The C character set:

The character denotes any alphabets, digit or special symbol used to represent information. The characters allowed in C are:

Alphabets $\Rightarrow$ A to Z both upper case and lower case.

Digits $\Rightarrow$ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Some special symbols $\Rightarrow$**∼ ‘ ! @ # % ^ & * ( ) - _+ = / \ { } [ ] : ; “ ‘ < > . , ?**

Constant and variables:

The alphabets, numbers and special symbols when properly combined form constants and variables. A constant is a quantity that does not change during execution. These constants can be stored at a location in the memory of the computer. A variable can be considered, as a name given to the location in memory where this constant is stored. The content of variables can change.

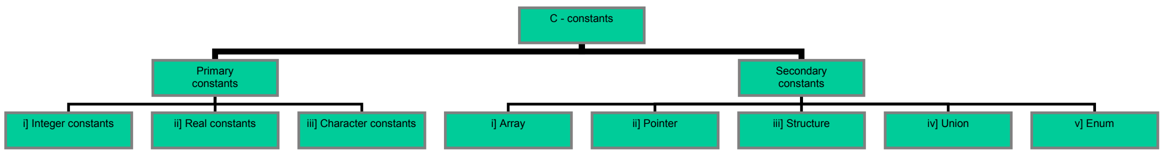
e.g., 3X+2Y=20; in this equ^{n} 3, 2, 20 cannot change, they are called constants. Whereas, the quantities X and Y can vary or change, hence they are called variables.

Types of **C** constants:

C constants can be divided into two major categories: $\Rightarrow$

(a) Primary constants: $\rightarrow$ i] Integer constant
ii] Real constant
iii] Character constant.

(b) Secondary constants: $\rightarrow$
i] Array
ii] Pointer
iii] Structure
iv] Union
v] Enum etc.



Rules for constructing different types of constants:

[i] *Integer constant:*

1. An integer constant must have at least one digit.
2. It must not have a decimal point.
3. It could be either positive or negative.
4. No commas or blanks are allowed.

5. The range of an integer constant depend upon the word size of the computer. For 16 bit computer the range would be -32768 to +32768.

e.g., 426, +782, -400, -980 etc.

[ii] *Real constants:*

Real constants are often called floating point constants. The real may be written in fractional or exponential form.

1. A real constant must have at least one digit.
2. It must have a decimal point.
3. It may be either positive or negative.
4. No commas or blanks are allowed.

e.g., +324.56, 432.87, -98.87 etc.

The exponential form is used if the value of the constants be either too small or too large. E.g., +3.2e $^{-5}$, 4.2e^{6}, -5.2e^{5} etc.

[iii] *Character constants:*

1. A character constant is a single alphabet, a single digit or a single special symbol with in single inverted commas.
2. The maximum length of a character constant can be of one character.

e.g., ‘A’, ‘d’, ‘1’, ‘=’ etc.

N.B. Secondary constants will be discussed later.

<u>C – variables:</u>

In C the quantity that may vary during program execution is called variable. Variables are the names given to locations in the memory of the computer where different constants are stored. There are three types of variables according to the constant stored in the variables. Viz. integer variable, real variable and character variable. For example integer variable can store only integer constants real variables store real constants and character variables can store character constants.

Rules for constructing variable name:

1. A variable name is any combination of 1 to 8 alphabets, digits or underscores.
2. The first character in the variable name must be an alphabet.
3. No commas or blank are allowed with in the variable name.
4. No special symbol other than underscore is used.

e.g., si_int, y_o_j, ab etc.

Rules for constructing all the three types of variable name are same. However it is compulsory to declare the type of any variable name used in the program.

Integer variables are declared using “int”

```
Syntax:     int (blank) si;
            Int (blank) a_b,sah;
```

Here integer constants are stored with in the variables si, a_b, sah etc.

Floating variables are declared using “float”.

```
Syntax:     float (blank) a;
            Float (blank) sa_h,sah,x_y;
```

Here the variables a, sa_h, sah, x_y are used to store real constants.
Character constants are declared using "chr".
Syntax: char (blank) asd, phy;
Char (blank) ah;
Here the variables asd, phy, ah are used to store character constants.

Double precision floating point number:

When a variable name is declared as float the number of mantissa digits stored and the exponent size depends on the computer on which the **C** program runs. For machines with 32 bit word size (St Vax and IBM PCs) it is seven digit mantissa and an exponent range of ±38. In some calculation the mantissa length may not be sufficient. For this C provides a type name called "*double*". The use of double in declaring floating point variable provides 16 mantissa digits storage for the variable name.

Syntax: double (blank) < variable name >;

Long or short integer:

C also provides the facility to use short and long integers. Integer size depends on the word size of a machine. In a 32 bit machine it has integer range $+(2^{31}-1)$ to -2^{31}; short integer declaration means it has the range $(2^{15}-1)$ to -2^{15}. However the size of long/short integer depends on the machine.

C-instructions:

There are four types of instructions used in a **C** program. These are-

[a] Type declaration instruction: Used to declare the types of variables used in a **C** program.

[b] Input/Output instruction: Used to perform the function of supplying input data to a program and obtaining the output result from it.

[c] Arithmetic instruction: Used to perform arithmetic operations between constants and variables.

[d] Control instruction: Used to control the sequence of execution of various statements in **C** program.

[a] Type declaration instruction: $\Rightarrow$ This type of instruction is used to declare the type of variables being used in the program. Any variable used in the program must be declared before using it in any statement. The type declaration statement is usually written at the beginning of the **C** program.

e.g., int (blank) a,b;
float (blank) a,b;
char (blank) a,b; etc.

[c] Arithmetic instruction: $\Rightarrow$ Such instruction consists of a variable name on the left hand side of "=" and variable names and/or constants on the R.H.S. of the "=" connected by arithmetic operators like +, -, *, / etc.

e.g., a = b+32;
c = d + 3.2;
e = a + b; etc.

The variables and constants together are called ‘operands’ that are operated upon by the arithmetic operators and the result is assigned using assignment operator to the variable on L.H.S. Again there are three types of arithmetic statement that are used in **C**:

(i) Integer mode arithmetic statement:$\rightarrow$In such arithmetic instructions all the operands are either integer variables or integer constants.

```
e.g.,   int (blank) i,sum,j;
        sum = i + 32;
        sum = i + j;    etc.
```

(ii) Real mode arithmetic statement:$\rightarrow$Here all the operands are either real constants or real variables or both.

```
e.g.,   float (blank) a,b,c,d,sum;
        c = 32.4 + 1.2;
        sum = a + b + c;
        d = sum/2.1;
```

(iii) Mixed mode arithmetic statements:$\rightarrow$These are arithmetic statements in which some of the operands are integers and some of the operands are real.

```
e.g.,   float (blank) a,b;
        int (blank) sum,d;
        sum = a + b – d + 32.1;     etc
```

N.B. Input/Output and control statement will be discussed later.

Note:

a] **C** allows only one variable on LHS of ‘=’.That is, z = k*i; is valid whereas k*i= z; is invalid.

b] A statement similar to arithmetic instruction is many a times used for storing character constants in character variables.

```
e.g.,   char (blank) a,b,c;
        a = ‘f’;
        b = ‘g’;
        c = ‘+’;
```

c] Arithmetic operations can be performed on integers, floating point and character constant or variables.

```
e.g.,   char (blank) x,y;
        int (blank) z;
        x = ‘c’;
        y = ‘a’;
        z = x + y;       all are valid.
```

d] No operator is assumed to be present. It must be written explicitly.

```
e.g.,   a = cbd(xy)  [ usual arithmetic statement]
but     a = c*b*d*(x*y); is a C statement.
```

e] Unlike other high-level languages, there is no operator for performing exponential operation. It can be done by repeated multiplication or using Library function. Thus a = 3**2 or a = 3^2 is not allowed. The correct one is a = 3*3.

Integer and float conversion: $\Rightarrow$

(a) An arithmetic operation between an integer and other integer always yields another integer result.
(b) Operation between a real and other real always produces real result.
(c) Operation between a real with an integer always yields real result.

e.g., $\frac{5}{2} \Rightarrow 2$ $\qquad \frac{2}{5} \Rightarrow 0$

$\frac{5.0}{2} \Rightarrow 2.5$ $\qquad \frac{2.0}{5} \Rightarrow 0.4$

$\frac{5}{2.0} \Rightarrow 2.5$ $\qquad \frac{2}{5.0} \Rightarrow 0.4$

$\frac{5.0}{2.0} \Rightarrow 2.5$ $\qquad \frac{2.0}{5.0} \Rightarrow 0.4$

When the type of the expression and the type of the variable on the left hand side of the assignment operator are not same then the value of the expression is promoted or demoted depending on the type of the variable on LHS of '='.

e.g.,
```
int  i;
float   j;
i = 3.2;
j = 30;
```
Here 3 will be stored to variable 'i' and 30.000000 will be stored to 'j'.

e.g., Let k $\Rightarrow$ integer and j $\Rightarrow$ floating point variable.

Arithmetic instruction	Result	Arithmetic instruction	Result
$k = \frac{2}{9}$	0	$j = \frac{2}{9}$	0.0
$k = \frac{2.0}{9}$	0	$j = \frac{2.0}{9}$	0.2222
$k = \frac{2}{9.0}$	0	$j = \frac{2}{9.0}$	0.2222
$k = \frac{2.0}{9.0}$	0	$j = \frac{2.0}{9.0}$	0.2222
$k = \frac{9}{2}$	4	$j = \frac{9}{2}$	4.0
$k = \frac{9.0}{2}$	4	$j = \frac{9.0}{2}$	4.5
$k = \frac{9.0}{2.0}$	4	$j = \frac{9}{2.0}$	4.5
$k = \frac{9}{2.0}$	4	$j = \frac{9.0}{2.0}$	4.5

Hierarchy of operators:

Priority	*Operators*	*Description*
First	***, /, %**	Multiplication, division and remainder of division
Second	**+, -**	Addition, subtraction
Third	**=**	Assignment.

In the expression $a = 3/2 * 5;$ if '/' is performed before '*' the result is 5, whereas if '*' is performed before '/' the result is 0. Thus the compiler cannot generate the same result. Hence the tie is settled using associativity (parentheses).

With in parentheses the same hierarchy is operated according to the rule as mentioned in the above table. Again if there are more than one set of parentheses, the operation with in the inner most parentheses would be performed first, followed by the operations with in the second inner most pair and so on.

# Determine the hierarchy of the operations and evaluate the following expressions.

[i] $a = 2 * 3/4 + 4/4 + 8 - 2 + 5/8 - 2$

[ii] $sah = 3/2 * 4 + 3/8 + 3 - 1$.

Conversion of algebraic expression to **C** expressions:

Algebraic expressions		***C** expressions*
$a \times b - c \times d$	$\Rightarrow$	$a * b - c * d$
$(m+n)(m-l)$	$\Rightarrow$	$(m+n) * (m-l)$
$3x^2 + 2x + 6$	$\Rightarrow$	$3 * x * x + 2 * x + 6$
$\frac{a+b+c}{d+e}$	$\Rightarrow$	$(a+b+c)/(d+e)$
$[\frac{2AX}{C+1} - \frac{x}{3(Z+y)}]$	$\Rightarrow$	$2 * a * x/(c+1) - x/3 * (z+y)$

# The first **C** program: $\Rightarrow$

(i) Blank spaces may be inserted between two words to improve the readability of the statement. However no blank spaces are allowed with in a variable, constant or keyword.

(ii) Usually all statements are written in small case letters.

(iii) **C** has no specific rules for the position at which a statement is to be written. That is why it is often called a free-form language.

(iv) All **C** – statement always must end with a semicolon (;).

Program to calculate the area and perimeter of a rectangle: ⇒

Example program-1
Filename: EP1.C

```
/* This program finds the area and perimeter of a rectangle */
main()
{
        int  p, q, area, perimeter;
        p = 4;
        q = 6;
        area = p*q;
        perimeter = 2*(p+q);
        printf (“area = %d”, area);
        printf (“perimeter =%d”, perimeter);
}       /* end of main */
```

Output:

```
area = 24
perimeter = 20
```

/*   *comment*   */   ⇒ Anything written with in /* */ can be used as comment and will not be executed during operation of program. Comment can occur anywhere in the program.

main( )   ⇒ This is a special function which tells the computer where the program starts. This function should be used at the beginning of all **C** programs.

{ braces } ⇒ Braces {} enclose the computation carried out by main.

printf( )   ⇒*printf( )* function is used to print out the values of variables or everything with in the parentheses. To print any string, i.e., characters it must be with in double quotation marks.

The general format of *printf( )* statement:

printf (“< format string >”, < variable >);
printf (“< characters >”);

< format string > could be,

%f ⇔ for printing real values.
%d ⇔ for printing integer values.
%c ⇔ for printing the string values.

e.g., printf (“ How are you?”); ⇔ Output: How are you?
printf (“How\nare you?”); ⇔ Output: How
are you?

The characters ‘\’ and ‘n’ are called newline characters. It instructs the control to go to the next line (new line).

Input statement**:** $\Rightarrow$ In the previous example the variables p and q are assigned values. If we want to find the area/perimeter of another rectangle with sides 8 and 12 unit then we have to replace p=4 and q=6 by p=8 and q=12 and then run the program again. But this is not a good idea. By using input statement we can assign any desired values to p and q by feeding the data's through the input unit. The input statement in **C** uses the library function *scanf( )*. The general format of *scanf( )* statement are**:**

```
        scanf ("<format string>",&<variable>);
e.g.,   scanf ("%d",&a);  where    a ⇒ integer variable.
        scanf ("%f",&d);           d ⇒ real variable
        scanf ("%c",&b);           b ⇒ character variable.
        scanf ("%d%d",&p,&q);
        scanf ("%d%f",&x,&y);      etc.
```

<u>Example program-2</u>
Filename:EP2.C

```
/* Program to demonstrate input/output statement */
main( )
{
        int p, q, area, perimeter;
        printf ("Enter the sides of the rectangle\n");
        scanf ("%d%d", &p, &q); /* Reads p and q from key board */
        area = p*q;
        perimeter = 2*(p+q);
        printf (" area=%d", area);
        printf ("perimeter=%d", perimeter);
}       /* End of main */
```

Output:

```
Enter the two sides of the rectangle
8 ,<blank> 12  [ supplied by key board]
area=96
perimeter= 40
```

<u>Example program-2</u>
Filename:EP3.C

```
/* Program to convert a Celsius temperature to Fahrenheit */
main( )
{
        float fahrenheit, celsius;
        printf ("enter the temperature in Celsius scale");
        scanf ("%f", &celsius );
        fahrenheit = 1.8*celsius +32.0;
        printf ("Temperature in Celsius scale is = %f ⁰C ", celsius);
        printf ("temperature in Fahrenheit scale is = %f ⁰F ", Fahrenheit);
}       /* End of main */
```

Output:

```
Enter the temperature in Celsius scale
5        [ supplied by key board]
Temperature in Celsius scale is = 5 ⁰C
Temperature in Fahrenheit scale is = 41 ⁰F
```

Arithmetic operator symbol

Operation	For float	For integer
i) Unary minus	-	-
ii) Division	/	/
iii) Remainder obtained in integer division	*Nil*	%
iv) Multiplication	*	*
v) Addition	+	+
vi) Subtraction	-	-

In **C**

e.g., $a \div b \Rightarrow a/b$

$a \times b \Rightarrow a * b$

$a + b \Rightarrow a + b$

$a - b \Rightarrow a - b$

Defining constants:$\Rightarrow$

In **C** a value can be assigned to a variable name when it is declared. If no value is assigned it is undefined. Unless a variable is defined it can't be used in an arithmetic expression.

e.g.,

```
int x = 2;
int y = 3, p = -232;
float m = 0, n = 0;
float b, d;
```

In the above declaration statements the variables b and d are undefined whereas all the other variables are defined.

# *define* : $\Rightarrow$ This is a pre-processor directive instruction and defines value to a symbolic constant for use in the program. '*#define*' is a compiler directive and not a statement and hence this line should not ends with a semicolon. Symbolic instructions are written in upper case to distinguish them from the variable name (usually written in lower case).

e.g., #define <blank> PI <blank> 3.1415927
#define <blank> MAXSPEED <blank> 200

Suppose a constant occurs a dozen times in a program and its value is to be changed. If the constant is declared using *#define*, the change is done in only one place, namely where it is defined and not in all other places.

We mainly use *#define* line to specify a constant when:
i) the constant is used at many places in a program.
ii) the constant is subject to frequent change.
iii) a meaningful name for a constant would aid in understanding a program.

Summary of variable declaration and constants:

Data type	Examples
int	int i, j, m;
unsigned int	unsigned int k=5;
long int	long int p, q;
float	float a=3.1, x;
double	double y,t;

Conversion with float and double: ⇒

If in an expression real variables are declared as float and double appear together, all variables are converted to double.
For example, in the following statement:
float x, y, s;
double p, q, z;
z = y*p/q +s;
x = q/(y + s);

The variable names y and s are assumed to be float but in the 3rd statement as z is double they will be considered as double. In the last statement (y+s) is calculated using single precision but as q is double (y+s) is converted to double before division. As x is single precision, the answer is made single precision and stored in to x.

Assignment expression: ⇒

We have already defined the assignment operator '=' which assigns the value calculated for the expression on the RHS of the operator to the variable name on the LHS of the operator.

e.g., a = x + y*m;
This is the normal case.

C unlike most other languages provides a new assignment operator.
e.g., x+ = y;
This expression is taken as, x = x+y;
In general;
< variable name><operator> = expression> It is interpreted as;
variable name = (variable name) (operator) (expression)

e.g., x* = y; ⇔ x = x*y;
x- = y; ⇔ x = x – y;

Increment and Decrement operator: ⇒

C has two useful operators for incrementing (++) and decrementing (--) the values stored in variable name.

Increment operator(++): ⇒

e.g. 1) y = ++x; ⇔ i) x = x + 1;
ii) y = x;

Here at first x is incremented by 1 and then the result is stored in y.

2) y = x++; ⇔ i) y = x;
ii) x = x + 1;

Here at firstl the value of x is assigned to y and then incremented by 1.

Decrement operator(--): ⇒

e.g., 1) p = --q; ⇔ i) q = q-1;
ii) p = q:

Here at 1 is subtracted from q and then the value is assigned to p.

2) p = q--; ⇔ i) p = q:
ii) q = q-1;

Here the value of q is assigned to p first and then the value of q is decreased by 1.

Multiple assignment: ⇒

C provides the user to do multiple assignments:

e.g., 1] $a = b = c = 1; \Leftrightarrow \begin{array}{l} a = 1; \\ b = 1; \\ c = 1; \end{array}$

2] $a = b = c + d - e; \Leftrightarrow \begin{array}{l} a = c + d - e; \\ b = c + d - e; \end{array}$

3] $a+ = b* = c + d; \Leftrightarrow \quad a = a + b = a + (b * (c + d));$

4] $x* = y + 1; \Leftrightarrow \quad x = x * (y + 1);$

5] $a = (b = d* = c); \Leftrightarrow \begin{array}{l} a = c * d; \\ b = c * d; \end{array}$

# Output function: ⇒

The general form of an output function is;

printf "< format string >", var_1, var_2, ………,var_n;

The *printf( )* function may sometime display only a message and not any variable value. The syntax is;

printf ("<Message>");

e.g., printf ("Hello!");

Output: Hello!

After displaying a message the cursor will remain at the end of the string (message). If we want it to move the next line to display information on the next line we use “\n”.

The syntax is:

```
printf (“Hello\n”);
```