UNIT 1: Introductory

Structure

1.0 INTRODUCTION

This Unit presents the basic features common to all C programs. Quite inevitably, use is made of operators and functions that are discussed at greater length in subsequent Units: for example, operator priorities and associativities, expressions, pointers and input/output form the subject of the following Unit; but they are implicitly used in the present Unit. You will find that it is possible, in fact easy, to write simple programs that make use of the many facilities that C provides, the more intricate features of which you may not perhaps fully comprehend right away. This shouldn't worry you. That knowledge will come with further reading, and practice. We would like you to create and execute the programs listed anyway, and experiment with different values for program variables. Try to see why you get the results that you do, and before you know it you'll have mastered the language!

1.1 OBJECTIVES

After working through this Unit you should be able to

-declare, define and compute the values of program variables via simple arithmetic operation
-create and execute simple C programs
-recognise examples of C strings, escape sequences, and program comments
-use the simplest functions for input and output.

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 had recently been 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. 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 was required was a language that would permit assembly-like (i.e. low or hardware level) operations, such as bit manipulation, and at the same time be runnable on different computers. None of the languages then available could have served the purpose: a new one was called for, and C was born. The rest is history: Unix became spectacularly successful, in large measure because of its portability, and C, in which most of it was written, has since come to occupy a pre-eminent position in die development of systems programs.

The success of **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 which 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 upto ten million which are of the form **k * k + 1,** where k is an integer (2, 5, 17 and 37 are trivial examples of such primes). Then one module of the program could be written to generate the number **k * k + 1** for the next admissible value of k; and 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, in the manner that Pascal is. 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 any 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 long; and they are not **portable:** an assembly language program written for one computer cannot be executed on another with a different achitecture 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 an intangible quality which can only be termed elegance. This derives from the versatility of it's control structures and the 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. While brevity is undeniably a virtue (specially when you consider the vastnesses 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) it is wise to resist that temptation to parsimony, that C by its nature seems to encourage.

One common classification of computer languages divides them into two broad categories: high-level languages (HLLS) and low-level languages (LLLs). An HLL is a language that is easy for 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      LETZ = 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 another. Consider the following
statement from the MACRO Assembly Language of the VAX series of computers manufactured by Digital Equipment Corporation, USA:

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

It is highly unlikely that one who is not familiar with the 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 LLL's: 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): with the additional merit that its programs can be concise, elegant and easy to debug; and finally, because it has a single authority for its definition, there is a greater degree of standardisation and uniformity in C compilers than for other HLLs.

Ninety percent of the code of the UNIX operating system and of its descendants is written in **C.** Apart from **B,** which had been created by Ken Thompson for the first UNIX, one of the ancestors of C was a language called **BCPL_B C**omputer **P**rogramming **L**anguage, that had earlier been developed by Martin Richards: hence the name C is doubly appropriate: the successor of **B** and **BCPL!** It has often been said, and 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 systems software without having to worry about the architecture of the target machine. (Where architecture-dependent code i.e. assembly code is necessary, it can usually be invoked from within the C environment.) Today it is the language of choice for systems programming, for the development of 4GL packages such as DBase, and also for the creation of user-friendly interfaces for special applications. But it is not less beloved of applications programmers, who admire it for its elegance, brevity, and the versatility of its operators and control structures. C is a mid- 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. If you plan to use it for writing systems programs, you will require some familiarity with machine architecture: octal and hexadecimal numbers, memory addresses, registers, bit operations, etc.

Given the vast popularity of C with the programming community it was no surprise that an object-oriented extension, C++, should have appeared on the scene. In the OOP world C++ has created as strong a foothold for itself as did C in systems programming in the seventies and eighties. C++ was created by Bjarne Stroustrup, also at Bell Labs. Object-oriented languages have several advantages over procedure-oriented languages such as FORTRAN, Pascal and C: extremely large pieces of program code (common in contemporary applications) become easier to maintain, are made more reliable and are conveniently reusable if they're written with an "object" orientation rather than a "procedure" orientation. For this reason UNIX is being rewritten by AT&T in C++, and Apple Corporation have chosen this language for the development of systems software for the Macintosh. Although it is risky to make predictions about future turns the software industry will take, it is likely that C/C++ skills will remain in demand for several years.

Many C compilers for the PC also include graphics capability; compiler writers (primarily for PC compilers) added this feature to make the language even more popular in the industry, though the ANSI (American National Standards Institute) definition of the language does not support graphics. There is very little standardisation in C graphics routines. They differ from compiler to compiler in minor and even in major detail. For this reason a discussion of graphics routines has not been included here.

In this Block we will describe the Kernighan and Ritchie version called "K & R C", or Classic C, made famous by their classic: Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice-Hall: Englewood Cliffs, New Jersey, 1978. But we will also mention the ANSI extensions of (or departures from) that version, wherever applicable. Though ANSI C is defined in the second edition of Kernighan and Ritchie (1989), many C compilers in use today comply for the most part with the earlier definition of the language.

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:

```
/*                    Program                    1.1                    */
#include                                                    <stdio.h
main ( )
        {
                printf ("This is easy !!\n");
            }
```

Note: Exactly how you would create, compile, link and execute a C program depends very much on the software and hardware available to you. On a PC running DOS you may be able to use **Turbo C, Lattice C, MicroSoft C, Quick C, Aztec C, Lightspeed C** or any other of a host of excellent compilers available today, each with its particular set of compile options, commands, diagnostics and also idiosyncrasies; while in a UNIX or similar environment the commands and other features will be quite distinct. You'll have to read your system's manuals or consult a local guru to learn the nitty-gritties of program creation, compilation, linkage and execution.

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 he 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. In **ANSI** C Program 1.1 is correctly written as follows:

```
/*          Program          1.1,          ANSI          C          Version          */
#include                                                              <stdio.h
void main (void)
    {
            printf                ("This                is                easy!!\n");
            return 0;
        }
```

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 compiler, ready for you to use. 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 (more properly: string) of characters 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 must not include a "hard" carriage-return (), i.e. all its characters must be keyed in on a single line on the terminal.

```
"This is most certainly not a C string.
It contains a carriage return character."
```

As we have seen in the case of printf () when a function is invoked, it is passed (the values of) its arguments. [Note: In C, as in Pascal, the arguments passed to a function may be values of program variables, or they may be the memory addresses of program variables: recall the distinction that Pascal makes between value parameters and var parameters. Later in this Block we shall find that C strings are in fact more like the var parameters of Pascal than value parameters.] 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 a 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. (In a program for listing primes of the form k * k + 1, one function, to which is passed the next vaue of k, could be created to compute k * k + 1, and another function could be written to test this result for primeness. And these functions could be invoked repeatedly, for each different value of k.) Functions will formally be 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 Program 1.1.

Preprocessing is a phase of compilation that is performed prior to the analysis of the program text. Though in this and later Units we will have more to say about preprocessor directives there are two ways in which they differ from statements that you should know about straightaway: one, they must begin in the first column, with no spaces between the # and the include; and two, they are not terminated by a semicolon. stdio.b is a header rile that comes with your C compiler 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 source 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 Program 1. 1. 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. C, like Pascal (where the blocks are delimited by a begin- end pair) is a block-structured language. Although the left and right braces which mark a block may be placed anywhere on a line, for convenience of reading and debugging they are generally vertically aligned in 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                          1.2                          */
#include                                                                          <stdio.h
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    ("Ths    is    the    innermost    block.\n");
                                        }
                                   }
                         }
                }
```

In Program 1.2 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; this is one significant distinction between C and Pascal: while Pascal uses the semicolon to separate statements, in contrast C uses it to terminate statements. Note that the Pascal program which does the same thing as Program 1.1 above works very well without the semi- colon after the writein:

```
program printf (input, output);
        begin
                writeIn (This is easy!!')
            end.
```

C statements may be 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, (as in Pascal) 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.

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 be placed wherever the syntax allows "white space" charters: blanks , tabs or newlines.

Check Your Progress 1

Question 1: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.

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 1.1 or 1.2 you will have noticed that the \n doesn't appear in the output Each \n in the string argument of a printf ()causes the cursor to be placed at the beginning of die 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 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                    1.3                    */
main ( )
        {
                printf ("This\nstring\nwill\nbe\nprinted\,nin\n9\nlines.\n");
            }
```

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.

Note that the difference between the writeIn and write statements of Pascal is mirrored precisely in C by including, or not including, a \n at the end of the string argument of printf ( ).

Though s cannot be included in a string, if you must deal with a long stringof characters that cannot fit conveniently in a single line, there is a way of continuing it into the next line. This is done by inserting a backslash (\) followed by a 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 then 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 es- cape sequence, \\ helps print \, and \' the single quote character,'. All escape sequences in C begin with a backslash.

Check Your Progress 2

Question 1: Give the output of the following program:

```
/*                          Program                          1.4
#include                                                    <stdio.h
main ( )
        {
            printf ("This is the first line of outputs");
            printf ("But is this the second\nline of output?"):
        }
```

Question 2: Execute Program 1.5 below and obtain the answer to the question in its printf ( ):

```
/*                  Program              1.5                  *1
main ( )
        {
            printf ("In how many lines will the output\
            of this program be printed?"),,
        }
```

Question 3: Give the out putof the following programs:

(i)

```
/*                  Program                          1.6*/
#include                                                <stdio.h
main ( )
    {
     printf ("\"A \\", the teacher said, \"is used to");
     printf (" insert a new line in a C string.\"");
     printf ("\n\"I C, I C\", said the blind student.\n");
    }
```

(ii)

```
/*                  Program                          1.7*/
#include                                                <stdio.h
main ( )
```

```
{
 printf ("There was a naughty Boy,\nAnd a");
 printf("naughty Boy was he,\nHe ran awa");
 printf ("y to Scotland\n The people for t"),
 printf ("o see -\n there he found\n That t");
 printf ("he ground\n was as hard,\n That a");
 printf (" yard\n Was as long,\n That a son");
 prinif ("g\nWas merry,\nThat a cherry");
 printf ("\n Was as red --\n that led\n Was");
 printf ("as weighty,\n That fourscore was");
 printf (" as eigty,\nThat a door\n Was a");
 printf ("s wooden\n As in England-\nSo h");
 printf ("e stood in his shoes\n And he wo");
 printf ("nder\'d,\n He wonder\'d,\nHe sto" );
 printf ("od in his shoes\n And he wonder\'d.");
}
```

Question 4: Write a C language program that gives for its output:

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

Question 5: 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
\f carriage return
\f Formfeed
\t horizontal tab
\V vertical tab
\O ASCII null character

Placing any of these within a string causes either the indicated action, or the related character to be output.

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. 1 1 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.

```
/* Program 1.8 */
#include                                                    <stdio.h
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 number
                                        */

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= 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);

                                        /*
                                        %% is the escape sequence to print the percent sign.
                                        */
z = y % X;

Printf ("z = y %% x = %d\n", z);
                }


    /*                  Program                 1.9                 */
    #include                                                    <stdio.h
    main ( )
            {
                        /* Read values from the keyboard, see how scanf () works*/
```

```
                int x, y, z;
                printf ("Enter a value for x. Type a small integer, press : " );
                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);
        }
/*Program 1. 10 */
# include <stdio.h
main ( )
    {
        int x, y;
        printf ("Enter a value for x: ");
        scanf ("%d", &x);
        printf ("Enter a value for y: ");
        scanf ("%d", &y);
        if (x y) /* Is x greater than y? Then say so: */
            printf ("x is greater than y:\n");
          else /* else, if it's not, deny it. */
            printf ("x is not greater than y.\n");
                /* The computer can tell if one number is greater than
                another.*/
        }
/* Program 1. 11 */
#include <stdio.h
int addtwo (int x, int y); /* a function that adds two ints */
main ( )
    {
        int val_1, val_2, sum;
        printf ("Enter a number: ");
        scanf ("%d", &val_l);
        printf ("Enter another:");
        scanf ("%d", &val_2);
        printf ("\n\n Will let the function addtwo () add them ... \n");
        sum = addtwo (val_l, val_2);
                /* transfer control to
                    addtwo 0, with arguments
                    val-1 and val-2
                */
            printf ("\nNow we\'re back in main () ... What have we
            here.?\n");
            printf ("\naddtwo () tells. us their sum is %d.\n", sum);
        }
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);
    printf ("\ntheir sum is ... am working on it ... \n");
    return (p + q);
    }

/*Program 1.12: non-ANSI C version of Program 1.11 */
#includ <stdio.h
int addtwo ( ); /* note the difference here */
main ( )
        {
            int val_1, val_2, sum;
            printf ("Enter a number: ");
```

```
        scanf ("%d", &vat_1);
        printf ("Enter another:");
        scanf ("%d", &val_2);
        printf ("n\nWill let the function addtwo () add them ... \n");
        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");
        printf ("\naddtwo () tells us their sum is %d.\n", sum);
    }
int addtwo (p, q) /* Note difference between K&R and ANSI C */
  int p, 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),
        printf ("\ntheir sum is am working on it \,n");
        return (p + q);
    }
```

1.6 SUMMARY

In this unit we presented a historical introduction of the origin of the C programming language, explained the advantages it offers to both systems and applications programmers, and showed how to write very simple programs.

The following important terms were presented: main (), function, block, statement, statement terminator, string ,null string, null character, escape sequences, preprocessing, #include <**stdio.h**. We also saw how **printf** () prints strings.