MASTER OF COMPUTER APPLICATION

SEMESTER 1

PROGRAMMING & PROBLEM-SOLVING USING C

Unit 1

Introduction to C Programming

Table of Contents

SL No	Topic	Fig No / Table / Graph	SAQ / Activity	Page No
1	Introduction	-	-	3-4
	1.1 Objectives	-	-	
2	Features of C and its Basic Structure	1	1	5-7
3	Simple C programs Constants	-	2	7-9
	3.1 Integer Constants	-	-	
	3.2 Real Constants	-	-	
	3.3 Character Constants	-	-	
	3.4 String Constants	-	-	
	3.5 Backslash Character Constants	-	-	
4	Constraints	2, 1	-	10-13
5	Concept of an Integer and Variable	2	3	13-16
6	Rules for naming Variables and assigning values to variables	-	4	17-18
7	Summary	-	-	19
8	Terminal Questions	-	-	19
9	Answers	-	-	20-21

1. INTRODUCTION

C is a general-purpose, structured programming language. Its instructions consist of terms that resemble algebraic expressions, augmented by certain English keywords such as **if, else, for, do** and **while**. C was the offspring of the 'Basic Combined Programming Language' (BPCL) called B, developed in the 1960's at Cambridge University. B language was modified by Dennis Ritchie and was implemented at Bell laboratories in 1972. The new language was named C. Since it was developed along with the UNIX operating system, it is strongly associated with UNIX. This operating system, which was also developed at Bell laboratories, was coded almost entirely in C.

In C, the type of a variable determines what kinds of values it may take on. The type of an object determines the set of values it can have and what operations can be performed on it. This is a fairly formal, mathematical definition of what a type is, but it is traditional (and meaningful). There are several implications to remember:

1. The "set of values'' is finite. C's int type cannot represent *all* of the integers; its float type cannot represent *all* floating-point numbers.
2. When you're using an object (that is, a variable) of some type, you may have to remember what values it can take on and what operations you can perform on it. For example, there are several operators which play with the binary (bit-level) representation of integers, but these operators are not meaningful for and may not be applied to floating-point operands.
3. When declaring a new variable and picking a type for it, you have to keep in mind the values and operations you'll be needing.

This unit will introduce you to C programming with its various features, structure and how the C programs are constructed with simple examples.

1.1 Objectives

After studying this unit, you should be able to:

- *Discuss the features of C programming language*
- *Explain the basic structure of a C program*
- *Write simple programs in C*
- *Construct and use the concept of Constants and Integers*
- *Use the variable and its declaration in C*

2. FEATURES OF C AND ITS BASIC STRUCTURE

C is characterized by the ability to write very concise source programs, due in part to the large number of operators included within the language.

It has a relatively small instruction set, though actual implementations include extensive library functions which enhance the basic instructions.

The language encourages users to write additional library functions of their own. Thus, the features and capabilities of the language can easily be extended by the user.

C compilers are commonly available for computers of all sizes. The compilers are usually compact, and they generate object programs that are small and highly efficient when compared with programs compiled from other high-level languages.

Another important characteristic of C is that its programs are highly portable, even more so than with other high-level languages. The reason for this is that C relegates most computer dependent features to its library functions. Thus, every version of C is accompanied by its own set of library functions, which are written for the particular characteristics of the host computer.

A C program can be viewed as a group of building blocks called functions. A function is a subroutine that may include one or more statements designed to perform a specific task. To write a C program we first create functions and then put them together. A C program may contain one or more sections shown in Fig. 1.1.

The documentation section consists of a set of *comment* (remarks) lines giving the name of the program, the author and other details which the programmer would like to use later. Comments may appear anywhere within a program, as long as they are placed within the delimiters /* and */ (e.g., /*this is a comment*/). Such comments are helpful in identifying the program's principal features or in explaining the underlying logic of various program features.

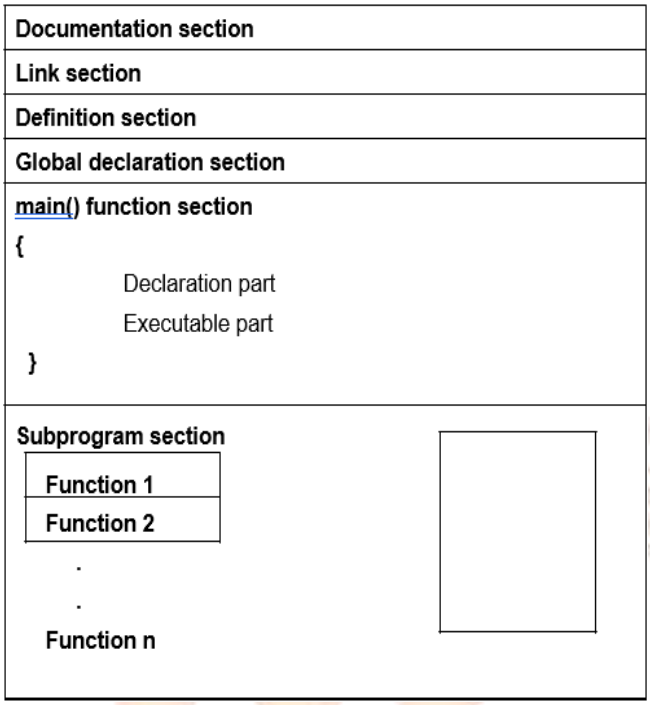

Fig 1.1: Sections in a C program

The link section provides instructions to the compiler to link functions from the system library. The definition section defines all symbolic constants.

There are some variables that are used in more than one function. Such variables are called *global* variables and are declared in the global declaration section that is outside of all the functions.

Every C program must have one **main** function section. This section contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between opening and closing braces ({ and }). The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end with a semicolon(;).

The subprogram section contains all the user-defined functions that are called in the **main** function. User-defined functions are generally placed immediately after the **main** function, although they may appear in any order.

All sections, except the **main** function section may be absent when they are not required.

SELF-ASSESSMENT QUESTIONS – 1

1. Using C language programmers can write their own library functions. (True/False)
2. C is a _____ level programming language.
3. The documentation section contains a set of _____ lines.
4. Every C program must have one **main()** function. (True/False)

3. SIMPLE C PROGRAMS

Let us see a simple c program given below:

```
#include <stdio.h>
main()
{
printf("Hello, world!\n");
return 0;
}
```

If you have a C compiler, the first thing to do is figure out how to type this program in and compile it and run it and see where its output went.

The first line is practically boilerplate; it will appear in almost all programs we write. It asks that some definitions having to do with the "Standard I/O Library'' be included in our program; these definitions are needed if we are to call the library function **printf** correctly.

The second line says that we are defining a function named **main**. Most of the time, we can name our functions anything we want, but the function name **main** is special: it is the function that will be "called'' first when our program starts running. The empty pair of parentheses indicates that our **main** function accepts no *arguments,* that is, there isn't any information which needs to be passed in when the function is called.

The braces { and } surround a list of statements in C. Here, they surround the list of statements making up the function **main**.

The line

printf("Hello, world!\n");

is the first statement in the program. It asks that the function **printf** be called; **printf** is a library function which prints formatted output. The parentheses surround **printf**'s argument list: the information which is handed to it which it should act on. The semicolon at the end of the line terminates the statement.

printf 's first (and, in this case, only) argument is the string which it should print. The string, enclosed in double quotes (""), consists of the words "Hello, world!'' followed by a special sequence: \n. In strings, any two-character sequence beginning with the backslash \ represents a single special character. The sequence \n represents the " 'new line'' character, which prints a carriage return or line feed or whatever it takes to end one line of output and move down to the next. (This program only prints one line of output, but it's still important to terminate it.)

The second line in the **main** function is

return 0;

In general, a function may return a value to its caller, and **main** is no exception. When **main** returns (that is, reaches its end and stops functioning), the program is at its end, and the return value from **main** tells the operating system (or whatever invoked the program that main is the **main** function of) whether it succeeded or not. By convention, a return value of 0 indicates success.

Program 1.1: Area of a circle

Here is an elementary C program that reads in the radius of a circle, calculates the area and then writes the calculated result.

```
#include <stdio.h>                              /* Library file access */
/* program to calculate the area of a circle */ /* Title (Comment) */
main()                                          /* Function heading */
{
float radius, area;                             /*Variable declarations */
```

```
printf("Radius=?");                    /* Output statement(prompt) */
scanf("%f", &radius);                  /* Input statement */
area=3.14159*radius*radius;            /* Assignment statement */
printf("Area=%f",area);                /* Output statement */
}
```

Program 1.2: Print a few numbers

Here is a program to illustrate a simple loop.

```
#include <stdio.h>
/* print a few numbers, to illustrate a simple loop */
main()
{
int i;
for(i = 0; i < 10; i = i + 1) /* Looping statement */ printf("i is %d\n", i);
return 0;
}
```

Program 1.3: Program to add two numbers

```
#include <stdio.h>
main()
{
int i,j,k; // Defining variables
i = 6;    // Assign values
j = 8;
k = i + j;
printf("sum of two numbers is %d \n",k); // Printing results

}
```

SELF-ASSESSMENT QUESTIONS – 2

5. The information that needs to be passed in when a function is called is _____.
6. The main() function doesn't return any value. (True/False)

4. CONSTANTS

Constants in C refer to fixed values that do not change during the execution of a program. C supports several types of constants as illustrated in Fig. 1.2.

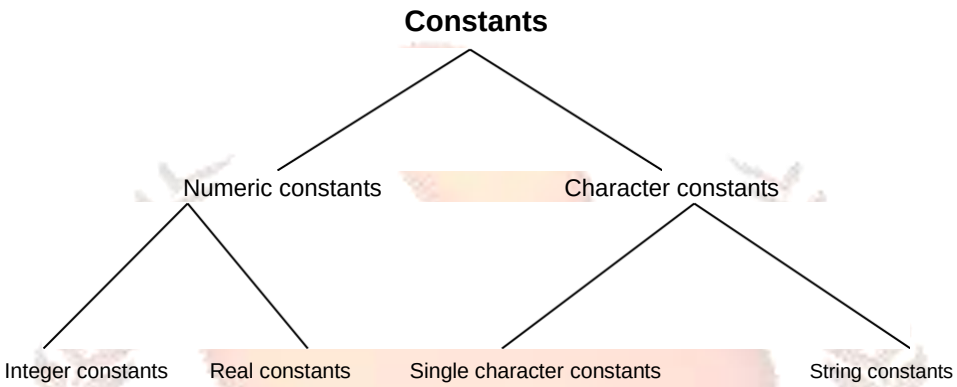

Fig. 1.2

4.1 Integer Constants

An integer constant refers to a sequence of digits. There are three types of integers, namely *decimal, octal and hexadecimal.*

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional – or +.

Examples: 12, -546, 0, 354647, +56

An *octal* integer constant consists of any combination of digits from the set 0 through 7, with a leading 0.

Examples: 045, 0, 0567

A sequence of digits preceded by 0x or 0X is considered as *hexadecimal* integer. They may also include alphabets A through F or a through f. The letters A through F represent numbers 10 through 15.

Examples: 0X6, 0x5B, 0Xbcd, 0X

The largest integer value that can be stored is machine-dependent. It is 32767 on 16-bit machines and 2,147,483,647 on 32-bit machines. It is also possible to store larger integer constants on these machines by appending *qualifiers* such as U, L and UL to the constants.

Examples: 54637U or 54637u (unsigned integer)

65757564345UL or 65757564345ul (unsigned long integer)

7685784L or 7685784l (long integer)

Program 1.1: Program to represent integer constants on a 16-bit computer

```
/* Integer numbers on 16-bit machine */
main()
{
        printf("Integer values\n\n");
        printf("%d %d %d\n",
        32767,32767+1,32767+10); printf("\n");
        printf("Long integer values\n\n");
        printf("%ld %ld %ld\n", 32767L, 32767L+1L, 32767L+10L);

}
```

Type and execute the above program and observe the output

4.2 Real Constants

The numbers containing fractional parts like 67.45 are called real (or floating point) constants.

Examples: 0.0045, -8.5, +345.678

A real number may also be expressed in *exponential (scientific)* notation.

The general form is:

mantissa e exponent

The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer number with an optional plus or minus sign. The letter **e** separating the mantissa and the exponent can be written in either lowercase or uppercase.

Examples: 04e4, 12e-2, -1.3E-2

7500000000 may be written as 7.5E9 or 75E8.

Floating point constants are normally represented as double-precision quantities. However, the suffixes f or F may be used to force single precision and l or L to extend double-precision further.

4.3 Character Constants

A single character constant (or simple character constant) contains a single character enclosed within a pair of single quote marks.

Examples: '6', 'X', ';'

Character constants have integer values known as ASCII values. For example, the statement

printf("%d", 'a');

would print the number 97, the ASCII value of the letter a. Similarly, the statement

printf("%c", 97);

would print the letter a.

4.4 String Constants

A string constant is a sequence of characters enclosed within double quotes. The characters may be letters, numbers, special characters and blank space.

Examples: "Hello!", "1947", "5+3"

4.5 Backslash character constants

C supports some special backslash character constants that are used in output functions. A list of such backslash character constants is given in Table 1.1. Note that each one of them represents one character, although they consist of two characters. These character combinations are called *escape sequences.*

Table 1.1: Backslash character constants

Constant	Meaning
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab

'\''	single quote
'\" '	double quote
'\\'	backslash
'\0'	null

SELF-ASSESSMENT QUESTIONS – 3

7. An octal integer constant consists of any combination of digits from the set _____ through _____.
8. A sequence of digits preceded by 0x or 0X is considered as _____ integer.
9. A string constant is a sequence of _____ enclosed within double quotes.

5. CONCEPT OF AN INTEGER AND VARIABLE

Integers are whole numbers with a range of values supported by a particular machine. Generally, integers occupy one word of storage, and since the word sizes of machines vary (typically, 16 or 32 bits) the size of an integer that can be stored depends on the computer. If we use 16 bit word length, the size of the integer value is limited to the range -32768 to +32767 (that is, -2^{15} to $+2^{15}-1$). A signed integer uses one bit for sign and 15 bits for the magnitude of the number. Similarly, a 32 bit word length can store an integer ranging from -2,147,483,648 to 2,147,483,647.

In order to provide some control over the range of numbers and storage space, C has three classes of integer storage, namely **short int, int** , and **long int,** in both signed and unsigned forms. For example, **short int** represents fairly small integer values and requires half the amount of storage as a regular **int number** uses. A signed integer uses one bit for sign and 15 bits for the magnitude of the number, therefore, for a 16 bit machine, the range of unsigned integer numbers will be from 0 to 65,535.

We declare **long** and **unsigned integers** to increase the range of values. The use of qualifier **signed** on integers is optional because the default declaration assumes a signed number. The

Table 1.2 shows all the allowed combinations of basic types and qualifiers and their size and range on a 16-bit machine.

Table 1.2: Basic types, qualifiers and their size

Type	Size (bits)	Range
int or signed int	16	-32,768 to 32,767
unsigned int	16	0 to 65535
short int or signed short int	8	-128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	-2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295

Informally, a *variable* (also called an *object*) is a place where you can store a value so that you can refer to it unambiguously. A variable needs a name. You can think of the variables in your program as a set of boxes, each with a label giving its name; you might imagine that storing a value "in'' a variable consists of writing the value on a slip of paper and placing it in the box.

Now let us see how integers and variables are declared. A *declaration* tells the compiler the name and type of a variable you'll be using in your program.

In its simplest form, a declaration consists of the type, the name of the variable, and a terminating semicolon:

```
int i;
```

The above statement declares an integer variable i.

```
long int i1, i2;
```

We can also declare several variables of the same type in one declaration, separating them with commas as shown above.

The placement of declarations is significant. You can't place them just anywhere (i.e. they cannot be interspersed with the other statements in your program). They must either be placed at the beginning of a function, or at the beginning of a brace-enclosed block of

statements, or outside of any function. Furthermore, the placement of a declaration, as well as its storage class, controls several things about its *visibility* and *lifetime*, as we'll see later.

You may wonder *why* variables must be declared before use. There are two reasons:

1. It makes things somewhat easier on the compiler; it knows right away what kind of storage to allocate and what code to emit to store and manipulate each variable; it doesn't have to try to intuit the programmer's intentions.
2. It forces a bit of useful discipline on the programmer: you cannot introduce variables wherever you wish; you must think about them enough to pick appropriate types for them. (The compiler's error messages to you, telling you that you apparently forgot to declare a variable, are as often helpful as they are a nuisance: they're helpful when they tell you that you misspelled a variable, or forgot to think about exactly how you were going to use it.)

Most of the time, it is recommended to write one declaration per line. For the most part, the compiler doesn't care what order declarations are in. You can order the declarations alphabetically, or in the order that they're used, or to put related declarations next to each other. Collecting all variables of the same type together on one line essentially orders declarations by type, which isn't a very useful order (it's only slightly more useful than random order).

A declaration for a variable can also contain an initial value. This *initializer* consists of an equal sign and an expression, which is usually a single constant:

```
int i = 1;
int i1 = 10, i2 = 20;
```

SELF-ASSESSMENT QUESTIONS – 4

10. The size of the Integers in C language is same in all the machines. (True/False)
11. A _____ is a place where we can store values.
12. Size of int is _____ bits.
13. Which of the following tells the compiler the name and type of a variable you'll be using in your program? (declaration)
    a) declaration
    b) variables
    c) integers
    d) assignments
14. The _____ consists of an equal sign and an expression, which is usually a single constant. (initializer)
15. A single declaration statement can contain variables of different types. (True/False)

6. RULES FOR NAMING VARIABLES AND ASSIGNING VALUES TO VARIABLES

Within limits, you can give your variables and functions any names you want. These names (the formal term is “identifiers'') consist of letters, numbers, and underscores. For our purposes, names must begin with a letter. Theoretically, names can be as long as you want, but extremely long ones get tedious to type after a while, and the compiler is not required to keep track of extremely long ones perfectly. (What this means is that if you were to name a variable, say, *supercalafragalisticespialidocious,* the compiler might get lazy and pretend that you'd named it *super-calafragalisticespialidocio,* such that if you later misspelled it *super-calafragalisticespialidociouz,* the compiler wouldn't catch your mistake. Nor would the compiler necessarily be able to tell the difference if for some perverse reason you deliberately declared a second variable named *supercalafragalisticespialidociouz.)*

The capitalization of names in C is significant: the variable names *variable, Variable*, and *VARIABLE* (as well as silly combinations like variAble) are all distinct.

A final restriction on names is that you may not use *keywords* (the words such as **int** and **for** which are part of the syntax of the language) as the names of variables or functions (or as identifiers of any kind).

Now let us see how you can assign values to variables. The assignment operator = assigns a value to a variable. For example,

```
x = 1;
```

sets x to 1, and

```
a = b;
```

sets a to whatever b's value is. The expression

```
i = i + 1;
```

is, as we've mentioned elsewhere, the standard programming idiom for increasing a variable's value by 1: this expression takes i's old value, adds 1 to it, and stores it back into i. (C provides several “shortcut'' operators for modifying variables in this and similar ways, which we'll meet later.)

We've called the = sign the “assignment operator'' and referred to “assignment expressions'' because, in fact, = is an operator just like + or -. C does not have “assignment statements''; instead, an assignment like a = b is an expression and can be used wherever any expression can appear. Since it's an expression, the assignment a = b has a value, namely, the same value that's assigned to a. This value can then be used in a larger expression; for example, we might write

```
c = a = b;
```

Which is equivalent to?

```
c = (a = b);
```

and assigns b’s value to both a and c. (The assignment operator, therefore, groups from right to left.) Later we'll see other circumstances in which it can be useful to use the value of an assignment expression.

It's usually a matter of style whether you initialize a variable with an initializer in its declaration or with an assignment expression near where you first use it. That is, there's no particular difference between

```
int a = 10;
```

and

```
int a;
/* later... */
a = 10;
```

SELF-ASSESSMENT QUESTIONS – 5

16. In C, variable names are case sensitive. (True/False)
17. A variable name in C consists of letters, numbers and _____.

7. SUMMARY

C is a general-purpose, structured programming language. Its instructions consist of terms that resemble algebraic expressions, augmented by certain English keywords such as **if, else, for, do** and **while**. C is characterized by the ability to write very concise source programs, due in part to the large number of operators included within the language. Every C program consists of one or more functions, one of which must be called **main**. The program will always begin by executing the **main** function. Additional function definitions may precede or follow **main**.

Integers are whole numbers with a range of values supported by a particular machine. Generally, integers occupy one word of storage, and since the word sizes of machines vary (typically, 16 or 32 bits) the size of an integer that can be stored depends on the computer. A *variable* (also called an *object*) is a place where you can store a value. A *declaration* tells the compiler the name and type of a variable you'll be using in your program. The assignment operator = assigns a value to a variable.

8. TERMINAL QUESTIONS

1. Explain the history of C language.
2. What are the advantages of C language?
3. What are the major components of a C program?
4. What significance is attached to the function **main**?
5. What are arguments? Where do arguments appear within a C program?
6. Distinguish between signed and unsigned integers.
7. What are the components of declaration statement?
8. State the rules for naming a variable in C.
9. What is assignment operator and what are its uses?

9. ANSWERS

Self-Assessment Questions

1. True
2. High
3. Comment
4. True
5. Arguments
6. False
7. (0, 7)
8. Hexadecimal
9. characters
10. False
11. Variable
12. 16
13. (a) declaration
14. initializer
15. False
16. True
17. Underscores

Terminal Questions

1. C was developed by Dennis Ritchie and came after B in the year 1972. (Refer Section 1.1 for more details)
2. C is fast, efficient and structured. (Refer to section 1.2 for more details)
3. Documentation section, Link section, Definition section, Global declaration section, main() function section, Subprogram section
4. **main()** is the function that will be "called'' first when our program starts running.
5. The arguments are symbols that represent information being passed between the function and other parts of the program. They appear in the function heading.
6. A signed integer uses one bit for sign and remaining bits for the magnitude of the number, whereas an unsigned integer uses all the bits to represent magnitude.

7. A declaration consists of the type, the name of the variable, and a terminating semicolon.
8. Variables (the formal term is “identifiers'') consist of letters, numbers, and underscores. The capitalization of names in C is significant. You may not use *keywords* (the words such as int and for which are part of the syntax of the language) as the names of variables or functions (or as identifiers of any kind).
9. The assignment operator (=) assigns a value to a variable.

Exercises

1. Explain the basic structure of a C program with an example.
2. What are the different steps in executing a C program?
3. Write a C program to convert Celsius to Fahrenheit and vice versa.
4. Write a program in C to add, subtract, multiply any 3 numbers.