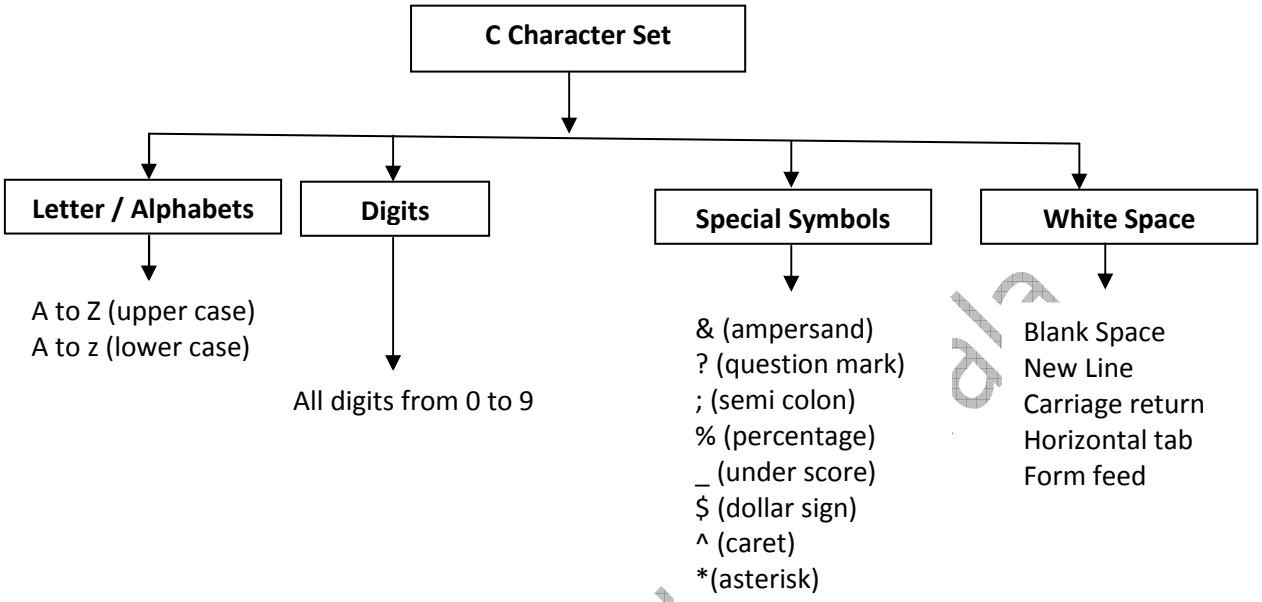
C Language

❖ **List of Character Set**

The main use of character set is to make words, numbers and arithmetic expression in program depending upon our requirement. In C language, Character set is classified in following categories.



❖ **C - Tokens, Keywords and Identifiers**

- **Token**

Token is one type of special symbol or word that is a combination of different character set.

Token is classified in six different groups.

No.	Token Name	Examples
1	Identifiers	Price, Qty, Amt, Sum, Total
2	Keywords (Reserve words)	Int, float, break, return, for
3	Constants	28, 10.50, 0.004
4	Operators	+,-,*,/,<,>,=,%
5	Strings	“Hello”, “Students”, “good morning”
6	Special Symbol	, ; & “ ? ! ^ ~ # @

- **Keywords**

They are the predefined words in a programming language and every keyword has its own specific meaning and purpose. It is also called predefined words or reserve words. They are always written in lower case letter. There are ***32*** keywords in C Language.

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

- **Identifiers**

  It is nothing but a user defined names of a variable, arrays, and functions used in a program.

  Identifiers are made with the help of alphabets (A to Z or a to z), digits(0 to 9) and special symbols(_).

  Rules for naming an Identifier in program

  (1) C is a case sensitive language, so if the same variable name is written with the different case then they are considered as a different identifiers. For example, variable name amount and AMOUNT are totally different.
  (2) We can declare identifier using upper or lower case in a program but you cannot declare any keyword as an identifier in a program because all the keywords are reserved words in C and every keyword has special meaning.
  (3) Every identifier name must be start with character and only _ (underscore symbol) is used in between name.
  (4) Generally, specify identifier name with maximum upto 31 characters.
  (5) Space is not allowed in identifier name.

  Examples of Identifier

  (1) Valid Name: Emp_No, Sub1, Sub2, X12345, Xif, Xprint
  (2) Invalid Name: Emp No, 123A, #Rs, void, break

❖ **Concept of Variables & Constants**

- **Variables**

  Variables is a container for a data or value that may change during program execution. It is an identifier for a memory location where the value can be stored and it is changed during program execution and user can retrieve value for variable also.

  Variable specifies an area of memory that contains value of given type that can be changed during program execution.

  Naming rules for variables:

  1. Variable name can be in form of upper case or lower case that contains letters, digits and only underscore symbol.
  2. No other punctuation mark is allowed in variable name.
  3. Variable name always start with alphabet.
  4. Only first 32 characters of a variable name are considered.
  5. White space is not allowed in variable name.
  6. Keyword cannot be used as variable name.
  7. Variable name is case sensitive so total and Total are the different variable name.
  8. Variable name should be unique in an entire program.

  The following table illustrate the different and invalid variable name:

Variable Name	Valid / In valid	Reason
Qty	Valid	All characters
Emp No	In valid	Space is not allowed
Roll-no	In valid	No special symbol allowed
Auto	In valid	Keyword
Pin123	Valid	First characters
1X	In valid	Numbers not allowed at first position
Item_Qty	Valid	Underscore Symbol

Declaration of Variable:

1. To mention data type before variable name according to which type of data is to be stored in it. For example, int, float, char
2. To assign appropriate value depending upon data type.
   General form of variable declaration (Syntax):
   <Data type> <variable name> [=Assign value]
   For example: int rollno=123; float price=35.50; char gender='M'

Garbage Value: whenever any variable declared at the beginning of the function block and does not assign any value at that time, C compiler gives an unknown value to that variable during execution process, so this unknown value is called garbage value.
Note: [All variables must be declared before they can be used in a program otherwise it will generate compile time error.]

Reason to declare a variable before use:

1. It makes thing easier on the compiler during program execution it knows right away what kind of storage to allocate with a variable name and what value is stored in that location and also know the amount of memory that will be required to a variable.
2. It provides that variable list to the compiler with memory location so that it can be used for cross verification of other variable that whether that variable is declared more than one time or not.

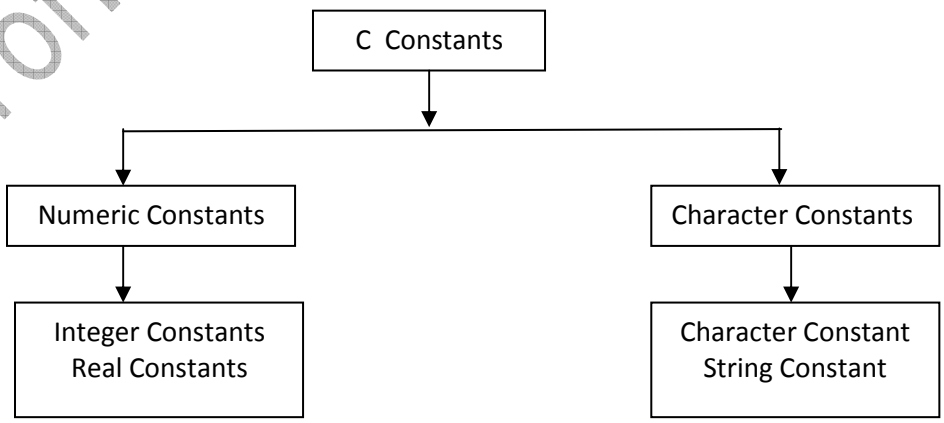
- **Constant**

Constant refers to a value that cannot be changed during program execution, C allows you to declare constants same as variable declaration and its value cannot be changed in a program during execution, and the **const** keyword is used to declare a constant like as follows:

```
int const x=10;
const int x=20;
```

Note: const keyword can be specified before or after data type of a variable.

There are several types of constants in C, they are classified into following groups as follows:-



- **Numeric Constant**

i) Integer Constant

Integer constant is a combination of digit without decimal point or fractional part, an integer constant does not have decimal point, it may be positive, negative or zero number. Eg: there are three types of integer constant

(a) Decimal Integer constant
(b) Octal Integer constant
(c) Hexadecimal Integer constant

No.	Constant Types	Digits allowed	Examples
1	Decimal Integer	0 to 9	125, 0, -550, +50, 35000
2	Octal Integer	0 to 7	234, 030, 561
3	Hexadecimal Integer	0 to 9 and A for 10, B for 11, C for 12 D for 13 E for 14 F for 15	0x5D, 0X25A (Here, 0x or 0X is considered as hexadecimal number)

ii) Real Constant

They are often known as floating point constants. Integer constant are not fit to represent many quantities. Many quantities or parameters are defined not only in integers but also in real numbers. For example, height, length, price and distance are major in real numbers.

Example:

2.5, 5.521, 3.14

The real number may represent in exponential notation, which contains fractional part and exponential part. For example, 512.72 may be written as 5.1272e2 in exponential notation. Here e2 means multiply numbers by 102.

The general format of the real numbers contains mantissa and exponent. The mantissa is either the real number represent in decimal or integer and the exponent is an integer number and may be positive or negative. The letter 'e' separating the mantissa and exponent can be written in lowercase or uppercase

❖ **Pre-processor Directories:**

The pre-processor offers several features called pre-processor directives. Each of these pre-processor directives begins with a # symbol. The directives can be placed anywhere in a program but are most often placed at the beginning of a program, before the first function definition. We would learn the following pre-processor directives here:

1. Macro expansion
2. File inclusion
3. Conditional Compilation
4. Miscellaneous directives

- **Macro expansion**

```
# include<stdio.h>
# define UPPER 25
void main()
{
    int i;
    for(i=1;i<=UPPER;i++)
```

```
        printf("\n%d",i);
    }
```

In this program, instead of writing 25 in the for loop we are writing it in the form of UPPER, which has already been defined before **main()** through the statement,

define UPPER 25

This statement is called 'macro definition' . During pre-processing, the pre-processor replaces every occurrence of UPPER in the program with 25.

#define directive could be used even to replace even an entire C statement, as shown below:

```
#include<stdio.h>
# define FOUND printf("The Trojan horse Virus");
void main()
{
        char signature;
        if(signature=='Y')
            FOUND
        else
            printf("Safe....as yet!")
        endif
}
```

A # define directive is many a time used to define operators as shown below:

```
#include <stdio.h>
#define AND &&
#define OR ||
Void main()
{
        Int f=1, x=4, y=90;
        If((f<5) AND (x<=20 OR y<=45))
   Printf("\nYour PC will always work fine....");
Else
  Printf("\n In front of the maintenance man");
}
```

The macros that we have used so far are called simple macros. Macros can have arguments, just as functions can. Here is an example that illustrates this fact:

```
#include <stdio.h>
#define SQUARE(n) n*n

Void main()
{
    Int j;
    J=(64/SQUARE(4));
     Printf("j=%d",j);
}
```

The output of the above program would be j=4.

- **File inclusion**
  This file inclusion directive causes one file to be included in another. The pre-processor command for file inclusion looks like this:

#include “filename”
And it simply causes the entire contents of **filename** to be inserted into the source code at that point in the program. Of course, this presumes that the file being included exists. It can be used in two cases:

(a) If we have a very large program, the code is best divided into several different files, each containing a set of related functions. It is a good programming practice to keep different sections of a large program separate. These files are #included at the beginning of main program file.
(b) There are some functions and some macro definitions that we need almost in all programs that we write. These commonly needed functions and macro definitions can be stored in a file, and that file can be included in every program we write, which would add all the statement in this file to our program as if we have typed them in.

It is common for the files that are to be included to have .h extension. This extension stands for ‘header file’, possibly because it contains statements which when included go to head of your program. The prototypes of all the library functions are grouped into different categories and then stored in different header files. For example, prototypes of all mathematics related functions are stored in the header file ‘math.h’, prototypes of console input/output functions are stored in the header file ‘conio.h’ and so on.

Actually there exist two ways to write #include statement. These are:

#include “filename”
#include <filename>
The meaning of each of these forms is as below:
#include “goto.h”
This command would look for the file **goto.h** in the current directory as well as the specified list of directories as mentioned in the include search path that might have been set up.

#include<goto.h>
This command would look for the file **goto.h** in the specified list of directories only.

- **Conditional Compilation**

We can, if we want, have the compiler skip over part of a source code by inserting the pre-processing commands #ifdef and #endif, which have the general form:

```
#ifdef macroname
        Statement1;
        Statement2;
        Statement3;
#endif
```

If macroname has been #defined, the block of code will be pre-processed as usual; otherwise not.

#ifdef is useful when we like to compile only a part of our program:
(a) To “comment out” obsolete lines of code. It often happens that a program is changed at the last minute to satisfy a client. We can put that code in comments, but we have limitation of nested comment s in ‘c language’. Therefore , the solution is to use conditional compilation as shown below:

```
Void main()
{
    #ifdef OKAY
            Statement1;
            Statement2;
            Statement3;
            Statement4;
    #endif
            Statement5;
            Statement6;
            Statement7;
}
```

Here, statements 1, 2, 3 and 4 would get compiled only if the macro OKAY has been defined, and we have purposefully omitted the definition of macro OKAY. At a later date, if we want that these statements should also get compiled, all that we are required to do is to delete the #ifdef and #endif statements.

(b) A more sophisticated use of #ifdef has to do with making the programs portable, i.e. to make them work on two totally different computers. For example:

```
Void main()
{
   #ifdef INTEL
      Code suitable for an Intel PC
  #else
      Code suitable for Motorola PC
#endif
      Code common to both the computers
}
```

When you compile this program, it would compile only the code suitable for Motorola PC and the common code. This is because the macro INTEL has not been defined. These statements are working same as if-else control instruction of C.

#if and #elif Directives

The #if directive can be used to test whether an expression evaluates to a non zero value or not. If the result of the expression is non zero, then subsequent lines upto a #else, #elif or #endif are compiled, otherwise they are skipped.

A simple example of #if directive is shown below:

```
Void main()
{
    #if TEST<=5
            Statement 1;
            Statement 2;
            Statement3;
    #else
            Statement4;
            Statement5;
            Statement6;
    #endif
}
```

If the expression, TEST<=5 evaluates to true, then statements1, 2 and 3 are compiled, otherwise statements 4, 5 and 6 are compiled.

The #elif directive can be used for more than one condition testing, a simple example of #elif directive is shown below:

```
#if Test<=50
    Statement1;
    Statement2;
    Statement3;
#elif TEST>=51 && TEST<=60
    Statement4;
    Statement5;
    Statement6;
#else
    Statement 7;
#endif
```

- **Miscellaneous directives**

There are two more pre-processor directives available, though they are not very commonly used. They are:

(a) #undef

On some occasions, it may be desirable to cause a defined name to become 'undefined'. This can be accomplished by means of the #undef directive. In order to undefine a macro that has been earlier #defined, the directive.

#undef macro template
Can be used. Thus the statement,
#undef PENTIUM
Would cause the definition of PENTIUM to be removed from the system. All subsequent #ifdef PENTIUM statements would evaluate to false. But not required to undefined a macro.

(b) #pragma

This directive is another special purpose directive that you can use to turn on or off certain features. Pragmas vary from one compiler to another. Turbo C/C++ compiler has got a pragma that allows you to suppress warnings generated by the compiler.
(a) # pragma start-up and #pragma exit: these directives allow us to specify functions that are called upon program start-up or program exit. Their usage is a s follows:

```
Void fun2();
# pragma start-up fun1
#pragma exit fun2

Void main()
{
   Printf("\nInside main");
}
Void fun1()
{
        Printf("\nInside fun1");
}
Void fun2()
```

```
{
        Printf("\nInside fun2");
}
```

And here is the output of the program:
Inside fun1
Inside Main
Inside fun2

Note that the functions fun1() and fun2() should neither receive nor return any value. If we want two functions to get executed at start-up then their pragmas should be defined in the reverse order in which you want to get them called.

(b) #pragma warn: On compilation the compiler reports Errors and Warnings in the program, if any; errors provide the programmer with no options, apart from correcting them. Warnings, on the other hand, offer the programmer a hint or suggestion that something may be wrong with a particular piece of code. Two most common situations when warnings are displayed are as under:

- If you have written code that the compiler's designers consider bad C programming practice. For example, if a function does not return a value then it should be declared as void.
- If you have written code that might cause run-time errors, such as assigning a value to an uninitialized pointer.

For example:

```
# pragma warn –rvl /* return value */
# pragma warn –par/* parameter not used */
# pragma warn –rch/* unreachable code */

Int f1()
{
 Int a=5;
}
Void f2(int x)
{
 Printf("\nInside f2");
}
Int f3()
{
 Int x=6;
 Return x;
 X++;
}
Void main()
{
 F1();
 F2(7);
 F3();
}
```

Here in this program, we can notice three problems immediately:
(a) Though promised, f1() doesn't return a value.
(b) The parameter x that is passed to f2() is not being used anywhere in f2().

(c) The control can never reach x++ in f3().

However, this does not happen since we have suppressed the warnings using the #pragma directives. If we replace the '-' sign with a '+' sign the these warnings would be flashed on compilation.