C Programming Language

The ‘C’ Character set:

A character is a symbol used to represent information and to write statements. All the characters of ‘C’ language can be classified as follows

Printable Characters	***Non Printable Characters (Space Characters)***
1. Alphabets	Space Bar
A to Z	Enter key
a to z	Page up
2. Digits or Numerics	Page down
0 to 9	Arrow Keys
3. Special characters	Alt
Ex: + - * / % < > ( ) { } , ‘ “ : ; ! #	Ctrl
$ ^ & _	etc

In addition to the above there can be some backslash characters like
\n \t \v \a \b \f \\ \’ \” \0 etc
For Every character of C Language there is a pre defined 7 digited binary code called as ASCII value.

Tokens :

The smallest individual units in a program are called as tokens .C has different types of tokens like

1. Keywords	2. Identifiers
3. Constants	4. Variables
5. Operators	6. Data types

The entire program can be written by using tokens, spaces and syntaxes of languages.

Key words:

Key words are the words whose meaning was already explained to the compiler or system. For each keyword there is a pre defined meaning, purpose and syntax. These keywords are used as building blocks to construct the C program. All the keywords have to be used according to the given syntax for the given purpose whenever those are needed.

In ‘C’ there are totally 32 keywords, those are,

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

Identifiers:

Identifiers are the words prepared by users for naming purpose. To name variables, functions, pointers, files, arrays etc., we can use these identifiers.

1. Identifiers are constructed with alphabets, digits and one special character underscore.
2. Underscore has to be present only at middle part.
3. Max length is 32 characters.
4. First character must be alphabet.
5. Keywords are not used as characters.

Ex: -

```
abcd
a12_cd
area
max_Sal      etc
```

Constants:

Constants are fixed quantities those do not change their values during the execution of a program .In C language different types of constants are there, those are as follows,

<u>Numerical Constants</u>	<u>Non – numerical constants</u>
1. Integer constants.	1. Character constants.
2. Real constants.	2. String constants.

Integer constants:

- It must have at least one digit.
- No decimal point and part.
- It could be either +ve or –ve.
- No commas or blanks.

```
Ex: -    426
       + 328
       - 6246
       - 6439
```

Integer constants can be represented either by decimal or by octal or by hexadecimal number systems.

Generally we can use decimal system. If we want to use octal constants we can use it by prefixing with 0. If we want to use hexadecimal constants then we have to used it by prefixing with ox or with oX.

Real constants:

A real constant is a number with decimal part. Those are generally called as floating point constants.

The real constants can be represented either by fractional from or by exponential form.

1. Real constant must have at least one digit.
2. Decimal point and decimal part.
3. It can be either +ve or –ve.
4. No commas and blanks.

The fractional format is,

Integer part . Decimal part

Ex: 10.05
-40.08
+100.756 etc.

The exponential format is,

a e b

Here 'a' is mantissa and b is exponent.

a e b is a * 10^{b}.

Ex:

+3.6 e –10
-6.3 e 2
-0.006 e 10

The Real constants also can be represented either by using octal or decimal or hexadecimal number system.

Character constants:

A character constant is any symbol of c character set that is enclosed between single quotation marks.

Ex:

'A'	'7'
'P'	'+'
'Q'	'-' etc.

For each character constant there is an equivalent integer value called as Ascii values. C also supports a special type of characters called as backslash characters. Each backslash character is the combination of one backslash and one character. Even though these are the combination of 2 characters it can be treated as single character constant. For backslash characters also there are ascii values.

Ex:

'\n'	'\q'
'\t'	'\\'
'\b'	'\"
'\"'	etc.

Character constants can be used either as character or as an integer.

String Constants:

It is the collection of zero or more characters enclosed between double quotation marks. Generally these are used to represent the names of places, rivers ,persons etc.

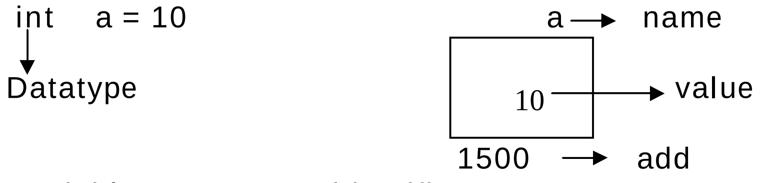
Ex: “abcd” “Rama Rao”
“Krishna Academy” “ab1234” “ab+123”

Variables:

A variable is a part of the memory with a particular name in which we can store any values of a particular type. The value present in this memory location can be changed, altered during the program by using deferent statements.

For each variable these must be a data type, name, value and an address.

Ex: - int a = 10



To name the variables we can use identifiers.

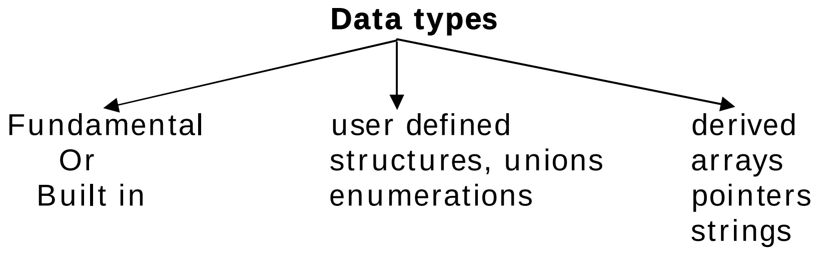
Operators: -

Operators are the symbols with predefined meanings and purposes in C, different types of operators are there like,

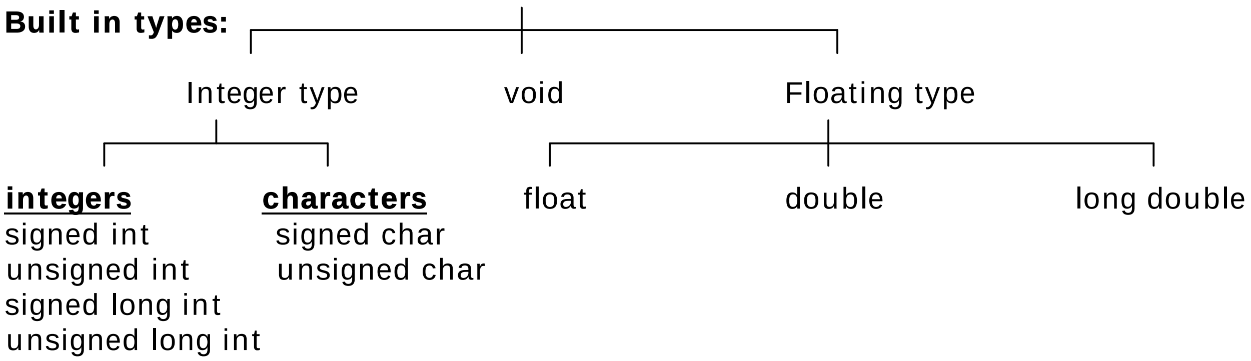
Arithmetical operators.
Relational operators. etc.

Data types: -

Data types in C can be classified under various categories as shown below,


Data types
Fundamental Or Built in
user defined structures, unions enumerations
derived arrays pointers strings

Built in types:



int : for this type 2 bytes of memory is needed i.e. 16 bits.

Among those

1 bit	–	sign
15 bits	–	magnitude

So Range is -2^{15} to $2^{15} - 1$

i.e. –32768 to 32767

Here int, signed int, short int, signed short int, short signed int all are same.

Unsigned int:

For this also 2 bytes of memory is needed. i.e., 16 bits among those.
16 bits totally for magnitude, so memory can be given store sign.

Range is 0 to $2^{16} – 1$

i.e. 0 to 64535.

Here there is no sign, the default sign is +ve. Unsigned int and unsigned short int are same.

Long int:

For this 4 – bytes of memory is needed ie 32 - bits

Among those

1	–	sign
31	–	magnitude

range is $– 2^{31}$ to $2^{31} –1$

- 2147483648 to 2147483647.

Here long int and signed long int both are same.

Unsigned long int:

Here also 4 – bytes of memory is needed ie 32 – bits

0	–	sign
32	–	magnitude

range is 0 to $2^{32} – 1$

is 0 to 4294967295.

Float:

4 – bytes
32 – bits
6 bits for decimal part
1 bit for sign
25 bits for magnitude of integer part
range is 3.4 E- 38 to 3.4 E + 38.

Double:

8 – bytes
64 – bits
12 bits for decimal part
1 bit for sign
51 bits for magnitude of integer part
range is 1.7 E – 308 to 1.7 E 308.

Long double:

10 – bytes
80 – bits
15 bits for decimal part
1 bit for sign
64 bits for magnitude for integer part
range is 3.4 E – 4932 to 1.1 E 4932

Char:

1 – byte
8 – bits
1 - bit for sign
7 – bits for magnitude
range is -2^7 to $2^7 - 1$
is -128 to 127.
Char, signed char both are same.

Unsigned char:

1 byte
ie 8 bits
0 - bits for sign
8 – bits for magnitude
range is 0 to $2^8 - 1$
is 0 to 255.

Void:

This is a data type used to indicate that no value is there. This is not used to declare normal variables. So no memory is needed.

Operators:

C language is rich with its operators. ‘C’ operators can be divided into different groups.

1. Arithmetical operators.
2. Relational operators.
3. Assignment operators.
4. Logical operators.
5. Increment and decrement operators.
6. Bit wise operators.
7. Conditional operator.
8. Unary operators.
9. Mislaneous operators.
   . (dot) , -> (cap) ,size of
   , (comma)
   type cost
   {}
   and [] etc..

Arithmetical operators:

C has 5 arithmetical operators.

Operator	Purpose	Example
+	addition	a + b
-	subtraction	a – b
*	multiplication	a * b
/	division	a / b
%	modular division (remainder)	a % b

Arithmetical operators are used to prepare arithmetical expressions along with numerical operators and parenthesises. During the evaluation of arithmetic expression system follows the priorities of operators.

Operator	Precedence	Associativity
*, /, %	1st	left to right
+, -	2nd	left to right

- If all the operands in the exp are integer then that exp is called as integer arithmetic expression and the final result of that expression is integer only.
- If at least one real operand is there then entire expression is called as real arithmetic expression and the final result is real value.

- % Operator is generally used on integer data.
- Sign of result of the % operator is the sign of numerator.
- During division operation if both numerator and denominator have same signs then result is +ve. Otherwise the result can be either +ve or –ve according to ANSI ‘C’. But the result is –ve according to Turbo ‘C’.

Relational operators:

Relational operators are used to find the relegation between two quantities. These allow the comparison of two or more variables.

C language has 6 relational operators. Those are

= =	is equal to	<=	is less than or equal to
!=	is not equal	>	is greater than
<	is less than	>=	is greater than or equal to

All these operators results either true or false as result . True is represented by 1 and false is represented by 0.

Syntax is

exp1 opr exp2

Ex**: -**

2 > 3 is 2 is less than 3
a+b == 5*d is a+b is equal to 5*d
a <= b is a is less than or equal to b

Relational operators are used to prepare simple conditions. These conditions are used for decision making in selective statements, loops and in jump control statements

Assignment operators:

In ‘C’ we have two types of assignment operators, those are,

1. Simple assignment operator.
2. Short cut assignment operators.

= Is the simple assignment operator. It is used to store the value of an expression in a memory variable.

Syntax is

var = exp;

Here exp can be constant, var or an expression.

First the right operand exp can be simplified and the resultant value can be palced in left side memory variable.

```
Ex: -     a = 10;
          a = b;
          a = b + c + d;
```

Short cut assignment operators are + =, - =, * =, / =, % =, these are used to perform arithmetical operations on a variable and to place the result again in that variable.

Syntax is

var opr exp;

```
Ex: -  a + = b
       a * = c + d etc..
```

The equivalent expression of

```
a + = b         is    a = a + b
a * = c + d   is    a = a * (c + d)
```

Here the left side variable acts as both input and output.

Advantages

1. Length of the statement will be reduced.
2. System complexity reduces.
3. Fastly expressions can be simplified.

Logical operators:

Logical operators are generally used to combine simple conditions and to make compound conditions. These are used as connectors in between conditions.

In C we have only 3 logical operators.

&&	Logical AND
||	Logical OR
!	Logical NOT

Among these &&, || are binary and ! is unary operators.

The syntaxes of operators are

Cond1 && Cond2
Cond1 || Cond2
! Cond

The resultant truth-value of the compound condition depends upon the individual truth-values of simple conditions and the logical operators.

The truth tables of logical operators are as follows.

Cond 1	Cond2	&& Result	Result
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

Cond	! Result
T	F
F	T

Increment and decrement operators:

++	Increment
--	Decrement

Increment and decrement operators are used to increase and to decrease the values of a variable respectively.
These two operators are unary operators

a + + is a = a + 1
a - - is a = a - 1

These operators can be used only on variables.

Increment operator:

Increment operator can be used in two ways,

1. Pre increment ex: + + a
2. Post increment ex: a + +

If the statement is simple statement then there is no difference between those two.

i.e. a = 2 a + + then a is 3
a = 2 + + a then also a is 3

If we use these operators in an assignment statement then the evaluation process includes 4 steps,

1. Pre evaluations.
2. Calculation of result.
3. Storing of the result in left side variable.
4. Post evaluations.

Ex.1: a = 2; ans = a + +;
then
a is 3

ans is 2

Ex.2: a = 2;

ans = + + a;
then
a is 3
ans is 3

Ex.3: a = 2
ans = a + + * a + + * a + +
then
ans is 8
a is 5.

Ex.4: a = 2
ans = + + a * + + a * + + a
then
a is 5.
ans is 125.

Ex.5: a = 5
ans = a + + + a + + ; is an error
ans = (a + +) + (a + +)
then a =7
ans = 10 etc..

Similarly for decrement operator

Bit wise operators:

Bit wise operators are used to operate with bit level data. In ‘C’ we have 6 bit wise operators, those are,

&	bitwise AND
!	bitwise OR
^	bitwise X – OR
<<	bitwith shift left
>>	bitwith shift right
~	One’s complement

If both bits are ones then only & result is 1.

If both bits are zeros then only | result is 0.

If both bits are same then ^ result is zero.

>> is used to shift the bits of the given variable constant for given no of times by filling the left most locations with zeros.

<< in left direction in the same manner.

~ is used to change all 1s to 0s and 0s to 1s.
Among those &, !, ^ are binary operations and >>, << and ~ are unary operators.

Consider all operands as unsigned integers.

```
 Ex: - a = 10 is at bit level     0000  0000  0000  1010
       b = 12 is at bit level     0000  0000  0000  1100
c = a & b;
  =0000     0000  0000  1000
       i.e., c is 8.

C = a | b;
   = 0000     0000       0000  1110
       i.e., c is 14.
C = a ^ b;
     = 0000     0000      0000  0110
       i.e., c is 6.
C = a >> 2;
Then
       C = 0000     0000 0000  0010
       i.e., c is 2.
C = a << 3;
Then
       C = 0000     0000   0001 0000
       C is 16.
C = ~ a
Then
       C = 1111      1111       1111  0101
       i.e., c = 65530.
```

C provides short cut assignment operators for these bit wise operator.

```
& =          >> =
! =          << =
^ ==
```

usage of these shortcut assignment operators is like arithmetical assignment operators.

Unary operators:

Unary operators are the operators work with only one operand.

```
 Ex: -      a negative of a
            -   70.
```

Certain unary operators are already discussed previously in different sections.

Conditional operator:

This conditional expression operator takes THREE operands. The two symbols used to denote this operator are the ? and the : . The first operand is placed before the ?, the second operand between the ? and the :, and the third after the :. The general format is,

condition ? expression1 : expression2;

If the result of condition is TRUE (non-zero), expression1 is evaluated and the result of the evaluation becomes the result of the operation. If the condition is FALSE (zero), then expression2 is evaluated and its result becomes the result of the operation.

Ex: max = (x > y) ? x : y;
If x is greater than y then max = x otherwise max = y

Mislaneous operators:

1. .(dot), →(cap) operators are used to access data members form user defined data types like structures and unions.

2. Size of operator is used to know the memory requirements (no of mem bytes needed) of data types, constants and mem variables.
   sizeof (int) is 2
   sizeof (100.05) is 4
   Double c;
   Sizeof(c) is 8

3. , (comma) operator is used to combine simple statements as a compound statement.
   Int a;
   Int b;
   is equivalent to
   int a, b;

4. typecast operator is used to change the data type of variable, constants and
   expressions.
   (int) 20.25
   (float) a + b + c

5. {} is used to combine diff types of simple statement as a compound statement
   as a single block of statements.
6. [ ] is used to access elements from an array

7. & is used to know the add of a variable.

8. * is used to know the content of a pointer variable.

9. ( ) is used to give well defined shapes and meaning for diff parts of expressions.

Declaration of variables:

Variables are needed to store inputs, outputs, intermediate results and temporary values. In C all the variables needed in a function or in a block or ina program has to be declared at the beginning of function or block or program definitions.

Syntax: -

Data type var 1, var 2, var 3-----------var n;

Ex: -

```
int a, b;
float c;
char ch;      etc....
```

For each variable separately memory can be allocated starting from even addressed memory location. If we did not initialize the memory then garbage values can be placed in those locations.

Variable declaration statements are used to allocate (reserve) the memory for variables.

Declaration and initialization of variables:

Syntax: -

Data type var = values;

int a = 10;

This statement is used to declare the variable and to initialize the memory to a value simultaneously.

If is equivalent to

```
Int a;
a = 10;
```

Type casting:

Type casting is a process through which we can convert the data types of values of variables and constants. We can do it implicitly or explicitly.

Generally after evaluating an expression while placing the result in a variable, by depending upon the data type of destination the data type of value of variable can be changed automatically. This is called implicit type casting.

```
Ex: -    int    a = 10.05;
         float  b = 20;
          int   p = 'A';
         char   c = 105;
```

While calculating result according to our requirement we can change the data type of value of variable or constant explicitly.

```
Float avg = (float) total /3;
    Int a;
```

(float) a is float
(char) a is char etc..

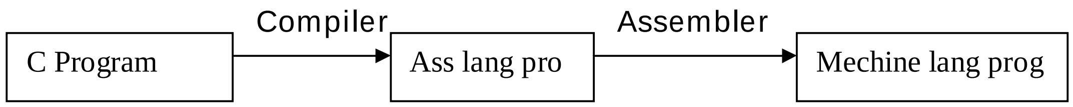
Compilation:

Compilation is a process through which a 'C' language program can be converted into machine language program.

To compile a program we can use an application program called as compiler.

During the conversion process first the given 'C' language program can be converted into assembly language program by using compiler and then assembly language program to machine language program by using assembler.

i.e. In compilation process both compiler and assembler can be participated.



Stages to be gone through to prepare and to run a C program:

1. Composing stage.
2. Linking stage.
3. Compiling stage.
4. Loading stage.
5. Executing.

In the first stage, composing, programmer (we) can compose the C program. To compose a program one editor is needed. C has its own editor. So generally we can type our program in that editing area.

Generally while composing the program we can use so many modules from supporting files (Header files). In the second stage our program has to be linked with the supporting files in which definitions and prototypes of used modules are present to check the validity of used modules. Here an application program linker is used.

In third stage compilation can be taken place. Here the entire C program can be converted into machine language.

In the 4th stage loading, the used modules definitions from the supporting files can be loaded to the compiled code. This can be done with the help of an application program called loader. After loading, program becomes as a complete versioned program.

In the last stage the execution of the program can be taken place. Processor can process the instructions of the program in the given sequence.

'C' program structure:

```
# include <stdio.h>
        -------

main ()
{
        -------
        -------
    }
```

Here # include <stdio.h> is a pre processor directive used to include that header file to our program. This file contains the definitions of library functions like printf() and scanf()

Whenever we use a library function in our program, we have to include the respective header file to our program.

Comments are used to provide explanation to a statement, or to a formula or to a function call or to a block of code. Comments can be written at any where with in the program. Any no of comments can be written in our program.

A comments always in between the symbols /* ,*/. This can be either single line comment or multiple line comment. Comment with in the comment is not allowed.

```
    /*------------*/                                  /*-------------
                                                        -------------
                                                        -------------*/
```

Main () is a function, which is essential for C program. Always program execution starts from the beginning of main () and ended at the end of main (). In each program exactly there must be only one main function. When ever we gave the run command to run a program then operating system will call the main function of the program.

Programmers can give the definition to main function. This body contains,

1. Declarative statements.
2. Expressions & assignment statement.
3. Function calls.
4. I/O functions like printf () and scanf ().

In addition to main () a C program can contain any no of blocks or function definitions.

Formatted out put function:

Printf () is the only formatted output function in C language. This is used to print messages, values of any type and any no of variables and constants simultaneously and separately.

Syntax is

` **printf (“control string”);**

printf ("control string", arg1, arg2,--------argn);

First one is used to print only messages where as second one is used to print both messages and values of variables and constant.

The control string can contain,

- ♦ Words or characters that are to be printed as they are.
- ♦ Conversion characters that begins with % sign.
- ♦ Escape sequence characters those starts with \ .
- ♦ Field specifications.

Here arg1, arg2, ------, argn are arguments that represent the individual output data items.
The arguments can be return as constants, variables, expression and function references.

Note

- ♦ The number of conversion characters must be equal to the no of arguments.
- ♦ The mapping b/w conversion characters and arguments can be taken place from right to left.

The conversion characters and their purposes for printf function are as follows.

Conversion Character	***Meaning***
%c	To display a single character (both signed and unsigned).
%d	To display a signed integer in decimal format.
%e	To display a float value in exponential format.
%f	To display a float value in decimal format.
%g	To display a float to any format.
%i	To display a signed integer in any format.
%o	To display a integer in octal format.
%s	To display a string.
%u	To display a unsigned integer.
%x	To display a integer in hexadecimal format.
%ld	To display a long integer.
%lu	To display a unsigned long integer.
%lf	To display a double.
%Lf	To display a long double.

Controlling the cursor position

In printf () function we can use back slash character to control the position of cursor un to print the output in required format. There black slash character are called as escape sequence characters.

The escape sequence characters and their purposes are as given.

Character Purpose

\b	backspace
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\"	double quote
\'	single quote
\	line continuation
\nnn	nnn = octal character value
\0xnn	nn = hexadecimal value (some compilers only)

Formatted input function:

Scanf () is the formatted input function used to take any no of inputs of any type.
i.e. this function can be used to enter any combination of numerical values single characters and strings this function returns the number of data items that have been entered successfully.

As syntax is:

scanf ("control string", arg1, arg2,-------argn);

Here the control string refers to string containing certain formatting information.

The control string can contain the information like,

- ♦ Conversion characters.
- ♦ Fields specification.
- ♦ White spaces.

arg1, arg2,--argn are arguments that represent the address of data items.

Conversion Character	***Meaning***

%c	To read a single character (both signed and unsigned).
%d	To read a signed integer in decimal format.
%e	To read a float value in exponential format.
%f	To read a float value in decimal format.
%g	To read a float to any format.
%h	To read a short integer.
%i	To read a signed integer in any format.
%o	To read a integer in octal format.
%s	To read a single word string.
%u	To read a unsigned integer.
%x	To read a integer in hexadecimal format.
%ld	To read a long integer.
%lu	To read a unsigned long integer.
%lf	To read a double.
%Lf	To read a long double.
[...]	To read a multi word string.

Note: - For all type of variables except character and strings we have to write & variable name in the place of arguments. For characters and strings & is optional.

Program Control Sequences:

While executing the program, processor can execute the statements of the program in a particular order. That order of execution of the statements is called as program control sequences.

'C' language supports the following program control sequences.

- ♦ Linear or sequential program control.
- ♦ Selective program control statements.
- ♦ Loop control statements.
- ♦ Jump control statements.

Sequential or linear program control:

In this sequence all the statements of the program can be executed in the same sequence that were given in the program.

Selective statements:

In this sequence many alternatives are available. To move forward one alternative has to be selected from those alternatives. For this a decision has to be made. By depending upon the result of that decision one alternative can be selected and all other can be rejected.

Selective statements are available in different formats.

1. simple if.
2. if – else.
3. nested if – else
4. if – else ladder
5. switch

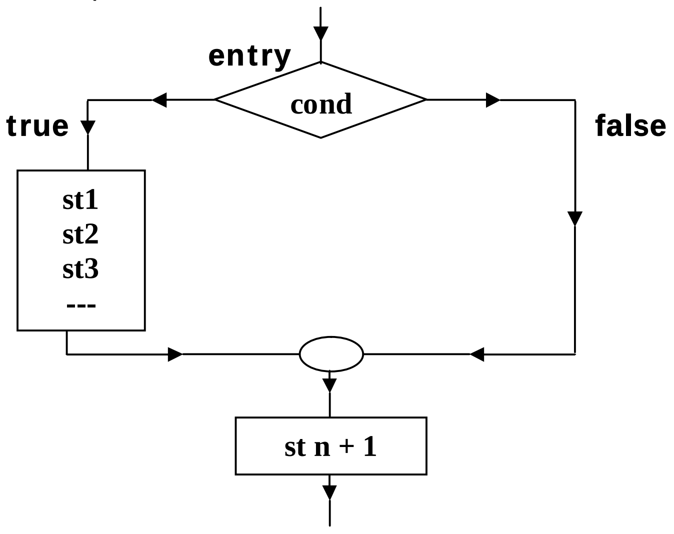
Selective statements are also called as **conditional statements**.

Simple if:

Syntax:

```
if (cond)
{
        st1;
        st2;
        ---
        ---
        stn;
}
st n + 1;
st n + 2;
```

The control flow of simple if is as follows.



During the execution, first condition will be evaluated. If the truth value of the condition is true that block will be selected and all statements of that block st1, st2, ----stn can be executed. If the truth value of the condition is false then that block can be rejected. Finally st n + 1, st n + 2 can always executed.

Here a decision has to be made to select or to reject that block.

Example: 1

```
if (qty>=1000)
{
        dis = 10;
}
```

Example: 2

```
if (sal>=3000)
{       da = 10;
        hra = 20;
        pf = 5;
}
```

If – else statement:

Here two alternatives are available, one has to b e selected .To select one alternative among those two we can use if –else statement. Here a condition will be validated, by depending upon the truth value of the condition one block can be selected and can be rejected.

Syntax is

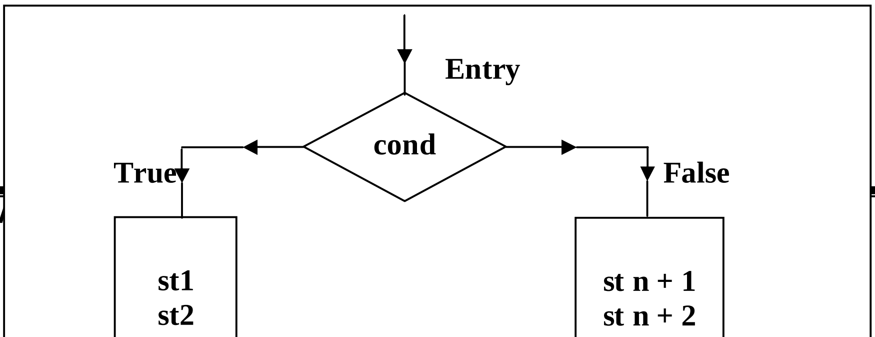
```
 if ( cond )
   {
         st1;   ─┐
         st2;    │ if block
         ---     │
         stn;   ─┘
   }


else
   {
         st n + 1;  ─┐
                     │ else block
         ---------   │
         st m;      ─┘
   }
   st m + 1
   st m + 2
```

If the truth-value of the condition is true then if block can be selected, else block can be rejected. If the truth-value of cond is false then if block can be rejected, else block can be rejected.

Finally statements st m + 1, st m + 2 can be executed without bothering about the truth value of condition.

The control flow of if else is as follows,



Example 1:

```
if (num % 2 = = 0)
{
        printf("the number is even");
}
else
{
        printf("the number is odd");
}
```

Example 2:

```
if (marks>= 35)
{
        printf("u r result is pass");
}
else
{
        printf("u r result is fail");
}
```

Nested – if else:

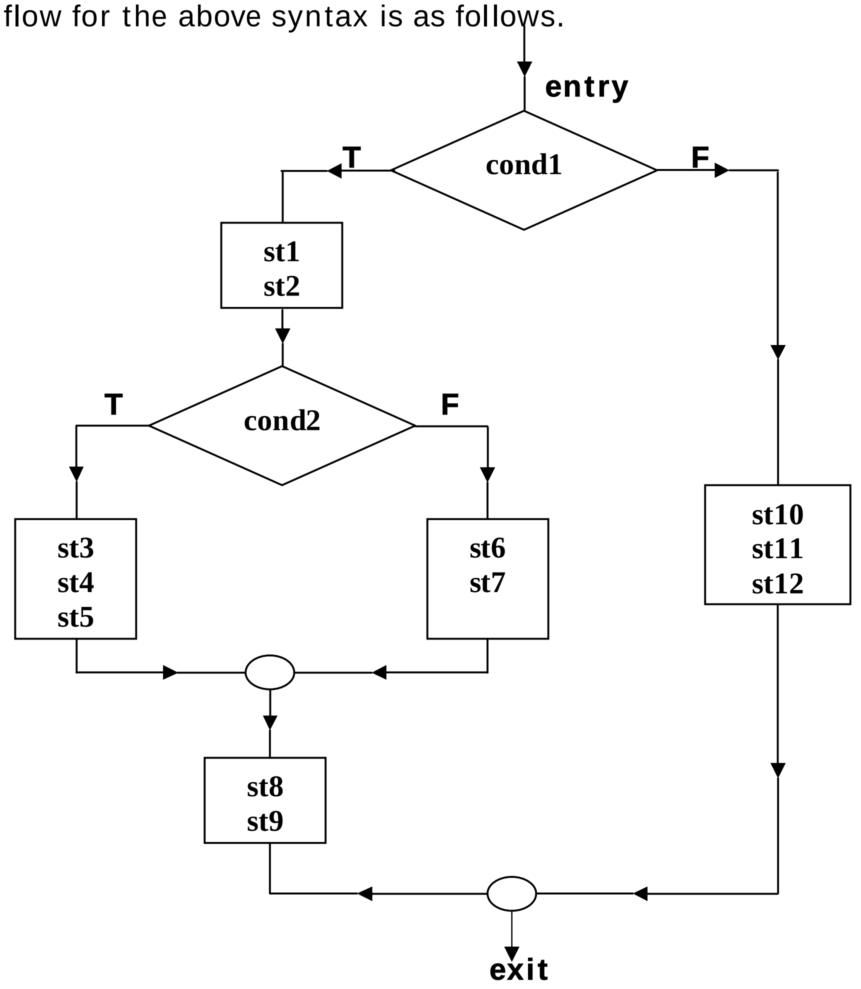
If these is an if –else statement with in the if block or else block of another if – else statement. Then that arrangement is called as nested if else.

Syntax: -

```
if (cond)                                  st1;
{                                          st2;
```

```
        if (cond2)
        {
                st3;
                st4;
                st5;
        }
        else
        {
                st6;
                st7;
        }
        st8;
        st9;
}
else
{
        st10;
        st11;
        st12;
}
```

Here in the above syntax, one if else is given with in the if block of another if else. The control flow for the above syntax is as follows.



An if - else statement can be present either in if block or in else block or in both the blocks of another if – else statements.

if – else ladder:

If there are more than two alternatives, to select one from those we can use if –else ladder. To deal with more than two alternatives one condition is not sufficient, because for a condition only two outcomes (true and false) are there.

By using those two outcomes we can deal with only two alternatives. To deal with more than two alternatives we have to use more number of conditions.
If there are 'n' alternatives n –1 conditions are needed.

Syntax is: -

```
if (cond1)
  {
        block1
  }
else
        if (cond2)
        {
                block2
        }
        else
                if (cond3)
                {
                        block3
                }
                else
                        {
                        block4
                }
```

During the execution first cond1 will be evaluated, if the truth-value of the cond is true block1 will be selected otherwise 2nd cond will be evaluated. If its truth-value is true 2nd block will be selected. Otherwise 3rd cond will be evaluated and so on. If all the conditions are failed then the last block will be selected.

If a block was selected then all the remaining conditions and blocks will be ignored.

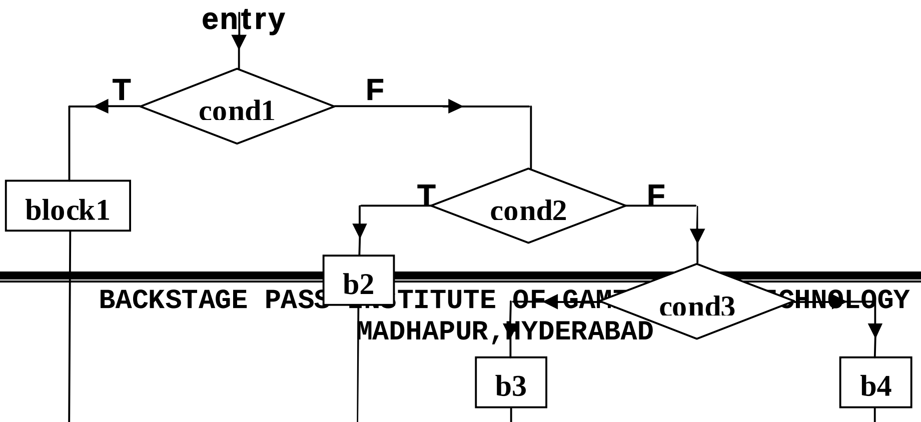
Cond1 is used to make a decision regarding selection or rejection of 1st block.

Cond2 is used to make a decision regarding selection or rejection of 2nd cond. And is needed when the first cond is failed.

To select different blocks different no of conditions has to be validated and the selected blocks and truth-values of conditions are as follows.

Block / conditions	**cond1**	**cond2**	**cond3**
block1	T	-	-
block2	F	T	-
block3	F	F	T
block4	F	F	F

The control flow of the above syntax is as follows.



T **F**

exit

In if – else ladder to select different alternatives different number of conditions has to be evaluated. It leads to non-uniform time complexities for different runs of the same program. To deal with more number of alternatives more number of {and} are needed whose will lead to much complexity in debugging. With the usage of if – else ladder the program will be skewed from top left to bottom right.

To overcome all these problems we can use switch selection statement.

Switch: -

Syntax:

```
Switch (exp)
{
case v1:    st1;
            st2;
            st3;
            break;

case v2:    st4;
            st5;
            st6;
            break;

case v3:    st7;
            st8;
case v4:    st9;
            st10;
            break;

case v5:
case v6:    st11;
            st12;
            st13;
            break;

default:    st14;
            st15;
            st16; break;
}
```

Here exp is any expression which can result an integer. In the place of exp we can use int or char variable or an arithmetical expression.

With in the body of switch there are so many case labels and finally an optional default case.

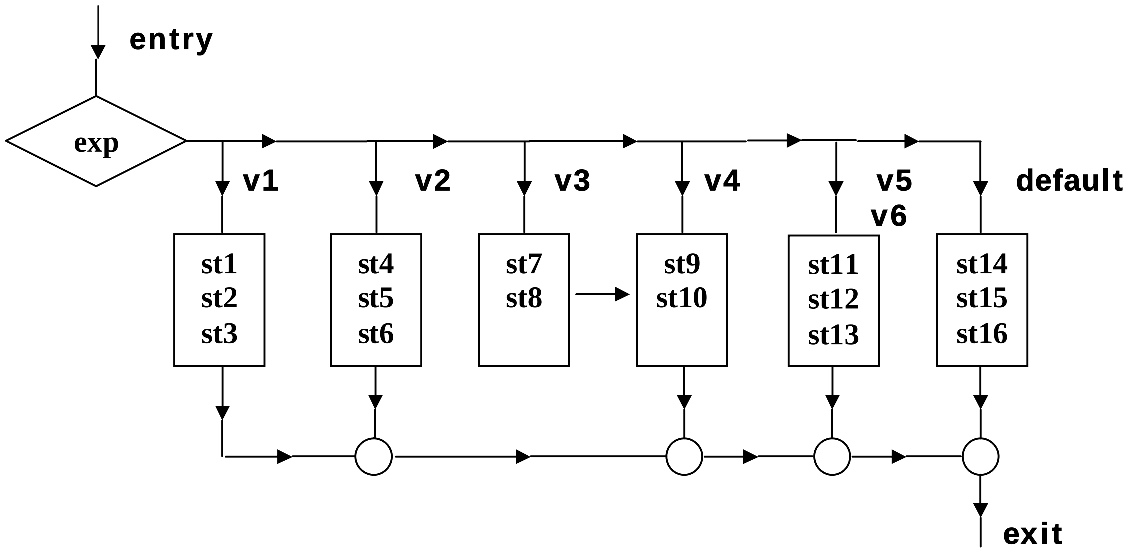
Break is a key word written as last statement of so many blocks. Whenever the break statement was executed system control will be shifted to out of switch.

During the execution first the exp will be evaluated as a result we can get an integer value. Then the system will search for the case with that integer label. If there is such case system control will jump to the beginning of that case and all the statement of that block will be executed. Finally break statement will be executed, as a result system control will be shifted to out of switch.

If there is no matching case then system will search for default case. If it is there system control will be jumped to default case. If the default case is also not there then system control will be shifted out of switch with out executing any case with in the switch body.
If there is no **break** statement then system control will simply enter into next case.

- Switch cases can be written in any order.
- No need to use brasses around the blocks.
- All the blocks can be written vertically.
- To select any block only one expression has to be evaluated.

The control flow of switch statement is as follows.



Loop control structures:

The loop control structures are used to execute a set of statements repeatedly for more then one time. In C language as along as a condition was satisfied system can execute the set of statements.

For each loop in C the following parts are needed.

1. Body

If is the set of statements those has to be executed repeatedly.

2. Condition

For each loop a condition is needed. As long as the condition is satisfied the loop body can be repeatedly executed.

This condition must have at least one variable.

3. Initialization

For the variable of cond a value has to be given to check the cond for first time, this is called as initialization. It can be done by either an assignment statement or by an input function call.

4.Updatation

To check the condition for the 2nd time onwards the value of the variable has to be updated. This is called as updatation. This can be done by either an expression or by an input function call.

In C language, we have 3 loop structures, those are,

1. for loop.
2. while loop.
3. do – while loop.

for loop:

syntax:

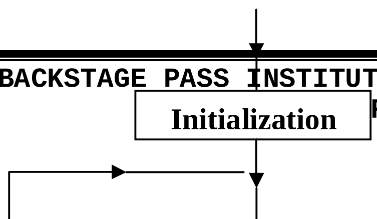
```
for (initialization; condition; updatation)
      {
            st1;
            st2;
            st3;
            ---
            ---
            stn;
      }
```

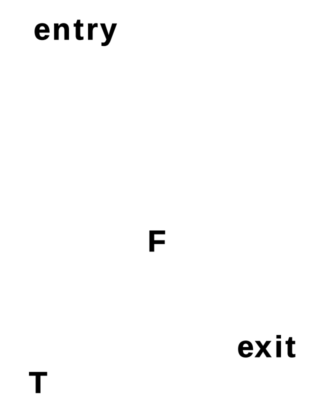
Here first initialization will be taken place, then condition will be evaluated. If the truth-value of the condition is true the control will enter into the loop and all the statements will be executed and then updatation will be taken place. With that new value again condition will be evaluated. If the truth-value of the condition is again true same thing will be repeated.

In this way as long as the truth - value of the cond is true the loop body will be executed. Once the truth-value of the cond is false system control will be shifted out of loop.

Here first cond will be evaluated, if the truth - value of the cond is true then only body will be executed. Hence this loop structure is called as pre – tested loop structure.

If the condition was failed at the beginning it self then without executing the body at least once the sys control will comes out from loop.





If the body was executed for n times, then different parts of the loop were executed as follows.

Initialization	1 time.
Updatation	n times.
Condition	n + 1 (n Ts and 1 F).

Example 1:
```
for (i = 1; i<= 10; i++)
    {  printf("%d", i);  }
```

Example 2:
```
for (count = 0; count < = 5; count ++)
{
cum_value = cum_value + count;
printf("%d %d, count, cum_value);
  }
```

Example 3:
```
for (n = 1; n<= 10; n ++)
{
sqr = n*n;
cube = n*sqr;
printf("%d %d %d, n, sqr, cude);
  }
```

Note:

♥ The body of the loop in for statement need not contain any statement at all. In that Case, it becomes a do nothing loop or time delay loop.
Ex: for (i = 1; i<=2000; i++) ;

This loop executes 2000 times, without doing anything.

♥ The condition can be a compound expression made up of various relational expressions connected by logical AND, OR operators.
Ex: for (i = 1; i<20&&sum<100; i++)

```
            {
                    sum = sum + i;
                    printf("%d %d\n", i, sum);
            }
```

♥ In the for construct, the presence of two semicolon within the parenthesis is compulsory. But any or all the three parts may be blank. The initialization can be done outside the loop. The incrementing step can be incorporated within the loop.
Ex:

```
      int i = 0;
    for ( ;i<= 4;  )
    printf("%d:, i ++);
```

♥ If the test condition is omitted, the for statement sets up an infinite loop.
Ex: int i =1;

```
    For ( ; ;)
            {
                    ---
                    ---
            }
```

♥ The for statement will never be executed if the test condition fails at the beginning itself.

Ex:

```
    for (x = 10; x<10; x = x –1)
            printf("%d", n)
```

♥ More than one variable can be initialized / incremented a time in the for statement using comma operator. That is, we may have more than one expression in place of the first and third expressions of the for loop.

Ex:

```
    for (i = 0, j = 9; i<9; ++ i, j--)
    {
            printf("%d %d", i, j)
    }
```

Here both i and j are initialized and modified.

While loop:

Syntax:

```
While (loop cond)
    {
    Body
    }
```

Here

- Initialization can be taken place before the loop.
- Updatation can be with in the loop body.

- Here first initialization will be taken place, then condition will be evaluated. If the truth-value of the condition is true the control will enter into the loop and all the statements will be executed and then again condition will be evaluated. If the truth-value of the condition is again true same thing will be repeated.
- In this way as long as the truth - value of the cond is true the loop body will be executed. Once the truth-value of the cond is false system control will be shifted out of loop.
- Here first cond will be evaluated, if the truth - value of the cond is true then only body will be executed. Hence this loop structure is called as pre – tested loop structure.

- If the condition was failed at the beginning it self then without executing the body at least once the sys control will comes out from loop.

Do-while:

Syntax:

```
do
{
  Body
} while (cond);
```

During the execution control can directly enter in to the loop body all the statements of the loop body can be execute, then loop condition can be tested. If the loop cond was satisfied control again enter in to the loop body and the same can be repeated.

- In this way as long as the truth - value of the cond is true the loop body will be executed. Once the truth-value of the cond is false system control will be shifted out of loop.

- Here once the body can be executed definitely.
- Here first body can be executed the cond can be tested, Hence this loop can be called as post tested loop.

Jump Control Statements:

- These can be used to move the system control from one place of the program to another place.

In C language 5 Jump control statements are There, Those are

1. Break

2. Continue
3. Go to
4. Function calling statement
5. Return

Break:

In can be used only in two cases

1. In switch.
2. In loops.

In switch statement generally at the end of each case block we can write break. During the execution whenever the break statement was executed immediately system control can be switched to outside of switch.

In loops we can write the break under a condition. In each iteration of the loop, condition can be tested, if it was satisfied break can be executed, immediately control can be shiftedto out side of loop.

```
for( init; loop cond; updatation)
{
        st1;
        St2;
        St3;
        if (cond)
           break;
        st4;
        st5;
}
```

In any loop body whenever the break statement was executed immediately system control can exit the loop.

Continue:

Continue is the Jump control statement which can be used only in loops.
Continue can be written with in the loop body under a condition. During the execution whenever the continue statement was executed control can be moved to next round by skipping the remaining statements of the current round.

```
for(int; loop cond; updatation)
{
        st1;
```

```
    st2;
    st3;
    if(cond)
      continue;
    st4;
    st5;
    }
```

- Whenever the continue was executed control can be moved to updatation and then to loop cond by skipping the st4, st5.

```
While(loop cond)                    do
{                                   {
    st1;                                st1;
    st2;                                st2;
    st3;                                st3;
    if(cond)                            if(cond)
    continue;                           continue;
    st4;                                st4;
    st5;                                st5;
    }                               } while (loop cond);
```

- In case of while and do-while loops whenever the continue statement was executed control can be directly moved to loop cond.
- Continue can be used to move the system control to the next iteration.

Goto:

goto is the Jump control statement which can be used to move the system control from anywhere to anywhere within the program.

Syntax:

goto label;

Ex:- goto abc;

- Whenever the goto statement was executed system can search for the label and then control can be directly moved to the respective label.
- With goto control can be moved from anywhere to anywhere ie in inforward direction, in breakward direction, from loop out to loop in loop in to loop out ect.
- With the usage of goto statement the structure and logic of the program can be disturbed the program can be called as unstructured program.

Hence itd is advise not to use the goto statement.

Nested Loops:

- If we have the loop body with in the body of another loop body then that arrangement can be called as nested loops.
- During the execution, in each iteration of outer loop the entire inner loop i.e all possible iterations inner loop can be taken place.

```
for(i=1; i<=5; i++)
    {
            printf("\n hi");
            for(j=1; j<=3; j++)
             {
                printf("\n %d %d", i,j);
              }
             Pf("\n bye");
    }
```

```
o/p:         Hi          Hi                    Hi
             11          21                    51
             12          22 … … … … …          52
             13          23                    53
             Bye         Bye                   Bye
```

ARRAYS

An Array is the collection of similar type of data from the continues memory locations with common name.

Declaration of Array:

Syntax:-

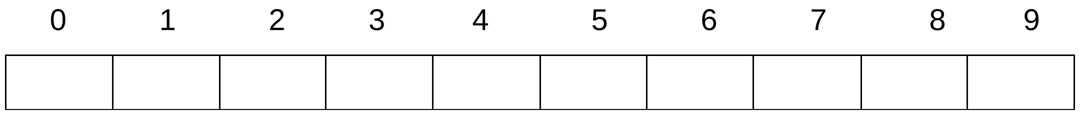
datatype arrayname[size];

here,

datatype	–	any valid fundamental or user defined type.
arrayname	–	any identifier .
size	–	must be +ve int constant.

- ➢ Here size is the no. of locations which in the array.
- ➢ With the declaration memory can be allocated. The amount of memory = size*sizeof (data type);

```
Ex:- int a[10];
          mem=10*sizeof(int)
                   = 10*2
                 = 20 bytes.
```



20 bytes

- ➢ To access each location of the array unique index values can be given to the locations called as subscripts. These can be the integer values from 0 to size -1.
- ➢ To access each location of the array we can use the array name and subscript as follows.

```
a[ i ]          a[ 0 ]
                a[ 1 ]
                a[ 2 ]
                      ect....
```

- ➢ If we dint initialize the array all the locations of the array contains garbage(default) values.

Initialization of Array:

An array can be initialized with a set of elements as follows.

```
int a[10]={10, 20, 30, 40, 50};
```

0	1	2	3	4	5	6	7	8	9
10	20	30	40	50	\0	\0	\0	\0	\0

- Here first array can be created with the given size and the given elements can be placed fron the first location of the array. The remaining locations can be automatically initialized to null values (zeros).
- If we dint mention the size in the declaration and initialization statement then no. of elements can be considered as size of the array as follows.

```
int a[  ] = {10, 20, 30, 40, 50};
```

Here size is 5

0	1	2	3	4
10	20	30	40	50

Reading the values from keyboard:

To read n elements to any integer array **a** we can use the following code.

```
for (i=0; i<n; i++)
 scanf (“%d”, &a[i]);
```

Printing the contents of array:

To print the first n elements of an integer array a we can use the following code.

```
for(i=0; i<n; i++)
 printf(“%d \t”, a[i]);
```

2D Arrays

A 2D Array is the collection of similar type of data from the continuous memory locations with common name organized in certain rows and columns i.e in a rectangular Arrangement.

Declaration of 2D Array:

Syntax to declare 2D Array is as follows

datatype arrayname [size 1] [size 2];

Here

- datatype can be any valid fundamental or user defined type.
- Array name can be any identifier.
- Size1 is the no. of rows and Size2 is the no. of columns.
- Size1, Size2 must be +ve int constants.

With the declaration memory can be allocated.

The amount of mem = Size1 * Size2 *Sizeof(datatype)

Ex**:**   int a[4] [5];

With this 40 bytes of memory can be allocated from the continuous locations and can be distributed for 20 locations and then can be arranged in 4 rows and 5 columns.

0	1	2	3	4

- ➢ To identify each row unique index values were given to rows called as row subscripts. These can be int values from 0 to size1-1.
- ➢ Similarly for columns from 0 to size2-1.
- ➢ To access each location of the array we can use array name, row sub script and column sub script as follows.

a[i] [j]

Here a $\longrightarrow$ array name

i $\longrightarrow$ row subscript

j $\longrightarrow$ column subscript

Initialization of 2D Array:

Always a 2D Array can be considered as the collection of 1D Arrays i.e each row as a 1D Array.

Consider a 2D Array

Int a [4] [5].

It can be considered as the collection of 4 1D Arrays each of size 5. To initialize a 2D Array we can use a set of element sets as follows.

int a [5] [5] = { {10, 20, 30, 40}, {5, 6, 7}, {5, 15, 20, 25, 30}, {100, 200} };

0	1	2	3	4
10	20	30	40	50
5	6	7	\0	\0
5	15	20	25	30
100	200	\0	\0	\0
\0	\0	\0	\0	\0

Each set can be used to initialize a row .If any locations are remaining in the row those can be initialized to zeros .If any rows are remaining those also can be initialized to zeros.

If the sizes were not given in the declaration statement then it can be as follows.

int b[ ] [ ] ={ {10, 20, 30, 40}, {5, 6, 7}, {100, 200, 300}, {5, 6, 7}};

Here….

Size1 $\longrightarrow$ no. of sets i.e 4

Size2 $\longrightarrow$ Max no. of elements in any set i.e 4

0	1	2	3
10	20	30	40
5	6	7	\0
100	200	300	\0
5	6	7	\0

We can also initialize the 2D Array with a single set of elements also as follows.

int a[4] [4] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

0	1	2	3
10	20	30	40
50	60	70	80
90	100	\0	\0
\0	\0	\0	\0

Reading the elements from Keyboard:

To read elements to an integer Array a of m rows and n columns we can use the following code.

```
for (i=0; i<=m-1; i++)
for (j=0; j<=n-1; j++)
  scanf (“%d”, &a[ i ] [ j ]);
```

Printing the contents of 2D Array:

To print the contents of an integer array a of m rows and n columns.

```
for (i=0; i<=m-1; i++)
  {
    For (j=0; j<=n-1; j++)
       printf(“%d\t”, a[ i ] [ j ]);
   printf(“\n”);
}
```

Functions

A function is a set of statements with a particular name which can be used to perform a particular task.

For each function these can be 4 parts

1. body
2. function name
3. parameter list
4. return type

- The function body is set of statements to do the task. Total procedure and logic can be implemented in the body.

- Function name can be any identifier. Relative names can be used for functions. It can be used as handle to call the function.

- For each function zero or more parameters can be passed. The names, data types and order of parameters can be specified in the parameter list.

- From each function at most one value can be returned as result. If any value is returning its data type can be mentioned as return type. If no value is returning then return type can be void.

All the functions of C language can be classified into 2 types.

1. Library functions.
2. User defined functions.

Library Functions:

These are the functions whose definitions were already given by the designers of C language. The definitions of all these functions are available in C Standard Library.

C Library is the collection of header files. Each header file contains some pre-defined functions related to an application.

Examples of header files are…..

Stdio.h
Conio.h
Math.h
String.h
ect….

Examples of functions are……

Printf( )
Scanf( )
getchar( )
putchar( )
sqrt( )
pow( )
clrscr( )
getch( )
ect…..

- ➢ To use any library function we have to know the name, purpose, parameter list, return type and header file name.
- ➢ First we can include the header file to our program and then we can directly call the function with appropriate inputs.
- ➢ To use the library function no need to know its body or logic.

User defined functions:

Library functions are available only for certain specific purposes not for each requirement of the user.

If we want to do a particular task and if library function is not available then we can define our own function and we can use it.

- ➢ User defined function is the function which can be defined by the user for his own purpose.
- ➢ While working with user defined functions we need to go through 3 stages, those are…..

1. Defining the function
2. Declaring or prototyping the function.
3. Calling or using the function.

Defining the function:

The syntax to define the function is as follows.

```
returntype     functionname(parameter list)
  {
      Body
  }
```

- Here the body can contain declarative and executable statements.
- Declarative statements can present at the beginning of body. These can be used to declare additional variable it needed in that function.
- Executable statements can be used to implement the logic or procedure.
- Finally we can have return statement. It can present in any one of the following formats.

  return; $\longrightarrow$ in void type functions.
  Return exp; $\longrightarrow$ in other functions.

- Return can be used to return the system control along with result if any.
- It is the lastly executable statement in the body of function. Once it was executed control can be returned to the calling statement present in the calling function.
- If the return type was not specified to any function, default return type is int.

Prototyping

The function can be prototyped as follows

returntype functionname (para list);

- This can be used to make intimation about the creation and usage of that function to the compiler.
- It can be given either locally or globally.
- Within the parameter list types are important no need to mention the names.
- If the function definition was given before it's calling in the program no need to write the prototyping statement.

Calling of the Function:

Once the function was defined and prototyped, then we can call it within the program at anywhere and for any no. of times with proper inputs.

Example:-

```
main()
{
int X;
Long ans;
Long fact (int);
clrscr();
pf("enter any integer");
sf("%d",&x);
ans=fact(X);
pf("factional is %\d",ans);
getch();
}
```

```
Long fact (int n)
{
Long prod=1;
Int i;
For (i=1; i<=n; i++)
Prod=prod*I;
Return prod;
}
```

- The parameters in the function definition can be called as formal parameters.
- The parameters in the calling statement can be called as actual parameters.
- There must be proper mapping between the actual and formal parameters regarding to
  1. No. of parameters.
  2. Datatypes of parameters.
  3. Order of parameters.
- Program execution can be started from the beginning of main function and can be ended at the end of main function.
- Whenever the function calling statement was executed control can be moved to the definition of called function.
- With the execution of function calling statement, mapping between actual and formal parameters can be taken place. As result new variables can be created as formal parameters and the actual parameter value can be copied to them.

- Whenever the return statement was executed control can be returned to the calling statement present in calling function.

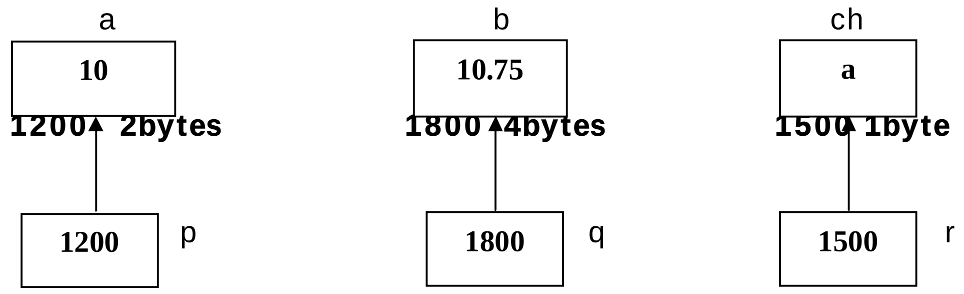
Pointers

- Consider the following declaration

  int a=10;

- If we store the address of a variable in some other variable then it can be called as pointer variable
- In C language we can have 2 types of variables, those are..
  a. Data variables
  b. Pointer variables.
- Data variables are the variables used to store user data like inputs, outputs, intermediate results, temp values ect.
- Pointer variables are the variables used to store the addresses of some other variables.
- Here P is the pointer variable which conditions the address of a i.e p is pointer to a.
- Symbolically it can be represented with an arrow from p to a.
- Consider the following declarations.

  Int a=10; float b=10.75 char ch='a'



- Here a,b,c are data variables.
- P,q,r are pointer variables.
- p is the integer pointer variable.
- q is the float pointer variable

- r is the char pointer variable
- All the pointer variables used to store the addresses. All the address are unsigned integers.
- To store the unsigned integer as address for any pointer variable 2 (4) bytes of memory is needed.
- The type can be given to the pointer variable by depending upon the source variable.
- In the above p can be called as integer pointer variable because if contains the address of int variable.
  - Similarly q is float pointer variable
  - r is char pointer variable

Declaration of pointer:

The Syntax to declare pointer variable is as follows

Datatype * ptrvar;

- Here datatype can be any fundamental or user defined pointer.
- The * Symbol before the variable name indicates that it is the pointer variable.
- We can have multiple stars, the no. of stars indicates the level of pointer.

Ex:
```
int * p;          // 1 level pointer variable
Float ***q;       // 3 level pointer variable
    ect...
```

Initialization of Pointers:

- We can initialize a pointer variable with the add of same type of data variable.

```
Int a;
Int *p;        // declaration
P= &a;         //Initialization
```

- We can declare and initialize simultaneously as follows.

```
Int  a;
```

Int *p=&a;

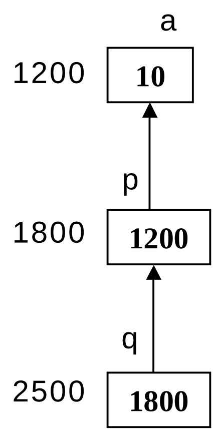
Accessing the pointers:

Consider the declaration

Int a=10;

Int * p;

P=&a;



Here a is 10

&a is 1200

P is 1200

*P is 10

Hence we can conclude that

a=*p=value of a

p=&a=add of a

- Accessing the data of a variable with variable name is called as direct access i.e **a**.
- Accessing the data of a variable with pointer can be called as dereference access as ***P**.
- If q is another pointer which contains the add of P then ,

int **q; $\longrightarrow$ declaration

q=&p; $\longrightarrow$ initialization

Here q is 1800

&p is 1800

*q is 1200

**q is 10

Operations on pointers:

- If p1, p2 are two pointers of same type then P1+p2 , P1*p2 , P1/ P2 , P1% P2 are not valid operations

  Only p1-p2 is valid.

Here The subtraction is not direct subtraction. The result of subtraction operation is the no. of that type of elements those can be placed in between the respective two memory locations.

Ex:- int *p1, *p2;

p1=&a;

p2=&b; p1 p2

$\boxed{1800}$ $\boxed{1200}$

p1-p2

= (1800-1200) / sizeof (int)

= 600/2 bytes

= 300 elements

- The final result is scalar but not pointer again.

Pointer + Scalar:

Adding a scalar value of the pointer is not direct addition. If is adding of the memory of scalar no. of elements.

Ex:- int *p;

P=&a; P

P+25 $\boxed{1200}$

= 1200+25*sizeof (int)

= 1200+25*2bytes

=1250 $\longrightarrow$ again address.

Pointer – Scalar:

It is the Subtraction of memory of the given scalar number of elements from the given address i.e. pointer.

Ex: int *P;

P=&a; P

p-15 $\boxed{1200}$

=1200-15*sizeof (int)

=1200-15*2bytes

=1200-30

=1170 $\longrightarrow$ again address.

Ptr++ :-

Ptr++ is equivalent to ptr=ptr+1

Ex:- int *p;

P=&a; P

P++ $\boxed{\begin{matrix}\textbf{1202}\\ \text{~~1200~~}\end{matrix}}$

- p=p+1

=1200+2bytes

=1202

Ptr-- :-

ptr--

ptr=ptr-1

Advantages of Functions:

- Functions facilitate the factoring of code. Every C program consists of one main( ) function typically invoking other functions, each having a well-defined functionality.
- Functions therefore facilitate:
  - Reusability
  - Procedural abstraction
- By procedural abstraction, we mean that once a function is written, it serves as a black box. All that a programmer would have to know to invoke a function would be to know its name, and the parameters that it expects.

Invoking Functions:

- In C, functions that have parameters are invoked in one of two ways:
  - Call by value
  - Call by reference

Call by Value

```
main()                                          }
 {
        int a=10, b=20;
        void swap(int, int );
        clrscr( );                                  void swap (int x, int y)
        swap(a, b);                                         {
        printf("%d%d\n",a,b);                               int temp = x;
x=  y;
 y=temp;
}
```

- Here by passing the values of inputs function can be called, Hence it can be called as call by value.

- Here actual parameters can be either data variables or constants. Whereas formal parameters must be the data variables.
- Whenever the function calling statement was executed mapping between formal and actual parameters can be taken place as follows

```
Int x= a;
Int y=b;
```

- As the result of mapping new data variables can be created in the form of formal parameters which are local to called function and then values of actual parameters can be copied to them.

- Here two copies of input data can be maintained. One is in the form of actual parameters which is local to calling function and the Second is in the form of formal parameters which is local to called function.

- Due to the two individual copies of input data for calling and called functions if any modifications are made to the input data by the called function those can be reflected to it's local copy i.e to formal parameters but not to actual parameter copy and to calling function.

Call by Reference:

```
main()
  {
    int a=10, b=20;
    void swap( int *, int *);
    clrscr( );
     swap(&a, &b);
     printf(" %d %d \n",a,b);
    getch( );
}

void swap (int *x, int *y)
   {
 int temp;
       temp=*x;
        *x=*y;
        *y=temp;
}
```

- Here by passing the addresses of inputs function can be called, Hence it can be called as call by address or pointer or reference.

- Here actual parameters can be either addresses or pointers. Whereas formal parameters must be the pointers variables.

- Whenever the function calling statement was executed mapping between formal and actual parameters can be taken place as follows

```
Int  *x= &a;
Int  *y=&b;
```

- As the result of mapping new pointer variables can be created in the form of formal parameters which are local to called function and then addresses of inputs i.e actual parameters can be copied to them.

- Here one copy of input data can be maintained and can be accessed by both calling and called functions. Calling function can access directly whereas called function can access the same copy with pointers with de reference mechanism.

- Due to one copy of input data for calling and called functions if any modifications are made to the input data by the called function can be reflected to that copy and can be reflected to both the functions.

1D Arrays – Functions

In certain cases there is need to paas 1D arrays as parameters to functions and also there is a need to return an 1D Array as result from the body of the function , these can be as follows...

- To pass an 1D Array as parameter to function we can pass the Array Name and Array size as superate parameters. Their data types can be as Follows

  ```
  Datatype ArrayName[ ];
  Int  Size;
  ```
- If any modifications were made to the contents of the passed Array with in the called function then those can be automatically reflected to the calling function and also to the original Array.
- It is not possible to return an Array as Result from the body of the function with return statement directly.
- To return an Array as result from the body of the function, first we can pass the empty Array as parameter to the function and then calculate and store the result in that Array which can be automatically get reflected.

2D Arrays – Functions

In certain cases there is need to paas 2D arrays as parameters to functions and also there is a need to return 2D Array as result from the body of the function , these can be as follows...

- To pass a 2D Array as parameter to function we can pass the Array Name, Used no of rows and Used no of Columns as superate parameters. Their data types can be as Follows

  ```
  Datatype ArrayName[    ][size2];
  Int  rows;
  Int columns;
  ```
- If any modifications were made to the contents of the passed 2DArray with in the called function then those can be automatically reflected to the calling function and also to the original Array.
- It is not possible to return a 2DArray as Result from the body of the function with return statement directly.
- To return a 2DArray as result from the body of the function, first we can pass the empty Array as parameter to the function and then calculate and store the result in that Array which can be automatically get reflected.

Pointers – Arrays

Consider the following declaration

    Int a[10];

With this 20 Bytes of memory can be allocated from the continuous locations and can be distributed to 10 locations as follows.....

Whenever the Array was created after allocating the memory the starting address can be stored in the Array Name.
Always Array Name contains the starting address of the Array, Hence Array Name can be considered as pointer.
Array name can be considered as pointer constant but not pointer variable because always it contains the starting address of the array, not possible to change it.

```
Here
        a      = 1200 i.e starting Address
        a+0    = 1200  = &a[0]  i.e Address of a[0]
        a+1    = 1202  = &a[1]  i.e Address of a[1]
        a+2    = 1204  = &a[2]  i.e Address of a[2]
Similarly
        a+i    =  & a[i]  i.e Address of a[i]

Hence we can say that
        a+i    =  & a[i]  i.e Address of a[i]
      *(a+i)   =  a[i]      i.e Value of a[i]
```

We can access the values and the addresses of the locations of the Arrays in different ways as follows

```
for (  i=0 ;  i< n  ;  i++   )
 {
        A[i]     ---- >  value
        &a[i]    ---- >  address
 }
```

```
for (  i=0 ;  i< n  ;  i++   )
 {
       *(a+i)     ---- >  value
         a+i       ---- >  address
 }
```

```
 int *p;
for (  p=a ;  p< a+n  ;  p++   )
 {
       *p     ---- >  value
         p     ---- >  address
 }
```

Note:

- While passing arrays as parameters to the functions array names can be passed as parameters which contains the starting addresses of arrays.
- Arrays can be passed as parameters with call by reference only ,not possible with call by value.hence modifications can be reflected.
- While passing arrays as parameters their types can be specified as pointer as follows

  Datatype * arrayname;

Storage classes

- For each variable created there can be a memory source , Life, Scope and default initial value.
- Source of Memory is the part of the memory unit from which memory can be allocated.
- Life of the variable is the time period between the allocation of memory and de allocation of memory.
- Scope of the variable is the portion of the program where that variable is accessible.

By depending upon the source of memory, Life, Scope and default initial value we can classify the variables of C Language in to 4 Classes called as storage classes. Those are

1. Automatic or Local variables
2. Extern or Global variables.
3. Static Variables
4. Register Variables

Automatic or local Variables:

- These are the variables those can be created with in the bodies of the functions and blocks.
- To create these variables we can use the keyword auto.
- Syntax to declare the variable is as follows

  auto int a=10; Or int a=10;
- If no storage specifier was not given the default specifier is auto
- For these variables memory can be allocated from stack memory
- For these variables memory can be automatically allocated just before the execution of the function or block in which those were created and can be automatically de allocated at the end of the function or block execution. So life is from the beginning of the function execution to the end of function or block execution.
- These variables are accessible only the functions or blocks in which those were created. So scope is limited to the respective function or block in which those were created.
- If we dint initialize these variables unknown values can present in them.

Global or External Variables:

- These are the variables those can be created at out side of all the bodies of the functions and blocks.
- To create these variables we can use the keyword extern
- Syntax to declare the variable is as follows
- Extern int a=10; Or Int a=10;
- If no storage specifier was not given the default specifier is extern
- For these variables memory can be allocated from data memory

- For these variables memory can be allocated just before the execution of tha program and can be de allocated after the execution of the program. So lefe is from the beganing of the program execution to the end of program execution.
- These variables are accessible in all the functions or blocks of the program. So scope is to the entire program i.e to all the functions of the program.
- If we dint initialize these variables Zeros (null values)can present in them.

Note:

I. If we use the keyword while declaring the variable then its scope is to all the modules of the program irrespective of its place of declaration. If we dint use the extern keyword then its scope is from the place of declaration to the end of that file.

II. If we have a local variable with the same name of global variable then only local variable is accessible in that function or block. Not possible to access the global variable in that function or block.

Static Variables:

- While calling a single function for multiple times if we want to use a single variable that was created at the time of first calling for the remaining calls also then we can declare that local variable as static variable.
- These are the variables those can be created with in the bodies of the functions with the keyword static.
- To create these variables we can use the keyword static
- Syntax to declare the variable is as follows

  static int a=10;
- For these variables memory can be allocated from data memory
- For these variables memory can be allocated just before the execution of tha function in which those were created when ever it was called for first time and can not be automatically de allocated at the end of the function or block execution. Can be de allocated at the end of main function execution. So life is from the beganing of the function when ever it was called for the first time to the end of program execution.
- These variables are accessible only the functions or blocks in which those were created. So scope is limited to the respective function or block in which those were created.
- If we dint initialize these variables zero or null values can present in them.

Register Variables:

- Accessing of data from the memory variables are time taking operations. To save the system time and to speedup the execution memory can be allocated from registers.
- Register memory is small amount of memory present at the place of the CPU. Processor can directly read and write the data to that memory very fastly.
- To save the system time the most frequently accessed local variables can be declared as register variables.

- These are the variables those can be created with in the bodies of the functions and blocks.
- To create these variables we can use the keyword register
- Syntax to declare the variable is as follows

  register int a=10;
- For these variables memory can be allocated from register memory
- Registers can be given to the functions as long as the system control exists in those. Whenever control was shifted to someother function registers can be deallocated from that and can be given to new function. Life is along with system control.
- These variables are accessible only the functions or blocks in which those were created. So scope is limited to the respective function or block in which those were created.
- If we dint initialize these variables unknown values can present in them.
- Register variable creation is just a request to the system but not order. If registers are available those can be allocated to variables otherwise memory can be allocated from stack for the respective variables.

STRINGS

A string is the collection of characters from the continues locations with common name terminated with null character.

Declaration of string :

The syntax to declare a string is as follows.

Char stringname [size];

Here string name can be any identifier
size is the no of locations.

Ex :

char a [20];

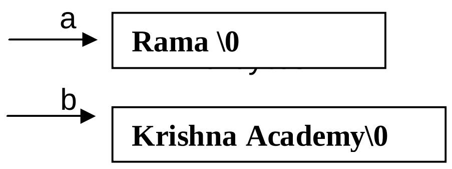
- Each string can be terminated with null character.
- For null character 1 byte of memory is needed.
- In the above string at most 19 characters can be stored.

Initialization :

A string can be initialized either with a set of character constants or with a string constant as follows.

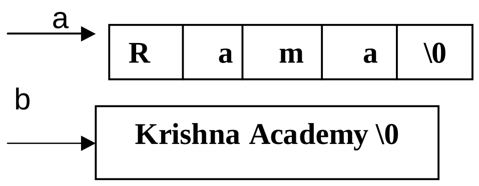
Char a[20]={'R','a','m','a'};

Char b[20]="Krishna Academy";



We can also initialize a string without specifying the size in the declaration and initialization statement.

char a[ ] = {'r','a','m','a',};
char b[ ] = "Krishna Academy";



- A string always can be considered either as the collection of characters or as a single entity.

- To accept a string constant as input to a string variable and to print the entire string as a single value in 'C' a conversion characters were given for printf( ) and scanf( ).
- % s can be used to read a string as input to a string variable with scanf.
- % s can be used with scanf to read single word string only.
- % s with printf can be used to print both single and multiword strings.

Ex:

```
        char a [ 20 ];
        scanf ("%s", a);
        printf ("%s", a);
```

- While reading the string and printing the contents of the string we can use characters also.

Ex:

```
        char a[ 20 ]
        To read 5 characters  to a string
                for (i=0; i<5; i++)
                    scanf("%c", &a[ I ]);
                a[ i ] = '\0';
        To print the contents of a string
                for (i=0; a[ i ] ! ='\0'; i++)
                        printf("%c", a[ i ]);
```

Char i/o and string i/o :

To read character, strings and to print characters and strings we have pre defined functions in stdio.h

(1) getchar()
(2) putchar()
} char i/o

(3) gets()
(4) puts()
} string i/o

getchar() :

- It can be used to read a character as input to a character variable.
- Only one character can be taken as i/p.

Syntax :

char var = getchar ();

Ex :

```
char ch;
ch=getchar();
```

It is equivalent to

```
scanf("%c", &ch);
```

- Scanf can read only printable character as i/p.
- getchar can be used to read printable or space character as i/o.

Putchar() :

It is the function which can be used to print the contents of a character.

Syntax :

```
putchar (character);
```

Here character can be either variable or constant.

Ex :

```
putchar('A');
o/p: A
```

It is equivalent to

```
printf ("%C", ch);
```

Ex:

```
char ch= 'B';
putchar (ch);
o/p:   B
```

gets() :

- It can be used to read a string as input to a string variable.
- It can be used to read either single word or multiword string as input.

Syntax :

```
gets (stringname);
```

Ex :

```
char a [ 20 ];
gets(a);
```

Puts() :

It is the function which can be used to print the contents of a string.

Syntax :

```
Puts (string);
```

Here string can be either variable or constant.

Ex :

```
char a [ 20 ] = "C language";
puts (a);
o/p:   C language
puts (" Krishna Academy");
o/p:   Krishna Academy.
```

- It can print the contents of both single word and multiword strings.
- While printing null character cannot be printed.

String functions:

To perform various operations on the strings and also to manipulate the strings pre defined functions are given to us in string.h file of the C standard library.
Some important functions of string.h are …….

strlen() function:

- This function counts and returns the number of characters in a string.
- The length does not include a null character.

Syntax:

n=strlen(string);

- Where n is integer variable. Which receives the value of length of the string.

Ex:

length=strlen("Krishna Academy");

here length is 15

/*writr a c program to find the length of the string using strlen() function*/

```
#include < stdio.h >
include < string.h >
void main()
{
char name[100];
int length;
printf("Enter the string");
gets(name);
length=strlen(name);
printf("\nNumber of characters in the string is=%d",length);
}
```

strcpy() function:

C does not allow you to assign the characters to a string directly as

name="Krishna";

Instead use the strcpy() function found in most compilers .

Syntax:

strcpy(dest,source);

- here dest must be string variable where as source can be either string variable or string constant.
- Strcpy function assigns the contents of source string to dest string.
- For proper coping the size of dest string must be at least greater by one than that of source string.
- During the copying process the existing contents of dest string can be overwritten with source string contents.
- After copying both dest and source can have the contents of source.

Ex:

```
char a[20],b[15];
strcpy(a,"krishna");
strcpy(b,a);
puts(a);          o/p: krishna
puts(b);          o/p: krishna
```

strcat() function:

when you combine two strings, you add the characters of one string to the end of other string. This process is called concatenation. The strcat() function joins 2 strings together.

Syntax:

strcat(dest,source);

- here dest must be string variable where as source can be either string variable or string constant.
- Strcpy function appends the contents of source string to dest string.
- For proper appendingthe size of dest string must be at least greater by one than the sum of the lengths of source and dest strings.
- If the dest string is empty then it just behaves like copy function.
- When the function strcat is executed source string is appended to dest string and source string remains unchanged.

Ex:

```
Char string1[20],string2[20];
strcpy(string1,"sri");
strcpy(string2,"Bhagavan");
strcat(string1,string2);
Printf("%s", string1);
```

From the above program segment the value of string1 becomes sribhagavan. The string at str2 remains unchanged as bhagawan.

strrev() function:
This function reverses the characters in a string.

Syntax:

strrev(string);

- here string must be the string variable

Ex:

```
Char a[]= "abcdefghi";
strrev(a);
puts(a);
o/p: ihgfedcba
```

strcmp function:

In c you cannot directly compare the value of 2 strings in a condition like if(string1==string2)

Most libraries however contain the strcmp() function, which returns a zero if 2 strings are equal, or a non zero number if the strings are not the same.

Syntax:

d=strcmp(string1,string2)

```
if
  d=0  then string1 = string2
  d>0  then string1 > string2
  d<0  then string1 < string2
```

- String1 & string2 may be string variables or string constants independently.
- Here character wise comparision can be takenplace. Whenever the first non matching characters were found then their ASCII difference can be returned.

strcmpi() function:
This function is same as strcmp() which compares 2 strings but not case sensitive.

Ex: strcmpi("THE","the"); will return 0.

strlwr () function:

This function converts all characters in a string from uppercase to lowercase.

Syntax:

strlwr(string);

Ex:

strlwr(“EXFORSYS”) converts to Exforsys.

strupr() function:

This function converts all characters in a string from lower case to uppercase.

Syntax:

strupr(string);

For example strupr(“exforsys”) will convert the string to EXFORSYS.

```
/* Example program to use string functions*/
#include < stdio.h >
#include < string.h >
void main()
{
char s1[20],s2[20],s3[20];
int x,l1,l2,l3;
printf(“Enter the strings”);
scanf(“%s%s”,s1,s2);
x=strcmp(s1,s2);
if(x!=0)
{printf(“\nStrings are not equal\n”);
strcat(s1,s2);
}
else
printf(“\nStrings are equal”);
strcpy(s3,s1);
l1=strlen(s1);
l2=strlen(s2);
l3=strlen(s3);
printf(“\ns1=%s\t length=%d characters\n”,s1,l1);
printf(“\ns2=%s\t length=%d characters\n”,s2,l2);
printf(“\ns3=%s\t length=%d characters\n”,s3,l3);
}
```

User Defined Datatypes

User defined datatypes are the types those can be prepared and used by the user according to his own requirement with the required domain and required format.

In ‘C’ language we have 3 different user defined datatypes.

1. Structures
2. Unions
3. Enumerators

While working with the user defined datatype we need to go through 3 stages.

1. Defining the datatype
2. Deelaring or prototyping the type
3. Using the datatype

- ➢ System did not know the format and shape of the type. So first we need to define the type according to our requirement with the given syntaxes.

- ➢ Declaration can be used to make an intimation to the compiler about the creation and usage of the new datatype.

- ➢ Once the datatype was defined and declared then it can be used for any purpose like declaration of variables, arrays, pointers, to pass parameters, to return the result from the function ect.

- ➢ If the definition was given before its usage with in the program then no need of declaration.

Structures

It is a user defined datatype which can be used to prepare the record formatted types to represent the data of entitles.

To represent the attributes of a real word entity we can use structures.

Defining the Structures:

The syntax to define a structure is a follows

```
Struct <tagname>
{
    Datatype  var1;
    datatype2  var2;
};
```

Here

- ➢ Struct is the keyword
- ➢ Tagname can be any identifier which can be used to differentiate the structure from other structures.
- ➢ Var1, var2….. are attributes with required types.

Ex:-

Student	**Struct Student**
	{
rollno.	int rno;
name	char name [20];
total	int total;
avg	float avg;
grade	char grade;
	};

- ➢ Once the definition was given it can be compiled by the system.
- ➢ With the compilation system can gather the complete info about the structure type like name, attributes names, their types, order their individual memory requirements, total memory reruireement for the structure ect
- ➢ For the above type of structure there is a need of 29 bytes (2+20+2+4+1) of memory.
- ➢ For the structure definition no memory can be allocated only for the variables created with this type memory can be allocated.
- ➢ The structure can be defined either with in the body of another definition locally or globally at out side of all the function definitions.

- Global definition can have the global scope where as the local def can have the local scope

Declaration of a structure:

The syntax to give the declaration is as follows

Struct tagname;

Ex:- struct student;

- If also can be called as forward declaration.

Using the datatype:-

Once the structure was defined it can be used as datatype to declare the variables, arrays, pointers ect.

- Struct keyword along with tagname can be combinely used as datatype.

We can declare the variables as follows

Struct student s1, s2;

With this two structure variables can be created. For each varable 29 bytes of memory can be allocated from the continuous locations and can be internally distributed to all members according to their types and order.

r no	name	total	avr	gr
2	20	2	4	1

Similarly for S2 separate 29 bytes can be allocated and can be distributed to all members.

Initializing the structure:

A structure can be declared and initialized simultariously as follows

Struct student s1={1001, “Rama”, 180, 60.00,’B’};

r no	name	total	avr	gr
1001	Rama\0	180	60.00	B
2	20	2	4	1

By observing the above figure we can define the structure as follows.

“**A structure is the collection of different type (heterogeneous) data from the continuous memory locations with common name”**

Accessing the elements of structure:

We can access the attributes or data members of each structure varable individually as follows.

varable . attribute

S1. r no.	S2. r no.
S1. name	S2 name
S1. total	S2. total
S1. avg	S2. avg
S1.gr	S2. gr

Reading the elements of a structure:

We can read the elements to attributes of the structure variable independently with the standard input functions as follows.

```
scanf ("%d", & s1. r.no);
scanf ("%d", & s1. total);
scanf ("%d", & s1. avr);
gets (s1. Name);
                etc..
```

Printing the Contents of a Structure:

Individually we can print the contents of a structure varable with standard o/p functions as follows.

```
printf(" roll no=%d", s1.rno);
printf (" name=%s", s1.name);
printf (" total=%d", s1.total);
printf (" average=%f", s1.avg);
printf (" grade=%c", s1.gr);
```

We can define , declare and initialize the structure as follows.

```
struct student
{   Int r no.;
   char name[20];
   int tot;
   float avg;
   char gr;
}s1, s2={1001, "Rama", 180, 60.00, 'B'}, s3;
```

We can define the structure like this also.

```
struct
{   int r no.;
   char name [2];
   int tot;
   float avg;
    char gr;
} student;
```

Structures – Arrays:

We can create the arrays by using the structure type as follows.

    Struct student s[10];

With this 290 bytes of memory can be allocated from the continuous locations and can be distributed for 10 structures.

Each structure can be accessed as

```
                s[0]
       s[i]     s[1]
                s[2]
                     etc..
```

The attributes of the structure can be accessible as

```
       s[i]. rno
       s[i]. name
       s[i]. tot
       s[i]. avg
       s[i]. gr
```

Structures – Functions:

- Structures can be passed as parameters to the functions either by using pass by value or pass by pointer mechanism.
- While passing with value it can be passed with its name.
- A structure can be returned as result from the body of the function

Structures – Pointers:

We can have the pointers to the structure variables as follows

```
    Struct student s1;
    Struct student *p;
        P=& s1;
```

- Here s1 is the structure varable and p is pointer to that.
- We can access the members of structure varable either with its name or by using its pointer.

```
       s1. r.no          p ⟶ r.no
       s1. name          p ⟶ name
       s1. tot           p ⟶ tot
       s1. avr           p ⟶ avr
       s1. gr            p ⟶ gr
```

- The cap operator ($\longrightarrow$) can be used to access the members of structure varable with the pointer as

```
                  ptr ⟶   member
```

Unions

A union is the user defined data type which can be used to prepare the record formatted type with memory saving mechanism.

- The syntaxes to define the union, declaration of the union , declaration of union variables, accessing the members of the union are just like structures. The only difference is in their memory allocation.
- While allocating the memory to a structure variable to all the data members memory can be allocated separately, where as while allocating for union a common location can be given all the members of union.

properties	**Structures**	**Unions**

Defining	struct example
	{
	int a;
	Float b;
	Char ch;
	};
		Union example
		{
		Int a;
		float b;
		char ch;
		};

Mem requirement	the sum of the individual requirements of the members.	Memory requirement of largest member
Declaration	Struct Example;	Union Example;
Declaration of variables	Struct Example e;	Union Example e;
Accessing of Members	e.a e.b } all the accessible e.c	e.a e.b } any one is accessible e.c

Enumerations:

- Is a set of named integer constants that specify all the legal values a variable of that type can have.
- The keyword enum signals the start of an enumeration type.
  The syntax to define enumeration is
  enum enum-type-name { enumeration list };
  Ex: enum coin { penny, nickel, dime, quarter, half_dollar, dollar};

```
enum coin money;
money = dime;
if (money = = quarter)
printf( “Money is a quarter. \n”);
```

  - The key point to understand about an enumeration is that each of the symbols stands for an integer value.
  - As such, they may be used anywhere that an integer may be used.
  - Each symbol is given a value one greater than the symbol that precedes it. The value of the first enumeration symbol is 0. Therefore,
    printf(“%d %d”, penny, dime);
    displays 0 2 on the screen.
  - You can specify the value of one or more of the symbols by using an initializer.
  - Do this by following the symbol with an equal sign and an integer value.
  - For example, the following code assigns the value of 100 to quarter:
    enum coin { penny, nickel, dime, quarter=100, half_dollar, dollar};
    Now, the values of these symbols are:

penny	0	quarter	100
nickel	1	half_dollar	101
dime	2	dollar	102

  - One common but erroneous assumption about enumerations is that the symbols can be input and output directly. This is not the case.
  - For example, the following code fragment will not perform as desired:
    money = dollar;
    printf(“%s”, money);
    Dollar is simply a name for an integer; it is not a string.

Typedef Statements:

- Creates synonyms (aliases) for previously defined datatypes
- Used to create shorter type names
- Format: typedef type new-type;
  - Example: typedef struct Card * CardPtr;
  - defines a new type name CardPtr as a synonym for type struct Card
- typedef does not create a new datatype
  - Only creates an alias

FILES

- File is a collection of logically related information stored in secondary storage memory with common name.
- Examples:
  - An employee file with employee names, designation, salary etc.
  - A product file containing product name, make, batch, price etc.
  - A census file containing house number, names of the members, age, sex, employment status, children etc.
- Two types:
  - Sequential file: All records are arranged in a particular order
- Random Access file: Files are accessed at random

File Access:

- The simplicity of file input/output in C lies in the fact that it essentially treats a file as a stream of characters, and accordingly facilitates input/output in streams of characters.
- Functions are available for character-based input/output, as well as string-based input/output from/to files.
- Before a file can be read from, or written into, a file has to be opened using the library function fopen( )
- The function fopen( ) takes a file name as an argument, and interacts with the operating system for accessing the necessary file, and returns a pointer of type FILE to be used in subsequent input/output operations on that file.
- This pointer, called the file pointer, points to a structure that contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and whether errors, or end of file have occurred.
- This structure to which the file pointer point to, is of type FILE, defined in the header file <stdio.h>.

- The only declaration needed for a file pointer is exemplified by:

  FILE *fp;

  fp = fopen(“file name”, “mode”);
- Once the function fopen( ) returns a FILE type pointer stored in a pointer of type FILE, this pointer becomes the medium through which all subsequent I/O can be performed.

File Access Modes:

- When opening a file using fopen( ), one also needs to mention the mode in which the file is to be operated on. C allows a number of modes in which a file can be opened.

Mode	Access	Explanation
“r”	Read only mode	Opening a file in “r” mode only allows data to be read from the file
“w”	Write only mode	The “w” mode creates an empty file for output. If a file by the name already exists, it is deleted. Data can only be written to the file, and not read from it
“a”	Append mode	Allows data to be appended to the end of the file, without erasing the existing contents. It is therefore useful for adding data to existing data files.
“r+”	Read+ Write mode	This mode allows data to be updated in a file
“w+”	Write + Read mode	This mode works similarly to the “w” mode, except that it allows data to be read back after it is written.
“a+”	Read+Append mode	This mode allows existing data to be read, and new data to be added to the end of the file.

Character-based File I/O:

- In C, character-based input/output from and to files is facilitated through the functions fgetc( ) and fputc( ).

- These functions are simple extensions of the corresponding functions for input/output from/to the terminal.

- The only additional argument for both functions is the appropriate file pointer, through which these functions perform input/output from/to these files.

A File Copy Program

```
#include<stdio.h>
main( )
{
  FILE *fp1, *fp2;
  fp1 = fopen( “source.dat”, “r”);
  fp2 = fopen( “target.dat”, “w”);
  char ch;
  while ( (ch = fgetc( fp1)) != EOF)
   {
     fputc (ch, fp2);
   }
fclose(fp1);
fclose(fp2);
}
```

The exit( ) function:

- ➢ The exit( ) function is generally used in conjunction with checking the return value of the fopen( ) statement.
- ➢ If fopen( ) returns a NULL, a corresponding error message can be printed, and program execution can be terminated gracefully using the exit( ) function.

```
if ((fp = fopen(“a.dat”, “r”)) = = NULL)
{
  printf(“Error Message”);
  exit( );
         }
```

Line Input/Output With Files:

- ➢ C provides the functions fgets( ) and fputs( ) for performing line input/output from/to files.
- ➢ The prototype declaration for fgets( ) is given below:

  char* fgets(char *line, int maxline, FILE *fp);
- ➢ The explanations to the parameters of fgets( ) is:
  - ❖ char* line – the string into which data from the file is to be read
  - ❖ int maxline – the maximum length of the line in the file from which data is being read
  - ❖ FILE *fp – is the file pointer to the file from which data is being read
- ➢ fgets( ) reads the next input line (including the newline) from file fp into the character array line;
- ➢ At most maxline-1 characters will be read. The resulting line is terminated with '\0'.

- Normally fgets( ) returns line; on end of file, or error it returns NULL.
- For output, the function fputs( ) writes a string (which need not contain a newline) to a file:
    - int fputs(char *line, FILE *fp)
    - It returns EOF if an error occurs, and non-negative otherwise.

File Copy Program Using Line I/O:

```
#define MAX 81;
#include<stdio.h>
main( )
{
  FILE *fp1, *fp2;
  fp1 = fopen( “source.dat”, “r”);
  fp2 = fopen( “target.dat”, “w”);
  char string[MAX];
  while ( (fgets(string, MAX, fp1)) != NULL)
  {
     fputs (string, fp2);
   }
fclose(fp1);
fclose(fp2);
}
```

Note: In the fgets( ) statement, the number of characters to be read has been specified by the macro MAX, which is 81. This is based on the assumption that the maximum line length in the file is 80 (i.e., one more than the maximum line length in the file). This is because fgets( ) reads one character less than the number specified, and adds a ‘\0’ at the end of the string. If however, a newline character is encountered before the specified number of characters, fgets( ) puts the ‘\0’ there. fgets( ) returns a NULL if an attempt is made to read beyond the last character in the file, i.e., end of file.

Formatted File Input/Output:

- C facilitates data to be stored in a file in a format of your choice. You can read and write data from/to a file in a formatted manner using fscanf( ) and fprintf( ).
- Apart from receiving as the first argument the format specification (which governs the way data is read/written to/from a file), these functions also need the file pointer as a second argument.
- fscanf( ) returns the value EOF upon encountering end-of-file.

- fscanf( ) assumes the field separator to be any white space character, i.e., a space, a tab, or a newline character.
- The statement printf("The test value is %d", i); can be rewritten using the function fprintf( ) as:
    - fprintf(stdout, "The test value is %d", x);
    - The statement scanf("%6s%d, string, &i) can be rewritten using the function fscanf( ) as:
    - fscanf(stdin, "%6s%d", string, &i);

Random Access:

- Input from, or output to a file is effective relative to a position in the file known as the current position in the file.
- For example, when a file is opened for input, the current position in the file from which input takes place is the beginning of the file.
- If, after opening the file, the first input operation results in ten bytes being read from the file, the current position in the file from which the next input operation will take place is from the eleventh byte position.
- It is therefore clear that input or output from a file results in a shift in the current position in the file.
- The current position in a file is the next byte position from where data will be read from in an input operation, or written to in an output operation.
- The current position advances by the number of bytes read or written.
- A current position beyond the last byte in the file indicates end of file.
- For example, when a file is opened for input, the current position in the file from which input takes place is the beginning of the file.
- If, after opening the file, the first input operation results in ten bytes being read from the file, the current position in the file from which the next input operation will take place is from the eleventh byte position.
- This is sequential access, in which a file is opened, and you start reading bytes from the file sequentially till end of file. The same argument cab be extended to sequential write.
- In sharp contrast to sequential access is random access that involves reading from any arbitrary position in the file, or writing to any arbitrary position in the file.

- Random access therefore mandates that we must have a mechanism for positioning the current position in the file to any arbitrary position in the file for performing input or output.
- To facilitate this, C provides the fseek( ) function, the prototype of which is as follows:

The fseek( ) Function:

- The function fseek( ) is used for repositioning the current position in a file opened by the function fopen( ).
- int rtn = fseek(file pointer, offset, from where)
- where,
- int rtn is the value returned by the function fseek( ). fseek( ) returns the value 0 if successful, and 1 if unsuccessful.
- FILE file-pointer is the pointer to the file
- long offset is the number of bytes that the current position will shift on a file
- int from-where can have one of three values:
    - from beginning of file (represented as 0)
    - from current position (represented as 1)
    - from end of file (represented as 2)

The rewind( ) Function:

- The function, rewind( ) is used to reposition the current position in the file (wherever it may be) to the beginning of the file.
- The syntax of the function rewind( ) is:
    - rewind (file-pointer);
    - where file-pointer is the FILE type pointer returned by the function fopen( ).
- After invoking rewind( ), the current position in the file is always the first byte position, i.e., at the beginning of the file.

The ftell( ) and feof( ) Function:

- The prototype declaration for the function ftell( ) is:
    - long ftell(FILE *fp)
- ftell returns the current file position for stream, or -1 on error.
- The prototype declaration for the function feof( ) is:
    - int feof(FILE *fp)
- feof returns non-zero if the end of file indicator for stream is set.

Priorities of Operators:

Description	Operator	Associates from the	Precedence
Function expr	( )	left to right	High (Evaluted first)
Array expr	[ ]	left to right	
struct indirection	->	left to right	
struct member	.	left to right	
Incr/decr	++ --	right to left	
One's complement	~	right to left	
Unary not	!	right to left	
Address	&	right to left	
Dereference	*	right to left	
Cast	(type)	right to left	
Unary plus	+	right to left	
Unary minus	-	right to left	
Size in bytes	sizeof	right to left	
Multiplication	*	left to right	
Division	/	left to right	
Modulus	%	left to right	
Addition	+	left to right	
Subtraction	-	left to right	
Shift left	< <	left to right	
Shift right	> >	left to right	
Less than	<	left to right	
Less than or equal	< =	left to right	
Greater then	>	left to right	
Greater than or equal	> =	left to right	
Equal	= =	left to right	
Not equal	! =	left to right	
Bitwise and	&	left to right	
Bitwise exclusive or	^	left to right	
Bitwise inclusive or	|	left to right	
Logical and	& &	left to right	
Logical or	| |	left to right	
Conditional	? :	right to left	
Assignment	= %= += -= *= /=	right to left	
	>>= <<= &= ^= |=		(Evaluted last) Low
Comma	,	left to right	