Chapter 1

Familiarizing with C++ Language

Learning Outcome

After reading this chapter, students will have the ability to

- explain the evolution of C++ language
- explain the key characteristics of C++ language
- explain the general structure of C++ program
- explain the program development life cycle
- explain the steps required in building a C++ program
- explain the usage of various C++ Compilers

1.1 INTRODUCTION

The C++ programming language is a very powerful general-purpose programming language that supports procedural programming as well as object-oriented programming. It incorporates all the ingredients required for building software for large and complex problems.

The C++ language is treated as super set of C language because the developer of C++ language have retained all features of C, enhanced some of the existing features, and incorporated new features to support for object-oriented programming.

Since its inception in the early 1980's, the language underwent a number of changes and improvements.

The importance of C++ language can well be judged from the following statement:

> *"Object-Oriented Technology is regarded as the ultimate paradigm for the modeling of information, be that information data or logic. The C++ has by now shown to fulfill this goal."*

1.2 EVOLUTION OF C++ LANGUAGE

The C++ programming language is a powerful programming language that supports procedural as well as object-oriented programming and had attracted worldwide attention because the software industry had adopted the language to great advantage. One of the most important reasons for this popularity is *portability*. Portability means that a program written in it for one machine can be transferred to another machine with minimal changes or none at all. Programs written in C++ language are fast and efficient. These features make it a much sought after programming language in the highly competitive software industry. Today, C++ has become an industry standard, and therefore every computer science student and professional must have good exposure of C++ language.

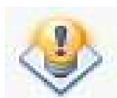
The C++ programming language was developed by *Bjarne Stroustrup* at AT&T Bell Laboratories, New Jersey, USA, in the early 1980's. He found that as the problem size and complexity grows, it becomes extremely difficult to manage it using most of procedural languages, even with C language.

He was strong admirer of Simula67 and C languages, and wanted to have a language that combines the best of both of the languages, *i.e.*, a language that support object-oriented programming and have power and elegance of C language. The outcome of his effort ultimately leads to the development of C++. Since the classes were a major addition to the original C language, he initially called the new language *'C with Classes'*. However, later in 1983, the name was changed to C++. The idea of suffixing C

with ++ came from the increment operator, since new feature are added to the features that existed and being used since long.

During the early 1990's, the language underwent a number of changes and improvements. In the year 1997, the ANSI standards committee standardized these changes and added several new features to the language specifications.

The C++ language is a superset of C. Most of what you may have learnt about C language also applies to C++ language. Therefore, all C programs are also C++ programs.



American National Standard Institute (ANSI) was founded in 1918. The main objective this institute was to suggest reform, recommend and publish standards for data processing in USA.

1.3 ANATOMY OF COMPUTER LANGUAGES

To see how C++ language compares with other programming languages, let us have a look on the programming languages in general. In the context of the present text, different programming languages can be classified according to the following criteria:

- Level of interaction with the hardware
- Way of organizing programs

1.3.1 Classification of Computer Languages depending on the Level of Interaction with the Hardware

Depending on the level of interaction with the hardware, different programming languages are classified as:

- Low-level Languages
- High-level Languages

1.3.1.1 Low-level Languages

In this category of languages, we have machine language and assembly language. These languages permit the efficient use of the machine through the features that let them interact with the hardware. But the problems with these languages are:

- These languages are hardware dependent, *i.e.*, programs written using these languages are not portable.
- Programming using these languages is not an easy job. One must have thorough knowledge of the architecture of the machine.

1.3.1.2 High-level Languages

In this category of languages, we have FORTRAN, BASIC, PASCAL, COBOL, PL/1, C, etc. These languages are designed for better programming efficiency, *i.e.*, faster program development, but most of these languages lack in features that let them interact with the hardware. These languages have following advantages:

- The syntax for writing program instructions is very much like English statements. This enables the readers to learn high-level languages (HLLs) quickly. In addition, the programs written in HLLs can be easily understood, which facilitates its maintenance.
- The programs written in HLLs are not hardware dependent. This means that program written for one machine can be transferred to another machine with minimal changes or none at all.

In procedural languages, there is another possible category of languages, called *middle-level* languages. A middle-level language is one that has possesses best of both the worlds, *i.e.*, a good programming efficiency as well as good machine efficiency. One such language is C language.

- To achieve programming efficiency, C has all the elements as of any other modern high-level language.
- To achieve machine efficiency, C has requisite features to access any hardware component of the system, to operate at register level, and to interface with high-speed assembly language routines.

1.3.2 Classification of Computer Languages depending on the way the Programs are Organized

Depending on the way the programs are organized, different programming languages are classified as:

- Procedural Languages
- Object-Oriented Languages

1.3.2.1 Procedural Languages

In this category of languages, we have FORTRAN, BASIC, PASCAL, COBOL, PL/1, C, etc. Using procedural languages, the programs are organized as set of functions/ procedures, where each function/procedure handles one or more aspect(s) of the over all problem in hand. In such programs, the entire data of the problem is distributed among these functions/procedures as local and global data. The local data is one that

is visible to a particular functions/procedure only, whereas the global data is the one that is visible to two or more functions/procedures.

The main problem with these languages is that when the size and complexity of the problem in hand grows, it becomes very difficult to manage them efficiently and effectively. Therefore, these languages are not much suitable for handling large and complex problems.

1.3.2.2 Object-Oriented Languages

In this category of languages, we have Smalltalk, Simula67, Ada, C++, Java, C#, etc. Using object-oriented languages, the programs are organized as set of objects, where each object represents a functional unit of the overall problem in hand. Further, each object carries its own data and set of functions that manipulate its data. These objects communicate with each other through messaging in manner to obtain the solution of the problem.

The entire data of the problem is localized in the functional units, and there is, generally, no place for global data that is normally the cause of most of the problems.

These languages are capable of handling problems of any size and any complexity. The real potential of object-oriented languages will be visible only when you will be working with real life problems. While learning these languages, you may not see much difference between them, the reason being that most of the people don't use these languages the way they should be used.

Since, C++ was designed and built on the top of C, it retains all the features of C, plus many more. Thus, C++ is also a middle-level language as well as procedural language. In addition, it has all the features of a object-oriented language, which is a major addition over C. Hence, C++ can be used as a procedural language as well as object-oriented language, however, its real benefits will be visible only when used as object-oriented language.

1.4 GENERAL STRUCTURE OF A C++ PROGRAM

The C++ compiler processes program units. A program unit can be compiled separately and later on linked together, without having to re-compile them, to make a single executable module. The usual order of statements in a C++ program unit is shown in Figure 1.1.

Section 1 is optional. If present it contains the description about the program. This description usually contains the information about the task being accomplished by the program. In addition, it can also contain the name of the author (programmer), the date on which it was written, the date on which it was last modified, etc.

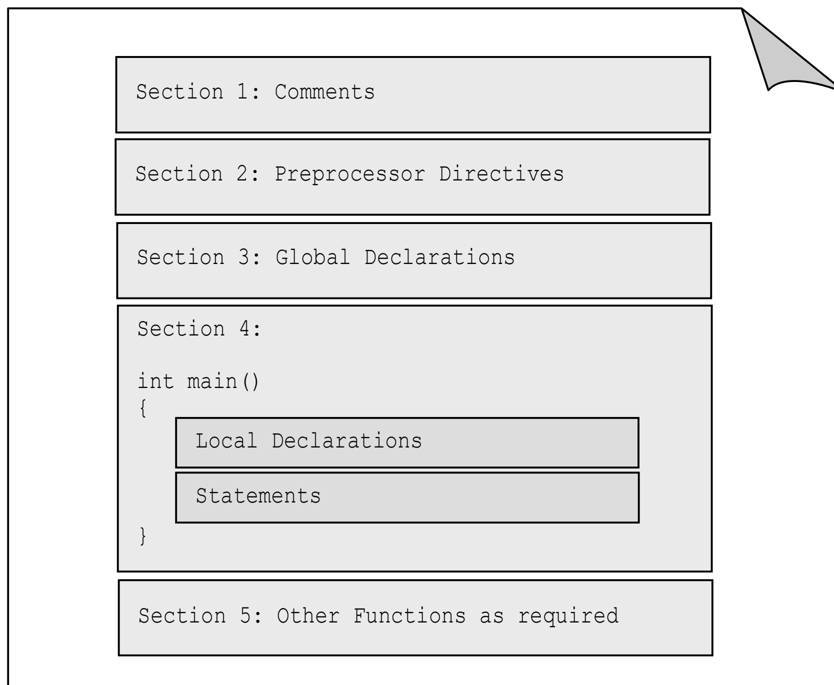


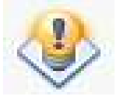
Figure 1.1: General Structure of a C++ Program

Section 2 contains the preprocessor directives. The frequently used preprocessor directives are *include* and *define*. These directives tell the preprocessor how to prepare the program for compilation. The *include* directive tells which header files are to be included in the program and the *define* directive is usually used to associate an identifier with a literal (constant) that is to be used at many places in the program.

Section 3 is optional. If present, it contains the global declarations. These declarations usually include the declaration of the data items (variables) which are to be shared between many functions in the program. In addition, these declarations can also include the decorations of functions (prototypes), except the *main()* function, to be used in the program.

Section 4 contains the *main()* function. The execution of the program always begins with the execution of the *main()* function. The *main()* function can call any number of other functions, and those called function can further call other functions. The first section in the *main()* function, as well as other functions, contains local declarations. These declarations are local in the sense that they pertain to the requirements of that function only. The second section in the *main()* function, as well as in other functions, contains the statements that defines the actions to be performed by the function.

Section 5 is also optional. If present, it contains the other functions.



If the problem to be solved is simple and small in size, then only the *main()* function is sufficient to accomplish the task. However, if the problem is complex and the size of the problem is large, it is divided into small and independent subproblems, and then we write separate functions for each subproblem. The *main()* function coordinates the execution of these functions by appropriate calls to these functions, and synthesizes the solutions of the subproblems obtained from these functions.

1.5 A SAMPLE C++ PROGRAM

In order to have a feel of the organization of a C++ program, let us consider the following sample C++ program.

Problem considered is very straightforward and familiar to you. We want to write a program that converts temperature in Celsius scale to its equivalent temperature in Fahrenheit scale. We know that the relation between temperature in Celsius and Fahrenheit is:

$$\frac{C}{100} = \frac{F-32}{180} \quad \Rightarrow \quad F = 1.8 \times C + 32$$

Listing 1.1

```
 1  //
 2  // Program to convert temperature from Centigrade scale
 3  // to Fahrenheit scale
 4  //
 5
 6  #include <iostream.h>     // preprocessor directives
 7  #include <iomanip.h>
 8
 9  int main()
10  {
11      float fahrenheit, centigrade;   // variable declaration
12      cout << "Enter temperature in Celsius scale : ";
13      cin >> centigrade;
14      fahrenheit = 1.8 * centigrade + 32;
15      cout.setf(ios::showpoint);
16      cout << setprecision(2)
17           << "Equivalent Temperature in Fahrenheit = "
18           << fahrenheit
19           << endl;
20      return 0;
21  }
```

Here the line numbers are added for ready reference.

Test Run

```
Enter temperature in Celsius scale : 30
Equivalent Temperature in Fahrenheit = 86.00
```

Dissection of the Program

Note that the entire program is written in lowercase letters. Also, it is important to remember that C++ is case sensitive, *i.e.*, it differentiates between lowercase and uppercase letters. Further, a C++ program is written in a free format, which means that an instruction can start anywhere in a line and can end anywhere. Even an instruction can span many lines. Spacing is of no consequence in C++, it is used to enhance the readability of the program.

Lines 1-4 demonstrate the one of the style used in C++ to add multiple line comments in a program. This is done by starting the comments with two characters '/*' and ending with the characters '*/'. Between these pair of characters, called *delimiters*, any number of lines can be included, which may contain characters in lowercase as well as uppercase.
In other words, a multiple line comment appears as follows:

```
/*   First comment line.
     Second comment line.
     Third comment line.
*/
```

Starting the comments with two successive slashes does the second style of adding comments ('/'). This style of commenting is preferred if the comments comprise few words or a single line. It can be used as a separate line or one the same line as that of an instruction.

```
//  include header file named 'iostream.h'
#include <iostream.h>
        or
#include <iostream.h>     //include header file named 'iostream.h'
```

Note that comments may not be essential in a simple program, but in complex programs they are lifesaver. For any program, no matter how well it is written, the day will come when we have to make amendments to incorporate the changing conditions. The chances are quite high that the original author of the program may have left the organization. Even the original author may not remember the logic of a program that he wrote several months or years ago.

In order to reduce the burden on the person, who will make amendments to the program, ample comments should be included at the time when the program is being written. Too often we postpone the task of adding comments until the program is finally Okayed, but this practice is not very useful.

Some programmers feel that C++ is such a concise and cryptic language that every line of code should be accompanied by a comment. But too many comments actually reduce the readability of the program. Still, it is better to have too many comments than few.

Line 5 is a blank line.

Lines 6-7 demonstrate the way pre-processor directives are used. These directives instruct the compiler process the source code in a specific way. In this case, *include* directive, also called *file inclusion* directive, instructs the compiler to include the specified file at this point. In this case, the specified file is a header file (a file with extension *.h*) that is part of the C++ compiler.

The other often-used pre-processor directive is #*define* whose syntax is

```
#define   MAX  100
```

This directive, called *macro expansion* directive, tells the compiler to replace every occurrence of MAX with value 100 during pre-processing. There are many other directives that we will discuss in *Appendix A*.

Line 8 is a blank line.

Line 9 specifies a function named *main*. This is a special name that is recognized by the system. It points to the precise place in the program where execution begins. Every C++ program must have a *main* function. We cannot have more than one main function in a program.

Every function has a return type associated with it. Where return type is one of the data types that determine the type of the value returned by the function.

A pair of parentheses follows the word main. The matter, if any, contained in these parentheses specifies the formal arguments of the function. If there are no arguments, we leave it empty.

Line 10 contains character '{', called left brace or left curly bracket. There is also matching right brace '}' in line 21, which appears at the end of the main function. These pair of matching braces encloses the body of the function.

Line 11 declares two variables, *fahrenheit* and *centigrade* of type *float*, which can represent and store two real numbers (float) in computer memory. Variable *centigrade* is used to hold the value, representing temperature in Celsius scale, entered by the user during program execution. Variable *fahrenheit* is to used to hold the computed value, representing temperature in Fahrenheit scale, equivalent to given temperature in Celsius scale. Note that the names of the variables used are self-explanatory. It is always recommended to use meaningful and self-explanatory names. These will help to increase readability of program.

Line 12 uses output stream object *cout* along with operator **<<**, called *insertion operator*, that inserts the contents "*Enter temperature in Celsius scale :* " into the output stream that are then displayed on the computer screen.

These kinds of messages are called *user prompts* as they guide the user to enter the desired input. In this case, value to be entered is temperature in Celsius scale.

Line 13 uses input stream object *cin* along with operator >>, called *extraction operator*, that extracts the contents from the input stream that user has entered as input from keyboard and stores in variable *centigrade* (in fact, in a memory location that is reserved for the *centigrade*).

Line 14 is an assignment statement that computes the temperature in Fahrenheit scale equivalent to given temperature in Celsius scale, and assigns to variable *fahrenheit*, *i.e.*, stores in variable *fahrenheit*.

Lines 15 specifies that real numbers in the output will be displayed with decimal point even the fractional part is zero. It uses *setf()* function is a member function of output stream class and *showpoint* is a data member of *ios* class.

Lines 16-19 uses output stream object *cout* along with cascaded insertion operators (<<) that outputs the stored value in variable *fahrenheit* along with the message "*Equivalent Temperature in fahrenheit* = ". The use of these kinds of messages is not mandatory, but is very useful as they make the output easy to interpret.

Here, *setprecision()* is a manipulator, declared in header file *iomanip.h*, and is used to specify the decimal position to displayed (here it specify two decimal positions);. And *endl* is another manipulator that causes the cursor to go to next line and its use is equivalent to '\n', called *new line* character.

The statements in line 16-19 constitute a single instruction, as you know each instruction is terminated by semicolon and that appear only in line 19. This task can also be accomplished using following statements

```
cout << setprecision(2);
cout << "Equivalent Temperature in Fahrenheit = ";
cout << fahrenheit;
cout << endl;
```

These are treated as different instructions. However, we will continue to use first style.

Line 20 marks the logical end of the *main* function with return statement which terminates the execution of the main functions and return the control back to the operating system (OS), and additionally return value 0 to indicate that program finished its execution on success, i.e., there were no failures.

Line 21 marks the end of the block of the *main* function.

1.6 CREATING, COMPILING, AND EXECUTING A PROGRAM

Once the algorithm is ready, the next step is to convert the algorithm into a computer program. During this conversion, each step of the algorithm is coded as one or more C++ language instructions. It is recommended that students should write the program first on a piece of paper before typing it into the computer.

To demonstrate the various steps, we will consider Turbo C++ Compiler, which is easy to use for beginners.

1.6.1 Creating and Editing a Program

Once the program is ready on paper, we type in computer memory using a *text editor*. A text editor helps us to enter the character data into computer memory, allows editing (changing) the data in computer memory, and saves the data from memory in a disk file to secondary memory with the extension ".*cpp*".

This stored file is known as *source file*, and its contents are known as *source code*. This source file will be the input for the compiler.

The programmer must carefully follow the C++ language rules. Violation of language rules results in grammatical errors, more precisely known as *syntax errors*. The *Compiler* will check for syntax errors. These errors must be eliminated before moving further.

1.6.2 Compiling a Program

The source code in the source file, stored on the disk, must to be translated into machine language. This job is done by the *Compiler*. The C++ compiler actually is a combination of two separate programs – the *preprocessor* and the *translator*.

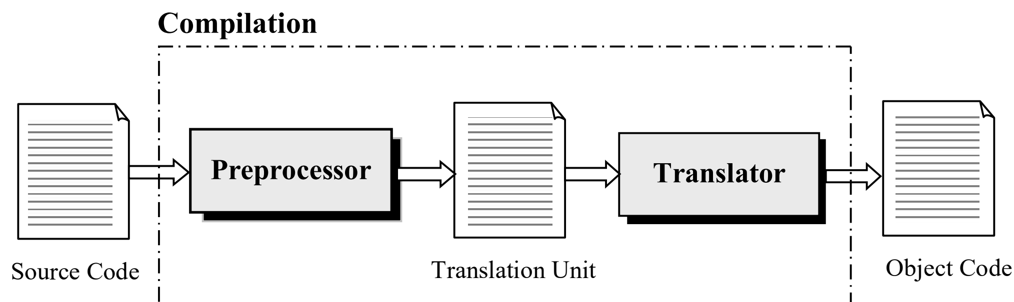


Figure 1.2: Compilation process

The *preprocessor* reads the source code and prepares it for translation. While reading the source code, it scans the code for preprocessor directives and processes them accordingly.

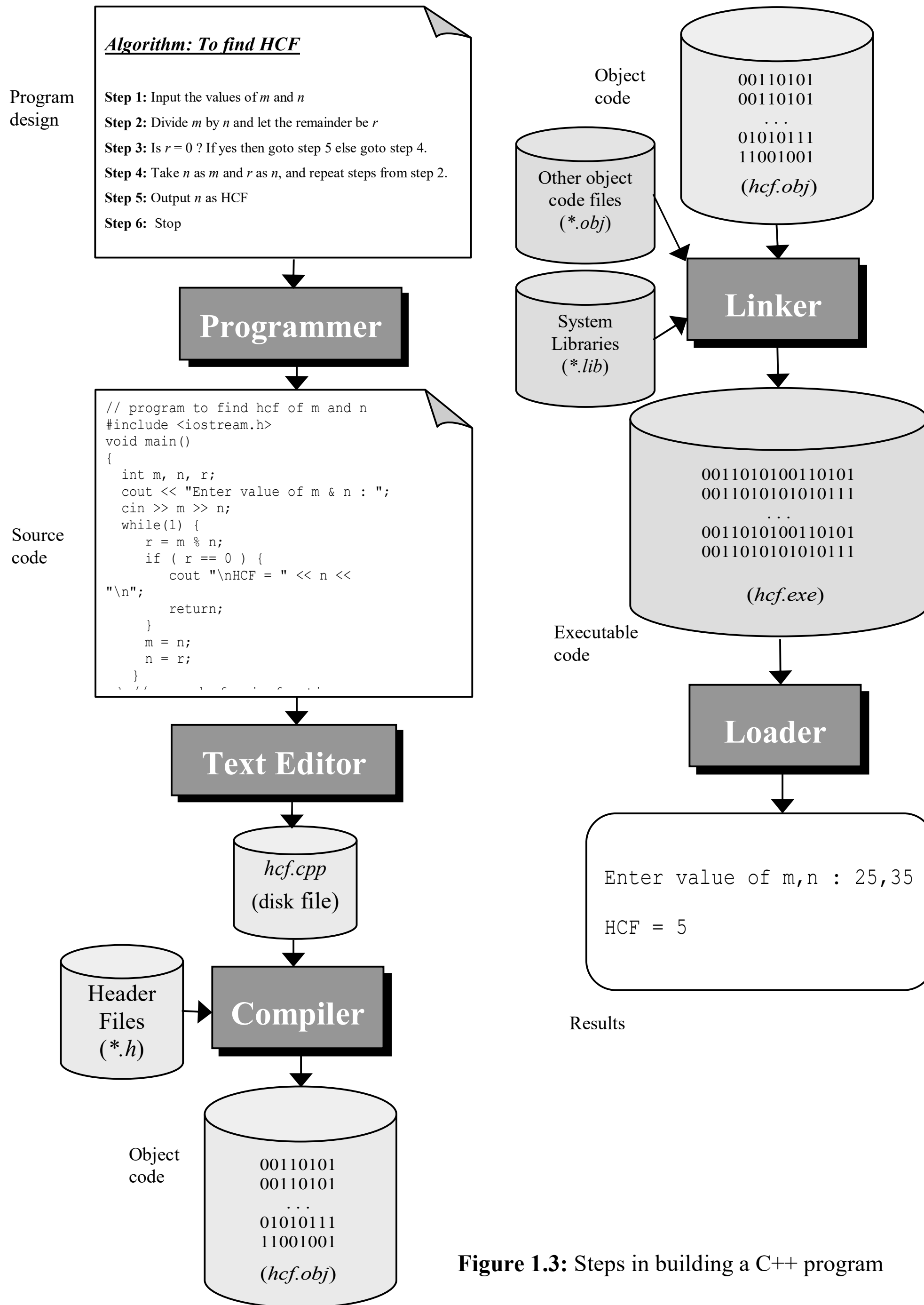


Figure 1.3: Steps in building a C++ program

For example, when it encounters the #*include* directive, it substitutes that directive with the contents of the specified header file (such as *iostream.h*), and when it encounters the #*define* directive, it substitutes the identifier with the specified literal (constant). The output from the preprocessor is an intermediate file, known as *translation unit*.

The translator reads the translation unit instruction-by-instruction and checks them for their grammatical accuracy. If there is any syntax error, it flags an error message – called *diagnostic message* on the screen. These diagnostics messages help the programmer to identify the cause of these errors and the places where they are present.

Therefore, if there is even a single syntax error, the translation process, known as *compilation*, is terminated. In this case, open the source file using the text editor, and make the necessary corrections and repeat the compilation.

However, if there are no syntax errors in the translation unit, the translator rereads the instruction from the beginning, translates them into machine language, and writes them onto a disk file. The translated version of the source code is known as *object code*, and is stored in the disk file with extension "*.*obj*".

1.6.3 Linking a Program

Once the source code is translated into object code, though it is in machine language, still it is not in executable form. The reason being is that it may be referring to library functions (pre-written functions supplied with the compiler in the form of libraries). All these functions also need to be included in the object code to get a final machine code, which is in the executable form, known as *executable code,* and that is stored in disk file with extension "*.*exe*". This executable code is the final form of the program that is ready for execution.

1.6.4 Executing a Program

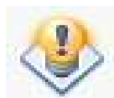
Once the program is linked, it is ready for execution. To execute a program we give an operating system command, such as *run*, to load the program into computer memory and execute it. Getting the program into memory is the function of an operating system program known as *loader*. The loader locates the executable program in the secondary storage, reads it and brings it into the computer memory. Once the program is loaded, the operating system transfers the control to the program and the program begins its execution.

All the programs either developed by you or purchased off-the self are executed under the supervision of the operating system. Remember that operating system is resource manager of the computer system, and the resources required by the executing

programs allocated by the operating system and are reclaimed when the program finishes its execution or program explicitly hands over the resources.

1.6.5 Testing the Program

Even when the program is executing, the output of the program may not be correct. This will be because of logical errors in the program. A *logical error* is a mistake that the programmer made while designing the solution to the problem. For example, a programmer tells the computer to calculate the net pay by adding deductions to the gross salary instead of subtracting. A program development tool, such as compiler, cannot detect these errors. Therefore, the programmer must find and correct logical errors by carefully examining the program output for a set of data for which results are already known. Such type of data is known as *test data*.



Syntax errors and logical errors collectively are known as *bugs*. The process of identifying and eliminating these errors is known as *debugging*.

1.7 SOME KEY POINTS TO REMEMBER ABOUT C++

Following is a list of some key points of which every C++ programmer must remember:

1. C++ language is a free form language, *i.e.*, any instructions can start from anywhere and end anywhere. Even a single instruction can span many lines.
2. Pre-processor directive can be used anywhere in the C++ program but the recommended place is in the beginning of the file.
3. Each instructions end with character semicolon ';'.
4. C++ language is a case sensitive, *i.e.*, it will treat identifiers abc, Abc, aBc, ABC, abC, etc. as different identifiers.
5. C++ program is usually written in lower case. Uppercase letters are usually used for symbolic names and predefined names.
6. Comments can be placed anywhere.
7. Execution of the program always begins with the first executable statement in the *main* function.
8. Variable can be declared anywhere in the program, but again the recommended practice is to declare them in the beginning of the body of the function or block.
9. Entire calculations involving real numbers are done in double precision mode, *i.e.*, with 12 decimal digits.

REVIEW EXERCISE . . .

1. Who developed C++ language?
2. Name popular C++ compilers available on PCs.
3. What symbol or character is used to terminate every C++ statement?
4. How is the *newline* character formed?
5. What is the effect of executing the following statement?
```
cout << "\nOne\nTwo\nThree\nFour\n";
```
6. What are different ways of adding comments in a C++ program?
7. What is the purpose of adding comments in a program?
8. Can comments span more than one line?
9. Give the general structure of a C++ program.
10. Describe the various stages in the development of a C++ program.
11. What is PDLC? Describe various phases of PDLC.
12. Describe the following - *text editor*, *compiler*, *linker*, *loader*.