UNIT – I

INTRODUCTION TO C

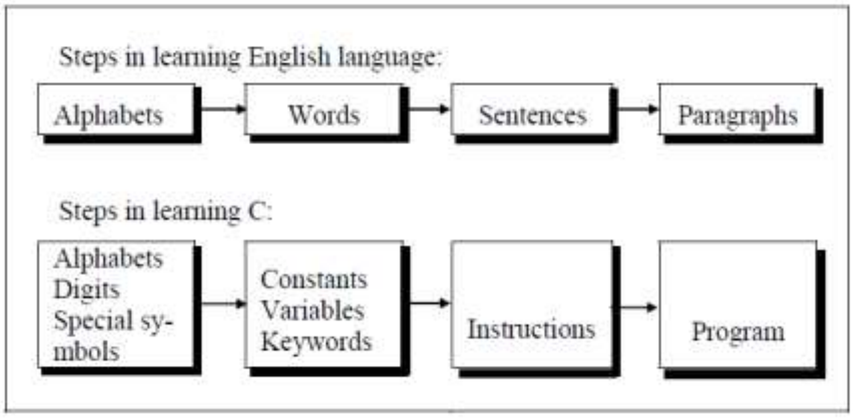
C is a programming language developed at AT & T's Bell Laboratories of USA in 1972. It was designed and written by a man named Dennis Ritchie. In the late seventies C began to replace the more familiar languages of that time like PL/I, ALGOL, etc.

ANSI C standard emerged in the early 1980s, this book was split into two titles: The original was still called Programming in C, and the title that covered ANSI C was called Programming in ANSI C. This was done because it took several years for the compiler vendors to release their ANSI C compilers and for them to become ubiquitous. It was initially designed for programming UNIX operating system. Now the software tool as well as the C compiler is written in C.

Major parts of popular operating systems like Windows, UNIX, Linux is still written in C. This is because even today when it comes to performance (speed of execution) nothing beats C. Moreover, if one is to extend the operating system to work with new devices one needs to write device driver programs. These programs are exclusively written in C. C seems so popular is because it is reliable, simple and easy to use. Often heard today is – "C has been already superseded by languages like C++, C# and Java".

PROGRAM:

There is a close analogy between learning English language and learning C language. The classical method of learning English is to first learn the alphabets used in the language, then learn to combine these alphabets to form words, which in turn are combined to form sentences and sentences are combined to form paragraphs. Learning C is similar and easier. Instead of straight-away learning how to write programs, we must first know what alphabets, numbers and special symbols are used in C, then how using them constants, variables and keywords are constructed, and finally how are these combined to form an instruction. A group of instructions would be combined later on to form a program. So a computer program is just a collection of the instructions necessary to solve a specific problem. The basic operations of a computer system form what is known as the computer's instruction set. And the approach or method that is used to solve the problem is known as an algorithm.


Steps in learning English language:
Alphabets
Words
Sentences
Paragraphs
Steps in learning C:
Alphabets Digits Special sy-mbols
Constants Variables Keywords
Instructions
Program

Programming languages are of two types:

1) Low level languages 2) High level languages

LOW LEVEL LANGUAGE:

Low level languages are machine level and assembly level language. In machine level language computer only understand digital numbers i.e. in the form of 0 and 1. So, instruction given to the computer is in the form binary digit, which is difficult to implement instruction in binary code. This type of program is not portable, difficult to maintain and also error prone. The assembly language is on other hand modified version of machine level language. Where instructions are given in English like word as ADD, SUM, MOV etc. It is easy to write and understand but not understand by the machine. So the translator used here is assembler to translate into machine level. Although language is bit easier, programmer has to know low level details related to low level language. In the assembly level language the data are stored in the computer register, which varies for different computer. Hence it is not portable.

HIGH LEVEL LANGUAGE:

These languages are machine independent, means it is portable. The language in this category is Pascal, COBOL, Fortran etc. High level languages are understood by the machine. So it need to translate by the translator into machine level. A translator is software which is used to translate high level language as well as low level language in to machine level language.

Three types of translators are there:

1) Compiler 2) Interpreter 3) Assembler

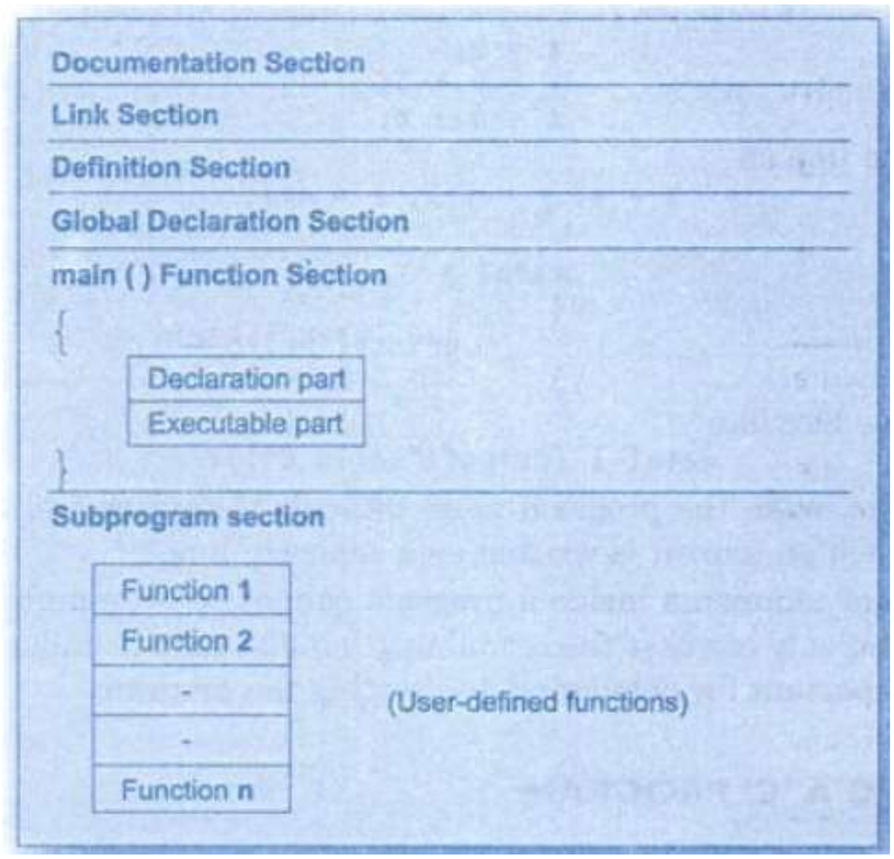
Compiler and interpreter are used to convert the high level language into machine level language. The program written in high level language is known as source program and the corresponding machine level language program is called as object program. Both compiler and interpreter perform the same task but there working is different. Compiler read the program at-a-time and searches the error and lists them. If the program is error free then it is converted into object program. When program size is large then compiler is preferred. Whereas interpreter read only one line of the source code and convert it to object code. If it check error, statement by statement and hence of take more time.

INTEGRATED DEVELOPMENT ENVIRONMENTS (IDE)

The process of editing, compiling, running, and debugging programs is often managed by a single integrated application known as an Integrated Development Environment, or IDE for short. An IDE is a windows-based program that allows us to easily manage large software programs, edit files in windows, and compile, link, run, and debug programs.

On Mac OS X, CodeWarrior and Xcode are two IDEs that are used by many programmers. Under Windows, Microsoft Visual Studio is a good example of a popular IDE. Kylix is a popular IDE for developing applications under Linux. Most IDEs also support program development in several different programming languages in addition to C, such as C# and C++.

BASIC STRUCTURE OF C PROGRAM



- Documentation section contains comment statements
- Link section contains preprocessor directive statements
- Definition section contains symbolic constants definitions
- Global declaration section contains global variables declaration statements
- Every C program must contain ***main*** function
- It is not possible to run a program without ***main***
- ***main*** is a special name it is not user defined word nor reserved word.
- Operating system calls ***main*** function as soon as we run the program.
- ***main*** is user defined function.
- Body of the main function contains declaration as well as executable statements
- Declaration statements can be used to declare local variables to be used in that function
- Sub program section contains user defined functions
- Function is nothing but group of statements that is designed to perform one specified task

COMMENT LINE:

It indicates the purpose of the program. It is represented as

/*…………………………….*/

Comment lines are used for increasing the readability of the program. They are useful in explaining the program and generally used for documentation. They are enclosed within the delimiters. Comments can be single or multiple lines but should not be nested. It can be anywhere in the program except inside string constant & character constant.

PREPROCESSOR DIRECTIVE:

#include<stdio.h> tells the compiler to include information about the standard input/output library.

It is also used in defining symbolic constant such as #define PI 3.14(value).

The stdio.h (standard input output header file) contains definitions & declarations of pre-defined functions such as printf( ), scanf( ), getchar(), etc.

GLOBAL DECLARATION:

This is the section where variables are declared globally so that they can be accessed by all the functions used in the program and it is generally declared outside the function.

main()

It is the user defined function and every program has one main() function from where actually program execution begins and it is enclosed within the pair of curly braces. The main( ) function can be anywhere in the program but in general practice it is placed in the first position.

Syntax:

```
main()
{
……..
……..
……..
}
```

The main( ) function return value when it declared by data type as

```
int main()
{
……..
……..
……..
return 0;
}
```

The main function does not return any value when void (means null/empty) is specified.

```
void main(void ) or void main()
{
        printf(“C Language”);
}
```

Output: C Language

The program execution start with opening brace and end with closing brace. In between the two braces declaration part as well as executable part are mentioned. And at the end of each line, the semi-colon is given which indicates statement termination.

```
/*First C program with return statement*/

#include <stdio.h>
int main (void)
{
        printf ("Welcome to C Programming Language.\n");
        return 0;
}
```

Output: Welcome to C Programming Language.

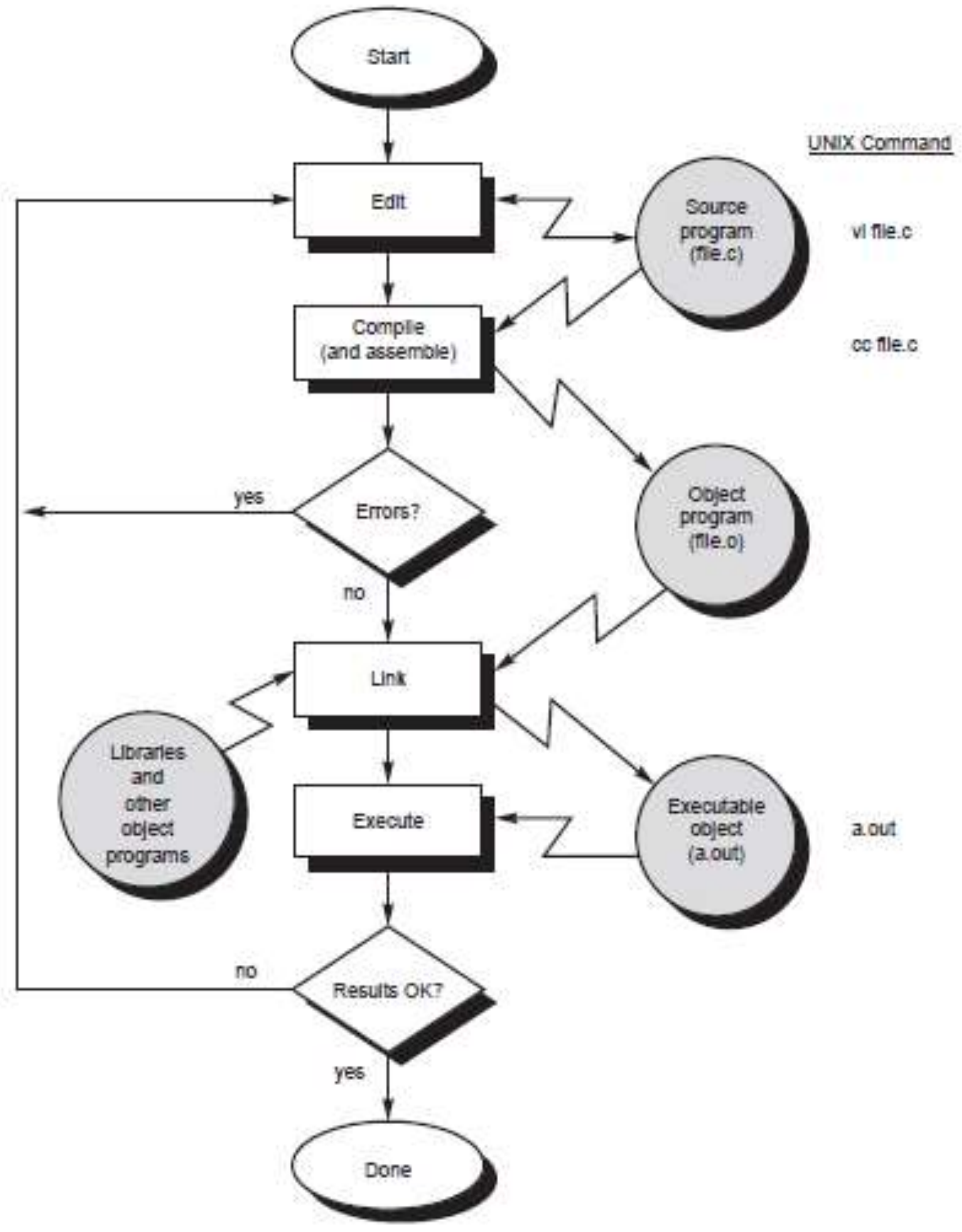
Features of C:

- The C compiler combines the capabilities of an assembly language with the features of a high level language.
- So it is useful for developing both system and application software.
- C is highly portable means that C programs written for one computer can be run on another with little or no modification.
- C is well suited for structured programming language.
- The ability of C is, it can extend itself.
- Programs written in C are efficient and fast.
- C is a structured programming language.
- C is case sensitive language (i.e., upper case and lower case characters are recognized differently).
- It is an example for middle level language.

Steps for Compiling and executing the Programs:

A compiler is a software program that analyzes a program developed in a particular computer language and then translates it into a form that is suitable for execution on a particular computer system.

Figure below shows the steps that are involved in entering, compiling, and executing a computer program developed in the C language and the typical Unix commands that would be entered from the command line.



Step 1: The program that is to be compiled is first typed into a file on the computer system. There are various conventions that are used for naming files, typically be any name provided the last two characters are ".c" or file with extension .c. So, the file name prog1.c might be a valid filename for a C program.

A text editor is usually used to enter the C program into a file. For example, **vi** is a popular text editor used on Unix systems. The program that is entered into the file is known as the source program because it represents the original form of the program expressed in the C language.

Step 2: After the source program has been entered into a file, then proceed to have it compiled. The compilation process is initiated by typing a special command on the system. When this command is entered, the name of the file that contains the source program must also be specified. For example, under Unix, the command to initiate program compilation is called cc. If we are using the popular GNU C compiler, the command we use is gcc.

```
gcc prog1.c or cc prog1.c
```

In the first step of the compilation process, the compiler examines each program statement contained in the source program and checks it to ensure that it conforms to the syntax and semantics of the language. If any mistakes are discovered by the compiler during this phase, they are reported to the user and the compilation process ends right there. The errors then have to be corrected in the source program (with the use of an editor), and the compilation process must be restarted. Typical errors reported during this phase of compilation might be due to an expression that has unbalanced parentheses (syntactic error), or due to the use of a variable that is not "defined" (semantic error).

Step 3: When all the syntactic and semantic errors have been removed from the program, the compiler then proceeds to take each statement of the program and translate it into a "lower" form that is equivalent to assembly language program needed to perform the identical task.

Step 4: After the program has been translated the next step in the compilation process is to translate the assembly language statements into actual machine instructions. The assembler takes each assembly language statement and converts it into a binary format known as object code, which is then written into another file on the system. This file has the same name as the source file under Unix, with the last letter an "o" (for object) instead of a "c".

Step 5: After the program has been translated into object code, it is ready to be linked. This process is once again performed automatically whenever the cc or gcc command is issued under Unix. The purpose of the linking phase is to get the program into a final form for execution on the computer.

If the program uses other programs that were previously processed by the compiler, then during this phase the programs are linked together. Programs that are used from the system's program library are also searched and linked together with the object program during this phase.

The process of compiling and linking a program is often called building.

The final linked file, which is in an executable object code format, is stored in another file on the system, ready to be run or executed. Under Unix, this file is called a.out by default. Under Windows, the executable file usually has the same name as the source file, with the c extension replaced by an exe extension.

Step 6: To subsequently execute the program, the command a.out has the effect of loading the program called a.out into the computer's memory and initiating its execution.

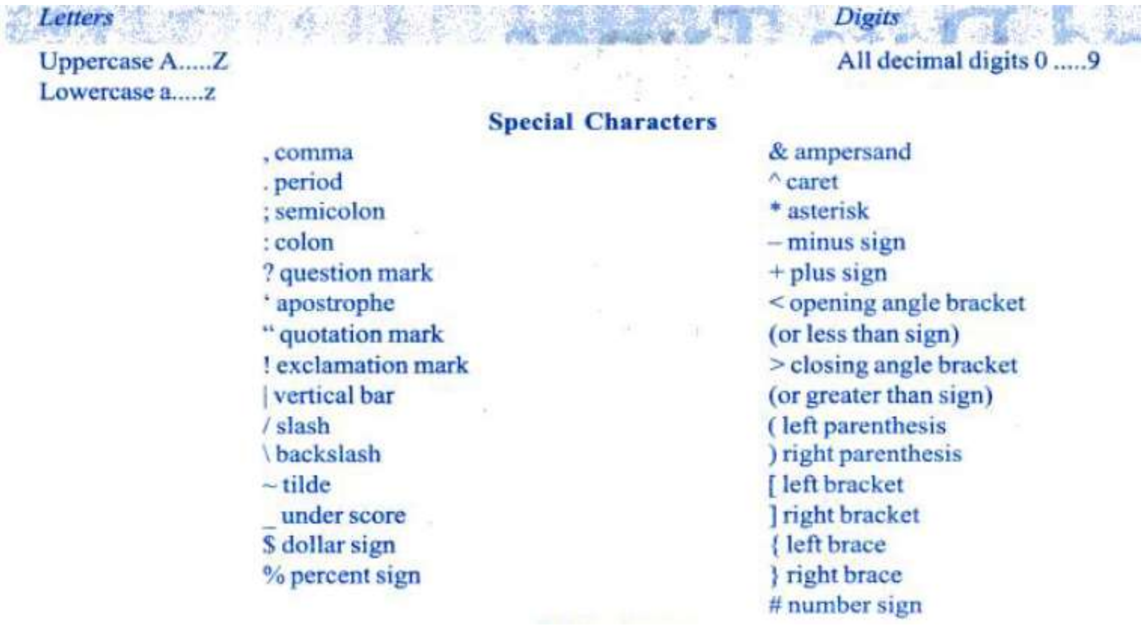
When the program is executed, each of the statements of the program is sequentially executed in turn. If the program requests any data from the user, known as input, the program temporarily suspends its execution so that the input can be entered. Results that are displayed by the program, known as output, appear in a window, sometimes called the console. If the program does not produce the desired results, it is necessary to go back and reanalyze the program's logic. This is known as the debugging phase, during which an attempt is made to remove all the known problems or bugs from the program.

```
/* Sample program to add two numbers*/
#include <stdio.h>
int main (void)
{
int  v1, v2, sum;                    //v1, v2, sum are variables and int is data type
v1 = 150;
v2 = 25;
sum = v1 + v2;
printf ("The sum of %d and %d is = %d\n", v1, v2, sum);
return 0;
}
```

Output: The sum of 150 and 25 is = 175

CHARACTER SET:

A character denotes any alphabet, digit or special symbol used to represent information. Valid alphabets, numbers and special symbols allowed in C are:



Letters

Uppercase A.....Z
Lowercase a.....z

Digits

All decimal digits 0 9

Special Characters

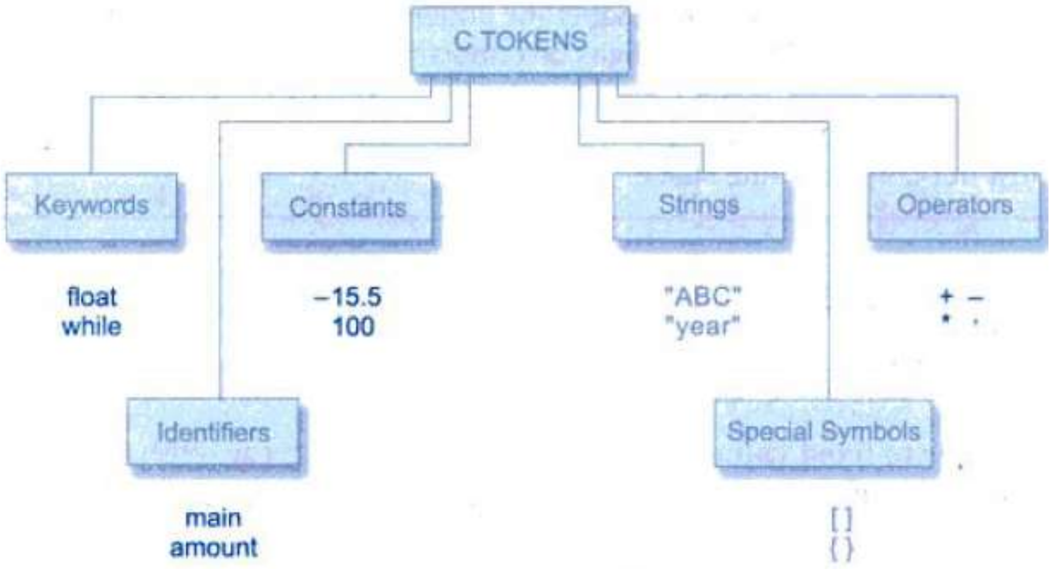
, comma	& ampersand
. period	^ caret
; semicolon	* asterisk
: colon	– minus sign
? question mark	+ plus sign
' apostrophe	< opening angle bracket (or less than sign)
" quotation mark	> closing angle bracket (or greater than sign)
! exclamation mark	(left parenthesis
| vertical bar	) right parenthesis
/ slash	[left bracket
\ backslash	] right bracket
~ tilde	{ left brace
_ under score	} right brace
$ dollar sign	# number sign
% percent sign	

White Spaces
Blank space
Horizontal tab
Carriage return
New line
Form feed

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords.

TOKENS: The Smallest individual units in a program/statement are called tokens.

The tokens are: 1. **Keywords** 2. **Constants** 3. **Identifiers** 4. **Operators** 5. **Special Symbols** 6. **Strings**

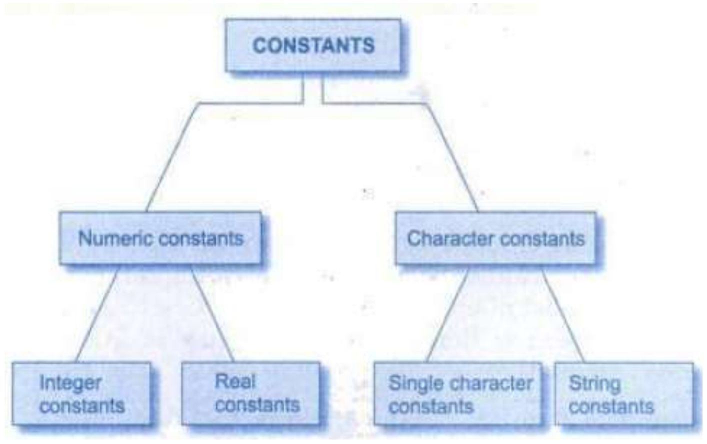


1. Keywords: The C keywords are reserved words by the compiler. All the C keywords have been assigned fixed meaning and they cannot be used as variable names.

There are 32 keywords provided by ANSI C. They are:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

2. Constants: The constants in C have fixed value, which do not change during the execution of program.



a. Numeric Constants

i) Integer Constants: The sequence of numbers from 0 to 9 without decimal or floating points or other symbols. Integer constant may be positive or negative or zero.

Example: 10, -29, 0, -991

Integer constants can also be represented by octal number or hexadecimal number based on requirement.

Example: 027, 037 – octal number

0X9, 0Xab, 0X4 – hexadecimal

ii) Real Constants: Real constants are often known as floating – points.

Example: 2.4, 5.433, 12.32

The real constants can be written in exponential form, which contains fractional part and exponential part.

Example: 2456.123 can be written as 2.4561 X e+3.

b. Character Constants

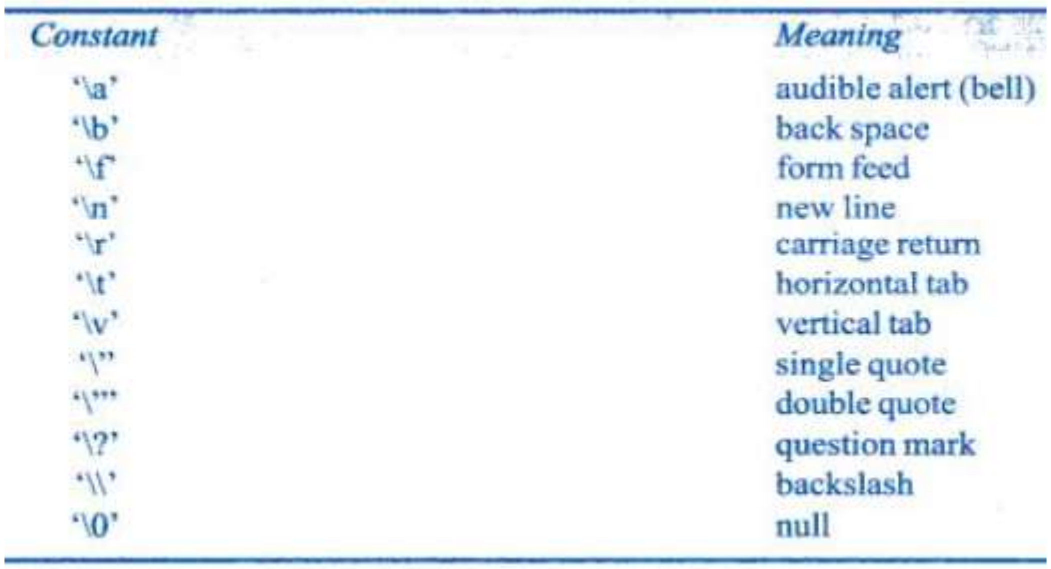
i) Single Character Constants: It is a single character that may be a single digit or special symbol or white space enclosed with in a pair of single quote marks.

Example: ‘a ’, ‘3 ’, ‘- ’

ii) String Constants: These are a sequence of characters enclosed with in a double quote marks. The string may be a combination of all kinds of symbol.

Example: “hello”, “abc123”, “1234”, “a”

Backslash Character Constants: C supports some backslash character constants that are used in output functions. Note that each one of them represents one character, although they consist of two characters. These character combinations are known as escape sequences.



Constant	*Meaning*
‘\a’	audible alert (bell)
‘\b’	back space
‘\f’	form feed
‘\n’	new line
‘\r’	carriage return
‘\t’	horizontal tab
‘\v’	vertical tab
‘\’’	single quote
‘\”’	double quote
‘\?’	question mark
‘\\’	backslash
‘\0’	null

3) IDENTIFIERS: Identifiers are user defined words used to name entities like variables, arrays, functions, structures etc. Rules for naming identifiers are:

1) Name should only consists of alphabets (upper & lower case), digits and underscore (_) sign.

2) First characters should be alphabet or underscore.

3) Name should not be a keyword.

4) Since C is a case sensitive language, the upper case and lower case are considered differently. For example code, Code, CODE etc. are different identifiers.

5) Identifiers are generally given some meaningful name such as value, net_salary, age, data etc. An identifier name may be long, some implementation recognizes only first eight characters, most recognize 31 characters. ANSI standard compiler recognize 31 characters. Some invalid identifiers are 5cb, int, res#, avg no etc.

VARIABLES:

Variable is a data name which is used to store some data value or symbolic names for storing program computations and results. The value of the variable can be changed during the execution. The rule for naming the variables is same as the naming identifier. Before used in the program it must be declared. Declaration of variables specify its name, data types and range of the value that variables can store depends upon its data types.

Syntax: data type variable_name;

Examples:

```
int a;
char c;
float f;
```

Variable Initialization:

When we assign any initial value to a variable during the declaration, is called initialization of variables. When variable is declared contain undefined value which is called garbage value. The variable is initialized with the assignment operator such as

data type variable_name=constant;

Example: int a=20;

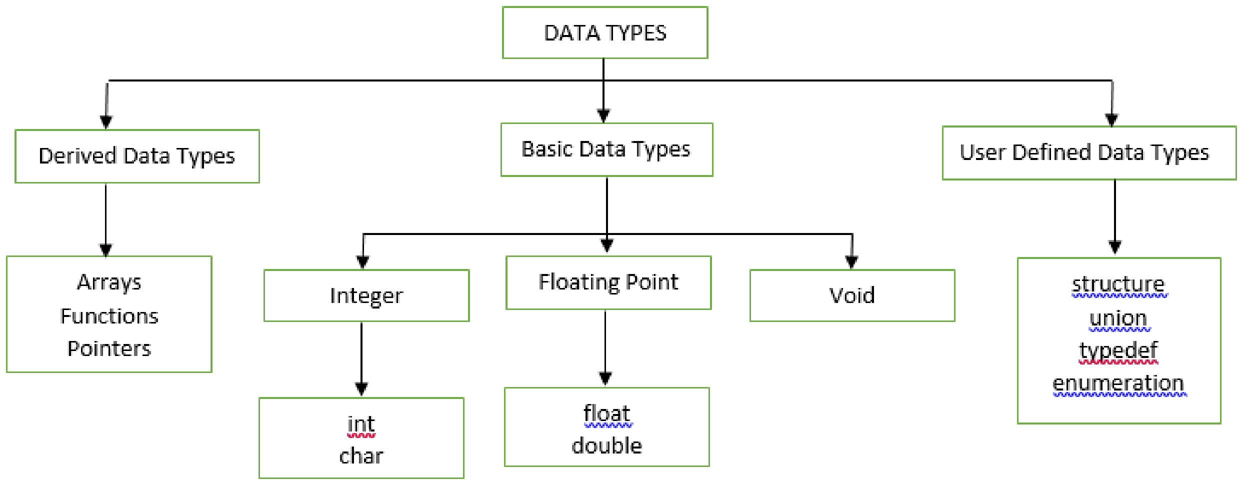
Or int a;

a=20;

DATA TYPES

Data types refer to an extensive system used for declaring variables or functions of different types before its use. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. The value of a variable can be changed any time.

C has the following types of data types:



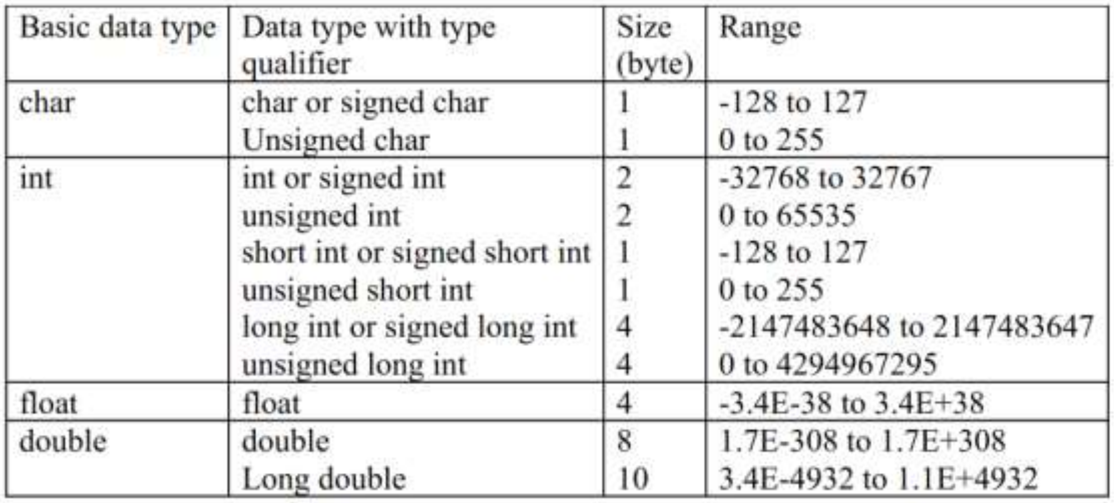
A variable declared to be of type int can be used to contain integral values only—that is, values that do not contain decimal places. A variable declared to be of type float can be used for storing floating- point numbers (values containing decimal places). The double type is the same as type float, only with roughly twice the precision. The char data type can be used to store a single character, such as the letter a, the digit 6, or a semicolon. Similarly a variable declared char can only store character type value.

There are two types of type qualifiers in C:

Size qualifier: short, long

Sign qualifier: signed, unsigned

When the qualifier unsigned is used the number is always positive, and when signed is used number may be positive or negative. If the sign qualifier is not mentioned, then by default signed qualifier is assumed. The range of values for signed data types is less than that of unsigned data type. Because in signed type, the left most bit is used to represent sign, while in unsigned type this bit is also used to represent the value. The size and range of the different data types on a 16 bit machine is given below:



Basic data type	Data type with type qualifier	Size (byte)	Range
char	char or signed char	1	-128 to 127
	Unsigned char	1	0 to 255
int	int or signed int	2	-32768 to 32767
	unsigned int	2	0 to 65535
	short int or signed short int	1	-128 to 127
	unsigned short int	1	0 to 255
	long int or signed long int	4	-2147483648 to 2147483647
	unsigned long int	4	0 to 4294967295
float	float	4	-3.4E-38 to 3.4E+38
double	double	8	1.7E-308 to 1.7E+308
	Long double	10	3.4E-4932 to 1.1E+4932

EXPRESSIONS:

An expression is a combination of variables, constants, operators and function call. It can be arithmetic, logical or relational. For example:

```
x + y          // arithmetic expression
a > b          // relational expression
a == b         // logical expression
func(a, b)     // function call
```

Expressions consisting entirely of constant values are called constant expressions.
So, the expression 121 + 17 – 110 is a constant expression because each of the terms of the expression is a constant value. But if j were declared to be an integer variable, the expression 180 + 2 – j would not represent a constant expression.

OPERATORS:

This is a symbol used to perform some operations on variables, operands or with the constants. Some operator requires 2 operands to perform operation or some requires single operand.

<u>Types of Operators</u>

1. Arithmetic operators (**+, -, *, /, %**)
2. Relational operators (**<, >, ==, >=, <=, !=**)
3. Logical operators (**&&, || , !**)
4. Bitwise operators (**&, |, ^, >>, <<, ~**)
5. Increment and Decrement operators (**++,- -**)
6. Assignment operator (**=, +=, -=, *=, /=, %=, &=, |=, ^=, >>=, <<=**)
7. Conditional operator (**? :**)
8. Special operators (,(comma), sizeof, etc.)

<u>ARITHMETIC OPERATORS</u>:

There are two types of arithmetic operators:
i) Unary operator ii) Binary operator

<u>Unary Operator</u>: The operator which require only one operand is called unary operator.

Unary Minus (-): unary minus is used for indicating or changing the algebraic sign of a value.

Example: int x= -10;

<u>Binary Operator</u>:

The binary operator takes two operands to perform numerical calculations. These operators are used in most of the computer languages.

Operator	**Operator Name**	**Example**	**Return Value**
+	Addition	2 + 4	6
-	Subtraction	4 – 2	2
*	Multiplication	5 * 4	20
/	Division	9 / 3	3
%	Modulo division	9 % 3	0

But modulus cannot be applied with floating point operands, as well as there is no exponent operator in C.

Unary (+) and Unary (-) is different from addition and subtraction.

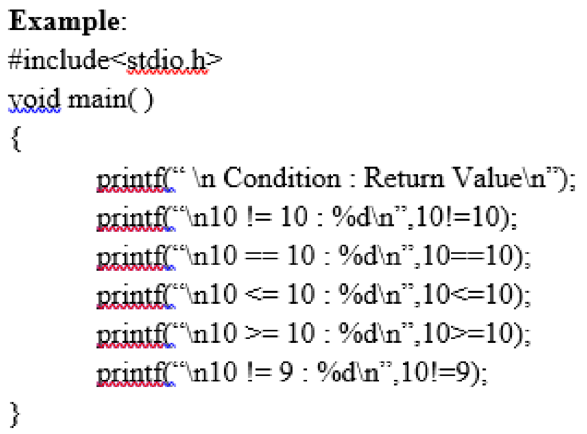
When both the operands are integer then it is called integer arithmetic and the result is always integer. When both the operands are floating point then it is called floating arithmetic and when operand is of integer and floating point then it is called mixed type or mixed mode arithmetic. And the result is in float type.

<u>RELATIONAL OPERATORS</u>:

These relational operators are used to test the relationship between two expressions. If the relation is true then it returns a value 1 otherwise returns 0 (zero).

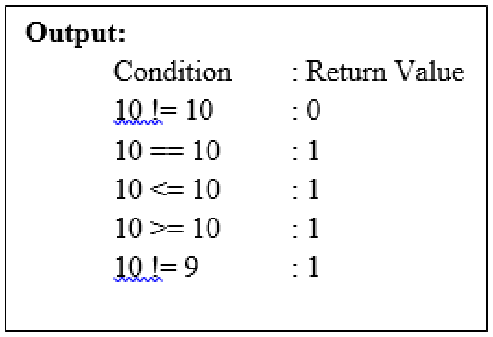
Operator	**Operator Name**	**Description**	**Example**	**Return Value**
==	Equal to	Checks if the values of two operands are equal or not. If the values are equal, then the condition becomes true.	A = 10, B = 20 (A == B) is not true.	0
!=	Not equal to	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	A = 10, B = 20 (A != B) is true.	1
>	Greater than	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	A = 10, B = 20 (A > B) is not true.	0

<	Less than	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	A = 10, B = 20 (A < B) is true.	1
>=	Greater than or equal to	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	A = 10, B = 20 (A >= B) is not true.	0
<=	Less than or equal to	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	A = 10, B = 20 (A <= B) is true.	1



Example:

```
#include<stdio.h>
void main( )
{
        printf(" \n Condition : Return Value\n");
        printf("\n10 != 10 : %d\n",10!=10);
        printf("\n10 == 10 : %d\n",10==10);
        printf("\n10 <= 10 : %d\n",10<=10);
        printf("\n10 >= 10 : %d\n",10>=10);
        printf("\n10 != 9 : %d\n",10!=9);
}
```



Output:

```
        Condition       : Return Value
        10 != 10        : 0
        10 == 10        : 1
        10 <= 10        : 1
        10 >= 10        : 1
        10 != 9         : 1
```

LOGICAL OPERATORS:

The logical operator test the relationship between two expressions and it also joins two expressions. The result can be either true (1) or false (0).

Operator	**Opeartor Name**	**Description**	**Example**	**Return Value**
&&	Logical AND	If both the operands are true or non-zero, then the condition becomes true.	A=0, B=1 (A && B) is false.	0
||	Logical OR	If any of the two operands is true or non-zero, then the condition becomes true.	A=0, B=1 (A || B) is true.	1
!	Logical NOT	It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	A=0 !(A) is true	1

Example:

```
void main( )
{
        printf(" \n Condition        : Return Values\n");
        printf("\n 5>3 && 5<10 : %d\n", 5>3 && 5<10);
        printf("\n8>5 || 8<2 : %d\n", 8>5 || 8<2);
        printf("\n!(8==8) : %d\n",!(8==8));
}
```

Output:

```
Condition       : Return Values
5>3 && 5<10 : 1
8>5 || 8<2      : 1
!(8==8)          : 0
```

BITWISE OPERATORS:

Bitwise operators permit programmer to access and manipulate data at bit level.

Various bitwise operators are:

one's complement	(~)
bitwise AND	(&)
bitwise OR	(|)
bitwise XOR	(^)
left shift	(<<)
right shift	(>>)

These operator can operate on integer and character values but not on float and double. In bitwise operator the function showbits( ) function is used to display the binary representation of any integer or character value.

TRUTH TABLE OF BITWISE OPERATORS:

p	**q**	**p & q**	**p | q**	**p ^ q**
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Example:

```
void main( )
{
        int x, y, z ;
        printf("Enter integer value of x: ");
        scanf("%d",&x);
        printf("Enter integer value of y: ");
        scanf("%d",&y);
        z = x &y;
        printf(" Bitwise AND of x and y: %d", z);
        z = x | y;
        printf(" Bitwise OR of x and y: %d", z);
        z = x>>2;
        printf("Right shift of x by 2:  %d", z);
        z= y<<2;
        printf("Left shift of y by 2: %d", z);
}
```

OUTPUT:

```
Enter integer value of x: 8
Enter integer value of y: 4
Bitwise AND of x and y: 0
Bitwise OR of x and y: 12
Right shift of x by 2: 2
Left shift of y by 2: 16
```

INCREMENT AND DECREMENT OPERATORS:

The Unary operators ++, -- acts upon single operand. Increment operator increases the value of operand by one. .Similarly decrement operator decreases the value of the operand by one. And these operators can only be used with the variable, but can't be used with expression or constant as ++6 or ++(x+y+z).

They are categorized into prefix and postfix operators. In the prefix the value of the operand is incremented by 1, then the new value is used, where as in postfix the value of the operand is used then it is incremented by 1. Postfix operator is written after the operand (such as m++, m--).

EXAMPLE:
z = ++y; is equivalent to y = y+1; z = y;
y = x++; is equivalent to y=x; x= x+1;

ASSIGNMENT OPERATOR:

A value can be stored in a variable with the use of assignment operator. The assignment operator (=) is used in assignment statement and assignment expression. Operand on the left hand side should be variable and the operand on the right hand side can be variable or constant or any expression.

When variable on the left hand side occurs on the right hand side then we can avoid by writing the compound statement using shorthand assignments operators.

Operator	Description	Example
=	Simple assignment operator. Assigns values fromright-hand operand to left-hand side variable.	C = A + B
+=	Add AND assignment operator. It adds the right operand to left operand and assign to left variable.	C += A (or) C=C+A
−=	Subtract AND assignment operator. It subtract the right operand to left operand and assign to left variable.	C −= A (or) C=C–A
*=	Multiply AND assignment operator. It multiply the right operand to left operand and assign to left variable.	C *= A (or) C=C*A
/=	Divide AND assignment operator. It divides the right operand to left operand and assign to left variable.	C /= A (or) C=C/A

%=	Modulus AND assignment operator. It apply modular division the right operand to left operand and assign to left variable.	C %= A (or) C=C%A
<<=	Left shift AND assignment operator.	C <<= 2 (or) C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 (or) C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 (or) C=C&2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 (or) C=C^2

CONDITIONAL OPERATOR (? :):

The conditional operator contains condition followed by two expressions or values. If the expression1 or condition is true, the expression2 is executed, otherwise expression3 is executed.

The conditional operator is also called as **ternary operator**.

Syntax: **expression1? (expression2) : (expression3) ;**

Example: (3 >2) ? printf(“true”) : printf(“false”) ; **Output**: true

<u>**Comma Operator (,)**</u>**:** The comma operator is used to separate two or more expressions or variables. It has the lowest priority among all operators.

Ex: c = (a=10,b=20,a + b);

sizeof Operator:

Size of operator is a Unary operator, which gives size of operand in terms of bytes that occupied in the memory. An operand may be variable, constant or data type qualifier.

Generally it is used make portable program (program that can be run on different machines). It determines the length of entities, arrays and structures when their sizes are not known to the programmer. It is also used to allocate size of memory dynamically during execution of the program.

Example:

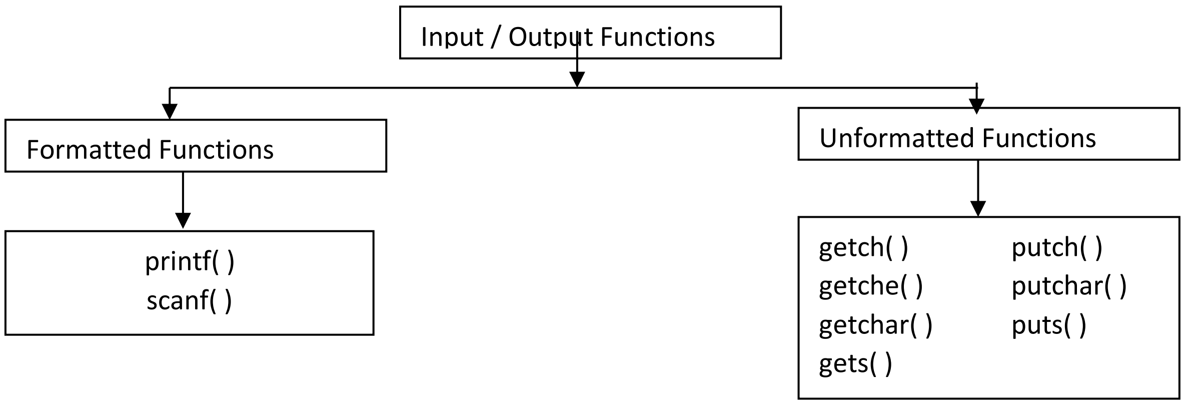
```
main( )
{
int sum;
float f;
printf( "%d %d" ,sizeof(f), sizeof(sum) );
printf("%d %d", sizeof(235 L), sizeof(‘A’));
}
```

INPUT AND OUTPUT IN C:

- Every program has to do three tasks, reading input from input devices, process it using operators and displaying the result on the screen.
- Sequence of data flow in all input and output operations. In input operations data flow from input devices such as keyboard, disk drive to main memory and in output operation data flow from main memory to output devices such as monitor, file etc.
- There are two ways to accept the data, one is the data values are assigned to the variable with assignment statements.

  Ex: int year = 2016;
  char letter ='a';
  double money = 1209.50;
- Second is accepting the data using functions in c.
- The input/output functions are classified into two categories:

1. Formatted Functions **2. Unformatted Functions**



1. Formatted Functions:

- With formatted functions, the input and output is formatted as per our requirement.
- All I/O functions are defined in **stdio.h** header file.
- Formatted functions can read and write all types of data values and it needs format string or format specifier.
- In formatted functions, **printf( )** and **scanf( )** functions are used for inputting and outputting the data.

The printf( ) Statement:

➢ **printf( )** statement is used to display the results as per the programmer requirement on the screen.

Syntax: printf ("control string", variable1, variable2 … variablen);

Here control string specifies the field formats such as %d, %s, %f etc., and variables are given by the programmer.

➢ It prints all types of data values and the variables according to the sequence mentioned in printf( ).

- ➢ It translates internal values to characters. It requires format conversion symbol or format string and variable names to print the data.

Example:

```
void main ( )
{
int x = 2;
float y = 2.2;
char z = ‘a’;
printf (“%d %f %c”, x, y, z);
}
```

Output:

```
2 2.200000 a
```

<u>The scanf( ) Statement</u>:

- ➢ The scanf( ) statement is used for runtime assignment of values to variables. It reads all types of data.
- ➢ The scanf( ) statement requires format string to identify the data to be read during the execution of the program.
- ➢ **Syntax**:

  scanf (“control string”, arg1, arg2 … argn);
- ➢ The scanf( ) statement stops functioning when some input entered does not match with format string.

<u>Example</u>:

```
void main ( )
{
int x;
printf (“\n enter value of x: “ );
scanf (“%d”,&x);
printf (“\nx = %d”, x);
}
```

Output:

```
Enter value of x: 20
x = 20
```

2. Unformatted Functions:

- The unformatted input and output functions works only on character data type.
- They do not require format specifier (or) format string for formatting of data because they work only on character data type.
- In case other data type values are passed to these functions they are treated as character data.
- Unformatted functions are categorized into 3 types:

  i) Character I/O   ii) String I/O   iii) File I/O

i) Character I/O:

getchar( ): This function reads one character at a time from standard input.

Syntax: VariableName = getchar( );

Example: char c;
c = getchar( );

putchar( ): This function prints one character at a time, read by standard input.

Syntax: putchar(VariableName);

Example: char c;
putchar(c);

```
void main( )
{
char c;
printf("Enter a character: ");
c = getchar( );
putchar(c);
}
```

Output:

```
Enter a character:  A
A
```

getch() & getche(): These functions read any alphanumeric character from the standard input device. The character entered is not displayed by the getch( ) function.

Syntax for getch( ): VariableName = getch( );

Syntax for getche( ): VariableName = getche( );

putch(): This function prints any alphanumeric character taken by the standard input device.

Syntax: putch(VariableName);

Example:

```
void main()
{
char ch;
printf("Enter any key to continue");
ch = getch();
printf("You pressed: ");
putch(ch);
}
```

Output:
```
Enter any key to continue
You pressed: 5
```

ii) String I/O:

gets( ): This function accepts any string through stdin (keyboard) until enter key is pressed.

Syntax: gets(StringName) ;

puts( ): This function prints the string or character array.

Syntax: puts(StringName);

Example:
```
void main()
{
char ch[30];
printf("Enter string: ");
gets(ch);
puts(ch);
}
```

Output:
```
Enter string: welcome
welcome
```

Flags: Flags are used for output justification, numeric signs, decimal points, and trailing zero.

Width: It sets the minimum field width for an output value. Width can be specified by decimal point or using an asterisk '*'.

Example:
```
void main ( )
{
int x = 22;
printf ("\n %3d", x);
printf ("\n %5d", x);
printf ("\n %*d", 7, x);
}
```
Output:
```
22
22
22
```

Precision: It specifies number of digits required after decimal point. The precision specifier always starts with dot in order to separate from width specifier.

Example:

```
void main ( )
{
double x=22.1234567;
printf (“\n %.2lf”, x);
printf (“\n %.3lf”, x);
printf (“\n %.4lf”, x);
}
```

Output:

```
22.12
22.123
22.1234
```

STRONG POINTS FOR UNDERSTANDABILITY:

The following points produce neat output:

- Give space between numbers.
- Provide suitable and problem – related variable names and headings.
- Provide user prompt so that the user can understand what to do.
- Provide gap between two lines so that the text should be readable.
- Alert the user about what to do and what not to do.
- Use formatted input and output functions for inputting the data and outputting the results.
- It is recommended to use escape sequence character such as \t, \n, \b etc.

CONSTANT AND VOLATILE VARIABLES:

Constant Variable: The value of variable is declared as constant means the values does not change or remains same during the program execution. If you try to change, it raises an error.

The keyword **const** is used for defining the variable constant.

Example:

```
void main( )
{
const int n=10;
n++;
}
```

Output: It shows error message.

Volatile Variable: The value of variable that can change at any time in the same program or by other external program.

```
volatile int d;
```

Example:
```
void main( )
{
volatile int x;
scanf("Enter value of x: %d",&x);
printf("Entered value: %d",x);
x=10;
printf(" Value of x: %d",x);
}
```

Output:
```
Enter value of x: 5
Entered value: 5
Value of x: 10
```

Format Modifiers: The different types of format modifiers are as follows:

%d - decimal integer

%ld - long integer

%f - float

%c - character

%s - string

%e - exponentiation

%lf or %g - double

%o - octal integer

%h - hexa decimal integer

%i - any integer

Precedence and Associativity

Precedence is used to determine the order in which different operators in a complex expression are evaluated. Associativity is used to determine the order in which operators with the same precedence are evaluated in the complex expression.

Precedence: Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others. The multiplication operator has a higher precedence than the addition operator.

For example, **x = 7 + 3 * 2**; here, **x** is assigned **13**, not **20** because operator ***** has a higher precedence than **+**, so it first gets multiplied with **3*2** and then adds into **7**.

Associativity: It can be left-to-right or right-to-left. **Left-to-right associativity** evaluates the expression by starting on the left and moving to the right and **right-to-left associativity** evaluates the expression by proceeding from the right to the left.

Left-to-right associativity: The following shows an example of left-to-right associativity. Here we have four operators of the same precedence **(* / % *).**

3 * 8 / 4 % 4 * 5

Associativity determines how the sub expressions are grouped together. All of these operators have the same precedence. Their associativity is from left-to-right. So they are grouped as follows.

((((**3 * 8)/ 4) % 4) * 5)**

The value of this expression is 10.

Right-to-left-Associativity: Several operators have right-to-left associativity for example, when more than one assignment operator occurs in an assignment expression, the assignment operator must be interpreted from right to left.

a + = b * =c - = 5

is evaluated as

(a + = (b * =(c - = 5)))

Which is expanded to?

(a = a+ (b = b * (c = c - 5)))

If **a** has an initial value of **3**, **b** has an initial value of **5**, and **c** has an initial value of **8**, this expressions becomes

(a = 3+ (b = 5 * (c = 8 - 5)))

Which results in **c** being assigned a value of **3**, **b** being assigned a value of **15**, and **a** being assigned a value of **18**.The value of the complete expression is also **18**.

Type Conversion:

- ✓ It is a process of converting two different types of values into a common type.
- ✓ It is needed to manipulate expressions.
- ✓ These are 2 types.
    - ✓ Implicit type conversion
    - ✓ Explicit type conversion

Implicit Type Conversion: When the types of the two operands in a binary expression are different, C automatically converts one type to another. This is known as implicit type conversion.

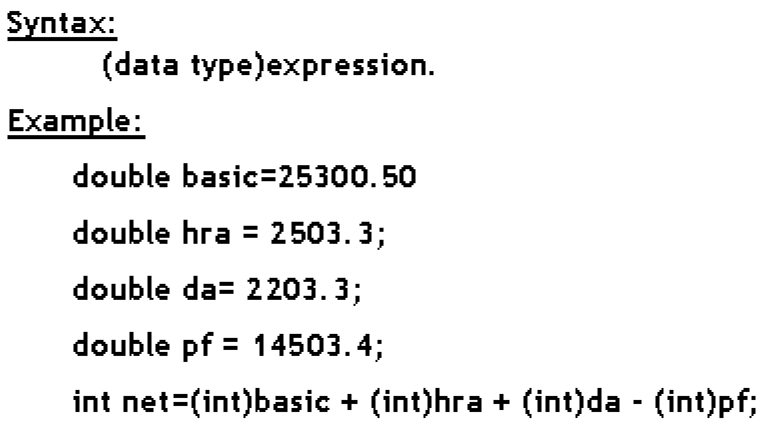
Ex-1:

```
char ch='a';
ch=ch-32;
```

Ex-2:

```
int a=10;
float b=2.5;
float sum=a+b;
```

Explicit Type Conversion: It is a type of conversion, which is explicitly defined within a program. Conversion will be done with force. It is also called type casting.



<u>Syntax:</u>

```
    (data type)expression.
```

<u>Example:</u>

```
    double basic=25300.50
    double hra = 2503.3;
    double da= 2203.3;
    double pf = 14503.4;
    int net=(int)basic + (int)hra + (int)da - (int)pf;
```

CONDITIONAL CONTROL STRUCTURES (DECISION MAKING AND BRANCHING):

'C' language supports decision making capabilities like the following statements known as control or decision making statements. They are listed as follows:

1. If statement
2. Switch statement
3. Conditional operator statement
4. Go to statement

if Statement: The **if** statement is powerful decision making statement and is used to control the flow of execution of statements.

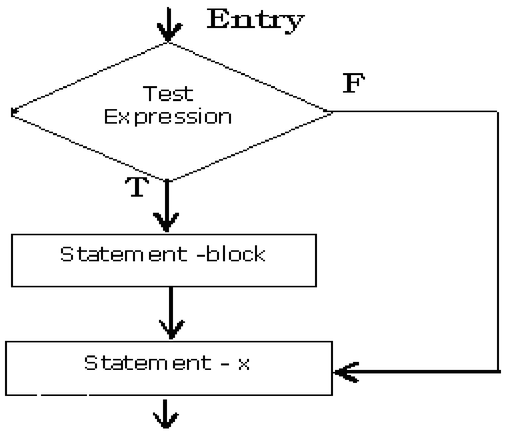
a) Simple if statement
b) If else statement
c) Nested If-else statement
d) Else –If ladder statement

Simple if Statement (null else statement): The general form of simple if statement is:

```
if (test expression)
{
        Statement block;
}
Statement - x;
```

The statement block may be a single statement or a group of statements. If the test expression is true then the statement block will be executed. Otherwise the statement block will be skipped and the execution will jump to the statement – x. If the condition is true both the statement - block and statement-x in sequence are executed.

Flow Chart:



Example:

```
if(category = sports)
{
marks = marks + bonus marks;
}
printf("%d", marks);
```

If the student belongs to the sports category then additional bonus marks are added to his marks before they are printed. For others bonus marks are not added.

Program to explain about simple if statement

```
#include<stdio.h>
int main()
{
        int x;
        printf("enter the value of x");
        scanf("%d", &x);
        if (x == 1)
                printf(" x value is one");
        printf("%d", x);
        return 0;
}
```

if – else Statement:

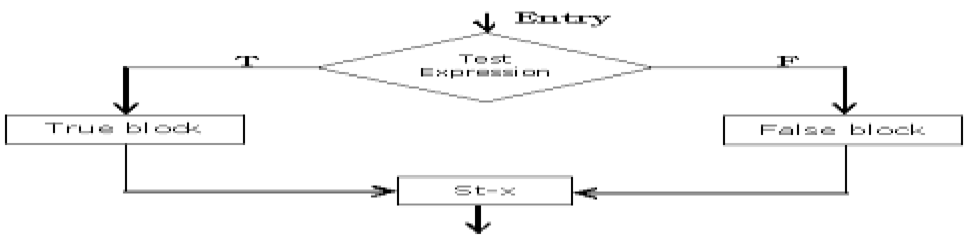
The if - else statement is an extension of the simple if statement. It is a composite statement used to make decision between two alternatives. The general form is

```
if (test expression)
{
        true-block statements;
}
else
{
        false-block statements;
}
statement – x;
```

Points to note

- ✓ The expression must be enclosed in parenthesis.
- ✓ No semicolon is required for if else statement
- ✓ The expression can have side effect
- ✓ Both true and false block may contain null statement
- ✓ Both blocks may contain either one statement or group of statements if group then they should be presented between a pair of braces called compound statement.
- ✓ We can use complement of the original statement in some cases

Flow Chart:



```
        }
```

The expression is a C expression. After its evaluation its value is either true or false. If the test expression is true then true-block statements are executed, otherwise the false–block statements are executed. In both cases either true-block or false-block will be executed but not both.

Example:

```
        If (code == 1)
                boy = boy + 1;
        else
                girl = girl + 1;
        st-x;
```

Here if the code is equal to ‘1’ the statement *boy=boy+1;* is executed and the control is transferred to the *statement st-x*, after skipping the else part. If code is not equal to ‘1’ the statement *boy =boy+1;* is skipped and the statement in the else part *girl =girl+1;* is executed before the control reaches the statement st-x.

Program to explain about if else statement

```
#include<stdio.h>
int main()
{
        int num;
        printf(“enter the value “);
        scanf(‘%d”, &num);
        if(num%2==0)
        {
                printf(“the number is even”);
        }
        else
        {
                printf(“the number is odd”);
        }
        return 0;
}
```

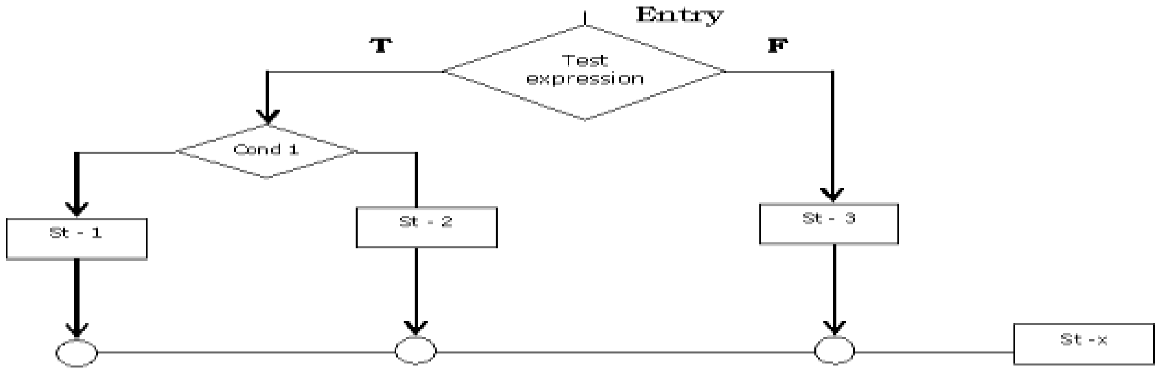
Nested if–else Statement:

When a series of decisions are involved we may have to use more than one if-else statement in nested form. An if-else is included in another if-else is known as nested if else statement. The syntax is as follows.

```
if(test expression1)
{
        if(test expression2)
        {
                st –1;
        }
        else
        {
                st – 2;
        }
}
else
{
        st – 3;
}
st – x;
```

If the test expression1 is true then the inner if is evaluated, if the inner if is true then st-1 is executed otherwise st-2 is executed. If the test expression1 is false then st-3 is executed. Later finally st-x is executed.

Flow Chart:



If the condition is false then st-3 will be executed otherwise it continues to perform the nested if – else structure (inner part). If the condition 1 is true the st-1 will be executed otherwise the st-2 will be evaluated and then the control is transferred to the st-x.

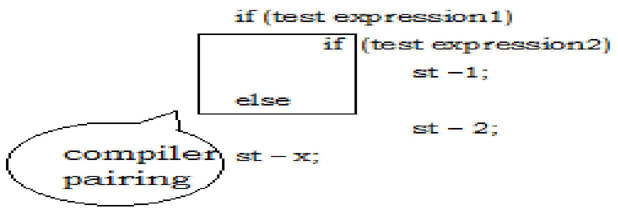
Example:

```
if(sex ==female)
{
        if(balance>5000)
                bonus=0.5*balance;
        else
                bonus=0.2*balance;
}
else
{
        bonus=0.6*balance;
}
balance=balance+bonus;
printf("%f", balance);
```

Program to explain about nested if else statement

```
#include<stdio.h>
int main()
{
        int a, b, c;
        printf("enter the values of a, b, c");
        scanf("%d%d%d", &a, &b, &c);
        if(a>b)
        {
                if(a>c)
                {
                        printf("%d a is big", a);
                }
                else
                {
                        printf("%d c is big", c);
                }
        }
        else
        {
                if(c>b)
                {
                        printf("%d c is big", c);
                }
                else
                {
                        printf("%d b is big", b);
                }
        }
        return 0;
}
```

Dangling else Problem: In nested if-else we have a problem known as dangling else problem. This problem is created when there is no matching **else** for every **if**. In C, we have the simple solution. **"always pair an else to most recent unpaired if in the current block"**.



But however it may lead to problems hence the solution is to simplify if statements by putting braces.

```
if(test expression1)
{
        if(test expression2)
                st –1;
}
else
                st – 2;
st – x;
```

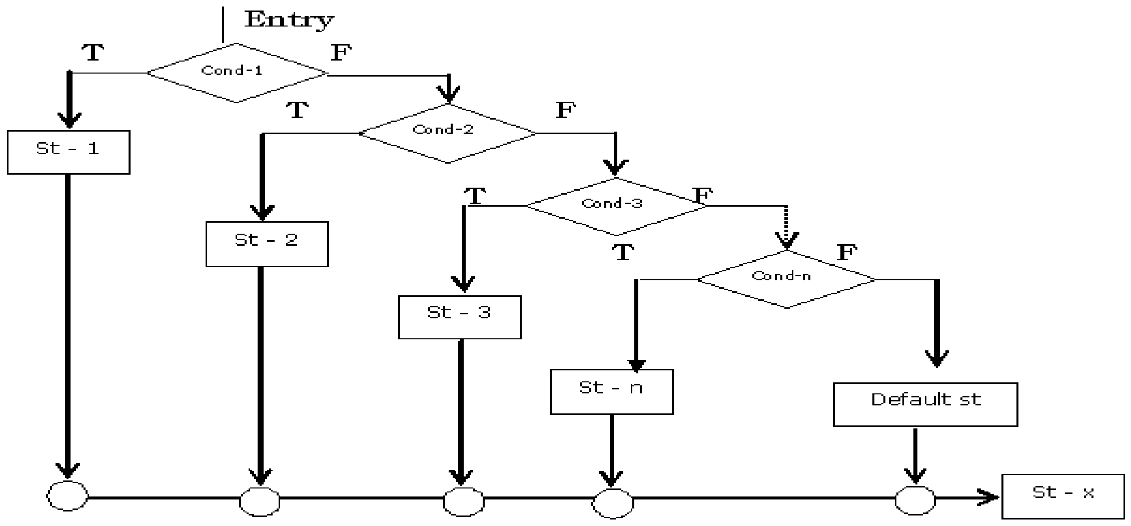
MULTI WAY SELECTION:

else - if Ladder: A multi path decision is chain of **if's** in which the statement associated with each else is an **if**. It takes the following general form.

```
if (condition1)
        St –1;
else  if (condition2)
                St –2;
        else  if (condition 3)
                St –3;
                  :
                  :
        else
                default – st;
St –x;
```

The conditions are evaluated from the top of the ladder to downwards. As soon as a true condition is found the statement associated with it is executed and the control is transferred to the st-X (i.e., skipping the rest of the ladder). When all the n-conditions become false then the final else containing the default – st will be executed.

Flow Chart:



Example:

```
        if (code = = 1)
                Color = “red”;
        else if ( code = = 2)
                        Color = “green”
                else if (code = = 3)
                                Color = “white”;
                        else
                                Color = “yellow”
    If code number is other than 1, 2 and 3 then color is yellow.
```

Program to explain about if else ladder

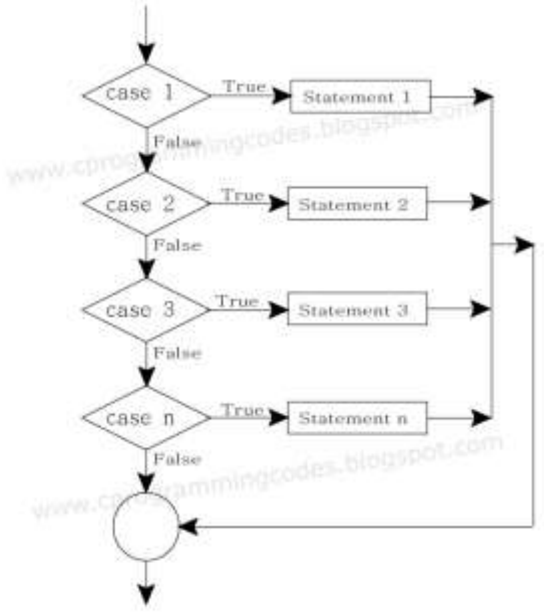
```
#include<stdio.h>
int main()
{
    int m1,m2,m3,m4,m5, total; float per;
    printf(“Enter the marks for five subjects: ”);
    scanf(“%d%d%d%d%d”, &m1, &m2, &m3, &m4, &m5);
    total = m1 + m2 + m3 + m4 + m5;
    per = total / 500;
    printf(“%f”, per);
    if(per >= 90)
        printf(“A”);
    else if(per >= 80)
        printf(“B”);
        else if(per >= 70)
            printf(“C”);
            else if(per >= 60)
                printf(“D”);
                else
                    printf(“F”);
    return 0;
}
```

Switch Statement: Instead of else – if ladder, ‘C’ has a built-in multi-way decision statement known as a switch. The general form of the switch statement is as follows.

```
switch (expression)
{
    case value1:    block1;
                        break;
    case value 2  : block 2;
                        break;
                  :
                  :
```

```
        default :       default block;
                             break;
}
st – x;
```

Flow Chart



The expression is an integer expression and character value1, value-2------ are constants or constant expressions known as case labels. Each of the values should be a unit within a switch statement and may contain zero or more statements.

When the switch is executed the value of the expression is successively compared against the values value-1, value-2------- If a case is found whose value matches with the expression then the block of statements that follows the case are executed.

The **break** statement at the end of each block signal the end of a particular case and causes an exit from the switch statement transferring the control to the st-x following the switch. The default is an optional case. It will be executed if the value of the expression doesn’t match with any of the case values and then control goes to the St-x.

Example:

```
        switch (number)
        {
                case 1 :        printf(“Monday”);
                                break;
                case 2 :        printf(“Tuesday”);
                                break;
                case 3 :        printf(“Wednesday”);
                                break;
                case 4 :        printf(“Thursday”);
                                break;
```

```
                    case 5 :        printf("Friday");
                                    break;
                    default :       printf("Saturday");
                                    break;
        }
```

Program to explain about the arithmetic operators using switch statement

```
main()
{
        int a , b , choice;
        float c;
        printf("enter the values of a & b");
        scanf("%d%d", &a, &b);
        printf("enter the choice");
        scanf("%d", &choice);
        switch(choice)
        {
                case '1' :      c=a+b;
                                printf("%f", c);
                                break;
                case '2' :      c=a-b;
                                printf("%f", c);
                                break;
                case '3' :      c=a/b;
                                printf("%f", c);
                                break;
                case '4' :      c=a*b;
                                printf("%f", c);
                                break;
                case '5' :      c=a%b;
                                printf("%f", c);
                                break;
                default case :  printf("Invalid option");
                                break;
        }
}
```

goto Statement:

This is an unconditional control jump statement. This statement passes control anywhere in the program i.e., control is transferred to another part of the program without testing any condition. goto requires a label to specify where the control to be transferred. Label is placed just before the statement where the control to be transferred. Label must be specified with a colon.

Syntax: goto label;

Example: Write a program to detect the entered number is even or odd using goto.

```
void main( )
{
    int s, d;
    printf(“Enter a number: ”);
    scanf(“%d”,&s);
    if(s %2 == 0)
        goto even;
    else
        goto odd;
    even:
    {
        printf(“\n Even Number”);
        return;
    }
    odd:
        printf(“Odd Number”);
}
```

Output:

```
Enter a number: 25
Odd Number
```

Character Test Functions:

In the following table the required functions and their tests are given.

Function	Test
isalnum(c)	Is c an alphanumeric character?
isalpha(c)	Is c an alphabetic character?
isdigit(c)	Is c a digit?
islower(c)	Is c a lower case letter?
isprint(c)	Is c a printable character?
ispunct(c)	Is c a punctuation mark?
isspace(c)	Is c a white space character?
isupper(c)	Is c an upper case later?

Character Converting Functions: The converting functions are listed below:

Function	Conversion
tolower(c)	Converts an upper case alphabet to lower case
toupper(c)	Converts a lower case alphabet to upper case

Note: When the above functions are used in the program, ctype.h header file must be included.