SECTION 2 C PROGRAMMING

2.0 INTRODUCTION

In the earlier unit we introduced you to the architecture of Unix operating system, especially the Unix command required by the users. In this unit we present C language – because BSD sockets library and its interface are written in C, hence it is essential for students to learn programming in C language, also students should learn about the compilation, execution and testing of programs, system calls and how to get Unix help. C programming language known for its power 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 computor 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 which is essential for socket programming.

2.1 OBJECTIVES

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

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

2.2 INTRODUCTION TO C

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.

Prior to writing C programs, it would be interesting to find out what really is the 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.

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-I I 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 a 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.

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. 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. Structured language is one that allows only one entry and one exit point **to/from** a block or module. 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 of this course. 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.

C Program Structure

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? 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 Figure3 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 3: 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. Its 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 functions we have to write and others are used from the standard libraries.

#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 { } show 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 or a block. 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 new line 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 that describes complete program development life cycle given below:

Example

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

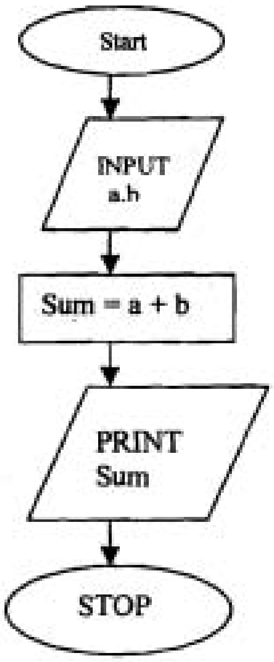

Figure 4: Flowchart to add two numbers

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), which is the data type to indicate integer values. The next statement is the printfo 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 printfo 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 printfo statement.

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

Writing a Program

A C program can be executed on platforms such as DOS, Unix etc. like DOS, 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.

Compiling a 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 directly. 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*** 5.

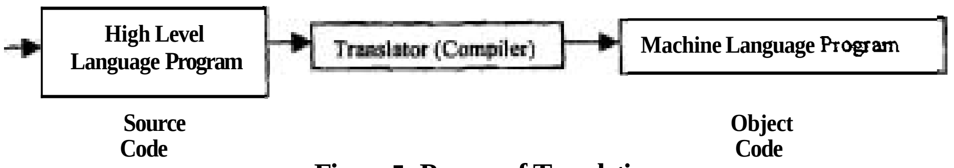

Figure 5: Process of Translation

The C Compiler

If the name of the program file is **testprog.c,** the simplest method is compile it to type **cc testprog.c**

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

cc testprog.c -o testprog

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

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.

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 codes 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.

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.**

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 a 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.

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. Suppose we have to find the average of three numbers and we write the following code:

Example: 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 a run time error message are flashed on the screen. Let us look at the following example:

Example: 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.**

Diagrammatic Representation of Program Execution Process

The Figure 6 shows the diagrammatic representation of the program execution process.

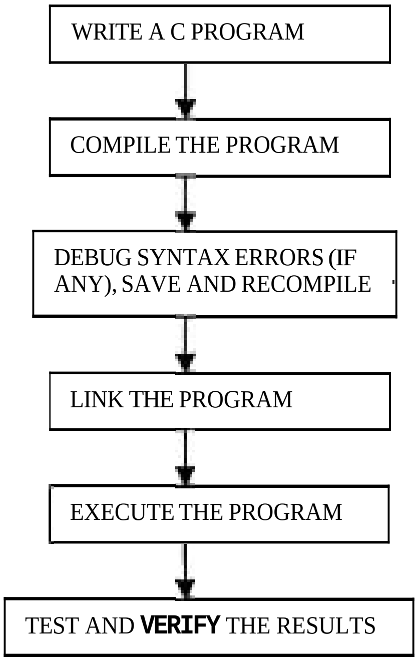
WRITE A C PROGRAM
COMPILE THE PROGRAM
DEBUG SYNTAX ERRORS (IF ANY), SAVE AND RECOMPILE
LINK THE PROGRAM
EXECUTE THE PROGRAM
TEST AND VERIFY THE RESULTS

Figure 6: Program Execution Process

Debugging and Optimisation

In the previous section you have studied the basics C programming, and got elementary knowledge for compiling and executing programs but you need some more detailed information about compilation, execution, debugging, optimization, linking and standards libraries of Unix operating system.

Debugging

Debugging of a program includes many things like running it step by step, setting the break point before a giver. command is executed, checking at contents of variables during program execution, etc. To relate between the executable program and the original source code, we should ask the compiler to insert information to the resulting executable program to help the debugger or tester. This information is called "debug information". To add the debugging information to your program you should compile it with **"-g"** flag option, as shown below in the command.

- **% cc -g Prog_Debug.c -o Prog_Debug**

The **'-g'** option informs the compiler to use debug info. You will find that the resulting file is larger in size than the file created without **'-g'** flag. This difference in size is due to the additional debug information. But if you want to remove this **information** you can **use** the strip command, as given below:

- **%** strip **Prog_Debug**

Creating an Optimized Code

Once you created a program and debugged it you should compile it into an efficient and optimize code, you may ask why should I optimize the code? After optimization of a code, it runs faster and it takes smaller space. To create an optimized program progl, you can write the following command:

% cc **-O Prog1.c -o Prog1**

The **'-O'** option informs the compiler to optimize the code. Because various optimization algorithms are applied to code so the compiler process it slower than normal compilation with **'-O'** option. Further you can explore different optimization levels associated with **'-O'** option using Unix help.

This .also means the compilation will take longer, as the compiler tries to apply various optimization algorithms to the code. This optimization is supposed to be conservative, in that it ensures that the code will still perform the same functionality as it did when compiled without optimization (well, unless there are bugs in our compiler). Usually we can define an optimization level by adding a number to the **'-O'** flag. The higher the number - the better optimized the resulting program will be, and the slower the compiler will complete the compilation.

Compiler Warnings

In general the compiler generates error messages only. **However,** we can instruct the compiler to give us details about warnings also. Testing and removing all warning from the program improve the quality of your program. To get the compiler to use all types of warnings, use a command as given below:

- **% cc -Wall Prog1.c -o Prog1**

Multi Source Programs

Till now you learned how to compile a single-source program but you can compile multi source programs also. Unix provides two possible ways to compile a multi-source C program. The first is to use a single command line to compile all the files. For example, you have a program whose source is found in files **"main.c", "hello.c"** and **"world.c"**. you can **compile** as given below:

- **% cc main.c hello.c world.c -o hello–world**

This will cause the compiler to compile each of the given files separately, and then link them altogether to one executable file named "hello–world". **A** different way to compile multi source C programs is to divide the compilation process into two phases compiling and linking. Let's see the previous example following a different method:

```
cc -c main.cc

cc -c hello.c

cc -c world.c

cc main.o hello.o world.o -o hello_world
```

The first 3 commands have used **-c** option (which inform the compiler only to create an object file) and each have taken one source file, and compiled it into something called "object file", with the same names, but with a **".o"** extension. The 4th command is to link all object files into one program. The linker invoked by the compiler takes all the symbols from the 3 object files, and links them together into one executable file hello–world.

System Library

For most system calls and library functions we have to include an appropriate header file. **e.g., stdio.h, math.h.** To use a function, ensure that you have made the required #includes in your C file. Then the function can be called as though you had defined it yourself. To compile a program with including functions from **math.h** library header file, you can write the command as given below:

- cc **mathprog.c -o** mathprog **-lm**

The **"-lm"** option gives an instruction to the compiler to link the maths library with the program.

Unix System Calls

System calls are functions that a **programmer can** call to perform the services of the operating system. Some of them involve access to data that users must not be permitted to corrupt or even change. It is **difficult** to determine what is a library routine like **printf()** and what is a system call sleep(). To obtain information about a system call or library routine, you can read the Unix manual pages. You can read these manual pages by writing the following command for system call and to library routine:

- **% man 2 read // if read is a system call**
- **% man 3 read // if read is a library routine**

Note: All of the entries in Section 2 of the manuals are system calls, and all of the entries in Section 3 are library routines. As you know Unix system calls are interfaces, between the Unix kernel and the user programs and system calls are the only way to access kernel facilities such as the file system, the multitasking mechanisms, the inter-process communication and the socket primitives.

How does a C programmer actually issue a system call? There is no difference between a system call and any other function call. For example, the read system call might be issued like this:

```
nav= read(fd, buf, numbytes);
```

Where the implementation of the read may vary. It is better to understand that system calls are simply C subroutines. System calls takes longer time than a simple subroutine call because a system call involves a context switch between user and kernel. Generally, system calls return some value either it was successful or it failed. For example, read system call returns the number of bytes read, -1 is returned if an error occurred. When we use read () system call it should be as we have given below:

```
if ((nav = read(fd, buf, numbytes)) = = -1)
  {
   printf("Read operation failed\n");
   exit(1);
  }
```

To add more interactivity in programs you can use "perror" the library routine. It gives the description of the error condition stored in error number (errno which contains a code that indicates the reason). So, that you can handle the errors in a better way, an example is given below:

```
 if ((nav= read(fd, buf, numbytes)) == -1)
  {
   perror("read");
   exit(1);
  }
```

Output: "file does not exist" on an error.

2.3 SUMMARY

In this section we studied that basic introduction to the C language, C is developed at **AT&T's** Bell Laboratory of USA in 1972 written by Dennis Ritchie. C is a general purpose, structured programming language often called a middle level language. C program can be executed on platforms such as DOS, Unix etc. like DOS, 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***. We studied the procedure and command required to compile, run and debug the C program, further we have studied the difference between run time and logical errors.
The next section of this manual covers the exercises on the **TCP/IP** programming.

2.4 FURTHER READINGS

1) Brian W.Kernighan and Dennis M. Ritchie; The Cprogramming language Prentice Hall.
2) W. Richard Stevens, "*UNIX Network* Programming", Prentice Hall.
3) C language, MCS-011 course material of MCA, IGNOU.
4) http://www.programmersheavep.com.
5) http://www.ee.surrey.ac.uk/Teaching/Unix/unixintro.html.