avoid prefixing std each time with the input and output objects `cin` and `cout` by declaring the namespace std as given below at the beginning of the program.

```
using namespace std ;
```

Another notable difference between the earlier version and the standard version is that the return data type of the main function has to be integer. In the previous versions also we could specify integer as the return data type but in such cases, we had to return

0 for normal exit from the main function. In the standard C++, return 0 is not mandatory. In the earlier versions, we could also specify void as the return data type for the main function. But that will not work with the standard C++. The main function returns 0, if specified, on successful exit from the main function. If there is no return statement in the standard C++ program, the system will still receive a value indicating successful exit from the main function. If the main function returns a non-zero value, it indicates that the exit from main was not normal.

NOTES

Standard C++ compatible compilers

The standard library shall be a part of every C++ compiler compatible with the standard. In addition, some compilers may offer Graphical User Interface (GUI) systems.

There could always be a gap between the requirements of the standard and compiler implementation. However, the reader should be careful to use a standard-compatible compiler. A freeware compiler for C and C++ from an organization called The Free Software Foundation is available. It is called GNU g++. It can be downloaded from the following URLs:

http://www.gnu.ai.mit.edu/software/gcc (for UNIX)

http://www.delorie.com/djgpp/ (for PC)

Some other C++ compilers are: Microsoft Visual C++ as part of their Visual Studio.net and Borland C++ builder 6.0.

There may be many other sources. The best place to search for the sources of standard C++ compiler is the Internet. However, the reader should find a suitable standard compatible compiler for executing the programs given in this book.

If the implementation of the compiler does not support namespace, the above program can be made to work by writing it in the same way as Program 1.1A

Program 1.1A

```
#include<iostream.h>
class E1x1
{
};
int main()
{
    cout<<'Om Vinayaga';
    return 0;
}
```

return 0 The purpose of this statement is to terminate the program after executing all the instructions in the main function. When the execution is terminated in the normal manner or otherwise, a value known as exit status will be returned to with the system. If the execution is normal, the program function returns a value of 0. This is called zero exit status indicating that the job has been completed as planned. Zero exit status indicates that the program execution was successful.

From the above experimentation, the set of statements, which work with the compiler used, may be identified. This may be followed in future programs. However, the book contains programs which are compatible with the standard.

Now let us look at some more programs to assimilate the concepts discussed above.

Program 1.2

```
#include<iostream>
int main(){
    std::cout<<'Om Vinayaga '<<'we worship you';
}
```

The above example illustrates that a C++ program can be written without a class, similar to C programs. But every program shall have a main function as illustrated. In the main function, we have one `cout` object but two insertion operators. We can have any number of insertion operators, with one `cout` object. The insertion operator is also called **put to operator**. The program can now be compiled and then run or executed. The result of execution of the program is shown below.

Result of Program 1.2

```
Om Vinayaga we worship you
```

We have actually passed the following two values or arguments to the `cout` object, one after another:

1. 'Om Vinayaga '
2. 'we worship you'

Both are string constants. The strings were printed one after the other in the same line, as the result of the program indicates.

There are no return statements in the main function, but still the program worked. In the earlier versions of C++, it was mandatory to return 0 before the program exited the main function. However, in the standard C++, it is not required. In the standard C++, on successful exit from main, the system will return some value. That is why there is no harm in adding a statement as given below in the program:

```
return 0;
```

We will insert this statement in the next program.

We now try to modify the program to print the strings one after another in separate lines. We might feel that if the strings are directed to `cout` objects individually, one at a time, they might be printed in separate lines. Let us see whether it works! The modified program is given in Program 1.3 below.

Program 1.3

```
#include<iostream>
int main(){
    std::cout<<'Om Vinayaga ';
    std::cout<<'we worship you';
    return 0;
}
```

Here, we have two statements where the insertion operators are directing one string each to the console monitor. Let us execute and see the result.

Result of Program 1.3

```
Om Vinayaga we worship you
```

Again the output appears on the same line. Why? Before the first print occurs, the cursor is at column 1 on the monitor. Now, before it encounters the second print, the cursor has not moved to the first column of the next line but remains where it was after the first print. Then, when it gets the second string, it continues to print from where it was. That is the reason why both the strings are printed on the same line. Therefore the programs 1.2 and 1.3 are identical in functioning. The print statements produce the same results although the number of cout objects varies. In fact, any number of strings can be printed one after another by cascading the strings with the insertion operators. Incidentally, including the statement, return 0, did not cause any problem. However we will omit it in future since it is not essential.

We actually need the second string to be printed in the new line. Therefore, we have to pass a newline character `'\n'` after the first string. The newline character is called escape sequence. Escape sequences start with `'\'` as in C language. The example below prints the strings in different lines.

Program 1.4

```
#include<iostream>
int main()
{
    std::cout<<'Om Vinayaga '<<'\n'<<'we worship you';
}
```

Result of Program 1.4

```
Om Vinayaga
we worship you
```

Escape sequences

Adding '\' gives a different meaning to the character. The escape sequences and their interpretations are given below:

Escape sequence	ASCII name	Purpose
\n	NL (LF)	newline
\b	BS	backspace
\f	FF	form feed
\r	CR	carriage return
\t	HT	horizontal tab
\v	VT	vertical tab
\a	BEL	alert
\’	‘	single quote
\’	‘	double quote

They are the same as in C, but are mentioned here for quick reference. The escape sequences can be used to get the printout in the desired manner.

1.3 CHARACTERISTICS OF OBJECT ORIENTED LANGUAGES

NOTES

Characteristics of OOP

Let us look at the three essential characteristics an object oriented program should support. They are listed below.

- Encapsulation
- Inheritance
- Polymorphism

We will make an attempt to understand the above from a logical point of view.

Encapsulation

The example of a TV set is often taken to illustrate objects and characteristics of OOP. The TV receiver is totally encapsulated. We cannot see the components inside or meddle with them. However, we can use the TV set easily. We can switch it on and select channels with the external interfaces provided in the receiver. This is the concept of encapsulation of objects in an OOP.

Wrapping together data and functions creates the objects in OOP. They are bound together. This represents encapsulation in OOP. We can use the encapsulated objects through the designated interfaces only. Thus, the inner parts of the program are sealed or encapsulated to protect from accidental tampering. This feature is not available in the conventional procedure oriented programming languages where the data can be corrupted since it is easily accessible. In C++, like other object oriented programming languages such as Java and C#, encapsulation is achieved through what is known as classes.

Class A class is a blueprint for making a particular kind of object. It defines the specifications for constructing an object with data and functions. It generally specifies the private (internal) working of objects and their public interfaces. The data, known as data member(s), defines the state of the proposed object and function of its behaviour. We will discuss about classes in more detail in later units.

In C++, class structure is used for declaring two types of members as given below:

- Data
- Functions

Data are nothing but declaration of variables for holding data. Functions, called member functions, contain sequence of instructions that operate on the data. 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 as shown in Fig. 1.2 below.

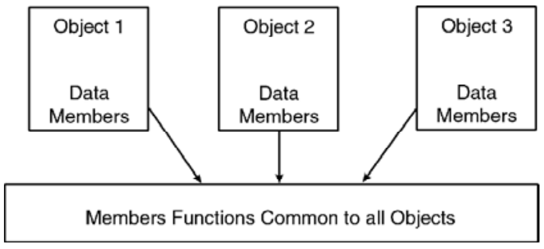


Figure 1.2 *Schematic representation of objects*

Note carefully that a class can give rise to a number of objects, but is not an object on its own. It is only a structure or blueprint which helps in creating replicas with common structures, but different characteristics. This means, 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, 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. Although we will discuss classes in detail later on, let us see a simple example to get a feel about classes and objects.

NOTES

A *class* is an expanded concept of a data structure like integers, floating point numbers, arrays, etc. The data structures mentioned above hold only data, whereas a class can hold both data and functions.

An *object* is an instantiation of a class. In terms of variables, a class would be the type like int, double, etc., and an object would be like a variable of the given class. While int is a universal class of integers, each class is of a specific type. We can have a class called Transport. In that case, a car of type Maruti is a variable of the class Transport. Maruti is in fact an object instantiated from the class Transport.

Classes are generally declared using the keyword class, in the following format:

```
class class_name {
     access_specifier_1:
       member1;
     access_specifier_2:
       member2;
     ...
     } object_names;
```

Where class_name can be any valid identifier for the class, object_names is an optional list of names for objects of this class. The class body may contain members, which can be either data or function declarations.

Thus it is similar to the declaration of any data structure, except that we can now also include functions as members of the data structure class. In object oriented programming, we can control access to the data members and member functions through what is known as *access specifiers*. Each member, either data member or member function, can have individual access specifiers. An access specifier is one of the following three keywords: private, public or protected. These specifiers modify the access rights that the respective members get.

- Private members of a class are accessible only from within other members of the same class or from their *friends*.
- Protected members are accessible from members of the same class and from their friends, but also from members of their derived classes.
- Finally, public members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members. For example:

```
  class Transport {
      int x, y;
  string name;
    public:
                              void set_values (int,int, string);
                                } Maruti;
```

NOTES

Declare a class (i.e., a type) called Transport and an object (i.e., a variable) of this class called Maruti. This class contains four members as listed below:

- Two data members of type int (members *x* and *y*)
- One data member of type string
- One member set_values()

All three data members have private access because private is the default access level. We have only included declaration of the member function set_values but not its definition. This function is allowed public access through the keyword public. Notice the difference between the class name and the object name. In the previous example, Transport is the class name (i.e., the type), whereas Maruti is an object of type Transport. It is the same relationship int and var have in the following declaration:

```
int  var;
```

where int is the type name (the class) and var is the variable name (the object). But the difference between the variable var and Maruti are as follows:

- var contains only data and no member functions or control over access to members.
- Maruti, a variable of type Transport, has both data members and member functions. Also, access to members is controlled.

Thus, in OOP, both data and functions are tied together or encapsulated. The operations are usually given in the form of functions operating upon the data items. Such functions are also called *methods* in certain object-oriented programming languages. The data items and the functions that manipulate them are combined into a structure called a *class*. A class is an abstract data type.

Access to members is appropriately controlled to enhance data integrity. Thus, encapsulation is the process of combining data and functions into a single unit called class. Using the method of encapsulation, a programmer cannot directly access data; it is only accessible through the functions present inside the class. Data encapsulation led to the important concept of data hiding, which involves hiding the implementation details of a class from the user. The concept of restricted access led programmers to write specialized functions or methods for performing operations on hidden members of a class. Thus, encapsulation is a very important feature of OOP.

When using C++, a programmer has the facility to encapsulate data, and the operations or functions that manipulate that data, in an object. This enables the programmer to use collections of data and functions, called *objects,* in programs other than those originally created. This is a simple form of encapsulation. Objects carry encapsulation a step further. With objects, a programmer can define not only the way a function operates, or its *implementation*, but also the way an object can be accessed, or its *interface.* The programmer can specify access differently for different entities. For example, he could make the function setdata() contained inside Object A accessible to Object B but not to Object C. This access qualification can also be used for data members inside an object. The encapsulation of data and the intended operations on them prevents the data from being subjected to operations not meant for them. This is what really makes objects reusable and portable!

NOTES

Data hiding

Related to the idea of encapsulation is the concept of data hiding. Encapsulation can hide data from classes and functions in other classes. In the example of the class *Transport*, we can access the three data members through the member function set_values(). Although there may be many other attributes and data elements in the class *Transport*, these are all hidden from view. This makes programs more reliable, since declaring a specific interface, such as public or private, to an object prevents inadvertent access to data in ways that it was not designed for. In C++, access to an object and its encapsulated data and functions is treated very carefully using the keywords private, protected and public. The programmer has the facility to decide the access specifications for data objects and functions as being private, protected or public while defining a class. Only when the declaration is 'public' can other functions and objects access the object and its components without question. On the other hand, if the declaration happens to be 'private', there is no possibility of such access. When the declaration given is 'protected', access to data and functions in a class by others is not as free as when it is public, nor as restricted as when it is private. You can declare one base class as being derived from another, which will be discussed shortly. Derived classes and their corresponding base class, however, have access to the components of objects that are declared 'protected'. The three types of declarations of access specifications can be different for different components of an object; for example, some data items can be declared as public, some as private and the others as protected. The same situation can occur with the functions in an object. When no explicit declaration is made, the default specification is private.

Inheritance

A software object does not exist by itself; it is always an instance of a class. A class gives a blueprint for building a software object and can inherit the features of another class and add its own modifications. Inheritance is similar to human beings. A child can inherit his/her father's property and add/acquire more properties; s/he can modify the inherited property; or dispose of a property. In a similar manner, a new class can inherit its properties from another existing class.

A class can have a 'child', which means one class can be derived from another. The original or parent class is known as the base class and the child class is known as the derived class, as illustrated in Figure 1.3.

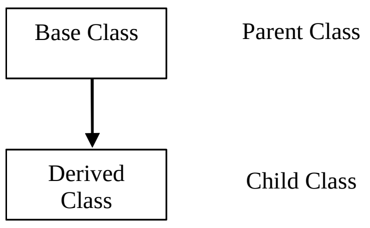


Figure 1.3 *Parent and Child classes*

A derived class inherits all the attributes and behaviour of the base class and may have additional ones as well. The behaviour of a class is represented by the operations it can perform. The attributes and operations for base class Transport are shown in Figure 1.4.

NOTES

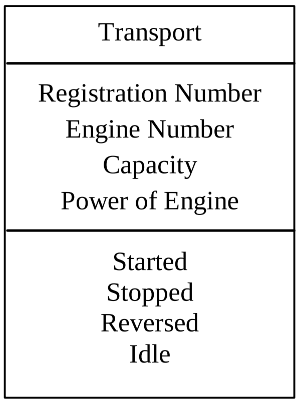


Fig 1.4 *Attributes and Operations of base class* Transport

Transport can have two derived classes – Surface Transport and Air Transport, as illustrated in Figure 1.5.

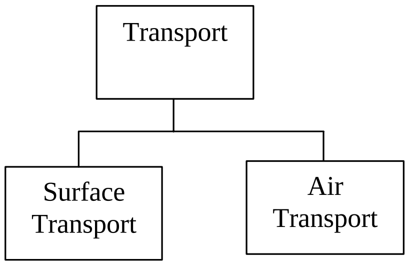


Fig 1.5 *Attributes and Operations of base class* Transport

The extra operation that needs to be added for surface transport is that it works only on land and air transport needs one more extra operation, namely that it works in the air. Operations such as reversed and idle may not be required in the case of air transport. Thus, the inheritance property of OOP facilitates reuse of existing code with necessary modifications.

Polymorphism

Polymorphism is a useful concept in OOP languages. In simple terms it means one name, many duties. It provides a common interface to carry out similar tasks. In other words, a common interface is created for accessing related objects. This facilitates the objects to become interchangeable black boxes. Hence they can be used in other programs as well. Therefore, polymorphism enables reuse of objects built with a common interface.

We know that a class contains data members and member functions. An object of the class has a replica of the variables and the functions. Each function has a name. We can communicate with it by passing data or in other words arguments. A function is similar to that in other languages such as C. It receives data and returns some value or information. In conventional languages, we need three different functions to perform the task of addition of three different data types such as int, float or double since the data types have to be explicitly declared. The three functions have necessarily got to have different names. Therefore, we need to remember three function names for addition of three different data types. However, in C++, we have three functions for performing addition of three different data types but with the same name. Depending on the type of data received, the compiler will pass it to the appropriate function. Thus, we have to remember and use one function name only for the particular operation. This is function overloading where we have multiple functions with the same name. The appropriate function is chosen at the time of compilation through matching of actual data or arguments passed. Function overloading is also known as Polymorphism.

C++ also supports operator overloading. The common operators such as +, – etc. can be programmed to carry out similar operations on different data types or objects depending on the context in which the operator is used. Thus, overloading is execution of multiple functions with the same operator or function name. Such operators and functions are said to be overloaded with different duties, depending on the context in which they are called. Essentially, it enables the same types of operations working for many different data types. Thus, overloading is essentially assignment of multiple meanings or duties or functions to operator names and function names. Such operators and functions are said to be overloaded with different duties. The effect of this is simplification of a program by making the same operators and functions work for many different types of operands or arguments.

Advantages of OOP

Now that we have discussed the three essential principles of OOP – encapsulation, inheritance and polymorphism, let us briefly discuss the advantages of object-oriented programming. These are given below:

- **Reusability:** It refers to the ability for multiple programmers to use existing written and debugged classes in their programs. This is a time-saving and quality improvement mechanism that adds coding efficiency to a language. Moreover, the programmer can incorporate new features into an existing class, further developing the application and allowing users to achieve improved performance. This time-saving feature optimizes code, helps to gain secured applications and facilitates easier maintenance of applications. Object oriented programs are designed for reuse. The features which facilitate ease of reuse, are encapsulation and inheritance.
- **Maintainability:** Since each class is self-contained with data and functions and also the functions are grouped together, these programs are easily maintainable. Encapsulation enhances maintainability of programs.
- **Natural:** The software objects represent the real objects in the problem domain and hence the programming is quite natural to reality. They can be given real names such as transport, cycle, etc., thereby enhancing understandability of the code.
- **Modular:** Modularity is unique not only to OOP, but to function oriented programming as well. Modularity in OOP is facilitated through the division of a program into well-defined and closely knit classes. Thus, any complex program can be divided or partitioned into modules, and the divide and conquer strategy can be used for program design through modularization of the code through classes.
- **Extensibility:** The inheritance mechanism of OOP facilitates easy extension of the features of classes. Thus, old and proven code can be extended easily through the inheritance property of OOP.
- **Data Integrity:** Avoiding global variables, goto statements and binding the data within the class ensures data integrity and restricted access – which is the very purpose of OOP.

- **Improved Quality:** One of the objectives of OOP is to improve the quality of programs. C++ constructs reduce the chances of programming errors with their in-built modularity and reusability.

NOTES

1.4 C++ BASICS

In addition to main(), our Dictionary program contains two functions: LoadDictionary (...), and FoundWord (...). LoadDictionary (...) reads the DICT.DAT file and loads the dictionary into the computer memory. The FoundWord (...) function looks up a given word in the dictionary and retrieves its translation.

Functions may take *arguments* and may *return* a value. For example, the function

```
bool FoundWord (ENTRY dict[], char word[], char translation[]);
```

takes three arguments, written in parentheses after its name: dict, a pointer to the dictionary; word, a character string that contains the original word; and translation, a string to which the translation is copied if the word has been found. The FoundWord(...) function returns a bool (short for *Boolean*, i.e. true/false) value that can be true if the word has been found and false otherwise.

In giving names to functions and their arguments, the programmer tries to choose names that will make the code more readable.

Whenever a statement in the program mentions a function name with some arguments in parentheses, the program carries out that function. We say that the function is *called* with certain arguments. A function can be called from main() or other functions.

For example, the first executable statement in the main program is

```
dict = LoadDictionary('DICT.DAT');
```

This statement calls the LoadDictionary(...) function with one argument, the name of the dictionary file. The function reads the dictionary data from the file into memory and returns a pointer to the dictionary. The returned value is saved in a *variable* called dict.

A function uses the return statement to return a value:

```
return expression;
```

A void function cannot return any value, but it can use a return statement without a return value:

```
if (...)
     return;
```

Basically, the program places the arguments where the function can get them, saves the place the function is called from, and passes control to the first instruction in the function code. When the function has finished its work, it places the return value where the calling statement can get it and goes back to the point where the calling function left off. A program can call a function as many times as it needs a certain task carried out.

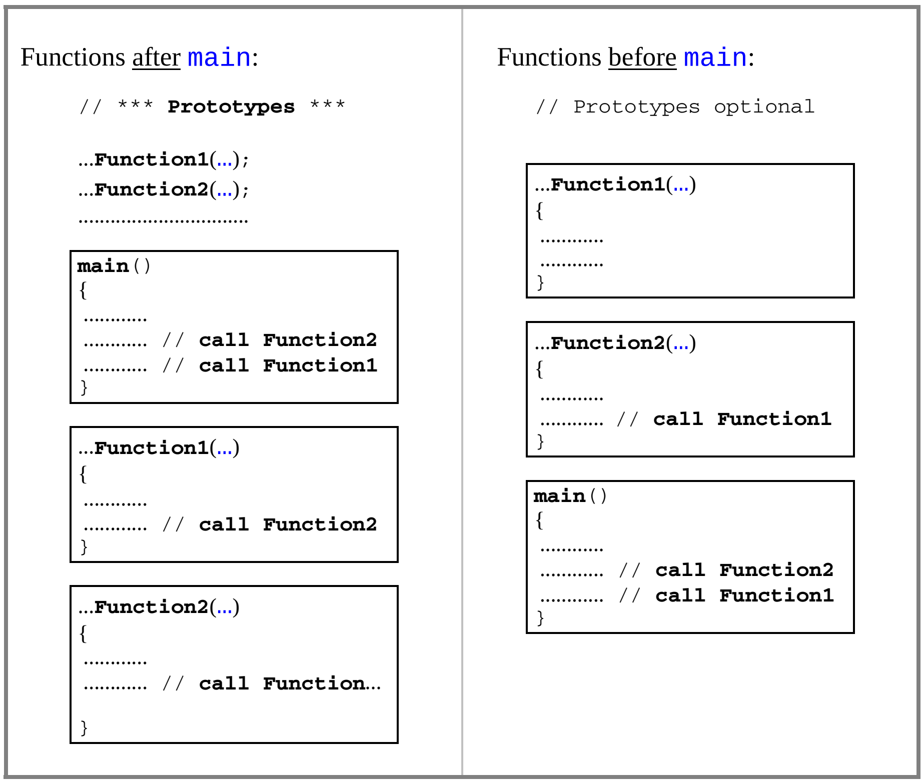
The order of functions in source code is largely a matter of taste. Sometimes programmers put the main program first, followed by the functions; that is what we have done in the Dictionary program. Others may put the main program at the end. In any case, all functions have to be *declared* before we can call them: the compiler has to know what type of arguments they take and what type of values they return. This is

NOTES

accomplished by using *function prototypes*, which are usually placed somewhere near the top of the program. In our Dictionary program we put the prototypes for the LoadDictionary(...) and FoundWord(...) functions above main()so that when we call these functions from main(), the compiler will know what to expect:

```
...
// Function prototypes
ENTRY *LoadDictionary(char filename[]);
bool FoundWord(ENTRY dict[], char word[], char translation[]);
/
*************************************************************/
/*************         Main Program         *************/
/
*************************************************************/
void main()
{
    ...
```

Function prototypes *declare* functions and are also known as *function declarations*. Later in the program come the *function definitions*, which contain the actual function code. The code, placed between opening and closing braces, is called the *body* of the function. Function definitions must agree with their prototypes: they must take the types of arguments and return the types of values that their prototypes say they do. If a function is defined above its first use, no prototype is necessary (see Figure 1.3).


Functions after main:

```
   // *** Prototypes ***

   ...Function1(...);
   ...Function2(...);
   ...............................

   main()
   {
    ...........
    ........... // call Function2
    ........... // call Function1
   }

   ...Function1(...)
   {
    ...........
    ........... // call Function2
   }

   ...Function2(...)
   {
    ...........
    ........... // call Function...

   }
```

Functions before main:

```
   // Prototypes optional

   ...Function1(...)
   {
    ...........
    ...........
   }

   ...Function2(...)
   {
    ...........
    ........... // call Function1
   }

   main()
   {
    ...........
    ........... // call Function2
    ........... // call Function1
   }
```

Figure 1.6 *Placement of functions and functions' prototypes in the source code*

Usually, each function definition is accompanied by a comment that briefly describes what that function does, its arguments, and its return value, if any. The following is the complete definition of the FoundWord(...) function from our Dictionary program:

```
bool FoundWord(DICTENTRY *dict, char word[], char
translation[])
// Finds a word in the dictionary.
//  dict — pointer to the dictionary
//  word — word to translate
//  translation — returned translation from the dictionary.
// Returns true if the word has been found, false otherwise.
{
    bool found = false;
    for (int i = 0;   !found && *dict[i].word != '\0';   i++)
{
        if (stricmp(dict[i].word, word) == 0) {
            strcpy(translation, dict[i].translation);
            found = true;
        }
    }
    return found;
}
```

Sometimes it is convenient to view a function as a procedural abstraction, somewhat like the ice machine in a refrigerator. We know what goes in, we know what comes out, but until we are ready to build the machine, we are not too concerned about exactly what happens inside. Likewise, when we begin to write a program, we can (at some level of abstraction) assume that we will need functions to handle certain tasks—without worrying too much about the details of their code. We can create the prototypes for them, on the assumption that the functions will be coded later. When we have grasped the big picture and are ready to work out all the details, we can go back and write the code for all our functions.

To summarize, there are three main reasons for using functions. One is to split the code into manageable smaller pieces that can be conceptualized, programmed, and even tested separately. Another is to be able to perform the same task in the program several times without duplicating the same code in your program, simply by calling the same function from different places in the program. Finally, thinking about functions keeps a programmer from getting bogged down in the nitty-gritty of a program before he has worked out the overall design.

Library functions and header files

If we look again at the code fragment above, two cryptic lines

```
...
    if (stricmp(dict[i].word, word) == 0) {
        strcpy(translation, dict[i].translation);
    ...
```

look as if they might contain calls to functions stricmp(...) and strcpy(...). But if this were indeed the case, where do these functions come from, and where are they declared and defined? The answer is that stricmp(...) and strcpy(...) are *standard C++ library* functions.

The C++ development environment supplies a vast collection of preprogrammed functions for use in your programs. Like prefabricated construction blocks, these functions can save you a lot of time. The functions are provided to you in separate, already compiled modules that are collected in the standard library. The library functions, strictly speaking, are not a part of the C++ programming language, but over the years of evolution of C and C++, they have become 'standard.'

NOTES

The library contains hundreds of functions that do all kinds of things: manipulate character strings, return the current time and date, convert numbers into character strings and vice versa, calculate mathematical functions, read and write files, and so on. Actually, the libraries may differ slightly among suppliers of C++ compilers, but you can expect a vast majority of core functions to come with any compiler.

For example, in the above code in the FoundWord (...) function, stricmp(...) and strcpy(...) help find a word and retrieve its translation from the dictionary. stricmp(...) is provided with Borland C++ compilers. It compares two character strings (disregarding the difference between uppercase and lowercase letters) and returns 0 if there is a match. strcpy(...) is provided with all C++ compilers. It copies str2 into str1. Both functions work with so-called *null-terminated* strings, which can be of any length and use a *null* (zero) character as an end marker - a standard method in C++. Note that stricmp(...) is also called from the main program to check whether the user entered the quit command ('q'):

```
...
if (stricmp(word, 'q') == 0)
   quit = true;
...
```

The compiler needs the prototypes for the library functions, and, for your convenience, they are provided in the so called *include* or *header* files that come with your compiler. Instead of copying the required prototypes into your program, you can simply include the appropriate header file by using the #include directive. 'Includes' are usually placed at the very top of the program. The Dictionary program, for example, has two 'includes:'

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

fstream.h contains structures and declarations for C++ file input and output, and string.h contains function prototypes for the string handling functions. The names for the header files are not a part of the C++ language, but they, too, have become standard over time. A compiler's library documentation will tell you which header file to include with each function.

A header file is essentially a kind of a source code file. The #include directive instructs the compiler to replace the '#include' line with the full text of the specified file. #include works with any source file, not just system header files. Eventually you will write your own header files.

C++ is a superset of C language. All C programs can be executed as C++ programs, but the reverse cannot happen. In this section we learn the following concepts of C++ language:

- Tokens
- C++ keywords
- Identifiers
- Built-in data types
  - o Integer types, size and range of values
    - ▪ Character literals
    - ▪ Integer literals
  - o Real number types, size and range of values
  - o Floating point notation
  - o Scientific notation
    - ▪ Real number constants
  - o Boolean
  - o String literal
- Constants
- Variables
- Scope of variables
- Dynamic initialization
- Reference variables
- User defined data types
  - o Arrays
  - o Multi-dimensional arrays

Tokens

A language is defined by the use of tokens, which are the smallest elements or the atomic elements. A program can be written using the tokens. Similar to C language, the tokens of C++ language are given below:

- Keywords
- Identifiers
- Constants
- String literals
- Operators
- Punctuators (Separators)

In this unit, we will discuss about the tokens of the language briefly.

C++ keywords

When a language is defined, one has to design a set of instructions to be used for communicating with the computer to carry out specific operations. The sets of instructions which are used in programming, are called keywords. These are also known as reserved words of the language. In the first program, *class* and *int* are reserved words, also known as keywords. They have a specific meaning for the C++ compiler and should be used for giving specific instructions to the computer. These words cannot be used for any other purpose, such as naming a variable. C++ has 74 keywords. These keywords are given below.

C++ Keywords

and	and_eq	asm	auto	bitand	bitor
bool	break	case	catch	char	class
compl	const	const_cast	continue	default	delete
do	double	dynamic_cast	else	enum	explicit
export	extern	false	float	for	friend
goto	if	inline	int	long	mutable
namespace	new	not	not_eq	operator	or
or_eq	private	protected	public	register	reinterpret_cast
return	short	signed	sizeof	static	static_cast
struct	switch	template	this	throw	true
try	typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchar_t	while
xor	xor_eq				

C++ Keywords Not Used in C

and	and_eq	asm	bitand	bitor	bool
catch	class	compl	const_cast	delete	dynamic_cast
explicit	export	false	friend	inline	mutable
namespace	new	not	not_eq	operator	or
or_eq	private	protected	public	reinterpret_cast	static_cast
template	this	throw	true	try	typeid
typename	using	virtual	wchar_t	xor	xor_eq

C++ Keywords that are C Macros

and	and_eq	bitand	bitor	compl	not
not_eq	or	or_eq	wchar_t	xor	xor_eq

Identifiers

Identifiers are nothing but names. The identifiers in C++ are the names assigned to files, programs, classes, functions, objects, constants, variables, etc. An Identifier in C++ is defined as an unlimited sequence of characters, the first of which must be an alphabetic character. Digits or underscores or other letters can follow it. In C++, lowercase and uppercase letters are treated separately and not used interchangeably. VAR and var represent different names in C++. However, you can use both uppercase and lowercase letters in the same identifier.

Note carefully that reserved words cannot be used as identifiers. An identifier may be coined in such a way that it easily visualize what it represents. Therefore, instead of giving names such as A, B or C to variables, a programmer can be liberal and coin identifiers such as Average, Maximum, Minimum, etc.

NOTES

Check Your Progress

1. Explain the principle of object oriented programming.
2. What are the advantages of OOP?
3. Name the features that are added to standard C++.

Data types

Data is the basic raw material for any program. Each programming language has its own set of rules about data representation. C++ being an object oriented programming language, the following two categories of data types are supported:

- Built-in data type, which can also be called a primitive data type
- User-defined type such as structure, class, union, enumeration

Built-in data types

Primitive data types are the fundamental types. The types supported in C++ can be grouped into the following:

· Characters	– to represent symbols such as letters and numbers
· Integers	– to represent whole numbers
· Real numbers	– to represent numbers with a fractional part
· Boolean	– to represent a logical true or false
· Void	– to represent the absence of information

Let us now consider each group of data types, one by one.

Integer types, sizes and range of values

C++ defines four integer types, including characters. Their names and the number of bits allocated to them, as well as range of values they can handle, are given in Table 1.1 below:

Table 1.1 *Data Types – Integers*

Integer Type	*No. of bits*	*Range of values*
Char	8-bit signed	–128 to 127
Unsigned char	8-bit unsigned	0 to 255
short int	16-bit signed	–32,768 to 32,767
unsigned short int	16-bit unsigned	0 to 65535
Int	16-bit signed	–32,768 to 32,767
unsigned int	16-bit unsigned	0 to 65535
Long int	32-bit signed	–2147483648 to 2147483647
unsigned long int	32-bit unsigned	0 to 4294967295

The size of the data type is measured in terms of bytes (1 byte=8 bits). All characters can be represented as American Standard Code for Information Interchange (ASCII) characters and, hence, 1 byte, i.e. 8 bits, is good enough to store a character, which is represented as *char*.

Real number types and sizes

Real numbers can be expressed with single precision or double precision. Double precision means that real numbers can be expressed more precisely. Double precision also means more digits in the mantissa. The type '*float*' means single precision and '*double*' means a double precision real number. There is another data type, the *long double,* which extends the range of numbers of type *double.*

The size and range of values of real numbers just discussed are given in Table 1.2.

Table 1.2 *Real Numbers*

Type	*No. of bits*	*Range of values*
Float	32	3.4e-38 to 3.4 e+38
Double	64	1.7e-308 to 1.7e+308
long double	80	3.4 e-4932 to 1.1e+4932

NOTES

Preprocessor

#include is an example of a preprocessor directive. The preprocessor is a component of the compiler that does some preliminary work with your code before the actual compilation. The preprocessor strips comments from the code and expands #include directives into the actual text of the included file. There are other preprocessor directives. The most common two are #define for defining a constant or a *macro* and #ifdef-#else-#endif for *conditional compilation*.

The #define directive has the general form of

```
#define someName expression
```

The preprocessor goes through your code, substituting *expression* for someName everywhere it appears. For example, we might say:

```
#define MAXWORDS 1000   /* Max number of words in the
dictionary */
```

Defining a constant and referring to it in your program by name, rather than using its specific value, makes your code more general and easier to change.

For example, if we want to change the maximum size of the dictionary from 1000 to 3000, all we have to do is change one line in the program. Otherwise, we would have to go through and change every relevant occurrence of '1000' by hand.

#define was more useful in C. In C++ the preferred method of naming a constant is a constant declaration:

```
const int MAXWORDS = 1000;  // Max number of words in the

    dictionary
```

Conditional compilation allows you to include or exclude fragments of code from the program based on additional conditions. It takes the form:

```
#ifdef someName
... // some code (A)
#else
... // other code (B)
#endif
```

This means that if someName is defined, the first fragment (A) is compiled and included in the program; otherwise, the second fragment (B) is included. (The second part of the statement is optional. Without the #else and the second fragment, the compiler would simply skip fragment (A).) someName can be defined by using #define through a special compilation option or may be predefined under certain compilation conditions. The preprocessor includes the appropriate fragment into your program prior to compilation.

There is also an #ifndef directive – 'if *not* defined.' It instructs the compiler to include a particular fragment only if the name is *not* defined.

One use of conditional compilation is to include special 'debugging' statements that assist in testing the program but are left out of the final version. For example:

```
    // Comment out the following line for compiling without
DEBUG:
    #define DEBUG 1
       ...
       inp >> dict[i].word >> dict[i].translation;
    #ifdef DEBUG
       cout << dict[i].word << dict[i].translation;
    #endif
```

Another use is for handling code that is written for a specific system or compiler. For example:

```
    #ifdef  _Windows
    ...// MS Windows specific code
    #else
    ... // MS DOS code
    #endif
```

The preprocessor directives stand out from the rest of the program. Each directive is written on a separate line and starts with the # character (which may be preceded by some 'white space' – spaces or tabs). Preprocessing is a separate step that precedes the actual compilation.

Reserved words and programmer-defined names

In the C++ language, a number of words are reserved for some special purpose while other words are arbitrary names given by the programmer. A partial list of the C++ reserved words is shown below:

char	sizeof	if	default
int	typedef	else	goto
float	const	for	return
double	static	while	extern
short	void	do	class
long	enum	switch	private
unsigned	struct	continue	public
signed	union	break	protected

Reserved words are used only in a strictly prescribed way.

NOTES

The C++ compiler is case-sensitive: changing one letter in a word from lowercase to uppercase makes it a different word. All reserved words use only lowercase letters.

It is very important to choose names that are somewhat self-explanatory and improve the readability of the program. It is also desirable to follow a convention – for yourself or within a group or organization. For example, you could make all function names start with a capital letter, all names of variables start with a lowercase letter, all names of constants and structures uppercase, and so on.

C++ compilers allow very long names—up to at least 30 characters. But names that are too long clutter the code and actually make it harder to read.

1.4.1 Program Statements

Every program consists of set of instructions for the computer. These instructions are in the form of program statements. Each statement may consist of the following, in particular

- Keyword(s)
- Valid identifiers coined as per the syntax of the language
- Operators and operands
- Punctuators

Every statement in C++ ends with a semicolon.

Syntax and style

Every C++ program must have one special function, called main(), which receives control first when the program is executed.

The C++ compiler (or, more precisely, the linker) builds into your executable program a small initialization module that makes some system-specific preparations and then passes control to `main()`. When your program is finished with `main()`, the initialization module takes over and does some cleaning up before it returns control back to the operating system. `main()` is often informally referred to as the *main program* (even though it is only a function and the program may have other functions). In a program, `main()` is declared void, which indicates that this function does not explicitly return any value:

```
...
void main()
...
```

Both `main` and `void` are *reserved words* in C++, because they have a specific purpose in the language and cannot be used for other purposes.

Normally, you have to keep preprocessor directives, double-slash comments, and text within quotes on the same line. Aside from that, the compiler uses line breaks, spaces and tabs only to separate consecutive words, and one space works the same way as 100 spaces. The redundant white space (spaces, tabs and line breaks) is ignored.

As opposed to English or any other natural language, programming languages have virtually no *redundancy*. Redundancy is a term from information theory that refers

NOTES

to less-than-optimal expression or transmission of information; redundancy in language or code allows the reader to interpret a message correctly even if it has been somewhat garbled. Forgetting a parenthesis or putting a semicolon in the wrong place in an English paragraph may hinder reading for a moment, but it does not usually affect the overall meaning. Anyone who has read a text written by a six-year-old can appreciate the tremendous redundancy in natural languages, which is so great that we can read a text with no punctuation and most words misspelled.

Not so in C++ or any other programming language, where almost every character is essential. We have already mentioned that in C++ all names and reserved words have to be spelled exactly right with the correct rendition of the upper- and lowercase letters. In addition, every punctuation mark or symbol in the program has a precise purpose; omitting or misplacing one symbol leads to an error. At the beginning, it is hard to get used to this rigidity of syntax.

The compiler catches most syntax errors, but in some cases it has trouble diagnosing the problem precisely. For example, braces in a C++ program are logical marks that set blocks of code apart from each other. Suppose we have accidentally omitted one closing brace in the example program shown below:

```
void main()
{
    ENTRY *dict;
    char word[20], translation[20];
    dict = LoadDictionary('DICT.DAT');
    if (!dict) {
        cout << '*** Cannot load dictionary ***\n';
     return;

    }    <-----  This closing brace has
                 been accidentally omitted.

    ...
```

When we compile the program, the compiler can tell that something is not right but cannot figure out where the missing brace was supposed to be. Indentation could help a programmer to locate and fix the missing brace.

Notwithstanding the compiler's somewhat limited capacity to diagnose your syntax errors precisely, you can never blame the compiler for errors. You may be sure that there is *something, wrong* with your code if it does not compile correctly.

Unfortunately, the converse is not always true: the program may compile correctly but still contain errors – 'bugs.' The compiler certainly won't spot logical mistakes in your program. And just as a spell-check program will not notice if you type 'was' instead of 'wad' or 'you' instead of 'your,' a compiler will not find errors that it can mistake for something else. So it is easy to make a minor 'syntax' error that conforms to all the syntax rules but happens to change the meaning of your code.

C++ syntax is not very forgiving and may frustrate a novice. The proper response is attention to detail. Beginners can usually save time by carefully reading their code a couple of times before running it through the compiler. Get in the habit of checking that semicolons, braces, equal signs and other punctuation marks are where they should be.

Statements, blocks, indentation

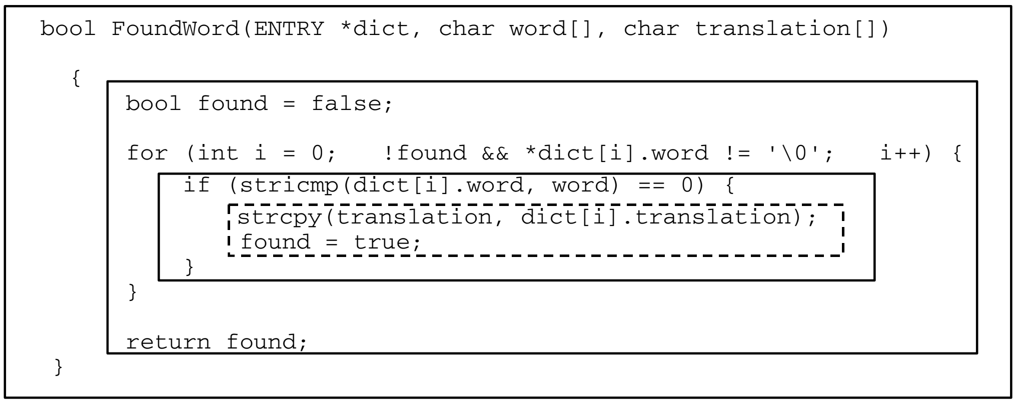
C++ code consists mainly of declarations, definitions, and statements. Declarations and definitions describe and define objects; statements describe actions.

NOTES

> ***Declarations and statements in C++ are usually terminated with a semicolon; statements are grouped into blocks using braces { }. Semicolons should not be omitted before a closing brace.***

Semicolons are only required *after* the closing brace in a few situations, which we will learn later.

Braces divide the code into *nested blocks*. Statements within a block are usually indented by a fixed number of spaces or one tab. In this book we indent nested blocks by four spaces. The blocks are used to indicate that a number of statements form one *compound* statement that belongs in the same control structure, for example an *iteration* (for, while, etc.) loop or a *conditional* (if) statement. The outer block is always the body of a function. Figure 1.5 shows two nested blocks within a function.



```
bool FoundWord(ENTRY *dict, char word[], char translation[])

  {
     bool found = false;

     for (int i = 0;   !found && *dict[i].word != '\0';   i++) {
         if (stricmp(dict[i].word, word) == 0) {
            strcpy(translation, dict[i].translation);
            found = true;
         }
     }

     return found;
  }
```

Figure 1.7 *Nested blocks in the body of a function*

There are somewhat different styles of placing braces. Some programmers prefer placing both opening and closing braces on separate lines, as follows:

```
...
for (int i = 0;   !found && *dict[i].word != '\0';   i++)
{
     if (stricmp(dict[i].word, word) == 0)
      {
   strcpy(translation, dict[i].translation);
   found = true;
      }
}
...
```

This way it is easier to see the opening brace but easy to put, by mistake, an extra semicolon before it.

Another important way to improve the readability of your code is by spacing lines vertically. Use special comment lines and blank lines to separate sections, blocks, and procedural steps in your code.

1.5 VARIABLES AND CONSTANTS

1.5.1 Variables

NOTES

Variables and constants are fundamental data types. A variable can be assigned one value at a time; however, its value can change during program execution. A constant, as the name indicates, cannot be assigned a different value during program execution. If PI is declared as a constant, its value cannot be changed in a given program. Suppose PI has been declared as a constant = 3.14, it cannot be reassigned any other value in the program. A program can declare any number of constants. Variables are useful for any programming language. If PI has been declared as a variable, then its value can be changed in the program to any value. This is the fundamental difference between a variable and a constant. Whether an identifier is constant or variable depends on how it has been declared. Both belong to one of the data types int, float, etc.

The term 'variable' is borrowed from algebra because, as in algebra, variables can assume different values and can be used in *expressions*. The analogy ends there, however. In a computer program, variables are actively manipulated by the program. A variable may be compared to a slate on which the program can, from time to time, write a new value and from which it can read the current value. For example, a statement

```
a = b + c;
```

does not represent an algebraic equality, but rather a set of instructions:

1. Get the current value of b;
2. Get the current value of c;
3. Add the two values;
4. Assign the result to a (write the result into a).

The same is true for

```
a = 4 - a;
```

It is *not* an equation, but a set of instructions for changing the value of a:

1. Take the current value of variable a;
2. Subtract it from 4;
3. Assign the result to a (write the new value into a).

In C++, a statement

```
someName = expression;
```

represents an *assignment* operation which *evaluates* (finds the value) of the expression on the right side of the = sign and assigns that value to (writes it into) the variable someName. The '=' sign in C++ is pronounced 'gets the value of.' (If you want to *compare* two values, use another operator, = =, to mean 'is equal to.')

C++ recognizes different of variables depending on what kind of data they can contain. A variable of type int, for example, represents an integer, and a variable of type float represents a real number.

A data type is a logical notion used in programming languages—the computer memory itself is just a uniform sequence of bits and bytes, 0's and 1's. The help the compiler check the code for errors and allow more efficient computations.

We have used variables in a number of programs. It is nothing but a storage location and must be declared before it is used. A variable is actually the name we assign to a storage location such as *var1*, *var2*, etc. At run time, a memory location is allocated to each variable name, making it convenient for the programmer. The intermediate data structure provides the link between the variable name known to us and the corresponding memory address allocated by the system. Every variable has the following features:

NOTES

- name or identifier
- associated type also known as compile-time type
- a value compatible with the declared type

Once we make such a declaration as above in a program, the value of *ccon* will always remain 600. It can't be changed. If we want to hold different values in an identifier, then we can declare it as a variable. For instance,

```
int  var;
```

declares an identifier *var* as a variable of type integer.

Now let us write a program involving integers.

Program 1.5

```
/*Program for multiplying two integers*/
#include<iostream>
using namespace std;
int main(){
int var1;  //multiplicand
int var2;  //multiplier
int var3;  //product
cout<<"\n Enter var1=   ";
cin>>var1;
cout<<"\n Enter var2=  ";
cin>>var2;
var3=var1*var2;
cout<<"\n product of var1 & var2 =" << var3;
}
```

This is a simple program to multiply two numbers. Three variables, *var1*, *var2* and *var3* are declared as integers. The two numbers *var1* and *var2* are multiplied. The result is stored in *var3*.

Declaration of variables

The main() function above uses three variables: *var1*, *var2* and *var3*.

All variables must be declared before they can be used.

The general format of a declaration is

```
sometype someName;
```

where *sometype* defines the type of the variable (a built-in data type for now, but it can also be a *user-defined* data type, as explained later), and someName is the name given by the programmer to his particular variable. Several variables of the same type may be declared together. For example:

```
double amt1, amt2, amt3;
```

A variable can be declared only once within its scope (scope refers to the space in the program where the variable is 'visible').

After declaring the three integer variables, we have a statement to display a message as given below:

```
cout<<"\n Enter var1=   ";
```

You may wonder where the *std* prefix for *cout* is. At the top of the program there is a statement which says:

```
using namespace std;
```

When we make such a declaration (as above) at the beginning of a program, we omit the *std* prefix to objects in the C++ standard library.

Ensure that the escape character "\n" precedes the message we propose to display so that before the message is displayed, the cursor goes to the next line. The message will appear in the next line after one space. This is similar to the printf function we often use in C programming. The next statement is the counterpart of the scanf function of C. The next statement is:

```
cin>>var1;
```

This statement will assign the value typed for variable *var1*. Notice that the arrow points towards *var1*. This can be assumed to mean that what is typed in the standard input device, i.e. the keyboard, is transferred to *var1*. But for declaring the namespace on top of the program, we should have typed with the *std* prefix as given below:

```
std::cin>>var1;
```

Again *cin* is a built-in object of *iostream* class in the C++ standard library.

The functions scanf() and printf() can also be used in C++ as in C, for reading from a standard input device and writing to a standard output device. However, since *cin* and *cout* are more convenient, we will use them in this book.

In the next line we display another message to enter *var2* as given below:

```
cout<<"\n Enter var2=  ";
```

We then receive the input and store what was typed in *var2 (*because of the following statement):

```
cin>>var2;
```

Then we multiply *var1* and *var2* and store the result in *var3* with the next statement of the program reproduced below:

```
var3=var1*var2;
```

In the next statement, we first display what we are going to display and then the name of the variable *var3* itself.

```
cout<<"\n product of var1 & var2 =" << var3;
```

We can write this as two statements, as shown below:

```
cout<<"\n product of var1 & var2 =";
cout << var3;
```

The first statement above displays the message in the new line because of the new line character “\n”. The next statement displays the contents of *var3* in the standard output device, the monitor. In Program E2x1, we have combined both the above with one cout *object* and two insertion operators”<<“. Now look at the result of the program on execution.

Result of Program 1.5

```
Enter var1 = 25
Enter var2 = 24
product of var1 & var2 = 600
```

We typed 25 and 24 as values of *var1* and *var2*. The system printed the balance. You can use the program to multiply any set of numbers.

The operator “>>” is called extraction operator. It gets the values from the standard input device and assigns them to the variable on the right hand side. We can have multiple extraction operators for the *cin* object similar to the *cout* object. For instance, in the above example we can combine, as shown below:

```
cin>>var1>>var2;
```

In this case the first value typed will be assigned to *var1* and the next value typed to *var2*. Before typing the second value you may press the Enter key to indicate that the first value has been typed and what follows is the next value.

Scope of Variables

In C language, we have to declare the variables at the beginning of the main function. We cannot declare them after the program which contains the executable statements. However, in C++, we can declare a variable anywhere in the program – but before it is used. A function starts with an opening brace and terminates with the corresponding closing brace. Similarly, a class has a definite boundary, identifiable by the respective opening and closing braces.

The main functions have variables declared with or without initial values. The scope, i.e. where in the program the variables will appear, is an important question. The variables declared at the beginning of the main function will be visible throughout the function. As soon as program execution exits the function, the variables declared at the beginning of the function will be lost since the memory allocated for these variables get de-allocated. Thus, variables will be created when the scope is entered and destroyed as soon as the scope is lost.

We have also been creating one block in each main function. There can be any number of blocks in a function. Each block is identifiable by an opening brace and a corresponding closing brace. Each block can declare its own variables. The block variables are created when the program enters the block and destroyed as soon as the program exits the block. Blocks can also be nested.

Program 1.6

```
/*Program for demonstrating scope of variables*/
#include<iostream>
using namespace std;
int main(){
//first block
```

NOTES

```
int var1=10;
cout<<"\n value of var1 in the first block =" << var1;
   { //second block
     int var2=20;
    cout<<"\n value of var1 in the second block =" << var1;
    cout<<"\n value of var2 in the second block =" << var2;
       { //Third block
         int var1=30, var2=40;
          cout<<"\n value of var1 in the third block =" <<
var1;
          cout<<"\n value of var2 in the third block =" <<
var2;
                              }
         //Fourth block
         cout<<"\n value of var1 in the fourth block =" <<
var1;
         cout<<"\n value of var2 in the fourth block =" <<
var2;
                                     }
                                             }
```

Before we analyse, study the result of the program given below:

Result of Program 1.6

```
value of var1 in the first block =10
value of var1 in the second block =10
value of var2 in the second block =20
value of var1 in the third block =30
value of var2 in the third block =40
value of var1 in the fourth block =10
value of var2 in the fourth block =20
```

In the above program, var1 has been declared at the beginning of the main function in the first block, therefore it will be available throughout the function. As the second print statement indicates, the value of var1 remains 10 in the second block also. Now, var2 has been declared and assigned an initial value of 20 in the second block. It will also be available in the entire block. Blocks lie between a pair of braces – the opening and the corresponding closing brace. Now, the interesting part is the third block. Here, we have declared var1 again and assigned the initial value as 30. Since var1 has been redefined in the third block, the var1 declared in main function will be hidden. In this block, var1 will have a value of 30 only, as shown in the print statement. Similarly, the original value of var2 is also hidden in the third block and its value will be 40 as per the new declaration. Then the program comes out of the third block (indicated by the closing brace). It is indicated as "Fourth block" in the comment statement of the program. This is actually the second block; the third block is nested in the second block, which is again nested in first block. Since we have come out of the third block, the values assigned in the third block for var1 and var2 will be lost. Therefore, var2 will have value of 20, since that is the value assigned in the second block. Similarly, var1 will have the value of 10 since that is its value in the second block. The print statements confirm this.

The program demonstrates two points:

1. A variable can be declared as and when it is used. In C language, we have to declare all the variables at the beginning of the function.
2. The variables are local to blocks. In other words, the value assigned to a variable in a block prevails. The old values, if any, of the same variable name are hidden in the block in which it is overridden. However, the value is not lost, as demonstrated by the last two print statements.

Dynamic initialization of variables

In a C program, the initialization of variables is to be carried out with actual values of the appropriate types. But in C++, we can assign expressions as initial values to variables. In such a case, the variables are assigned initial values at run time. In the former case, as also in a C program, the initial values are assigned at compile time. The following program demonstrates the dynamic initialization of variables or, in other words, run time initialization of variables.

Program 1.7

```
/*Program for demonstrating dynamic initialization of
variables*/
#include<iostream>
using namespace std;
int main()
{
  int var1=20;
  int var2=var1*var1;
  cout<<"\n value of var1  =" << var1;
  cout<<"\n value of var2  =" << var2;
             }
```

In the above program, var1 is assigned a value of 20 in the conventional manner. However, var2 is assigned the initial values dynamically. The variable var2 is the square of var1. This is called dynamic initialization and is used in object oriented programming languages such as C++. The result of the program confirms that it works.

Result of Program 1.7

```
value of var1  =20
value of var2  =400
```

This method is also convenient to programmers.

Reference variables

C++ has another feature, which is called reference variables. With this feature we can assign the value of an already defined and initialized variable to a new variable. This is illustrated in the program below:

NOTES

Program 1.8

```
/*Program for assigning value of a defined and initiliazed
variable to a new variable*/
#include<iostream>
int main(){
  int var1=20;
  int & var2=var1;
  std::cout<<"\n value of var2  =" << var2;
            }
```

Result of Program 1.8

```
value of var2  =20
```

In the above program, var1 has been declared as an integer and the value of 20 is assigned to it. In the next statement through an & operator, we have made var2 also to refer to var1. In the next statement, we print the value of var2 and it prints exactly the value as that of var1. Thus, we have created a reference variable to var1. var2 is an alternate name to var1. In the above example, the & does not mean address, but it is a reference to integer var1. This concept can be useful when extended to objects (to be discussed later).

Use of comments

The first thing we notice is that the code contains some phrases in plain English. These are *comments* inserted by the programmer to explain and document the program's features. It is a good idea to start any program with a comment explaining what the program does, who wrote it and when, and how to use it. The comment may also include the history of any revisions: who made changes to the program, when, and why. The author must assume that his program will be read, understood, and perhaps modified by other people.

In C++, the comments may be set apart from the rest of the code in two ways. The first method is to place a comment between /* and */ marks:

```
/* Maximum number of words
            in the dictionary */
const int MAXWORDS = 1000;
```

In this method, the comment may be placed *anywhere* in the code, even within expressions. For example:

```
/* This is allowed, but bad style: */
const int MAXWORDS /* Maximum number of words in the dictionary
*/ =
    1000;
```

The only exception is that *nested comments* (i.e. one set of /*...*/ within another) are normally not allowed:

```
/* /* This nested comment */  will NOT be processed
  correctly, unless your compiler has
  a special option enabled for handling nested comments. */
```

The second method is to place a comment after a double slash mark on one line. All the text from the first double slash to the end of the line is treated as comment. For example, we can write:

```
const int MAXWORDS = 1000;  // Max number of words in the
dictionary
or
// Maximum number of words in the dictionary:
const int MAXWORDS = 1000;
```

Judicious use of comments is one of the tools in the constant struggle to improve the readability of programs. Comments document the role and structure of major code sections, mark important procedural steps, and explain obscure or unusual twists in the code.

On the other hand, excessive or redundant comments may clutter the code and become a nuisance. A novice may be tempted to comment each statement in the program even if the meaning is quite clear from the code itself. Experienced programmers use comments to explain the parts of their code that are less obvious.

Comment marks are also useful for *commenting out* (temporarily disabling) some statements in the source code. By putting a set of /*...*/ around a fragment of code or a double slash at the beginning of a line, we can make the compiler skip it on a particular compilation. This can be useful for making tentative changes to the code.

1.5.2 Constants

Character literal

A literal means a constant. In a program, a constant value of a particular type can be assigned to an identifier. For instance, if we want to assign character ‘c’ to a character constant *ccon*, we write as given below:

```
const char ccon=‘c’;
```

Here the keyword *const* indicates that it is the declaration of a constant. The keyword char indicates that the identifier *ccon* is of type *char*. We are also assigning value ‘c’ to *identifier ccon* in the same statement.

A character constant is a character enclosed in single quote as in ‘c’. Characters can be letters, digits or special symbols and occupy 8 bits.

Examples

Valid Character Constants:

```
‘A’
‘c’
```

Integer literal

An integer literal can be any one of the following types:

```
char
int
long int
unsigned long int
```

If the integer is suffixed with l or L, it signifies a long integer. If it is suffixed with u then it is unsigned. If it is suffixed with ul or lu then it is of type unsigned long int.

Examples

Valid integers

```
+345         /*    integer  */
-345         /*    integer  */
-2678112345L /*   Long integer */
+112345ul     /*   Unsigned Long integer */
```

Invalid integers

```
345.0        /*    decimal point not allowed */
112  345UL   /*    =     blank not allowed */
±345l        /*    ± not allowed */
```

An integer constant can be declared as follows:

```
const int icon = 600;
```

Constants represent memory locations whose values do not change while the program is running. Your source code may include *literal constants* and *symbolic constants*. Examples of *literal constants* are decimal representations of integers and real numbers and characters in single quotes, for example:

```
'y', 'H'        - characters;
 7,  -3         - integers;
 1.19, .05, 12. - float or double numbers.
```

Character constants also include a special set of non-printable characters that are sometimes called *escape* characters (the term derived from printer control commands). Escape characters are represented in C++ by an alias—a designated printable character—preceded by a backslash. The escape characters include:

```
\a alert (bell)
\t tab
\n newline (line feed)
\r carriage return
\f form feed
\' single quote
\' double quote
\\ backslash
```

For example, an output statement

```
cout << 'Change: ' << '\a' << change << endl;
```

inserts a 'bell' into the output. This will normally sound the speaker on a personal computer in addition to displaying the dollar amount of change on the screen.

Symbolic constants are *named by the programmer* and are declared in a manner similar to variables, except that the declarations are preceded by the reserved word const and some value must be assigned to the constant. For example:

```
const double hamburgerPrice = 1.19;
```

NOTES

The general form of symbolic constants' declarations is

```
const sometype name1 = value1, name2 = value2,...;
```

where *sometype* is a data type (a built-in data type or a previously defined type) followed by a list of symbolic names with their values. A constant may be also initialized to some expression, but the expression must contain only constants, either literal constants or previously declared symbolic constants. For example:

```
const double hamburgerPrice = 1.19;
const double cheeseburgerPrice = hamburgerPrice + .20;
```

or

```
const double hamburgerPrice = 1.19,
             cheeseburgerPrice = hamburgerPrice + .20;
```

It may seem, at first, that symbolic constants are redundant and we can simply use their literal values throughout the program. For example, instead of writing

```
...
const double taxRate = .05;
...
taxAmt = amt * taxRate;
```

we could simply write

```
...
taxAmt = amt * .05;     // Sales tax rate = 5%
```

The most important reason for using symbolic constants is easier program maintenance. If the program is modified in the future and the value of a constant has to be changed, only the constant declaration has to be changed by the programmer.

A programmer who uses literal constants will have to search through the whole source code and replace the old value with the new one wherever it occurs. This is tedious and can easily cause errors.

Another advantage of symbolic constants is that they may make the code more readable and self-explanatory if their names are well chosen. The name can explain the role a constant plays in the program, making additional comments unnecessary.

It is also easier to change a symbolic constant into a variable if a program modification requires that. For example, in a simplified version of the Fast food program, we can declare prices as constants. In a later upgrade, we can simply remove the const specifier and add a function for setting new prices.

Symbolic constants, like variables, are declared with a particular data type and are defined only within their scope. This introduces more order into the code and gives the compiler additional opportunities for error checking—one more reason for using symbolic constants.

On the other hand, there is no need to clutter the code with symbolic names assigned to universal constants such as 0 or 1 if these values inherently belong in the code.

Look at one more example involving an integer constant.

Program 1.9

```
/*Program for finding square of an integer*/
#include<iostream>
int main(){
const int var=25;
std::cout<<"\nsquare of "<<var<<"=" << var*var;
}
```

Result of Program 1.9

```
square of 25=625
```

This is a simple program. We are declaring the *var* as an integer constant by using the keyword *const*. We directly multiply 25 by it to produce the square of the number. The simplicity lies in calculating it as part of the print statement.

1.5.3 Scope of Variables and Constants

You might have noticed that each function in the above program uses its own variables. Can a variable or a constant declared in one function be used in another function? Can a variable or a constant be declared in such a way that it is usable in several functions? These questions are related to the subject of *scope*.

In C++ a variable is defined only within a certain space in the program called the scope of the variable. The same is true for symbolic constants. The scope rules work exactly the same way for variables and symbolic constants.

Scope discipline helps the compiler to perform important error checking. If you try to use a variable or constant outside its scope, the compiler detects the error and reports an undeclared name. The compiler also reports an error if you declare the same name twice within the same scope and a warning if you try to use a variable before it has been assigned a value.

C++ programmers distinguish local variables declared within functions from global variables declared outside of any function. A beginner should declare all global variables near the top of the program and all local variables at the top of the function's code. The scope of a global variable extends from its declaration to the end of the program module (source file). The scope of a local variable declared at the top of a function extends from the declaration to the end of the function body (closing brace).

A *global* variable or constant is usable in any function below its declaration. The value of a global variable is maintained as long as the program is running. For example, it can be set in one function and used in another function. A global variable *goes out of scope* and its memory location is released only when the program finishes its execution.

A *local* variable exists only temporarily while the program is executing the function where that variable is declared. When a program passes control to a function, a special chunk of memory (a *frame* on the system *stack*) is allocated to hold that function's local variables. When the function is exited, that space is released and all local variables are destroyed.

(There is a way to override the temporary nature of local variables by using the C++ reserved word static. A *static* local variable still has local scope, but its value is preserved between successive calls to the function.)

As we have mentioned earlier, local variables and constants in C++ do not have to be declared at the top of the function but can be declared anywhere in the function code. But the rules for declarations inside nested blocks may be confusing and such declarations may lead to elusive 'bugs.' We recommend that you place all declarations of local variables at the top of functions.

> ***It is good practice to use as few global variables and constants as possible and to always use different names for local and global variables.***

C++ allows you to use the same name for a global and a local variable, with the local variable taking precedence over the global one in the function where it is declared. This may lead to errors that are hard to catch if you inadvertently declare a local variable with the same name. Consider, for example, the following code:

```
...
const double hamburgerPrice = 1.19; // global constant
double amt;                         // global variable
void TakeOrder()
{
    ...
    amt = hamburgerPrice;
                     // amt is not declared in TakeOrder(),
                   // so this refers to the global variable
    ...
}
void main()
{
    double amt;  // local variable declared here by mistake.
                     //   It has the same name as a global
variable.
                   //   The syntax is OK, but that was not
the
                   //   intention!
    TakeOrder();
    cout << amt; // output is garbage, because the value of
(local)
                   //   amt is undefined.
}
```

Use global variables and constants only when they indeed represent quantities you will refer to throughout the program, and give them *conspicuous* names. Excessive use of global variables is a sure sign of bad program design, because it is not

NOTES

obvious where and how they are used. Making changes to such a program may be difficult.

It is perfectly acceptable to use the same name for local variables in different functions. In fact this is a good practice if the variables represent similar quantities and are used in a similar way.

But never try to economize on declarations of temporary local variables within functions and on passing arguments to functions by resorting to global variables.

Floating point notation

Let us try and understand the difference between floating point and integer numbers.

a) Integers are whole numbers without a decimal point, but float has always a decimal point. Even if the number is whole number it is written with decimal point; for example, 42 is an integer while 42.0 is a floating point number.

b) Floating point numbers require more space for storage.

A real number in the simple form consists of one or more integers before the decimal point and also one or more decimal numbers following the decimal point, which constitutes the fractional part. This form of representation is known as fractional form. It could be either positive or negative. As usual, the default sign is positive. No commas or blanks or special characters allowed in between.

Examples

Valid floats

```
144.00
226.012
```

Invalid floats

```
+144   /* no decimal point  */
1,44.0 /* comma not allowed */
```

Scientific notation

Floating point numbers can also be expressed in scientific notation. For instance, 3.0 E2 is a floating point number. The value of the number will be equal to 3.0 x 10^2 = 300.0.

Instead of uppercase E, lowercase e can also be used.

0.453 e+05 will be equal to 0.453 x 10^5 = 45,300.

There are two parts in a scientific notation of a real number:

- Mantissa (before E)
- Exponent (after E)

A scientific notation must follow the rules explained below:

a) The mantissa can be positive or negative

b) The exponent must have at least one digit, which can be a positive or negative integer. Remember, the exponent cannot be a floating point number.

Real number constants

These constants are suffixed as shown below:

```
F or f: float
no suffix: double
```

Examples

Valid floating point constants

```
1.0   e 5
123.0 f          /*    float       */
11123.05    /*    double           */
```

Invalid real constants

```
456      /* It is an integer  */
2.0 e 5.0   /*    an exponent cannot be a real number */
```

When they are declared as constants, they can be declared as follows:

```
const float a = 3.12 f;
```

When they are declared as variables, they can be declared as follows:

```
float a, b, c;
float val1;
float val2;
```

Let us use real numbers and write a program. The following program finds the average of 3 real numbers given in three different forms:

Program 1.10

```
/*Program for finding average of three real numbers*/
#include<iostream>
int main(){
float var1=10.0E1, var2=20.0f, var3=30.0, var4;
var4=(var1+var2+var3)/3;
std::cout<<"\n average of the numbers you entered =" <<
var4;
        }
```

In this program, float var1 is assigned an initial value in the scientific notation, var2 is declared as a float constant (note also the suffix f), and the third number var3 is declared as double. Note the combining of four declarations. The average is stored in double var4 and hence we get the average with double precision as shown in the result.

Result of Program 1.10

```
average of the numbers = 50
```

Boolean

The Boolean type is represented as *bool*. A literal or variable of type *bool* can hold one of the following values:

- true
- false

It is used to express the result of logical operations. If we convert *bool* to integer we get the following:

```
true = 1
false = 0
```

NOTES

Similarly, we can convert an integer to *bool* value as shown below:

```
Nonzero integer = true
Zero = false
```

NOTES

Let us look at an example to understand *bool*.

Program 1.11

```
/*to demonstrate Boolean*/
#include<iostream>
using namespace std;
int main()
{
int var1=4, var2=0;
bool var3, var4;
var3=var1;
var4=var2;
cout<<"\n value of var3 ="<< var3;
cout<<"\n value of var4 ="<< var4;
var1=var3;
var2=var4;
cout<<"\n value of var1 ="<< var1;
cout<<"\n value of var2 ="<< var2;
}
```

Determine what should be the result before looking at the result of the program given below:

Result of Program 1.11

```
value of var3 =1
value of var4 =0
value of var1 =1
value of var2 =0
```

String literal

A character constant is a single character enclosed within single quotes. A string constant is a number of characters, arranged consecutively and enclosed within double quotes. C++ strings can begin and end in the same line.

Valid strings

```
"God"
"is within me"
```

A string constant can contain blanks, special characters, quotation marks, etc.

Invalid strings

```
'Yoga' /* should be enclosed in double quotes */
```

All the Escape sequences, or for that matter any character, if enclosed within double quotes constitute a string literal.

1.5.4 Data Types

C++ has the following eleven *built-in* types designated by reserved words:

```
    char            unsigned char
    int             unsigned int
    short           unsigned short
    long            unsigned long
    float
    double
    long double
```

NOTES

Table 1.3 *Built-in Data Types*

TYPE	SIZE	USE
char	1 byte (8 bits)	One character, or a small integer in the range from –128 to 127.
unsigned char	1 byte (8 bits)	A small non-negative integer in the range from 0 to 255 or one byte of memory.
int	2 or 4 bytes	An integer in the range from -2^{15} to $2^{15}-1$ or from -2^{31} to $2^{31}-1$, respectively.
unsigned int	2 or 4 bytes	A non-negative integer in the range from 0 to $2^{16}-1$ or from 0 to $2^{32}-1$.
short	2 bytes	An integer in the range from -2^{15} to $2^{15}-1$.
unsigned short	2 bytes	A non-negative integer in the range from 0 to $2^{16}-1$.
long	4 or 8 bytes	An integer in the range from -2^{31} to $2^{31}-1$ or from -2^{63} to $2^{63}-1$.
unsigned long	4 or 8 bytes	A non-negative integer in the range from 0 to $2^{32}-1$ or from 0 to $2^{64}-1$.
float	4 bytes	A real number in floating point representation.
double	8 bytes	A double-precision real number in floating point representation.
long double	8, 12 or 16 bytes	An extended-precision real number in floating point representation.

Because variables of different types occupy different amounts of memory space, we say that they have different sizes.

The exact implementation of types depends on the computer system, compiler, and operating environment. In the 16-bit architecture, for example, the int type may be implemented as a two-byte (16-bit) memory location (the same as a short). The most significant bit represents the sign of the number. In this case, its range is between -2^{15} and $2^{15}-1$. These are useful numbers to remember:

```
-215  = -32768
215-1 =  32767
```

In the 32-bit architecture the int type is usually implemented as a four-byte (32-bit) memory location.

NOTES

Table 1.1 summarizes the common implementation and use of built-in types. char, int, short, long, and the corresponding unsigned types are collectively called integral types.

In this book we will mostly use the char, int, and double .

You can find the sizes of different types for your compiler and environment by using the C++ sizeof(x) operator, where x may be the name of a data type or of a particular variable. The following program, (we call it SIZE.CPP) will print out a table of sizes for four common types:

```
#include <iostream.h>
void main()
{
    cout << endl;
    cout << 'TYPE  SIZE ' << endl;
    cout << '------ ' << endl;
    cout << 'int     ' << sizeof(int) << endl;
    cout << 'long    ' << sizeof(long) << endl;
    cout << 'float   ' << sizeof(float) << endl;
    cout << 'double ' << sizeof(double) << endl;
}
```

C++ compilers also provide a special header file, limits.h, which defines the ranges for integral types, and another header file, float.h, which provides useful constants for floating-point types.

It is a programmer's responsibility to make sure that the results of computations fit within the range of the selected type and that precision is adequate for the floating point computations.

Renaming Data Types with `typedef`

C++ provides the typedef statement for naming new . For example, a statement in your program:

```
    typedef unsigned char BYTE;
```

can introduce a new type name, BYTE, for the unsigned char type. Then, the declarations

```
    unsigned char b;
```

and

```
    BYTE b;
```

become identical and can be used interchangeably.

Sometimes it is desirable to use an alias (an abstract name) for a data type, because that lets you change the type for all variables that are used for a particular purpose using only one typedef statement. In our POS program, for example, we could use the alias MONEY for the double type and declare all variables that represent dollar amounts with the MONEY type:

```
...
typedef double MONEY;
...
void main()
{
    MONEY amt;
    ...
```

NOTES

typedef statements are usually placed near the top of the program or in your own header file, which you can include by using #include.

1.6 LOOPS AND DECISION

1.6.1 Iterative Statements: `while, for, do-while`

Loops or *iterative statements* tell the program to repeat a fragment of code several times or as long as a certain condition holds. Without iterations, a programmer would have to write separately every instruction executed by the computer, and computers are capable of executing millions of instructions per second. Instead, programmers can implement solutions to problems using fewer instructions, some of which the computer repeats many times. A formal description of the procedural steps needed to solve a problem is called an *algorithm*. Designing, implementing, and understanding algorithms is a crucial programming skill, and iterations are a key element in non-trivial algorithms.

Iterations are often used in conjunction with arrays. We need to use iterations if we want to perform some process on all the elements of an array. For example, we might want to find the largest element of an array, or the sum of all the elements.

C++ provides three convenient iterative statements: `while`, `for`, and `do-while`. Strictly speaking, any iterative code can be implemented using only the while statement, but the other two add flexibility and make the code more concise and idiomatic.

`while` and `for` Loops

The general form of the while statement is:

```
while (condition) {
    statement1;
    statement2;
    ...
}
```

condition can be any arithmetic or logical expression; it is evaluated exactly the same way as in an if statement.

Informally the while statement is often called the *while loop*. The statements within braces are called the *body* of the loop. If the body consists of only one statement, the braces surrounding the body can be dropped:

```
while (condition)
   statement1;
```

It is important *not* to put a semicolon after while *(condition)*. With a semicolon, the body of the loop would be interpreted as an empty statement, which would leave *statement1* completely out of the loop.

The following function returns the sum of all integers from 1 to n:

```
int AddUpTo (int n)
// Returns the sum of all integers from 1 to n, if n >= 1;
//  (and 0 otherwise).
{
    int sum = 0;
    int i = 1;
    while (i <= n) {
        sum += i;
        i++;        // increment i
    }
    return sum;
}
```

We can discern three elements that must be present, in one form or another, with any while loop: initialization, a test of the condition, and incrementing.

Initialization

The variables tested in the condition must first be initialized to some values. In the above example, i is initially set to 1 in the declaration int i = 1.

Testing

The condition is tested *before* each pass through the loop. If it is false, the body is not executed, iterations end, and the program continues with the next statement after the loop. If the condition is false at the very beginning, the body of the while loop is *not executed at all*. In the AddUpTo(...) example, the condition is i <= n. If n is zero or negative, the condition will be false on the very first test (since i is initially set to 1). Then the body of the loop will be skipped and the function will return 0.

Increment

At least one of the variables tested in the condition must change within the body of the loop. Otherwise, the loop will be repeated over and over and never stop, and your program will hang. The change of a variable is often implemented with increment or , but it can come from any assignment or input statement. At some point, however, the tested variables must get such values that the condition becomes false. Then the program jumps to the next statement after the body of the loop.

In the AddUpTo(...) function, the change is achieved by incrementing the variable i:

```
    ...
        i++;        // increment i
        ...
```

These three elements — initialization, test, and increment (change) — must be present, explicitly or implicitly, with every `while` loop.

In the following more concise and efficient but less obvious implementation of the AddUpTo(...) function, the numbers are added in reverse order, from n to 1:

```
int AddUpTo (int n)
// Returns the sum of all integers from 1 to n, if n >= 1;
//  (and 0 otherwise).
{
    int sum = 0;
    while (n > 0)
        sum += n—;
   return sum;
}
```

In this code, initialization is implicit in the value of the argument n passed to the function, and the 'increment' (actually a *decrement*) is buried inside a compound assignment statement.

The for loop is a shorthand for the while loop that combines initialization, condition, and increment in one statement. Its general form is:

```
    for (initialization;   condition;   increment) {
    statement1;
    statement2;
        ...
    }
```

where *initialization* is a statement that is *always* executed once *before the first pass* through the loop, *condition* is tested *before each pass* through the loop, and *increment* is a statement executed *at the end of each pass* through the loop.

A typical example of a for loop is:

```
    for (i = 0;   i < n;   i++) {
          ...
    }
```

The braces can be dropped if the body of the loop has only one statement.

The AddUpTo(n) function can be rewritten with a for loop as follows:

```
int AddUpTo (int n)
// Returns the sum of all integers from 1 to n, if n >= 1;
//  (and 0 otherwise).
{
    int sum = 0;
    int i;
    for (i = 1;   i <= n;   i++)
        sum += i;
    return sum;
}
```

Initialization and increment statements may be composed of several statements separated by commas. For example:

```
int AddUpTo (int n)
// Returns the sum of all integers from 1 to n, if n >= 1;
//  (and 0 otherwise).
{
    for (int sum = 0, i = 1;   i <= n;   i++)
        sum += i;
    return sum;
}
```

Or, as written by a person determined not to waste any keystrokes:

```
int Add (int n)
{
    for (int s = 0; n > 0; s += n——);
    return s;
}
```

(In this inconsiderate code, the body of the for loop is empty—all the work is done in the 'increment' statement.)

The following function calculates *n*! (*n factorial*), which is defined as the product of all numbers from 1 to *n*:

```
long Factorial (int n)
// Returns 1 * 2 * ... * n, if n >= 1 (and 1 otherwise).
{
    long f = 1;
    int k;
    for (k = 2;   k <= n;   k++)
        f *= k;
    return f;
}
```

Lab: Fibonacci numbers

Write and test a program that calculates the *n*-th Fibonacci number.

The sequence of Fibonacci Numbers is defined as follows: the first number is 1, the second number is 1, and each consecutive number is the sum of the two preceding numbers. In other words,

```
F1 = 1;
F2 = 1;
Fn = Fn-1 + Fn-2    (for n > 2).
```

The first few numbers in the sequence are 1, 1, 2, 3, 5, 8, 13, ... The numbers are named after Leonardo Pisano (Fibonacci), who invented the sequence in 1202. The numbers have many interesting mathematical properties and even some computer applications.

Your main program should prompt the user for a positive integer n, call the Fibonacci(n) function, and display the result. Note that Fibonacci numbers grow rather quickly, so it is a good idea to keep them in variables of the long or even long double data type and to request small *n* when you test your program.

NOTES

The Fibonacci(n) function can be based on one iterative loop. You can have it keep two previously calculated numbers f1 and f2, find the next number f3 = f1 + f2, and then, before the next pass through the loop, shift the values between variables as follows:

```
...
f1 = f2;
f2 = f3;
...
```

The `do-while` Loop

The do-while loop differs from the while loop in that the condition is tested *after* the body of the loop. This assures that the program goes through the iteration at least once. The general form of the do-while statement is:

```
do {
   ...
} while (condition);
```

The program repeats the body of the loop as long as *condition* remains true. It is better to always keep the braces, even if the body of the loop is just one statement, because the code without them is hard to read.

do-while loops are used less frequently than while and for loops. They are convenient when the variables tested in the condition are calculated or entered within the body of the loop rather than initialized and incremented. For example:

```
...
char answer;
do {
   ProcessTransaction();
   cout << 'Another transaction (y/n)? ';
   cin >> answer;
} while (answer == 'y');
...
```

If for some reason you do not like do-while loops, they can be easily avoided by using a while loop and initializing the variables in such a way that the condition is true before the first pass through the loop. The above code, for example, can be rewritten as follows:

```
 ...
char answer = 'y';   // Initially answer is set equal to
'y'.
 while (answer == 'y') {
     ProcessTransaction();
     cout << 'Another transaction (y/n)? ';
     cin >> answer;
 }
 ...
```

NOTES

`break and continue`

The reserved words break and continue can be used only inside a body of a loop. (The break statement can be also used inside a switch; see Section 7.6.) break instructs the program to immediately break out of the loop and continue with the next statement after the body of the loop. continue tells the program to skip the rest of the statements on the current iteration and go to the next pass through the loop. Both these statements must always appear inside a conditional (if or else) statement—otherwise some code in the body of the loop would be skipped every time, and the compiler would generate a warning: 'Unreachable code in ...'.

In the following example, the program calculates the sum of the reciprocal squares of the first *N* positive integers:

$$\frac{1}{1^2}+\frac{1}{2^2}+\frac{1}{3^2}+\ldots+\frac{1}{N^2}$$

This series converges to $\pi^2/6$ and can be used to approximate π. The loop ends when we have added 100 terms, or when the last added term is less than a given threshold epsilon, whichever happens first:

```
    #include <iostream.h>
    #include <math.h>
        const double epsilon = .000001;
    void main()
    {
      double x, xSum = 0.;
      int k;
      for (k = 1;   k <= 100;    k++) {
          x = 1. / (double)k;    // Calculate x = 1/k;
          x *= x;                // Square x;
          xSum += x;             // Add it to the sum.
          // *** Break out of the loop if x is less than epsilon
  ***
          if (x < epsilon)
               break;
      }
      cout << 'Pi is approximately equal to ' << sqrt(6.* xSum)
  << endl;
    }
```

The following function checks whether an integer *n* is prime. We have to check all potential factors *m* but only as long as $m^2 <= n$ (because if *m* is a factor, then so is *n/m* and one of the two must be less or equal to the square root of *n*). The function uses break to reduce the number of iterations:

```
  bool IsPrime(int n)
  // Returns TRUE if n is a prime, false otherwise.
  {
      int factors = 0;
      if (n <= 1)
```

```
            return false;
        for (int m = 2;   !factors;   m++) {
            if (m * m > n)
                break;
            if (n % m == 0)
                factors++;
        }
        return (factors == 0);
    }
```

NOTES

Another way to break out of the loop is to put a return statement inside the loop. For example:

```
    ...
    for (int m = 2;   ;   m++) {
        if (m * m > n)
         break;
        if (n % m == 0)
        return false;      // Not a prime.
    }
    return true;
    ...
```

In the above code, the condition in the for loop is empty. An empty condition is considered always 'true.' The break or return is used to break out of the loop.

There is a C++ idiom

```
    for(;;)
    ...
```

which means simply 'repeat.' The only way to get out of this loop is to use break or return.

The following code uses continue in calculating the sum and product of all primes less than or equal to N:

```
    ...
    const int N = 100;
        ...
        int sum = 0;
        long product = 1;
        int p;
        for (p = 2;   p <= N;   p++) {
            if (!IsPrime(p))
                continue;
            sum += p;
            product *= p;
        }
        ...
```

Note that although the increment statement is actually executed at the end of each iteration through a for loop, continue does not skip it.

Thus, in the above example, p++ is properly executed on every iteration.

This is *not so* with while loops, where the 'increment' statement is a part of the body of the loop.

Be careful with continue in while loops: it may inadvertently skip the increment statement, causing the program to hang.

This would happen in the following version of the above example:

```
    ...
    int p = 2;
    while (p <= N) {        // This code hangs for N >= 4,
because p
          if (!IsPrime(p))     //    never gets incremented
after
           continue;       //   the first non-prime p = 4 is
                             //   encountered.
        sum += p;
        product *= p;
        p++;
    }
    ...
```

A Word About `goto`

C++ also has the `goto` statement, which implements an unconditional jump to a specified statement in the program, but its use is considered highly undesirable because it violates the principles of structured programming. The format is:

```
    goto label;
```

where *label* is some name chosen by the programmer. The label, followed by a colon, is placed before the statement to which you want to jump. For example:

```
{
    ...
    if (cmd == 'y')
        goto quit;
    ...
 quit:
   return;
}
```

goto does not have to be inside a loop.

The use of the goto statement is strongly discouraged.

1.6.2 Logical Expressions and `if-else` Statements

The sequential flow of control from one statement to the next during program execution may be altered by the four types of control mechanisms:

1. Calling a function
2. Iterative statements (loops)
3. Conditional (if-else) statements
4. Switch statements

NOTES

We have already seen that calling a function is a convenient way to interrupt the sequential flow of control and execute a code fragment defined elsewhere. We have also used a while loop, which instructs the program to repeat a fragment of code several times. while is an example of an iterative statement; these are fully explained later in the unit.

In this unit we will study the if-else statement, which tells the program to choose and execute one or another fragment of code depending on the values of some variables or expressions. The if-else control structure allows *conditional branching*. Suppose for instance, we want to find the absolute value of an integer. The function that returns an absolute value may look as follows:

```
int abs(int x)
{
  int ax;
  if (x >= 0)       // If x is greater or equal to 0
     ax = x;        //   do this;
  else              // else
        ax = -x;      //   do this.
  return ax;
}
```

```
or, more concisely:
int abs(int x)
{
  if (x < 0)       // If x is less than 0
        x = -x;      //   negate x;
  return x;
}
```

There are special CPU instructions called *conditional jumps* that support conditional branching. The CPU always fetches the address of the next instruction from a special register, which, in some systems, is called the Instruction Pointer (IP). Normally, this register is incremented automatically after the execution of each instruction so that it points to the next instruction. This causes the program to execute consecutive instructions in order.

A conditional jump instruction tests a certain condition and tells the CPU to 'jump' to the specified instruction depending on the result of the test. If the tested condition is satisfied, a new value is placed into the IP, which causes the program to skip to the specified instruction (Figure 1.7). For example, an instruction may test whether the result of the previous operation is greater than zero, and, if it is, tell the CPU to jump backward or forward to a specified address. If the condition is false, program execution continues with the next consecutive instruction.

NOTES

```
        ...
        cmp AH,7             ; Test for BW card
        je M11               ; Goto BW card init
        xor ax,ax            ; Fill for graphics modes
        jmp short M13        ; Goto clear buffer
M11:    mov CH,08h           ; Buffer size on BW card (2048)
        mov AX,' '+7*256     ; Fill char for alpha
M13:    rep stosw            ; Fill the regen buffer with
        ...                  ;   blanks
```

Figure 1.8 *80 ´ 86 Assembly language code with the je ('Jump if Equal to 0') instruction.*

(There are also *unconditional jump* instructions that tell the CPU to jump to a specified address in the program by unconditionally placing a new value into the IP, for example, jmp short in Figure 1.7.)

In high-level languages, conditions for jumps are written using relational operators such as 'less than,' 'greater than,' 'equal to,' and so on, and the logical operators 'and,' 'or,' and 'not.' Expressions combining these operators are called *logical* or *Boolean* expressions in honor of British mathematician George Boole (1815–64), who studied formal logic and introduced *Boolean Algebra*, an algebraic system for describing operations on logical propositions. The value of a Boolean expression may be either true or false.

`if-else` Statements

The general form of the `if-else` statement in C++ is:

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

where *condition* is a logical expression and *statement1* and *statement2* are either simple statements or *compound statements* (blocks surrounded by braces). The else clause is optional, so the if statement can be used by itself:

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

When an if-else statement is executed, the program evaluates the condition and then executes *statement1* if the condition is true and *statement2* if the condition is false. When if is coded without else, the program evaluates the condition and executes *statement1* if the condition is true. If the condition is false, the program skips *statement1*.

'True' and 'False' Values

C++ does not have special values for 'true' and 'false.' Instead, 'true' is represented by any non-zero integer value, and 'false' is represented by the zero value. Thus a logical expression is simply a special case of an arithmetic expression: it is considered 'false' if it evaluates to zero and 'true' otherwise.

At the time of this writing, there is no built-in 'Boolean' data type in C++, nor are there predefined 'true' or 'false' constants. A proposal is being considered by the C++ Standards Committee to add the 'Boolean' data type to the C++ built-in types. The proposal would make true and false—the only two values a Boolean variable can take—reserved words.

If their compiler does not support the bool type, C++ programmers do it themselves. They might add the following definitions to their code (or place them in a header file):

```
typedef int bool;  // defines bool as an alias for int
#define false 0    // defines false as an alias for 0
#define true  1    // defines true as an alias for 1
```

After the bool type and the true and false constants have been defined, programmers can write declarations as follows:

```
...
bool someName = false;
...
```

Relational operators

C++ recognizes six relational operators:

Operator	Meaning
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
==	Is equal
!=	Is not equal

In C++, the result of a relational operation has the int (integer) type. It has a value equal to 1 if the comparison is true and 0 otherwise.

Note that in C++ the 'is equal' condition is expressed by the '= =' (double '=') operator, while a single '=' means assignment. Inadvertently writing = instead of = = renders your conditional statement worse than meaningless – *without* generating a syntax error. However, some of the more 'thoughtful' compilers do generate a warning against this pernicious bug.

Relational operators are mostly used for conditions in if statements and in iterative statements. Strictly speaking, they could be used in arithmetic expressions, too, because they have a value of 0 or 1 and will be promoted to the int type; however, such usage is unusual.

In C++, it is common to use integer variables or expressions as logical expressions. For example:

```
int count;
...
if (count)
   ...
```

The above is a C++ idiom, which means exactly the same thing as:

```
int count;
...
if (count != 0)  // if count not equal to 0
   ...
```

NOTES

Logical operators

C++ has two binary logical operators, 'and' and 'or,' and one unary logical operator, 'not.' They are represented by the following symbols:

Operator	Meaning
&&	and
||	or
!	not

The expression
```
    condition1 && condition2
```
is true if and only if both condition1 and condition2 are true.

The expression
```
    condition1 || condition2
```
is true if either condition1 or condition2 or both are true.

The expression
```
    !condition1
```
is true if and only if condition1 is false.

The following code:
```
bool match;
...
if (!match)
  ...
```
is identical to:
```
bool match;
...
if (match == false)
  ...
```
Like relational operators, the results of the logical operators &&, ||, and ! have the int (integer) data type: 1 represents true and 0 represents false.

The 'and,' 'or,' and 'not' operations are related to each other in the following way:
```
not (p and q) = not(p) or  not(q)
not (p or  q) = not(p) and not(q)
```
These two formulae are called *De Morgan's laws*. De Morgan's laws are properties of formal logic, but they are useful in practical programming as well. In C++ notation, De Morgan's laws take the following form:
```
!(p && q) == (!p || !q)
!(p || q) == (!p && !q)
```
A programmer may choose one of the equivalent forms; the choice depends on which form is more readable. Usually it is better to use ! (not). For example:
```
if (size <= 0 || a[0] == -1)
```

is much easier to read than:

```
 if (!(size > 0 && a[0] != -1))
```

Order of operators

NOTES

In general, all binary operators have *lower* precedence than unary operators, so unary operators, including ! ('not'), are applied first. You have to use parentheses if ! applies to the entire expression. For example:

```
 if (!x > 0)  //   You probably wanted:
         //   if (!(x > 0));
         //   You got:
         //   if ((!x) > 0);
     ...
```

Relational operators (>, < , = =, etc.) have lower rank than all binary arithmetic operations (+, *, etc.), so they are applied after the arithmetic operators. For example, you can write simply:

```
 if (a + b >= 2 * n)          // OK
    ...
```

when you mean:

```
 if ((a + b) >= (2 * n))       //  Redundant  inside
parentheses
     ...                      // needlessly clutter
                              // the code.
```

The binary logical operators && and || have lower rank than arithmetic and relational operators, so they are applied last. For example, you can write simply:

```
 if (x + y > 0 && b != 0)      // OK
```

as opposed to:

```
 if ((x + y > 0) && (b != 0))  // Redundant parentheses.
```

When && and || operators are combined in one logical expression, && has higher rank than || (i.e. && is performed before ||), but it is a good idea to always use parentheses to avoid confusion and make the code more readable. For example:

```
 // Inside parentheses not required, but recommended for
clarity:
 if ((x > 2 && y > 5) || (x < -2 && y < -5))
    ...
```

The rules of precedence for the operators that we have encountered so far are summarized in the table given below:

Highest	! (unary) – (cast) ++ -- sizeof
↑	* / %
	+ –
	< <= > >=
	== !=
↓	&&
Lowest	||

In the absence of parentheses, binary operators of the same rank are performed left to right, and unary operators right to left. If in doubt—use parentheses!

NOTES

Short-circuit evaluation

In the binary logical operations && and ||, the left operand is always evaluated first. There may be situations when its value predetermines the result of the operation. For example, if *condition1* is false, then *condition1* && *condition2* is false, no matter what the value of *condition2*. Likewise, if *condition1* is true, then *condition1* || *condition2* is true.

If the value of the first (left) operand in a binary logical operation unambiguously determines the result of the operation, the second operand is `not` evaluated. This rule is called `short-circuit evaluation`.

If the expression combines several && operations at the same level, such as

```
condition1 && condition2 && condition3 ...
```

the evaluation of conditions proceeds from left to right. If a *false* condition is encountered, then the remaining conditions are *not evaluated*, because the value of the entire expression is false. Similarly, if the expression combines several | | operations at the same level,

```
condition1 || condition2 || condition3 ...
```

the evaluation proceeds from left to right only until a *true* condition is encountered, because then the value of the entire expression is true.

The short-circuit evaluation rule not only saves the program execution time but is also convenient in some situations. For example, it is safe to write:

```
if (y != 0 && x/y > 3)
    ...
```

because x/y is not calculated when y is equal to 0. Similarly,

```
if (i >= 0 && i < SIZE && a[i] == 0)
   ...
```

makes sure that an element a [i] of the array is tested only when the index i is within the legal range between 0 and SIZE-1.

Case Study: Day of the week program

To illustrate the use of conditional statements and Boolean expressions, let us consider a program that deals with dates. The program will calculate the day of the week for a given date.

The dialog with the program, which we call Weekday, looks as follows:

```
Please enter a date (e.g. 11 23 2001) ==> 6 1 2000
6-1-2000, Thursday
```

It is easy to foresee which functions will be needed to carry out this task. We will need a function to check whether the entered date is valid. Then we will have to deal with leap years, which will require a function that determines whether a given year is a leap year. Of course, we also need a function that finds the day of the week for a given date. The easiest way to figure this out is to calculate the total number of days elapsed from some known fixed date, such as 01-01-1900, which was a Monday. We will need a function for that, too.

We will first implement these functions, then write the main program. This is called a *bottom-up* approach. The date functions we have identified are fairly general: they will help us to solve the problem at hand, but we can also use them later for other, similar tasks. By not focusing exclusively on a particular task but taking a slightly more general view of relevant functions, we can enhance the *reusability* of code, usually without extra cost for the project we are working on.

Let us begin with the LeapYear(...) function. This function takes an int argument (the year) and returns a Boolean value:

```
bool LeapYear (int year)
// 'year' must be between 1900 and 2999.
// Returns true if 'year' is a leap year.
{
        //  true, if year is divisible by 4, and ...
        //    ... either not divisible by 100, or divisible
by 400.
   return (year % 4 == 0 &&
                 (year % 100 != 0 || year % 400 == 0));
}
```

This function returns the value of one logical expression, which has the type bool. Extra parentheses around

```
(year % 100 != 0 || year % 400 == 0)
```

are important, because && has precedence over ||.

The other two functions require some tables: the number of days in each month and the number of days from the beginning of the year to the beginning of a month. We can implement them as global constant arrays:

```
const int daysInMonth[12] = {
    31, 28, 31, 30, 31, 30,
    31, 31, 30, 31, 30, 31
};
// Days from the beginning of the year
//   to the beginning of the month:
const int daysToMonth[12] = {
     0,  31,  59,  90, 120, 151,
   181, 212, 243, 273, 304, 334
};
const char *dayName[7] = {
    'Sunday',
    'Monday',
    'Tuesday',
    'Wednesday',
    'Thursday',
```

NOTES

```
     'Friday',
     'Saturday'
};
const int DAY0 = 1; // Day of week for 01-01-1900 is Monday
= 1.
```

The ValidDate(...) function simply verifies that the month, day, and year have valid values. This function returns a bool value - true if the date is valid, and false otherwise:

```
bool ValidDate (int month, int day, int year)
// returns true if month-day-year is a valid date between
//   01-01-1900 and 12-31-2999.
{
    bool valid = false;  // First assume the date is invalid.
     int days;
     // If year and month have valid values:
     if (year >= 1900 && year <= 2999 &&
                                 month >= 1 && month <= 12)
{
          // Get the number of days in this month from the
table:
          days = daysInMonth[month-1]; // (-1, because the
indices
                                        //   of an array start
from 0.)
        // If February of a leap year — increment the number
         //   of days in this month:
       if (month == 2 && LeapYear(year))
             days++;
         // Check that the given day is within the range.
         //   If so, set valid to true:
         if (day >= 1 && day <= days)
             valid = true;
     }
     return valid;
}
```

The third function calculates the total number of days elapsed since 01-01-1900. Let us call it DaysSince1900(...). This number may be quite large, more than 400,000 for the year 2999, and it may overflow an int variable if the int data type is implemented as a16-bit value. To be on the safe side, let's make this function return a long value:

```
long DaysSince1900 (int month, int day, int year)
// Returns the number of days elapsed since 01-01-1900
//  to month-day-year
{
     long days;
     // Calculate days to 01-01 of this year with correction
for
```

```
      //   all the previous leap years:
      days = long(year - 1900) * 365
          + (year - 1897) / 4          // +1 for each 4th year
           - (year - 1801) / 100         // -1 for each 100th
  year
           + (year - 1601) / 400;        // +1 for each 400th
  year
      // Add days for previous months with correction for
      //   the current leap year:
      days += daysToMonth[month-1];
      if (LeapYear(year) && month > 2) days++;
      // Add days since the beginning of the month:
      days += day - 1;
      return days;
  }
```

NOTES

Note the cast to long in the calculation of the number of days:

```
      days = long(year - 1900) * 365
          ...
```

Without it, an int value for the entire expression would be calculated first, because all the operands would have the int type and the result could be truncated before being assigned to the long variable days. Note also the use of integer division to compensate for the leap years since 1900:

```
          ...
          + (year - 1897) / 4          // +1 for each 4th year
           - (year - 1801) / 100         // -1 for each 100th
  year
           + (year - 1601) / 400;        // +1 for each 400th
  year
```

The expression +(year - 1897) / 4 adds a day to the calculation for every fourth year (starting in 1901) that has gone by. Likewise, –(year - 1801) / 100 subtracts a day for every 100th year that has passed. Finally, (year - 1601) / 400 adds a day back on for every 400th year.

The DayOfWeek(...) function returns an integer between 0 (Sunday) and 6 (Saturday). This function takes some known date (01-01-1900), takes its day of the week (Monday = 1), adds the number of days elapsed since that date, divides the result by 7 and takes the remainder:

```
  int DayOfWeek (int month, int day, int year)
  // Returns the day of the week for a given date:
  //   0 -- Sunday, 1 -- Monday, etc.
  {
      return int((DAY0 + DaysSince1900(month, day, year)) %
  7);
  }
```

Note that the expression has to be cast back into the int type, because DaysSince1900(...) returns a long value.

Check Your Progress

4. Define data and object with reference to OOP.
5. Explain the advantages of abstraction.
6. What is data abstraction?
7. Describe the characteristics of OOP?

Finally, we can create the main program that implements the user interface and calls the functions described above:

This program calculates on which day falls the user's birthday (or any other date).

```
#include <iostream.h>
/                                                          /
***********************************************************
//*************   Tables and functions    ******************
/                                                          /
***********************************************************
...
/                                                          /
***********************************************************
//*************           main          ******************
/                                                          /
***********************************************************
void main()
{
    int month, day, year;
    int weekday;
    cout << 'Please enter a date (e.g. 11 23 2001) ==> ';
    cin >> month >> day >> year;
    if (!ValidDate(month, day, year)) {
        cout << '*** Invalid date ***\n';
        return;
    }
    weekday = DayOfWeek(month, day, year);
   // Display the entered date and the name of the calculated
    //  day of week:
    cout << month << '-' << day << '-' << year << ', '
         << dayName[weekday] << endl;
}
```

Note the expression:

```
if (!ValidDate(month, day, year)) {
        ...
```

which reads: 'if *not* ValidDate...'

Lab: Holidays

Write a program that calculates and displays the dates of Labor Day (first Monday in September), Memorial Day (last Monday in May), Thanksgiving (fourth Thursday in November), and Election Day (first Tuesday after the first Monday in November) for a given year. Reuse the functions and definitions from the WEEKDAY program, but replace the main program with new code. The program should prompt the user for a desired year and calculate and display the holiday names and their respective dates.

`if-else if` and Nested `if-else`

Sometimes, a program needs to branch three or more ways. Consider the sign(x) function:

$$sign(x) = \begin{cases} -1, \text{ if } & x < 0; \\ 0, \text{ if } & x = 0; \\ 1, \text{ if } & x > 0. \end{cases}$$

NOTES

sign(x) can be implemented in C++ as follows:

```
int Sign (double x)    // Correct but clumsy code...
{
    int s;
    if (x < 0.)
        s = -1;
    else {
        if (x == 0.)
            s = 0;
        else
            s = 1;
    }
    return s;
}
```

This code is correct, but it is a bit cumbersome. The x < 0 case seems arbitrarily singled out and placed at a higher level than the x = = 0 and x > 0 cases. Actually, the braces in the outer else can be removed, because the inner if-else is one complete statement. Without braces, the compiler always associates an else with the nearest if above it. The simplified code without braces looks as follows:

```
int Sign (int x)   // Correct, but still clumsy...
{
    int s;
    if (x < 0.)
        s = -1;
    else
        if (x == 0.)
            s = 0;
        else
            s = 1;
    return s;
}
```

It is customary in such situations to arrange the statements differently: the second if is placed next to the first else and one level of indentation is removed, as follows:

```
int Sign (int x)   // The way it should be...
{
    int s;
    if (x < 0.)
        s = -1;
```

NOTES

```
    else if (x == 0.)  // This arrangement of if-else is a
matter
        s = 0;         //   of style.  The second if-else is
    else               //   actually nested within the first
else.
        s = 1;
    return s;
}
```

This format emphasizes the three-way branching that conceptually occurs *at the same level* in the program, even though technically the second *if-else* is *nested* in the first *else*.

A chain of if-else if statements may be as long as necessary:

```
if (condition1) {
    ...                  // 1st case
}
else if (condition2) {
    ...                  // 2d case
}
else if (condition3) {
    ...                  // 3d case
}
...
...
else {
    ...                  // Last case
}
```

This is a rather common structure in C++ programs and it is usually quite readable. For example:

```
...
if (points >= 92)
   grade = 'A';
   else if (points >= 84)
      grade = 'B';
      else if (points >= 70)
         grade = 'C';
         else if (points >= 55)
            grade = 'D';
            else
                  grade = 'F';
...
```

A different situation occurs when a program requires true hierarchical branching with nested if-else statements, as in a decision tree:

```
                •
               / \
          if(…)  else
           /       \
          •         •
         / \       / \
    if(…)  else  if(…) else
      /      \    /     \
```

NOTES

Consider, for example, the following code:

```
...
// Surcharge calculation:
if (age <= 25) {
   if (accidents)
      surcharge = 1.4;  // Premium surcharge 40%
   else
      surcharge = 1.2;  // Surcharge 20%
   }
else {    // if age > 25
   if (accidents)
      surcharge = 1.1;  // Surcharge 10%
     else
      surcharge = .9;   // Discount 10%
   }
...
```

Here the use of nested if-else is justified by the logic of the task. It is possible to rewrite the second part of it as if-else if, but then the logic becomes confusing:

```
...
// Surcharge calculation (more confusing):
if (age <= 25) {
    if (accidents)
        surcharge = 1.4;  // Premium surcharge = 40%
    else
        surcharge = 1.2;  // 20%
}
else
    if (accidents)
       surcharge = 1.1;  // Premium surcharge = 10%
    else
       surcharge = .9;   //  Discount 10%
}
...
```

NOTES

When if-else statements are nested in your code to three or four levels, the code becomes intractable. This indicates that you probably need to restructure your code, perhaps using separate functions to handle individual cases.

Nested if's can often be substituted with the && operation:

```
  if (condition1)
      if (condition2)
          statement;
is exactly the same (due to the short-circuit evaluation)
as:
  if (condition1 && condition2)
      statement;
```

Common `if-else` Errors

ERROR	COMMENTS	CORRECTED

// 1. Extra ';'
 	
if (condition);
  statement;
	// 1. Compiled as
	
	if(condition)
	  ; // do nothing
	statement;
		// 1.
		
		if (condition)
		  statement;

/*****************/	/****************/	/*****************/

// 2. Missing ';'
//   before "else"
 	
if (condition)
  statement1
else
  statement2;
	// 2.
	
	Syntax error
	 caught by
	 the compiler
		// 2.
		
		if (condition)
		  statement1;
		else
		  statement2;

/*****************/	/****************/	/*****************/

// 3. Omitted {}
 	
if (condition)
  statement1;
  statement2;
	// 3. Compiled as
	
	if (condition)
	   statement1;
	statement2;
		// 3.
		
		if (condition) {
		  statement1;
		  statement2;
		}

/*****************/	/****************/	/*****************/

// 4. "Dangling
      else"
 	
if (condition1)
  if (condition2)
    statement1;
else
  statement2;
	// 4. Compiled as
	
	if (condition1) {
	  if (condition2)
	    statement1;
	  else
	    statement2;
	}
		// 4.
		
		if (condition1) {
		  if (condition2)
		    statement1;
		}
		else
		  statement2;

1.7 ARRAYS

An array is a user-defined data type. There can be an array of integers, of characters, of floating point numbers, and so on. What is an array? It represents a set containing one or more elements of integers or floating point numbers or a number of items of the same data type. For instance A = {2, 3, 5, 7}

Here, A is a set of prime numbers or an array of prime numbers.

B = {Red, Green, Yellow}

Here, B is an array of colours.

Thus the set has a name, which is A in the former case and B in the latter. How do we give a name to each element? While in Mathematics the first element can be assigned a subscript 1, in C++ the first element always has the subscript 0.

Thus A [0] = 2

A [1] = 3

A [2] = 5

A [3] = 7

Similarly B [0] = Red

B [1] = Green

B [2] = Yellow

A has four elements, therefore the size of array A is 4. Similarly, B has three elements and hence the size of array B is 3. However, the first element of any array will have a subscript of 0 and the final element a subscript of *n*-1, where *n* is the size of the array.

Array declaration

Suppose there are 40 employees in an office and if we want to store their age, this can be stored in an array of size 40. The two statements below declare an array.

```
int emp_age [40];
```

This declaration allocates space for storing 40 integer elements in consecutive locations in the memory. Here we have declared an integer variable known as emp_age with a dimension of 40. But for this feature, we have to write 40 lines to declare the same with 40 different names.

Some examples of one-dimensional arrays are given below:

```
float mark[100];
char name[25];     /*  name contains 25 characters  */
```

Let us now look at a program involving arrays.

Program 1.12

```
/*Program to demonstrate arrays*/
#include<iostream>
int main(){
float emp_age[3];
emp_age[0]=44.0f;
emp_age[1]=24.5f;
emp_age[2]=55.2f;
std::cout<<"age of third employee=  "<<emp_age[2];
}
```

In this program we have declared an array of float and assigned the values, element by element. Then we are printing the age of the third employee. Note how we access individual elements of an array. The result of the program is given below.

NOTES

Result of Program 1.12

```
      age of third employee=  55.2
```

Array initialization

If the elements of an array are known beforehand, they can be declared at the beginning itself. Suppose the employees' ages are known, they can be declared as:

```
int   emp_age [] = {40, 32, 45, 22, 27};
```

The data elements are written within braces and separated by commas. In this case, where the data elements are declared, there is no need to declare the size (as given above).

Now let us try a program.

Program 1.13

```
/*Program to demonstrate arrays*/
#include<iostream>
int main(){
int emp_age[]={40, 55, 24};
std::cout<<"age of third employee= "<< emp_age[2];
}
```

This program does the same as the previous program, and is simple.

Result of Program 1.13

```
age of third employee= 24
```

Till now we were looking at one-dimensional arrays. Now let us look multidimensional arrays.

Multidimensional arrays

The multidimensional array operates on the same principles. We have to declare the dimensions of a two-dimensional array with two subscripts.

```
int student[10][5]
```

is a two-dimensional array with two different subscripts.

Here there will be 50 (10 x 5) different elements. The first element can be denoted as student[0][0]. The next element will be student[0][1]. The fifth element will be student[0][4]. The last element will be student[9][4].

This can be considered as a row and column representation. There are 10 rows and 5 columns in this example. When data is stored in the array, we traverse from the first element to the last, at first the second subscript will change from 0 to 4 with first subscript remaining constant at 0. Then the first subscript will become 1 and the second subscript will keep increasing from 0 to 4. This repeats till the first subscript becomes 9 and the second 4. This array can be used to store the names of 10 persons, each name containing 5 characters. The first subscript refers to the name of the 0th person, 1st person, 2nd person and so on. The second subscript refers to the 1st character, 2nd character and so on of the name of a person. Thus, 10 such names can be stored in this array.

The dimension of the array can be increased to 3 and represented as shown below:

```
Marks [50][3][3];
```

The first element will be Marks [0][0][0]

The last element will be Marks [49][2][2].

It is easy to add more dimensions to array, can become difficult to comprehend in the normal circumstances. Such arrays, however, may be useful for solving complicated scientific applications. Now let us understand the concept of multidimensional arrays using a simple problem.

We want to write a program to multiply the corresponding elements of two arrays (both two-dimensional) and store them in another two-dimensional array. To make the problem simpler, we will use [2][2] arrays.

Let us call the arrays x, y and z.

```
We have   x = {x[0][0], x[0][1] }  y =  {y[0][0],  y[0][1] }
              {x[1][0],  x[1][1] }       {y[1][0], y[1][1]  }
```

We want to multiply x [0][0] and y [0][0] and store the result in z [0][0] and so on.

The values of x and y are given in the program itself.

x = {1 2}   y = {5 6}

{3 4 }   {7 8}

x [0][0] = 1   x [1][1] = 4

y [0][0] = 5   y [1][1] = 8

Therefore, after multiplication of the respective elements, we get

z = {5 12}

{21 32}

The program prints out the values of the products stored in array z.

Now look at the program.

Program 1.14

```
/*Program to demonstrate 2 dimensional arrays*/
#include<iostream>
using namespace std;
int main()
{
int x[2][2] = {1,2,3,4};

int y[2][2] = {5,6, 7,8};
int z[2][2];

z[0][0]=x[0][0]*y[0][0];
z[0][1]=x[0][1]*y[0][1];
z[1][0]=x[1][0]*y[1][0];
z[1][1]=x[1][1]*y[1][1];
```

NOTES

```
 cout<<"Resultant matrix is given below\n";
 cout<<"{ "<<z[0][0]<<"  ,"<<z[0][1]<<"}\n";
 cout<<"{ "<<z[1][0]<<" ,"<<z[1][1]<<"}";
 }
```

See carefully how arrays x and y are initialized. It is interesting to see that the values are entered in a row.

The output appears as follows:

Result of Program 1.14

```
Resultant matrix is given below
{5 ,12}
{21,32}
```

A multidimensional array can be considered to be an array of one-dimensional arrays. However, depending on the dimensions of the array, values are assigned to the corresponding elements, as the above example illustrates. Also note that declaration of an array becomes very simple when values are assigned immediately on creation of the array.

Size of data types

The size of built-in data types are fixed whereas that of the user defined types may vary. Size refers to the amount of space required for storing them in memory. The unit for size is number of bytes. User defined data types such as arrays will occupy the size actually required. For instance, let us declare an array as given below:

```
int arr[2][4][5];
```

The number of elements in the above array will be equal to 2 x 4 x 5 = 40. The 40 integer elements will occupy 40 x 4 = 160 bytes. To find out the size of in-built and user defined data type, we can use the keyword sizeof(), as you will see in Program E2x11.

Program 1.15

```
/*to demonstrate finding size of
built in and user defined types */
#include<iostream>
using namespace std;
int main()
{
cout<<"\n size of bool   ="<< sizeof(bool);
cout<<"\n size of char   ="<< sizeof(char);
cout<<"\n size of short  ="<< sizeof(short);
cout<<"\n size of int    ="<< sizeof(int);
cout<<"\n size of long   ="<< sizeof(long);
cout<<"\n size of float  ="<< sizeof(float);
cout<<"\n size of double ="<< sizeof(double);
cout<<"\n size of long double="<< sizeof(long double);
int arr[2][4][5];
cout<<"\n size of user defined arr="<< sizeof(arr);
}
```

NOTES

Result of Program 1.15

```
size of bool   =1
size of char   =1
size of short  =2
size of int    =4
size of long   =4
size of float  =4
size of double =8
size of long double=12
size of user defined arr=160
```

In a similar manner, the size of any user defined data type also can be found out.

`goto` keyword

This is one keyword which should not be used in any program because it causes problems during maintenance. However, for the sake of completion of the learning of C++, it has used only in this case study.

We might like to go to one of the statements in the function depending on some outcome. Such transfers from one statement to another can be implemented by using the goto keyword. For instance,

```
Start: statement1;
if (a>b) goto Start ;
```

In the above fragment of code, we have defined a label called Start. The label is any valid identifier followed by a colon (:). Somewhere in the same function and depending on a particular outcome, we may want to go to the given label. In this case, it is Start. Thus, whenever a program has to go to a label, we say goto, followed by the label. Then, the program jumps to the statement following the label declaration.

We will also use an if statement in the case study.

The case study is for a binary search. Searching and sorting are popular operations in computer science. An array can consist of any valid data type and can be sorted in an ascending or descending order. A search operation is used to find out whether a given data type is available in the list or array. A binary search is carried out on an array which is already sorted. A good example is telephone directory, which is a list consisting of names arranged in ascending order.

We should understand the binary search algorithm before developing a program for the same. The binary search algorithm is explained below.

Algorithm for binary search

For the sake of simplicity we will assume an array size of 5.

Declarations

```
int Array [5] = {100, 200, 300, 400, 500} // Declaring and
initializing an array
```

```
int left = 0, right = 4, mid = 0 ; //  Index left, right and
middle declared
boolean found = false ;
int key ; // The item to be searched is called key.
Enter the key
Label Start: mid = (left + right) / 2
if Array [mid] = = key, then found = = true.
else
if Array [mid] < key, left = mid + 1
else
right = mid – 1
if ((found = = false) && (left <= right)) goto Start ;
if (found = = true) print key found
else
print key not found
```

A program implementing the above algorithm is given below:

```
//Case Study Binary Search
/*Program for demonstrating arrays*/
#include<iostream>
using namespace std;
int main(){
int left=0, mid=0, right=4;
bool found=false;
int key;
int array[]={100, 200, 300, 400, 500};
cout<<"Enter the key to search \n";
cin>>key;
Start:  mid=(left+right)/2;
        if(array[mid]==key)
        {
        found=true;
        goto End;
        }
        else
        if((array[mid]) < key)
        left=(mid+1);
        else
        right=(mid-1);
if ((left<=right)&&(found==false)) goto Start;
End:if(found==true)  cout<<"found  the  key  in
“<<(mid+1)<<“position”;
        else
    cout<<“key NOT found”;
               }
```

NOTES

The program was tested with both valid and invalid inputs. A valid input is one of the numbers in the array. The invalid input is a number not in the array. The result of the case study is given below:

Result of Case Study

```
Enter the key to search
400
found the key in 4position

Enter the key to search
600
key NOT found
```

The above case study illustrates the usage of arrays and implementation of the binary search algorithm, which is quite popular in computing.

1.8 SUMMARY

In this unit, you have been taught the basic concepts of object oriented programming. You have seen that there are six types of tokens in C++ programming language, the important ones being identifiers, constants and operators. Identifiers have to start with a letter and can contain an unlimited sequence of letters, digits and underscore. There are 74 keywords or reserved words for programming in C++. The fundamental or primitive data types are:

```
short
int
long
float
double
char
long double
```

The size of a data type depends upon its type. The maximum and minimum values for each type also vary. Floatingpoint numbers can be expressed in the normal form such as 10.3 or in the exponential notation such as 10.3 E3. Using unsigned representation can double the range of values that could be stored in a data type. There are rules for constructing the various types of constants or literals. A string literal should begin and end in the same line. The suffixing rules for real constants are as given below:

```
Float: f
Double:
```

The rules for Integers are:

```
Long: L or l
Unsigned: U or u
```

Constants are defined using the keyword *const*. They cannot be reassigned any other value in the program, but a variable can be. This is the difference between constants and variables.

The definition and scope of variables have also been discussed. A variable is visible in the block in which it is declared. It is created when the program enters its scope and is destroyed when it leaves the scope. Functions can be nested. Variables can be initialized dynamically. You have learned about reference variables using & operators. The reference variables function helps to assign two names to one variable.

Arrays are variables which contain more than one element of the same data type. In C++, an array subscript starts at 0 and ends at *n*-1 for an array of size *n*. An array can be declared as one-dimensional or multidimensional. The type of an array such as int, float or char must be declared before its usage. The initial values of array elements can also be declared. Multidimensional arrays and the operation for using them have also been discussed. A two-dimensional array can be considered to be in the form of rows and columns. The first element of an array indicates the number of rows while the second element indicates the number of columns.

In this unit, the usage of the `goto` keyword has also been discussed. Finally, the binary search algorithm has been explained with the help of a case study.

1.9 KEY TERMS

- **Data abstraction:** In object oriented programming, each object will have external interfaces which can be made use of. The object itself may be made of many smaller objects, again with proper interfaces. The internal details of the objects are hidden which makes them abstract. The technique of hiding internal details in an object is called data abstraction.
- **Class:** It is a blueprint for defining an object which has similar data and functions. A class is a user-defined type consisting of data members and member functions.
- **Objects:** These are types of classes with unique data.

1.10 ANSWERS TO 'CHECK YOUR PROGRESS'

1. The principle of object oriented programming is to combine both data and the associated functions into a single unit called a class.
2. The following are the advantages of OOP:
   1. ***Reusability***-The object oriented programs are designed for reuse. Encapsulation, polymorphism and inheritance facilitate ease of reuse.
   2. ***Maintainability***-Each class is self-contained with data and functions, and the functions are grouped together providing easily maintainability.
   3. ***Modular***-Modularity in OOP is obtained by the division of the program into

well defined and closely knit classes.

4. ***Extensibility***-Inheritance mechanism facilitates extending the feature of the classes easily.
5. ***Data Integrity***-Avoiding global variables and goto statements, binding the data within the class and providing restricted access ensures data integrity.

3. The new features added to standard C++ include the following:
   - Namespaces
   - Exceptions
   - Templates
   - Runtime type identification
   - Standard library including Standard Template Library(STL)
4. Data members establish the state or attributes for an 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 nucleus, or 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.
5. The advantage of abstraction is that we are able to understand an object's overall characteristics without looking at other diversionary details.

   The object in turn may have a number of distinct objects with unique states and behaviour. This helps the user to use an object with minimum information, i.e., just enough information about how to interact through the external interface. Object oriented programming is based on the concept of hierarchical abstraction.
6. In object oriented programming, each object will have external interfaces through which it can be used. The object itself may be made of many smaller objects, again with proper interfaces. The user needs to know the external interfaces only to make proper use of an object. The internal details of the objects are hidden which makes them abstract. The technique of hiding internal details in an object is called data abstraction.
7. The following are the three essential characteristics of an object oriented program:
   - Encapsulation
   - Inheritance
   - Polymorphism

   Encapsulation: Wrapping together data and functions creates objects in OOP. They are bound together. This represents encapsulation in OOP. We can use the encapsulated objects through the designated interfaces only. Thus, the inner parts of the program are sealed or encapsulated to protect from accidental tampering. This feature is not available in the conventional procedure oriented programming languages where the data can be corrupted since it is easily accessible.

   Inheritance: Inheritance is the property in which a new class can derive its properties from another existing class. Thus, it becomes a derived class of the parent class, which is called a base class. A derived class inherits all the properties of the base class. The complexity of the derived class may grow as the level of inheritance grows. There is no limit to the level of inheritance.

Polymorphism: Polymorphism is a useful concept in OOP. It provides a common interface to carry out similar tasks. In other words, a common interface is created for accessing related objects. This facilitates the objects to become interchangeable black boxes. Hence they can be used in other programs as well. Therefore, polymorphism enables reuse of objects built with a common interface.

1.11 QUESTIONS AND EXERCISES

Short-Answer Questions

1. State whether True or False:
   a. C++ language has a reserved word for the print command.
   b. There are 24 reserved words in C++ language.
   c. case is a reserved word.
   d. There are three classes of tokens in C++ language.
   e. Constants are one of the classes of tokens.
   f. Formula and FORMULA identifies the same variable.
   g. A character constant or a variable occupies 2 bytes in storage.
   h. A long integer occupies 4 bytes in memory.
   i. The exponent of a floating point number expressed in scientific notation can be a floating point number.
   j. String constants can have one character within double quotes.
   k. Any Escape sequence is treated as a single character.
   l. All Eescape sequences start with a back slash.
   m. /* Indicates beginning of a comment statement.
   n. Dynamic allocation means declaration of variable and assignment are carried out in 2 steps.
   o. An array can contain floats and integers, both together.
   p. In C++, the first element of an array always has a subscript of [0].
   q. Arrays help in simplifying the declaration of variables of the same type.
   r. A four-dimensional array will contain 4 pairs of square brackets after the variable name.
2. Check which of the following are valid integer constants:
   a) 4.13
   b) 4.0
   c) ‘z’
   d) 240L
   e) 1E10
   f) + 240

g) - 245
h) ± 245

3. Check which of the following are valid character constants:
   a) ‘c’
   b) ‘C’
   c) A
   d) ‘35’
   e) ‘\0’
   f) ‘character’
4. Check which of the following are valid real number constants:
   a) 123
   b) 123 E 2
   c) 123.0 E 2
   d) 123.0 e 2
   e) 3350.0001L
   f) -267.1 E 2 f
   g) 335.02 e - 2
   h) 225.12 e 2.5
5. Check which of the following are valid string literals:
   a) “me”
   b) ‘me’
   c) ‘u’
   d) “U”
   e) “this is a string literal”
   f) “this is
   a string literal”
6. Match the following:

	A	B
a)	short int constant	1. 246 L
b)	string constant	2. ‘c’
a)	char constant	3. 5000
b)	double constant	4. 3.4 E 20
c)	scientific notation	5. 423.4f
d)	long integer	6. 5432.7
e)	float	7. 345
f)	integer	8. “C”

7. Write a program to find p^4, by getting the value of p from a standard input device.
8. Write a program to find the difference between two numbers entered through a keyboard.

NOTES

9. Find out the syntax error, if any, in the following statements:
   a) int var x=y*y
   b) var2 A=2B+C-E;
   c) double & Var Var3;
   d) float emp_age[];
   e) mp_age[0],54.0f;
   f) float var1=10.0E1; var2=20.0f, var3=30.0;
   g) var4=(var1+var2+var3)3;
   h) std:cout<<"\n";
10. Explain the important features of OOP.
11. Why C++ is better than C and other programming languages?
12. Define Class.
13. What are the reserved words?
14. Define the importance of a scope resolution operator.
15. What is a standard library?
16. Explain the advantage of functions in a program.

Long-Answer Questions

1. Write a program for the following:
   (i) To add three numbers.
   (ii) To print as follows:
   ```
   1   2   3   4      5
   6   7   8   9      10
   ```
   (iii) To evaluate the following expression :
   44/2 + 235*22
   (iv) To display the following text in font size of 20
   Welcome
   To
   Programming in C++
2. Write short notes on the following:
   (i) Encapsulation
   (ii) Polymorphism
   (iii) Inheritance
   (iv) Class
   (v) Object and abstraction
3. Find the errors, if any, in the following statements:
   1. `#include <namespace>`
   2. `#include iostream`

3. `std::c out<<"Om";`
4. `std::cin<<var1;`
5. `std::cout>>var2;`
6. `std::cout>>" ;`

NOTES

UNIT 2 FUNCTIONS

NOTES

Structure

2.0 INTRODUCTION

In this unit, you will learn the concept of functions in a programming language. Functions mean operations. You will learn how functions are used in a program with a group of statements performing specific operations. The function main, written as `main()`, is an important function for starting the execution of a C++ program. The functions implement the behaviour of objects in a program. The dynamic properties of objects are facilitated only through functions. Thus, functions provide interfaces for communicating with the objects.

You will also learn about user-defined functions. These functions are coined by the user to fulfil specific requirements. A function has to be declared before using it. It may be declared either in the main function or in a class. A general function consists of three parts, namely, function declaration (or prototype), function call and function definition. Function declaration has to be done in the standard format known as function prototype. A function definition consists of a header matching the prototype, followed by a set of statements.

You will also learn that even when a program does not return a value, the program control returns to the calling function and resumes execution from that point. We can call a single function, or many, in a program, using 'call by value'. We call the function by giving values in basic form. A function called by value can return only one value. You will learn how an array can be passed to a function. An `inline` function can be declared by prefixing the keyword `inline` to the function prototype. Inline functions are suitable for small functions. A pointer is a variable that contains the address of a data type. In C++, pointers can be applied not only to built-in types but also to user-defined types such as objects. You will also learn that arrays and structures have similarities as well as differences between them. Both represent the collection of a number of data items. An array is a collection of items of the same data type, whereas a structure is not. But structures can represent items of varying data types pertaining to an item. A structure is synonymous with records in a database. It contains fields or variables. The variables can be of any of the valid data types.

NOTES

2.1 UNIT OBJECTIVES

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

- Define a function
- Understand the concept of function arguments
- Declare a function
- Call a function
- Understand the concept of passing by value
- Define arrays and pointers
- Understand functions and strings
- Define functions and structures
- Write a C++ program using functions and pointers

2.2 DEFINING A 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 # (pronounced as Sharp). The behaviour 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. We have used overloaded operator functions such as + and == with strings in the previous unit. In addition to library functions, the user can develop additional functions. In this unit we will discuss about user-defined functions only.

Three requirements of a function

Deployment of a function requires the following:

i. Function declaration
ii. Function call – passing arguments
iii. Function definition

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 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, we may declare a function as,

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

If it does not return any value at all, we 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, we give the data type of the argument and the name of the variable – for instance, `float arg1,` 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. We can even omit it and write:

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

The programmer can use either method.

NOTES

2.3 FUNCTION ARGUMENTS

Function call — passing arguments

We call the function when needed, either directly or indirectly. When we call the function, we pass actual arguments or values. Calling a function is also known as function reference.

There must be one to one correspondence between argument types in the declaration and actual arguments as sent with the call. They should be of the same data type and in the same order.

Function definition

The function definition is the actual body of the function. It can be written anywhere in the file with a declarator or a function header as in the declaration, followed by declaration of local variables and statements. Thus the function definition consists of two parts namely, function header and function body.

We will demonstrate the use of functions with a simple program.

Program 2.1

```
/* use of function*/
#include <iostream>
using namespace std;
int main(){
    int a=0, b=0, sum=0;
    int add(int,int); /*function declaration*/
    cout<<"enter 2 integers\n";
    cin>>a>>b;
    sum =add(a, b); /*function call*/
```

NOTES

```
    cout<<"\n sum of "<< a << " and "<< b<<"="<< sum;
}
/*function definition*/
int add (int c, int d) /*function declarator*/
{
    int e;
    e= c+d;
    return e;
}
```

Result of Program 2.1

```
enter 2 integers
345
678
sum of 345 and 678=1023
```

In line 6 of the example, the declaration of function 'add' is given. Note that the function will return an integer. Hence, the return type is defined as int. The arguments are declared as int and int. As a variable cannot be used without declaring, function also cannot be used with out declaration.. Also note that a function declaration ends with a semicolon, similar to declaration of any other variable. Function declaration should appear in the calling function before the function is called. It provides hint to the compiler that the function is going to call another function add, later in the program. In the above example, we get the values for `a` and `b` from the keyboard. After that we call the function add and assign the value returned by the function to an already declared int variable called sum as,

```
sum = add (a,b);
```

Note that `add(a,b)` is the function call or function reference. Here, the return type and the data type of the arguments are not to be given. But the function name and the names of the arguments passed, if any, should be there in the function call. When a function reference or function call is encountered, a search is made for the appropriate function and the arguments are transfered to it.

The function definition consists of two parts i.e. the function declarator or the function header and function body.

i. Function header is a replica of function declaration. The only difference is that the declaration in the calling function will end with semicolon and the declarator in the called function will not end with semicolon.
ii. Similar to the main function, the entire function body will be enclosed within braces. The set of statements between the braces forms the function body.

In this program, we have a statement 'return e' before the closing brace, which sends the program control to the main function with values of e. This value will be assigned to sum as,

```
sum = (returned value)
```

Therefore, sum receives the value returned which is then outputted, in the next statement. This is how the function works. Even when the program does not return a value, the program control returns to the calling function and resumes from there.

2.4 PASSING BY VALUE

NOTES

We can call a single function or multiple functions in a program using 'call by value'. We call the function by giving values in basic form. In call by value only one value, can be returned.

The return value, as we have seen, is the result of computation in the called function. We return a value, which is stored in a data type in the function. The return statement implies that a value thus computed in the called function is assigned or copied to a variable in the calling function i.e. main. The return statement can be in any of the following form:

```
return (sum) ;
return ' Z ' ;
return 0;
```

We can even return expressions. If return statement is not present, then it means return data type is `void`.

We can also have multiple return statements in a function. But, for every call, only one of the return statements can be reached and only one value will be returned.

2.5 ARRAYS AND POINTERS

Pointer to function

We can declare a pointer to a function. For instance, the following declares a pointer to a function returning void:

```
void  (*fung) (arg list);
```

A function can return a pointer of any type. For instance, the following prototype is for a function-returning pointer to character:

```
char * fung(arg list);
```

Note carefully the difference between declaration of pointer to function and function returning pointers. Now let us execute a few programs involving pointers.

Function call by reference

We want to pass *var1* and *var2* to a function; and divide *var1* by *var2* and we want to get both the quotient and remainder to the main function. It is not possible to do this by call by value. Call by value can return a single value only, either the quotient or remainder, but not both. This can be achieved by call by reference as the following example demonstrates.

Program 2.2

```
/* to demonstrate function call by reference*/
#include <iostream>
using namespace std;
int main(){
int var1=100, var2=13;
void div(int *, int *);/*indicates call by reference*/
div(&var1, &var2);/*addresses of var1 and var2 are passed*/
cout<<"quotient ="<<var1<<" remainder= "<<var2;
}
/*function definition*/
void div(int *px, int *py)   /*function declarator*/
{
int temp1, temp2;
temp1=*px;
temp2=*py;
*px=temp1/temp2;
*py=temp1%temp2;
}
```

Result of Program 2.2

```
quotient =7 remainder= 9
```

How does the program work?

In the declaration part, we have declared *div* as a function passing two pointer variables and getting back void.. We call div(&var1, &var2). We don't pass the values, but reference to the values. We actually pass the address of *var1* & *var2* to function *div*. Thus we are calling the function by reference.

The function *div* receives the reference i.e. addresses of var1 & var2.

```
px  =  &var1;
py  =  &var2;
```

px points to *var1* and *py* points to *var2*. Hence *px gets the value of *var1* and *py gets the value of *var2*. Now the contents of *px and *py i.e. *var1* and *var2* are copied to *temp1* and *temp2* respectively.

We divide *temp1* by *temp2* and place the result in variable *px whose address is known to both main and the function *div*. In the main function the address corresponds to var1 and in the function *div* it corresponds to **px*. Therefore the address of quotient is returned to the main function indirectly. Similarly ***py** contains the remainder. It will be stored in *var2* through the reference. Thus, the values of quotient and remainder are stored in locations &var1 and &var2 of the main function. Thus &var1 and &var2, after execution of the function *div* contain the quotient and remainder respectively. Thus, we have indirectly returned two values to main function through call by reference.

The function declaration indicates that there is a function *div,* which returns nothing. It passes two pointers to integers. The arguments are declared as int* and int* to indicate that they are pointers to integers. They also indicate that addresses of two integers are to be passed while calling the function. The function declarator before the function body should match the prototype.

There is a close relationship between arrays and pointers in C++. Suppose we have declared an array of 100 elements of the data type double:

```
double a[100];
```

The elements of the array can be referred to in the program as a[0] ... a[99]. When the program is compiled, the compiler does not save the addresses of all the elements, but only the address of the first element, a[0]. When the program needs to access any element, a[i], it *calculates* its address by adding i units to the address of a[0]. The number of bytes in each 'unit' is, in our example, equal to the sizeof(double) (e.g. 8). In general, it is equal to the number of bytes required to store an element of the array.

The address of a[0] can be explicitly obtained using the & ('address of') operator: &a[0]. Since the data type of a[0] is double, the data type of &a[0] is, as usual, double* (pointer to double).

C++ allows us to use the name of the array a, without any subscript, as another name for &a[0].

The name a can be used as an rvalue of the type double*. It cannot be used as an lvalue, because it cannot be changed. We can assign this value to any double* variable. For example:

```
double a[100];
double *p;
p = a;  // Same as p = &a[0];
...
```

As long as p points to the first element of the array, *p becomes an alias for a[0]. In general, we can assign

```
 p = &a[i];
```

Then *p becomes temporarily an alias for a[i].

C++ supports arithmetic operations on pointers that mimic calculations of addresses of array elements. In this pointer arithmetic, we can add an integer to a pointer or subtract an integer from a pointer. For convenience, the integer operand signifies 'units' corresponding to the pointer's data type, not bytes.

For example, if we have

```
double a[100];
double *p;
p = &a[0];
```

then p+1 points to a[1], p+2 points to a[2], and so on. The actual difference in bytes between p+1 and p is sizeof(double) (e.g. 8).

If p is equal to a, then we can refer to a[1] as *(p+1) and, in general, to a[i] as *(p+i).

We can also increment and decrement a pointer by using the ++ and -- operators.

In the expression *p++, the increment operator applies to the *pointer*, and *not to the value to which it points*. It means: take the value *p, then increment p, (*not* the dereferenced value *p).

The statement	Is the same as
`x = *p++;`	`{ x = *p; p++; }`

Relational operators <, >, <=, >= can be applied to pointers that point to elements of the same array. For example:

```
...
char s[20], *p1, *p2;
p1 = &s[0];
p2 = &s[19];
while (p1 < p2) {
    ...
    p1++;
}
```

All of the above allows us to scan through an array using a pointer variable rather than subscripts. Instead of

```
for (i = 0;   i < 100;   i++)
    cout << a[i] << ' ';
```

we can write:

```
p = a;
for (i = 0;   i < 100;   i++) {
     cout << *p << ' ';
     p++;
}
```

Or, even more economically, utilizing all the shortcuts that C++ provides:

```
for (p = a, i = 0;   i < 100;   i++)
    cout << *p++ << ' ';
```

This idiom is widely used, especially with null-terminated strings.

This relationship between arrays and pointers is reciprocal. Any pointer may be construed as pointing to the first element of *some* logical array, albeit undeclared. If p is a pointer, C++ allows us to write p[0] instead of *p, and in general, p[i] instead of *(p+i).

Consider the following example of a function that shifts the elements of an array to the left by 3 (starting at a[3]):

```
void ShiftLeftBy3(double a[], int size)
// Shifts elements:
//   a[0] = a[3];
//   a[1] = a[4];
//    ...
//   a[size-4] = a[size-1];
{
    double *p = a + 3;
    for (int i = 0;   i < size-3;   i++)
        a[i] = p[i];
}
```

In view of the reciprocity between pointers and arrays, we have to conclude that the most appropriate way of looking at the expression p[i] is to think of [ ] as a 'subscript' operator: we are applying the operator [ ] to the operands p (of a particular pointer data type) and i (an integer).

We have to be a little careful, though. When we declare a pointer `p`, this by itself does not declare any array. Before we start using `p[i]`, we have to make sure that `p` points to some element in an array declared elsewhere, and that `p[i]` is within the range of that array.

NOTES

In a nutshell, whether `ap` is declared as an array or as a pointer, the following expressions are equivalent:

Expression	Is the same as
ap	&ap[0]
ap + i	&ap[i]
*ap	ap[0]
*(ap+i)	ap[i]

If a is declared as an array, enough memory is reserved to hold the specified number of elements, and a cannot be used as an lvalue (i.e. you cannot set a equal to a new address). If `p` is declared as a pointer, the declaration by itself *does not* reserve any memory to which p points; `p` can be used as an lvalue, and, in fact, before `p` is used as a pointer, it must be set to some valid address (the address of some variable or constant, or an array element).

Return by reference

So far we have seen functions returning only values, whether functions were called by value or by reference. Functions can also return reference or pointers. The program below explains the concept of function returning pointer.

Program 2.3

```
/* to find greatest number in an array*/
#include <iostream>
using namespace std;
int main(){
int array[]= {8, 45, 5, 131, 2};
int size=5, * max;
int* fung(int *p1, int size);/*function retuns pointer to
int*/
max=fung(&array[0], size); /*max is a pointer*/
cout<<"\n greatest number="<< *max;
}
int * fung(int *p2, int size)
{
int i, j, maxp=0;
for (j=0; j<size; j++)
     {
     if (*(p2+j) > maxp)
    {
    maxp=*(p2+j);
    i=j;
```

```
        }
         }
        return (p2+i);  /*pointer returned*/
    }
```

Result of Program 2.3

```
greatest number=131
```

The called function returns the address of the greatest number in the array. Look at the function declaration. The function returning a pointer is indicated by the following (a star mark between return data type and function name).

```
int * fung(...)
```

The address of array [0] is received by fung() and stored in *p2. Or in other words p2 points to the 0th location of array.

At this point we must note another way of representing the elements of the array.

The address of 0th element is stored in location p2 or address p2 +0. The element with subscript 1 of the array will be at location one above or at p2 + 1. Thus, address of nth element of this array * p2 will be at address p2 + n. The value of the integer stored at nth location can be represented as * (p2 + n) just like the value at address (p2+0) is * (p2 + 0) or * p2. This notation is quite handy.

The if statement compares maxp with * (p2 + j) or p2 [j] or array [j]. You will easily understand the logic as to how we get the maximum or greatest number in * (p2 + j). Therefore at the end of the iterations * (p2 + j) which is stored at location p2 + j contains the maximum value in the array. Therefore we are returning an address or reference to the called function. In this example (p2 + 3) is the address of the greatest number in the array. After return from the function, max gets the value of p2 + j. In the print statement **max* which is the value stored in memory location *max* is printed. We have already defined *max* as a pointer to an integer in the main function. Thus the function *fung* returns addresses of a value or a pointer. Therefore by returning the address, the value is retrieved automatically by the main function.

Pointers and strings

Pointer is flexible because it points to the first memory location storing a constant or variable. It can be of any type and size. If it is a pointer to character, the character will be stored in the same location pointed to by the pointer. On the contrary, if it is a one-dimensional array of double numbers, the pointer will point to the first byte of the first element in the array. From there we can calculate the address of any element. For instance,

```
double * array[5];
```

Assume that the array[0] is stored at location 1000. The first element will occupy eight cells. Therefore, the second element will be stored from 1008th location onwards. Thus, if we know the starting address of any element in an array, we can find out the address of the required element mathematically. Therefore, we can add and subtract integers from the address pointed to by the pointer or in other words from pointer to go to various elements in the array. In fact in the above program, we have been incrementing the pointer by one each time in the loop. Similarly we can also decrement the pointers. We can also compare pointers by using ==, <, > operators.

Let us now write a program for comparing strings using pointers. A program for the same is given below:

Program 2.4

```
//Example E6x3.cpp
//string compare
#include<iostream>
using namespace std;
int compare(char *p, char *q){
while(*p++==*q++){
        if(*p==’\0') return 0;
                }
        return (*p-*q);
        }
int main(){
int ret;
ret=compare(“Joseph”,”Joseph”);
if (ret==0)
cout<<“strings are same”;
else
cout<<“strings are different”;
                 }
```

NOTES

In the above program, we have a function called *compare*. It receives two pointers to characters. This means it can receive a character each or array of characters. When we compare strings, we have to compare character by character. If the strings are same, when we start from the left and advancing to right one step at a time, we will be finding same character in both the strings. We are checking this in the condition of the while loop as given below:

```
while(*p++==*q++){
```

The condition will be true so long as the corresponding characters are the same. In the condition, we first check the equality of the characters by *p = = *q. Then we increment the pointers. Therefore, in the first iteration, the first characters in both the strings will be checked for equality. If they are equal, we will enter the loop. Then, we check the condition whether the character pointed to by p = NULL. If so, we return 0. If we pass the same string of say four characters, the loop will iterate four times. On the fifth occasion, both p and q will contain NULL. Therefore, the function will return 0.

Suppose, we pass different strings, what happens? As soon as the function finds for the first time, that the characters are not equal, it will come out of the while loop. It will then return the difference between the ASCII value of the characters due to the following statement:

```
return (*p-*q);
```

Thus, when the character arrays are equal we will execute the while loop so long as there are characters and finally will return 0. If the strings are not same, the very first time the two characters are not same, then we return the difference between the characters pointed to by p and q.

Note that in the condition part of the while loop, we did not increment the values but the pointers. We are incrementing the pointers of both p and q simultaneously.

In the main function, we call the function *compare* with two strings. If the returned value is 0, we print that the strings are same. Otherwise, we print strings are different. The result of the program is given below:

Result of Program 2.4

```
strings are same
```

Look at the way the pointers have been deployed. But for the pointers, writing the above program would have consumed more lines of code. However, one has to be careful while using pointers, since it can directly access the memory and spoil important programs.

2.6 FUNCTION AND STRINGS

There is no restriction in passing any number of values to a function; there is restriction only in return values of a function. Therefore, arrays can be passed to a function. Actually, the entire array can be passed to a function, irrespective of its size, by suitable declaration, as the following example indicates.

Program 2.5

```
/* to find greatest number in an array*/
#include <iostream>
int main(){
    int array[]= {8, 45, 5, 911, 2};
    int size=5, max;
    int fung(int array[], int size);
    max=fung(array, size);
    std::cout<<"\n greatest number = "<<max;
}
int fung(int a1[], int size){
    int j, maxp=0;
    for (j=0; j<size; j++){
       if (a1[j] > maxp)    {
          maxp=a1[j];
       }
    }
    return maxp;
}
```

Result of Program 2.5

```
greatest number = 911
```

The objective of the above example is to find out the greatest number in an array. In the program, an array called 'array' is initialized with 5 values and a variable called size is assigned a value of 5. Then a function called fung has been declared which passes an array and an integer to the called function. The array size has not been mentioned. The called function returns an integer. The next statement calls `fung` and passes all elements of the array and an integer 5 equal to size. The function gets the actual values and `size at 5`. The maximum value in the array is found in the for loop and stored in maxp. The value of maxp is returned to the main function and thereafter outputted. Thus, the function is called by value again.

We have passed an integer array into a function. We can pass any other type of array also. What is an array of characters? It is nothing but a string. A string is terminated by Null or \0. It is different from the character 0 whose ASCII value is 48. There is an essential difference between C and C++ with regard to size of string. When we store a string in C, the size of the array is declared equal to the number of characters in the array. But in C++, the size of the array must be incremented by one to store the null character. An example to pass a string is given below:

NOTES

Program 2.6

```
/* to reverse a string*/
#include <iostream>
using namespace std;
int main(){
    char w1[]="abcdefghij";
    void rev(char w1[]);
    rev(w1);
}
/*function definition*/
void rev(char w2[]){
    char w3[]=" ";
/*array w3 initialized with 10 blanks*/
    int i=0, j=0;
    for (i= 9, j=0; i>=0, j<=9; i--, j++)
    w3[j]=w2[i];
    cout<<("original string:\t");
    cout<<(w2);
/*original string printed*/
    cout<<("\nReversed string:\t");
    cout<<(w3);
/*reversed string printed*/
}
```

Result of Program 2.6

```
original string:     abcdefghij
Reversed string: jihgfedcba
```

The purpose of the program is to reverse a string. A string is declared as an array of characters and initialized as, `w1[]="abcdefghij"`; NULL character will be inserted at the end of the string by the compiler. The programmer need not worry. Since we have not specified the size of the array, the compiler will count the number of characters in the string and add 1 to accommodate the terminating character NULL. A function called rev is then declared as given below:

```
void rev(char w1[]);
```

Where `void` signifies that the function returns nothing. The function can be called in a simple manner as given below:

```
rev(w1);
```

Such a notation will not work in the case of arrays of other data types. It will work only in case of strings. In the called function, a character array `w3` is initialized with 10 blanks. The for loop reverses the string in `w2` and assigns to `w3`; `w2[0]` is

Check Your Progress

1. Define function and function main().
2. What do you mean by the term 'function declaration'?
3. Describe the concept of function definition.
4. Name the parts of a general function.

NOTES

copied to `w3[9]`, `w2[1]` is copied to `w3[8]` and so on. Then w2 and w3 are printed one after the other.

The above is the traditional style of programming for strings. We can do the same without declaring a string as a character array, but as a string. Converting the program is left as an exercise for the reader.

We have been calling functions and passing actual values to it. This method is called call by value. When we call functions, we pass actual arguments as per list provided in the declaration. In call by value, a function can return only one value. This puts restrictions on the usage of functions. This can be overcome by using call by reference, where any number of values can be returned indirectly. Call by reference needs understanding of the concept of pointers which will be discussed in the next unit.

2.6.1 `inline` Functions

A function can be defined as an inline function by prefixing the keyword inline to the function header as given below:

```
inline int add(int c, int d){
}
```

When we don't specify inline, i.e. is in the normal circumstances there will be only one copy of the object code for the function, irrespective of the number of function calls as depicted in Figure 2.1 below:

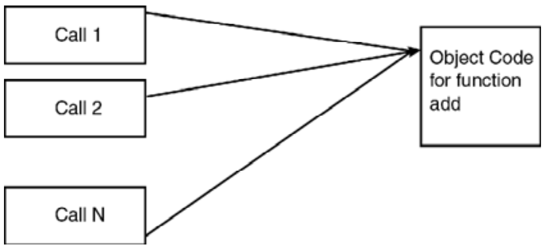


Figure 2.1 *Call to function*

In such cases, whenever a function is called, the program control is transferred to the beginning of the function code. After execution of the function, the program control returns to the calling function. This results in saving in terms of code size, but there are overheads connected with calling and return. The overheads turn out to be small in case of large functions. But if the size of the function is small, then it is advisable to substitute the code at the places where the function is called as shown in Figure 2.2 below:

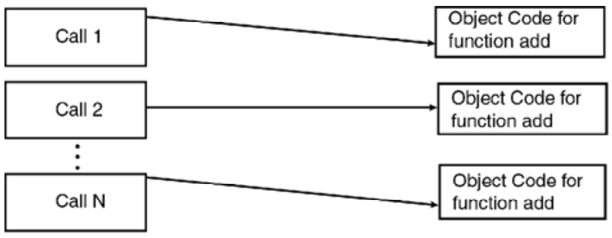

Call 1
Call 2
Call N
Object Code for function add
Object Code for function add
Object Code for function add

Figure 2.2 *Call to inline functions*

This kind of substitution of function code at every place of call can be achieved through prefixing inline to the function header. An example of use of an inline function is given below:

Program 2.7

```
//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 2.7

```
Programming in C++
ya it works
```

The function 'conc' in the above example concatenates two strings.

2.6.2 Dynamic Allocation of Memory in Free Store or Heap

An object refers to fundamental data types, user defined types such as structure, union, class objects and so on representing data. Objects declared with out using pointer notation in a straightforward manner such as int a, account b etc. are called static objects. They exist till the program execution remains within the scope of the object. The object is lost as soon as the program execution goes out of its scope. For instance, we know that the scope of a variable is local to the block in which it is declared. Therefore, it cannot be called back afterwards.

C++ provides two operators as given below for dynamic allocation of memory.

- *new* for allocation of memory
- *delete* for deleting the allocated memory space

The *new* and *delete* can be used only in conjunction with pointer types. With operator *new,* objects can be created just when needed and memory allocated in the heap or what is known as free store. When the program execution reaches the statement, they will be created. The objects created with *new* can be deleted only with *delete* These operators are also called free store operators to indicate that they allocate and deallocate memory on the free store just when needed. These two operators give more flexibility to the programmer to allocate at any time and delete at will. The objects will not be lost when they go out of scope. They have to be specifically deleted, pending which the garbage collector, if any will deallocate the space after making sure that it is no longer required. We can allocate memory dynamically as given below:

```
float * float_var ;
float_var = new float ;
```

The above two statements together allocate space dynamically for the variable *float_var* of type *float*. Such a concept can also be extended to objects and other data types.

NOTES

Check Your Progress

5. What is the nature of a function called by value?
6. Can an array be passed to a function?
7. Describe the `inline` function.
8. Define a pointer.

We can also combine the two statements into one as given below :

```
float * float_var = new float ;
```

This is about declaration. Once a float variable is declared as above, we can assign initial values as given below:

```
*float var = 20.0f ;
```

Even this can be combined with the previous statements as given below:

```
float * float_var  = new float (20.0f) ;
```

Thus the memory space for float variable has been created using the operator *new* and initial values assigned. To delete the above variable we can declare as follows:

```
delete float_var ;
```

We can also declare arrays using the new operator as given below:

```
float * array ;
array = new float [3] ;
```

To delete an array we declare as follows:

```
delete [] array.
```

We will look at usage of *new* for objects later.

Let us look at a program using the new and delete operator which is given below:

Program 2.8

```
 demonstrate new and delete*/
 #include<iostream>
 using namespace std;
 int main()  {
 int * int_var= new int(1234);
 cout<<*int_var<<"\n";
 double *var2=new double(667.9);
 cout<<++(*var2)<<"\n";
 delete int_var;
 delete var2;
 }
```

Result of Program 2.8

```
1234
668.9
```

The Example demonstrates the use of *new* and *delete*. The advantage of these operators is that memory will be kept allocated for just the period required.

Let us take one more example of dynamic allocation of memory. We want to create an array of dimension 4. Then we want to write them on to the console and also find out the addresses at which the elements are actually stored on the heap or free store. The program below does exactly the same.

Program 2.9

```
//demonstartion of new with array
#include<iostream>
using namespace std;
int main(){
   int *array=new int[4];
   cout<<"Type 4 integers \n";
   for(int i=0; i<4; i++)
      { cin>> *(array+i);
      }
   cout<<"The array you typed was\n";
   for(int i=0; i<4; i++)
      { cout<< *(array+i)<<"\n";
      }
   cout<<"The array elements were stored at the address \n";
   for(int i=0; i<4; i++)
      { cout<< (array+i)<<"\n";
      }
      delete[]array;
                    }
```

NOTES

In the above program, we have created an array of size 4 using the new operator. Then we get the elements from the standard input. We have used the pointer notation for declaration of the array as given below:

```
int *array=new int[4];
```

We could have created array with any size. Receiving the integers has been carried out a bit differently as given below:

```
 cin>> *(array+i);
```

First time the integer will be stored as *(array+0) which is nothing but array[0]. In the next iteration, the index i will be equal to 1. Therefore, the value will be stored as array[1] and so on till we store the fourth integer typed at the console at array[3]. After we receive the values from the console, we print them on to the console using the *for* loop. Later on, we find out the address where these elements are stored. Look at the notation for finding out the address in the above program. Removal of the * before expression gets us the address.

Result of Program 2.9

```
Type 4 integers
11 22 44 77
The array you typed was
11
22
44
77
The array elements were stored at the address
0xc3cb4
0xc3cb8
0xc3cbc
0xc3cc0
```

Note that the array elements are stored contiguously as can be inferred by the starting address of each element in the array. Since a float occupies 4 bytes the starting address increases by 4 for each successive element.

2.7 FUNCTIONS AND STRUCTURES

2.7.1 Structure Declaration

Structure is synonymous with records in database. It contains fields or variables. The variables can be of any of the valid data types. The definition of the record, book, which we call structure is given below:

```
struct book {
    string title;
    string author;
    string publisher;
    float price ;
    unsigned year;
};
```

The struct is a keyword of C++ language. A structure tag or name follows which is book in this case. Opening brace indicates the beginning of the structure. Thereafter, fields of the record or data elements are given one by one. The variables or fields declared are also called members of the structure. It consists of different types of data elements, which is in contrast to the array. Let us now look at the members of struct book.

The title of the book is declared as a string; similarly the author and publisher are strings. Then the price is declared as float to take care of the fractional part of the currency. The year is defined as an unsigned integer.

Note carefully the appearance of semicolon after the closing brace. This is unlike other blocks. The semicolon indicates the end of the declaration of the structure. Thus you have to understand the following when you want to declare a structure.

a. struct is the header of a structure definition.
b. It is followed by a name for the structure.
c. Then the members of the structure are declared one by one within a block.
d. The block starts with an opening brace, but ends with a closing brace followed by a semicolon.
e. The members can be of any data type.
f. The members have a relation with the structure; i.e. all of them belong to the defined structure and have identity only as members of the structure and not otherwise. Because if you assign a name to author, it will not be accepted. You can only assign values to book.author.

The structure declaration as above is similar to the function declaration. The system does not allocate memory as soon as it finds structure declaration. It is only for information and checking consistency later on. The allocation of memory takes place only when structure variables are declared. What is a structure variable? It is similar to other variables. For instance int var; means that var is an integer variable. Similarly the following is a structure variable declaration.

```
struct book s1;
```

Here s1 is a variable of type structure. Suppose we define,

```
struct book s1, s2 ;
```

It means that there are two variables `s1` and `s2` of type struct book. These variables can hold different values for its members. In C++ you can omit the keyword struct while declaring structure variables. We can declare:

```
book s1, s2;
```

NOTES

If we want to define a large number of books, then how will we modify the structure variable declaration? It will be as follows:

```
book array[1000];
```

The above will allocate space for storing 1000 structures or records of books. But how much storage space would be needed for each element of the array? It will be the sum of storage space required for each member.

Structure elements

Let us define another structure for bank account as given below:

```
struct account{
    unsigned number;
    char name[15];
    int balance;
};
account a1;
```

We can assign initial values for the structure members individually. For instance, to assign the account number for variable a1 we have to declare as follows:

```
a1. number = 0001;
```

There is a dot operator between the structure variable name and the member name. We can also assign initial values directly as given below:

```
account a1 = { 0001, "Vasu", 1000};
account a2 = { 0002, "Ram", 1500 };
```

This is similar to declaration of initial values for arrays. But, note the semicolon after the closing brace. Therefore a1 will receive the values for the members in the order in which they appear. Therefore we must give the values in the right order.

Let us write a program to create a structure account, open 2 accounts and print the initial deposit in the accounts. Deposit Rs.1000 to Vasu's account and withdraw 500 from Ram's account and print the balance. The following example demonstrates the above.

Program 2.10

```
/*To demonstrate structures*/
#include<iostream>
using namespace std;
int main(){
    struct account {
       unsigned number;
       string name;
       int balance;
    };
    struct account a1= {001, "VASU", 1000};
```

NOTES

```
    struct account a2= {002, "RAM", 2000};
    a1.balance+=1000;
    a2.balance-=500;
    cout<<("A/c No Name Balance\n");
    cout<<a1.number<<"\t  "<<  a1.name<<"\t  "<<
  a1.balance<<"\n";
    cout<<a2.number<<"\t "<< a2.name<<"\t "<< a2.balance;
  }
```

Result of Program 2.10

```
  A/c No    Name        Balance
  1         VASU        2000
  2         RAM         1500
```

A simple program was written for a bank transaction. For a deposit, we write

```
    a1.balance + = 1000 ;
```

Therefore the balance is updated. Similarly, when an amount is withdrawn then the balance is suitably adjusted.

Nested structures

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

Program 2.11

```
   /*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 ;
   }
```

Result of Program 2.11

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

Analysis of the program

NOTES

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 2.12

```
/*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.

Result of Program 2.12

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

2.7.2 Structure with Function

Structure is the forerunner to class. Structure provides a framework for declaration of a user-defined data type. It can be defined with various data types. We can call the data types in the structure as its data members. We can also define member functions in a structure. An example of a structure is shown below. The structure definition is on top of the program. The nametag of the structure is Account. We have two data members namely number and balance. Followed by that is a member function display. This completes the definition of the structure.

Program 2.13

```
//To demonstrate a simple structure and object
#include<iostream>
using namespace std;
struct Account {  // structure tag
int number;      // Data member declaration
double balance;
void display() { //Member function declaration
cout<<"Account number =" <<number;
cout<<"Balance =" <<balance;}
};
int main() {
Account Vinay;                    //creating structure
object
Vinay.number =001;                //Assigning values to
object variables
Vinay.balance =10001.00;
Vinay.display();                        //calling a member
function
}
```

In the main function, we declare a structure variable called Vinay. The structure members are assigned values in the following statements in main(). Thereafter we call function display for the variable Vinay. The result of the program is given below:

Result of Program 2.13

```
Account number =1 Balance        =10001
```

The above program also includes a member function as part of the structure.

2.8 SUMMARY

NOTES

In this unit, you have learnt that functions are the livewire of any program. The function prototype or declaration is meant to give all the characteristics of the function in one sentence. It consists of:

- Return data type
- Function name
- Argument list

You have learnt that the usage of functions requires declaration of the function prototype in the calling function. The declaration ends with a semicolon. A function can be called once it has been declared. While calling functions, we simply provide the identifiers of the data that are to be passed to the function, which is defined usually outside the calling function. The function definition consists of the following parts:

- Function header or declarator, which is the replica of a function declaration but gives the details of arguments received, including their data types
- Function body, which is a group of statements to be executed in the required operation

You now know that functions have to be declared and called with the appropriate arguments in the calling function. You have learnt about passing single values and arrays (strings included) to a function. This is known as 'call by value'. Using this feature, we can pass as many values as required but can return only one. You have learnt that this limitation can be solved if we call functions by reference. Calling a function and returning to the calling function involves many procedures. The small functions can be declared as `inline`. Such `inline` functions will be substituted wherever a call to the function is made.

You have also learnt that a pointer is a variable that contains the address of a data type. In C++, a pointer can be applied not only to built-in types but also to user-defined types, such as objects. Pointers can be used for calling by reference of arrays and other derived data types. This means, the function returns a reference to a variable or a pointer. Arrays and structures have similarities as well as differences between them. Both represent collection of a number of data items. Array is a collection of items of the same data type, whereas structure is not. But structures can represent items of varying data types pertaining to an item. Structure is synonymous with records in database. It contains fields or variables. The variables can be of any of the valid data types.

2.9 KEY TERMS

- **Function:** In programming, functions mean operations. In a program, functions are used with a group of statements to perform specific operations and implement the behaviour of objects in a program.
- **main():** Written as main(), this is an important function as it starts the execution of a C++ program.

- **Function definition:** It is the actual body of the function and can be written anywhere in the file with a declarator or function header (such as in the declaration), followed by declaration of local variables and statements.
- **Inline function:** It is a function which is declared by prefixing it with the keyword inline to the function prototype.
- **Pointer:** It is a variable that contains the address of a data type. Pointers can be used for calling by reference arrays and other derived data types.

2.10 ANSWERS TO 'CHECK YOUR PROGRESS'

1. Functions mean operations. A function is used in a program with a group of statements performing specific operations.

   The function main, written as main(), is an important function as it starts the execution of a C++ program. Functions implement the behaviour of objects in a program. The dynamic properties of objects are facilitated only through functions. Thus, functions provide interfaces for communicating with objects.
2. The usage of a function requires declaration of the function prototype in the calling function. It is usually defined outside the calling function. A declaration ends with a semicolon. A function can be called only after it has been declared.
3. The function definition has the following parts:
   (i) The function header, which is a replica of the function declaration. The only difference is that the declaration in the calling function ends with a semicolon while the declarator in the called function does not.
   (ii) As in the main function, the entire function body is enclosed within braces. The set of statements between the braces forms the function body.
4. A general function consists of three parts: the function declaration (or prototype), function call and function definition.
5. A function called by value can return only one value.
6. Yes, an array can be passed to a function.
7. An inline function can be declared by prefixing keyword inline to the function prototype. Inline functions are suitable for small functions.
8. A pointer is a variable that contains the address of a data type. In C++ programs, pointers can be applied not only to built-in types but also to user-defined types, such as objects. Pointers can be used for calling arrays and other derived data types by reference.

2.11 QUESTIONS AND EXERCISES

Short-Answer Questions

1. Explain the importance of a function in a program.
2. Write a C++ statement to declare a function.
3. Write the C++ statement for calling a function and passing an argument.

4. Write the C++ statement for passing values and returning values.
5. Describe the advantage of arrays and pointers in a program.
6. Explain strings and their applications.
7. Differentiate between a function and a structure.
8. Differentiate between an array and a structure.

Long-Answer Questions

1. State the output of the following programs.

(i).
```
#include<iostream>
using namespace std;
int main()    {
   int a[7],b[7];
   int i;
   void print(int a[7], int b[7]);
   for (i=0; i<=6; i++){
         cout<<"Enter a and b values\n";
         cin>>a[i]>>b[i];
   }
   print (a,b);
}
void print(int a[7], int b[7]){
   int sum[7], j;
   for (j=0; j<=6; j++)  {
         sum[j] = a[j]+b[j];
         cout<< sum[j]<<"\n";
   }
   return;
}
```

(ii).
```
#include <iostream>
using namespace std;
int main()    {
   char s[13]="Selva Murugan";
   int st=7;
   void trunc(char p[13], int q);
   trunc(s, st);
}
void trunc(char a[13], int start){
   int k=0;
   for(k=start-1; k<=12; k++)
   cout<<a[k];
}
```

(iii)
```
#include<iostream>
using namespace std;
```

```
int main()    {
   struct autos {
         char brand [5];
         float price;
   };
   autos auto1 = { "xyz", 50000.50 };
   autos auto2 = { "abc", 50000.00 };
   if (auto2.price > auto1.price)
         cout<<"xyz is cheaper\n";
   else
         cout<<("abc is cheaper\n");
}
```

(iv).
```
#include<iostream>
using namespace std;
int main()    {
   int a, b, sum, dif;
   a=4; b=5;
   int fad(int *a, int *b);
   int fsub(int *a, int *b);
   sum=fad(&a,&b);
   dif=fsub(&a, &b);
   cout<<"sum="<<sum<<"difference="<<dif;
}
int fad(int *a, int *b) {
   int s;
   s=*a + *b;
   return s;
}
int fsub(int *a, int *b){
   int d;
   d= *a-*b;
   return d;
}
```

2. Write C++ programs for the following:
   (i) To print the sum of floats passed as command line arguments.
   (ii) To pass a string along with the position number to a function and return the string deleted up to the position.
   (iii) To pass a string along with position number up to which the string has to be printed by use of a function.
3. Find out the syntax error, if any, in the following statements:
   (i) `inline mul(int p, int q){`
   (ii) `char(char a, char b){`
   (iii) `float func (int, int, float)`

(iv) `static int var = 'p';`

(v) `int main (int argc, int argv){`

(vi) `#define rama sita;`

(vii) `return int var1;`

(viii) `inline var(int, int){`

NOTES

4. Write programs for implementation of the following:

   (i) To pass a string along with the position number to a function and return the string deleted up to the position using pointers.

   (ii) To pass a string along with position number up to which the string has to be printed by use of a function using pointers.

   (iii) To credit and debit a bank account created with structure.

   (iv) To add and issue items in a store.

   (v) To compute Fibonacci numbers less than 1000, store them in an array and print them.

5. Find out the errors, if any, in the following statements:

   (i) `void div(int *, int *){`

   (ii) `div(&*var1, &var2);`

   (iii) `void div(* int *px, int *py){`

   (iv) `char * fung(arg list);`

   (v) `return (*p-q);`

   (vi) `float add (const * float var2);`

   (vii) `enum week_day (Sunday, Monday, Tuesday, Wednesday);`

```
struct account {
      unsigned number;
      string name;
      int balance;
}
```

UNIT 3 CLASSES AND OBJECTS

NOTES

Structure

3.0 INTRODUCTION

In this unit, you will learn about classes and objects in C++. Classes and structures provide a convenient mechanism to the programmer to construct his own data types for convenience in representing real entities. These user-defined data types are required since built-in types cannot be used to represent real entities so easily. They are useful for representing various real entities like bank account information, student records, payroll data, etc. A class definition is similar to a structure. It will have declaration of data elements as well as functions. You will learn that an object is an instance of a class or its replica. The purpose of a constructor is to initialize objects. Built-in data types such as int, float, etc. are initialized as and when they are created. Overloading implies that there is more than one constructor in a class with the same name. A constructor builds an object whereas a destructor destroys an object after it is no longer in use. The destructor, like the constructor, is a member function with the same name as the class name. But it is preceded by the character Tilde (~). There is another useful methodology in C++ called operator overloading. The language allows not only functions to be overloaded, but also most of the operators, such as +, - , *, /, etc. You will learn how conventional operators can be programmed to carry out more complex operations. This overloading concept is fundamentally the same, i.e., the same operators can be made to perform different operations depending on the context. For addition of two integers or two doubles, we use the same '+' operator. You will learn how to extend this overloading concept for addition of objects. Such operators have to be specifically defined and appropriate functions programmed.

3.1 UNIT OBJECTIVES

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

- Define classes and objects
- Understand class constructors and destructors
- Explain the importance of operator overloading in a C++ program
- Write a C++ program using classes and objects
- Write a C++ program using the operator overloading function

3.2 DEFINING CLASS AND OBJECT

3.2.1 Class

NOTES

Class in C++ is similarly a framework for user defined data type. Class and structure provide convenient mechanism to the programmer to build their own data type. These types are convenient to represent real entities. They are needed since built-in types cannot be used to represent real entities so easily. They are quite handy to represent various real entities like bank account, student record, payroll etc.

A structure can also be built with data elements and functions as in the above example. But, structures are seldom built with functions. A class definition is similar to structure. A class will have declaration of data elements as well as functions. A class has a name tag. It contains variables and functions as illustrated in Figure 3.1.

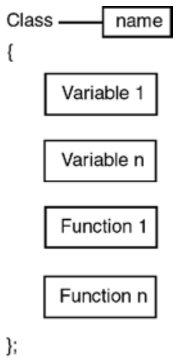


Figure 3.1 *Structure of class*

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 additionally function declaration(s). The class is a keyword just like struct.

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 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. In this case, the function name is display. It returns void or nothing. It does not receive any parameter. The function header is similar to "C" function headers. It follows the same as the function proto-type of "C" language. Here we have given the complete function as part of the class. However alternate ways of function definition exist and will be discussed later.

Member functions and data members

Functions declared in a class are called member functions. The member functions provide the interface to access the variables in the class. The variables are called data members. There is no restriction as to the maximum or minimum number of member functions or data members. They can be specified in any order although it is a common practice to list the data before functions.

Object

Object is an instance of a class or in other words object is a replica of the class. When analyzed 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. But all accounts use the same template. Thus, any number of replicas (objects) can exist for each class. This is the relationship between object and a class. The relationship is illustrated pictorially below.

While class holds generic data structure, objects hold specific and unique data as illustrated in Figure3.2. Thus one class can give rise to many objects.

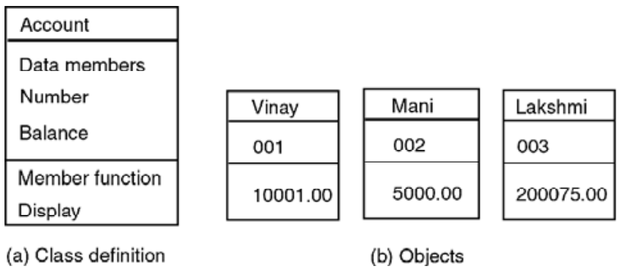


Figure 3.2 *Objects or Instances of class account*

Objects have names and 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. 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.

Access control

The access to data members and member functions of a class or structure can be controlled by using the following keywords:

- private
- public

The members, either the data members or the member functions, if declared private, can be accessed only from within the class. In the examples of class and structure given earlier, we have not specified any access control keyword. The default access is public for structure and private for class. Since default access is private in the class, the access control will be assumed to be private in the class i.e. the data can be accessed only from within the class. The objects outside the class can access its private data members only through the member functions of the class that are declared public. The member functions cannot be accessed from outside if they are also declared private. Therefore, the member functions have to be declared public for meaningful programs.

If the member functions or data members are declared public then they can be accessed from outside the class. The data members should not be declared public since it will defeat the very purpose of data hiding, which is one of the essential requirements of OOP. However, the member functions are usually declared public. Declaring both the functions and data as private will totally shield the class from outside world and therefore it does not serve any useful purpose.

Using the concepts learnt so far, let us write a program to create a class for Account and then display the account number and balance. Look at the program given below:

Program 3.1

```
//To demonstrate a simple class and object
#include <iostream>
using namespace std;
class Account {       // class tag
public:
int number;           // Data member declaration
double balance;
public:
void display() {            //Member function declaration
    cout<<"Account number =" <<number;
    cout<<"Balance        =" <<balance;
    }
};
int main() {
    Account Vinay; //creating object
    Vinay.number =001;//Assigning values to object variables
    Vinay.balance =10001.00;
    Vinay.display(); //calling a member function
}
```

Let us read line by line and try to understand the program. The first line is a comment statement with no implications with regard to execution. The second line contains the name of the file to be included, namely `<iostream>`.

We have a class declaration at line 4, followed by opening brace as given below:

```
class Account {
```

The opening brace indicates the beginning of the block for class Account. The fifth line contains access control statement public: for the data members. Although the variables are generally not declared public, it is done here to illustrate a concept. We have two variables declared as given below at lines 6 and 7.

Line 6: `int number;`

Line 7: `double balance;`

We want to declare the account number as an integer and the balance as a double precision floating point number. Then at line 8, we have access control statement for the member function, public: It is followed by declaration for a function display

```
Line 9: void display() {
```

The opening brace indicates beginning of the block for the function display. The "display" is the name of the function. The function does not return any data and hence return data type is declared as void. The function does not receive any values as indicated by empty parentheses after the name of the function.

The statement in the function at line 10 is for printing 'Account number =' followed by the value contained in the variable number. Line 11 is for printing 'Balance =' followed by the value contained in the variable balance. Note that number and balance are the variables. This is followed by two closing braces, one corresponding to function display() and the other corresponding to class Account. A semicolon follows the second closing brace. This completes the definition of class Account that contains two variables and one function. One more line is left blank to improve readability.

At line 15, we find the declaration of the familiar main function. We have been using the function main() in the examples seen so far. Note that all programs will have main() function. The main() indicates where the program execution has to start.

Line 16 is new to us. We declare

```
Account Vinay;
```

Here, we are creating an object named Vinay of type Account class. This is similar to declaration of structure variable. Note that we have created an object Vinay of class Account. Therefore, Vinay will have two variables of number and balance of basic data type int and double respectively. Now, we assign values to the variables in the object Vinay in the next two lines.

We assign 001 to number in Vinay by stating

Vinay. number = 001 ;

Again we assign 10001.00 to balance in Vinay object as follows:

Vinay. balance = 10001.00;

Notice that we are making the assignment of values directly to the variables of the object outside the class Account. This has become possible since the data members were declared public. Had they been declared private we cannot directly assign the values. In that case we have to access the data members only through a public function in the class.

Now we call display pertaining to Vinay in the next statement as given below:

```
Vinay.display ( ) ;
```

The next line contains closing braces corresponding to the main function.

Note: In C++, we type the opening brace in the next blank line after the class declaration and function declaration. In the program above and in the rest of the book, the author has given the brace just after the class name as well as function name. It does not matter how it is written. What is needed is the declaration of the class or function, followed by an opening brace. The style is entirely left to the choice of the programmer.

The result of the program is given below:

Result of Program 3.1

```
Account number =1Balance =10001
```

NOTES

We have successfully executed an object oriented program. To understand the program completely, we must understand the concepts clearly.

The objects, the special data types are to be created as given in the program. The simple data types were created in a simple manner as given below:

```
    int number;
```

Note that in creating objects, we had followed a set pattern as given below:

We first declared the user defined data type namely Account which is a class. Then, we gave the name of its instance as Vinay. When we create an object, the system automatically allocates the required space for the object. In fact, both the fundamental data variables such as number and objects, which are variables of type class, are created in similar manner. Although, the effect of the statements will be widely different, the declarations are similar. In the case of a simple variable, it is just assignment of memory for one variable. On the other hand, in the case of an object, the assignment of memory should be for all the data members. Furthermore, there should be a way of linking the data members to the corresponding functions.

Assigning values to objects

Now that we have created the object Vinay, we have a replica of the class Account available. Now, we have to assign values to the variables of the object. This is achieved by using dot operator. We declared as given below:

Vinay. number = 001 ;

We can access the variable number using the dot operator. This is similar to structure members in C language. We have associated the object name along with the names of the variables. Therefore if we simply state, balance = 10001.00, it will not be recognized. We have to be specific, as to the balance of which object we are assigning the value of 10001.00.

Now, let us look at the main function of the program above to understand it better.

We created Vinay as an object of type Account. Then we assigned values to the variables by using dot operator. Thereafter we called the member function display for the object Vinay as given below:

```
    Vinay. display() ;
```

Though we do not pass any parameter, the program works fine. Since the object Vinay has called the function, the function display prints the values of number and balance pertaining to Vinay. Had some other object called the member function display, then the corresponding values of the object would have been printed.

A modification of the above program is given below:

Program 3.2

```
//To demonstrate class without function
#include<iostream>
using namespace std;
class Account {
public:
    int number;
    double balance;
};
int main() {
    Account Vinay;
    Vinay.number =001;
    Vinay.balance =10001.00;
    cout<<"Account number ="<<(Vinay.number);
    cout<<"Balance        ="<<(Vinay.balance);
}
```

Here, the class Account contains no function. We create object Vinay of type class Account in the `main()`. Similarly we assign values to its variables. Then we have two print statements (implemented using cout object) which means the printing is done by the main() function itself. We do not have to call any function to print. But we have to qualify the number as Vinay.number and balance as Vinay.balance. If we omit the qualifier then they will not be recognized. In their simple form without reference to any specific object, number and balance will be recognized only in class Account. We get the same result on execution of the program, as given below:

Result of Program 3.2

```
Account number =1Balance        =10001
```

Now, let us modify the program to create and display three account details. This is to reinforce the concept and prove that the above methodology works fine. But the data members have to be declared public. The program is given below:

Program 3.3

```
//To demonstrate creation of three objects
#include<iostream>
using namespace std;
class Account {
public:
int number;
double balance;
public:
void display() {
      cout<<"Account number ="<<number<<"\t";
      cout<<"Balance        ="<<balance<<"\n";
    }
};
int main() {
    Account Vinay;                //creation of object Vinay
    Vinay.number =001;            //Assigning  value  to
number
    Vinay.balance =10001.00; //Assigning value to balance
    Vinay.display(); //printing account number and balance
    Account Mani; //creation of object Mani
```

```
      Mani.number =002; //Assigning value to number
      Mani.balance =2467.45; //Assigning value to balance
      Mani.display(); //printing number and balance
      Account Lakshmi; //creation of object Lakshmi
      Lakshmi.number =003; //Assigning value to number
      Lakshmi.balance =200075.00;//Assigning value to balance
      Lakshmi.display(); //printing number and balance
  }
```

In main function, we first create object `Vinay` and assign values to the variables. Then we call `Vinay.display()` to print the account number and balance of object `Vinay`. Then we create objects Mani and Lakshmi and assign values and then print them. After Compiling and executing the program we get the results as given below:

Result of Program 3.3

```
  Account number =1        Balance        =10001
  Account number =2        Balance        =2467.45
  Account number =3        Balance        =200075
```

We have got the values of all the three objects.

Difference between structure and class

We can say that struct is also a class. The difference is that the members of structure are public by default whereas the members of class are private by default. Once we specify public: or private:, each member following such declaration will have an access specifier, Viz. public and private respectively. In fact all the above programs can be implemented by just changing class with struct. If the data members are public we can use struct. To conclude struct is a simple form of class with the default access specifier being public.

Private data members

We have been creating classes and objects. We created object, an instance of a class. We declared them similar to other variables. However, the objects belong to the user defined data type namely, class (here Account). The objects must have their own data in accordance with the definition of the corresponding class. Since the data members were declared as public, we could use them outside the class. To give values to the variables of the objects, we assigned:

```
      Vinay. number = 001;
      Vinay. balance = 10001.00;
```

Vinay specifies the exact object whose values are being assigned. This was possible since we had declared the data members as public. As reiterated, data members should not be declared as public, as a rule in order to implement data hiding, which is the unique feature of OOP. When, data members are declared private, we can assign values by using a member function in the class. Furthermore, when the data members are declared as private, this is the only way to assign values to the object variables. The Program below illustrates the concept.

Program 3.4

```
  /*To demonstrate passing data to an object through member
  function-private data*/
```

NOTES

```
#include<iostream>
using namespace std;
class Account {
private:
int number;
double balance;
public:
void indata(int x, double y) {    /* function signature-
formal parameters */
       number = x;
       balance =y;
   }
void display() {
       cout<<"Account number ="<<number;
       cout<<"Balance        ="<<balance;
     }
};
int main() {
    Account Vinay;
    Vinay.indata(001, 10001.00);//calling function with
arguments
    Vinay.display();  //calling function with no arguments
}
```

In class Account, we have a function named indata() with void as the return data type. This means that the function does not return any value. It receives two parameters namely int x and double y. The received values are assigned to the variables namely number and balance respectively. Since the member function is part of the class, the number and balance in the function indata are recognized without any qualifiers. This function is now common to all objects, in the sense that the objects can call it to assign values to their respective variables. We are already familiar with the display function in the class.

Now look at the main function. The object Vinay calls function in data with specific values in bracket.

The values received will be assumed to be in the correct order and will be assigned to the variables of the object as above.

Now the main function calls Vinay.display(). The function knows the values of Vinay. number and Vinay. balance and hence it will print them one after the other. The result of the program is given below:

Result of Program 3.4

```
Account number =1Balance =10001
```

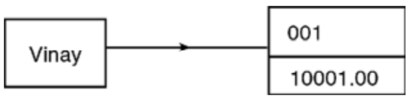
To summarize, in order to provide data hiding, which is one of basic purposes of OOP, the data members should be declared private. A private data member can be assessed through a public member function in the class. The above program illustrates the practical methodology for assigning values to object variables.

`inline` Functions

In the above example, the function as a whole is defined as part of the class. We can declare the function prototype in the class and the function body outside the class. When the function prototype and body are defined within the same class, such functions are treated as inline functions although the inline prefix is missing in the function prototype.

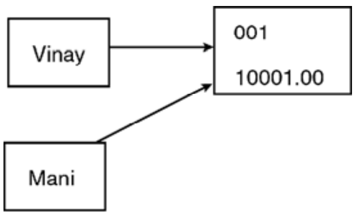
Object copy

The name of the object is the reference to the object. The name can be viewed similar to structure variables as depicted pictorially below:



Basically, Vinay points to the values stored in the object.

Now we write a program to declare object reference variable Vinay, assign values and declare another object reference variable Mani which points to the same object as (illustrated below)



There is only one object, but pointed by 2 variables. When we print values of the object data we will get identical values for both the object references.

The program is given below:

Program 3.5

```
//To demonstrate Object Reference
#include<iostream>
using namespace std;
class Account {
int number;
double balance;
public:
void indata (int x, double y) {
       number=x;
       balance=y;
    }
void display() {
       cout<<"Account number ="<<number<<"\t";
       cout<<"Balance        ="<<balance<<"\n";
    }
};
int main() {
    Account Vinay;
```

```
        Vinay.indata(001, 10001.00);
        Account Mani = Vinay;         // Mani references Vinay
        Vinay.display();
        Mani.display();
    }
```

The new assigned statement is:

```
        Account Mani = Vinay;
```

Here, the object Vinay is assigned to object Mani.. We may find it similar to structure copy. When we execute the program, the same values repeat for Mani. display.

Result of Program 3.5

```
Account number =1Balance        =10001
Account number =1Balance        =10001
```

At this stage, we are not sure whether a copy of the object is made when we assign an object reference to an existing object. The following program will clear our doubt as to whether a copy is made of the object or simply the second object makes a reference to the first object.

Program 3.6

```
//To find out whether a copy of the object is made or not
#include<iostream>
using namespace std;
class Account {
private:
int number;
double balance;
public:
void indata (int x, double y) {
       number=x;
       balance=y;
    }
void display() {
       cout<<"Account number ="<<number<<"\t";
       cout<<"Balance  ="<<balance<<"\n";
    }
};
int main() {
    Account Vinay;
    Vinay.indata(001, 10001.00);
    Account Mani=Vinay;     // Mani references Vinay
    cout<<"original values of objects- Vinay and Mani"<<"\n";
    Vinay.display();        //printing  account  number  and
balance (Vinay)
    Mani.display();  //printing account number and balance
(Mani)
    Vinay.indata(005, 768.75);
    cout<<"values of objects after changing Vinay object
"<<"\n";
    Vinay.display();        //printing account number and
balance (Vinay)
    Mani.display();                    /*printing account
number and balance (Mani)*/
    Mani.indata(10, 1234.56);
    cout<<("values of objects after changing Mani object
")<<"\n";
    Vinay.display();                   //printing account
number and balance (Vinay)
    Mani.display();                    //printing account
number and balance (Mani)
}
```

The following steps are performed in the above example :

Step1: Object Vinay is assigned initial values

Step2: Mani also refers to Vinay

Step3: Displays the contents of both objects

Step4: Change values in Vinay

Step5: Display contents after above

Step6: Change values in Mani

Step7: Display contents of both objects

The result of executing the program is given below:

Result of Program 3.6

```
original values of objects- Vinay and Mani
Account number =1       Balance        =10001
Account number =1       Balance        =10001
values of objects after changing Vinay object
Account number =5       Balance        =768.75
Account number =1       Balance        =10001
values of objects after changing Mani object
Account number =5       Balance        =768.75
Account number =10      Balance        =1234.56
```

We will find that the first time, the contents of both the object variables remain the same since we have assigned the reference of Vinay to Mani. With this result alone, we cannot make out whether a copy of the object was made. But the subsequent results confirm that as soon as an object reference had been made, an identical copy of the object has been created by the system. Otherwise, the contents of both the objects would have always remained the same. When the data of object Vinay was changed in the above program it does not automatically change the data of object Mani. Similarly the data of object Mani could be changed independently. All these confirm that the object reference is used to copy the contents of one object to other, similar to ordinary variables.

3.3 CLASS CONSTRUCTORS AND DESTRUCTORS

NOTES

Constructors

As you know the purpose of constructor is to initialize the objects. We know that built in data types such as int, float etc. are initialized as and when created. For instance,

```
float var;
```

Although no initial values are assigned explicitly, var will get an initial value, depending on the type of variable like static, automatic, register or global. It can be zero or a garbage value. We can also assign specific initial value. For instance,

```
float var = 10.0f;
```

In this case, value of 10.0 is assigned to var, on its creation. Another interesting aspect is that when the program execution goes outside the block, the variable will be destroyed or not available in all cases except the static types. Such a concept is not automatic in the case of objects. In the initial examples, we were using specific functions to assign values to the data members. We had not even thought of destruction of the objects. C++ has special arrangements for assigning initial values to an object and destroying the objects when they go out of scope. This is achieved through constructors and destructors respectively. When a class contains a constructor, then definitely the class will be initialized, like initializing a built in data type on creation.

We are familiar with the constructor construction. The constructor has the same name as that of the class with no prefixing of return data type. We are also familiar with parameterized constructor. Essentially, it receives the values for the data members. In short, constructor is a function with minor modifications. Its purpose is to initialize the objects on creation. Now we will look at overloading of constructors.

The principle of constructor overloading is the same as that of function overloading, which is to be discussed in the next chapter. Overloading implies that there is more than one constructor in a class with the same name. As you know, the name of the constructor has to be same as the class name. Therefore more than one constructor means many constructors with the same name in a class. But, the difference will be in the argument list. If there is no difference in the argument list, multiple, identical constructors do not serve any purpose. Let us now look at an example of multiple constructors in a class. It is given in example below:

Program 3.7

```
//To demonstrate overloading of constructors
#include<iostream>
using namespace std;
class Account {
    int number;
    double balance;
    public:
    Account(){
          number=0;
          balance=0;
       }
```

NOTES

```
        Account(int x){
        number=x;
        balance=0;
    }
    Account (int x, double y) {
        number=x;
        balance=y;
    }
    void display() {
        cout<<"\n Account number ="<<number;
        cout<<"Balance ="<<balance;
        }
    };
    int main() {
        Account Vinay; //no arguments
        Vinay.display();
        Account Mani(002); //one argument
        Mani.display();
        Account Lakshmi(003, 200075.00); //2 arguments
        Lakshmi.display();
    }
```

In the class Account there are three constructors as given below:

```
Account(){
    number=0;
    balance=0;
}
Account(int x){
    number=x;
    balance=0;
}
Account (int x, double y) {
    number=x;
    balance=y;
}
```

The first constructor doesnot take any arguments, the second one takes one argument and the last takes two arguments. All the three are valid declarations. It is interesting to see how the initial values of the data members are assigned in all the three constructors. The last one is straightforward. It accepts one integer and one double argument. It assigns the integer value received to the data member number. It assigns the double value received to the data member balance. In fact, this is the constructor, which we are familiar with, and is known as parameterized constructor.

Look carefully what we have done when there is only one argument. The argument of type integer received, is assigned to the data member number. In the constructor, we are initializing the value of balance to be zero. Thus, with one argument, we are able to initialize both the data members of the object of type Account.

Now let us look at the constructor with no arguments. In this case, we can initialize both the data members to set values. In the above example, we have initialized them to zero.

In main function, we have created an object `Vinay`. Look carefully absence of parentheses after the object name. This statement means that `Vinay` does not pass any values to the constructor. So the constructor will initialize both the data members with predefined values. We call function display to display the values contained in the data members.

Then we create an object Mani by specifying only one argument. This will be assigned to the variable number. After displaying the contents of Mani, we create third object Lakshmi with two arguments. In the function display, we display the values of both the data members. The result below confirms that overloading of constructors works well.

Result of Program 3.7

```
Account number = 0    Balance =   0
Account number = 2    Balance =   0
Account number = 3    Balance =   200075
```

Default arguments in constructors

We can slightly modify the parameterized constructor, to construct constructor with default arguments. For instance, a Bank Account may be opened with an initial deposit of say Rs 100. In this case, we can have a constructor with balance equal to 100. But, some account holders might deposit more money. So we would like to have a provision, that if nothing is specified to one of the data members, it should be initialized to a default value. If a value is specified to the same data member by the user, then that should override the default value. This is the purpose of constructors with default arguments. The example below illustrates the concept with a default value of 100.

Program 3.8

```
//To demonstrate constructors with default arguments
#include<iostream>
using namespace std;
class Account {
    int number;
    double balance;
    public:
       Account (int x, double y=100) {
          number=x;
          balance=y;
       }
    void display() {
       cout<<"\n Account number = "<<number;
       cout<<" Balance = "<<balance;
    }
};
```

```
int main() {
    Account Mani(002); //one argument
    Mani.display();
    Account Lakshmi(003, 200075.00); //2 arguments
    Lakshmi.display();
}
```

In the above program, the constructor has a default value of 100 for variable balance. Look at the main function. We create object Mani by sending only one argument. Therefore, the program will assign a value of 100 to the balance, since the default is 100 and the object is silent about the value of balance. However, the object Lakshmi has two parameters. Therefore, although the second argument has been given a default value of 100, the parameter passed by the object overrides the default. Hence, the balance will be taken as 200075.00

Result of Program 3.8

```
Account number = 2 Balance = 100
Account number = 3 Balance = 200075
```

We must particularly note that there was only one constructor in the class. But the second argument 'y' was assigned default value of 100 on the declaration of the constructor. However, in the body, the variable balance was assigned the value of 'y'.

When the second parameter was absent as in the first case, the value of balance is assigned to be 100. When it is present, the second parameter gets the value passed by the object. This is the concept of constructors with default arguments. An important point to be noted is that in such cases one of the arguments may be missing while the constructor is called. The compiler assumes that the missing argument is the latter one and not the first one. Usually, if two arguments are required and only one is supplied, it can lead to run time errors. But, in the case of constructors with default arguments, it works without any error.

Default constructor

The default constructor for any class is the constructor with no arguments. When no arguments are passed, the constructor will assign the values specifically assigned in the body of the constructor. It can be zero or any other value. The default constructor can be identified by the name of the class followed by empty parentheses.

Copy constructor

The copy constructor copies contents of another constructor or to be precise an object. Here the constructor assigns the values of data members of an existing object to a new object at the time of creation. The following example illustrates the concept of copy constructor.

Program 3.9

```
//To demonstrate copy constructor
#include<iostream>
using namespace std;
class Account {
    int number;
    double balance;
```

```
      public:
         Account (int x, double y=100) {
         number=x;
         balance=y;
      }
      Account (Account & Ac1) {
         number=Ac1.number;
         balance=Ac1.balance;
      }
      void display() {
         cout<<"\n Account number = "<<number;
         cout<<" Balance = "<<balance;
      }
   };
   int main() {
      Account Mani(002, 1000);
      Mani.display();
      Account Lakshmi(Mani); //object passed
      Lakshmi.display();
   }
```

The copy constructor in the program is reproduced below.

```
   Account (Account & Ac1) {
      number=Ac1.number;
      balance=Ac1.balance;
   }
```

Look at the declaration. It receives a reference to an object Ac1 of class Account. In the body of the constructor the respective variables of Ac1 are assigned to the data members of the object. Or in other words, the data members of the passed object are copied to the receiving object. This is an example of copy constructor.

In the main function, the object Mani is created and initialized through the parameterized constructor. Now copying the object Mani through the copy constructor as illustrated below creates object Lakshmi.

```
      Account Lakshmi(Mani);
```

The parameter here is the object itself. Since object Lakshmi is created by copying data members of object Mani, they will hold identical values as the result of the program indicates.

Result of Program 3.9

```
   Account number = 2 Balance  = 1000
   Account number = 2 Balance  = 1000
```

We have been creating objects using constructors. If there is a parameterized constructor, initial values are to be passed along with the object declaration. Using default constructor, we can initialize the variables as we please. Constructor is a special form of member function. But the difference is that while other member functions are called explicitly, constructors are called implicitly and automatically. To confirm this, let us look at the example below.

Program 3.10

```
//To demonstrate that constructors
//are called implicitly
#include<iostream>
using namespace std;
int object_count;
class Account {
    int number;
    double balance;
    public:
    Account(){
       number=0;
       balance=0;
       object_count++;
       cout<<"\n first constructor- object No. ";
       cout<<object_count<<" created";
    }
    Account(int x){
       number=x;
       balance=0;
       object_count++;
       cout<<"\n second constructor- object No.";
       cout<<object_count<<" created";
    }
    Account (int x, double y) {
       number=x;
       balance=y;
       object_count++;
       cout<<"\n third constructor- object No. ";
       cout<<object_count<<" created";
    }
    void display() {
       cout<<"\n Account number ="<<number;
       cout<<"Balance ="<<balance;
    }
};
int main() {
    Account Vinay; //no arguments
    Vinay.display();
    Account Mani(002); //one argument
    Mani.display();
    Account Lakshmi(003, 200075.00); //2 arguments
    Lakshmi.display();
}
```

Look at the program. Print statements are added as part of the constructors. Furthermore a variable object_count is declared as a global variable. Each constructor has an increment statement of the global variable whose value is actually printed in the print statement in the constructor. Now look at the result.

Result of Program 3.10

```
first constructor- object No. 1 created
Account number =0Balance          =0
second constructor- object No.2 created
Account number =2Balance =0
third constructor- object No. 3 created
Account number =3Balance =200075
```

The print statement of the constructor is followed by the display of member variables. In the above program we create objects using the constructors in the same order. Therefore the first object uses first constructor and so on. The above result confirms that the objects are created using constructors and the constructors are called automatically.

Destructors

Constructor builds an object. A destructor destroys an object after it is no longer in use. The destructor, like constructor, is a member function with the same name as the class name. But it will be preceded by the character Tilde (~). For instance destructor of the class Account will be coded as follows:

```
~Account(){
}
```

We can insert code between the braces if we desire.

Unlike constructors, destructors do not receive any arguments, not even void. In the past, we were not declaring destructors. However the compiler will automatically destroy the objects after the program execution is completed. Specifying constructors will ensure that all objects created are destroyed before the program execution stops. Destroying essentially means deallocating the memory space.

Let us look at a program using constructors and destructors.

Program 3.11

```
//To demonstrate overloading of constructors
#include<iostream>
using namespace std;
int object_count;
class Account {
    int number;
    double balance;
    public:
    Account(){
       number=0;
       balance=0;
       object_count++;
       cout<<"\n first constructor- object No. ";
```

```
            cout<<object_count<<" created";
        }
        Account(int x){
            number=x;
            balance=0;
            object_count++;
            cout<<"\n second constructor- object No.";
            cout<<object_count<<" created";
        }
        Account (int x, double y) {
            number=x;
            balance=y;
            object_count++;
            cout<<"\n third constructor- object No. ";
            cout<<object_count<<" created";
        }
        //Destructor
        ~Account(){
            cout<<"\n from destructor - object No. ";
            cout<<object_count--<<" destroyed";
        }
    };
    int main() {
        Account Joseph(001); //one argument
        Account Vinay; //no arguments
        Account Mani(002); //one argument
        Account Lakshmi(003, 200075.00); //2 arguments
    }
```

In the above program we have three constructors and only one destructor. When we use a constructor every time, we increment the variable object_count. Every use of destructor decrements the same variable after printing the value of the object_count. Notice how the output is combined with the destructor. It will be interesting to look at the result of the program, given below:

Result of Program 3.11

```
second constructor- object No.1 created
first constructor- object No. 2 created
second constructor- object No.3 created
third constructor- object No. 4 created
from destructor - object No. 4 destroyed
from destructor - object No. 3 destroyed
from destructor - object No. 2 destroyed
from destructor - object No. 1 destroyed
```

We first create object No.1 (Joseph) with the second constructor. Then the other

3 objects are created. See that the destructor is never called explicitly. But it acts automatically and destroys all the four objects, one at a time. The result confirms that the same destructor is called 4 times, as many times as the number of objects-nothing more or nothing less.

In the above program we never called the destructor. The compiler called them implicitly. We need to call the destructors explicitly when we create objects using new keyword. To delete such objects we use keyword delete. This will be discussed later.

Case Study: Students Mark List

Like other variables, we would require to declare and use array of objects. There is only a simple difference when we want to declare an array of objects. For instance,

```
class grade {}
grade Tenth ;
```

The above declares an object called Tenth of class grade. If we want to declare an array of objects of class grade, we have to simply declare it as an array as given below:

```
grade Tenth [10] ;
```

In the above statement we had declared an array of objects called Tenth with dimension 10 belonging to class grade. It is similar to declaring an array of any built-in type. There is absolutely no other difference. Let us now look at a case study, where we create an array of objects. In this case study, we prepare a mark list consisting of students name and marks obtained. The program is given below:

Program 3.12

```
//Case Study - Students Mark List
#include<iostream>
using namespace std;
class grade{
    private:
       string name;
       int mark;
    public:
       void getdata(){
          cin>>name>>mark;
       }
    void display(){
       cout<<name<<"\t"<<mark<<"\n";
    }
};
int main(){
    grade Tenth[5];
    int i=0;
    cout<<"Enter name and mark 5 times\n";
    do{
       Tenth[i].getdata();
```

NOTES

```
      i++;
   }while(i<5);
   cout<<"Name of Student\t"<<"Marks \n";
   for( i=0; i<=4; i++){
      Tenth[i].display();
   }
}
```

In the above program, we have a class grade with two private data members and two public member functions. We declare an array of objects of the class grade called Tenth [5]. This will allocate space for holding an array of five objects of class grade. Look at the way we get the data corresponding to each object. We call the member function getdata with one element of the array of objects each time. Similarly, we call function display of the class every time with one element of the array of object (–) Tenth [i].

Result of Program 3.12

```
Enter name and mark 5 times
Ramasamy        450
Sriraman        452
Devikala        550
Daniel          569
Rajanna         458
Name of Student Marks
Ramasamy        450
Sriraman        452
Devikala        550
Daniel          569
Rajanna         458
```

The above case study gives a feel for creating array of objects.

3.4 FUNCTION AND OPERATOR OVERLOADING

3.4.1 Function or Method Overloading

Programmers of conventional languages such as C are taught that every function is unique with unique names. Functions are called to carry out specific operations and they should be called strictly with the same number of parameters of the given type and in the same sequence. Suppose we call a function to add two floating point numbers, then we have to call the function with two floating point numbers only as an argument. If by chance, they are called with two integers or two characters, it can lead to error(s). But in OOP we can define multiple functions with the same name but with different types of arguments. We can have a function 'add' with two floating point numbers as arguments, another function 'add' with three integers, and a third function 'add' with four doubles, all in the same program. When we call function 'add' with four doubles, then the function with four doubles will be called and similarly if we call 'add' with three integers then the corresponding function will be called. This concept is also known as Function or Method overloading.

Let us look at an example to illustrate this concept.

Program 3.13

```
//To demonstrate function overloading
// with three objects
#include<iostream>
using namespace std;
class MethOl {P
public:
void add(int a, int b, int c){
cout<<"Sum = " <<(a+b+c)<<"\n";
}
void add(float a, float b){
cout<<"Sum = " <<(a+b)<<"\n";
}
void add(double a, double b, double c, double d){
cout<<"Sum = " <<(a+b+c+d)<<"\n";
}
};

int main() {
MethOl intadd;;
intadd.add(10, 20, 30);
MethOl floatadd;
floatadd.add(10.5f, 20.5f);
MethOl doubadd;
doubadd.add(10.5, 20.5, 25.0, 17.3);
                                                }
```

Result of Program 3.13

```
Sum = 60
Sum = 31
Sum = 73.3
```

NOTES

Look at the program and its result. We have defined three functions with the name *add*, but with different sets of parameters – one with three variables of type *int*, the second with two variables of type *float* and the third with four variables of type *double*. In main function, we create three objects called *intadd*; *floatadd* and *doubadd*. Then we call the respective *add* function with the data of corresponding types. Notice that the return data types are similar in the above example. They could also be different. The difference in return data type is not a consideration for function overloading. Notice also that the above example does not contain any data members.

Let us see whether we can avoid creating three objects, as in the above example. A revised program that creates a single object *xadd* for addition of various types of numbers, is given below:

Program 3.14

```
//To demonstrate function overloading
// with single object
#include<iostream>
using namespace std;
class MethOl {
public:
void add(int a, int b, int c){
cout<<"\n Sum = " <<(a+b+c);
}
void add(float a, float b){
cout<<"\n Sum = " <<(a+b);
}
void add(double a, double b, double c, double d){
cout<<"\n Sum = " <<(a+b+c+d);
}
};

int main() {
MethOl xadd;
xadd.add(10, 20, 30);
xadd.add(10.5f, 20.5f);
xadd.add(10.5, 20.5, 25.0, 17.3);
            }
```

What is different in this example? We have created only one object *xadd* and adding int, float and double of the right types and quantity. There is no other difference between the previous example and this. Now look at the result of this program.

Result of Program 3.14

```
 Sum = 60
 Sum = 31
 Sum = 73.3
```

We got an identical result. Therefore, with the same object and function name, we are able to perform similar operations with different types of data. This is function overloading. One may wonder, are we short of function names? No, that is not the idea. Suppose in the above program we have three names for adding three different data types then we are providing three interfaces. By function overloading we have reduced the number of interfaces to one for carrying out many similar types of operations. That is the advantage.

The important point to be noted about function overloading is that with a single interface we are able to access multiple functions. Hence, it is also known as polymorphism. This is a simple type of polymorphism. The functions with the same names have different signatures. Signature is nothing but the arguments. The difference in return data type is not a consideration for overloading, but the difference has to be in the set of arguments. The difference could be in the types of parameters or number of parameters or both, as the above programs illustrated.

Although overloading is aimed at providing a common interface to carry out similar operations, it is not mandatory. Let us overload functions but carry out dissimilar operations. The following illustrates this concept.

Program 3.15

```
 /*To demonstrate function overloading
for different operations*/
#include<iostream>
using namespace std;
class MethOL {
public:
void add(int a, int b, int c){
cout<<("\nSum = ")<<(a+b+c);
}
void add(double a, double b){
cout<<("\nProduct = ")<<a*b;
}
};

int main() {
MethOL xadd;
xadd.add(10, 20, 30);
xadd.add(10.5, 20.5);
            }
```

Here we have defined function 'add' to add three integers in class MethOL. Then we have overloaded it to multiply two doubles. Our calling it *add* is not logical. However the program works and the result is given below.

Result of Program 3.15

```
Sum = 60
Product = 215.25
```

Thus, function overloading is basically giving same name to more than one function and calling them with distinct parameters. The functions will carry out the functions dictated by the programmer. It will be logical only if they carry out similar operations. Carrying out dissimilar operation through function overloading is not a good programming practice and hence should not be done.

3.4.2 Operator Overloading

There is another useful methodology in C++ called operator **overloading**. The language allows not only functions to be overloaded, but also most of the operators, such as `+, -, *, /,` etc. As the name suggests, here the conventional operators can be programmed to carry out more complex operations. This overloading concept is fundamentally the same i.e. the same operators can be made to perform different operations depending on the context. Even in normal circumstances, operators such as `+, -, *, /` etc. do the respective operations on different data types. For addition of two integers or two doubles, we use the same + operator. To that extent, the operators are already overloaded. We can extend this overloading concept for addition of objects. Such operators have to be specifically defined and appropriate function programmed. We will first look at overloading of binary operators.

Overloading binary operators

Suppose we want to define an operator minus to carry out subtraction of a complex number. We have to define the operator minus as part of a class. Let us assume that we call the class which contains complex number as number. The complex number consists of the real part and an imaginary part. A class definition for the same is given below:

```
class number {
    int real;
    int imag;
};
```

Here both the real number and imaginary number are declared to be integers for the sake of simplicity. We can add the constructor to initialize a given complex number as given below:

```
public :
number (int x, int y) {
    real = x;
    imag = y;
}
```

We can now declare an operator minus in the class as given below:

```
    number operator - (number num);
```

This is the declaration of the overloaded operator minus. It is actually declaration of a member function. The function name is operator minus. The return data type and the argument are objects of class number. We can write the function to subtract one complex number from another outside the class or even within the class. Before we do that, we have to understand how operator overloading works. Let us take the case of binary subtraction or overloading of binary operators. To invoke the overloaded binary operator, we need two objects. In C++, as you can see in the above declaration, only one object num of type number is received directly in the operator function. The other object is implied. This means that when we call the overloaded binary operator, the first object (operand) is implied and the second object is received through the argument. We will now write the function associated with the binary minus operator.

```
number operator -(number num){
    number temp_num;
    temp_num.real=real-num.real;
    temp_num.imag=imag-num.imag;
    return(temp_num);
}
```

In the above function, we have defined another temporary object called `temp_num`. In the next statement, when we address real, it corresponds to the real variable of the first object. The num.real refers to the second object. In the above, the real part of the second complex number is subtracted from the first. Similarly, the imaginary part of the second complex number is subtracted from the first in the next statement. The result is stored in temp_num and it is returned. Thus, overloaded operator function carries out more advanced calculation than a simple operator. Note that `operator` is a keyword. The return data type is the object itself and the argument is also an object. Look at the complete program given below:

Program 3.16

```
//To demonstrate binary operator –
#include<iostream>
class number {
    int real;
    int imag;
    public:
       number (){};
       number (int x, int y) {
          real=x;
          imag=y;
       }
    number operator -(number num){
       number temp_num;
       temp_num.real=real-num.real;
       temp_num.imag=imag-num.imag;
       return(temp_num);
    }
    void display(){
       std::cout<<real<<"+ j"<< imag;
    }
};
int main() {
    number num1(12, 14);
    number num2(4, 6);
    number num3;
    num3 = num1-num2;
    num3.display();
}
```

In the program, we define the following:

```
num1 = 12 + j 14 (although j is not mentioned)
num2 = 4 + j 6
```

Then num3 is declared with no values. This means the default constructor will be used for creating an empty object num3. Then we subtract num2 from num1 and store the result in num3. The subtraction is carried out using minus, which is an overloaded operator. When we look at the main function, it appears as if we have simply subtracted one number from the other. Actually, we have subtracted one object from the other through the overloaded operator function minus. Although we have overloaded the operator minus, the rules for operator precedence apply. For instance, if we have a high precedence operator such as * in an expression along with the overloaded operator minus, the * will get precedence over minus. This shall be noted.

Result of Program 3.16

```
8+ j8
```

The result of program confirms that the overloaded operator has performed the intended function. In a similar manner we can overload any other basic operators.

NOTES

Overloading unary operators

To overload unary operators, we use a single operand. Let us use the same complex number example to demonstrate overloading the operator ++.

Program 3.17

```
//To demonstrate unary operator prefix ++
#include<iostream>
using namespace std;
class number {
    int real;
    int imag;
    public:
    number (){};
    number (int x, int y) {
       real=x;
       imag=y;
    }
    void display(){
       cout<<real<<"+ j"<< imag<<"\n";
    }
    number operator ++();
};
number number:: operator ++(){
    number temp;
    temp.real=++real;
    temp.imag=++imag;
    return (temp);
}
int main() {
    number num1(5, 10);
    ++num1;
    num1.display();
    ++num1;
    num1.display();
}
```

When we overloaded binary operators and if the operator was a member function, it received one object.

This is because argument on the left hand side of the operator is an object. It is this object, which calls the member function. Thus the member function belongs to the object. Therefore, it is not necessary to additionally pass this object, which contains the

Check Your Progress

4. What is a copy constructor?
5. What is the role of a default constructor?
6. Why do we use the operator overloading function in a C++ program?

operator function as its member function. The operator function will understand the data members of the calling object without prefixing the object name. This is the secret of passing single object in case of binary operator functions.

In the same way, in case of unary operator function, it is a single object which calls its member function. Therefore, there is no need to pass the object again. Thus, when a unary operator is overloaded as a member function, it will not receive any object as in the example above. However, in the above example, it returns the object number.

Look at the declaration of the unary operator function, ++ which is a member function of class number. In the function, we have defined a temporary object of the same type called temp. We increment the real part of the calling object and store it in temp.real. We also increment the imag part of the calling object and store it in temp.imag. Then the temporary object temp is returned to the calling function.

In the main function, we create an object num1. See the ease with which the object itself is incremented as shown in the next statement namely ++num1. This is the interesting part of operator overloading. Then we display the object. In the next statement, we increment the object again and then display. Look at the result of the program given below:

Result of Program 3.17

```
6+ j11
7+ j12
```

In this case prefix and postfix of the overloaded operator will make a difference. After defining a prefix operator overloading, if we try to carry out a postfix operation the compiler will flag an error.

We said that the first object in the binary operator function is implied in the operator function. In the case of unary operator, object calling the function was also implied. We will see what facilitates this in the next section.

`this` Pointer

It is facilitated by another interesting concept of C++ called `this` pointer. `'this'` is a C++ keyword. 'this' always refers to an object that has called the member function currently. We can say that 'this' is a pointer. It points to the object that has called this function this time. While overloading binary operators, we use two objects, one that called the operator function and the other, which is passed to the function. We referred to the data member of the calling object, without any prefix. However, the data member of the other object had a prefix. Always 'this' refers to the calling object. 'this' can be used in place of the object name. The modified example of unary operator ++ using this pointer is given below:

Program 3.18

```
#include<iostream.h>
class number {
    int real;
    int imag;
    public:
        number (){};
```

```
         number (int x, int y) {
            real=x;
            imag=y;
         }
      void display(){
         cout<<real<<"+ j"<< imag<<"\n";
      }
      number operator ++();
   };
   number number:: operator ++(){
      real++;
      imag++;
      return *this;
   }
   void main() {
      number num1(5, 10);
      num1++;
      num1.display();
      ++num1;
      num1.display();
   }
```

Look at the operator function ++. Since it is unary operator, there is no need to pass an object. The calling object itself will be used for performing the operation. In this case, both the real and imaginary parts of the object are incremented. Using the following syntax returns the object:

```
      return * this ;
```

To compare the difference we must refer to the Program 9.6. Thus 'this' can be utilized conveniently. The result of the program is given below:

Result of Program 3.18

```
6+ j11
7+ j12
```

We can understand that there is really no need to create an object temp as in the previous example. When we call a function, 'this' refers to the object calling the function. Therefore, to return the object, we can simply return 'this'.

We may recall that in Chapter 7 we did not have to specifically refer to the object while displaying data members of the calling object. All this is due to the fact that each member function has access to a pointer called '`this`'. '`this`' pointer of a member function at any time of its execution refers to the object which invoked it. Actually 'this' points to the address of the corresponding object. Therefore using 'this' pointer we can access the data members of the object. That is the reason why when we simply refer to the name of the data member (without prefixing the object) it points to the data member of the object that called the operator function or for that matter any member function.

Combining operator and constructor overloading

Let us look at one more example, where the operator overloading and constructor overloading are carried out together.

Program 3.19

```
//To demonstrate operator & constructor overloading
#include<iostream>
class number {
    int real;
    int imag;
    public:
       number (){};
       number (int x, int y) {
          real=x;
          imag=y;
       }
       number (int x) {
          real=x;
          imag=0;
       }
    void display(){
       std::cout<<real<<"+ j"<< imag;
    }
    number operator +(number num);
};
number number:: operator +(number num){
    number temp_num;
    temp_num.real=real+num.real;
    temp_num.imag=imag+num.imag;
    return(temp_num);
}
int main() {
    number num1(12, 14);
    number num2(4, 6);
    number num3(7);
    number num4;
    num4 = num1+num2;
    std::cout<<"\n sum of num1 and num2=";
    num4.display();
    num4 = num4+num3;
    std::cout<<"\n sum of num1, num2 and num3=";
    num4.display();
}
```

NOTES

In this program, the operator + is overloaded. The operator is defined outside the class. Therefore, the name of the class is prefixed to the operator using scope resolution operator. There is no other difference in the operator function. The difference is in the number of constructors. We have one constructor with a single argument. The second argument in it namely imag is assumed to be zero. Now, look at the main function. The objects num1 and num2 are created using the parameterized constructor. The object num3 is created with a single value and so it will invoke the constructor with single argument. There is no other difference. In this program, we have used num4 to store the sum of num1 and num2. Then we use the same object to store the sum of num4 and num3. The result of the program is given below:

Result of Program 3.19

```
sum of num1 and num2=16+ j20
sum of num1, num2 and num3=23+ j20
```

Note that the operator overloading is defined in member functions in the above examples .

Operator overloading using friend function

As indicated in the previous chapter, the friend function is quite useful for operator overloading. The essential difference when we use a friend function instead of a member function is that it will receive two objects as arguments as against one in the above examples. The same example is modified to overload plus operator using friend function. It is given in program below:

Program 3.20

```
//To demonstrate binary operator using friend function
#include<iostream>
using namespace std;
class number {
    int real;
    int imag;
    public:
       number (){};
       number (int x, int y) {
          real=x;
          imag=y;
       }
       number (int x) {
          real=x;
          imag=0;
       }
    void display(){
       cout<<real<<"+ j"<< imag;
    }
    friend number operator +(number, number);
};
number operator +(number num1, number num2){
    num1.real+=num2.real;
    num1.imag+=num2.imag;
    return number(num1.real, num1.imag);
}
int main() {
        number num1(12, 14);
        number num2(4, 6);
        number num3(7);
        number num4;
        num4 = operator +(num1,num2);
        cout<<"\n sum of num1 and num2=";
        num4.display();
        num4 = operator+(num4,num3);
        cout<<"\n sum of num1, num2 and num3=";
        num4.display();
    }
```

NOTES

As could be seen the friend operator function is indicated by the `friend` keyword. Here the operator is defined outside the class. It is interesting to look at the operator function. It receives two objects, num1 and num2 of type number. We add the real part and imaginary part separately. Then we return both the parts through the return statement. We actually return the object number with two variables. In the earlier examples, we were returning an object which was declared in the function namely temp_num. In this example, we have not declared a separate object. Here, both the objects are received explicitly. Therefore, it is easy to add both the parts separately and return back. We must also look at the way in which the operator is invoked in the main function as given below:

```
        num4 = operator +(num1,num2);
```

Notice the difference in calling the operator. In the previous examples, we simply added using the overloaded plus as given below:

```
        num4 = num1+num2;
```

When we overload operator using friend function, we have to specifically use operator + and pass the two objects to be added. In the latter case the addition of object cannot be understood so intuitively. The result of the program given below confirms that the overloading works fine.

Result of Program 3.20

```
   sum of num1 and num2=16+ j20
   sum of num1, num2 and num3=23+ j20
```

We can pass the objects either by value as given in the above program or by reference.

When we use a member function for operator overloading, we pass one object for binary operation, such as addition, subtraction etc. It worked because the object used to call the operator function (the object on the left) is known through 'this' pointer. That is the reason why it was sufficient to pass only one object for binary operator overloading using member functions. On the contrary, when we use a friend function we have to pass two objects for binary operator overloading since it is not a member function and hence it does not have access to 'this' pointer. In all the examples seen above, we will notice that the return data type was the class itself. Actually, we were returning the objects in all the three cases seen above. Further, we followed a set pattern for achieving operator overloading. The first task was to define a class. Then in the class, we declared an operator overloaded function to carry out the operation. In the main function, we used operator as we use a simple operator.

Let us overload the equality `(==)` operator. The example below demonstrates overloading of the same.

Program 3.21

```
//To demonstrate operator ==
using namespace std;
#include<iostream>
class number {
    int real;
    int imag;
    public:
       number (){};
       number (int x, int y) {
          real=x;
          imag=y;
       }
    void display(){
       cout<<real<<"+ j"<< imag<<"\n";
    }
    void operator ==(number);
};
void number:: operator ==(number num){
    if ((real== num.real)&&(imag==num.imag))
       cout<<"numbers are equal \n";
    else
    cout<<"numbers are NOT equal \n";
}
int main() {
    number num1(5, 10);
    number num2(10, 5);
    num1==num2;
    number num3=num1;
    num1==num3;
}
```

In the above program, we have declared an overloaded operator function `(==)` as a member function of class number. Note here that the function returns void. This means the function will carryout some operations but does not return the object. Look at the definition of the operator function. It receives one object as argument like other programs. If the real and imag of the calling object is equal to the object passed namely num then the function will print ‘numbers are equal’; if not it will print that the ‘numbers are NOT equal’. In the main function, we create two dissimilar objects num1 and num2 and then we check the equality. Obviously, the program will return that the "numbers are not equal". Then we assign num1 to num3 and then check whether num1 is equal to num3. The result now will be that the numbers are equal. The result of the program confirms the above.

Result of Program 3.21

```
numbers are NOT equal
numbers are equal
```

Since we have seen a number of programs concerning overloading of operators, we can arrive at some conclusions. Comparing two objects as in the above example, will be cumbersome but for operator overloading. The operator overloading makes the statement such as num1 = = num2, obvious. Although, num1 and num2 are objects, we can easily guess that the statement above compares the respective parts of the objects as the operator will do with any primitive data types.

Overloading arithmetic assignment operator

We will now look at overloading of arithmetic assignment operators. Let us take `-=`. When we use this operator with a simple data type, it subtracts the second named variable from the first named and stores the result in the first named variable. For instance, var1– = var2. Here, we will have to do this operation for the object as a whole. The program below implements the arithmetic assignment operator – =.

Program 3.22

```
//To demonstrate Arithmetic assignment operators
#include<iostream>
using namespace std;
class number {
    int real;
    int imag;
    public:
       number (){};
       number (int x, int y) {
          real=x;
          imag=y;
       }
    void display(){
       cout<<real<<"+ j"<< imag<<"\n";
    }
    void operator -=(number);
};
void number:: operator -=(number num){
    real-=num.real;
    imag-=num.imag;
}
int main() {
    number num1(15, 27);
    number num2(5,7);
    num1-=num2;
    num1.display();
}
```

Look at the operator function. Here we are returning void since there is no necessity to return the object. As you know, if at all, we have to return the first named object. There is absolutely no need for returning the first named object, because it is visible in the class. Therefore, we return nothing from the operator function. In the main function, we call the overloaded operator in a very easy manner as given below:

```
    num1 -= num2 ;
```

In the next line, when we call the function display, the real and imag are available to the function since it is in the same class. Thus, the program implemented overloading of a shorthand notation operator in the same fashion.

Result of Program 3.22

```
10+ j20
```

Overloading + for concatenation of C style strings

We have seen in Chapter 4 that two strings can be concatenated using the overloaded + operator. The C++ standard library facilitated this. To understand operator overloading further, let us now try to implement the operator overloading concepts using C style string (character array) objects. Two programs are given below to understand the operator overloading, when the objects are strings. For this purpose, we define a class String. It has only one data member of type string. Assignment of a string to a string variable can be carried out through the library function strcpy(). Now look at the program which implements string concatenation which is basically adding one string after the other. This can be mathematically represented as plus.

Program 3.23

```
//To demonstrate operators using string
#include<iostream>
using namespace std;
class String {
    char st[100];
    public:
    String(){strcpy(st, " ");};
    String (char string1[]) {
       strcpy(st, string1);
    }
    void display(){
       cout<<st<<"\n";
    }
    String operator +(String);
};
String String:: operator +(String string2){
    String temp;
    strcpy(temp.st, st);
    strcat(temp.st, string2.st);
    return(temp);
}
```

Check Your Progress

1. Can a class contain a member function?
2. Explain public and private access specifiers.
3. Define a constructor.

```
int main() {
    String string3("My God is");
    String string4(" in my heart");
    String s=string3+string4;
    s.display();
}
```

NOTES

The above program has two constructors. The first one accepts a string without any characters. The second one accepts a string, which is an array of characters called string1.

The operator function is called `plus`. In the function, a temporary object called `temp` is created which is an empty string. We first copy the string corresponding to the calling object to temp. Then in the next statement, we use the function strcat(), which concatenates two strings. Here, the two strings are temp.st (which already contains the first string) and the string2.st which corresponds to the second object. Thus the last two operations help in achieving the following:

- Transferring the string corresponding to the calling object to the string of the object temp
- Adding after the first string, the second string corresponding to the second object

The function returns the concatenated string contained in the object temp. The usage of strcpy and strcat requires the inclusion of namespace std which we include in the program file.

Now look at the main function, `string3` and `string4` are the two String objects defined there. Now, we declare another String object and to it we assign the sum of `string3` and `string4`. [Note: String is the name of the class defined by us and string means array of characters. So look at the first letter carefully.] Any reader will understand easily that we are adding string3 and string4. However, this is implemented through overloading the operator plus in the operator function. The result of the program confirms the correctness of the program.

Result of Program 3.23

```
My God is in my heart
```

Although we have done the same operation in a very simple manner in Chapter 4, this example has been implemented from first principles to illustrate operator overloading of C style character arrays.

Multiple overloading

We can combine a program, which overloads plus operator for string concatenation, and also another program, which overloads the plus operator in the complex number class. If these two programs are put together, we might feel that there may be an ambiguity as to which overloaded plus operator has to be used. Actually the operator is always considered along with the associated data types. Therefore, when the operator is used along with two strings, the corresponding operator function will be called. When it is used along with the complex numbers, then again the operator function in the complex number class will be called. When the operator is used with primitive data type, then a simple addition will be carried out. When we combine programs with the same operator overloaded with different objects, then it is called multiple overloading. Let us confirm this through an example.

NOTES

Program 3.24

```
//To demonstrate multiple overloading
#include<iostream>
using namespace std;
class String {
    char st[100];
    public:
       String(){strcpy(st, " ");};
       String (char string1[]) {
          strcpy(st, string1);
       }
       void display(){
       cout<<st<<"\n";
    }
    String operator +(String);
};
String String:: operator +(String string2){
    String temp;
    strcpy(temp.st, st);
    strcat(temp.st, string2.st);
    return(temp);
}
class number {
    int real;
    int imag;
    public:
       number (){};
       number (int x, int y) {
          real=x;
          imag=y;
       }
       number (int x) {
          real=x;
          imag=0;
       }
    void display(){
       cout<<real<<"+ j"<< imag<<"\n";
    }
    number operator +(number num);
};
number number:: operator +(number num){
    number temp_num;
    temp_num.real=real+num.real;
    temp_num.imag=imag+num.imag;
```

```
        return(temp_num);
    }
    int main() {
        number num1(12, 14);
        number num2(4, 6);
        number num3(7);
        number num4;
        num4 = num1+num2;
        cout<<"\n sum of num1 and num2= \n";
        num4.display();
        num4 = num4+num3;
        cout<<"\n sum of num1, num2 and num3= \n";
        num4.display();
        String string3("My God is");
        String string4(" in my heart \n");
        String s=string3+string4;
        s.display();
        double var1=10.6, var2 =23.8;
        cout<<"sum of var1 and var2="<<var1+var2;
    }
```

Result of Program 3.24

```
    sum of num1 and num2=
    16+ j20
    sum of num1, num2 and num3=
    23+ j20
    My God is in my heart
    sum of var1 and var2=34.4
```

In the above program we have a class number that implements addition of complex numbers. We have another class String that implements addition of strings. In the main function we create objects of each class and add them. Before we exit the program we use the + operator to add two doubles. Thus in the same function we use the + operator to add different types of objects. Note also that we have overloaded the function display.

Almost all operators can be overloaded except the following.

dot operator `(.)`

conditional operator `(?)`

pointer to member operator `(.*)`

scope resolution operator `(::)`

When we use friend functions for operator overloading, additionally we cannot use the following operators for overloading:

Assignment operator `(=)`

Function call operator `()`

Array index operator `[]`

Class member access operator `->`

3.5 SUMMARY

In this unit, you have learnt the concept of classes and objects. Real entities or objects can be more easily defined using classes rather than built-in data types. A class provides a template or blueprint for defining user-defined data types. A class will contain declarations for data members, i.e., variables in a class. A class may also contain member functions. The access control keywords should be specified for both data members and member functions before the declarations. There are two types of access control keywords. The access specifier 'public' provides access to the members even from outside the class, whereas, if a member is declared private, then it can be accessed only from within the class. You have learnt that declaring both the data member and member function as private will not serve any useful purpose. Since they provide the interface to access data, the member functions are declared public. Since one of the objectives of OOP is data hiding, the data members are generally declared private.

An object is an instance of a class. It is like a variable of type class. It is referenced through an object reference variable.

You have learnt that constructors are special functions which are useful for assigning initial values to data members, and have the same name as that of the corresponding class. They have no return data type, not even void. The constructor assigns initial values to objects as soon as they are created. A constructor provides an easy way to assign initial values to data members. The constructors can be overloaded. The constructors can also be defined with default arguments. The destructor will destroy each object created using a constructor after it is no longer required. The destructors are called implicitly except when the memory is allocated using `new` keyword. Thus, both constructors and destructors are called implicitly. Even when destructors are not defined, the system destroys objects which are no longer required.

This unit also explained operators can be overloaded. Binary operators such as +, -, etc. can be overloaded to carry out similar types of operations on objects. When we overload a binary operator using a member function, only one object has to be passed to the overloaded operator function since the other object is assumed to be the one invoking the function. We can also use multiple overloading by combining more than one type of operator overloading.

3.6 KEY TERMS

- **Constructor:** These are special functions used to assign initial values to data members, and have the same name as that of the corresponding class. It also assigns initial values to the objects as soon as the object is created. Constructor provides easy way to assign initial values to data members.
- **Destructor:** It destroys each object created using a constructor after it is no longer required. Destructors are called implicitly – except when memory is allocated using a new keyword.
- **Default constructor:** The default constructor for any class is the constructor without any arguments. When no arguments are passed, the constructor assigns values specifically assigned in the body of the constructor. It can be zero or any other value. The default constructor can be identified by the name of the class followed by empty parentheses.

- **Copy constructor:** It copies the contents of another constructor or, to be precise, an object. Here the constructor assigns the values of data members of an existing object to a new object at the time of creation.

3.7 ANSWERS TO 'CHECK YOUR PROGRESS'

1. Yes, a class can contain declarations for data members, i.e., variables in the class and member functions.
2. The access specifier 'public' provides access to members even from outside the class, whereas, if a member is declared private, then it can be accessed only from within the class.
3. Constructors are special functions used to assign initial values to data members and have the same name as that of the corresponding class. It also assigns initial values to objects as soon as they are created. Constructors provide an easy way to assign initial values to data members.
4. A copy constructor copies the contents of another constructor or to be more precise, an object. Here the constructor assigns the values of data members of an existing object to a new object at the time of creation.
5. The default constructor for any class is the constructor with no arguments. When no arguments are passed, the constructor will assign the values specifically assigned in the body of the constructor. It can be zero or any other value. The default constructor can be identified by the name of the class followed by empty parentheses.
6. Operators can be overloaded in a C++ program. Binary operators such as +, -, etc. can be overloaded to carry out the same type of operations on objects. When we overload a binary operator using a member function, only one object has to be passed to the overloaded operator function since the other object is assumed to be the object invoking the function. We can also use multiple overloading by combining more than one type of operator overloading.

3.8 QUESTIONS AND EXERCISES

Short-Answer Questions

1. Find syntax errors, if any, in the following statements:

   a.
   ```
   class E7x2.cpp{
   ```
   b.
   ```
   class E7x1; {
   ```
   c.
   ```
   public
   void display(){
   ```
   d.
   ```
   Private:
   int var1;
   ```
   e.
   ```
   class Sample{
   int var1;
   public:
   void Sample(int a){
   ```

2. Find the errors, if any, in the following statements.
   (i) `~ className()`
   (ii) `~ className();`
   (iii) `className objectName();`
   (iv) `className();`
   (v) `number operator -(int num);`
   (vi) `int operator -(number num);`
   (vii)
```
operator mks()
{
   double x;
   int y;
   x=(foot*12+inch)*25.4;
   y=(int) x/10;
   z=int x %10;
   return mks;
```

Long-Answer Questions

1. State the output of the following programs:
   (i)
```
#include<iostream>
using namespace std;
class Sp16 {
   public:
   int number;
   int mark;
};
int main(){
   Sp16 swamy;
   swamy.number=709;
   swamy.mark=88;
   cout<<"Roll number ="<<swamy.number;
   cout<<"Mark        ="<<swamy.mark;
}
```
   (ii)
```
#include<iostream>
using namespace std;
class Student {
   public:
   int number;
   int mark;
   void display() {
         cout<<"Roll number ="<<number;
         cout<<"Mark        ="<<mark;
   }
 };
int main() {
      Student Vinay;
      Vinay.number =709;
      Vinay.mark =88;
      Vinay.display();
   }
```

(iii)
```
   #include<iostream>
   using namespace std;
   class Student{
      int number;
      int mark;
      public:
      void indata(int x, int y) {
            number =x;
            mark =y;
            cout<<"number ="<<number;
            cout<<"mark   ="<<mark;
      }
   };
   int main() {
      Student t;
      t.indata(709, 100);
   }
```

(iv)
```
   #include<iostream>
   class num{
      int a, b;
      public:
      num(int var1, int var2){
            a=var1;
            b=var2;
      }
      inline int mul(){return a*b;}
   };
   int main(){
      num ob1(10, 20);
      std::cout<<ob1.mul();
   }
```

(v)
```
   #include<iostream>
   using namespace std;
   int object_count;
   class Account {
      int number;
      public:
            Account(int x){
```

NOTES

```
                cout<<++object_count<<" created\n";
                number=x;
            }
        ~Account(){
            cout<<object_count--<<" destroyed\n";
        }
    };
    int main() {
        Account Ac1(10);
        Account Ac2(20);
        Account Ac3(30);
        Account Ac4(40);
        Account Ac5(50);
        Account Ac6(60);
    }
```

(vi)
```
#include<iostream>
using namespace std;
class Book {
    int number;
    double price;
    public:
            Book (int x, double y=100) {
                    number=x;
                    price=y;
            }
    Book (Book & b1) {
            number=b1.number;
            price=b1.price;
    }
    void display() {
            cout<<"\n "<<number;
            cout<<" = "<<price;
    }
};
int main(){
    Book book1(001, 100.0);
    book1.display();
    Book book2(book1);
    book2.display();
    Book book3(book2);
    book3.display();
    Book book4(002, 200.0);
    book4.display();
}
```

NOTES

(vii)
```
#include<iostream>
using namespace std;
class B;
class A{
   int real;
   double im;
   friend void Add(A &, B &);
   public:
         A(int x, double y){
               real=x;
               im=y;
         }
};
class B {
   int re;
   double imag;
   friend void Add(A &, B &);
   public:
         B(int x, double y){
               re=x;
               imag=y;
         }
};
void Add(A& Ac1, B& Ac2) {
   cout<<"real part =" <<(Ac1.real+Ac2.re);
   cout<<" Imaginary part =" <<(Ac1.im+Ac2.imag);
}
int main() {
   A obj1(25, 10.50);
   B obj2(35, 20.70);
   Add(obj1, obj2);
 }
```

(viii)
```
#include<iostream>
using namespace std;
class rational{
   long den, num;
   public:
   rational(long numerator=0, long denominator=1){
         num=numerator;
         den=denominator;
   }
   void assign(long numerator, long denominator){
         num=numerator;
         den=denominator;
```

NOTES

```
   }
   void operator*=(rational n){
         num*=n.num;
         den*=n.den;
   }
   void display(){
         cout<<num<<"/"<<den;
   }
};
int main() {
   rational a(4,5);
   rational b(3, 5);
   a*=b;
   a.display();
}
```

(ix)
```
#include<iostream>
using namespace std;
class currency {
   int dollar;
   int cent;
   public:
   currency (int x, int y) {
         dollar=x;
         cent=y;
   }
   void display(){
         cout<<" dollar="<< dollar;
         cout<<" cent="<<cent<<"\n";
   }
   operator int()
   {
         int var;
         var=dollar*100 +cent;
         return var;
   }
};
int main() {
   currency L1(10, 9);
   L1.display();
   cout<<L1<<"\n";
   currency L2(5, 10);
   L2.display();
   cout<<L2<<"\n";
}
```

2. Write programs for the following.
   (i) To subtract the length of one object (in feet and inches) from that of another.
   (ii) Write a friend function for checking the equality of data members (length in cm and mm) of objects of two classes.
   (iii). Write a program with copy constructor for data members of the following class.

```
class employee{
     string Emp_name;
     unsigned int Emp_no;
     double basic_pay;
     char is_regular;
     double tax;
     double allowanc;
};
```

UNIT 4 CLASS INHERITANCE

NOTES

Structure

4.0 INTRODUCTION

In this unit, you will learn about the inheritance property which facilitates the reusability of software components. User-defined types, namely classes, facilitate inheritance. A class which has already been declared 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 the properties of the base class according to requirement. However, adding a new class does not alter the base class. C++ facilitates multiple and multi-level inheritance. You will learn that a protected member can be accessed from a derived class, but not by any other outside class. This is the use of the protected access specifier. Thus, a protected member is accessible from a derived class but not from any other class outside. If the access specifier is public, any other class can access the member; but if it is private, it cannot be accessed by any other class; if it is protected it can be accessed from the derived class, but not any other class outside.

4.1 UNIT OBJECTIVES

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

- Explain single inheritance
- Define a derived class and a base class
- Explain a virtual class
- Understand friend and static functions
- Explain multiple inheritance
- Understand the concept of polymorphism
- Write a C++ program using derived and base classes
- Write a C++ program using multiple inheritance

4.2 DERIVED CLASS AND BASE CLASS

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 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.

The latest versions of C++ allow multiple inheritance. Thus a number of possibilities exist with regard to inheritance in C++. They are listed below diagrammatically:

Simple inheritance

A simple inheritance is a relationship between one parent and one child as depicted in Figure 4.1 below:

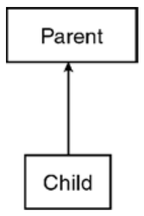


Figure 4.1 *Simple inheritance*

The arrow points up indicating that the child borrows properties from the parent. The arrow can point the other way around indicating that the parent lends his properties to his child.

Multi-level inheritance

The inheritance can continue more than once depending on the need. Figure 10.2 below indicates a multi-level inheritance.

In the multi-level inheritance depicted above, the child inherits from P3, which again is a derived class of P2 and so on. Therefore, the child may have some properties of P1. P1, P2 and P3 are base classes to one class each. P2, P3 and child are derived classes, each deriving from one base class.

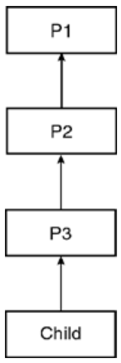


Figure 4.2 *Multi-level inheritance*

Multiple inheritance

On the contrary, multiple inheritances refer to a class inheriting the properties of multiple base classes as illustrated in Figure 4.3.

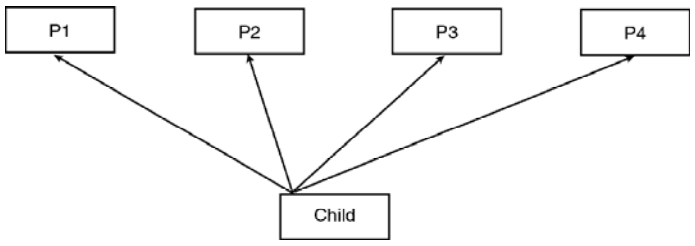


Figure 4.3 *Multiple inheritance*

In the above case, the child can inherit the properties of P1, P2, P3 & P4, which are all base classes. The child is the only derived class..

Hierarchical inheritance

A base class can be a parent for a number of derived classes which can, in turn, be a parent for a number of classes. This is known as hierarchical inheritance. Hierarchical inheritance is illustrated in Figure 4.4.

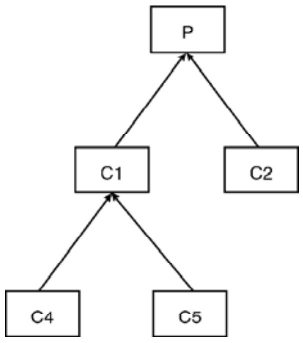


Figure 4.4 *Hierarchical inheritance*

Thus, using common sense, various combinations of inheritance can be deduced. Now let us look at the features of inheritance with specific reference to C++.

Features of inheritance

C++ facilitates multiple and multi-level inheritance. Inheritance provides an elegant method for adding new attributes and behaviour to a class. A simple inheritance is illustrated pictorially in figure 4.5.

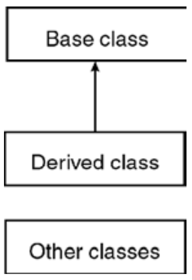


Figure 4.5 *Simple inheritance*

To reiterate, the derived class inherits the behaviour and attributes of the base class. However the vice versa is not true. The derived class can add its own properties i.e. data members (variables) and functions. It can extend or use properties of the base class without any modification to the base class. We declare the base class and derived class as given below:

```
class base_class {
};
class derived_ class : public base_ class {
};
```

The names of base_class and derived_class can be any valid identifiers. Note the public prefix to the base_class. Another example of inheritance is given below:

```
class Account   {
};
class BankAccount :public Account {
};
```

Here Account is the base class and BankAccount is its derived class. In both the above examples, only the declarations of the classes are given. Each class, in turn, may contain data elements and functions.

Let us look at an example where a derived class uses the data members of a base class.

Program 4.1

```
//To demonstrate Inheritance of classes
#include<iostream>
using namespace std;
class Account {
    public:
    int number;
    double balance;
};
class SubClass : public Account{ //subclass declaration
    public:
```

```
    double crediting(double deposit) { //credit operation
    return (balance + deposit);
    }
};
int main()
{
    double newbal;
    SubClass Lakshmi;
    Lakshmi.number=003;
    Lakshmi.balance= 2000.00;
    newbal=Lakshmi.crediting(1000.00);
    cout<<"\n Account number ="<<Lakshmi.number;
    cout<<"\t old Balance ="<<Lakshmi.balance;
    cout<<"\n new Balance ="<<newbal;
}
```

Let us analyze the program to understand inheritance.

Here Account is a base class with two data members. It is a base class since it does not inherit from any other class. For the sake of convenience, we have named the derived class as SubClass. We can give other names to derived classes. The derived class SubClass inherits properties from Account. In the main function, we create an object Lakshmi of SubClass. Then using the data members of base class, we assign values to the object of the SubClass. Although number and balance are data members of base class Account we refer to them with ease even for the object of derived class. This is possible since data members of the base class are public. It is also because we inherit from public class Account. Hence data members of base, class are transparent to the derived class. Then we call method crediting of the derived class. If we look at the print statements all the data members of the base class are simply used by the derived class using the dot operator. The result of the program is given below :

Result of Program 4.1

```
Account number =3 old Balance =2000
new Balance =3000
```

The base class Account was available with two data members, namely number and balance. Assume that it had already been tested and working. Suppose now there is a need to provide for credit operation. In the procedure oriented programming paradigm, we have to scrap the old program and start all over again to develop a new one with added facility for credit operations. In OOP, we simply extend a class through inheritance. We have done exactly the same in the above program. Since no new data members need to be added, we have not declared any in SubClass. However the SubClass can use the public members of the class Account. This has saved declaration of the data members again in the SubClass. However, we need to add a new function for crediting. Hence it has been added.

Since the data members of the base class are declared as public they can be accessed from outside, in this Example, from function main. Since there is a public qualifier before the base class name while declaring SubClass, the latter can inherit the public members of class Account. Only because of the above two declarations, we are able to use the (.)dot operator for assigning and printing values of the variables of the

NOTES

SubClass. The above program has been built with public data members to explain the concept of inheritance. As we go along we will refine our concepts and develop professional programs.

Now let us modify the program so that the derived class uses both the attributes and behaviour of the base class. Look at the program given below :

Program 4.2

```
//To demonstrate Inheritance of classes
#include<iostream>
using namespace std;
class Account {
    public:
    int number;
    double balance;
    public:
    double crediting(double deposit) {        //crediting
       return (balance + deposit);
    }
};
class SubClass :public Account{
};
int main() {
    double newbal;
    SubClass Lakshmi;
    Lakshmi.number=003;
    Lakshmi.balance=2000.00;
    newbal=Lakshmi.crediting(1000.00);
    cout<<"\n Account number ="<<Lakshmi.number;
    cout<<"\t old Balance ="<<Lakshmi.balance;
    cout<<"\n new Balance ="<<newbal;
}
```

Here, the base class contains the member function crediting in addition to the data members. The derived class does not have any variables or functions in this case. It is rather an empty class. But in main function, an object Lakshmi of SubClass is created. It uses both the variables and a method of the parent class, without any difficulty. The result of the program is given below :

Result of Program 4.2

```
Account number =3 old Balance =2000
new Balance =3000
```

In this program too, we have declared the data members as public to demonstrate inheritance easily.

Access specifiers and inheritance

One of the major objectives of OOP is data integrity. Modifying data in the programs needs a thorough understanding of the concept of encapsulation (where the data members and functions are closely tied to a class). The earlier programs allowed access of the data members to the objects through the dot operators. As discussed in Chapter 7, we can restrict the access by using access specifiers before the declaration of data members as given below:

```
private:
int number ;
```

Here, private is an access specifier. The other two access specifiers are public and protected. The access specifiers are applicable to members of a class as well as the class itself. The members of a class are nothing but the data members and member functions.

Let us understand the difference between the access specifiers, private and public. The objects outside the class can also use dot operators, just like structure members, to access the members that are declared public. But only the members of the same class can access a private member. If a code outside the class wants to access a private data member, it can access it through a public member function of the class. Thus the access of private members is restricted. The default access specifier i.e. when no access specifier precedes the member, is private in a class.

In addition to the known specifiers private and public, there is one more access specifier namely protected. Let us discuss the impact of the prefix of the three types of access specifiers on the accessibility of the members, both data and functions:

In the class where the member is declared (own class)

When a member is declared `public`, `protected` or `private`, it can be accessed within the same class without any difficulty.

From (objects) outside the class

As mentioned earlier, if a member is declared public even the objects from outside the class can access it. We have been accessing public data members from main function which was outside the class.

We also know that if a member is declared private in the class then it cannot be accessed from outside the class. Similarly, objects from outside the class cannot access a member with access specifier `protected`: However there is an exception which is given below.

From derived class

A protected member can be accessed from a derived class, but not by any other outside class. This is the use of the access specifier protected. Thus a protected member is accessible from a derived class, but not from any other class outside.

To summarize, there is absolutely no problem for access of members of a class within itself or from its friends. But when it comes to access of members of a class outside it, then one has to look at the access specifier. If the access specifier is public, any other class can access the member. But if it is private, it can not be accessed by any other class. If it is protected then it can be accessed from the derived class, but not any other class outside (except its derived class).

Let us now look how it is relevant to Inheritance. We have to modify a previous statement. We concluded there, that the data members of a base class are transparent to the derived class. It is so, if the data members are public or protected. A private member of a base class is not accessible even to its derived class. This point should be noted carefully. A program to illustrate the concept is given below:

NOTES

Program 4.3

```
/*To demonstrate that private variable
is not transparent to subclass*/
#include<iostream>
using namespace std;
class Account {
    public:
    int number;
    private:
    double balance;
    public:
    void getvalue(double var1){
    balance=var1;
    }
    double crediting(double deposit) { //credit operation
    return (balance + deposit);
    }
};
    class SubClass: public Account{
};
int main() {
    double newbal;
    SubClass Lakshmi;
    Lakshmi.number=003;
    /*Lakshmi.balance=2000.00; If you include this line, it
will not compile*/
    Lakshmi.getvalue(2000.00);
    newbal=Lakshmi.crediting(1000.00);
    cout<<"Account number ="<<Lakshmi.number;
    /*cout<<"old Balance ="<<Lakshmi.balance); This is also
problem*/
    cout<<"new Balance ="<<newbal;
}
```

The class Account has two data members, one public and the other private. Look at main function. We can assign value to the base class variable number easily by using dot operator as given in the program. But if we try to do the same for balance as given within the commented statement, the program will not compile, since balance has been declared private. Lakshmi.balance cannot be accessed since balance is a private variable of the base class and Lakshmi is an object of the derived class. To access it however, we have declared a public function getvalue in the base class. Since the function is part of the same class, it can access balance. The function is accessible from the main function since it is declared public. Since Lakshmi is an object of the derived class, it can use the public function of its base class. Therefore, Lakshmi.getvalue() is valid. Again printing number is easy. But, we cannot print balance because it is not transparent. If we are still interested in printing the balance, we have to create a public function in the base class for printing and the object can call it. Try to do it yourself as an exercise. The result of the program is given below:

Result of Program 4.3

```
Account number =3new Balance =3000
```

The program explains the concept of access specifiers clearly.

Constructors and inheritance

A constructor provides an elegant methodology for assigning initial values to the data elements of the objects. Let us see, how a derived class can inherit the constructor of a class. An example program using constructors is given below:

Program 4.4

```
/*To demonstrate inheritance of classes
with constructors*/
#include<iostream>
using namespace std;
class Book {
    protected:
       int number;
       double price;
    protected:
       Book(int a, double b) {
          number = a;
          price = b;
       }
};
class SubClass : public Book {
    public:
       SubClass(int x, double y): Book(x, y)
       { }
    public:
       void display()
       {
          cout<<"number = "<<number;
          cout<<"price = "<<price;
       }
};
int main() {
    SubClass javab(1,342.0);
    javab.display();
}
```

Study the program carefully before we discuss any further.

There is nothing new as far as constructor in the base class is concerned. We have declared the base class constructor as protected. It can be either declared as public or protected. The latter is more safe. Since the constructor is declared protected, it can be accessed only from the derived class and not from anywhere else. Had it been declared public it could have been accessed from anywhere. Remember that a constructor cannot be declared private. In the derived class, we declare the constructor as given below:

```
SubClass(int x, double y): Book(x, y)
{ }
```

The prototype of the constructor of the derived class consists of the declaration of the constructor. Then it is related to the constructor of the base class by the following:

```
    : Book(x,y)
```

In the above case, the declaration of the constructor of the derived class contains the same number of parameters as that of the base class. This is the simplest constructor. Here, the variable x corresponds to the first parameter of the constructor of the base class. Similarly, the variable y corresponds to the second parameter of the base class. The derived class inherits both the data members of the base class. It has not added any other data element and hence the braces following the prototype of the derived class constructor is empty. Thus when an object of the derived class is declared, its initial values will be assigned automatically through the constructor of the base class. Now let us look at the program.

The derived class contains a function display. In the main function, we declare an object javab and pass initial values. These values will be passed to the constructor in SubClass. It is accessible from main function because it is public. This in turn will access the base class constructor which, being protected, is accessible from the derived class. Now the values for number and price will be assigned.

The data members and the constructors of the base class are all declared protected. Since they are accessed only in the derived class, there is no problem. However, they cannot be accessed from outside the base class or its derived class.

Then the main function calls function display of the SubClass which displays the values as given below:

Result of Program 4.4

```
number = 1price = 342
```

Assume that there is only one variable of type float in the base class and derived class, then the constructor of the derived class will appear as given below:

```
    SubClass(float f): Book(f)
    { }
```

Note the correspondence between the formal parameter in the constructor of the derived class and the actual parameter to be passed to the constructor of the base class-it is "`f`" at both places.

Now let us look at one more example involving a constructor. The braces have been provided to declare data members of the derived class, if any. In the two examples given above, there weren't any additional data members of the derived class. Hence there was nothing in between the braces. The following example shows the derived class using all the data members of the base class and having its own data members.

Program 4.5

```
/*To demonstrate constructors when derived class has its own
member*/
```

NOTES

```
#include<iostream>
using namespace std;
class Book {
    protected:
        int number;
        double price;
    protected:
        Book(int a, double b) {
            number = a;
            price = b;
        }
};
class SubClass: public Book {
    private:
        int pages;
    public:
        SubClass(int var1, double var2, int var3): Book(var1,
var2)
        {
            pages=var3;
        }
    void display()
    {
        cout<<"\n number ="<<number;
        cout<<"\n price ="<<price;
        cout<<"\n pages ="<<pages;
    }
};
int main() {
    SubClass javab(1,342.0,450);
    javab.display();
}
```

There is a marked difference in the derived class. We give a list containing 3 parameters in the constructor declaration of the SubClass as given below:

```
SubClass(int var1, double var2, int var3): Book(var1, var2)
```

The first two parameters refer to the base class and the last one (var3) corresponds to the derived class. The statement, `Book(var1, var2)` assigns the variables of the base class constructor to the derived class constructor. The third parameter belonging to the derived class is given in between the braces that follow (reproduced below).

```
    {
        pages=var3;
    }
```

Thus, the constructor of the derived class is defined efficiently and unambiguously.

This actually means that the constructor of the base class is called with the arguments var1 and var2. Therefore, the constructor Book in the base class will be called with var1 and var2. The result of this is that number will get the value of var1 and price will get the value of var2. Then, the variable pages of derived class will get the value of var3.

In the main function, an object javab is created with the values shown. The values are passed to appropriate constructors i.e. 1 and 342.0 are passed to Book() through the derived class and pages gets the value 450. The object calls display function to print the values. The result of the program is given below:

Result of Program 4.5

```
number =1
price =342
pages =450
```

The point to be noted while inheriting constructors is:

The declaration of the constructor of the base class precedes that of the derived class. The constructor of the derived class initializes its own variables. However, those data members inherited from the base class are initialized by the base class constructor only.

Let us now summarize the effect of access specifiers assuming that the derived class inherits a public base class, which is given in Table 4.1.

Table 4.1 *Accessibility of 3 types of members*

Access type	In own class (or its friend)	From outside the class (except derived classes)	From derived class
`Public:`	Accessible	Accessible	Accessible
`Private:`	Accessible	Not Accessible	Not Accessible
`Protected`	Accessible	Not Accessible	Accessible

Inheritance types

The three types of inheritance are as given below:

- Public
- Private
- Protected

In all the examples seen so far, the inheritance was declared public, indicated by the public keyword before the name of the base class. So the derived class can access the public/protected data members and member functions. They remain so in the derived class too. Objects outside the class can assess the public members of the derived class inherited from the base class using the dot operator. However the protected members can only be accessed from within the derived class. Private members of the base class cannot be accessed in the derived class.

If the inheritance is declared `private`, then the `public/protected` members of the base class become private members of the derived class. Hence they can be accessed from within the derived class and cannot be accessed from outside the derived class. But any object will be declared outside the class, in the main function. Hence the members of the base class will not be available to the objects of the derived class in private inheritance. Such inheritance will serve no useful purpose since the objects of the derived class cannot access the members of the base class. If inheritance is declared protected, then the public and protected members of the base class are accessible as protected members of the derived class. The accessibility of the data members in the three types of derivation as depicted in Figure 4.6.

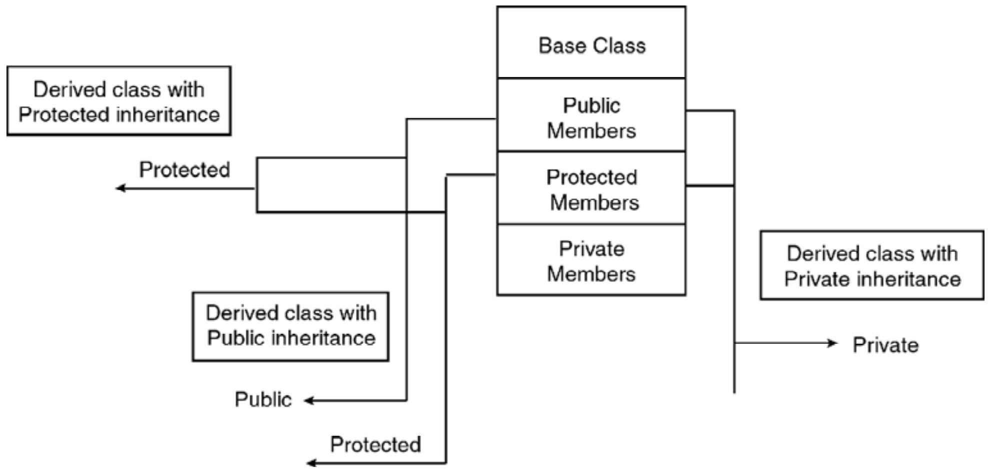

Base Class
Public Members
Protected Members
Private Members
Derived class with Protected inheritance
Protected
Derived class with Public inheritance
Public
Protected
Derived class with Private inheritance
Private

Figure 4.6 *Effect of inheritance types on data types*

Essentially `protected` is similar to `private`. But, protected members are accessible in the derived classes. Therefore, generally the data members are declared `private`. If there is any chance of inheritance of the class in future, then they may be declared protected. The default access specifier for inheritance is `private`. A `protected` member of a base class remains protected in all subsequent derived classes provided the type of inheritance is either `public` or `protected`.

Use of scope resolution operator to avoid conflict between data members

In the previous examples, we have given unique names to the data members of the base class and derived class. On the contrary let us see what happens if we give identical names to the data members of base class and derived class. We can avoid any conflict, by adding scope resolution operators to the variable of the base class as illustrated below :

Program 4.6

```
//use of scope resolution operator
using namespace std;
#include<iostream>
class Book {
    protected:
       int number;
       double price;
    public:
       Book(int a, doubled)
       {  number=a;
          price=d;
```

```
        }
};
class SubClass :public Book {
    int number;
    public:
       SubClass(int p, double q, int r):Book(p, q)
       {
          number=r;
       }
    void display(){
       cout<<"\n number ="<<Book::number;
       cout<<"\n price ="<<price;
       cout<<"\n pages ="<<number;
    }
};
int main() {
    SubClass javab(1,240.0, 405);
    javab.display();
}
```

We have a variable of the base class Book named number. In the Derived class also we have a variable called number. Look at the derived class method display. There the number refers to the variable of the derived class. We refer to the variable number of the base class using scope resolution operator as, Book::Number. Since the variable price is unique there is no need to prefix it with Book. The moment we give the same name to a derived class variable it hides the base class variable. Therefore, to bring it out, we use scope resolution operator.

In the main function, we create object javab and initialize the variables using the constructor. Then we call the method display. The result of program is given below:

Result of Program 4.6

```
 number =1
 price =240
 pages =405
```

What we have actually done in the above program is hiding a variable i.e. the base class variable was hidden by the derived class variable in the derived class.

Multi-level inheritance

So far we have seen only one level of inheritance - one parent and one child. We can have any number of levels of inheritance. The child can in turn become a parent for another child and so on as the program below illustrates.

Program 4.7

```
//Multi-level inheritance
#include<iostream>
using namespace std;
class Book {
    protected:
       int number;
       double price;
    public:
       Book(int a, double b){
          number=a;
       price=b;
        }
};
class SubClass :public Book {
    protected:
       int pages;
    public:
       SubClass(int p, double q, int r):Book(p, q){
          pages=r;
       }
};
class SubSubClass :public SubClass {
    protected:
       double discount;
    public:
       SubSubClass(int c, double d, int e, double f)
          :SubClass(c,d,e){
          discount=f;
    }
    void display(){
       cout<<"\n number ="<<number;
       cout<<"\n price ="<<price;
       cout<<"\n pages ="<<pages;
       cout<<"\n discount ="<<discount;
    }
};
int main() {
    SubSubClass javab(1,240.0, 405, 0.3);
    javab.display();
}
```

NOTES

In the above program, we have a base class `Book`. It has two data members namely `number` and `price`. The data members have been declared as protected anticipating inheritance of the class. Then there is a constructor for the class, which will accept an integer for the variable number and a double for price.

The class `SubClass` extends the class `Book` and so it is a derived class of `Book`. It has its own data member namely, pages which is declared as protected, again anticipating inheritance of the class. Now, look at the constructor of the `SubClass`, which is declared public. When an object of the derived class is created, the variable

NOTES

pages will receive data through its own constructor - the values for the other two variables will be received through the constructor of the class `Book`.

The `SubSubClass` is a derived class of `SubClass`. This is the second level of inheritance. There is absolutely no difference with regard to declaration of the second level derived class. It inherits from public `SubClass`. The `SubSubClass` really doesn't know whether `SubClass` is itself a derived class. It only knows that `SubClass` is its parent. Look at the way the constructor of the `SubSubClass` is defined. It uses four data members. Three of them are derived from its base class namely, `SubClass` which in turn derives two of its data members from its base class namely class Book. As far as the `SubSubClass` is concerned, it has its own data member discount and the other three variables are inherited from its base class. Thus, it is quite logical and straightforward. The `SubSubClass` class has also a function display. See the ease with which the data members are recognized in the function display. There is no prefix to the data members. Let us analyze how it is feasible. For instance, the variable number is not known to `SubSubClass`. So it will look for the variable number in its base class namely `SubClass`. Since, `SubClass` also doesn't know, it will look up to its base class Book, where it is defined and hence the value corresponding to number of the object of `SubSubClass`, (in this case javab) will be assigned using the constructor of the class Book. Similarly, the value corresponding to the pages of javab will be assigned through the class `SubClass`. However, the constructor of the class `SubSubClass` will assign the value corresponding to the variable discount of the object javab. Therefore, when the object javab is created, its initial values for the data members will be assigned through the constructors of the respective classes. Then, the function main calls display, which is a public function of `SubSubClass`. Therefore, the values of all the four data members of the object javab will be displayed as given below:

Result of Program 4.7

```
number =1
price =240
pages =405
discount =0.3
```

See the case with which we can refer to the first level variables namely, `number` and `price`. This is the advantage of inheritance. We can keep extending the properties of the classes in a linear manner. The first and second level classes can have functions also. For the sake of simplicity, here they do not have any functions.

Function overriding or method overriding

Both the terms function overriding and method overriding are used interchangeably. Some call it function hiding. It is defined as the ability to change the definition of an inherited method or attribute in a derived class. When multiple functions of the same name exist with different signatures it is called function overloading. When the signatures are the same, they are called function overriding. Function overriding allows a derived class to provide specific implementation of a function that is already provided by a base class. The implementation in the derived class overrides or replaces the implementation in the corresponding base class.

As we learnt in the previous unit, function overloading arises when there are multiple functions with different signatures. In such cases, to call a particular function, we have to match the signature. We were carrying out different operations with the functions having same name, but different parameter list. This was also called polymorphism.

When we deal with inheritance we may come across, one function in the base class and another function with the same signature in the derived class as illustrated in program below:

Program 4.8

```
//To demonstrate function hiding
#include<iostream>
using namespace std;
class Book {
    protected:
       int number;
       double price;
    public:
       Book(int a, double d){
          number=a;
       price=d;
    }
    void display(){
       cout<<"\n number ="<<number;
       cout<<"\n price ="<<price;
    }
};
class SubClass :public Book {
    int pages;
    public:
       SubClass(int p, double q, int r):Book(p, q)
       {
          pages=r;
       }
    void display(){
       cout<<"\n number ="<<number;
       cout<<"\n price ="<<price;
       cout<<"\n pages ="<<pages;
    }
 };
int main() {
    Book CSharp(2, 180.0);
    CSharp.display();
     SubClass javab(1,240.0, 405);
     javab.display();
 }
```

We have a method display receiving and returning nothing in the base class Book. In the derived class we have the method display also receiving and returning nothing.

Thus the name and the signatures of both the functions are same. However, they contain different program statements. The function in the derived class hides the function in the base class.

In the above example, in the main function, we create object `Csharp` of class `Book` and then call function display. Then we create object javab of `SubClass`. Then the object calls `display()`. Although both the base class and derived class have the functions named display, there is no confusion. The function in the respective classes is called. This was possible because the object calling the function is identifiable as belonging to `Book` and `SubClass` respectively. The result confirms this

Result of Program 4.8

```
number =2
price =180
number =1
price =240
pages =405
```

It is possible to use both the functions in the derived class. The following program illustrates how so.

Program 4.9

```
//To demonstrate Function hiding
#include<iostream>
using namespace std;
class Book {
    protected:
        int number;
        double price;
    protected:
        Book(int a, double d){
            number=a;
            price=d;
        }
    void display(){
        cout<<"\n number ="<<number;
        cout<<"\n price ="<<price;
    }
};
class SubClass:public Book {
    int pages;
    public:
        SubClass(int p, double q, int r):Book(p,q){
            number=p;
            price=q;
            pages=r;
        }
    void display(){
         Book::display();
         cout<<"\n pages ="<<pages;
      }
  };
  int main()
  {
      SubClass javab(1,240.0, 405);
      javab.display();
  }
```

NOTES

We call base class function in the derived class function display by using the scope resolution operator as given below :

```
      Book::display() ;
```

When we call the function display in the SubClass, it will first print the values of the data members of the base class by invoking the function of the base class. The next print statement will then be executed.

Result of Program 4.9

```
   number =1
   price =240
   pages =405
```

This program shows how to invoke hidden function by using scope resolution operator. Thus the hidden members namely data and functions can be invoked by using the scope resolution operator (: :).

Object slicing—base class reference to derived class

We have seen instances of derived class referring to base class. We will now take an example to show that base class can refer to derived class.

Program 4.10

```
  //Base class reference to derived class
  #include<iostream>
  using namespace std;
  class Book {
      int number;
      double price;
      public:
         Book(int a, double d){
            number=a;
            price=d;
         }
      void display(){
         cout<<"\n number ="<<number;
         cout<<"\t price ="<<price;
      }
  };
  class SubClass :public Book {
        int pages;
        public:
            SubClass(int p, double q, int r):Book(p,q) {
                pages=r;
            }
    };
    int main() {
        SubClass javab(1,240.0, 405);
        javab.display();
        Book cb=javab; //subclass object is assigned to base
    class
        cb.display();
    }
```

In the program, we have a method display in the base class, which can be used by the inheritor, namely derived class. In main function, we have created an object javab of derived class with initial values. Then we call the method display. Till now we have not seen anything new. Then we create an object cb of the base class and we assign the object of the derived class to that of the base class. This is new.

We know that the characteristics of the derived class object are not known completely to the base class. It only knows the properties lent by it to the derived class. Therefore, when such an assignment is made, it will only take properties given by it. Thus, the object will have the data members number and price and not pages. This phenomenon is known as object slicing. If we try to access cb.pages, it will result in compilation error. The result of the program is given below :

Result of Program 4.10

```
number =1 price =240
number =1 price =240
```

Thus, we have demonstrated that a base class object can refer to a derived class object and get the values corresponding to its data members. This property is quite useful.

Function overriding and multi-level inheritance

Function hiding can be extended to multi-level inheritance as the following program shows.

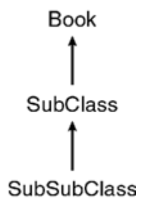
Program 4.11

```
//Function hiding in multiple inheritance
#include<iostream>
using namespace std;
class Book {
    protected:
    int number;
    double price;
    public:
        Book(int a, double b){
            number=a;
            price=b;
         }
      void display(){
         cout<<"\n Printing from class Book";
         cout<<"\n number ="<<number;
         cout<<"\n price ="<<price;
      }
};
class SubClass: public Book {
      protected:
      int pages;
      public:
         SubClass(int p, double q, int r): Book(p, q){
            pages=r;
         }
      void display(){
         cout<<"\n Printing from class SubClass";
         cout<<"\n number ="<<number;
         cout<<"\n price ="<<price;
         cout<<"\n pages ="<<pages;
      }
};
class SubSubClass : public SubClass{
      protected:
      double discount;
      public:
         SubSubClass(int c, double d, int e, double f)
         :SubClass(c,d,e){
         discount=f;
      }
      void display(){
         cout<<"\n printing from class SubSubClass";
         cout<<"\n number ="<<number;
         cout<<"\n price ="<<price;
         cout<<"\n pages ="<<pages;
         cout<<"\n discount ="<<discount;
      }
};
int main()
{
      SubSubClass javab(1,240.0, 455, 0.25);
      Book cb=javab;
      cb.display();
      SubClass kb=javab;
```

```
    kb.display();
    javab.display();
}
```

NOTES

We have the same classes as given below:



All the three classes have a function named `display()`, but there are differences in the statements in the functions. This is also a case of function hiding i.e. the base class function is hidden by the `SubClass` function, which is in turn hidden by `SubSubClass` function.

Now look at the main function. We create an object javab of class `SubSubClass`. Then, we create another object cb of class Book and assign to it javab. As already discussed, cb will get values for its data members from javab. We now call function display for object cb. We successively create object kb of class `SubClass` and assign to it the object javab. Then we call display function for object kb.

We have called the hidden functions, first the one corresponding to class Book and then the one corresponding to `SubClass`. Although all the three calls are for function display, the program calls the appropriate functions corresponding to the objects.

Result of Program 4.11

```
Printing from class Book
number =1
price =240
Printing from class SubClass
number =1
price =240
pages =455
printing from class SubSubClass
number =1
price =240
pages =455
discount =0.25
```

Let us look at one more example to understand the same concept without any doubt.

Program 4.12

```
//Multi-level inheritance and object reference
#include<iostream>
using namespace std;
class Book {
    protected:
        int number;
```

```
         double price;
      public:
      void display(){
         cout<<"\n Printing from class Book";
         cout<<"\n number ="<<number;
         cout<<"\n price ="<<price;
      }
   };
   class SubClass: public Book {
   protected:
      int pages;
   };
   class SubSubClass : public SubClass {
      protected:
      double discount;
      public:
         void getdata(int c, double d, int e, double f){
            number=c;
            price=d;
            pages=e;
            discount=f;
         }
      void display(){
         cout<<"\n Printing from class SubSubClass";
         cout<<"\n number ="<<number;
         cout<<"\n price ="<<price;
         cout<<"\n pages ="<<pages;
         cout<<"\n discount ="<<discount;
      }
   };
   int main()
   {
      SubSubClass javab;
      javab.getdata(1,240.0, 455, 0.3);
      Book cb=javab;
      cb.display();
      javab.display();
   }
```

The above program does not have any constructors. It implements function hiding. The class Book has two data members and a function display. The class `SubClass` has one data member, but no function at all. The `SubSubClass` has one data member and a function getdata to receive values for all the four data members, one of its own, one from its base class and two from the base class Book. It has also another function display to display the values of all the four data members. Thus, in this example, there are two functions called display. The function at the lower level class hides the one at the base class.

In the main function, an object javab is declared of type `SubSubClass`. Then it calls the function getdata to assign values to the data members. Then we create an object cb of type Book and assign to it the object javab of the `SubSubClass`. We then call the function display for object cb and finally call function display for object javab. The result of the program is given below:

Result of Program 4.12

```
Printing from class Book
number =1
price =240
Printing from class SubSubClass
number =1
price =240
pages =455
discount =0.3
```

Function hiding allows multiple functions with one interface i.e. one function name. The appropriate functions are called depending on the object, which calls them. Here, the determining factor is the object type calling the function

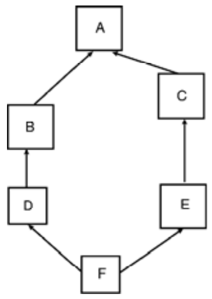
To summarize, when there are two identical (with identical prototype only) functions in the base class and the derived class, the function in the derived class hides the function in the base class. When a derived class object calls the function, the function in the derived class will be used. But, if the base class object calls the function, it will use the function in the base class only. This was amply made clear by the programs seen thus far. The hidden function could be invoked through the base class object. Thus when the functions are overloaded i.e. when these are multiple functions with the same name, the appropriate function is identified through the object invoking it.

In some cases abstract classes are designed which exist only in order to have specialized derived classes, derived from them. Such abstract classes have functions that do not perform any useful operations and are meant to be overridden by specific implementations of the functions in derived classes. Thus, the abstract base class defines a common interface which all the derived classes inherit.

4.3 VIRTUAL CLASS

Virtual base class

Inheritance of any sensible combination is possible. Some times we may have a derived class with two base classes, which in turn may derive from two other base classes. This chain may lead to a situation as given in Figure below:



NOTES

Here there is a possibility that the class F derives the property of class A twice in the following routes:

- A-B-D-F
- A-C-E-F

This can lead to ambiguity. To avoid the overlapping and ensure that the properties of A is derived only once by F, we can declare A as a virtual base class. Actually we don't attach the keyword virtual to the base class. We attach the keyword when we use the base class to extend its multiple derived classes as given below:

```
class B: public virtual A{
};
```

The following example illustrates use of virtual base class.

Program 4.13

```
// To demonstrate virtual classes
#include<iostream>
using namespace std;
class salary {
    protected:
        double basic;
        double perks;
    public:
        salary(double a, double b){
            basic =a;
            perks=b;
        }
    ~salary(){}
};
class total_income :virtual public salary {
    protected:
        double house;
    public:
        total_income(double x, double y, double z):
        salary(x,y){
            house=z;
        }
    ~total_income(){}
};
class deductions :virtual public salary{
    protected:
        double income_tax;
        double other_tax;
    public:
        deductions(double  r,  double  s,  double  p,  double
q):salary(r, s){
            income_tax=p;
```

NOTES

```
            other_tax=q;
         }
      ~deductions(){} //destructor
   };
   class pay: public total_income, public deductions {
      public:
         pay(double a, double b, double c, double d, double
   e): salary(a,b),                  total_income(a, b, c),
   deductions(a, b, d,e){ }
         ~pay(){}
      void display(){
         cout<<"Printing Salary statement\n";
         cout<<"Basic Pay :"<< basic<<"\n";
         cout<<"Perks    :"<< perks<<"\n";
         cout<<"Income from house :"<< house<<"\n";
         cout<<"Income Tax:"<< income_tax<<"\n";
         cout<<"Other Tax:"<< other_tax<<"\n";
         cout<<"Net Pay :"
         <<(basic+perks+house-income_tax-other_tax);
      }
   };
   int main() {
      pay John(20000.0, 10500.0, 7500.0, 12000.0, 150.0);
      John.display();
   }
```

The classes total_income and deductions both derive from the virtual base class salary. The keyword is prefixed to the class name. Note the constructor definition of the class pay reproduced below:

```
      pay(double a, double b, double c, double d, double e):
      salary(a,b), total_income(a, b, c), deductions(a, b, d,e){
   }
```

Normally we would have given the parameter list of the classes total_income and deductions from which the class pay is derived, after the colon. However in this example we have to also specify the parameter list of the virtual class first. This will indicate that the class should inherit these data members only once. The result of the program is given below:

Result of Program 4.13

```
   Printing Salary statement
   Basic Pay :20000
   perks :10500
   Income from house :7500
   Income Tax:12000
   Other Tax:150
   Net Pay :25850
```

4.4 FRIEND AND STATIC FUNCTIONS

Static class members

C++ lets you declare some data members of a class as *static*. Static members share the same memory for all instances of the class (i.e. for all declared variables of the class type). We can say that static members belong to the class as a whole, and not to its specific instances.

In addition to data members, member functions can be declared static, too. Unlike regular member functions, these functions can be called without attributing them to a specific class instance. For that reason, static functions can access and manipulate only static data members.

You can declare a member static by using the keyword static in its declaration. Both private and public members may be declared static. The following class for a mock-up soda vending machine declares a static data member, price, and a static member function SetPrice(...):

```
class SODA {    // Soda vending machine
  private:
    static int price;
    int dayTotal;
  public:
    static void SetPrice(int newprice) {price = newprice;}
    SODA() {dayTotal = 0;}
    void SellSoda() {dayTotal += price;}
    int GetDayTotal() {return dayTotal;}
    ...
};
```

This example demonstrates one of the more obvious uses of static members: defining a constant or a variable that has to be shared by all instances of the class, in this example the price of a can of soda.

A static data member *can not* be initialized with its declaration or in the constructor, because the constructor is called for every instance of the class. Instead, it is initialized separately in the source file, as follows:

```
#include "soda.h"
...
int SODA::price = 50;
...
```

This is how this class works in a test program:

```
#include <iostream.h>
#include <iomanip.h>
#include "soda.h"
void main()
{
    SODA machine1, machine2;
    SODA::SetPrice(60); // Sets the price for all machines
(cents)
    int can;
    for (can = 0;   can < 100;  can++)
        machine1.SellSoda();
    for (can = 0;   can < 50;  can++)
        machine2.SellSoda();
    float dollars =
        .01 * (float)machine1.GetDayTotal() +
        .01 * (float)machine2.GetDayTotal();
    cout << setprecision(2)
         << setiosflags(ios::fixed | ios::showpoint)
         << "Day's sales: $" << dollars << endl;
}
```

A public static member can be accessed through any instance of the class. If price were a public member, for example, we could write:

```
    ...
void main()
{
    SODA machine1, machine2;
    machine1.price = 60;
    ...
}
```

This usage would be misleading, though, because it would suggest that price was set only for machine1 and would obscure the fact that the same price has been set for all 'machines.' We would do better to write:

```
      ...
    SODA::price = 60;
    ...
```

The same is true for function members. We can call a static member function without attributing it to a specific class object by preceding its name with the class scope prefix:

```
       ...
    SODA::SetPrice(60);
    ...
```

A less obvious use of static members is for allocating some temporary shared work space, for example a large array, especially for use in a recursive member function.

Static members may also be used for more esoteric tasks related to the management of the class as a whole. Suppose we want to count all currently existing instances of a class. This can be done by using a static counter:

```
class SOMECLASS {
  private:
    static int nObjects;
    ...
    public:
      static int NumberOfObjects() {return nObjects;}
      SOMECLASS() {      // Constructor
          nObjects++;
          ...
      }
      ~SOMECLASS() {     // Destructor
          ...
          nObjects--;
      }
      ...
  };
  int SOMECLASS::nObjects = 0;
  ...
  #include <iostream.h>
  void main()
  {
      SOMECLASS object1, object2;
      cout << SOMECLASS::NumberOfObjects() << " objects\n";
                                 // Output: "2 objects"
      if (SOMECLASS::NumberOfObjects() == 2) {
          SOMECLASS object3;
        cout << SOMECLASS::NumberOfObjects() << " objects\n";
                                     // Output: "3 objects"
      }
      cout << SOMECLASS::NumberOfObjects() << " objects\n";
                                 // Output: "2 objects"
  }
```

Friend classes and functions

We have already seen that the keyword friend is used to give a non-member function or an operator access to all members (including the private members) of a class. For example:

```
  class VECTOR {
    friend ostream &operator<< (ostream &outp, const VECTOR
  &vector);
    friend istream &operator>> (istream &inp, VECTOR &vector);
    private:
      ...
  };
```

These declarations give operator<< and operator>> access to all private members of the class VECTOR.

A whole class may be declared a "friend," which gives all its function members access to all the members of the given class. For example:

```
  class FIRSTCLASS {
```

NOTES

```
    friend class SECONDCLASS;
    private:
      int x;
      ...
  };
  class SECONDCLASS {
    private:
      int y;
    public:
      void Copy(FIRSTCLASS &object) {y = object.x;}
      ...
  };
```

The member function Copy in SECONDCLASS has access to object.x because SECONDCLASS is declared a friend in FIRSTCLASS's definition.

Friend declarations may appear anywhere within the class definition. A friendship is not necessarily symmetrical: as a class designer, you determine your class's friends; users of your class cannot declare their classes friends to your class and gain access to its private members.

Friend classes and functions should be used judiciously because they weaken encapsulation.

4.5 MULTIPLE INHERITANCE

Multiple inheritance

We now continue the discussions about inheritance we had in section 4.2. Inheritance enables reuse of classes already developed and tested, with necessary modifications. For instance, if a class exists for *stack* and if another developer wants to modify it slightly for his application, he can write a derived class for the existing *stack* class. This will save lot of time and effort. Any developer can first search for available class libraries and then customize them for his requirement. In the previous chapter, we discussed about single inheritance and multi-level inheritance. In both these methods, there is only one base class for a derived class. Original C++ supported only this feature. But the latest versions support multiple inheritances, which means that a derived class can inherit from more than one base class as illustrated below:

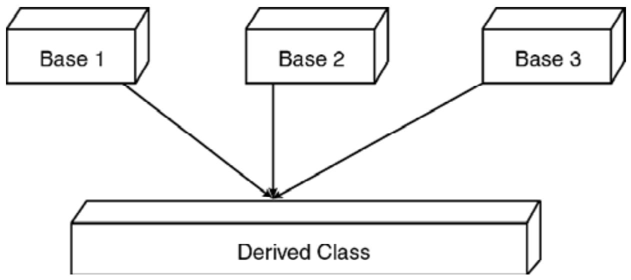


Figure 4.7 *Multiple inheritance*

In the above figure, we have three base classes. The derived class inherits properties from all the three base classes. It has three parents. A derived class inheriting from more than one base class is called multiple inheritance.

NOTES

Now let us implement multiple inheritances through a simple program. Here, *Book* and *BookPrice* are two base classes with their own protected data members. We have *BookData* as a derived class and it derives from both the base classes as shown in Figure 4.8.

In such a case declaring a derived class is a logical extension of what we have already discussed. It will be declared as give below:

```
Class BookData : public Book, public BookPrice {…….
```

We have listed both the base classes derived by the class one after another. When there was only one base class, we had listed only one. Therefore, depending upon the number of base classes and the type of derivation such as public or private, suitable declarations can be made.

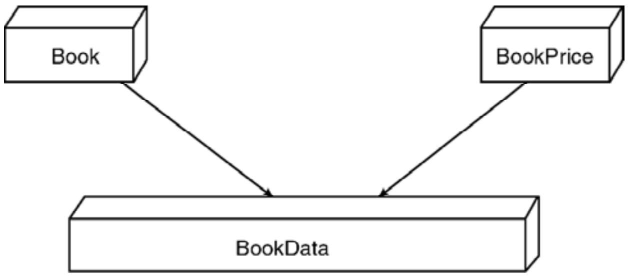


Figure 4.8 *Multiple inheritance*

We shall make use of the existing class *Book* which is in the file "*Book.h*". We develop a new class *BookPrice* and save it in file "BookPrice.h". The source code of file BookPrice.h is given below:

```
//BookPrice.h
class BookPrice {
    protected:
       double price;
    public:
       BookPrice(double d){
          price = d;
    }
};
```

The above class has only one data member namely *price*.

This source code is available whenever required. We will include the same file, which has been thoroughly tested. Now let us develop the program for implementing the derived class *BookData*, which is given below:

Program 4.14

```
//#include"Book.h"
```

```
#include"BookPrice.h"
#include<iostream>
using namespace std;
class BookData:public Book, public BookPrice {
    int pages;
    double discount;
    public:
       BookData(int p, string q, string r, double d,
       int e, double f):Book(p, q,r), BookPrice(d){
          pages = e;
          discount =f;
       }
    void display(){
       cout<<"\n number ="<<number;
       cout<<"\n Name of the book "<<bookName;
       cout<<"\n Name of the author "<<authorName;
       cout<<"\n price  ="<<price;
       cout<<"\n pages  ="<<pages;
       cout<<"\n maximum discount  ="<<discount;
      }
};
int main() {
    BookData javab(1,"C++", "Subburaj", 243.0, 501, 0.2);
    javab.display();
}
```

The first base class has three data members and the second, one data member. The derived class has its own data members. The derived class thus derives from the two base classes. It has in addition one member function for displaying values. In the main function an object *javab* of derived class is created with values to all data members. Since the protected members of the two base classes are visible in the derived class, there is no difficulty in assigning the values to them through the member function of the derived class. Then we call function *display* of the derived class and the result of the program is given below:

Result of Program 4.14

```
number =1
Name of the book C++
Name of the author Subburaj
price  =243
pages  =501
maximum discount  =0.2
```

What we have seen now is that a class can derive from more than one base class and with the same logic from any number of base classes. The program makes evident the visibility of the *protected* data members of the base classes in the derived class.

Constructors in mulitple inheritance

The class *Book*, one of the base classes has a constructor. Similarly the second class BookPrice has also its own constructor. We must notice how the constructor of the derived class is formulated when there is more than one base class. It is reproduced below for ready reference.

```
public:
BookData(int p, string q, string r, double d,
int e, double f):Book(p, q,r), BookPrice(d){
    pages = e;
    discount =f;
}
```

The constructor lists all the six data members of the derived class. It is followed by a colon, then the first base class name followed by its parameter list i.e. Book (p,q,r) is given. After that the parameter list of the second base class namely BookPrice (d) is given. These two are separated by a comma. After that we have a brace, which will assign values to new data members of the derived class. If there are no additional data members in the derived class, empty braces are to be appended. In multiple inheritance instead of one base class there are multiple base classes. There is no other difference.

4.6 POLYMORPHISM

A virtual function is a special member function. Virtual functions allow programmers to declare functions in a base class and redefine the same with the same name in its derived classes. Prefix of `virtual` keyword indicates that the function acts as an interface. The additional feature of virtual functions is that the compiler and linker will guarantee that the correct function is associated with the objects of each of the derived class at run time. Although it may appear to be similar, there are important differences between virtual function and function hiding.

Early binding

As we know, function overloading is a polymorphic behaviour where there is one name for the function, but multiple forms. The multiple forms arise out of varying signatures or argument types received by the functions as well as the code. When an overloaded function is called, the compiler matches the arguments passed with the signature of the various functions with the same name. Once the match is found, it associates that code of the function with the call. Therefore, there is no confusion and appropriate function is linked at compile time itself. This is called early binding or static binding. There is no need for any special action at the runtime. Operator overloading also falls under the same category as function overloading.

Function hiding is a special case of function overloading. We gave a different name (hiding) because the signatures of the overloaded functions are same here unlike overloading. In function hiding also we invoke functions with the same name in different classes. But the objects of the base class and derived classes call the functions. The compiler knows which appropriate function is to be associated. For instance, if the base class object calls a function `display()`, the function in the base class will be attached to the call by the compiler. If a derived class object calls the function `display()`, the function called `display` in that particular derived class will be referred to. There is no ambiguity to the compiler. The right function to be associated with the function call is decided based on the object calling it. Thus, function hiding results in early binding. i.e. at the time of compilation.

Late binding

When we implement virtual functions, there will be functions of the same name and signature at the base class and derived classes. Here, the functions in the base class as well as every derived class will be invoked or called through the same base class pointer. Hence compiler may be confused as to which function to refer to for each call and hence passes the buck to the run-time system. Although, we may call the function with base class pointer, the actual function (i.e. function in the base class or in the derived classes) to be executed will be determined at run-time. The object to which the base pointer points to, which base class or derived class object, will be known only at run-time. Since the function to be executed is determined only at run-time this is called late binding, i.e. binding of the function to the called object.

The ambiguity arising out of the same function name was overcome in early binding because of the variations in the signatures or the invoking objects. But, if all the signatures are the same and pointer to the invoking object is same, what happens? The compiler will get puzzled. So it does not attach the function to any call and leaves the job to the run-time system to decide the function to be called. So calling of the appropriate function takes place at run-time and hence it is known as run-time polymorphism. To implement run-time polymorphism, we need virtual functions. To achieve this, the functions of the same name are implemented in the derived classes. The function in the base class is declared as the virtual function by prefixing the word “virtual”. Look at the program below:

Program 4.15

```
//To demonstrate virtual functions
#include<iostream>
using namespace std;
class grade{
    protected:
        int maths_mark;
    public:
        grade(int mm){
            maths_mark=mm;
        }
    virtual void display(){
        cout<<"displaying from base class \n";
        cout<<" maths_marks :"<<maths_mark<<"\n";
    }
};
class der :public grade{
      int physics_mark;
      public:
         der(int a, int pp):grade(a){
            physics_mark=pp;
         }
      void display(){
         cout<<"displaying from derived class \n";
         cout<<"\n Maths marks :"<<maths_mark<<"\n";
         cout<<"\n Physics marks:"<<physics_mark<<"\n";
      }
   };
   int main(){
      grade M(100);
      der P(100, 98);
      grade *mptr;
      mptr=&M;
      mptr->display();
      mptr=&P;
      mptr->display();
   }
```

Notice that the function implementing run-time polymorphism is `display()`. The keyword `virtual` is prefixed to the function in the base class. The same function in the derived class does not have the prefix. The virtual function can be called only through pointers to objects. Look at the main function. We had created one object of type `grade` belonging to the base class and another object `der` of the derived class using constructors. Then, `mptr` a pointer to object of the base class is declared by the following statements:

```
      grade *mptr;
      mptr=&M;
```

Now `mptr` points to base class object `M`. When we call display function, with `mptr`, the function in the base class will be executed. In the next statement, we assign the same pointer, that is the base class pointer to the derived class object P. When we call `display` by using the same pointer, the program will now call the *display* function in the derived class. We will see the secret of virtual function shortly.

To recapitulate, we have to declare the overloaded function in the base class as `virtual`. Then, we have to declare a pointer to the base class object. Only this pointer can be used with the overloaded member functions. The member functions may either belong to the base class or to any one of the derived classes. When the base class pointer refers to the base class object, the function in the base class will be called. When it refers to any of the derived class objects, the corresponding function will be called. The base pointer may appear to be calling a function in the base class, but actually it may be calling a function in the derived class.

The virtual function needs usage of pointers to base class object. It is essential that a virtual function be defined in the base class. For implementing run-time polymorphism, it is essential that there are more than one function with identical name and identical signature. For instance, the function `display` in the above program. If the signatures are different, then obviously, it will be treated as function overloading. The pointer to base class object can refer to derived class objects but vice versa is not possible. As seen in the above example we are using a single pointer variable but it has to be pointer variable of the base class. The run-time system looks for the keyword `virtual` to implement run-time polymorphism. The result of the program confirms that by assigning the base pointer to different objects, we are able to invoke the corresponding functions.

Result of Program 4.15

```
displaying from base class
maths_marks :100
displaying from derived class
Maths marks :100
Physics marks:98
```

It is important that the argument types in the functions in derived class are same as the argument types in the virtual function in the base class. Minor changes are allowed for the return data types.

Upcasting

We have been assigning object of the derived class to the base pointer. Effectively, this means that we are assigning the address of the object of the derived class to the address of the object of the base class. In this case, the base pointer treats the address of the derived class as that of the base class. Treating the address of the object of the derived class as the address of the base class is known as upcasting. In normal circumstances, the compiler would have flagged an error. Upcasting is used in virtual functions. The compiler infers from the keyword `virtual` that it should not try to carryout early binding, but postpone it to run-time.

Virtual table

The secret of run-time polymorphism lies in the virtual function table, which is also called `vtbl`. When a compiler recognizes the virtual function from the `virtual` prefix, it builds the `vtbl`. The `vtbl` is a table of pointers to functions. The `Vtbl` is not visible to the programmer. For instance, for the above program a table as shown in figure 4.9 will be creaed.

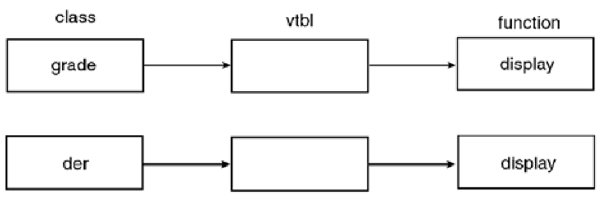


Figure 4.9 *Representation of virtual table*

We can see from the above that each class has its own `vtbl`. The `vtbl` consists of pointers to functions or addresses of the functions. Thus, a link between a class and the address of the function of the class is established through `vtbl`.

Virtual pointer

Whenever we create an object of a class, another pointer called virtual pointer or `vptr` is created. Each object created has its unique `vptr` associated with it, which points to the vtbl of the corresponding class. The vptr is also not visible to the programmer. From the vtbl the run-time system can find out the address of the function of the class. Thus, a link between the specific object and the address of the function is clearly established. The vtbl is like a look-up table consisting of addresses of the functions.

NOTES

When we implement the virtual functions, we first assign one of the derived class objects to point to the base class pointer. When this is carried out, the vptr of the derived object will in turn be linked to the base class pointer. Next, when we call the function with the base class pointer, the exact function can be linked through the respective vptr and vtbl. A simple representation of this is given below in Figure 4.10.

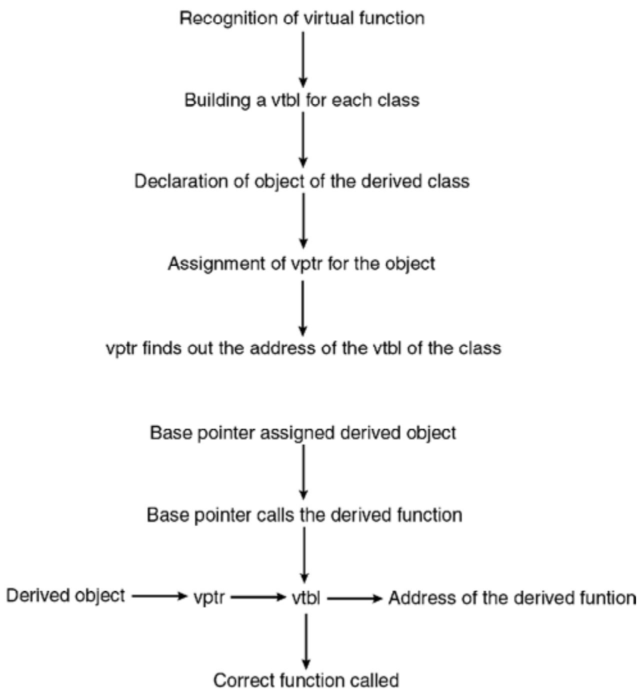

Recognition of virtual function
Building a vtbl for each class
Declaration of object of the derived class
Assignment of vptr for the object
vptr finds out the address of the vtbl of the class
Base pointer assigned derived object
Base pointer calls the derived function
Derived object
vptr
vtbl
Address of the derived funtion
Correct function called

***Figure 4.10**. Flow of events in late binding*

The implementation of vtbl as above, calls for additional space for a pointer of the base class (which contains the virtual function) and one vtbl for each class. Thus, we have two numbers of vtbl in the above program. This mechanism will help the run-time system to connect the object pointed to by the base class pointer and the appropriate function. Thus, the prefix of virtual to the function in the base class is an important qualifier for implementing virtual function, since only on seeing this keyword the vtbl will be established by the system.

Now we look at another variation to virtual function known as pure virtual function.

Pure virtual function

Generally, virtual functions are not used for any purpose other than providing declaration. In the above example, we also used virtual function to carry out some tasks. When we define virtual functions only for providing an interface, we can define it as given below:

For instance,

```
 virtual void display()=0;
```

In this case, the derived classes must build and expand the empty function as defined above. Virtual functions, which do nothing are called pure virtual functions. An example is given below:

Program 4.16

```
//To demonstrate Pure virtual functions
#include<iostream>
using namespace std;
class school{
    public:
       school(){}
       virtual void display()=0;
};
class grade:public school{
    protected:
       int maths_mark;
       public:
          grade(int mm){
             maths_mark=mm;
          }
    void display(){
       cout<<"now we will display from grade class \n";
       cout<<" maths_marks :"<<maths_mark<<"\n";
    }
};
class der :public school{
    int physics_mark;
    public:
       der(int pp) {
          physics_mark=pp;
       }
    void display(){
       cout<<"now we will display from der class \n";
       cout<<"\n Physics marks:"<<physics_mark<<"\n";
    }
};
int main(){
    grade G( 100);
    der P(98);
    school *mptr;
    mptr=&G;//pointer assigned to object of grade
    mptr->display();
    mptr=&P; //pointer assigned to der object
    mptr->display();
}
```

In the above example, the pure virtual function is defined in the base class. The base class is also empty. Therefore, it is also called an abstract class. The coding has been carried out only in the derived classes. Otherwise, the program is similar to the previous one. The result of the program is given below:

Result of Program 4.16

```
now we will display from grade class
maths_marks :100
now we will display from der class
Physics marks:98
```

Thus a pure virtual function does not carry out any activity for the base class. In such cases, it is mandatory that the derived classes redeclare and give code for the function. If we do not define virtual function in the derived class, it also becomes abstract. Since the function is empty, it is only an interface provider.

NOTES

Abstract class

Placing a pure virtual function inside a class makes it abstract. Abstract classes cannot be used for creating objects. In the above Example, class school is a base class. It is used only to extend the properties of other derived classes. Therefore the class school can be called an abstract class. In fact, since it has a pure virtual function, it cannot be used to create objects. If we try to create an object, the compiler will flag an error. Therefore the objective of abstract base class and virtual function is to provide a framework for the derived classes to expand on them. In this case the abstract base class provides for declaring a pointer to base class which is essential for late binding or what is known as run-time polymorphism.

Nested class – container class

Similar to inheritance, there is one more method of deriving properties of one class in another. This is achieved through nesting of classes or containership. Containership is implemented in the Example below:

Program 4.17

```
// To demonstrate nested classes
#include<iostream>
using namespace std;
class salary {
   public:
      double basic;
      double perks;
   public:
      salary(double a, double b){
         basic =a;
         perks =b;
   }
};
class deductions {
   public:
      double income_tax;
      double other_tax;
   public:
```

NOTES

```
      deductions(double c, double d){
         income_tax=c;
         other_tax=d;
      }
};
class pay {
   double house;
   salary ram; //object
   deductions lak; //object
   public:
      pay(double r, double s, double a, double b, double
c):ram(r, s), lak(a,b){
      house=c;
   }
   ~pay(){} //destructor
   void display(){
      cout<<"Printing Salary statement\n";
      cout<<"Basic Pay :"<< ram.basic<<"\n";
      cout<<"Perks    :"<< ram.perks<<"\n";
      cout<<"Income from house :"<< house<<"\n";
      cout<<"Income Tax:"<< lak.income_tax<<"\n";
      cout<<"Other Tax:"<< lak.other_tax<<"\n";
      cout<<"Net Pay :"    <<(ram.basic+ram.perks+house-
lak.income_tax-lak.other_tax);
   }
};
int main()
{
   pay sita(20000.0, 10500.0, 12000.0, 150.0, 7500.0);
   sita.display();
}
```

In the above Example, the classes salary and deductions are simple base classes with their own constructors. Look at the class pay which is different. It implements inheritance in a different and may be in a difficult way. After declaration of its own data member, an object ram of class salary and another object lak of deductions are declared. This is new. Only through these declarations, we are going to inherit the properties of the respective classes. Now look at the constructor of the class pay. Here we list the arguments of the base classes along with the name of the respective objects and not the classes. This is an important difference. The constructors are called in the same order in which the objects are declared in the class. Therefore the last parameter belongs to the class pay. Look at the member function display. There the variables are called using the dot operator with the object name, for instance, ram.basic. Had it been inherited in the normal way there is no need to associate the object name. To summarize, we can also inherit the properties of other classes through their objects. The function `display()` will indicate that the object of the class pay, namely sita, is a collection of objects of other classes. The class pay contains objects of other two classes. This is called containership where one class contains objects of other classes, thereby the object (of the class) is a collection of other objects. At the point recall the definition of vector in chapter 4 which is a container defined in the standard library. Nesting of classes is achieved in a class through declaring the objects of other classes. The result of the program confirms the concept.

NOTES

Result of Program 4.17

```
Printing Salary statement
Basic Pay :20000
Perks :10500
Income from house :7500
Income Tax:12000
Other Tax:150
Net Pay :25850
```

Difference between inheritance and container

When a class B inherits from class A, then B is a kind of A. For instance, if a class car is derived from class vehicle then *car* is a kind of vehicle. This is also called a kind of relationship.

On the contrary, car contains a wheel. If they are implemented as classes, then an object of wheel called in any name, say *W* will be a member of the class *car*. The class *car* contains object W of class *wheel*. Thus class *car has* a relationship with class *wheel.* This is called *has* relationship. This is the essential difference between inheritance and containership. In any case both provide relationship between classes.

Run-time type identification (RTTI)

The virtual functions provide a common interface to all the functions in the base class as well as its derived classes. In such cases, it may be difficult to understand the type of object, which has been invoked. We used a simple methodology in the first two programs in this chapter, where we inserted a print statement in the functions in various classes to identify the class from which the function was called. From this we could find out the object type. There is no need for such additional programming since C++ provides two keywords through which we can identify the objects at run-time.

The run-time type identification (RTTI) can be carried out using the following keywords.

- typeid
- dynamic_cast

A program to find out the type of object at run-time using *typeid* is given below. These mechanisms will work with virtual functions.

Program 4.18

```
//To demonstrate typeid virtual functions
#include<iostream>
using namespace std;
class Exam{
    public:
    virtual void display(){
```

NOTES

```
      }
  };
  class grade:public Exam{
      protected:
          int maths_mark;
      public:
          grade(int mm){
              maths_mark=mm;
          }
      void display(){
          cout<<" maths_marks :"<<maths_mark<<"\n";
      }
  };
  class der :public grade{
      int physics_mark;
      public:
          der(int a, int pp):grade(a){
              physics_mark=pp;
          }
      void display(){
          cout<<"\n Maths marks :"<<maths_mark<<"\n";
          cout<<"\n Physics marks:"<<physics_mark<<"\n";
      }
  };
  int main()
  {
      Exam *E1;
      cout<<typeid(E1).name()<<" class \n";
      grade M(100);
      der P(100, 98);
      E1=&M;
      cout<<typeid(*E1).name()<<" class \n";
      E1->display();
      E1=&P;
      cout<<typeid(*E1).name()<<" class \n";
      E1->display();
  }
```

In the above program, a virtual function `display` has been declared in the base class namely `Exam`. There are two derived classes namely `grade` and `der`. The virtual function has been defined in the derived classes. In the main function, we assign pointer `E1` to the base class `Exam`. In the next statement, we find out the name of the class through the typeid function. Note the syntax as given below for finding out the name of the class

```
      typeid(*E1).name();
```

The base pointer is the argument to the `typeid` function. It calls the function *name* of the standard library. This will bring out the name of the class to which the identifier `E1` belongs.

Next, we assign the address of the object `M` of the derived class grade to the base pointer. Then we find out its name. Similarly we find out the name of the class to which object `P` of the derived class `der` belongs to. The result of the program is given below:

Result of Program 4.18

```
P4Exam class
5grade class
maths_marks :100
3der class
Maths marks :100
Physics marks:98
```

Note that the name given for the class can be extracted at run-time using `typeid`. The name of the class is preceded by unique code such as P4,5 *& 3*. If we observe carefully, we will find the name of the class `Exam` consists of 4 characters. Similarly, the name of the class `grade` consists of 5 characters and `der` 3 characters. We will now modify the program to confirm the following:

- Whether this will work with a pure virtual function.
- Can we also find out the id of built-in data types?

To confirm this, the above program has been converted and given below:

Program 4.19

```
//To demonstrate typeid with pure virtual functions
#include<iostream>
using namespace std;
class Exam{
    public:
    virtual void display()=0;
};
class grade:public Exam{
    protected:
        int maths_mark;
    public:
        grade(int mm){
            maths_mark=mm;
        }
    void display(){
        cout<<" maths_marks :"<<maths_mark<<"\n";
    }
};
class der :public grade{
    int physics_mark;
    public:
        der(int a, int pp):grade(a){
```

NOTES

```
            physics_mark=pp;
        }
    void display(){
        cout<<"\n Maths marks :"<<maths_mark<<"\n"
        cout<<"\n Physics marks:"<<physics_mark<<"\
    }
};
int main()
{
    Exam *E1;
    cout<<typeid(E1).name()<<" class \n";
    grade M(100);
    der P(100, 98);
    E1=&M;
    cout<<typeid(*E1).name()<<" class \n";
    E1->display();
    E1=&P;
    cout<<typeid(*E1).name()<<" class \n";
    E1->display();
    cout<<typeid("A new thing").name()<<endl;
    cout<<typeid(15.0).name()<<endl;
}
```

Result of Program 4.19

```
P4Exam class
5grade class
maths_marks :100
3der class
Maths marks :100
Physics marks:98
A12_c
d
```

In the above program, the function `display` has been converted into a pure virtual function in the base class `Exam`. Look at the main function, there is no difference in this program except for the last two lines. In the last two lines, we try to find out the type name of a string and a double. The string consists of 11 characters. In C++ we have to earmark a space for holding *NULL* as a terminator. Thus, it is an array of 12 characters. The type is identified as `A12_c` by `typeid`. Similarly, the number 15.0 is identified as `d` indicating double.

Thus, using `typeid` we can find out the type of the object at run-time.

Dynamic cast operator

We can also use `dynamic_cast` to check the name of the class of the object at run-time. Usually, the `dynamic_cast` operator is used to compare two objects of the same class. A program is implemented to compare the type of two objects and given below:

Program 4.20

```
//To demonstrate dynamic cast
#include<iostream>
using namespace std;
class Exam{
    public:
    virtual void display()=0;
};
class grade:public Exam{
    protected:
        int maths_mark;
    public:
        void display(){
            cout<<" maths_marks :"<<maths_mark<<"\n";
        }
};
int main()
{
    Exam *E1;
    grade M1, *M2;
    E1=&M1;
    if(M2=dynamic_cast<grade*>(E1))
        cout<<"M2 is of type class M1 \n";
    else
        cout<<"Objects are not of the same class\n";
}
```

NOTES

In the above program, two objects `M1` and `*M2` of the derived class have been declared. The address of the object `M1` is assigned to the address of the base class. We are checking whether the objects `M2` and `M1` belong to the same class or not by the following statement:

```
 if(M2=dynamic_cast<grade*>(E1)) cout<<"M2 is of type class
M1 \n";
```

The `dynamic_cast <grade*>` refers to the object of class grade. We are checking in the above statement whether `M2` is an object of class grade. If so, we print that `M2` is of type `M1`. We can also word it suitably. The operator returns a non-zero integer, if `M2` is of type `grade` and zero if it is different. Therefore, the expression will become true when `M2` is of type grade. Note that the expression includes the name of the object of the base class. We also note that we can compare only a pointer to the object in the expression. The result of the program is given below:

Result of Program 4.20

```
M2 is of type class M1
```

The run-time type information shall be used only when it is essential.

Type casting

We can convert objects of one type to other. When we convert or cast a narrow type to a wider type, there will be no problem. However, when we convert a wider type to a

NOTES

narrower type, there could be problems. Explicit type conversions shall be avoided as far as possible. C++ provides the following 4 keywords for type casting.

- `const_cast`
- `dynamic_cast`
- `reinterpret_cast`
- `static_cast`

We have already discussed about `dynamic_cast`, which we used for finding the type of the object. We will discuss about the other three types in the following.

`const_cast`

As we know, the compiler will flag an error if we try to change the value of an identifier declared as a constant in the program. However, we can change the status using the keyword `const_cast`. The use of this keyword is to convert a constant into a variable. We can achieve this as given below:

```
const double dvar = 25.0 ;
double *pvar ;
pvar = const_cast <double *> (&dvar);
```

In the above statement, we have withdrawn the constant status given to `dvar`. What we have done is that we have assigned the address of the constant, to a pointer to a non-constant of the same type by using `const_cast` in the above expression.

We can extend this principle to change the constant status of the argument in a function.

`reinterpret_cast`

We can use `reinterpret_cast` when we want to assign the pointer to one type of object to the pointer to another type of object. The `reinterpret_cast` also permits conversion of pointer type to a non-pointer type and vice versa. When both the types are same, then it will not cause any error. However, this can cause problems if they are not of the same type. In such cases, the compiler ensures that the new type will have the same pattern of bits as the original type. A safe `reinterpret_cast` is given below:

```
double dvar = 25.0 ;
double *dptr = reinterpret_cast <double *> (dvar) ;
```

`static_cast`

There are number of types of `static_cast` as given below:

Conversion without cast

Examples

```
long double ldvar;
double dvar;
ldvar = dvar;
```

This does not cause any error because we are assigning a narrow type to a wider type.

Narrowing conversions

With the same definition as above, the following is an example of narrowing conversion.

```
dvar = ldvar;
```

This may lead to loss of data. The above statement can also be written as follows:

```
dvar = static_cast <double> (ldvar);
```

Explicit conversion

We use the keyword *static_cast* when we attempt to do explicit conversions. Some examples are given below:

```
ldvar = static_cast <double> dvar;
long double var2;
ldvar = static_cast <long double> (var2);
```

Modifying constant objects – mutable

A constant object can be declared by the prefix of keyword const while declaring the object. For instance,

```
const Account Lakshmi(003, 200075.00);
```

The above declares the object `Lakshmi` as constant object of type `Account`. Once we declare an object as constant, then the compiler would not allow us to modify its data members. If we try to change the `balance` then the compiler will flag an error. But there is a way out. We can declare the modifiable data member of the as `mutable`. The keyword `mutable` is prefixed to the declaration of the data member in the class. For instance, we can declare the balance of an account holder as follows:

```
mutable double balance ;
```

This is a very practical example, because in a bank account, while all other attributes such as Name, Account Number etc. will not change, the balance will change. Therefore, we can declare `balance` as a mutable type while declaring the class. The program below illustrates this concept:

Program 4.21

```
//To demonstrate mutable
#include<iostream>
using namespace std;
class Account {
    int number;
    mutable double balance;
    public:
       Account (int x, double y) {
          number=x;
          balance=y;
       }
    void display()const {
       cout<<"Account number ="<<number<<"\t";
       cout<<"Balance ="<<balance<<"\n";
    }
    void crediting(double deposit) const{
       balance=balance + deposit;
    }
};
```

```
int main(){
    const Account Lakshmi(003, 200075.00);
    Lakshmi.display();
    Lakshmi.crediting(100000.00);
    Lakshmi.display();
}
```

In the above program, if we declare the object of `Account` to be constant and declare the data members and member functions in the normal way, it will not compile. The reason for this is that although we had declared the object `Lakshmi` as constant, we are going to deposit some money, which will modify the balance. Now let us prefix mutable to the `balance` to make it modifiable. Even then it will not compile, because the data member `balance` is handled in both the member functions namely display and crediting. Therefore, such functions where the data members are going to be handled have to be suffixed with the keyword `const`.

To summarize, to modify a data member of constant object we have to do the following:

- Prefix keyword `mutable` to the data member, which is likely to be modified.
- Suffix keyword `const` to the member functions, which handle the data member.

With these two modifications, we can have a constant object whose members can be modified on a selective basis. The result of the program is given below:

Result of Program 4.21

```
Account number =3 Balance =200075
Account number =3 Balance =300075
```

Preventing use of constructors for data conversion – explicit

When we were discussing about operator overloading, we have seen that constructors can be used for data conversion. We were using a single argument constructor to convert it into an object as given below:

```
class length {
    int foot;
    int inch;
    public:
       length (int x) {
          foot=x/12;
          inch=x%12;
       }
}
```

Check Your Progress

1. Explain inheritance.
2. What are the different types of inheritances?
3. What do you mean by multiple inheritance in a program?
4. Explain a friend function.

Here we are using the single argument constructor to convert it to an object with two data members. This is also called implicit conversion. We can prevent usage of constructors for implicit conversion by declaring them as explicit. The explicit keyword prefixed to a constructor will allow it to initialize the objects but it will prevent this being used for data conversions as in the above program. The explicit constructor will be invoked only explicitly i.e. only for creation of an object. By specifying explicit, the constructor will look for receiving all data members of the class. Thus, it will prevent passing of lesser number of arguments.

NOTES

4.7 SUMMARY

In this unit, you have learnt the basic properties of inheritance. It provides a unique solution for adding the properties and behaviour of a base class to a new class, known as a derived class. A derived class can derive or inherit and use the properties and behaviour of its base class. It can add more properties and behaviour of its own. A class, whether it is derived or not, can be identified from the class declaration.

You have learnt that access specifiers can be prefixed to members of a class. The default specifier, i.e., when there is no specific prefix, is private. Members from outside the class can access public members. Only members of the same class, or a friend, can access private members. A code outside the class can access a private member through functions of the same class indirectly, provided the function is a public one. Protected members can be accessed in the derived classes with the same access control characteristics. There are three types of inheritances: public, protected and private. In public inheritance, the public and protected members are inherited with the same access rights. Both public and protected members become protected and private, respectively in protected and private inheritances. Private members cannot be inherited at all. In multiple inheritances, there is a possibility that a derived class inherits from one base class twice in two directions. This can be avoided by declaring such base classes as virtual base classes.

You have also learnt that a friend function can be created to provide access to the members of one class from another. This concept throws open any class to members, thus affecting security and data hiding.

4.8 KEY TERMS

- **Inheritance:** It provides a unique solution for adding properties and behaviour of a base class to a new class, known as a derived class. This new class can derive or inherit and use the properties and behaviour of its base class.
- **Friend function:** This function is created to provide access to the members of one class from another. The friend function can access the members of multiple classes.

4.9 ANSWERS TO 'CHECK YOUR PROGRESS'

1. Inheritance provides a unique solution for adding properties and behaviour of a base class to a new class, known as a derived class. This new class can derive or inherit and use the properties and behaviour of its base class. It can add more properties and behaviour of its own. A class, whether it is derived or not, can be identified from the class declaration.
2. There are three types of inheritances: public, protected and private. In public inheritance, the public and protected members are inherited with the same access rights. Both public and protected members become protected and private respectively in the protected and private inheritances. Private members cannot be inherited at all.

3. In multiple inheritance, a derived class can inherit from more than one base class. Also, there is a possibility that a derived class inherits from one base class twice in two directions.
4. A friend function is created to provide access to the members of one class from another. A friend function can access the members of multiple classes.

4.10 QUESTIONS AND EXERCISES

Short-Answer Questions

1. Write short notes on the following:
   (i) Function hiding
   (ii) Inheritance
   (iii) Use of constructors in inherited classes
   (iv) Base class reference to derived class objects
   (v) Building reusable classes
   (vi) Friend functions vs. inheritance
2. State the output of the following programs:
   (i)
```
#include<iostream>
using namespace std;
class Account {
   protected:
         int number;
         double balance;
   public:
         Account(int a, double b){
               number=a;
               balance=b;
         }
   void debiting(double debit) {
         cout<<(balance - debit);
   }
};
class SubClass :public Account{
   public:
         SubClass(int c, double d):Account(c,d){
   }
};
int main() {
   SubClass Lakshmi(001, 10000.0);
         Lakshmi.debiting(1000.00);
}
```

NOTES

(ii)
```
#include<iostream>
using namespace std;
class Area {
   protected:
         int length;
   public:
         Area(int a){
               length=a;
         }
   void display(){
         cout<<(length*length);
   }
};
class SubClass :public Area {
int breadth;
public:
   SubClass(int p, int r): Area(p){
         breadth=r;
    }
public:
   void display(){
         Area::display();
         cout<<(length*breadth);
   }
};
int main() {
   Area sq(10);
   SubClass rect(10, 40);
   rect.display();
}
```

(iii)
```
#include<iostream>
using namespace std;
class Vol {
   protected:
         int side1;
   public:
         void getdata(int x){
               side1=x;
         }
   void display(){
         cout<<(side1*side1*side1)<<"\n";
   }
};
```

NOTES

```
class SubClass:public Vol {
   protected:
         int side2;
   public:
         void getdata(int y, int z){
               side1=y;
               side2=z;
         }
   void display(){
         cout<<(3.14*side1*side1*side2)<<"\n";
   }
};
class SubSubClass :public SubClass {
   int side3;
   public:
         void getdata(int c, int d, int e){
               side1=c;
               side2=d;
               side3=e;
         }
   void display(){
         cout<<(side1*side2*side3)<<"\n";
   }
};
int main() {
   SubSubClass var3;
   var3.getdata(2,3,4);
   Vol var1;
   var1.getdata(5);
   SubClass var2;
   var2.getdata(10, 5);
   var1.display();
   var1=var2;
   var1.display();
   var1=var3;
   var1.display();
}
```

(iv)
```
#include<iostream>
using namespace std;
class counter{
   unsigned int value;
   public:
         counter(){
                value=0;
            }
            void increment(){
                if (value<65535) value++;
            }
        void decrement(){
            if (value>0) value—;
        }
        unsigned int access_value(){
            return value;
        }
    };
    class range:public counter{
    int max_val;
    public:
        range(int max){
            max_val=max;
        }
        void inc(){
            if (access_value()<max_val)
            counter::increment();
        }
    };
    int main()
    {
        range x(7), y(5);
        for(int i=0; i<10; i++){
            x.inc();
            y.inc();
            cout<<"\n x="<<x.access_value();
            cout<<"\n y="<<y.access_value();
        }
    }
```

3. Find out the errors, if any, in the following statements:

(i) `class SubClass ::public Account{`

(ii) `SubClass(int x, double y): Book(x, y)()`

(iii)
```
protected
Book(int a, double b) {
```

(iv)
```
SubClass(int var1, double var2, int var3):
Book(var1)
{
pages=var3;
```

(v)
```
SubClass(int c, double d, int e):Account(int
c,double d){
```

(vi) `SubClass( c, d, e):Account(intc c, double d){`

(vii)
```
public:
protected:
int side1;
```

(viii) `baseClass =derivedClass;`

Long-Answer Questions

1. Write programs for the following:
   - (i) To create a bank account with personal details (name, address) in the base class, savings bank account details (account number, balance) with credit and debit facilities in one of the derived classes, and current account details of the same person (number, credit limit) along with credit and debit facilities in another class.
   - (ii) To extend the above program for linking fixed deposit details in another derived class.
   - (iii) To extend the above program for linking the loan availed by an account holder.

UNIT 5 INPUT/OUTPUT FILES

NOTES

Structure

5.0 INTRODUCTION

In this unit you will learn how programs enable communication between the user and a computer. Software is developed for carrying out the required tasks using the data supplied. The computer communicates the results in the user-defined media. You will learn that the basic input for any computer system is from the keyboard and the output is displayed on the monitor, and error messages are displayed on the console by default. In this unit, you will learn how C++ programs communicate with Input/Output (I/O) devices such as the keyboard and video monitor. Thus, C++ performs unique I/O operations using the facilities provided in C++ standard libraries. The input is given from the keyboard using `cin>>` and the output is displayed on the monitor using `cout<<`. The library functions of C language, such as `printf` and `scanf` can still be used in C++. But in C++ we use the new functions since these are simpler to use and at the same time more flexible and powerful. In C++, we use classes, objects, functions and overloaded operators for executing I/O functions.

In this unit you will also learn that file I/O functions are an extension of the concepts of standard I/O functions. The file I/O classes are also inherited from `basic_ios` classes. The derived class basic_fstream which is in the **<fstream>** header file, provides a link to all file I/O classes. The **ifstream** class is assigned for file input operations. The **ofstream** class is the stream that is assigned for file output operations. The class **fstream** is derived both from **ifstream** and **ofstream** classes and hence can be assigned both for input and output operations/functions of files. A file can be assigned to objects of **ifstream** or **ofstream** through constructors of the respective classes. These objects point to the respective files. Hence, writing to files can be carried out by using insertion operators. The `eof()` function can be used for finding out whether an end-of-file has been encountered. The file objects themselves can also be used for knowing if the end of file has been reached or whether an error condition has occurred.

You will learn that file redirection is an important and command line interface mechanism to perform specific tasks. In C++, the `cin` command is used to read and input characters from the keyboard. Similarly, we use C++ standard output commands to print on screen. The specification of these commands can be changed in a program by specifying input and output redirection. The input redirection command inputs characters from the file specified in the program and the output redirection file sends the output to the file specified with `cout` statement. Hence, the same program can be executed and run to either read from a specified file or from the keyboard depending on the redirection parameter specification.

5.1 UNIT OBJECTIVES

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

- Describe streams
- Explain buffers and iostreams
- Understand and initialize header files
- Explain the redirection process
- Analyse the basics of file input and output operations
- Write a C++ program using redirection
- Write a C++ program to input a specified file
- Write a C++ program to get specific output

5.2 STREAMS

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. 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 in Figure 5. 1(a).

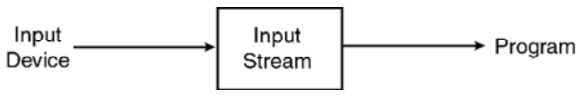


Figure 5.1(a) *Use of input stream*

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) below:

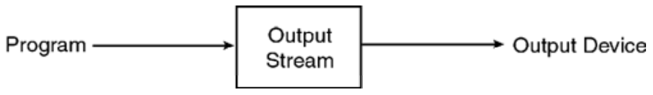


Figure 5.1(b) *Use of output stream*

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.

NOTES

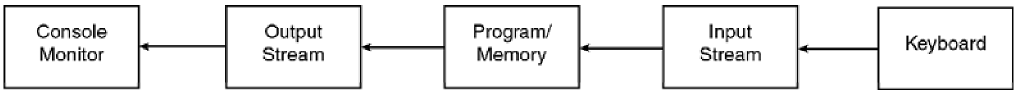


Figure 5.2 *I/O with keyboard and monitor*

The above figure also indicates the conceptual realization of standard I/O with streams.

5.3 BUFFERS AND `IOSTREAMS`

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.

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 below:

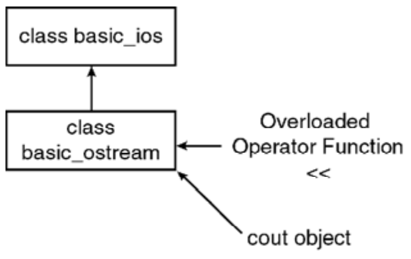


Figure 5.3 *Origin of << and cout*

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.

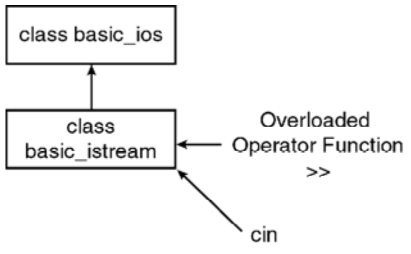


Figure 5.4 *Origin of >> and cin*

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 below:

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.

NOTES

`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.

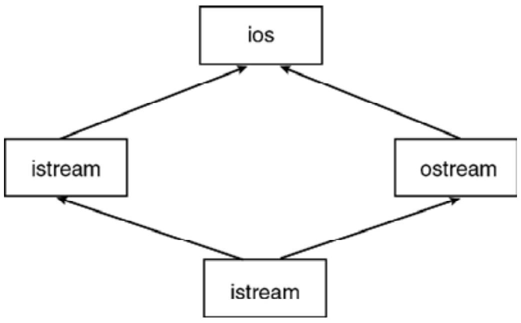


Figure 5.5 *iostream class derivation*

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 10 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, 10 characters (including white spaces) would have been received. The `for` loop will terminate since it has executed 10 times, 5 time receiving white space and five times the actual character. Hence we can receive only 5 characters interleaved with 5 white spaces. The characters entered after that will be ignored. The put function will therefore display 5 characters with spaces as the result below indicates.

Result of Program 5.1

```
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 5.2

```
#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 10 characters in contrast with the previous example. The next `put` function will display all the 10 characters typed, but without space. Note the difference between `get` function and >> operator. Had we used the `get` function, we would have received 5 characters interleaved with 5 spaces since it cannot ignore white spaces and the loop would have been executed 10 times getting 5 characters and 5 spaces.

In the next loop, since we use `get` function, although we entered 10 characters, because of the space in between, only 5 characters and spaces would have been received. This will be displayed as the result of the Example indicates.

Result of Program 5.2

```
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
```

`getline` and `write` functions

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 5.3

```
#include<iostream>
using namespace std;
int main(){
    char ch[15];
    cin.getline(ch,10);
    cout<<"\n"<<ch<<"\n";
}
```

Result of Program 5.3

```
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 5.4

```
#include<iostream>
using namespace std;
int main(){
    char ch[15];
    cin.getline(ch, 15);
    cout<<"\n";
    cout.write(ch, 10);
}
```

Result of Program 5.4

```
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.

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.

NOTES

NOTES

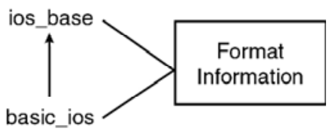


***Figure 5.6** Relationship between the top stream classes*

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 10 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 7 spaces, then 3 leading spaces will be left as shown in Figure 5.7.

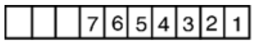


***Figure 5.5** Operation of width function*

The following example would confirm this.

Program 5.5

```
//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";
}
```

Result of Program 5.5

```
  123141
123141
```

In the latter case, since the width required exceeds the specified 5 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 4 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.

```
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 5 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 5 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 5.6

```
//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;
}
```

Result of Program 5.6

```
 256.141
256.141
256.14
```

Note that there are only 2 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 5.7

```
//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:

Result of Program 5.7

```
     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.

NOTES

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 5.8

```
//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;
}
```

Result of Program 5.8

```
$$$$$$$$$$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

NOTES

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 5.9

```
//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.

NOTES

Result of Program 5.9

```
Enter the length in foot and inch
56 78
foot = 56inch = 78
```

5.4 HEADER FILES

The standard C++ has a rich set of header files to accomplish various tasks, including Input/Output. A programmer can build his own libraries of Input/Output functions with the available header files. This makes the language flexible. The header files available in standard C++ are given below:

STANDARD C++ LIBRARY

The C++ Standard library provides support for the following:

- language support
- general utilities
- locales
- iterators
- numerics
- diagnostics
- strings
- containers
- algorithms
- input/output.

Headers

i. The elements of the C++ Standard library are declared or defined (as appropriate) in a header. The C++ Standard library provides 32 C++ headers, as shown in Table 5.4:

Table 5.4 *C++ Library Headers*

<algorithm>	**<iomanip>**	**<list>**	**<ostream>**	**<streambuf>**
<bitset>	**<ios>**	**<locale>**	**<queue>**	**<string>**
<complex>	**<iosfwd>**	**<map>**	**<set>**	**<typeinfo>**
<deque>	**<iostream>**	**<memory>**	**<sstream>**	**<utility>**
<exception>	**<istream>**	**<new>**	**<stack>**	**<valarray>**
<fstream>	**<iterator>**	**<numeric>**	**<stdexcept>**	**<vector>**
<functional>	**<limits>**			

ii. The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 5.5:

Table 5.5 *C++ Headers for C Library Facilities*

<cassert>	**<ciso646>**	**<csetjmp>**	**<cstdio>**	**<ctime>**
<cctype>	**<climits>**	**<csignal>**	**<cstdlib>**	**<cwchar>**
<cerrno>	**<clocale>**	**<cstdarg>**	**<cstring>**	**<cwctype>**
<cfloat>	**<cmath>**	**<cstddef>**		

iii. The .h headers dump all their names into the global namespace, whereas the newer forms keep their names in namespace std. Therefore, the newer forms are the preferred forms for all uses except for C++ programs which are intended to be strictly compatible with C.

iv. A translation unit may include library headers in any order. Each may be included more than once.

STRING LIBRARY

1. Applicable Headers

Character traits	<string>
String classes	<string>
C style strings	<cctype> <cwctype> <cstring> <cwchar> <cstdlib>

2. String Capacity

Functions	Applications
size()	Returns size_t
length()	Returns size_t
max_size()	Returns size_t = maximum size
capacity()	Returns size_t = capacity
clear()	Erases the string
empty()	Returns bool

3. String Modifiers

Str1+= Str2	Appends the Str2 to Str1
append(Str1, Str2)	Appends the Str2 to Str1
insert (pos, str)	Inserts str at pos
erase()	Erases the string
replace(pos,n,str)	Deletes n characters from pos in the string and replaces it with str

4. String Operations

find(str)	Returns the position of the string str if found. Else returns npos
rfind(str)	Traverses from the end of the string and returns the position of str if found
compare(str)	Returns an integer depending upon the result of comparison of the strings. Will return zero, if strings are identical.
= =	Returns bool

4.1 Header <cctype> synopsis

isalnum	Isdigit	isprint	isupper	tolower
isalpha	Isgraph	ispunct	isxdigit	toupper
iscntrl	Islower	isspace		

NOTES

4.2 Header <cstring> synopsis

memchr	strcat	strcspn	strncpy	strtok
memcmp	strchr	strerror	strpbrk	strxfrm
memcpy	strcmp	strlen	strrchr	
memmove	strcoll	strncat	strspn	
memset	strcpy	strncmp	strstr	

4.3 Header <cstdlib> synopsis

atol	mblen	strtod	wctomb
atof	mbstowcs	strtol	wcstombs
atoi	mbtowc	strtoul	

5. Containers Library Summary

Type	**Header(s)**
Sequences	<deque> <list> <queue> <stack> <vector>
Associative containers	<map> <set>
Bitset	<bitset>

6. Algorithms Library Summary

Type	**Header(s)**
Non-modifying sequence operations Mutating sequence operations Sorting and related operations	<algorithm>
C library algorithms	<cstdlib>

7. Syntax of Selected Algorithms

7.1 Non-modifying sequence operations

1. find

   InputIterator find(InputIterator first, InputIterator last, const T & value);

   The first iterator i for which *i = = value is returned. Returns last, if the value is not found.

2. find_first_of

   ForwardIterator1 find_first_of(ForwardIterator1 first1, ForwardIterator1 last1,

   forwardIterator2 first2, ForwardIterator2, last2, BinaryPredicate Pred);

   Returns the first iterator i in the range (last1 - first1) such that for some integer j in the range (last2 - first 2) *i = = *j and Pred (*i, *j) ! = false. Returns last1, if not found.

3. adjacent_find_first_of

   ForwardIterator adjacent_find_first_of(ForwardIterator first1, ForwardIterator, last1, BinaryPredicate Pred);

   Returns the first iterator i in the range (last1 - first1) such that *i = = *(i+1)and Pred (*i, *(i+1)) ! = false. Returns last1, if not found.

4. count

   count(InputIterator first, InputIterator last, const T & value);

   Returns the number of iterators i in the range (first, last) for which *i = = value.

5. mismatch

   pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2, InputIterator2 last2);

   Returns a pair of iterators i and j such that j = = first2 +(i – first1) and i is the first iterator in the range (first1, last1) such that

   ! (*i = = *(first2 + (i – first1)))

   Returns the pair last1 and first2 + (last1 – first1), if there is no mismatch.

6. equal

   bool equal(InputIterator1 first1, InputIterator1last1,InputIterator2 first2);

   Returns true, if for every iterator i in the range (first1, last1) the following conditions hold *i = = * (first2 + (i – first1))

   Returns false, otherwise.

7. search

   ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2);

   Returns the first iterator i in the range first1, (last1 – (last2 – first2)) such that for any non negative integer N less than (last2 – first2), the following is true.

   *(i + N) = = *(first2 + N)

7.2 Mutating sequence operations

1. copy

   OutputIterator copy(InputIterator1 first1, InputIterator1 last1, OutputIterator2 first2) ;

   Copies the elements from first1 to last1 and places them in position starting from first2.

   The first2 shall not be in the range from first1 to last1.

2. swap

   void swap(T & A, T & B);

   Exchanges A & B.

3. swap_range

   ForwardIterator2 swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2);

For each non-negative integer N < (last1 – first1) performs swap

(*(first1 + N), *(first2 + N))

4. transform

   OutputIterator transform(InputIterator first, InputIterator last, OutputIterator result, UnaryOperation OP);

   It performs the given operation OP in each value in the sequence and stores it from the position result.

5. replace

   void replace(ForwardIterator first, ForwardIterator last, const T & old_value, const T & new_value);

   Substitutes the elements referred by the iterator i in the range (first, last) with new_value, when *i = = old_value.

6. fill

   void fill(ForwardIterator first, ForwardIterator last, const T & value);

   Replaces all the elements by value.

7. remove

   ForwardIterator remove(ForwardIterator first, ForwardIterator last, const T & value);

   Deletes all the elements referred to by the iterator i in the range, first to last for which *i = = value.

8. unique

   ForwardIterator unique(ForwardIterator first, ForwardIterator last);

   Deletes all but the first element from every consecutive group of equal elements, referred to by iterator i such that *i = = *(i – 1).

9. reverse

   void reverse(BidirectionalIterator first, BidirectionalIterator last);

   It rearranges the sequence in the reverse order from the last element to the first element.

7.3 Sorting and related operations

1. sort

   void sort(RandomAccessIterator first, RandomAccessIterator last);

   Sorts the elements from first to last.

2. binary search

   ```
   bool binary_search(ForwardIterator first,
   ForwardIterator last, const T & value);
   ```

   Returns true if the value is found in the sequence.

3. merge

   ```
   merge(InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, InputIterator2 last2,
   OuputIteratorResult);
   ```

   Merges two sorted strings.

8. Numerics Library Summary

Complex numbers	<complex>
Numeric arrays	<valarray>
Generalized numeric operations	<numeric>
C library	<cmath> <cstdlib>

NOTES

9. Header <cstdio> Synopsis

Functions:					
clear	fgets	fscanf	gets	rename	tmpfile
fclose	fopen	fseek	perror	rewind	tmpnam
feof	fprintf	fsetpos	printf	scanf	ungetc
ferror	fputc	ftell	putc	setbuf	vfprintf
fflush	fputs	fwrite	putchar	setvbuf	vprintf
fgetc	fread	getc	puts	sprintf	vsprintf
fgetpos	freopen	getchar	remove	sscanf	

10. Headers for Input/Output

<iosbase>	base class
<iostream>	standard iostream objects and operations
<ios>	iostream base
<streambuf>	stream buffers
<istream>	input stream template
<ostream>	output stream template
<iomanip>	manipulators
<sstream>	stream to/from strings
<cstdlib>	character classification functions
<fstream>	streams to/from files

11. Headers for Language Support

<limits>	numeric limits
<new>	dynamic memory management
<typeinfo>	run-time type identification support
<exception>	exception-handling support
<cstddef>	C library language support
<cstdarg>	variable-length function argument lists
<csetjmp>	C-style stack unwinding
<cstdlib>	program termination
<ctime>	system clock

12. Iterator Functions

begin ()	Points to first element
end ()	Points to one-past-last element
rbegin ()	Points to first element of reverse sequence
rend ()	Points to one-past-last element of reverse sequence

13. Minimum and Maximum Functions in the Header <Algorithm>

min ()	Smaller of the two values.
max ()	Larger of the two values.
min_element ()	Smallest value in sequence.
max_element ()	Largest value in sequence.

NOTES

14. Predicates in the Header <Functional>

equal_to
not_equal_to
greater
less
greater_equal
less_equal
logical_and
logical_or
logical_not

15. Arithmetic Operations in the Header <Functional>

plus
minus
multiplies
divides
modulus
negate

5.5 REDIRECTION

File redirection is an important and command line interface mechanism to perform specific task. In C++, cin command is used to read and input characters from the keyboard. Similarly, we use C++ standard output commands to print onto the screen. The specification of these commands can be changed in a program by specifying input redirection and output redirection. The input redirection command will now input the characters from the file specified in the program and the output redirection file will send the output to the file specified with cout statement. Hence, the same program can be executed and run reading either from a specified file or from the keyboard depending on the redirection parameter specification.

Input redirection command is very useful to input various test record files into one specified FINAL file using the text editor. The text editor places the input into the specified FINAL file, and you can run the program any number of times reading from the specified file. Similarly, Output redirection helps to get the output from the specified file (say FINAL) which contains number of records for checking and interpretation. The text editor helps to get the output onto the screen or to print it as hardcopy.

Redirection is used by the programmers because it is flexible and easy to read or write a specified file explicitly in a program. The programmers get a file name from the input redirection, i.e., command line interface, open the same and perform the required action. The output of one program can be the input for another program.

Redirecting `stdin` and `stdout`: By default, in programming languages, standard input (`stdin`) and output (`stdout`) for the programs are assigned to the keyboard and display, respectively. To change the standard input and output can be done using the shell's redirection notation.

5.6 FILE INPUT AND OUTPUT

The beauty of C++ lies in the implementation of input/output mechanism. Irrespective of the type of device used, the input/output is handled in a similar manner.

Pictionial representation of a hierarchy of stream classes is given in Figure 14.1 below.

We have omitted the prefix basic to the class *ios* and its derived classes for better readability. However all of them are presented in the respective header files with out the prefix. For instance, the class basic_`fstream` can be found in `<fstream>` header file.

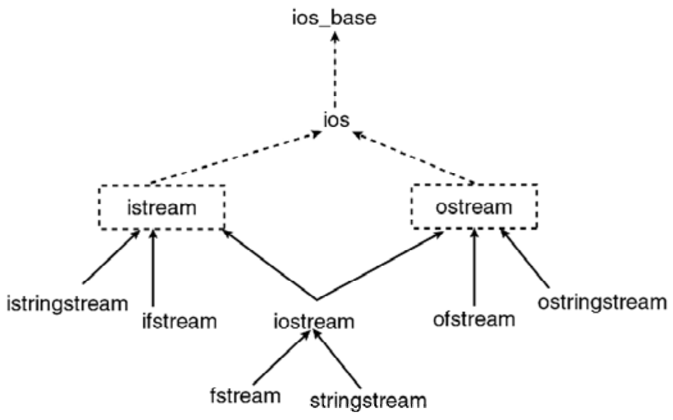


Figure 5.8 *Hierarchy of stream classes*

The additional streams included in this figure pertain to file stream and string stream. The file stream can be identified with appearance of the character 'f' in the name like `ifstream`, `fstream` etc. The names of the string streams can be identified by the appearance of `s`, `str` or `string`. The streams associated with the standard input/output will not contain these characters. Thus, the input/output is organized as three major types of streams.

- Standard input/output
- File input/output
- String input/output

File streams

The function of the three file streams classes are given below:

`ifstream` –	this is the class for input operations on files similar to `istream` for standard input. This contains the function `open` with default input mode. This class is derived from `istream`.
`ofstream` –	this is the file for output operations analogous to `ostream` class for standard output. It contains the function `open` with default output mode. This class is derived from `ostream`.
`fstream` –	this is common to both file input and output analogous to *iostream* and inherits properties of both the `istream` and the `ostream`. It also contains function `open` with default input mode. This class is derived from *iostream*.

With regard to standard input/output, we included *iostream* header file in our program files. The counterpart of *iostream* for file input/output is `fstream`. Therefore, `<fstream>` has to be included when we program file input/output. However, we may also need to include `<iostream>` file since we may direct the output to the console in such programs. Since we will be using std::cin and std::cout, we may also declare using 'namespace std' in the programs.

Assignment of file to stream through its constructor

In order to write to a file, we have to open a file or assign a file to output stream. We can open a file in the floppy disc drive. We can also open the file in hard disc drive. Let us choose the name of the file as `Temp.txt`. To open a file we have to create an object of class `ofstream`*:* Let us call it as **outfile**. The object name can be any valid identifier. We will use the constructor method to assign a file name to the object. The following statement will open a file for writing.

```
      ofstream outfile("Temp.txt");
```

Note that we declare an object of `ofstream` class and make a reference of a file to it. Thus we are assigning a stream to a file. Buffers will be allocated automatically. In order to write a string to the file we can use the overloaded insertion operator as shown below:

```
      outfile <<"the string";
```

The insertion operator is in the `ostream` class. Now let us see some examples of disk I/O.

The Program below writes strings to the specified file.

Program 5.10

```
  //To demonstrate writing to files using constructors
  #include<fstream>
  int main(){
      std::ofstream outfile("Temp.txt");
      outfile<<"Working with files is fun\n";
      outfile.close();
  }
```

In the above program the file will be closed by the system after the program execution is terminated.

Reading the same file can help check whether we have actually written to the disk. To open a file for reading, we creating an object *infile* of class `ifstream`. We supply the file name to the object through constructor of the `ifstream` class as given below:

```
      ifstream infile ("Temp.txt");
```

The next program explains how to read a line of string from a file.

Program 5.11

```
  //To demonstrate reading from file
  #include<fstream>
  using namespace std;
  #include<iostream>
  int main(){
      ifstream infile("Temp.txt");
      string buff;
      getline(infile, buff);
      cout<<buff;
      infile.close();
  }
```

Result of Program 5.11

```
Working with files is fun
```

The above 2 programs use operators and functions of `istream` and `ostream` classes. The getline function is also a member function of `istream` class. The cout enables display of the same in console monitor.

NOTES

`getline` function

The syntax of `getline` function is given below:

```
getline(istream obj, string & strobj char eol)
```

The default `eol` character is `'\n'`**.** Since we have omitted the last parameter in the program it will read till a new line character is found.

End of file

In doing disk I/O, we need to know when the end of file has been reached so as to stop reading, otherwise it might lead to an error. We can write as follows to read a file fully:

```
while(infile){ }
```

The group of statements will be executed when `infile` is true or returns non-zero values. The object `infile` will return non-zero when reading is normal. When error condition occurs or end of file is reached it will return zero.

The declaration of `eof()` function of the *ios* class is given below.

```
bool eof() const;
```

The function will return true when the end of file is reached. Otherwise it will return false or zero.

If we want to modify the while loop using *eof* function, we can code as follows:

```
while(!eof()){
read file}
```

There is another integer *EOF* defined in `ios`. When the end of file is reached, *EOF* will be a non-zero integer. Otherwise, it will be zero. Either the *eof* function or *EOF* integer can be used to detect the end of file.

Note the difference. The objects of stream classes return zero if error (including end of file) occurred. However, the *eof* function will return non-zero value if *eof* has occurred. The value of *eof()* or the return value of the stream objects can be checked for ascertaining whether end of file has been reached.

Opening Files using `open` ()

So far, the filenames have been assigned using constructor. The function `open` can be used to open files for reading and writing. For writing to a file, we open as follows:

```
ofstream outfile ;
outfile.open ("Temp2.txt") ;
```

We declare an `ofstream` object outfile. Then we call the function `open` with the object to write to a file. Later on if we want to write to another file, we do not have to reassign the object. We can simply call open as follows:

```
outfile.open ("Temp3.txt") ;
```

When we want to open a file for reading, we have to create an object of `ifstream` and then call open with it as given below:

```
ifstream infile ;
infile.open("Temp3.txt");
```

The program below demonstrates the use of `open` function.

Program 5.12

```
//To demonstrate writing and reading- using open
#include<fstream>
using namespace std;
#include<iostream>
int main(){                  //Writing
    ofstream outf;
    outf.open("Temp2.txt");
    outf<<"Working with files is fun\n";
    outf<<"Writing to files is also fun\n";
    outf.close();
                             //Reading
    char buff[80];
    ifstream inf("Temp2.txt");
    while(inf){
       inf.getline(buff, 80);
       cout<<buff<<"\n";
    }
    inf.close();
}
```

Result of Program 5.12

```
Working with files is fun
Writing to files is also fun
```

The object of `ofstream` is named outf and object of `ifstream` is named inf. We open and close the files with these objects. In the above program C style getline function is used.

So far, to open the files using `open`, we supplied only one argument, namely the name of the file such as Temp2.txt. Actually, the `open` function can take two arguments. Since we were not supplying the second argument, it is assumed to be absent. In order to append to a file rather than writing, we can indicate it in the arguments of the `open` function:

```
outf.open("Temp6.txt", ios::app);
```

If `outf` had been declared as an object of output stream, then the above statement will open the file `Temp6.txt` for append mode writing. We do the same in the following program:

Program 5.13

```
//To demonstrate append mode
#include<fstream>
#include<iostream>
```

```
using namespace std;
int main(){
            //Writing to file 1
   ofstream outf1;
   outf1.open("Temp6.txt");
   outf1<<"Working with files is fun\n";
   outf1<<"Writing to files is also fun\n";
   outf1.close();
            //appending to end of file Temp6
   outf1.open("Temp6.txt", ios::app);
   outf1<<"do your duty \n";
   outf1<<"and leave the rest to the lord\n";
   outf1.close();
            //Reading from file 1
   char buff[80];
   ifstream inf;
   inf.open("Temp6.txt");
   while(inf){
      inf.getline(buff, 80);
      cout<<buff<<"\n";
   }
inf.close();
}
```

NOTES

Here `outf1` is the object for writing and we write two lines of text each followed by a new line character. Then we close the file. Thereafter we open the same file using the same object, but with the second parameter for appending namely *ios::app*. Now, whatever we write, will be written at the end of the existing text. We of course close the file after every writing. Then we read the file and we get the results as given below:

Result of Program 5.13

```
Working with files is fun
Writing to files is also fun
do your duty
and leave the rest to the lord
```

File mode parameters

The function `open` was used with object of `ifstream` class and `ofstream` class for reading and writing respectively. As seen in the previous program, the `open` function can receive two arguments. The second argument is called file mode parameter. It specifies additional attributes. Some of the file mode parameters are given in Table 5.6.

The default second arguments for `open` functions are as follows:

When the file is opened for reading using `ifstream` object, *ios::in*

When the file is opened for writing using `ofstream` object, it is *ios::out*

This explains why the second argument was not given to the open function in programs seen so far.

When we open a file for writing, and not for appending, using `ofstream` object, the file if existing will be deleted. If we want to retain the contents of a file opened for writing, then we can open the file in the append mode using `ios::app`. Similarly, `ios::trunc` deletes the contents of the file, if it exists. Therefore, the mode *ios::out* and ios::*trunc* are identical.

Table 5.6 *File mode parameters*

Parameter	Function
`ios::app`	Writing in append mode
`ios::binary`	Binary file
`ios::in`	Reading
`ios::out`	Writing
ios::trunc	Delete contents
ios::ate	Go to end of file on opening
ios::noreplace	If file exists, open fails

`ios::ate` mode takes us to the end of the file on opening. However, this mode enables us to add data or write anywhere in the file. While `ios::app` allows us to write at the end of the file, it does not allow us to modify the existing contents. When we open the file in `ios::app` mode, if file is not existing, it will open a new file. It is also clear that the class `ifstream` is meant for reading and `ofstream` is meant for writing. There is also no need to specify the second arguments, unless we are interested in additional actions.

Error handling

We had assumed that no error will occur while opening a file or writing or reading. But in practice errors do occur.

Any of the following may result in an error

- Trying to read a non existent file
- Lack of space when writing
- Reaching the end of file

When we use statement like while (infile) and when the file reading is normal, it will return non-zero number. The loop will terminate when the object becomes zero. It occurs when end of file is reached.

While checking the condition of stream objects or eof(), we will get a status byte which indicates the status of the stream. The status byte can be read through a function called rdstate(). The first four bits of the status byte are not used and the next four bits indicate the status as given in Table 5.7.

Table 5.7 *Status byte*

Bit	Status
`xxxx 0000`	No error
`0001`	At end of file
`0011`	Read or write operation failed
`0110`	Invalid operation, such as opening a non-existing file
`1100`	Hard error

The program below has been formulated for error checking. We first open an existing file and check the status and later we open a non-existing file. This will help us to understand the various states.

Program 5.14

```
//To demonstrate error handling
#include<fstream>
#include<iostream>
using namespace std;
int main(){     //Reading from file
    ifstream inf;
    inf.open("Temp6.txt");
    if (!inf)
       cout<<"First file could not be opened \n";
    else
       cout<<"First file open successful \n";
    cout<<"Error state:"<<inf.rdstate()<<"\n";
    cout<<"good:"<<inf.good()<<"\n";
    cout<<"eof:"<<inf.eof()<<"\n";
    cout<<"fail:"<<inf.fail()<<"\n";
    cout<<"bad:"<<inf.bad()<<"\n";
    inf.close();
    inf.open("T9.dat");
    if (!inf)
       cout<<"Second file could not be opened \n";
    else
       cout<<"Second file open successful \n";
    cout<<"Error state:"<<inf.rdstate()<<"\n";
    cout<<"good:"<<inf.good()<<"\n";
    cout<<"eof:"<<inf.eof()<<"\n";
    cout<<"fail:"<<inf.fail()<<"\n";
    cout<<"bad:"<<inf.bad()<<"\n";
    inf.close();
}
```

Result of Program 5.14

```
First file open successful
Error state:0
good:1
eof:0
fail:0
bad:0
Second file could not be opened
Error state:4
good:0
eof:0
```

NOTES

```
fail:1
bad:0
```

NOTES

When we open the first file, it exists. Therefore, we will get the result `file open successful`. Therefore, the `error state` will be zero and `good` will be 1. The *eof* will be zero, since it has not reached the end of the file. The `fail` and `bad` will be zero. Zeros indicate false and 1 or any non-zero integer indicates true.

When we open the second file, it does not exist. Therefore, we will get the result `file could not be opened`. Therefore, the `error state` returned by `rd` state is 4, indicating invalid operation or opening a non-existing file. `Good` will be zero because we could not open the file. The `eof` will be zero, since it has not reached the end of the file. The `fail` is true because we could not open the file.

It is a good habit to include these statements as a routine in programs involving file I/O.

Standard Error

We are aware that standard input device is the keyboard. The default standard output device is the monitor where the error messages are displayed. The stream class object `cin` points to keyboard, `cout` points to the monitor. There are two objects of stream class, which are associated with the standard device for error messages. They are `cerr` and `clog`. The stream is unbuffered when the error messages are sent directly to the monitor without waiting either for the buffer to fill or for the new line character. When we want unbuffered error messages we use `cerr`. The `clog` object is used when we want a buffered error message system. In this case the error message will be displayed when buffer is full or new line character is encountered. In both cases the display is on the monitor. The following program explains the use of `cerr`.

Program 5.15

```
//To demonstrate cerr
#include<fstream>
using namespace std;
#include<iostream>
int main(){                        //Reading from file
    ifstream inf;
    inf.open("Temp6.txt");
    if (!inf) {
       cerr<<"file Temp6.txt could not be opened \n";
       exit(-1); }
    else
       cout<<"file open successful \n";
    cout<<"Error state:"<<inf.rdstate()<<"\n";
    cout<<"good:"<<inf.good()<<"\n";
    cout<<"eof:"<<inf.eof()<<"\n";
    cout<<"fail:"<<inf.fail()<<"\n";
    cout<<"bad:"<<inf.bad()<<"\n";
    inf.close();
    inf.open("T9.dat");
    if (!inf) {
         cerr<<"file T9.dat could not be opened \n";
         exit(-1); }
      else
         cout<<"file open successful \n";
      cout<<"Error state:"<<inf.rdstate()<<"\n";
      cout<<"good:"<<inf.good()<<"\n";
      cout<<"eof:"<<inf.eof()<<"\n";
      cout<<"fail:"<<inf.fail()<<"\n";
      cout<<"bad:"<<inf.bad()<<"\n";
      inf.close();
   }
```

NOTES

Result of Program 5.15

```
file open successful
Error state:0
good:1
eof:0
fail:0
bad:0
file T9 could not be opened
```

In the above program, we are using the object `cerr`. We open a file. After opening we check whether the file is being read or not. If not, an error message is printed. On top of the program, we open an existing file. Therefore, file open will be successful and it prints the associated error statements. When we try to open the non-existent file *T9*, the `inf` becomes zero and hence the error message. '*file T9 could not be opened'* appears. After execution of the error message, the program calls exit (*-1*), which will terminate the program. Hence, the second group of error statements will not be executed at all. The result of the program clearly confirms this. Therefore, the purpose of `exit()` is to terminate the program execution instantly. The purpose of *`cerr`* is to display user defined error messages in the console monitor, whereas the function *`rdstate`* can be used to get predefined error messages.

Sequential Access

We have been using the stream objects to write to files and also read from files by using insertion and extraction operators.

```
outfile <<var;
infile >>var;
```

We also used getline function to get one line at a time from the file. Now, let us look at the sequential input and output operations for writing and reading one character at a time in files:

- `put()`
- `get()`

`put` and `get` functions

They are designed to handle one character at a time. The get function reads one character at a time, while the put function writes to an output device, both through the designated stream. Look at the example given below to understand the use of these functions:

Program 5.16

```
//To demonstrate put and get functions
#include<fstream>
#include<iostream>
using namespace std;
int main(){
               //Writing to file
    ofstream outf;
    outf.open("Temp8.txt");
    char arr[]="demonstrating put function";
    int lenarr=strlen(arr);
    for (int i=0; i<lenarr; i++)
    outf.put(arr[i]);
    outf.close();
               //Reading from file 1
    ifstream inf;
    inf.open("Temp8.txt");
    char ch;
    while(inf){
       inf.get(ch);
       cout<<ch;
    }
    inf.close();
}
```

In the program, we define an object outf of class `ofstream`. We open file Temp8 in association with the outf object. Then we declared a character array and initialized it. The integer variable lenarr contains the length of the character array. Now, we write the character array, one character at a time to the file in the for loop. The statement, which writes one character, is given below:

```
    outf.put(arr[i]);
```

After the string has been written on to the file, the file is closed. To read, we declare an object inf and read the characters one at a time in the while loop and display the character read from the file using the following statement:

```
cout << ch ;
```

Result of Program 5.16

```
demonstrating put function
```

Binary files

There are two ways of storing data in a file as given below:

- `Binary form`
- `Text form`

Suppose, we want to store a five digit number say 19876 in the text form, then it will be stored as five characters. Each character occupies one byte, which means that we will require five bytes to store five-digit integer in the text form. This requires storage of 40 bits. Till now, we have been using text form of storage. On the contrary, to store a short integer, we need only two bytes. Therefore, if we can store them in binary form, then we will need only two bytes or 16 bits to store the number. The savings will be much more when we deal with floating point numbers.

NOTES

When we store a number in text form, we convert the number to characters. However, storing a number in binary form requires storing it in bits. However, for a character, the binary representation as well as the text representation are one and the same since, in either case, it occupies eight bits.

The text format is easy to read. We can even use a notepad to read and edit a text file. The portability of text file is also assured. In case of numbers, binary form is more appropriate. It also occupies lesser space when we store it in binary form and hence it will be faster.

The default mode is text. The Unix systems have only text mode for storage. There is no separate binary mode for storing in Unix systems. However, in other operating systems, we can store the data either in text mode or in binary mode. When we use binary mode, two more functions are available in the language as given below:

- `write()`
- `read()`

`write` and `read` functions

We can directly write text to a file. However, whenever we want to write any other type of variable, we have to do type casting as given below:

```
outf.write((char* )& array, size of array);
```

In the above statement, the *array* represents the variable or structure or array to be written on to the file in the binary format. So we are type casting the reference of the variable array (to be written in the file) to type pointer to character. Execution of the statement will copy the entire variable of the given size to the stream object outf. Subsequently, it will be written on to the file. Similarly, for reading the variable or a structure or an array, we have to use the read function in the same manner as given below:

```
inf.read((char *) & array, size of array);
```

The size of the array will be received in terms of number of bytes. Therefore, the above statement will copy the number of bytes equivalent to the size of array from the file to the array. Therefore, this approach can be used with any type of arrays or structures or records etc. The program below implements write and read member functions for file I/O. Each one is compliment of the other. Therefore, read can be used to recover the data written using write.

Program 5.17

```
//To demonstrate structure write and read - binary file
#include<fstream>
#include<iostream>
using namespace std;
int main(){
                //Writing to file
    ofstream outf;
```

NOTES

```
    outf.open(Temp9.dat");
    struct Account{
       int number;
       double balance;
    };
    Account arr[3]={1, 140.0, 2, 200.0, 3, 300.0};
    outf.write((char *) &arr, sizeof(arr));
    outf.close();
                //Reading from file
    ifstream inf;
    inf.open("Temp9.dat");
    inf.read((char *) &arr, sizeof(arr));
    for (int i=0; i<3; i++){
       cout<<"Account No. "<<arr[i].number<<" balance:
"<<arr[i].balance<<"\n";}
    inf.close();
}
```

In the above program, we want to write a structure on to a binary file. Therefore the structure *Account* is created. A structure array `arr` of size 3 is created and initialized. Then we write the structure to the file using write function. Note that the data is cast to *char*.

Then we open the same file and read the contents of the array. Look at the ease with which we have read the entire structure in one statement. Then the data read is printed. The result of the program demonstrates binary file and writing and reading from it.

Result of Program 5.17

```
Account No. 1 balance: 140
Account No. 2 balance: 200
Account No. 3 balance: 300
```

Writing and reading objects to files

We wrote and read a structure in the previous program. We can similarly write an object to a file. We open a file for writing through constructor of class `ofstream`. Then we call write function with the object as shown in program below:

Program 5.18

```
//To demonstrate writing object to file
#include<fstream>
#include<iostream>
using namespace std;
class Account {
   private:
      int number;
      double balance;
   public:
        Account(int num, double bal) {
           number=num;
           balance=bal;
        }
  };
  int main() {
      Account Vinay(001, 11001.00);
      ofstream outfile("Temp1.dat");
      outfile.write((char *)&Vinay, sizeof(Vinay));
  }
```

The class is very familiar one. After the object `Vinay` is created, it is written to the file using write function. We could have calculated the size of the object in bytes and given it as the second argument to the write functions. Instead we have specified `sizeof(Vinay)` so that the system will calculate the exact size. Further more we are casting the object `Vinay` as character array.

We can open the same file for reading using object infile of class `ifstream` as Program 5.19 demonstrates.

Program 5.19

```
  //To demonstrate reading object from file
  #include<fstream>
  #include<iostream>
  using namespace std;
  class Account {
      int number;
      double balance;
      public:
         void display() {
            cout<<"Account number: "<<number<<"\n";
            cout<<"balance: "<<balance<<"\n";
         }
      };
  int main() {
      Account Vinay;
      ifstream infile("Temp1.dat");
      infile.read((char *)&Vinay, sizeof(Vinay));
      Vinay.display();
  }
```

In this program, the name of the file is passed to the constructor of the `ifstream` class. However, to display what has been read, we call the member function of the class Account. Note also that we created object in the class and passed it on to the file as an object. While we read it, we get the object and then decipher the contents using the class. Therefore, we can read and write objects as part of the class only unlike reading / writing strings or simple variables. See the casting to character pointer.

NOTES

Result of Program 5.19

```
Account number: 1
balance: 11001
```

NOTES

We can write a program to read objects written in `Temp1.dat` file in Program 5.20 by using open.

Program 5.20

```
//To demonstrate reading object from file- using open
#include<fstream>
#include<iostream>
using namespace std;
class Account {
    int number;
    double balance;
    public:
       void display() {
          cout<<"Account number: "<<number<<"\n";
          cout<<"balance: "<<balance<<"\n";
       }
};
int main() {
    Account Vinay;
    ifstream inf;
    inf.open("Temp1.dat");
    inf.read((char *)&Vinay, 80);
    Vinay.display();
    inf.close();
}
```

We have modified only in the main function. We declare inf as an object of class `ifstream`. We call open with the object and pass the file name. The reading and display are as in previous program. It is mandatory close the file after reading.

File pointers

C++ also supports file pointers. A file pointer points to a data element such as character in the file. The pointers are helpful in lower level operations in files. There are two types of pointers:

- get pointer
- put pointer

The get pointer is also called input pointer. When we open a file for reading, we can use the get pointer. The put pointer is also called output pointer. When we open a file for writing, we can use put pointer. These pointers are helpful in navigation through a file.

When we open a file for reading, the get pointer will be at location zero and not 1. The bytes in the file are numbered from zero. Therefore, automatically when we assign an object to `ifstream` and then initialize the object with a file name, the get pointer will be ready to read the contents from 0th position. Similarly, when we want to write we will assign to an `ofstream` object a filename. Then, the put pointer will point to the 0th position of the given file name after it is created. When we open a file for appending, the put pointer will point to the 0th position. But, when we say write, then the pointer will advance to one position after the last character in the file.

File pointer functions

There are essentially four functions, which help us to navigate the file as given below in Table 5.8. These functions belong to file stream classes.

Table 5.8 *File Pointer Functions*

Function	Purpose
`tellg()`	Returns the current position of the get pointer
`seekg()`	Moves the get pointer to the specified location
`tellp()`	Returns the current position of the put pointer
`seekp()`	Moves the put pointer to the specified location

Now, let us look at an example for navigation through files, which is given below:

Program 5.21

```
//To demonstrate file pointers
#include<fstream>
#include<iostream>
using namespace std;
int main(){
    ifstream inf;
    inf.open("Temp6.txt");
    int gvar =inf.tellg();
    cout<<"\n current position of get pointer is: "<<gvar;
    inf.seekg(11);
    gvar =inf.tellg();
    cout<<"\n current position of get pointer is: "<<gvar;
    inf.close();
                                        //put    pointer
operation
    ofstream outf;
    outf.open("Temp6.txt", ios::app);
    int pvar =outf.tellp();
    cout<<"\n current position of get pointer is: "<<pvar;
    outf<<"Ten chars\n";
    pvar =outf.tellp();
    cout<<"\n current position of get pointer is: "<<pvar;
    outf.seekp(11);
    pvar =outf.tellp();
    cout<<"\n current position of get pointer is: "<<pvar;
    outf.close();
}
```

At the outset of the program, we open an existing file Temp6.txt for reading. Then we call tellg() with the object inf as given below:

```
int gvar = inf.tellg();
```

We assign the value returned by tellg to gvar. Hence we will get the current position of the get pointer, which is printed, in the next statement. Obviously the get pointer will be at the 0th location. Hence, cout will give a value of zero. Look at the next statement reproduced below:

```
inf.seekg(11);
```

The pointer is directed to move to the 11th position. The next statement prints out the current position of the get pointer. It will print 11. We close the file.

Now, we declare an object outf of class `ofstream`. Then we open the same file in the append mode as indicated by the mode parameter *ios::.app*. Then we have the following statement.

```
int pvar = outf.tellp();
```

This statement will return a pointer to pvar indicating the current position of the put pointer. In the append mode, the file pointer will move to the position after the last character only after we say write something. Therefore, we will again get a print of zero, indicating that the file pointer is at the 0th position. After printing the position, we write to the file as given below:

```
outf << "Ten chars\n";
```

The above statement will write the string to the file at the end of the existing text and therefore after the write, the file pointer will move to the last position. The result indicates that the file pointer was at 114th position, because of the existing text and added to that the writing of the 9 characters.

The next statement moves the put pointer from 114th position to the 11th position.

The result of the program is given below:

Result of Program 5.21

```
current position of get pointer is: 0
current position of get pointer is: 11
current position of get pointer is: 0
current position of get pointer is: 114
current position of get pointer is: 11
```

We moved the file pointer using `seekg` and `seekp` functions, which set the pointers to the specified positions. These functions can also be used with two arguments as given below:

```
seekg(offset, position);
seekp(offset, position);
```

The offset is the number of characters by which the pointers should move. Position denotes the initial position of the file pointer. There are three variations possible in the argument position as given below:

```
ios::beg – beginning of the file
ios::cur – current position of the pointer
ios::end – end of the file
```

Some illustrations of the usage of the pointers with two arguments are given below:

(i) `outf.seekp(-p, ios::cur);`
   This will move the put pointer backwards by *p* bytes from the current position.

(ii) `inf.seekg(-g, ios::end);`
   This will move the get pointer backwards by *g* bytes from the end of the file.

(iii) `inf.seekg(g, ios::beg);`
   This will move the get pointer by *g* bytes from the beginning of the file.

Writing multiple objects to file

We wrote one object to a file. The same program can be extended to write three objects in a file.

Program 5.22

```
//To demonstrate writing more objects to file
#include<fstream>
#include<iostream>
using namespace std;
class Account{
    private:
       int number;
       double balance;
    public:
       Account(int num, double bal) {
          number=num;
          Vinaybalance=bal;
        }
};
int main() {
    Account Vinay(001, 11001.00);
    ofstream outfile("Temp.dat", ios::app);
    outfile.write((char *)&Vinay, sizeof(Vinay));
    Account Mani(002, 5001.00);
    outfile.write((char *)&Mani, sizeof(Mani));
    Account sita(003, 6001.00);
    outfile.write((char *)&sita, sizeof(sita));
}
```

The program has a class Account. In the main function, we open a file in the append mode by specifying `ios::app` as the second argument. If the file is not existing, it will open a new file and write three records corresponding to the three objects. In the next example the contents of the file will be displayed.

NOTES

Case Study

Bank Account in File

Random access of files Accessing contents of files can be carried out either sequentially or at random. So far, we were reading and writing sequentially from beginning of the file. We can also directly reach the location of interest, which is known as random access. This will be useful when the file consists of records or structures of equal length as in a database file. Then using the file pointer we can jump to required record. If we know the size of the record in bytes, say x, and if we wish to move to the nth record for reading, then we can access it by seekg((n-1)x). Thus, we can use the file pointer for random access to the file. In database files or the file containing records, we may like to either delete or modify a record. For this purpose, we can access the location through file pointers. So we need to read and write in the same file in the same program. For this purpose, we can open a file in both read and write mode as given below:

```
fstream iofile ;
iofile.open (ios.in|ios:out);
```

In the above, we are creating an object of `fstream` class instead of `ifstream` or `ofstream` as we had been doing hitherto. Such files can be written into and read from. In the next statement we specify both read and write mode to carry out both of them simultaneously.

As we know, the `fstream` class is a derived through multiple inheritance of `istream` and `ostream` through iostream. Therefore, it inherits the following:

- `istream` objects and functions
- `ostream` objects and functions
- two buffers one for input and one for output
- synchronizing the handling of the two buffers

This facilitates achievement of I/O in the same program by output pointer in keeping both pointers active simultaneously. It can move the input pointer in the input buffer and the output pointer in output buffer at will. Yes, moving the pointer is basically moving in the buffer which contains the file and not in the actual file, since the `fstream` class uses input and output buffers to read and write to file.

The following modes will be required for such a program of random access involving records.

```
ios:in for reading the file
ios:binary for binary I/O
```

(This will not work in Unix files).

If we choose append mode for writing, we can only write at the end of the file. But, we want to modify the contents of the file. Therefore, we need (*ios::out*) mode. But, if we open the file in this mode, the entire contents of the file will be deleted! Hence, to facilitate write mode as well as to preserve the existing contents, we can combine *ios:ate* mode. Thus, we need to open the file in a combination of multiple modes as given below:

```
iofile.open (ios::in |ios::out |ios::ate|ios::binary)
```

We use get pointer seekg() for navigation during reading and put pointer seekp() for navigation during writing.

Now let us write a program Bank Account using random access.

Program 5.23

```
//To demonstrate random access of a binary file
//The file Temp.dat created in Program 14.13
#include<fstream>
using namespace std;
#include<iostream>
class Account {
    int number;
    double balance;
    public:
       void getdata(){
          cout<<"\n Enter Account Number";
          cin>>number;
          cout<<"\n Enter balance";
          cin>>balance;
    }
    void display() {
       cout<<"Account number: "<<number<<"\n";
       cout<<"balance: "<<balance<<"\n";
    }
};
int main(){
    Account Ac;
    fstream iofile;
    iofile.open("Temp.dat",
ios::ate|ios::in|ios::out|ios::binary);
                                  //Read  the  contents  of
Temp.dat
    iofile.seekg(0);
    cout<<"Details of Bank Account \n";
    while(iofile.read((char *) &Ac, sizeof Ac)){
       Ac.display();
    }
                                  //Modify a record
    cout<<"Enter Record No. to be modified";
    int rnum;
    cin>>rnum;
    int position=(rnum-1)*sizeof(Ac);
    if(iofile.eof()) iofile.clear();
    iofile.seekp(position);
    Ac.getdata();
    iofile.write((char *)&Ac, sizeof Ac)<<flush;
                                  //show updated file
    iofile.seekg(0);
```

NOTES

```
    cout<<"Details of updated Bank Account \n";
    while(iofile.read((char *) &Ac, sizeof Ac)){
       Ac.display();
    }
    iofile.close();
}
```

NOTES

The above program demonstrates creation, editing, writing and reading random access files. We wish to use the file created in the previous example in this program. We have declared a class Account with the same data members as that of the previous example with a function getdata and a function display.

Look at the main function. Traverse the program from the top line by line. We declare an object Ac of class Account. Then, we declare an object iofile of class `fstream`. Note that we are using `fstream` class in order to enable reading and writing. The next statement opens a file Temp.dat. The file is opened in the following modes:

- Input
- Output
- Binary
- go to end of file on opening (ate)

Thus the same file and same object can be used both for reading and writing. Before we proceed further, we want to check the contents of the file Temp.dat. If we had not erased the contents of the file, the three records pertaining to account will be there in the file. We move the file pointer to the 0^{th} position. Then, we read the contents of the file in the *while* loop. The contents are read on the header of the loop and display of the contents are made in the body of the loop.

Now we want to modify a record at random. The second block of statements in the main function does this. It asks for entering the record number to be modified. We can specify any number ranging from 1 to 3 since we have only three records in the file. Then, the variable position is assigned the value of (*rnum-1*) times size of the record. Then we seek the position corresponding to the beginning of the chosen record. In case, we want to locate the second record, then the *put* pointer will just cross the first record in the above method. Then we can get data using the member function of the class and write the same on to the file at the same location using the write function.

The third block is for display of the updated file. It is same as the first block in the main function. The result of the program is the initial data as well as the interactions with the system and finally the updated bank Account details as given below.

Check Your Progress

1. Define a header file fstream.
2. Write the output stream command for writing to a file.
3. Write the input stream command for reading from a file.
4. What is the role of eof()?
5. What are get pointer and put pointer commands?
6. Describe the various categories of stream classes.
7. Explain file redirection process.

Result of Program 5.23

```
Details of Bank Account
Account number: 1
balance: 11001
Account number: 2
balance: 5001
Account number: 3
balance: 6001
```

```
Enter Record No. to be modified 2
Enter Account Number 55
Enter balance 18765
Details of updated Bank Account
Account number: 1
balance: 11001
Account number: 55
balance: 18765
Account number: 3
balance: 6001
```

In the above program, we could move the file pointer to the desired location at random to modify the contents. Thus, the iostream classes facilitate random access of the files.

String stream

`Cout` is an `ostream` object while `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>`. `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 5.24

```
//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.

Result of Program 5.24

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 5.25

```
//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.

Result of Program 5.25

```
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.

5.7 SUMMARY

NOTES

In this unit you have learnt that C++ provides a common interface for input/output (I/O) functions using various devices. The standard input device is the keyboard and the standard output device is the monitor. In C++, the I/O operation is implemented using the classes in the standard library. This gives a lot of flexibility to programmers. The I/O operation is visualized as the exchange of a stream of bytes between the program and the I/O devices. A stream class is the intermediary between the I/O devices and the program. Buffers are designated in the computer through the class `basic_streambuf` to facilitate communication between incompatible devices. The `ios_base` class is on top of the hierarchy of classes. The `basic_ios` class is derived from it and it facilitates I/O operations. The stream classes are supplied along with the standard library, which comes with C++ IDE. The insertion operator is used in conjunction with the `cout` object of the `istream` class. The extraction operator is used in conjunction with the `cin` object of the `ostream` class. The `iostream` class inherits its characteristics from both `istream` and `ostream`. The `istream` class has the following functions for reading:

? `get()`

? read()

? getline()

The `ostream` class consists of the following functions for writing:

? `put()`

? write()

In C++, the output can be formatted using the functions of the `ios` class and `ios_base class`. You have also learnt that file I/O operations are an extension of the standard I/O operations. File I/O classes are also inherited from the `basic_ios` class. The derived class `basic_fstream`, which is in **<fstream>** header file, provides a link to all file I/O classes. The **ifstream** class is assigned for file input operations. The **ofstream** class is the stream that is assigned for file output operations. The class **fstream** is derived both from **ifstream** and **ofstream** classes and hence can be assigned both for input and output using files. A file can be assigned to objects of **ifstream** or **ofstream**, through constructors of the respective classes. The objects point to the respective files. Hence, writing can be carried out by using the insertion operators. The `eof`() function can be used for finding out whether end of file has been reached. The file objects themselves can also be used for finding out when the end of file has been reached or whether an error condition has occurred. Files both for reading and writing can be opened with the open function. An object of class **ifstream** or **ofstream** can call the open function. Program lines are read using the `getline()` function. We can read and write on multiple files by declaring appropriate objects. The file open function has two arguments. By default, the second argument is `ios::in` if the file is opened using **ifstream,** and `ios::out` if the file is opened using **ofstream** object. By specifying the second parameter, we can open files in different modes such as append or as a binary file.

You have also learnt that C++ supports error handling. By reading the status byte through a function called `rdstate()`, we can find out the type of error such as end of file, invalid operation and even no error. The error message can be obtained through the objects **cerr** and **clog** and displayed on the monitor. The program can be made to exit by using the `exit` function. File redirection is an important, command line interface mechanism for performing specific tasks. In C++, the `cin` command is used to read and input characters from the keyboard. Similarly, we use C++ standard output commands to display on a screen. The specifications of these commands can be changed in a program by specifying the input and output redirection. The input redirection command will now input the characters from the file specified in the program and the output redirection file will send the output to the file specified in the `cout` statement. Hence, the same program can be executed and run reading either from a specified file or from the keyboard, depending upon the redirection parameter specified.

5.8 KEY TERMS

- **Stream:** It is an intermediary for I/O operations and functions between a program and I/O devices.
- **Buffer:** It provides a temporary storage space for data. It can be visualized as a fast memory device. There are member functions to manage the buffer memory.
- **Stream class:** 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 the general properties required of a stream. These 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.
- **File redirection**: It is an important and command line interface mechanism to perform specific task. The specification of these commands can be changed in a program by specifying input redirection and output redirection.

5.9 ANSWERS TO 'CHECK YOUR PROGRESS'

1. The header file **fstream** is the counterpart of the header file **iostream**. All the stream classes, including both the above, are derived from the `basic_ios` class in the standard library.
2. The output stream for writing to a file can be declared and initialized as, **ofstream** outfile (“temp”)
3. The input stream for reading from a file can be declared and initialized as, **ifstream** infile (“temp”)
4. The `eof()` function is in the `ios` class. It will return false or zero when the end of file has not reached.
5. Structures and objects can be written to a file and read from a file using the binary mode. File pointers provide a mechanism for randomly accessing the contents of the file using the `get pointer` and `put pointer` commands.

   The `get` pointer is used for input and `put` pointer for output. The file pointer functions can tell the current position of the pointers and also move them to specified locations in the file.
6. There are three categories of stream classes, one each for I/O with standard input/output, file input/output, string input/output. An object of `istringstream`

class is by default opened for reading strings. Similarly an object of `ostringstream` class is by default opened for writing. The function `str()` can be used for reading the contents of any of the objects of the string stream classes. To carryout I/O using strings, we have to `#include sstream` header file.

7. It is an important and command line interface mechanism to perform specific task. The specification of these commands can be changed in a program by specifying input redirection and output redirection. Redirection is used by the programmers because it is flexible and easy to read or write a specified file explicitly in a program. The programmers get a file name from the input redirection, i.e., command line interface, open the same and perform the required action. The output of one program can be the input for another program.

5.10 QUESTIONS AND EXERCISES

Short-Answer Questions

1. What is a stream?
2. Explain the terms buffer and buffer memory.
3. What is the advantage of stream class?
4. Define *istream* class.
5. What is the importance of header files in a programming language?
6. Explain a file redirection with suitable example.
7. Describe file input and output with the help of C++ statements.

Long-Answer Questions

1. Write short notes on the following:
    1. Standard input/output
    2. Output manipulators
    3. Formatted input / output
    4. Streams
    5. Buffers
    6. `fstream` class
    7. Opening files in multiple modes
    8. File pointers
    9. Error handling
    10. Use of eof function
    11. Binary files
    12. Random access of binary files
2. State the output of the following programs:

    1.
       ```
       #include<iostream>
       using namespace std;
       int main(){
          float fvar=123.45678;
          cout.precision(4);
       ```

```
   cout.width(10);
   cout<<fvar<<"\n";
   cout.width(6);
   cout<<fvar<<"\n";
}
```

2.
```
#include<iostream>
using namespace std;
#include<iomanip>
int main(){
   float var=16572.7856;
   cout.fill('$');
   cout<<setw(10)<<var<<endl;
   cout.setf(ios::oct, ios::basefield);
   cout.setf(ios::showbase);
   cout<<1294.67<<endl;
   cout.setf(ios::oct, ios::basefield);
   cout.setf(ios::uppercase);
   cout<<7199<<endl;
}
```

3.
```
#include<iostream>
#include<iomanip>
using namespace std;
int main(){
   float fvar=256.141414f;
   cout<<setprecision(5);
   cout<<setw(2)<<fvar<<endl;
   cout<<setw(5)<<fvar<<endl;
}
```

4.
```
#include<iostream>
#include<iomanip>
using namespace std;
int main(){
   float fvar=23456.141414f;
   cout<<setprecision(3);
   cout<<setw(10)<<fvar<<endl;
   cout.setf(ios::right, ios::adjustfield);
   cout<<setw(10)<<fvar<<endl;
   cout.setf(ios::fixed, ios::floatfield);
   cout<<fvar<<endl;
   cout.setf(ios::dec, ios::basefield);
   cout<<654129.234<<endl;
   cout.setf(ios::showpos);
   cout<<fvar<<endl;
}
```

5.
```
#include<fstream>
#include<iostream>
using namespace std;
class Book {
   int number;
   double price;
   int pages;
   public:
   void getdata(){
         cin   >>number;
         cin>>price;
         cin>>pages;
   }
   void display() {
         cout<<number<<"\n";
         cout<<price<<"\n";
         cout<<pages<<"\n";
   }
};
int main(){
   Book Ac;
   fstream iofile;
   iofile.open("Temp.dat",
ios::ate|ios::in|ios::out|ios::binary);
   cout<<"Enter Record No. to be modified";
   int rnum;
   cin>>rnum;
   int position=(rnum-1)*sizeof(Ac);
   if(iofile.eof()) iofile.clear();
   iofile.seekp(position);
   Ac.getdata();
   iofile.write((char *)&Ac, sizeof Ac)<<flush;
   iofile.seekg(0);
   cout<<"Details of updated Bank Account \n";
   while(iofile.read((char *) &Ac, sizeof Ac)){
         Ac.display();
   }
   iofile.close();
}
```
6.
```
#include<fstream>
#include<iostream>
using namespace std;
int main(){
   ofstream outf;
```

NOTES

```
   outf.open("Temp8.txt");
   char arr[]="Do these Exercises to get a good
feeling";
   int lenarr=strlen(arr);
   for (int i=0; i<lenarr; i++)
   outf.put(arr[i]);
   outf.close();
   ifstream inf;
   inf.open("Temp8.txt");
   char ch;
   while(inf){
         inf.get(ch);
         cout<<ch;
   }
   inf.close();
}
```

7.
```
#include<iostream>
using namespace std;
#include<fstream>
int main(){
   //Writing to file 1
ofstream outf1;
outf1.open("Temp6.txt");
outf1<<"It is a simpler program\n";
outf1.close();
outf1.open("Temp6.txt", ios::app);
outf1<<"You would have understood it easily \n";
outf1.close();
char buff[80];
ifstream inf;
inf.open("Temp6.txt");
while(inf){
   inf.getline(buff, 80);
   cout<<buff<<"\n";
}
inf.close();
}
```

8.
```
#include<fstream>
#include<iostream>
using namespace std;
class Account {
   private:
         int number;
         double balance;
```

NOTES

```
   public:
         Account(int num, double bal) {
               number=num;
               balance=bal;
         }
         void display() {
               cout<<"Account       number:
"<<number<<"\n";
               cout<<"balance: "<<balance<<"\n";
         }
};
int main() {
   Account Vinay(001, 10001.00);
   ofstream outfile("Temp.dat");
   outfile.write((char *)&Vinay, sizeof(Vinay));
   Account Mani(002, 5001.00);
   outfile.write((char *)&Mani, sizeof(Mani));
   Account sita(003, 6001.00);
   outfile.write((char *)&sita, sizeof(sita));
   ifstream infile("Temp.dat");
   infile.read((char *)&Mani, sizeof(Mani));
   Vinay.display();
}
```

3. Find the errors, if any, in the following statements:
    1. `cout.setprecision(4);`
    2. `cout.fill($);`
    3. `cout<<setwidth(10)<<var<<endl;`
    4. `cout.setf(ios::oct, ios::dec);`
    5. `cout.setf(ios::lowerCase);`
    6. `cout<<precision(5);`
    7. `cout.setf(ios::right, ios::basefield);`
    8. `cout.setf(ios::float, ios::floatfield);`
    9. `cout.setf(ios::showneg);`
    10. `outfile.write((char **)&sita, sizeof(sita));`
    11. ifstream infile(Temp.dat);
    12. `infile.read(char *)&Mani, sizeof(Mani));`
    13. `while(inf()){`
    14. `get(inf.ch);`
    15. `cout<<"Error state:"<<rdstate()<<"\n";`
    16. `iofile.open("Temp.dat", ios::ate&&ios: :in |ios::out|ios::binary);`
    17. `cout<<ostr.str();`

NOTES

4. Write programs for the following:
   1. To print a salary statement as given below:

Name: (15 Characters)	Designation: (5 Characters)
Employee Number: (3 digits)	Division: (5 Characters)
Basic Pay: width (8)	Precision (2), Right Justified

      DA:
      HRA:
      CCA:
      Total:
      Deductions:
      Income Tax:
      Loan:
      Net Pay:
   2. To Input/Output of US dollar consisting of $ and cent and print the output with prefix of the amount with $ sign.
   3. To Input/Output details of a bank account using overloaded insertion and extraction operators. The details are:
      Account Number
      Name
      Balance.
   4. To Input/Output transactions of a departmental stores with details as given below:

Name of the item:	Quantity:
Unit Price:	Total Price:

PUNJAB TECHNICAL UNIVERSITY

LADOWALI ROAD, JALANDHAR

INTERNAL ASSIGNMENT

TOTAL MARKS: 25

NOTE: Attempt any 5 questions
All questions carry 5 Marks.

Q. 1. Write short notes on the following:
(i) Encapsulation
(ii) Polymorphism
(iii) Inheritance

Q. 2. Write the C++ statement for calling a function and passing an argument.

Q. 3. Describe the advantage of arrays and pointers in a program.

Q. 4. Write C++ programs for the following:
(i) To print the sum of floats passed as command line arguments.
(ii) To pass a string along with the position number to a function and return the string deleted up to the position.
(iii) To pass a string along with position number up to which the string has to be printed by use of a function.

Q. 5. Write programs for the following:
(i) To create a bank account with personal details (name, address) in the base class, savings bank account details (account number, balance) with credit and debit facilities in one of the derived classes, and current account details of the same person (number, credit limit) along with credit and debit facilities in another class.
(ii) To extend the above program for linking fixed deposit details in another derived class.

Q. 6. What is the importance of header files in a programming language?

Q. 7. Describe file input and output with the help of C++ statements.

Q. 8. Explain the terms buffer and buffer memory.

Q. 9. Write a program to find p^4, by getting the value of p from a standard input device.

Q. 10. Write a program to find the difference between two numbers entered through a keyboard.

PUNJAB TECHNICAL UNIVERSITY

LADOWALI ROAD, JALANDHAR

ASSIGNMENT SHEET

(To be attached with each Assignment)

Full Name of Student:____

(First Name) (Last Name)

Registration Number:

Course:__________ Sem.:________ Subject of Assignment:________________________________

Date of Submission of Assignment:

(Question Response Record-To be completed by student)

S.No.	Question Number Responded	Pages ____ - ____ of Assignment	Marks
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

Total Marks:_____________/25

Remarks by Evaluator:____

____

Note: Please ensure that your Correct Registration Number is mentioned on the Assignment Sheet.

Name of the Evaluator

Signature of the Student

Signature of the Evaluator

Date:________________

Date:________________