UNIT 2 BASICS OF C

Structure

2.0 Introduction
2.1 Objectives
2.2 What is a Program and what is a Programming Language?
2.3 C Language
  2.3.1 History of C
  2.3.2 Salient Features of C
2.4 Structure of a C Program
  A simple C Program
2.5 Writing a C Program
2.6 Compiling a C Program
  2.6.1 The C Compiler
  2.6.2 Syntax and Semantic Errors
2.7 Link and Run the C Program
  2.7.1 Run the C Program through the Menu
  2.7.2 Run from an Executable File
  2.7.3 Linker Errors
  2.7.4 Logical and Runtime Errors
2.8 Diagrammatic Representation of Program Execution Process
2.9 Summary
2.10 Solutions / Answers
2.11 Further Readings

2.0 INTRODUCTION

In the earlier unit we introduced you to the concepts of problem-solving, especially as they pertain to computer programming**.** In this unit we present C language - a standardized, industrial-strength programming language known for its power and portability as an implementation vehicle for these problem solving techniques using computer.

A language is a mode of communication between two people. It is necessary for those two people to understand the language in order to communicate. But even if the two people do not understand the same language, a translator can help to convert one language to the other, understood by the second person. Similar to a translator is the mode of communication between a user and a computer is a computer language. One form of the computer language is understood by the user, while in the other form it is understood by the computer. A translator (or compiler) is needed to convert from user's form to computer's form. Like other languages, a computer language also follows a particular grammar known as the syntax.

In this unit we will introduce you the basics of programming language C.

2.1 OBJECTIVES

After going through this unit you will be able to:

- define what is a program?
- understand what is a C programming language?
- compile a C program;
- identify the syntax errors;
- run a C program; and
- understand what are run time and logical errors.

2.2 WHAT IS A PROGRAM AND WHAT IS A PROGRAMMING LANGUAGE?

We have seen in the previous unit that a computer has to be fed with a detailed set of instructions and data for solving a problem. Such a procedure which we call an *algorithm* is a series of steps arranged in a logical sequence. Also we have seen that a *flowchart* is a pictorial representation of a sequence of instructions given to the computer. It also serves as a document explaining the procedure used to solve a problem. In practice it is necessary to express an algorithm using a *programming language*. A procedure expressed in a programming language is known as a *computer program*.

Computer programming languages are developed with the primary objective of facilitating a large number of people to use computers without the need for them to know in detail the internal structure of the computer. Languages are designed to be *machine-independent*. Most of the programming languages ideally designed, to execute a program on any computer regardless of who manufactured it or what model it is.

Programming languages can be divided into two categories:

(i) **Low Level Languages or Machine Oriented Languages:** The language whose design is governed by the circuitry and the structure of the machine is known as the **Machine language**. This language is difficult to learn and use. It is specific to a given computer and is different for different computers i.e. these languages are **machine-dependent**. These languages have been designed to give a better machine efficiency, i.e. faster program execution. Such languages are also known as Low Level Languages. Another type of Low-Level Language is the Assembly Language. We will code the assembly language program in the form of mnemonics. Every machine provides a different set of mnemonics to be used for that machine only depending upon the processor that the machine is using.

(ii) **High Level Languages or Problem Oriented Languages:** These languages are particularly oriented towards describing the procedures for solving the problem in a concise, precise and unambiguous manner. Every high level language follows a precise set of rules. They are developed to allow application programs to be run on a variety of computers. These languages are *machine-independent*. Languages falling in this category are FORTRAN, BASIC, PASCAL etc. They are easy to learn and programs may be written in these languages with much less effort. However, the computer cannot understand them and they need to be translated into machine language with the help of other programs known as Compilers or Translators.

2.3 C LANGUAGE

Prior to writing C programs, it would be interesting to find out what really is C language, how it came into existence and where does it stand with respect to other computer languages. We will briefly outline these issues in the following section.

2.3.1 History of C

C is a programming language developed at AT&T's Bell Laboratory of USA in 1972. It was designed and written by Dennis Ritchie. As compared to other programming languages such as Pascal, C allows a precise control of input and output.

Now let us see its historical development. The late 1960s were a turbulent era for computer systems research at Bell Telephone Laboratories. By 1960, many programming languages came into existence, almost each for a specific purpose. For example COBOL was being used for Commercial or Business Applications, FORTRAN for Scientific Applications and so on. So, people started thinking why could not there be a one general purpose language. Therefore, an International Committee was set up to develop such a language, which came out with the invention of ALGOL60. But this language never became popular because it was too abstract and too general. To improve this, a new language called Combined Programming Language (CPL) was developed at Cambridge University. But this language was very complex in the sense that it had too many features and it was very difficult to learn. Martin Richards at Cambridge University reduced the features of CPL and developed a new language called Basic Combined Programming Language (BCPL). But unfortunately it turned out to be much less powerful and too specific. Ken Thompson at AT & T's Bell Labs, developed a language called B at the same time as a further simplification of CPL. But like BCPL this was also too specific. Ritchie inherited the features of B and BCPL and added some features on his own and developed a language called C. C proved to be quite compact and coherent. Ritchie first implemented C on a DEC PDP-11 that used the UNIX Operating System.

For many years the *de facto* standard for C was the version supplied with the UNIX version 5 operating system. The growing popularity of microcomputers led to the creation of large number of C implementations. At the source code level most of these implementations were highly compatible. However, since no standard existed there were discrepancies. To overcome this situation, ANSI established a committee in 1983 that defined an ANSI standard for the C language.

2.3.2 Salient features of C

C is a general purpose, structured programming language. Among the two types of programming languages discussed earlier, C lies in between these two categories. That's why it is often called a ***middle level language.*** It means that it combines the elements of high level languages with the functionality of assembly language. It provides relatively good programming efficiency (as compared to machine oriented language) and relatively good machine efficiency as compared to high level languages). As a middle level language, C allows the manipulation of bits, bytes and addresses – the basic elements with which the computer executes the inbuilt and memory management functions. C code is very portable, that it allows the same C program to be run on machines with different hardware configurations. The flexibility of C allows it to be used for systems programming as well as for application programming.

C is commonly called a structured language because of structural similarities to ALGOL and Pascal. The distinguishing feature of a structured language is compartmentalization of code and data. Structured language is one that divides the entire program into modules using top-down approach where each module executes one job or task. It is easy for debugging, testing, and maintenance if a language is a structured one. C supports several control structures such as **while, do-while and for** and various data structures such as **strucs, files, arrays** etc. as would be seen in the later units. The basic unit of a C program is a **function -** C's standalone subroutine**.** The structural component of C makes the programming and maintenance easier.

Check Your Progress 1

1. "A Program written in Low Level Language is faster." Why?

   …………………………………………………………………………………

   …………………………………………………………………………………

   …………………………………………………………………………………

2. What is the difference between high level language and low level language?
…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

3. Why is C referred to as middle level language?
…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

2.4 STRUCTURE OF A C PROGRAM

As we have already seen, to solve a problem there are three main things to be considered. Firstly, what should be the output? Secondly, what should be the inputs that will be required to produce this output and thirdly, the steps of instructions which use these inputs to produce the required output. As stated earlier, every programming language follows a set of rules; therefore, a program written in C also follows predefined rules known as syntax. C is a case sensitive language**.** All C programs consist of one or more functions. One function that must be present in every C program is **main()**. This is the first function called up when the program execution begins. Basically, **main()** outlines what a program does. Although **main** is not given in the keyword list,it cannot be used for naming a variable. The structure of a C program is illustrated in Figure.2.1 where functions func1( ) through funcn( ) represent user defined functions.

```
Preprocessor directives
Global data declarations
main ( )      /* main function*/
{
        Declaration part;

        Program statements;
}

/*User defined functions*/
func1( )
{
        …………
}

func2 ( )
{
        …………
}
.
.
.
funcn ( )
{
        …………
}
```

Figure. 2.1: Structure of a C Program.

A Simple C Program

From the above sections, you have become familiar with, a programming language and structure of a C program. It's now time to write a simple C program. This program will illustrate how to print out the message "This is a C program".

Example 2.1: Write a program to print a message on the screen.

```
/*Program to print a message*/
#include <stdio.h>                /* header file*/
main()                            /* main function*/
{
 printf("This is a C program\n");  /* output statement*/
}
```

Though the program is very simple, a few points must be noted.

Every C program contains a function called **main()**. This is the starting point of the program. This is the point from where the execution begins. It will usually call other functions to help perform its job, some that we write and others from the standard libraries provided.

#include <stdio.h> is a reference to a special file called stdio.h which contains information that must be included in the program when it is compiled. The inclusion of this required information will be handled automatically by the compiler. You will find it at the beginning of almost every C program. Basically, all the statements starting with # in a C program are called preprocessor directives. These will be considered in the later units. Just remember, that this statement allows you to use some predefined functions such as, *printf(),* in this case.

main() declares the start of the function, while the two curly brackets { } shows the start and finish of the function. Curly brackets in C are used to group statements together as a function, or in the body of a loop. Such a grouping is known as a compound statement or a block. Every statement within a function ends with a terminator semicolon (;).

printf("This is a C program\n"); prints the words on the screen. The text to be printed is enclosed in double quotes. The **\n** at the end of the text tells the program to print a newline as part of the output. That means now if we give a second printf statement, it will be printed in the next line.

Comments may appear anywhere within a program, as long as they are placed within the delimiters **/*** and ***/.** Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features. While useful for teaching, such a simple program has few practical uses. Let us consider something rather more practical. Let us look into the example given below, the complete program development life cycle.

Example 2.1

Develop an algorithm, flowchart and program to add two numbers.

Algorithm

1. Start
2. Input the two numbers ***a*** and ***b***
3. Calculate the sum as ***a+b***
4. Store the result in ***sum***

5. Display the result
6. Stop.

Flowchart

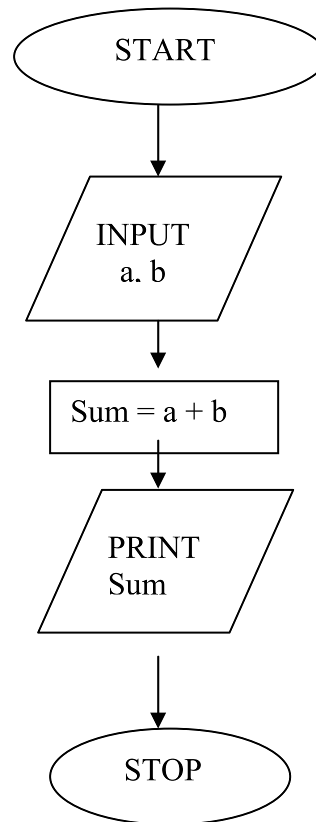

Figure 2.2: Flow chart to add two numbers

Program

```
#include <stdio.h>

main()
{
  int a,b,sum;                    /* variables declaration*/

  printf("\n Enter the values for a and b: \n");
  scanf("%d, %d", &a, &b);

  sum=a+b;

  printf("\nThe sum is %d",sum);    /*output statement*/
}
```

OUTPUT
Enter the values of a and b:
2 3
The sum is 5

In the above program considers two variables *a* and *b*. These variables are declared as integers **(int)**, it is the data type to indicate integer values. Next statement is the printf statement meant for prompting the user to input the values of ***a*** and ***b***. scanf is the function to intake the values into the program provided by the user. Next comes the processing / computing part which computes the **sum**. Again the **printf** statement is a

bit different from the first program; it includes a format specifier (%d). The format specifier indicates the kind of value to be printed. We will study about other data types and format specifiers in detail in the following units. In the printf statement above, sum is not printed in double quotes because we want its value to be printed. The number of format specifiers and the variable should match in the printf statement.

At this stage, don't go much in detail. However, in the following units you will be learning all these details.

2.5 WRITING A C PROGRAM

A C program can be executed on platforms such as DOS, UNIX etc. DOS stores C program with a file extension ***.c***. Program text can be entered using any text editor such as EDIT or any other. To edit a file called ***testprog.c*** using edit editor, gives:

C:> edit testprog.c

If you are using **Turbo C**, then Turbo C provides its own editor which can be used for writing the program. Just give the full pathname of the executable file of Turbo C and you will get the editor in front of you. For example:

C:> turboc\bin\tc

Here, tc.exe is stored in bin subdirectory of turboc directory. After you get the menu just type the program and store it in a file using the menu provided. The file automatically gets the extension of .c.

UNIX also stores C program in a file with extension is ***.c***. This identifies it as a C program. The easiest way to enter your text is using a text editor like *vi*, *emacs* or *xedit*. To edit a file called testprog.c using vi type

$ vi testprog.c

The editor is also used to make subsequent changes to the program.

2.6 COMPILING A C PROGRAM

After you have written the program the next step is to save the program in a file with extension . c . This program is in high-level language. But this language is not understood by the computer. So, the next step is to convert the high-level language program (source code) to machine language (object code). This task is performed by a software or program known as a **compiler**. Every language has its own compiler that converts the source code to object code. The compiler will compile the program successfully if the program is syntactically correct; else the object code will not be produced. This is explained pictorially in Figure 2.3.

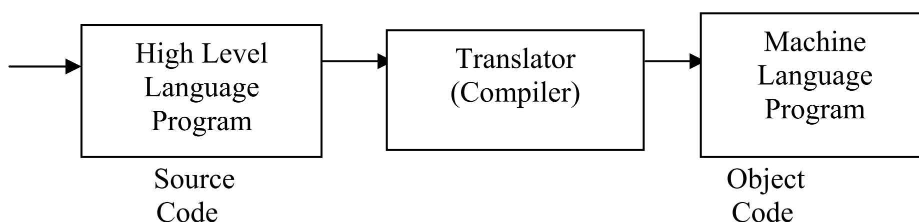

Figure 2.3: Process of Translation

2.6.1 The C Compiler

If you are working on UNIX platform, then if the name of the program file is **testprog.c**, to compile it, the simplest method is to type

cc testprog.c

This will compile testprog.c, and, if successful, will produce a executable file called ***a.out***. If you want to give the executable file any other, you can type

cc testprog.c -o testprog

This will compile ***testprog.c***, creating an executable file testprog.

If you are working with TurboC on DOS platform then the option for compilation is provided on the menu. If the program is syntactically correct then this will produce a file named as **testprog.obj**. If not, then the syntax errors will be displayed on the screen and the object file will not be produced. The errors need to be removed before compiling the program again. This process of removing the errors from the program is called as the **debugging**.

2.6.2 Syntax and Semantic Errors

Every language has an associated grammar, and the program written in that language has to follow the rules of that grammar. For example in English a sentence such a "Shyam, is playing, with a ball". This sentence is syntactically incorrect because commas should not come the way they are in the sentence.

Likewise, C also follows certain syntax rules. When a C program is compiled, the compiler will check that the program is syntactically correct. If there are any syntax errors in the program, those will be displayed on the screen with the corresponding line numbers.

Let us consider the following program.

Example 2.3: Write a program to print a message on the screen.

```
/* Program to print a message on the screen*/

#include <stdio.h

main( )
{
  printf("Hello, how are you\n")
```

Let the name of the program be **test.c** .If we compile the above program as it is we will get the following errors:

```
Error test.c 1:No file name ending
Error test.c 5: Statement missing ;
Error test.c 6: Compound statement missing }
```

Edit the program again, correct the errors mentioned and the corrected version appears as follows:

```
#include <stdio.h>
main( )
{
  printf ("Hello, how are you\n");
}
```

Apart from syntax errors, another type of errors that are shown while compilation are semantic errors. These errors are displayed as warnings. These errors are shown if a particular statement has no meaning. The program does compile with these errors, but it is always advised to correct them also, since they may create problems while execution. The example of such an error is that say you have declared a variable but have not used it, and then you get a warning "code has no effect". These variables are unnecessarily occupying the memory.

Check Your Progress 2

1. What is the basic unit of a C program?
   ……………………………………………………………………………………
   ……………………………………………………………………………………
   ……………………………………………………………………………………
2. "The program is syntactically correct". What does it mean?
   ……………………………………………………………………………………
   ……………………………………………………………………………………
   ……………………………………………………………………………………
3. Indicate the syntax errors in the following program code:

```
include <stdio.h>

main( )
[
  printf("hello\n");
]
```

   ……………………………………………………………………………………
   ……………………………………………………………………………………
   ……………………………………………………………………………………

2.7 LINK AND RUN THE C PROGRAM

After compilation, the next step is linking the program. Compilation produces a file with an extension **.obj**. Now this **.obj** file cannot be executed since it contains calls to functions defined in the standard library (header files) of C language. These functions have to be linked with the code you wrote. C comes with a standard library that provides functions that perform most commonly needed tasks. When you call a function that is not the part of the program you wrote, C remembers its name. Later the linker combines the code you wrote with the object code already found in the standard library. This process is called *linking.* In other words, Linker is a program that links separately compiled functions together into one program. It combines the functions in the standard C library with the code that you wrote. The output of the linker in an executable program i.e., a file with an extension ***.exe***.

2.7.1 Run the C Program Through the Menu

When we are working with TurboC in DOS environment, the menu in the GUI that pops up when we execute the executable file of TurboC contains several options for executing the program:

i) Link , after compiling
ii) Make, compiles as well as links
iii) Run

All these options create an executable file and when these options are used we also get the output on user screen. To see the output we have to shift to user screen window.

2.7.2 Run From an Executable File

An ***.exe*** file produced by can be directly executed.

UNIX also includes a very useful program called **make**. **Make** allows very complicated programs to be compiled quickly, by reference to a configuration file (usually called makefile). If your C program is a single file, you can usually use make by simply typing –

make testprog

This will compile **testprog.c** as well as link your program with the standard library so that you can use the standard library functions such as printf and put the executable code in **testprog.**

In case of DOS environment , the options provided above produce an executable file and this file can be directly executed from the DOS prompt just by typing its name without the extension. That is if the name of the program is test.c, after compiling and linking the new file produced is test.exe only if compilation and linking is successful.

This can be executed as:

c>test

2.7.3 Linker Errors

If a program contains syntax errors then the program does not compile, but it may happen that the program compiles successfully but we are unable to get the executable file, this happens when there are certain linker errors in the program. For example, the object code of certain standard library function is not present in the standard C library; the definition for this function is present in the header file that is why we do not get a compiler error. Such kinds of errors are called linker errors. The executable file would be created successfully only if these linker errors are corrected.

2.7.4 Logical and Runtime Errors

After the program is compiled and linked successfully we execute the program. Now there are three possibilities:

1) The program executes and we get correct results,
2) The program executes and we get wrong results, and
3) The program does not execute completely and aborts in between.

The first case simply means that the program is correct. In the second case, we get wrong results; it means that there is some logical mistake in our program. This kind of error is known as **logical error**. This error is the most difficult to correct. This error is corrected by debugging. Debugging is the process of removing the errors from the program. This means manually checking the program step by step and verifying the results at each step. Debugging can be made easier by a tracer provided in Turbo C environment. Suppose we have to find the average of three numbers and we write the following code:

Example 2.4: Write a C program to compute the average of three numbers

```
/* Program to compute average of three numbers *?
#include<stdio.h>
```

```
main( )
 {
   int a,b,c,sum,avg;

   a=10;
   b=5;
   c=20;

   sum = a+b+c;
   avg = sum / 3;
   printf("The average is %d\n", avg);
}
```

OUTPUT

The average is 8.

The exact value of average is 8.33 and the output we got is 8. So we are not getting the actual result, but a rounded off result. This is due to the logical error. We have declared variable **avg** as an integer but the average calculated is a real number, therefore only the integer part is stored in **avg.** Such kinds of errors which are not detected by the compiler or the linker are known as **logical errors**.

The third kind of error is only detected during execution. Such errors are known as **run time errors**. These errors do not produce the result at all, the program execution stops in between and the run time error message is flashed on the screen. Let us look at the following example:

Example 2.5: **Write a program to divide a sum of two numbers by their difference**

```
/* Program to divide a sum of two numbers by their difference*/

#include <stdio.h>

 main( )
 {

   int a,b;
   float c;

   a=10;
   b=10;

   c = (a+b) / (a-b);
   printf("The value of the result is %f\n",c);
}
```

The above program will compile and link successfully, it will execute till the first *printf* statement and we will get the message in this statement, as soon as the next statement is executed we get a runtime error of "Divide by zero" and the program halts. Such kinds of errors are **runtime errors**.

2.8 DIAGRAMMATIC REPRESENTATION OF PROGRAM EXECUTION PROCESS

The following figure 2.4 shows the diagrammatic representation of the program execution process.

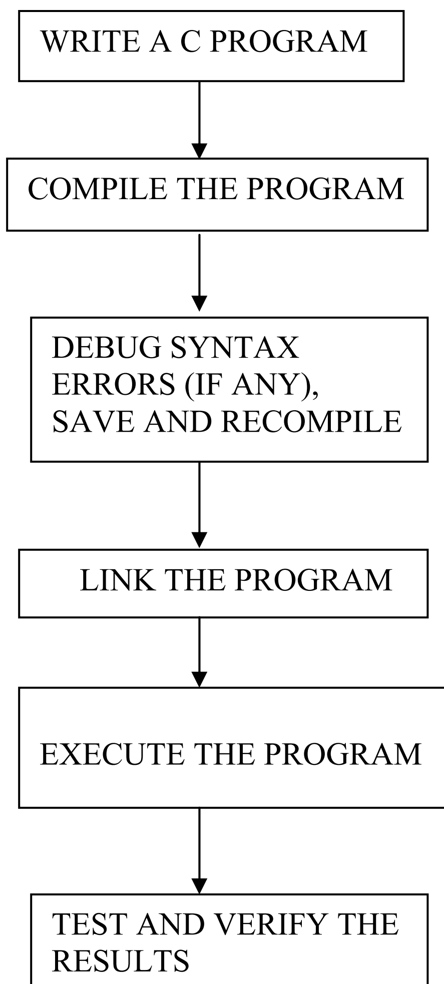

Figure 2.4: Program Execution Process

Check Your Progress 3

1. What is the extension of an executable file?

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

2. What is the need for linking a compiled file?

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

3. How do you correct the logical errors in the program?

   ……………………………………………………………………………………

   ……………………………………………………………………………………

   ……………………………………………………………………………………

2.9 SUMMARY

In this unit, you have learnt about a program and a programming language. You can now differentiate between high level and low level languages. You can now define what is C, features of C. You have studied the emergence of C. You have seen how C

is different, being a middle level Language, than other High Level languages. The advantage of high level language over low level language is discussed.

You have seen how you can convert an algorithm and flowchart into a C program. We have discussed the process of writing and storing a C program in a file in case of UNIX as well as DOS environment.

You have learnt about compiling and running a C program in UNIX as well as on DOS environment. We have also discussed about the different types of errors that are encountered during the whole process, i.e. syntax errors, semantic errors, logical errors, linker errors and runtime errors. You have also learnt how to remove these errors. You can now write simple C programs involving simple arithmetic operators and the *printf( )* statement. With these basics, now we are ready to learn the C language in detail in the following units.

2.10 SOLUTIONS / ANSWERS

Check Your Progress 1

1. A program written in Low Level Language is faster to execute since it needs no conversion while a high level language program need to be converted into low level language.

2. Low level languages express algorithms on the form of numeric or mnemonic codes while High Level Languages express algorithms in the using concise, precise and unambiguous notation. Low level languages are machine dependent while High level languages are machine independent. Low level languages are difficult to program and to learn, while High level languages are easy to program and learn. Examples of High level languages are FORTRAN, Pascal and examples of Low level languages are machine language and assembly language.

3. C is referred to as middle level language as with C we are able to manipulate bits, bytes and addresses i.e. interact with the hardware directly. We are also able to carry out memory management functions.

Check Your Progress 2

1. The basic unit of a C program is a C function.

2. It means that program contains no grammatical or syntax errors.

3. Syntax errors:

   a) # not present with include
   b) {brackets should be present instead of [brackets.

Check Your Progress 3

1. The extension of an executable file is .exe.

2. The C program contains many C pre-defined functions present in the C library. These functions need to be linked with the C program for execution; else the C program may give a linker error indicating that the function is not present.

3. Logical errors can be corrected through debugging or self checking.

2.11 FURTHER READINGS

1. The C Programming Language, *Kernighan & Richie*, PHI Publication.
2. Programming with C*,* Second Edition, *Byron Gottfried*, Tata Mc Graw Hill, 2003.
3. The C Complete Reference, Fourth Editon, *Herbert Schildt*, Tata Mc Graw Hill, 2002.
4. Programming with ANSI and Turbo C, *Ashok N. Kamthane*, Pearson Education Asia, 2002.
5. Computer Science A structured programming approach using C Second Edition, *Behrouza A. Forouzan, Richard F. Gilberg*, Brooks/Cole, Thomson Learning, 2001.