UNIT 1 GETTING STARTED

Structure **Page No.**

1.1 INTRODUCTION

In this unit we will introduce you to the C programming language. We begin this unit with an overview of the C programming language, its history and the reasons for its continued popularity in Sec. 1.2.

In Sec. 1.3, we start our discussion of the C programming language with a simple example program. We use the program to tell you how a C program is organised in general.

In Sec. 1.4, we discuss escape sequences which output tab, new line etc. In the last section, we discuss some example C programs to give a feel for the language.

Objectives

After studying this unit, you should be able to

- explain the advantages and disadvantages of C language;
- explain the general structure of C programs; and
- explain the purpose of escape sequences.

1.2 AN OVERVIEW

C is a general purpose computer programming language. It was originally created by Dennis M. Ritchie(circa 1971) for the specific purpose of rewriting much of the **Unix** operating system. This now famous operating system was at that time in its infancy; it was first written for a discarded DEC PDP-7 computer by Ken Thompson, an electrical engineer and a colleague of Ritchie's at Bell Labs. Thompson had written the first Unix in a language he chose to call **B**. **B** itself was based on **BPCL** a language that was developed by Martin Richards, another programmer.

Dennis M. Ritchie

The intention of these programmers was to port **Unix** on to other, architecturally dissimilar machines. Clearly, using any assembly language was out of the question; what they needed was a language that would permit assembly-like (i.e. low or hardware level) operations, such as bit manipulation. At the same time it should be usable on different computers. None of the languages then available served the purpose; they had to create a new one, and C was born. The rest is history; **Unix** became spectacularly successful, in large measure because of its portability. C, in which of most Unix was written, has occupied a pre-eminent position in the development of system programs.

The success enjoyed by **Unix**, and the consequent popularity of C for systems programming, forced it on the attention of applications programmers, who came to

appreciate the rich variety of its **operators** and its **control structures**. These enable, and in fact encourage, the practice of modular programming: The individual sub tasks that make up a large program can be written in independent blocks or modules, each of manageable size. In other words, the language possesses features that make possible a "divide and conquer" approach to large programming tasks. When programs are organised as modules they are necessarily well-planned, because each module contains only the instructions for a well-defined activity. Such programs are therefore more easily comprehensible by other readers than unstructured programs.

Another desirable quality of modular programs is that they may be modified without much difficulty. If changes are required, only a few modules may be affected, leaving the remainder of the program intact. And last but not least, such programs are easier to debug, because errors can be localised to modules, and removed. For a concrete example, suppose that you have to write a program to determine all prime numbers up to ten million which are of the form $k^2 + 1$, where k is an integer.(2, 5, 17 and 37 are some examples of such primes.) Then you can write one module to generate the number $k^2 + 1$ for the next admissible value of k; another module to test whether the last number generated is a prime. Organised in this way, not only will the program be elegant, it would also be easy to debug, understand or modify.

Yet, C is not rigidly structured. The language allows the programmer to forsake the discipline of structured programming, should he or she want to do so.

Another point of difference between C and other languages is in the wealth of its operators. Operators determine the kinds of operation that are possible on data values. The operators of C impart a degree of flexibility and power over machine resources which is unequalled by few other contemporary language, except assembly language.

But assembly language has disadvantages; it is not structured; it is not easy to learn; moreover, assembly programs for any substantive application tend to be too long; and they are not **portable**: An assembly language program written for one computer cannot be executed on another with a different architecture and instruction set. On the other hand, a C program created on one computer can be compiled and run on any other machine that has a C compiler.

Possibly the greatest merit of C lies in its intangible quality which can only be termed elegance. This derives from the versatility of its control structures and power of its operators. In practical terms this often means that C programs can be very short, and it must be admitted, sometimes even cryptic.(Indeed, there is a canard that C stands for cryptic!) While brevity is undeniably a virtue(specially when you consider the vastness of COBOL programs!)there can be too much of it. All too frequently in C programs one comes across craftily constructed statements that take as much time to comprehend, as they must have taken to create. In the interest of other readers therefore(as well as, you may find, in your own)you should resist that temptation to be cryptic that C by its nature seems to encourage.

One common classification of computer language divides them into two broad categories: high-level languages(HLLs) and low-level languages(LLLs). An HLL is a language that is easy for a human beings to learn to program in. A good example is BASIC, with its easily understood statements:

```
10 LET X=45
20 LET Y=56
30 LET Z=X=+Y
40 PRINT X,Y,Z
50 END
```

An LLL is a language that is closely related to machine or assembly language; it allows a programmer the ability to exploit all of the associated computer's resources. This

power however does not come without a price: LLLs are generally difficult to learn, and are usually so closely tied to the architecture of the host machine that even a person who is familiar with one LLL will have to invest a great deal of effort in learning other. Consider the following statement from the MACRO Assembly Language of VAX series of computers manufactured by the Digital Equipment Corporation, USA:

```
POPR#^M<R6,R8,R3>
```

It is highly unlikely that one who is not familiar with VAX's MACRO Assembler will be able to fathom any meaning from this instruction. Moreover, LLL programs are not portable; they cannot be moved across machines with different architectures.

C is a considerable improvement on LLLs: it's easy to learn; it's widely available(from micros to mainframes); it provides almost all the features that assembly does, even bit-level manipulation(while being fairly independent of the underlying hardware). It also has the additional merit that its programs can be concise, elegant and easy to debug. Finally, because there is a single authority for its definition, there is a greater degree of standardisation and uniformity in C compilers than for other HLLs.

It has been often said with some justification that C is the FORTRAN of systems software. Just as FORTRAN compilers liberated programmers from creating programs for specific machines, the development of C has freed them to write system software without having to worry about the architecture of the machine that the software is intended for.(Where architecture-dependent code i.e. assembly code is necessary, it can usually be invoked from within the C environment.)C is a middle level language, not as low-level as assembly, and not as high level as BASIC.

In short, C has all the advantages of assembly language, with none of its disadvantages; and it has all the significant features of modern HLLs. It's an easy language to program in, and makes programming great fun.

As we mentioned in the course introduction, we will discuss ANSI C, also known as C89. We will also mention some extra features provided by C99.

We close this section here. In the next section, we will examine some simple C programs to understand the structure of a C program.

1.3 A C PROGRAM

The best way to learn C or any programming language is to begin writing programs in it; so here's our first C program.(See the practical guide for instructions on how to compile and link programs.)

```
/* Program 1; file name: unit1-prog1.c */
#include<stdio.h>
int main(void)
{
    printf("This is easy!!\n");
    return 0;
}
```

Listing 1: First C program.

There are a few important points to note about this program, points which you will find common to all C programs.

Programs in C consist of **functions**, one of which must be `main()`. When a C program is executed, `main()` is where the action starts. Then other functions may be "invoked". A function is a sub-program that contains instructions or statements to perform a

specific computation on its variables. When its instructions have been executed, the function returns control to the calling program, to which it may optionally be made to return the results of its computations. Because `main()` is a function, too, from which control returns back to the operating system at program termination, in **ANSI C** it is customary, although not required, to include a statement in `main()` which explicitly returns control to the operating environment. Also, `main()` returns an integer value to the operating system. So, we have added the key word `int` before `main()` statement.

C99

The int key word before `main()` is not needed in ANSI C because a function without an explicit return type is assumed to return an integer value. This is not so in C99. If no return type is given, C99 compilers may compile with a warning(as it happens with gcc), but it is not compulsory to do so; the program may also fail to compile.

We'll learn more about functions as we go along, but for now you may recognise them by the presence of parentheses after their names.

Apart from `main()`, another function in the program above is the `printf()`. `printf()` is an example of a "library function". It's provided by the developers of the C library, ready for you to use. C library provides a variety of functions. For example, C library provides many common mathematical functions like the trigonometric functions, exponential function etc. Apart from this, you may use other libraries that provide, for example, graphics functions. These libraries contain the instructions for the function in a compiled form or in the form of object code. So, you do not have to write code in your program for these functions. To enable the use of such libraries, the compilation of C programs is a two part process. In the first part, the program compiles your C file into an object file. The C compiler 'remembers' the functions you have called and the linker combines the translated version of the code you wrote with the object code already found in the library. The second step of the process is called *linking.*

In addition to its variables, a function, including `main()`, may optionally have **arguments**, which are listed in the parentheses following the function name. The arguments of functions are somewhat different from the arguments that people have: they are values of the **parameters** in terms of which the function is defined, and are passed to it by the calling program. The instructions which comprise a function are listed in terms of its parameters and its variables.

In the example above, `main()` has no arguments. `printf()` has one: it's the bunch of characters(more properly: **string**) enclosed in double quotes:

```
"This is easy!!\n"
```

When the `printf()` is executed, its built-in instructions process this argument. The result is that the string is displayed on the output device, which we will usually assume is a terminal. The output of the program will be:

```
This is easy!!
```

with the cursor stationed at the beginning of the next line on the screen. In C a string is not a piece of thread: it's any sequence of characters enclosed in double quotes. A string may not include a "hard" carriage-return (<CR>), i.e. all its characters must be keyed in on a single line on the terminal.

```
"This is most certainly not a C string. <CR>
It contains a carriage return Character."
```

You usually put a carriage return by pressing the 'enter' key.

As we have seen in the case of `printf()`, when a function is invoked, it is passed (the values of) its arguments. It then executes its instructions, using these values. On completion the function returns control to the module from which it was invoked. The nice thing about a function is that it may be called as often as required in program, to

perform the same computational steps on the different sets of values that are passed to it. Thus there can be several invocations to `printf()` within a program, with different string arguments each time. Also, a function can call other functions. For example, in Listing 1 on page 9, the function main() calls the function printf() with the text 'This is easy!!' as the argument. In a program for listing primes of the form $k^2 + 1$ we can have a function that computes $k^2 + 1$ for the next value of k passed to it; another function can test whether the value of $k^2 + 1$ calculated by the first function is a prime or not. These functions could be invoked repeatedly, for different values of k. Functions will be formally introduced in the next Block.

With some compilers you may not require the **preprocessor directive:**

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

which we have written just before `main()` in Listing 1 on page 9. These directives, unlike other C statements, are **not terminated** by a semicolon.

Preprocessing is a phase of compilation that is performed prior to the analysis of the program text. We will have more to say about preprocessor directives later.

`stdio.h` is a **header file** that comes with your C compiler and runtime system and contains data that `printf()` needs in order to output its argument. The #include directive causes the inclusion of external text—in this case the file `stdio.h` —in your sources program before the actual compilation proceeds. We'll have more to say about header files later. If you have problems, consult a resident expert!

Let's go back to Listing 1 on page 9. Observe that the body of the program is enclosed in braces {}; a pair of braces defines a **block**. A program may consist of several blocks, which may include yet other blocks, and so on. The left and right braces which mark a block may be placed anywhere on a line. You will find it easier to read and debug the programs if you align the same column. For the same reason the braces of blocks which are nested inside other blocks are indented a few spaces to the right from the braces of the enclosing block.

```
/* Program 2; file name:unit1-prog2.c  */
#include<stdio.h>
int main()
{
    printf("This is the outermost block.\n");
    {
        printf("This is a nested block.\n");
        {
            printf("This block is nested still more deeply.\n");
            {
                printf("This is the innermost block.\n");
            }
        }
    }
    return (0);
}
```

Listing 2: Block structure.

In Listing 2, note that each left brace is balanced by a corresponding right brace. Blocks contain **statements** such as:

```
printf("This is the innermost block.\n");
```

C statements invariably end with a semicolon. C statements may be of any length and may begin in any column. Each statement may extend over several lines, as long as any included strings are not broken. In C a semicolon by itself constitutes a valid statement, the **null** statement. Null statements are often very handy, and we will quite frequently have occasion to use them.

While writing a C program, you can add comments, which are ignored by the compiler. Comments in programs are included between a backslash-asterisk pair, `/**/`:

```
/* This is a comment about comments.
   Comments may extend over many
 lines, but as a rule they should
          be short.*/
```

Comments are important because they help document a program, and should be written so that its logic becomes transparent. They may be placed wherever the syntax allows "white space" characters: blanks, tabs or newlines.

E1) Do you think comments can be nested, i.e. can you have a comment within a comment? Write a program with a nested comment and see if you can compile it.

In discussion above, we mentioned that a C string cannot contain a carriage return. What should we do if want to break a long string? We have to use an *escape sequence.* Escape sequences are the topic of our discussion in the next section.

1.4 ESCAPE SEQUENCES

You might be wondering about the `\n` (pronounced backslash n) in the string argument of the function `printf():`

```
"This is easy!!\n"
```

The `\n` is an example of an **escape sequence:** it's used to print the **newline** character. If you have executed Programs 1 or 2 you will have noticed that the `\n` doesn't appear in the output. Each `\n` in the string argument of a `printf()` cause the cursor to be placed at the beginning of the next line of output. Think of an escape sequence as a "substitute character" for printing hard-to-get characters. In an earlier section we learnt that < CR >s are not allowed in strings; but including `\n`'s within one makes it easy to output it in several lines: if you wish to print a string in two or more lines, place the `\n` wherever you want to insert a new line in the output, as in the example below:

```
/* Program 3; file name: unit1-prog3.c */
void main()
{
    printf("This\nstring\nwill\nbe\nprinted\nin\n9\nlines.\n");
}
```

Listing 3: Line breaks in a C program.

Each `\n` will force the cursor to be positioned at the beginning of the next line on the screen, from where further printing will begin. Here is part of the output from executing the above program:

```
This
string
will
etc ...
```

If a string does not contain a `\n` at its end, the next string to be printed will begin in the same line as the last, at the current position of the cursor, i.e. alongside the first string.

Though < CR >s cannot be included in a string, if you must deal with a long string of characters that cannot fit conveniently in a single line, there is a way of continuing it into the next line. You can insert a backslash(\) followed by a < CR > where you wish to break the string:

```
"This is a rather long string of characters.  It extends \
                    over two lines."
```

We have seen that the "double quotes" character is used to mark the beginning and end of a C string. How can the "double quotes" character itself be included within a string? This time the escape sequence\" comes to our aid: it prints as " in the output. Similarly, the escape sequence \\ helps print \, and \' the single quote character, '. All escape sequences in C begin with a backslash.

Here are some exercises to help you check your understanding of escape sequences.

E2) Give the output of the following program:

```
/* Program 4; file name: unit1-prog4.c  */
#include<stdio.h>
int main()
{
    printf("This is the first line of output.");
    printf("But is this the second\nline of output?");
    return (0);
}
```

E3) Execute the program below and obtain the answer to the question in its printf():

```
/* Program 5; file name: unit1-prog5.c */
#include<stdio.h>
int main()
{
    printf("In how many lines will the output\
 of this program be printed?");
    return (0);
}
```

E4) Give the output of the following programs:

a)
```
/* Program 6; file name: unit1-prog6.c */
#include<stdio.h>
int main()
{
    printf("\"A\\n\",the teacher said,\"is used to\n ");
    printf("insert a new line in a C string.\"");
    printf("\n\"IC,  IC\", said the blind student.\n");
    return (0);
}
```

b)
```
/* Program 7; file name: unit1-prog7.c */
#include<stdio.h>
int main()
{
  printf("To be, or not to be,--");
  printf("that is the question:--\nWhether \'tis ");
  printf("nobler in the mind to suffer\nThe slings ");
  printf("and arrows of outrageous fortune\nOr to ");
  printf("take arms against a sea of troubles,\nAnd ");
  printf("by opposing end them?--To die,--to sleep,--\n");
  return (0);
}
```

E5) Write a C language program that gives for its output:

```
/* This is a C comment.  */
```

E6) Debug the following program:

```
#include[stedio.h]
Main {
}
(print("print this\m" \ *very easy * \).)
```

Be careful to remember that "" is **not** the escape sequence for the "double quote" character (as it is in some versions of BASIC). In C ""is the **null string;** the null string is not "empty", as one might have thought; it contains a single character, the **null character**, ASCII 0 (in which all bits are set to zero). We will see later that the last character of every C string is the (invisible) null character. Its presence helps the computer determine the actual end of the string. Other escape sequences available in C are:

\a	Ring terminal bell (the **a** is for **alert**) [ANSI] extension]
\?	Question mark [ANSI extension]
\b	Backspace
\r	Carriage return
\f	Formfeed
\t	Horizontal tab
\v	Vertical tab
\0	ASCII null character

Placing any of these within a string causes either the indicated action, or the related character to be output. In the next section, we will see some more C programs that will give you a better idea about the structure of C program.

1.5 GETTING A 'FEEL' FOR C

In this section we present a few programs that involve concepts which have not so far been discussed; they'll be covered at a leisurely pace by and by. As you read these programs, you may discover that many of their statements are self-evident; those that may not be are commented. Create these programs on your computer, compile and execute them, and by the time you come to read about the features used in the programs, you shall have a fair idea about them already.

The major change between **ANSI C** and the language described in the first edition of **K & R** is in the declaration and definition of functions. Program 1.11 below is an **ANSI C** compliant program, but it may not run on your machine if you have a non-ANSI compiler. Program 1.12 is the same program modified to run on Classic C.

So far, all the programs that we have written do the same thing every time we run the program; they print some text, the same text every time. But, we would like to write programs that do different things at different times. For example, suppose we want a program that adds two numbers and gives us the answer. How do we do it? For such programs we need programs that use *variables.* As the name suggests, variables store values that can change during the run of the program. The next program uses two values, 5 and 7 and stores them in two variables called x and y. If you want the program to do all the various operations for different integers, say 11 and 12, you can edit the source file, change the values of x and y, compile and run the program again. Go ahead! Try out two three different values for x and y. Notice the int key word before the variables x and y. This indicates that x and y will be used to store integer values.

```
/* Program 8; file name:unit1-prog8.c */
#include<stdio.h>
int main()
{
```

```
    /* Elementary operations with small integers */
    /* C calls small integers ints */
    int x = 5, y = 7, z;
    /* x,y, and z are int variables x is 5,y is 7,
       z doesn't have a value yet; */
    printf("x=%d, y=%d\n", x, y);
    /* each %d prints a decimal integer. */
    z = x - y;
    /* now z is x minus y; */
    printf("z=x-y=%d\n", z);
    z = x * y;
    /*the * means "multiplied by";  */
    printf("z=x*y=%d\n", z);
    z = x / y;
    /* one int divided by another;  */
    printf("z=x/y=%d\n", z);
    z = y / x;
    printf("z=y/x=%d\n", z);
    z = x % y;
    /* guess from the output what
       the % in x % y stands for;  */
    printf("z=x%%y=%d\n", z);
    /* To print percent sign
       we use the escape sequence %% */
    z = y % x;
    printf("z=y%%x=%d\n", z);
    return (0);
}
```

Now, that wasn't very interesting, isn't it? Will it not be better if the programs asks us for values of x and y when it is run? We have given below such a program.

```
/* Program 9; file name: unit1-prog9.c */
#include<stdio.h>
int main()
{
    /* Read values from the keyboard,
       see how scanf () works */
    int x, y, z;
    printf("Enter a value for x.\n");
    printf("Type a samll integer, press<CR>:");
    scanf("%d", &x);
    /* mind that ampersand "&",
       just before x; */
    printf("Enter a value for y:");
    scanf("%d", &y);
    z = x * y;
    printf("z=x*y=%d\n", z);
    return (0);
}

/* Program 10; file name: unit1-prog10.c */
#include<stdio.h>
int main()
{
    int x, y;
    printf("Enter a value for x:");
    scanf("%d", &x);
    printf("Enter a value for y:");
    scanf("%d", &y);
    /* Is x greater than y? Then say so: */
    if (x > y)
        printf("x is greater than y.\n");
    /* else, if it's not, deny it. */
```

```
    else
        printf("x is not greater than y.\n");
    /* The computer can tell if one
       number is greater than another.*/
    return (0);
}

/* Program 11 file name:unit1-prog11.c */
#include<stdio.h>
int addtwo(int x, int y); /* A function that adds two ints. */
int main()
{
    int val_1, val_2, sum;
    printf("Enter a number:");
    scanf("%d", &val_1);
    printf("Enter another:");
    scanf("%d", &val_2);
    printf("\n\nWill let the function addtwo()\
add them...\n");/*Continued*/
    sum = addtwo(val_1, val_2);
    /* transfer control to addtwo (),
       with arguments val_1 and val_2  */
    printf("\nNow we\'re back in main ()...\
What have we here?\n");/*Continued*/
    printf("\naddtwo () tells us their sum is %d.\n", sum);
    return (0);
}
int addtwo(int p, int q)
{
    printf("\n\nNow I'm reporting from inside addtwo ()...\n");
    printf("\nThe numbers you typed \
did reach here...%d and %d\n", p, q);   /*Continued */
    printf("\ntheir sum is...am working on it...\n");
    return (p + q);
}
```

Here is a C program that solves a quadratic equation using the well known formula. Notice the line

```
#include<math.h>
```

in the program. It loads the definitions of maths library functions. We need this line because we are using the `sqrt()` function which gives the square root and `fabs()` that gives the absolute value of a number. This program uses the data type called **float** to store the absolute value of the coefficients and the discriminant because they may be numbers with a decimal point in them. We will discuss **float** date type in the next Unit.

```
/* Program to solve a quadratic equation;*/
/*File name:unit1-quad.c */
#include <math.h>
#include <stdio.h>
int main()
{
    float a, b, c, disc;/*Use decimals.*/
    printf("This program solves the\n the \
quadratic equation ax^2+bx+c=0\n");/*Continued*/
    /*Prompt for the coefficient
    of the second degree term.*/
    printf("Enter the value of a\n");
    /*Read the coefficient of
    the second degree term*/
    scanf("%f", &a);
    /*Stop with an error message if it is 0.*/
    if (a == 0) {
```

```
        printf("Value of a should not be zero. Exiting\n");
        return 0;
    };
    /*If a is not zero, prompt for other
    coefficients and read them in.*/
    printf("Enter the value of b\n");
    scanf("%f", &b);
    printf("Enter the value of c\n");
    scanf("%f", &c);
    printf("You entered a=%f,b=%f,c=%f", a, b, c);
    /*Find the discriminant*/
    disc = b * b - 4 * a * c;
    printf("\n The discriminant is %f\n", disc);
    if (disc == 0) {
        printf("This equation has repeated roots.\n");
        printf("The root is %f with multiplicity 2.", -b / (2 * a));
    }
    if (disc > 0) {
        printf("The roots are real.\n");
        printf("The roots are\n");
        printf("%f", (-b + sqrt(disc)) / (2.0 * a));
        printf("\n and\n");
        printf("%f\n", (-b - sqrt(disc)) / (2.0 * a));
    }
    if (disc < 0) {
        printf("The roots are complex. The roots are\n");
        printf("%f+I*%f", -b / (2.0 * a), sqrt(fabs(disc)) / (2.0 * a));
        printf("\n and\n");
        printf("%f-I*%f\n", -b / (2.0 * a), sqrt(fabs(disc)) / (2.0 * a));
    }
    return 0;
}
```

Here is a C program that performs numerical integration using Simpson's rule. The program computes $\int_0^1 \frac{dx}{1+x^2}$ by Simpson's rule using 4 sub-intervals. Recall Simpson's rule: Let $h = \frac{a-b}{n}$, where a is the lower limit of integration and b is the upper limit of integration and n is the number of subintervals, which should be **even**. If we write $y_i = f(a+ih)$, then

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[y_0 + 4\left(y_1 + y_3 + \cdots + y_{n-1}\right) + 2\left(y_2 + y_4 + \cdots y_{n-2}\right) + y_n\right]$$

Note the definition of the function $\frac{1}{1+x^2}$ here. Since, return can contain expressions, we have done the job of finding the value of $\frac{1}{1+x^2}$ within the return statement. return statement will simply return the value of the expression. We will see more about expressions in Unit 3.

```
/* Simpson's rule; file name:unit1-simpsonf.c*/
#include<stdio.h>
/* function to calculate 1/(1+x*x) */
float f(float x)
{
    return (1 / (1 + x * x));
}

int main()
{
    float integral = 0, h = 1 / 4.0;
    /* Apply Simpson's rule */
    integral =
        (h / 3.0) * (f(0) + 4 * (f(h) + f(3 * h))
              + 2 * f(2 * h)
```

```
                    + f(4 * h));
    printf("The integral is %f\n", integral);
    return (0);
}
```

1.6 SUMMARY

In this unit, we have studied the following points:

1) Because it was written for the purpose of porting Unix system and it was required that C permits low level operations. It also enjoys the advantages of high level languages, being comprehensible. It allows modular programming.
2) The C program is organised into functions and every C program must have a function called main.
3) To print certain special characters we need to use **escape sequences:**
   - \n New Line
   - \' Single quote '
   - \" Double quote "
   - \\ Backslash \
   - \a Ring terminal bell (the **a** is for **alert**) [ANSI] extension]
   - \? Question mark [ANSI extension]
   - \b Backspace
   - \r Carriage return
   - \f Formfeed
   - \t Horizontal tab
   - \v Vertical tab
   - \0 ASCII null character
4) We also examined some example C programs.

1.7 SOLUTIONS/ANSWERS

E1) No, you cannot have nested comments. This is because the compiler will match the first opening /* with the first closing */ and will interpret the rest of the comment as a part of the program. This will result in an error message from the compiler.

E2) **This is the first line of output.But is this the second line of output?**

E3) **In how many lines will the output of this program be printed?**

E4) a)
```
"A\n",the teacher said,"is used to
insert a new line in a C string."
"IC, IC", said the blind student.
```

b)
```
To be, or not to be,-that is the question:-
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune
Or to take arms against a sea of troubles,
And by opposing end them?-To die,-to sleep,-
```

E5)
```
/*File name:unit1-prog15.c*/
#include<stdio.h>
int main()
```

```
{
    printf("/* This is a C comment */");
    return (0);
}
```

E6) Following are the mistakes in the program:

1) Wrong brackets are used around and the stdio.h is also spelt wrongly. It should be `#include<stdio.h>`.
2) The function name main should not be in capitals. The brackets after main should be round brackets and not flower brackets. The correct way is to type `main ()`.
3) The brackets before and after print statement should be flower brackets and not round brackets. It should be `printf` not `print`. The escape sequence `\m` is not a legal escape sequence.
4) The slash in the comment `\* This is easy*\` is wrong. It should be a backward slash and it should be `/*This is easy*/`.