UNIT -1 Syllabus:

Computer fundamentals- Hardware, software, Computer Language, Translators, Compiler, Interpreter, Loader and Linder, Program Development steps- Algorithms, Pseudo code, flow charts, Specification for converting Algorithms into Programs basic.
Introduction to C Language – History, Simple C program, Structure of a C Program, Identifiers, Basic data types, user defined data types, Variables, Constants, type Qualifiers, Managing Input/ Output, Operators, Expressions, Precedence and Associativity, Expression Evaluation, Type Conversions, Simple C programming Examples.

1. Computer Fundamentals:

What is Computer?

Computer is an advanced electronic device that takes raw data as input from the user and processes these data under the control of set of instructions (called program) and gives the result (output) and saves output for the future use. It can process both numerical and non-numerical (arithmetic and logical) calculations.

A computer has four functions:

a. accepts data Input
b. processes data Processing
c. produces output Output
d. stores results Storage

Input (Data): Input is the raw information entered into a computer from the input devices. It is the collection of letters, numbers, images etc.

Process: Process is the operation of data as per given instruction. It is totally internal process of the computer system.

Output: Output is the processed data given by computer after data processing. Output is also called as Result. We can save these results in the storage devices for the future use. Computer System All of the components of a computer system can be summarized with the simple equations.

COMPUTER SYSTEM = HARDWARE + SOFTWARE+ USER

• Hardware = Internal Devices + Peripheral Devices All physical parts of the computer (or everything that we can touch) are known as Hardware.

• Software = Programs Software gives "intelligence" to the computer.

• USER = Person, who operates computer.

Software

Software, simply are the computer programs. The instructions given to the computer in the form of a program is called Software. Software is the set of programs, which are used for different purposes. All the programs used in computer to perform specific task is called Software.

Types of software

i) System software:

a) Operating System Software DOS, Windows XP, Windows Vista, Unix/Linux, MAC/OS X etc.

b) Utility Software Windows Explorer (File/Folder Management), Windows Media Player, Antivirus Utilities, Disk Defragmentation, Disk Clean, Backup, WinZip, WinRAR etc…

ii) Application software:

a) Package Software Ms. Office 2003, Ms. Office 2007, Macromedia (Dreamweaver, Flash, Freehand), Adobe (PageMaker, PhotoShop)

b) Tailored or Custom Software SAGE (Accounting), Galileo/World span (Travel) etc.

2. Computer Language:

An artificial language used to write programs that can be translated into machine language and then executed by a computer. As we know that computer understands only machine language. It does not understand the English language or any other what we normally use. So to perform tasks from computer we use programming language to write programs. These programs tell the computer what to do? Each natural language has a method of using symbols of that language. In English, the method is given by the rules of the grammar. These rules tell us which word to use when and how to use it. Similarly, the rules of a programming language must also be used. These rules are known as "**Syntax Rules**". In case of natural language people can use poor and incorrect language, but in case of programming language we can't use. Each language has strict syntax rules and these rules cannot be ignored. Each program has its own syntax and this is further converted to machine language. Now after the conversion, machine is able to understand the task. This conversion is done by compiler or interpreted. A compiler or interpreter is tool that converts the program into machine language. Each programming has its own compiler or interpreter. Programming language can be classified into three broad categories: **1) Machine Language 2) Assembly Language and 3) High Level Language.**

i) Machine Language:

It is the language of the computer machine itself. A machine can understand this language directly without any translator. It is also known as Binary language. It is only two digits language i.e. 0s and 1s. It is first generation of the programming languages. Machine language was used when the computer was of a huge size; approximately it covered one room.

Only the developers could work with the computer. Because it was not easy as it is today. Input was firstly converted into the Binary language of zeros and ones. Then fed to computer, computer processed it and output is generated. This output is also in the form of machine language. Again it was converted to human understandable language English. Thus it was a headache giver task.

Machine language is the fundamental language of the machines. Each machine we see today works on the machine language. We give input in English and machine works on the machine code. Some interfaces are there that converts our language into machine language. The circuitry of a machine is wired in such a way that it immediately recognizes the machine language and operates the machine

Machine language is not very easy language to learn. It is difficult to read and understand. The most important advantage of machine language is that programs written in machine language are executed very fast by the computer. This is mainly because machine language is directly understood by the CPU and no translation of the program is required. However writing a program in machine language has several disadvantages; Machine Dependent, Difficult to program and Difficult to modify. Machine dependent means the machine code of one machine will not work for other machine.

Machine Language is the toughest language of all the generations. Only developers of this language and experienced scientists can use this language. New learner or normal users can’t even understand it because of its complexity.

ii) Assembly Language:

In machine language we write the programs in binary code i.e. 0s and 1s. That is headache giver task. But this headache is reduced by the Assembly Language. In Assembly Language, programs can be easily written in alphanumeric symbols instead of 0s and 1s. For good and effective programming, meaningful and easily remember able symbols are selected. For example: ADD for addition, SUB for subtraction, CMP for comparison etc. Such symbols are known as **Mnemonics**. A language which uses these mnemonics symbols is known as Assembly Language. A program written in assembly language is known as assembly language program.

The advantage of assembly language is that the execution time of an assembly language program is less. An assembly language program runs faster to produce results.

The most disadvantage of assembly language is that programming in this language is difficult and time consuming. Assembly language is machine dependent. The programmer must have detailed knowledge of the structure of the computer. He must have the knowledge of registry of the computer. The program written in assembly language for one computer cannot be used in any other computer. It means that the assembly language program is not portable. Each processor has its own instruction sets and hence its own assemble language.

iii). High Level Language:

A language, in which instructions are written, is called High Level Language. The instructions written in a high level language are called statements. These statements are closer to

English and mathematics as compared to mnemonics in assembly languages. Example of high level languages are BASIC, PASCAL, FORTRAN, COBOL, ADA, C, C++, C# and JAVA etc.

High level languages are independent of computer architecture. A programmer does not need knowledge of architecture of the computer. The programming is easier. The same program can run on any other computer which has a compiler of that language. The compiler is machine dependent but not the language. Thus language is machine independent.

High level languages are very similar to English like language. These are easy to learn and use. The programmer needs not to learn anything about the computer. He need not to worry about how to store the code in the computer, where to store them, what to do with them etc. All this was in the previous languages.

Writing programs in high level languages requires less time and effort. Programs written in high level languages are easier to maintain than assembly language or machine language programs. This is because these are easier to understand, easier to correct and modify. Insertion or removal of instructions from a program is also possible.

Disadvantage of high level language is that it takes more time to run and requires more computers' main memory.

iv). Source and Object Language:

The language in which a programmer writes programs is called source language. It may be high level language or an assembly language. The language in which the computer works is called object language or machine language. A program written in a source language is called a source program. When source program is converted into machine code by an assembler or compiler it is known as an object program. In other words, a machine code program ready for execution is called an object program.

3. Translators

i). Assembler:

A computer will not understand any program written in a language, other than its machine language. The programs written in other languages must be translated into the machine language. Such translation is performed with the help of software. A program which translates an assembly language program into a machine language program is called an assembler. If an assembler which runs on a computer and produces the machine codes for the same computer then it is called self assembler or resident assembler. If an assembler that runs on a computer and produces the machine codes for other computer then it is called Cross Assembler.

Assemblers are further divided into two types: One Pass Assembler and Two Pass Assembler. One pass assembler is the assembler which assigns the memory addresses to the variables and translates the source code into machine code in the first pass simultaneously. A Two Pass Assembler is the assembler which reads the source code twice. In the first pass, it reads all the variables and assigns them memory addresses. In the second pass, it reads the source code and translates the code into object code.

ii). Compiler:

It is a program which translates a high level language program into a machine language program. A compiler is more intelligent than an assembler. It checks all kinds of limits, ranges, errors etc. But its program run time is more and occupies a larger part of the memory. It has slow speed. Because a compiler goes through the entire program and then translates the entire program into machine codes. If a compiler runs on a computer and produces the machine codes for the same computer then it is known as a self compiler or resident compiler. On the other hand, if a compiler runs on a computer and produces the machine codes for other computer then it is known as a cross compiler.

iii). Interpreter:

An interpreter is a program which translates statements of a program into machine code. It translates only one statement of the program at a time. It reads only one statement of program, translates it and executes it. Then it reads the next statement of the program again translates it and executes it. In this way it proceeds further till all the statements are translated and executed. On the other hand, a compiler goes through the entire program and then translates the entire program into machine codes. A compiler is 5 to 25 times faster than an interpreter.

By the compiler, the machine codes are saved permanently for future reference. On the other hand, the machine codes produced by interpreter are not saved. An interpreter is a small program as compared to compiler. It occupies less memory space, so it can be used in a smaller system which has limited memory space.

iv). Linker:

In high level languages, some built in header files or libraries are stored. These libraries are predefined and these contain basic functions which are essential for executing the program. These functions are linked to the libraries by a program called Linker. If linker does not find a library of a function then it informs to compiler and then compiler generates an error. The compiler automatically invokes the linker as the last step in compiling a program.

Not built in libraries, it also links the user defined functions to the user defined libraries. Usually a longer program is divided into smaller subprograms called modules. And these modules must be combined to execute the program. The process of combining the modules is done by the linker.

v). Loader:

Loader is a program that loads machine codes of a program into the system memory. In Computing, a **loader** is the part of an Operating System that is responsible for loading programs. It is one of the essential stages in the process of starting a program. Because it places programs into memory and prepares them for execution. Loading a program involves reading the contents of executable file into memory. Once loading is complete, the operating system starts the program by passing control to the loaded program code. All operating systems that support program loading have loaders. In many operating systems the loader is permanently resident in memory.

4. Program Development Steps:

i). Algorithm:

It is a sequence of instructions (or set of instructions) to make a program more readable. It is a process used to answer a question. In simple words an algorithm is a step-by-step procedure for solving a problem. Algorithms can be expressed in any language, from natural languages like English to programming languages like C. We use algorithms every day. For example, a recipe for baking a cake is an algorithm. Most programs consist of algorithms. Making algorithm is one of the principal challenges in programming language.*An algorithm must always terminate after a finite number of steps. Simple example of algorithm to add two numbers:*

An algorithm must have one starting point and one or more ending point. Its starting point can be labeled as START and its ending point can be labeled as STOP. Here is example of one starting point and one ending point.

Example of adding two numbers:

Step 1: *Start.*

Step 2: *Take three variables named num1, num2, num3.*

Step 3: *Input the values in two variables: num1 = 10, num2 = 5.*

Step 4: *Now add num1 & num2 and put its value to num3: num3 = num1 + num2. Here both operands num1 and num2 are added and its combined value is assigned to num3.*

Step 5: *Now print the value of num3. Your result is displayed on the screen.*

Step 6: *Stop.*

Example of finding bigger number between two numbers with one START and two STOP:

Step 1: *START.*

Step 2: *Take two variables named num1, num2.*

Step 3: *Read num1 and num2.*

Step 4: *if num1>num2 then*

Step 5: *print num1.*

Step 6: *STOP.*

Step 7: *else*

Step 8: *print num2.*

Step 9: *STOP.*

5. Introduction to C Language:

i). Introduction (History):

C language was designed by Dennis Ritchie at Bell Laboratories between 1969 and 1972. There were many developments of C language between 1969 and 1972. And final language came into existence in 1972. Like other languages C language was undergone a number of versions. In every new version many new features were added to make it more powerful and more useful. It was not written as a teaching help, but as an implementation language. It was developed for developing operating systems, utility programs and compilers. C is a powerful structured and module programming. Like most high level languages, C is a modular programming language. In which each task can be developed as a module. It was written when computers were big and they capital equipments.

C language is mainly influenced by the language B, which was further influenced by the BCPL language. C language was also influenced by the languages such as CPL, Assembly, FORTRAN and ALGOL. This C language further influenced some languages such as C++, C--, C sharp, Java, JavaScript and Perl etc.

C language is known as middle level language because it consist the features of low level language and high level language. Low level features are used for low level programming and High level features are used for high level programming and mostly the high level programming is developed.

The first major use of C language was to write an operating system called UNIX. The success of UNIX operating system and its features brought the consequent popularity for the C language. It was standardized in 1989 by ANSI (American National Standards Institute) known as ANSI C.

ii). Features:

C language is known as a middle level language, because it contains the features of both low level language and high level language. Its features are as below:

- **Clarity:** C language is quite close to English language. So its syntax is clearer and more understandable. Its variables, names etc are written in simple English.
- **Portability:** C language is a portable and can be run on any machine. It is independent of the type of CPU and machine.
- **Built in:** Various built in functions are available in C to perform required operations.
- **Modular:** C language is a modular programming language which makes a “Divide and Conquer” approach. In this approach we can module the programs and they are now easy to manage and understand.
- **Quicker Compiler:** The speed of executing the C programs is very fast. A program in C, created on one computer can be compiled and run on any other machine that has a similar C compiler.
- **Simplicity:** C language is easy to learn and easy to debug (Debug means to make a program error free). Because of its easiness it is widely used from micro computers to mainframe computers.

- **Detect ability:** It is a User Friendly language. C compiler detects the errors easily and displays along with the program. It means C compiler tells us that where the errors are. It tells us the type and number of the line in which the error is present.
- **Location of Data:** It supports to the Pointer concept which tells us that where variables are stored in the computer memory.
- C language is **case sensitive** language. It means lower case and upper case letters are considered as different.

6. Simple C program:

A C program consists of one or more functions or code modules. These are essentially groups of instructions that are to be executed as a unit in a given order and that can be referenced by a unique name. Each C program must contain a ***main()*** function. This is the first function called when the program starts to run. Note that while "main" is not a C keyword and hence not reserved it should be used only in this context.

A C program is traditionally arranged in the following order but not strictly as a rule.

Function prototypes and global data declarations
The main() function
Function definitions

Consider first a simple C program which simply prints a line of text to the computer screen. This is traditionally the first C program you will see and is commonly called the “Hello World” program for obvious reasons.

```
#include <stdio.h>
void main()
{
/* This is how comments are implemented in C
                to comment out a block of text */
// or like this for a single line comment

printf( "Hello World\n" );

}
```

As you can see this program consists of just one function the mandatory *main* function. The parentheses, ( ), after the word main indicate a function while the curly braces, { }, are used to denote a block of code -- in this case the sequence of instructions that make up the function.

Comments are contained within a /* ... */ pair in the case of a block comment or a double forward slash, //, may be used to comment out the remains of a single line of test.

The line

printf("Hello World\n ") ;

is the only C statement in the program and <u>must</u> be terminated by a semi-colon.

The statement calls a function called *printf* which causes its argument, the string of text within the quotation marks, to be printed to the screen. The characters \n are not printed as these characters are interpreted as special characters by the printf function in this case printing out a newline on the screen. These characters are called *escape sequences* in C and cause special actions to occur and are preceded always by the backslash character, \ .

All C compiler include a library of standard C functions such as printf which allow the programmer to carry out routine tasks such as I/O, maths operations, etc. but which are not part of the C language, the compiled C code merely being provided with the compiler in a standard form.

Header files must be included which contain *prototypes* for the standard library functions and declarations for the various variables or constants needed. These are normally denoted by a .h extension and are processed automatically by a program called the *Preprocessor* prior to the actual compilation of the C program.

The line

#include <stdio.h>

Instructs the preprocessor to include the file stdio.h into the program before compilation so that the definitions for the standard input/output functions including printf will be present for the compiler. The angle braces denote that the compiler should look in the default "INCLUDE" directory for this file. A pair of double quotes indicates that the compiler should search in the specified path e.g.

#include "d:\myfile.h"

Note : C is case sensitive i.e. printf() and Printf() would be regarded as two different functions.

7. Variables, Data Types, I/O and Operators

In order to be useful a program must be able to represent real life quantities or data e.g. a person's name, age, height, bank balance, etc. This data will be stored in memory locations called variables that we will name ourselves. However so that the data may be represented as aptly as possible the variables will have to be of different types to suit their data. For example while an integer can represent the age of a person reasonably well it won't be able to represent the pounds and pence in a bank balance or the name of an individual quite so well.

8. Basic Data Types

There are five basic data types char, int, float, double, and void. All other data types in C are based on these. Note that the size of an int depends on the standard size of an integer on a particular operating system.

char	1 byte (8 bits) with range -128 to 127
int	16-bit OS : 2 bytes with range -32768 to 32767 32-bit OS : 4 bytes with range -2,147,483,648 to 2,147,483,647
float	4 bytes with range 10^{-38} to 10^{38} with 7 digits of precision
double	8 bytes with range 10^{-308} to 10^{308} with 15 digits of precision
void	generic pointer, used to indicate no function parameters etc.

Modifying Basic Types

Except for type ***void*** the meaning of the above basic types may be altered when combined with the following keywords.

a. signed
b. unsigned
c. long
d. short

The *signed* and *unsigned* modifiers may be applied to types char and int and will simply change the range of possible values. For example an *unsigned char* has a range of 0 to 255, all positive, as opposed to a signed char which has a range of -128 to 127. An *unsigned integer* on a 16-bit system has a range of 0 to 65535 as opposed to a *signed int* which has a range of -32768 to 32767. Note however that the default for type int or char is signed so that the type *signed char* is always equivalent to type *char* and the type *signed int* is always equivalent to *int*.

The *long* modifier may be applied to type int and double only. A *long int* will require 4 bytes of storage no matter what operating system is in use and has a range of -2,147,483,648 to 2,147,483,647. A *long double* will require 10 bytes of storage and will be able to maintain up to 19 digits of precision. The *short* modifier may be applied only to type *int* and will give a 2 byte integer independent of the operating system in use.

Note: Note that the keyword *int* may be omitted without error so that the type *unsigned* is the same as type *unsigned int,* the type *long* is equivalent to the type *long int,* and the type *short* is equivalent to the type *short int.*

9. Variables:

A variable is a named piece of memory which is used to hold a value which may be modified by the program. A variable thus has three attributes that are of interest to us : its **type,** its **value** and its **address.**

The variable's type informs us what type and range of values it can represent and how much memory is used to store that value. The variable's address informs us where in memory the variable is located (which will become increasingly important when we discuss pointers later on).

All C variables must be declared as follows:-

type variable-list ;

For Example :-

```
int i ;
char a, b, ch ;
```

Variables are declared in three general areas in a C program. When declared inside functions as follows they are termed **local** variables and are visible (or accessible) within the function (or code block) only.

```
void main()
{
int i, j ;
...
}
```

A local variable is created i.e. allocated memory for storage upon entry into the code block in which it is declared and is destroyed i.e. its memory is released on exit. This means that values cannot be stored in these variables for use in any subsequent calls to the function .

When declared outside functions they are termed **global** variables and are visible throughout the file or have file scope. These variables are created at program start-up and can be used for the lifetime of the program.

```
int i ;
void main()
{
...
}
```

When declared within the braces of a function they are termed the formal parameters of the function as we will see later on.

```
int func1( int a, char b ) ;
```

i) Variable Names:

Names of variables and functions in C are called identifiers and are case sensitive. The first character of an identifier must be either a letter or an underscore while the remaining characters may be letters, numbers, or underscores. Identifiers in C can be up to 31 characters in length.

ii) Initializing Variables:

When variables are declared in a program it just means that an appropriate amount of memory is allocated to them for their exclusive use. This memory however is **not initialised** to zero or to any other value automatically and so will contain random values unless specifically initialised before use.

***Syntax* :- type var-name = constant ;**

For Example :-

```
char ch = 'a' ;
double d = 12.2323 ;
int i, j = 20 ; /* note in this case  i is not initialised */
```

iii) Storage Classes

There are four storage class modifiers used in C which determine an identifier’s storage duration and scope.

a) auto
b) static
c) register
d) extern

An identifier’s storage duration is the period during which that identifier exists in memory. Some identifiers exist for a short time only, some are repeatedly created and destroyed and some exist for the entire duration of the program. An identifier’s scope specifies what sections of code it is accessible from.

The auto storage class is implicitly the default storage class used and simply specifies a normal local variable which is visible within its own code block only and which is created and destroyed automatically upon entry and exit respectively from the code block.

The register storage class also specifies a normal local variable but it also requests that the compiler store a variable so that it may be accessed as quickly as possible, possibly from a CPU register.

The static storage class causes a local variable to become permanent within its own code block i.e. it retains its memory space and hence its value between function calls.

When applied to global variables the static modifier causes them to be visible only within the physical source file that contains them i.e. to have file scope. Whereas the extern modifier which is the implicit default for global variables enables them to be accessed in more than one source file.

For example in the case where there are two C source code files to be compiled together to give one executable and where one specific global variable needs to be used by both the extern class allows the programmer to inform the compiler of the existence of this global variable in both files.

iv). Constants

Constants are fixed values that cannot be altered by the program and can be numbers, characters or strings.

Some Examples :-

```
char :  'a', '$', '7'
int :  10, 100, -100
unsigned :  0, 255
float :  12.23456, -1.573765e10, 1.347654E-13
double :  1433.34534545454, 1.35456456456456E-200
long :  65536, 2222222
string : “Hello World\n”
```

Note : Floating point constants default to type double. For example the following code segment will cause the compiler to issue a warning pertaining to floating point conversion in the case of f_val but not in the case of d_val..

```
float f_val ;
double d_val ;
f_val = 123.345 ;
d_val = 123.345 ;
```

However the value may be coerced to type float by the use of a modifier as follows :-

f = 123.345**F** ;

Integer constants may also be forced to be a certain type as follows :-

```
100U --- unsigned
100L --- long
```

Integer constants may be represented as either decimal which is the default, as hexadecimal when preceded by "0x", e.g. 0x2A, or as octal when preceded by "O", e.g. O27.

Character constants are normally represented between single quotes, e.g. 'a', 'b', etc. However they may also be represented using their ASCII (or decimal) values e.g. 97 is the ASCII value for the letter 'a', and so the following two statements are equivalent. (See Appendix A for a listing of the first 128 ASCII codes.)

```
char ch = 97 ;
char ch = 'a' ;
```

There are also a number of special character constants sometimes called *Escape Sequences,* which are preceded by the backslash character '\', and have special meanings in C.

\n	newline
\t	tab
\b	backspace
\'	single quote
\"	double quote
\0	null character
\xdd	represent as hexadecimal constant

10. Managing Input / Output:

This section introduces some of the more common input and output functions provided in the C standard library.

i). Printf():

The printf() function is used for formatted output and uses a control string which is made up of a series of format specifiers to govern how it prints out the values of the variables or constants required. The more common format specifiers are given below

%c	character	%f	floating point
%d	signed integer	%lf	double floating point
%i	signed integer	%e	exponential notation
%u	unsigned integer	%s	string
%ld	signed long	%x	unsigned hexadecimal
%lu	unsigned long	%o	unsigned octal
		%%	prints a % sign

For Example :-

```
int i ;
printf(  "%d", i ) ;
```

The printf() function takes a variable number of arguments. In the above example two arguments are required, the format string and the variable *i*. The value of *i* is substituted for the format specifier %d which simply specifies how the value is to be displayed, in this case as a signed integer.

Some further examples :-

```
int i = 10, j = 20 ;
char ch = 'a' ;
double f = 23421.2345 ;
printf( "%d + %d", i, j ) ;  /* values are substituted from the variable list  in order as required  */
printf( "%c", ch ) ;
printf( "%s", "Hello World\n" ) ;
printf( "The value of f is : %lf", f ) ;/*Output as : 23421.2345 */
printf( "f in exponential form : %e", f ) ;       /* Output as : 2.34212345e+4
```

Field Width Specifiers

Field width specifiers are used in the control string to format the numbers or characters output appropriately .

Syntax :- **%[total width printed][.decimal places printed]format specifier**

where square braces indicate optional arguments.

For Example :-

```
int i = 15 ;
float f = 13.3576 ;
printf( "%3d", i ) ;   /* prints "_15 " where _ indicates a space character */
printf( "%6.2f", f ) ; /* prints "_13.36" which has a total width of 6 and displays 2 decimal
                          places */
printf( “%*.*f”, 6,2,f ) ;  /* prints  "_13.36" as above. Here * is used as replacement character for
                               field widths     */
```

There are also a number of flags that can be used in conjunction with field width specifiers to modify the output format. These are placed directly after the % sign. A - (minus sign) causes the output to be left-justified within the specified field, a + (plus sign) displays a plus sign preceding positive values and a minus preceding negative values, and a 0 (zero) causes a field to be padded using zeros rather than space characters.

ii). scanf()

This function is similar to the printf function except that it is used for formatted input. The format specifiers have the same meaning as for printf() and the space character or the newline character are normally used as delimiters between different inputs.

For Example :-

```
    int i, d ;
    char c ;
    float f ;
```

```
scanf( "%d", &i ) ;
scanf( "%d %c %f", &d, &c, &f ) ; /* e.g. type "10_x_1.234RET" */
scanf( "%d:%c", &i, &c ) ;                /* e.g.  type "10:xRET"  */
```

The & character is the *address of* operator in C, it returns the address in memory of the variable it acts on. (Aside : This is because C functions are nominally call--by--value. Thus in order to change the value of a calling parameter we must tell the function exactly where the variable resides in memory and so allow the function to alter it directly rather than to uselessly alter a copy of it.)

Note that while the space and newline characters are normally used as delimiters between input fields the actual delimiters specified in the format string of the scanf statement must be reproduced at the keyboard faithfully as in the case of the last sample call. If this is not done the program can produce somewhat erratic results! ‘

The scanf function has a return value which represents the number of fields it was able to convert successfully.

For Example :-

```
num = scanf( “%c %d”, &ch, &i );
```

This scanf call requires two fields, a character and an integer, to be read in so the value placed in *num* after the call should be 2 if this was successful. However if the input was “a bc” then the first character field will be read correctly as ‘a’ but the integer field will not be converted correctly as the function cannot reconcile “bc” as an integer. Thus the function will return 1 indicating that one field was successfully converted. Thus to be safe the return value of the scanf function should be checked always and some appropriate action taken if the value is incorrect.

iii). getchar() and putchar()

These functions are used to input and output single characters. The *getchar()* function reads the ASCII value of a character input at the keyboard and displays the character while *putchar()* displays a character on the standard output device i.e. the screen.

For Example :-

```
char ch1, ch2 ;
ch1 = getchar() ;
ch2 = 'a' ;
putchar( ch2 ) ;
```

Note: The input functions described above, scanf() and getchar() are termed buffered input functions. This means that whatever the user types at the keyboard is first stored in a data buffer and is not actually read into the program until either the buffer fills up and has to be flushed or until the user flushes the buffer by hitting **RET** whereupon the required data is read into the program. The important thing to remember with buffered input is that no matter how much data is taken into the buffer when it is flushed the program just reads as much data as it needs from the start of the buffer allowing whatever else that may be in the buffer to be discarded.

For Example :-

```
char ch1, ch2;
printf( "Enter two characters : " ) ;
ch1 = getchar() ;
ch2 = getchar() ;
printf( "\n The characters are %c and %c\n", ch1, ch2 ) ;
```

In the above code segment if the input is "abcdef**RET**" the first two characters are read into the variables all the others being discarded, but control does not return to the program until the **RET** is hit and the buffer flushed. If the input was "a**RET**" then a would be placed in ch1 and **RET** in ch2.

iv)._flushall()

The _flushall function writes the contents of all output buffers to the screen and clears the contents of all input buffers. The next input operation (if there is one) then reads new data from the input device into the buffers.

This function should be used always in conjunction with the buffered input functions to clear out unwanted characters from the buffer **after each** input call.

v). getch() and getche()

These functions perform the same operation as getchar() except that they are unbuffered input functions i.e. it is not necessary to type **RET** to cause the values to be read into the program they are read in immediately the key is pressed. getche() echoes the character hit to the screen while getch() does not.

For example :-

```
char ch ;
ch = getch() ;
```

11. Operators:

One of the most important features of C is that it has a very rich set of built in operators including arithmetic, relational, logical, and bitwise operators.

i) Assignment Operators:

```
int x ;
x = 20 ;
```

Some common notation :- lvalue -- left hand side of an assignment operation
rvalue -- right hand side of an assignment operation

Type Conversions :- the value of the right hand side of an assignment is converted to the type of the lvalue. This may sometimes yield compiler warnings if information is lost in the conversion.

For Example :-

```
        int x ;
        char ch ;
        float f ;
        ch = x ;/* ch is assigned lower 8 bits of x, the remaining bits are discarded so we have a
        possible information loss  */
        x = f ;  /* x is assigned non fractional part of f only within int range, information loss
        possible  */
        f = x ;  /* value of x is converted to floating point */
```

Multiple assignments are possible to any degree in C, the assignment operator has right to left associativity which means that the rightmost expression is evaluated first.

For Example :-

```
                x = y = z = 100 ;
```

In this case the expression z = 100 is carried out first. This causes the value 100 to be placed in z with the value of the whole expression being 100 also. This expression value is then taken and assigned by the next assignment operator on the left i.e. x = y = (z = 100) ;

ii). Arithmetic Operators:

+ - * / --- same rules as mathematics with * and / being evaluated before + and -.
% -- modulus / remainder operator
For Example :-

```
        int a = 5, b = 2, x ;
        float c = 5.0, d = 2.0, f ;
        x = a / b ;       //  integer division, x = 2.
        f = c / d  ;      //  floating point division, f = 2.5.
        x = 5 % 2 ;       //  remainder operator, x = 1.
        x = 7 + 3 * 6 / 2 - 1 ;// x=15,* and / evaluated ahead of + and -.
```

Note that parentheses may be used to clarify or modify the evaluation of expressions of any type in C in the same way as in normal arithmetic.

```
        x = 7 + ( 3 * 6 / 2 ) - 1 ;        // clarifies order of evaluation without penalty
        x = ( 7 + 3 ) * 6 / ( 2 - 1 ) ;    // changes order of evaluation, x = 60 now.
```

iii). Increment and Decrement Operators:

There are two special unary operators in C, Increment ++, and Decrement -- , which cause the variable they act on to be incremented or decremented by 1 respectively.

For Example :-

```
                x++ ;    /* equivalent to        x = x + 1 ;    */
```

++ and -- can be used in prefix or postfix notation. In prefix notation the value of the variable is either incremented or decremented and is then read while in postfix notation the value of the variable is read first and is then incremented or decremented.

For Example :-

```
        int i,  j = 2 ;
        i = ++ j  ;        /* prefix  :-   i has value 3, j has value 3  */
        i = j++ ;          /* postfix  :-  i  has value 3, j has value 4   */
```

iv). Special Assignment Operators:

Many C operators can be combined with the assignment operator as shorthand notation

For Example :-

```
        x = x + 10 ;
can be replaced by
        x += 10 ;
```

Similarly for -=, *=, /=, %=, etc.

These shorthand operators improve the speed of execution as they require the expression, the variable x in the above example, to be evaluated once rather than twice.

v). Relational Operators:

The full set of relational operators are provided in shorthand notation

```
        >      >=      <      <=      ==      !=
```

For Example :-

```
        if ( x == 2 )
                printf( “x is equal to 2\n” ) ;
```

vi). Logical Operators:

&&	--	Logical AND
||	--	Logical OR
!	--	Logical NOT

For Example :-

```
        if (  x >= 0 && x < 10  )
                printf( “ x is greater than or equal to zero and less than ten.\n” ) ;
```

Note: There is no Boolean type in C so TRUE and FALSE are deemed to have the following meanings.

FALSE -- value zero

TRUE -- any non-zero value but 1 in the case of in-built relational operations

For Example :-

```
2 > 1          -- TRUE  so expression has value 1
2 > 3          -- FALSE so expression has value 0
i = 2 > 1  ;   --  relation is TRUE -- has value 1, i is assigned value 1
```

Note : Every C expression has a value. Typically we regard expressions like 2 + 3 as the only expressions with actual numeric values. However the relation 2 > 1 is an expression which evaluates to TRUE so it has a value 1 in C. Likewise if we have an expression x = 10 this has a value which in this case is 10 the value actually assigned.

Note : Beware of the following common source of error. If we want to test if a variable has a particular value we would write for example

```
if ( x == 10 )  …
```

But if this is inadvertently written as

```
if ( x = 10 ) …
```

this will give no compilation error to warn us but will compile and assign a value 10 to x when the condition is tested. As this value is non-zero the if condition is deemed true no matter what value x had originally. Obviously this is possibly a serious logical flaw in a program.

vii). Bitwise Operators:

These are special operators that act on **char or int arguments only**. They allow the programmer to get closer to the machine level by operating at bit-level in their arguments.

&	Bitwise AND	|	Bitwise OR
^	Bitwise XOR	~	Ones Complement
>>	Shift Right	<<	Shift left

Recall that type char is one byte in size. This means it is made up of 8 distinct bits or binary digits normally designated as illustrated below with Bit 0 being the Least Significant Bit (LSB) and Bit 7 being the Most Significant Bit (MSB). The value represented below is 13 in decimal.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	1	1	0	1

An integer on a 16 bit OS is two bytes in size and so Bit 15 will be the MSB while on a 32 bit system the integer is four bytes in size with Bit 31 as the MSB.

Bitwise AND, &

RULE : If any two bits in the same bit position are set then the resultant bit in that position is set otherwise it is zero.

For Example :-

```
                    1011 0010         (178)
                                  &         0011 1111     (63)
                                  =         0011 0010     (50)
```

Bitwise OR, |

RULE : If either bit in corresponding positions are set the resultant bit in that position is set.

For Example :-

```
                                        1011 0010              (178)
                                 |        0000 1000             (63)
                              =          1011 1010              (186)
```

Bitwise XOR, ^

RULE : If the bits in corresponding positions are different then the resultant bit is set.

For Example :-

```
                                      1011 0010                    (178)
                               ^     0011 1100                   (63)
                              =            1000 1110             (142)
```

Shift Operators, << and >>

RULE : These move all bits in the operand left or right by a specified number of places.

Syntax : **variable << number of places**
variable >> number of places

For Example :-

2 << 2 = 8

i.e.

0000 0010 becomes 0000 1000

NB : shift left by one place multiplies by 2
shift right by one place divides by 2

Ones Complement

RULE : Reverses the state of each bit.

For Example :-

1101 0011 becomes 0010 1100

Note : With all of the above bitwise operators we must work with decimal, octal, or hexadecimal values as binary is not supported directly in C.

The bitwise operators are most commonly used in system level programming where individual bits of an integer will represent certain real life entities which are either on or off, one or zero. The programmer will need to be able to manipulate individual bits directly in these situations.

A *mask* variable which allows us to ignore certain bit positions and concentrate the operation only on those of specific interest to us is almost always used in these situations. The value given to the mask variable depends on the operator being used and the result required.

For Example :- To clear bit 7 of a char variable.

```
char ch = 89 ;          // any value
char mask = 127 ;       // 0111 1111
ch = ch & mask ;        // or  ch &= mask ;
```

For Example :- To set bit 1 of an integer variable.

```
int i = 234 ;           // any value
int mask = 2 ;          // a 1 in bit position 2
i |= mask ;
```

viii). Sizeof Operator

The sizeof operator gives the amount of storage, in bytes, associated with a variable or a type (including aggregate types as we will see later on).The expression is either an identifier or a type-cast expression (a type specifier enclosed in parentheses).

Syntax : sizeof (expression)

For Example :-

```
int x , size ;
size = sizeof ( x ) ;
printf("The integer x requires %d bytes on this machine", size);
printf( "Doubles take up %d bytes on this machine", sizeof ( double ) ) ;
```

12. Precedence and Associativity of Operators:

When several operations are combined into one C expression the compiler has to rely on a strict set of precedence rules to decide which operation will take preference. The precedence of C operators is given below.

Precedence	**Operator**	**Associativity**
Highest	( ) [ ] -> .	left to right
	! ~ ++ -- +(unary) -(unary) (type) * & sizeof	right to left
	* / %	left to right
	+ -	left to right
	<< >>	left to right
	< <= > >=	left to right
	== !=	left to right
	&	left to right
	^	left to right
	|	left to right
	&&	left to right
	||	left to right
	? :	right to left
	= += -= *= /= %= &= ^= |= <<= >>=	right to left
Lowest	,	left to right

Operators at the top of the table have highest precedence and when combined with other operators at the same expression level will be evaluated first.

For example take the expression

```
2 + 10 * 5 ;
```

Here * and + are being applied at the same level in the expression but which comes first ? The answer lies in the precedence table where the * is at a higher level than the + and so will be applied first.

When two operators with the same precedence level are applied at the same expression level the associativity of the operators comes into play.

For example in the expression

```
2 + 3 - 4 ;
```

the + and - operators are at the same precedence level but associate from left to right and so the addition will be performed first. However in the expression

```
x = y = 2 ;
```

as we have noted already the assignment operator associates from right to left and so the rightmost assignment is first performed.

Note : As we have seen already parentheses can be used to supersede the precedence rules and force evaluation along the lines we require. For example to force the addition in **2 + 10 * 5 ;** to be carried out first we would write it as **(2 + 10) * 5;**

13. Implicit & Explicit Type Conversions:

Normally in mixed type expressions all operands are converted **temporarily** up to the type of the largest operand in the expression.

Normally this automatic or implicit casting of operands follows the following guidelines in ascending order.

long double
double
float
unsigned long
long
unsigned int
signed int

For Example :-

```
int i ;
float f1, f2 ;
f1 = f2 + i ;
```

Since f2 is a floating point variable the value contained in the integer variable is temporarily converted or ***cast to*** a floating point variable also to standardise the addition operation in this case. However it is important to realise that no permanent modification is made to the integer variable.

Explicit casting coerces the expression to be of specific type and is carried out by means of the **cast operator** which has the following syntax.

Syntax : **(type) expression**

For Example if we have an integer x, and we wish to use floating point division in the expression x/2 we might do the following

```
( float ) x  /  2
```

which causes x to be temporarily cast to a floating point value and then implicit casting causes the whole operation to be floating point division.

The same results could be achieved by stating the operation as

```
x  /  2.0
```

which essentially does the same thing but the former is more obvious and descriptive of what is happening.

Note: It should be noted that all of these casting operations, both implicit and explicit, require processor time. Therefore for optimum efficiency the number of conversions should be kept to a minimum.