

---

## **C-LANGUAGE**

### **Features of C-Language:**

C-Language provides following Features:

- General Purpose Programming Language
- Middle Level Programming Language
- Modularity (Or) Procedure-Oriented Programming [POP] Language
- Portability
- Extendibility

### **General Purpose Programming Language:**

Using C-Language, we can develop many kinds of applications such as Business, Scientific, Mathematical and Graphical Applications. That is why C-Language is called General Purpose Programming Language.

### **Middle Level Programming Language:**

C-Language has both High-Level Language Features and Low-Level Language Features. That is why C-Language is called “Middle Level Programming Language”. High Level Languages are suitable to develop the Application Software. Low Level Languages are suitable to develop the System Software. Using C-Language, we can develop System Software and Application Software. “UNIX” Operating System, Many Programming Languages “Compilers & Interpreters” [System Software s] developed in C-Language. Google Chrome, Photoshop, Oracle & MS Office [Application Software s] developed in C-Language.

### **Modularity (Or) Procedure-Oriented Programming Language [POP Language]:**

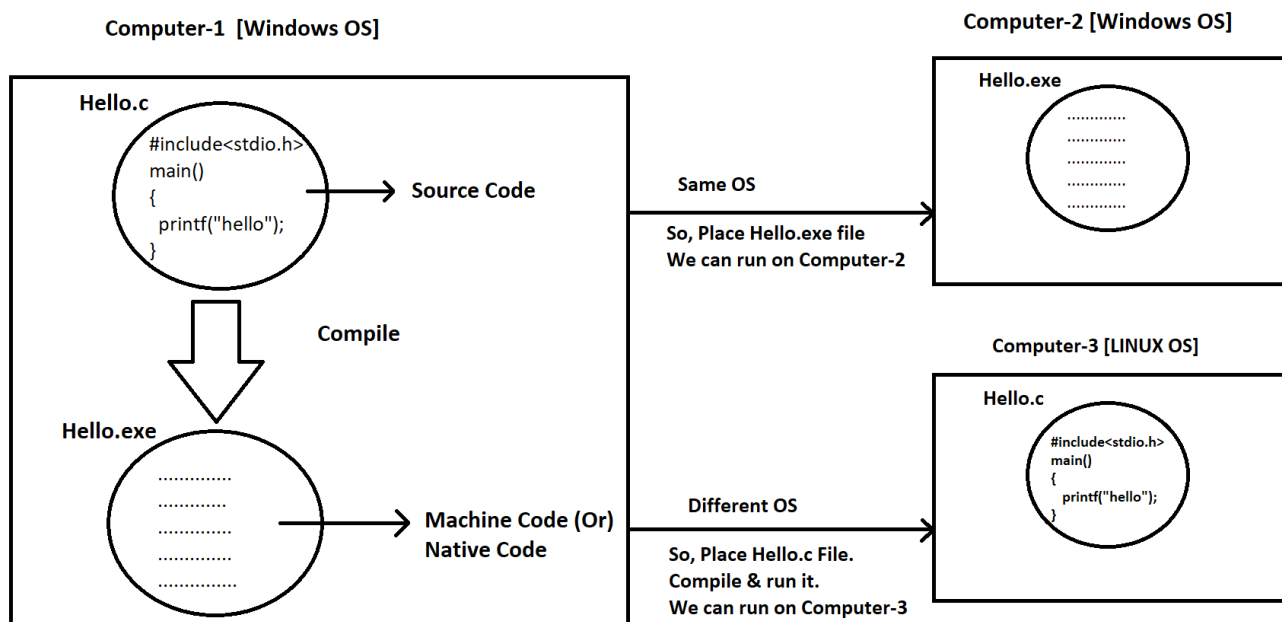
Modular Programming Approach (Or) Procedure -Oriented Programming Approach is a programming style in which we write a program in the form of Functions. In Some Languages we call them as modules (or) Procedures (or) Sub Routines. We can also call them as Sub Programs. C-Program is written in the form of Functions. That is why C-Language is called “Procedure-Oriented Programming Language” (Or) “Modular Programming Language”.

**Advantages with Modularity are:**

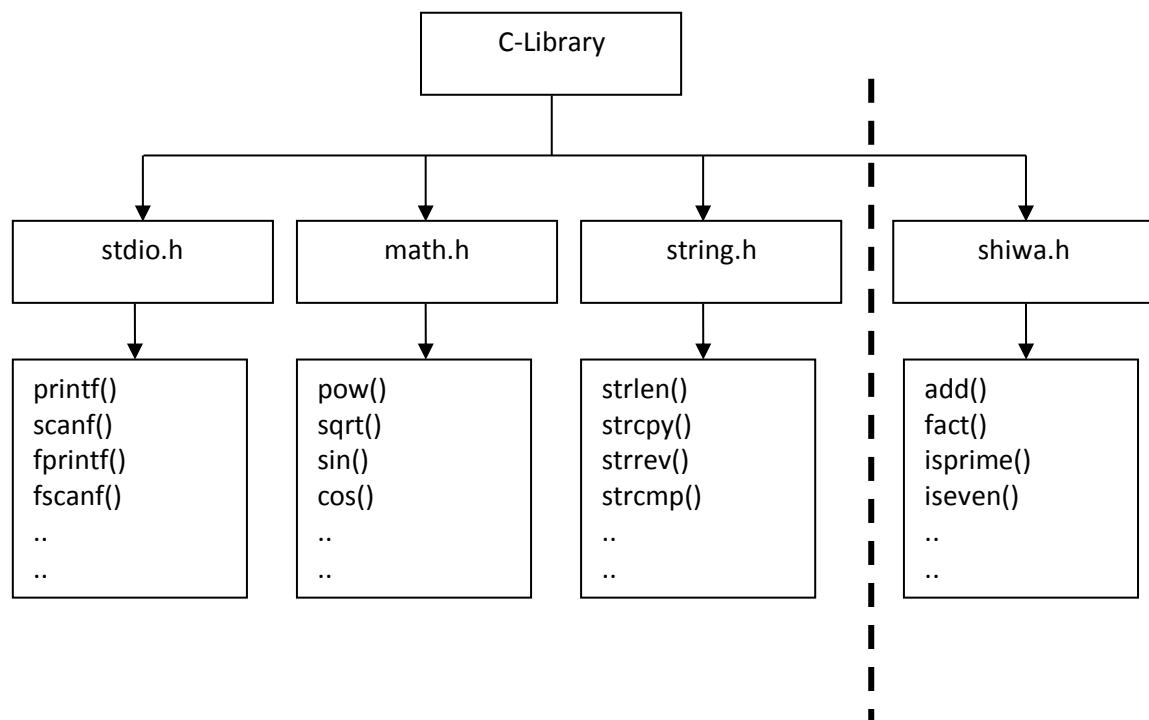
- It improves understandability.
- It allows reusability.
- It decreases length of code.

**Portability:**

The Application developed in one System using C-Language can run on any other System even if Operating System is different. If Operating System is same in another system, then place .exe file. If Operating System is different, then place Source Code File ( .c file ).

**Extendibility:**

We can extend the C-Library by adding our own header files. C-Language provides C-Library to us to develop the C-Programs. C-Library is a collection of Header Files where Header File is a collection of predefined functions.



## Extendibility

### Structure of a C-Program:

C-Language is a Procedure-Oriented Programming Language. C-Program is written in the form of functions.

Pre-processor Directives

Global Declaration

Function Declaration

<return\_type> <function\_name>([<arguments\_list>])

{

    //Local Declaration

    //Statements

}

·

·

int/void main([ int argc,char \*argv[ ] ])

{

    //Local Declaration

    //Statements

}

#### Note:

#### Notation (Symbol) & Meaning in Syntaxes:

Notation (Symbol)	Meaning
< >	Any
[ ]	Optional

**main() Function:**

- “main()” is a user-defined function in which we can write a set of statements.
- It is entry point of the program.
- Every C-Program execution starts from “main” function. Because it is entry point of the program.
- Its return type is “int / void” in C-Language.
- It can take two arguments. First argument is “int” type. Second argument is “char\*” type array. First argument takes number of arguments. Second argument takes a set of strings. Passing arguments to main function is optional. We can pass arguments to main function from MS DOS command prompt. This mechanism is called “Command Line Arguments”.
- Writing arguments for “main” function is optional.

**printf():**

- printf() is a predefined function or built-in function included in “stdio.h” header file.
- It is used to print the data on console screen [Output Screen].

**Example-1:**

```
printf("hello"); //prints hello
```

**Example-2:**

```
int x=20;  
printf("x=%d",x); //prints x=20
```

**Program to print “hello” on Console Screen:**

```
#include<stdio.h>  
main()  
{  
    printf("hello");  
}
```

**Output:**

```
Hello
```

**Escape Sequences (Or) Backslash Characters:**

- Escape sequence is a character constant.
- It has a \ and a following meaningful character.
- It is used to print the non-printing characters such as double quote ("), single quote (')...etc.
- It is used in printing (output) statements like printf().

Escape Sequence	Meaning
\n	New Line
\t	Tab
\b	Back space
\a	Alert Beep
\"	"
\'	'
\\	\

**Program to demonstrate Escape Sequences:**

```
#include<stdio.h>
int main()
{
    printf(" hello\n");
    printf(" welcome to \t C-Language");
    printf("\n \"hello\" ");
    printf("\n 'hi' ");
    printf("\a\n abcd\befg");
}
```

**Output:**

```
hello
welcome to    C-Language
"hello"
'hi'
abcefg
```

## Variable:

- Variable is a name of storage location that can hold a value.
- It can be changed.
- A variable can hold one value only at a time.

**Example:** `int rno=15;` //rno is a variable

rno

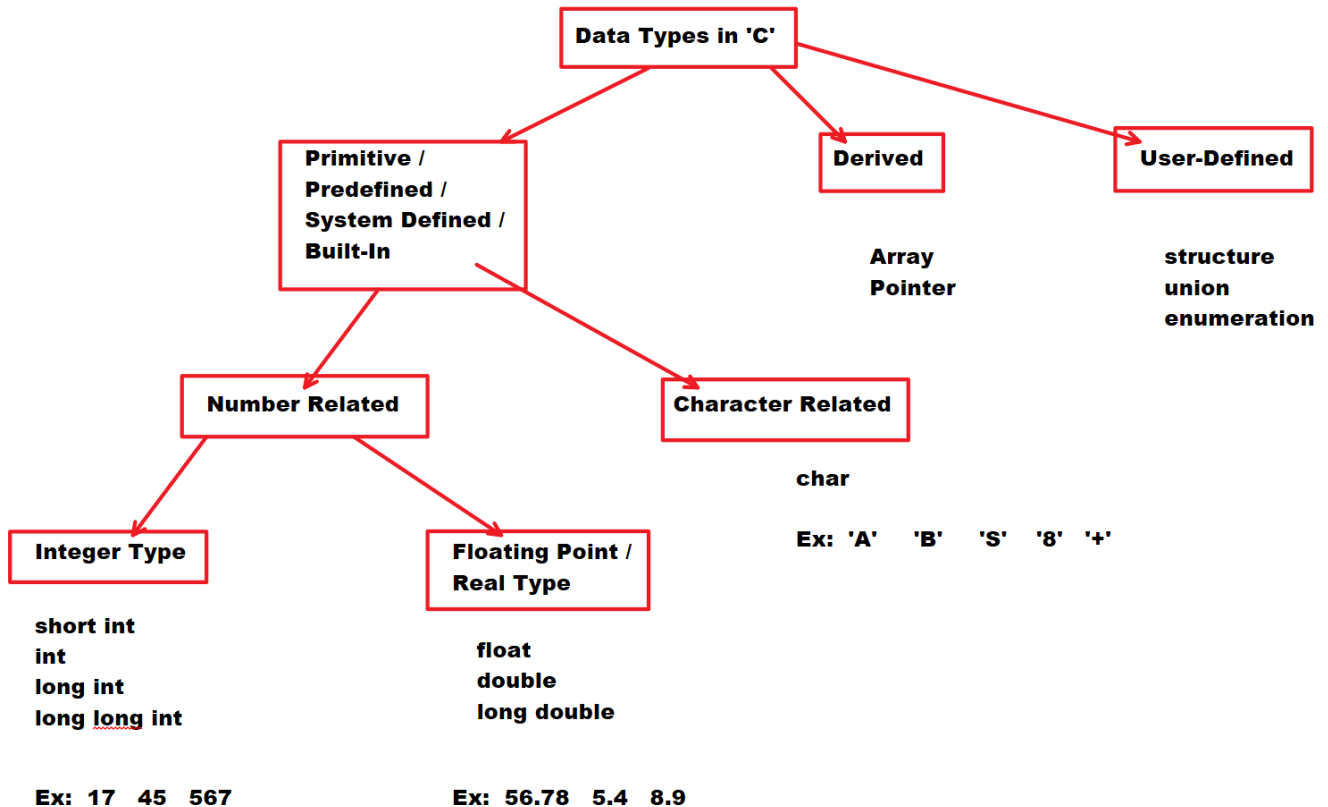
15

## Data Types in C-Language:

- Data Type tells the type of data which a variable can hold.
- It also tells how much memory should be allocated for a variable.

In C-Language, Data Types can be divided into 3 Types. They are:

- Primitive [Built-In / System-Defined / Predefined] Data Types
- Derived Data Types
- User-Defined Data Types



**Note:****Integer Type:**

Number without decimal places is called "Integer Type".

**Example:** 65 879 -16 -12

**Floating Point [Real]:**

Number with decimal places is called "Floating Point Type (or) Real Type".

**Example:** 123.4567 56.78 -34.56

Primitive Type	Memory [Size]
short int	2 Bytes
int	4 Bytes
long int	4 Bytes
long long int	8 Bytes
float	4 Bytes
double	8 Bytes
long double	16 Bytes
char	1 Byte

**Integer Type:**

Number without decimal places is called "Integer Type".

**Example:** 65 879 -16 -12

There are Four Integer related Data Types in C-Language. They are:

1. short int
2. int
3. long int
4. long long int

There are Two types in every Integer related Data Type. They are:

- i. signed
  - **signed:**  
It can accept positive, zero and negative values.
- ii. unsigned
  - **unsigned:**  
It can accept positive values and zero only. It cannot accept negative values.

Integer Type	Memory	Range	Format Specifier
short int	2 Bytes	32768 to 32767 (or) $-2^{15}$ to $2^{15}-1$	%hd (or) %hi
unsigned short int	2 Bytes	0 to 65535 (or) $0$ to $2^{16}-1$	%hu
int	4 Bytes	$-2^{31}$ to $2^{31}-1$	%d (or) %i
unsigned int	4 Bytes	$0$ to $2^{32}-1$	%u
long int	4 Bytes	$-2^{31}$ to $2^{31}-1$	%ld (or) %li
unsigned long int	4 Bytes	$0$ to $2^{32}-1$	%lu
long long int	8 Bytes	$-2^{63}$ to $2^{63}-1$	%lld (or) %lli
unsigned long long int	8 Bytes	$0$ to $2^{64}-1$	%llu

### Floating Point Type:

Number with decimal places is called “Floating Point Type (or) Real Type”.

**Example:** 123.4567    56.78    -34.56

There are Three Floating point related data types in C-Language. They are:

1. float
2. double
3. long double

Float Type	Memory	Precision [Decimal Places]	Format Specifier
float	4 Bytes	6	%f
double	8 Bytes	15	%lf
long double	16 Bytes	18	%Lf



**Format Specifiers:**

- Format specifiers are used to read the data or print the data.
- These are used in reading (input) and printing (output) statements.

FORMAT SPECIFIER	PURPOSE [To Read or Print]
%hd (or) %hi	short int
%hu	unsigned short int
%d (or) %i	Int
%u	unsigned int
%ld (or) %li	long int
%lu	unsigned long int
%lld (or) %lli	long long int
%llu	unsigned long long int
%f	float
%lf	double
%Lf	long double
%c	char
%s	string
%o	octal
%x (or) %X	hexa decimal
%e (or) %E	scientific

**Declaring Variables and Assigning Values:****Declaring Variable:****Syntax:**

<data\_type> <v1>[,<v2>,...,<vn>];

**Declaring Integer Type Variable:**

```
int rno; // Allocates 4 Bytes memory for rno
```

```
int m1,m2,m3;// Allocates 4 Bytes Memory for each variable m1,m2,m3
```

**Declaring Floating Point Type Variable:**

```
float avrg; // Allocates 4 Bytes memory for avrg
```

**Declaring Character Type Variable:**

```
char section; //Allocates 1 Byte memory for section
```

**Assigning Values:****Syntax:****<variable>=<constant> / <variable>;****Assigning Integer Value:**

```
rno=50;
```

```
y=x;
```

**Assigning Float Value:**

```
avrg=56.78;
```

**Assigning Charcater:**

```
section='A';
```

**Reading Data from Keyboard:****scanf():**

- “scanf()” is a predefined function included in “stdio.h” header file.
- It is used to read the data from keyboard.

**Example-1:****Reading an Integer value from keyboard:**

```
int rno;  
printf("Enter rno:");  
scanf("%d",&rno);
```

**Example-2:****Reading a float value from keyboard:**

```
float radius;  
printf("Enter radius of circle:");  
scanf("%f",&radius);
```

## C-Language Tokens:

A smallest individual unit of a C-program is called “C-Token”.

C-Language Tokens are:

- Identifiers
- Keywords
- Constants [Literals]
- Operators
- Separators

### Identifiers:

- The names of variables, functions, labels or any user-defined name in the program is called “Identifier.”
- It is used for identification purpose.

### Rules for naming an Identifier:

- An Identifier is made up of with Letters [A to Z, a to z], Digits [0 to 9], Underscore [ \_ ] and Dollar [ \$ ]. Other characters like space, @, # cannot be used in Identifier.

#### Example:

```
int m1; //valid
```

```
int total_mark$; //valid => _ and $ can be used
```

```
int sub1_mark$; //valid => 1, _ and $ can be used
```

```
int total marks; //Invalid => cannot contain space
```

```
int m@rks; // Invalid => cannot contain @
```

- It should not be started with digit.

#### Example:

```
int subject1_marks; //valid
```

```
int 1subject_marks; //Invalid => cannot be started with digit 1.
```

- It can be started with Letter or Underscore or Dollar.

#### Example:

```
int _x; //valid => can be started with _
```

```
int $x; //valid => can be started with $
```

- It can be of any length.

**Example:**

```
int abcdefghijklmnopqrstuvwxyz=20; //valid
```

- It is case sensitive. It means, Upper Case and Lower Case letters are treated as different.

**Example:**

```
int x=20,X=30; // 2 different variables x and X  
int rno, RNO, Rno, rNO; // 4 different variables
```

- Keywords cannot be used as an Identifier.

**Example:**

```
int if;    //Invalid  
int char; //Invalid  
int for;   //Invalid  
int IF;    // Valid => 'if' is a keyword. 'IF' is not a keyword.
```

**Keywords:**

- Keywords are the Reserved Words.
- Every keyword has specific meaning and that can perform specific functionality.
- We cannot use keywords as Identifiers.
- C-Language initially introduced with 32 keywords only. "C-17" version has 44 keywords.

**Example:**

```
if for while do switch else int float char
```

**Constants [Literals]:**

- Constants are the fixed values.
- We cannot change them.

**Example:****Integer Constants:**

```
17 459 -45 786 -68 => Decimal Constants
```

```
0b1110 => Binary constant [Prefix binary value with '0b']
```

```
036 => Octal Constant [Prefix octal value with '0']
```

```
0x1E => Hexa Decimal Constant [Prefix hexa decimal value with 0x]
```

123.4567 -45.67 67.893 => Floating Point Constants

'A' '+' '4' 'B' 'S' => Character Constants

"raju" "++" "1992" "Hyd" => String Constants

**Operators:**

- Operator is a symbol which is used to perform operations like Arithmetic Operations or Logical operations.

**Example:**

+ - \* / % > >= < <=

**Separators:**

- Separators are:  
variable separator ,  
statement separator ;  
block separator { } ...etc

## OPERATORS

- **“Operator”** is a symbol that is used to perform Arithmetic Operations or Logical Operations.
- **Arithmetic Operations:**  $a+b$   $a-b$   $a*b$   $a/b$
- **Logical Operations:**  $a>b$   $age<18$   $age\geq 18$   $m<35$   $m\geq 35$
- The variables that are participating in the operations are called **“Operands”**.
- Combination of Operands, Operators and Numbers is called **“Expression”**.

**Example:**  $2*pi*r$ ,  $pi*r*r$ ,  $b*b-4*a*c$ ,  $n\%2$ ,  $x+y*z$

In  $2*pi*r \Rightarrow$  2 is Number. Pi and r are Operands. \* is Operator.

C- Language provides following Operators:

Type	Operators	Purpose
Arithmetic Operators	+ [Addition Operator] - [Subtraction] * [Multiplication] / [Division] % [Modulus]	Used to perform Arithmetic operations
Relational(or) Comparison Operators	> [Greater Than Operator] >= [Greater Than or Equal To] < [Less Than] <= [Less Than or Equal To] == [Equal To] != [Not Equal To]	Used to compare two values
Logical Operators	&& [And]    [Or] ! [Not]	Used to perform logical operations

Assignment Operators	= [Assignment Operator] += [Addition Assignment] -= [Subtraction Assignment] *= [Multiplication Assignment] /= [Division Assignment] %= [Modulus Assignment]	Used to assign the values to a variable.
Bitwise Operators	& [Bitwise AND Operator]   [Bitwise OR] ^ [Bitwise XOR] ~ [Bitwise Complement] << [Bitwise Left Shift] >> [Bitwise Right Shift]	Used to work with the bits ( 0s and 1s)
Other Operators	++ [Increment Operator]	Used to increase one value of the variable
	-- [Decrement Operator]	Used to decrease one value of the variable
	- [Unary Minus]	Used to convert positive value to negative value or negative value to positive value.
	?: [Ternary/Conditional Operator]	Used to execute the expressions based on the condition
	sizeof [Size Of Operator]	Used to know the size of data type or variable
	, [Comma Operator]	Used to separate multiple declarations, assignments and expressions.
	& [Address Of Operator]	Used to know the address of a variable.

## Operator Precedence & Associativity:

Precedence	Type	Operators	Associativity
1	Postfix	() [] -> . ++ --	Left to right
2	Unary	+ - ! ~ ++ -- (type)* & sizeof	Right to left
3	Multiplicative	* / %	Left to right
4	Additive	+ -	Left to right
5	Shift	<<, >>	Left to right
6	Relational	< <= > >=	Left to right
7	Equality	== !=	Left to right
8	Bitwise AND	&	Left to right
9	Bitwise XOR	^	Left to right
10	Bitwise OR		Left to right
11	Logical AND	&&	Left to right
12	Logical OR		Left to right
13	Conditional	?:	Right to left
14	Assignment	= += -= *= /= %= >>= <<= &= ^=  =	Right to left
15	Comma	,	Left to right

### Operator Precedence:

Operator Precedence is used to evaluate the Expressions. It determines the order of operations to be performed in the Expression.

**Example-1:**  $x + y * z$

In the above Expression, '\*' has highest priority than '+'. So  $y*z$  is calculated first. Then '+' operation will be done.

**Example-2:**  $(x+y)*z$

In the above expression,  $(x+y)$  will be calculated first. Because, '(' has highest priority than '\*'.

### Associativity:

Associativity is used when multiple operators have same level precedence. Associative can be either Left to Right (or) Right to Left.



**Example-1:**  $a+b*c-d/e$

In the above Expression, \* and / are same level priorities. So, Associativity will be used. For \* and /, Associativity is Left to Right. So, '\*' will be calculated first. Then / will be calculated.

+ and – are same level priorities. Associativity of + and – is Left to Right. So, + will be calculated first. Then – will be calculated.

**Example-2:**  $a=b=c=d=50$

In the above Example, 4 Assignment operators are used. For Assignment operators, Associativity is Right to Left.

First 50 will be assigned to d. d will be assigned to c. c will be assigned to b. Then b will be assigned to a.

---

## **Control Structures**

### **[Control Statements / Flow Control]**

- Control Structures are used to control the flow of execution of statements.
- Normally, C-Program gets executed sequentially. To change the sequential execution & to transfer the control to our desired location, we use control structures.

There are 2 types of statements:

1. **Sequential Statements:** These statements get executed sequentially one after another.
2. **Control Statements:** These statements get executed randomly or repeatedly.

#### **Advantage:**

- It provides better control to programmer on flow of execution.

C-Language provides following control structures:

Type	Control Structures
Conditional Control Structures	if if else if else if nested if
Multi way Conditional Control Structure	switch
Iterative [Looping] Control Structures	while do while for
Jumping Control Structures	goto break continue return

**Conditional Control Structures:**

Conditional control structure executes the statements based on conditions.

C-Language provides following Conditional Control Structures:

- if
- if else
- if else if
- nested if

**if:**

**Syntax:**

```
if(<condition>)  
{  
    //statements  
}
```

- The statements in if block get executed when the condition is true. It skips the statements when the condition is false.
- It is suitable when we want to perform a task based on condition.
- If one statement is in 'if' block, we have no need to specify curly braces ( { } ). We must specify curly braces if multiple statements are in 'if' block.

**Execution Process:**

First, it checks the condition. If condition is true, executes the statements. Otherwise, skips the statements.

**Example:**

```
if(age>=18)  
    printf("Eligible to Vote");
```

**if else:****Syntax:**

```
if(<condition>
{
    //statements
}
else
{
    //statements
}
```

- The statements in if block get executed when the condition is true.
- The statements in else block get executed when the condition is false.
- It is suitable when we want to perform any one of the two tasks.

**Execution Process:**

First, it checks the condition. If condition is true, executes 'if' block statements. Otherwise, it executes else block statements.

**Example:**

```
if(age>=18)
    printf("Eligible to Vote");
else
    printf("Not Eligible to Vote");
```

**if else if:****Syntax:**

```
if(<condition-1>)
{
    //statements
}
else if(<condition-2>)
{
    //statements
}
else if(<condition-3>)
{
    //statements
}
.
.
else
{
    //statements
}
```

- The statements in 'if else if' get executed when corresponding condition is true.
- It executes 'else' block statements when all conditions are false.
- It is suitable when we want to perform any one of the multiple tasks (more than 2 tasks).
- In this, writing 'else' block is optional.

**Execution Process:**

First, it checks the condition-1. If the condition is true, it executes the statements of 'if' block and comes out of 'if else if'. If condition-1 is 'false', then only it checks second condition. If second condition is true, it executes the statements in this 'else if' block and comes out of 'if else if'. If second condition is 'false', then it checks another

condition and so on. If all conditions are false, 'else' block statements get executed.

**Example:**

```
if(n>0)
    printf("Positive");
else if(n<0)
    printf("Negative");
else
    printf("Zero");
```

**nested If:****Syntax:**

```
if(<condition-1>)
{
    if(<condition-2>)
    {
        //statements
    }
}
```

- Writing if in another if is called 'nested if'.
- The statements in inner if get executed when outer condition and inner condition are true.
- We can write any number of 'if's, 'if else's, 'if else if's in 'if' block.
- In 'else' block also we can write any number of 'if's, 'if else's, 'if else if's.

**Execution Process:**

First, it checks outer condition. If outer condition is true, it checks inner condition. If it is also true, then statements get executed. If outer condition is false, it will not check inner condition.

**Example:**

```
if(a>b)
{
    if(a>c)
    {
        printf("a is big");
    }
}
```

**Multi way Conditional Control Structure:**

Multi way conditional control structure executes the statements based on value matching criteria.

**switch:****Syntax:**

```
switch(<variable>/<expression>)
{
    case <constant1>:
        //statements
        break;
    case <constant2>:
        //statements
        break;
    .
    .
    default:
        statements
}
```

- The statements in 'switch' get executed when variable/expression value is matched with the constant value.
- When all constant values are not matched, it executes 'default' statements.
- Writing 'default' is optional.
- If we don't write "break" as last statement in every case, next case statements get executed irrespective of constant matching criteria. That's why we must write 'break' as last statement in every 'case' to come out of "switch".
- Case Constant can be Integer only. Characters are also accepted. Because, for every character ASCII value will be taken internally. ASCII value is integer only.
- Case Constant cannot be floating point constant (or) string constant.

**Execution Process:**

First, it captures variable/expression value. It checks with first constant value. If first constant value is matched with variable/expression value, first case statements get executed and come out of switch. If first constant value is not matched, checks with second constant value. If it is matched, second case statements get executed and comes out of switch. If not matched, checks with third constant and so on. If all constant values are not matched with variable/expression value, it executes 'default' statements.

**Example:**

```
switch(n%2)
{
    case 0:
        printf("Even");
        break;
    case 1:
        printf("Odd");
}
```



**Rules in 'switch':**

- Constant values must be unique. They should not be duplicated.
- Constant data type and Expression data type should be same. Because it tests equality only.
- Constant or expression can be of integer type or char type.
- Constant or Expression cannot be of float type or string type.

**Iterative (or) Looping Control Structures:**

Looping Control Structures are used to execute the statements repeatedly. It requires three things to execute the statements repeatedly. They are: Initialization, Condition and Step.

C-Language provides following looping control structures:

- while
- do while
- for

**'while' loop:****Syntax:**

```
while(<condition>)  
{  
    //statements  
}
```

- The statements in 'while' loop get executed as long as the condition is true. When the condition is false, it terminates the loop.
- It first checks the condition. Then executes the statements based on condition.
- It is also called as "Entry Controlled Loop". Because, before entering in the loop it checks the condition.
- Minimum number of execution times for 'while' loop is 0. Because when the condition is false, it will not execute the statements even for single time.

**Execution Process:**

First, condition is tested. If condition is true, statements get executed. After executing all statements of 'while' loop, condition is tested again. If the condition is true, statements get executed again. It performs this process as long as the condition is true. It terminates the Loop when the condition is false.

**Example:**

```
i=1;
while(i<=10)
{
    printf("%d\n",i);
    i++;
}

// Above code prints 1 to 10
```

**'do while' loop:****Syntax:**

```
do
{
    //statements
} while(<condition>);
```

- The statements in 'do while' loop get executed as long as the condition is true except first time.
- First time directly statements get executed. From second time onwards, statements get executed based on condition. When the condition is false, it terminates the loop.
- It can be also called as "Exit-Controlled Loop". Because, at end of the loop, it checks the condition.

**Execution Process:**

First, it executes the statements. Then checks the condition. If condition is true, statements get executed again. This process is performed as long as the condition is true. When the condition is false it terminates the loop.

**Example:**

```
i=1;
do
{
    printf("%d\n",i);
    i++;
} while(i<=10);

//Above code prints 1 to 10
```

**‘for’ Loop:****Syntax:**

```
for(<Initialization>;<condition>;<step>)
{
    //statements
}
```

Initialization → Starting value  
Condition → Ending Value  
Step → Increment/Decrement

- The statements in ‘for’ loop get executed as long as the condition is true. When the condition is false, it terminates the loop.
- In for Loop, we can write Initialization, Condition & Step in single step.

**Execution Process:**

First, it takes initial value. Then it checks condition. If the condition is true, statements get executed. After executing all statements of 'for' loop, it will take step. After taking step, it checks with the condition again. If the condition is true, statements get executed again. This process is performed as long as the condition is true. It terminates the loop when the condition is false.

**Example:**

```
for(i=1; i<=10; i++)  
    printf("%d\n",i);  
  
// Above code prints 1 to 10
```

**Jumping Control Structures:**

C-Language provides following Jumping Control Structures:

- goto
- break
- continue
- return

**goto:**

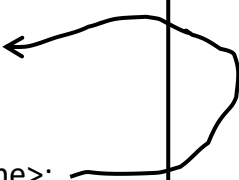
It is used to transfer the control to specified label.

It can be used in two ways. They are:

- Jumping Backward
- Jumping Forward

**Jumping Backward:****Syntax:**

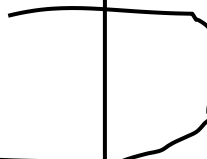
```
.....  
.....  
.....  
<Label_Name>: ←  
.....  
.....  
goto <Label_Name>;  
.....  
.....
```

A hand-drawn arrow originates from the right side of the 'goto <Label\_Name>;' statement and points back to the right side of the '<Label\_Name>:' label, indicating a backward jump.**Example:**

```
    i=1;  
    nareshit:  
        printf("%d\n",i);  
        i++;  
        if(i<=10)  
            goto nareshit;  
  
    //Above code prints 1 to 10
```

**Jumping Forward:****Syntax:**

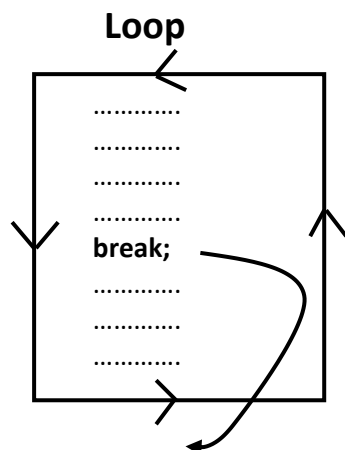
```
.....  
.....  
goto <Label_Name>;  
.....  
.....  
<Label_Name>: ←  
.....  
.....
```

A hand-drawn arrow originates from the right side of the 'goto <Label\_Name>;' statement and points forward to the right side of the '<Label\_Name>:' label, indicating a forward jump.

**Example:**

```
printf("Enter a +ve number:");  
scanf("%d",&n);  
  
if(n>0)  
    goto nareshit;  
  
n=-n;  
  
nareshit:  
    printf("%d",n);  
  
// Above code prints Absolute Value
```

- Jumping Backward is used to execute the statements repeatedly.
- Jumping Forward is used to skip the statements execution.

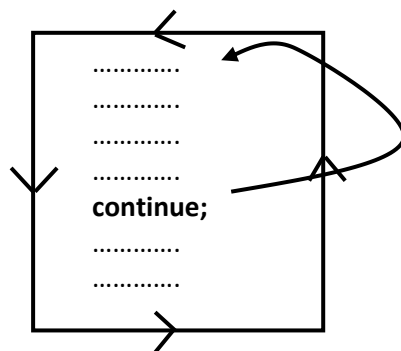
**break:****Syntax:**

- 'break' is used to terminate the corresponding loop in the middle of execution.
- It can be also used in 'switch' to come out of the 'switch'.
- It can be used in switch or loop only. It cannot be used directly.

**Example:**

```
for(i=1; i<=10; i++)
{
    printf("%d\n",i);
    if(i==5)
        break;
}

// Above code prints 1 to 5
```

**continue:****Syntax:**

- 'continue' is generally used to transfer the control to first statement of the corresponding loop.
- It is used to skip the iteration and continue the next iteration. It can be used in loop only.
- It cannot be used in 'switch'.

**Example:**

```
for(i=1; i<=10; i++)
{
    if(i==7)
        continue;
    printf("%d\n",i);
}

// Above code prints 1 to 10 except 7
```

**return:**

- “return” is a Jumping Control Structure.
- It can be used to come out of the function.
- It can return a value to the function call.

**Syntax:**

```
return <variable> / <constant>;
```

**Example:**

```
int add(int x,int y)
{
    return x+y;
}

// returns x+y to function call
```

**Nested Loops:**

Writing Loop in another Loop is called “Nested Loop”. A Loop is used to execute the statement repeatedly. A Nested Loop is used to execute the loop repeatedly.

**Example:**

```
for(i=1;i<=10;i++)
{
    for(j=1;j<=5;j++)
    {
        printf("hello\n");
    }
    printf("\n");
}
```



---

## Functions

- C-Language is a Procedure-Oriented Programming Language (or) Function-Oriented Programming Language.
- We write a C-Program in the form of Functions.

### **Function:**

- Function is a block of statements that gets executed on calling.
- It can be also called as a “Sub-Program”.
- Every Function is defined to perform specific task.

### **Advantages of Functions:**

- It improves understandability.
- It decreases length of code.
- It allows reusability of code.
- We can define any number of functions.
- We can call a function for any number of times.
- We can call a function from anywhere within the program. But that function must be called directly or indirectly from “main” function. Because “main” is entry point of the program.

### **Syntax to define a Function:**

```
<return_type> <function_name>(<argument_list>)\n{\n    //Statements\n}
```

### **Example:**

```
int add(int x,int y)\n{\n    return x+y;\n}
```

A Function contains 2 parts. They are:

- Function Header
- Function Body

**Function Header:**

It contains return type, function name and argument list.

**Example:**

```
int add(int x,int y)
```

**Function Body:**

It contains a set of statements which should be executed when it is called.

**Example:**

```
{  
    return x+y;  
}
```

**Types of Functions:**

There are two type of Functions. They are:

- Built-In Functions
- User-Defined Functions

**Built-In Functions:**

Built-In Functions can be also called as Predefined Functions (or) System-Defined Functions (or) Library Functions. These Functions were developed by C-Language developers. They placed related functions in a file. This file is called "Header File".

**Example:**

**math.h**

```
pow()  
sqrt()  
sin()  
cos()  
tan()
```

**string.h**

```
strlen()  
strcpy()  
strrev()  
strcmp()  
strcat()
```

**stdio.h**

```
printf()  
scanf()  
getc()  
putc()  
fprintf()
```

**User-Defined Functions:**

We can define our own functions in C-Language. These are called User-Defined Functions.

**Example:**

```
fact()      add()      nareshit()      add()
```

**What are required to make a Function useful?**

Three things are required to make a function useful. They are:

- Function Declaration (or) Function Prototype
- Function Definition
- Call of Function

**Function Declaration (or) Function Prototype:**

Declaring a function is called “Function Declaration” (or) “Function Prototype”. Like variable declaration we can declare a function in the program. It gives information to the compiler that program has a function with so and so name, return type and argument types. It ends with semicolon ( ; ).

**Example:**

```
int add(int,int);
```

**Function Definition:**

Defining a function is called “Function Definition”. It contains Function Header and Function Body.

**Example:**

```
add(int x,int y)
{
    return x+y;
}
```

**Call of Function:**

Calling a Function is called “Call of Function”. A function does not get executed without calling. So, we must call the function.

**Example:**

```
add(5,4);
```

**Program to demonstrate Functions:**

```
#include<stdio.h>
void nareshit();      //Function Declaration (or) Function Prototype
void nareshit()      // Function Definition
{
    printf("hello from nareshit\n");
    printf("Welcome to nareshit\n");
    printf("Bye from nareshit\n");
}
void main()
{
    printf("Hi from main\n");
    nareshit();        //call of function
    printf("Bye from main");
}
```

**Output:**

```
Hi from main
hello from nareshit
Welcome to nareshit
Bye from nareshit
Bye from main
```

In the above program,

“main()” is called “Calling Function” (or) “Caller Function”.

“nareshit()” is called “Called Function” (or) “Callee Function”.

**Calling Function:**

A function from where we are calling another function is called “Calling Function” (or) “Caller Function”.

**Called Function:**

A function which is called by another function is called “Called Function” (or) “Callee Function”.

**Argument [Parameter]:**

- An Argument can be also called as “Parameter”.
- Argument is a local variable declared in Function Header.
- It is used to bring a value into function.
- It acts like Input for a Function.
- We call it as local variable. Its scope is limited to function only. It cannot be used outside of the function.

**Example:**

```
int add(int x,int y) => x,y are called arguments
```

**Note:**

- The arguments in function header are called “Formal Arguments” or “Formal Parameters”.

**Example:**

```
int add(int x,int y) => x,y are Formal Arguments
```

- The arguments in function call are called “Actual Arguments” or “Actual Parameters”.

**Example:**

```
int a=5,b=4;  
add(a,b);    => a,b are Actual Arguments
```

**Return:**

- “return” is a jumping control structure.
- It can be used to come out of function.
- It can return a value to function call.
- “return” keyword is used to return a value.
- It is used to send a value outside of a function.
- In C-Language, a function can return one value only. It cannot return many values.
- Default return type is “int” in C-Language. If we don’t specify return type, int will be taken.

---

**Different ways of writing a Function:**

A Function can be defined in four ways based on Argument and return. They are:

- No Argument, No Return
- Argument, No return
- No Argument, return
- Argument, Return

**No Argument, No Return:**

This kind of functions cannot take any arguments and cannot return any value.

**Example:**

```
#include<stdio.h>
void add(); // Function Declaration
void add() //Function Definition
{
    int x,y;

    printf("Enter x,y:");
    scanf("%d%d",&x,&y);

    printf("sum=%d\n",x+y);
}
void main()
{
    add();
    add();
}
```

**Output:**

```
Enter x,y:6 4
sum=10
Enter x,y:10 20
sum=30
```

**Argument, No Return:**

This kind of functions can take arguments. But cannot return any value.

**Example:**

```
#include<stdio.h>
void add(int,int);
void add(int x,int y)
{
    printf("sum=%d\n",x+y);
}
main()
{
    add(5,4);
    add(60,40);
}
```

**Output:**

```
sum=9
sum=100
```

**No Argument, Return:**

This kind of functions cannot take any arguments. But they can return a value.

**Example:**

```
#include<stdio.h>
int add();
int add()
{
    int x,y;

    printf("Enter x,y:");
    scanf("%d%d",&x,&y);

    return x+y;
}
```

```
void main()
{
    int k;
    k=add();
    printf("sum=%d\n",k);

    printf("sum=%d\n",add());
}
```

**Output:**

```
Enter x,y:6 4
sum=10
Enter x,y:20 30
sum=50
```

**Argument and Return:**

This kind of functions can take arguments and they can return a value.

**Example:**

```
#include<stdio.h>
int add(int,int);
int add(int x,int y)
{
    return x+y;
}
void main()
{
    int k;
    k=add(1,2);
    printf("sum=%d\n",k);

    printf("sum=%d",add(4,3));
}
```

**Output:**

```
sum=3
```



sum=7

## What is “void”?

- “void” is a data type which indicates “nothing” (or) “no value”.
- Generally, when a function does not return any value, we write return as “void”. It indicates function is returning nothing.

## Creating our own Header File:

There are two types of header files. They are:

- Predefined Header Files
- User-Defined Header Files

### Predefined Header Files:

The header file defined by C-Language developers is called “Predefined Header File”.

**Example:** `stdio.h` `math.h` `string.h`

### User-Defined Header Files:

Like `stdio.h`, `math.h`, we can create our own header file in C-language. It is called “User-Defined Header File”.

**Example:** `nareshit.h` `shiwa.h` `sbibank.h`

## Steps to create User-Defined Header File (nareshit.h):

- Define multiple functions as following:

```
int max(int x,int y)
{
    return x>y ? x : y;
}
int min(int x,int y)
{
    return x<y ? x : y;
}
int power(int n,int p)
{
    int i,r=1;
    for(i=1;i<=p;i++)
        r=r*n;
    return r;
```

```
}  
int fact(int n)  
{  
    int i,f=1;  
    for(i=1;i<=n;i++)  
        f=f*i;  
    return f;  
}
```

- Save above code in a file with the name “nareshit.h”.
- Compile the code. [Execute menu => Compile or) Short cut key => F9]
- Open a new source file.
- Write following code in source file:

```
#include"nareshit.h" // "" searches in Current Directory  
#include<stdio.h> // <> searches in C-Library  
main()  
{  
    printf("fact=%d\n",fact(4));  
    printf("power=%d\n",power(2,3));  
    printf("max=%d\n",max(10,20));  
    printf("min=%d",min(10,20));  
}
```

- Save the file where “nareshit.h” header file is stored.
- Compile the Program & Run It.

## Recursion:

- If a function is called with in the same function, then it is called “Recursion”.
- Recursion is used to call a function within the same function for many times based on the condition.
- All the tasks which we can do using Iterative control structures can be done using “Recursion”.
- Recursion execution is slower than Iterative Control structures.
- To represent some solutions, to provide readability we use Recursion.

**Example:**

```
int fact(int n)
{
    if(n==1)
        return 1;
    else
        return n*fact(n-1);
}
```

**Types of Recursions:**

Recursion can be divided into two types according to number of recursive calls. They are:

- Linear Recursion
- Binary Recursion

**Linear Recursion:**

If a function has one recursive call, then it is called Linear Recursion.

**Example:**

```
int power(int n,int p)
{
    if(p==0)
        return 1;
    else
        return n*power(n,p-1);
}
```

//In the above Example, power function has only one recursive call.

**Binary Recursion:**

If a function has two recursive calls, then it is called Binary Recursion.

**Example:**

```
int fibo(int n)
{
    if(n==0 || n==1)
        return n;
    else
        return fibo(n-1)+fibo(n-2);
}
//In the above Example, fibo function has two recursive calls.
//First recursive call is evaluated first. Then Second.
```

**Storage Classes:**

- Storage class determines the behavior of a variable in terms of scope and lifetime.
- Scope means, Availability of usage.
- Lifetime means, The span of time a variable alive in the memory.

There are four storage classes in C-Language. They are:

- auto
- extern
- static
- register

**auto:**

- "auto" keyword is used to declare the auto variable.
- It can be also called as "Internal Variable" (or) "Automatic Variable" (or) "Local Variable".
- A variable which is declared with in the function (or) block is called "auto variable".
- Even if we use auto keyword or not, the variable declared in function or block will be taken as auto variable by default.
- Scope of auto variable is limited to Function only.

- Lifetime of auto variable is limited to function only.
- When function is terminated, its lifetime will be ended.
- Its default value is 0 [zero in Modern C-Language. Garbage value in Older C-Language].

**extern:**

- "extern" keyword is used to declare "extern variable".
- It can be also called as "External Variable" (or) "Global Variable".
- We declare it outside of all functions [Above of all].
- Scope of extern variable is limited to the Program.
- Lifetime of extern variable is limited to the Program.
- Its lifetime will be ended when the program is terminated.
- Other programs can also share extern variable.

**static:**

- A variable which is declared using "static" keyword is called "static variable".
- It can be used in 2 ways:
  - static local variable => it is declared in function
  - static global variable => it is declared outside of all functions
- For static variable memory will be allocated only once.
- It is initialized only once.
- Its lifetime is limited to program.
- When the program is terminated its lifetime will be ended.
- Its scope depends on the place where it is declared. If static variable is declared in function, then it can be used in function only. If static variable is declared outside of all functions, then it can be used in the program.

**register:**

- "register" is same as "auto" variable only.
- "register" variable will be stored in "CPU Registers".
- For fast accessing purpose we use it.
- Generally, loop variables are declared as registers.
- It cannot be declared outside of function.

Storage Class	Scope	Lifetime	Memory Location	Default Value (Older C)	Default Value (Modern C)
<b>auto</b>	Function	Function	RAM	garbage value	0
<b>extern</b>	Program	Program	RAM	0	0
<b>static (local)</b>	Function	Program	RAM	0	0
<b>static (global)</b>	Program	Program	RAM	0	0
<b>register</b>	Function	-	register	garbage value	garbage value

**Can we define multiple functions in a program?**

Yes. We can define.

**Can we return multiple values from a function?**

No. A function can return one value only in C-Language. We cannot write two return statements in a function. Because it returns the value to function call. So, execution jumps to function call. Error will not be given if we write multiple return statements in a function. Second return statement will not get executed.

**Example:**

```
int f1()
{
    return 5;
    return 8;
    /* Above statement will not get executed. Because return 5 means,
       jumps to function call */
}
```

**Can we call a function from another function?**

Yes. We can.

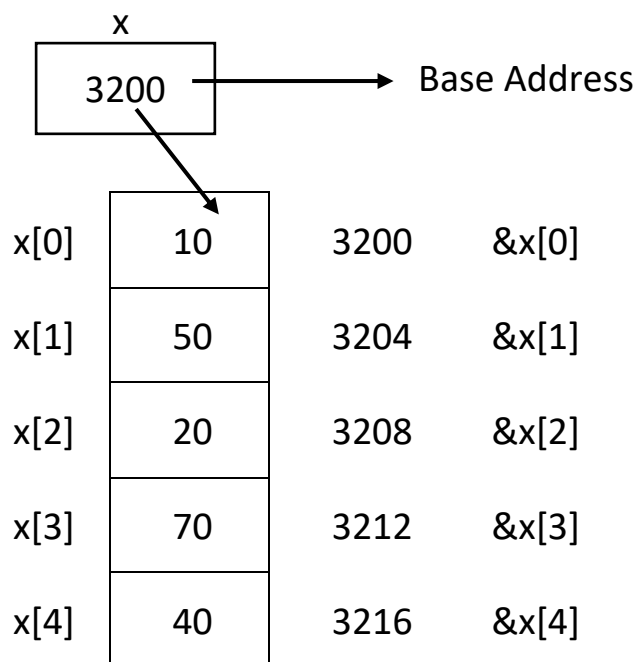
**Note:**

A function must be called directly or indirectly from “main” function. Because it is entry point.

## Arrays

- Array is a set of elements of same data type.
- Array elements share common name.
- To refer Array elements, we use Indexes.
- In C-Language, Array Indexing starts from zero.
- In C-Language, Array Indexes cannot be negative.
- Array elements are stored in sequential memory locations.
- Array Name holds first element's Address. This address is called "Base Address".

**Example:**     `int x[5];`    // x is array name. It holds first element's address



### **Advantages of Array:**

- It decreases number of variables.
- We can group same type elements & access them with same name.
- It provides Code Optimization. [We write less code].
- We can access array elements randomly.

## Disadvantages of Array:

- We must specify Array Size.
- Array size is fixed. Array size cannot be increased or decreased at run time.
- Memory wastage may be there or shortage may be there.
- To insert an element in the array, we need to rearrange almost all elements of the Array.
- To delete an element in the array, we need to rearrange almost all elements of the Array.

## Types of Arrays:

There are two types of Arrays. They are:

- Single Dimensional Array
- Multi-Dimensional Array

### Single Dimensional Array:

- An Array which has one-dimension i. e. either a row or a column is called "Single Dimensional Array".
- It can represent either a row or a column. But not both.

### Example:

10	80	50	(OR)	10	x[0]
				80	x[1]
				50	x[2]
x[0]	x[1]	x[2]			
Row				Column	



**Declaring Single Dimensional Array:****Syntax:**

`<type> <array_name>[<size>;`

**Example-1:**

```
int x[10]; // 40 Bytes Memory will be allocated
```

In above statement, 10 is the maximum size. 'x' array can hold maximum of 10 integer type elements. For every element 4 Bytes memory will be allocated. 40 Bytes memory will be allocated for 10 elements. First element's address will be stored in array name 'x'.

**Example-2:**

```
double x[10]; //80 Bytes memory will be allocated
```

In above statement, 10 is the maximum size. 'x' array can hold maximum of 10 double type elements. For every element 8 Bytes memory will be allocated. 80 Bytes of memory will be allocated for 10 elements. First element's address will be stored in array name 'x'.

**Example-3:**

```
char x[10]; //10 Bytes memory will be allocated
```

In above statement, 10 is the maximum size. 'x' array can hold maximum of 10 char type elements. For every element 1 Byte memory will be allocated. 10 Bytes of memory will be allocated for 10 elements. First element's address will be stored in array name 'x'.

### Initializing Single Dimensional Array:

**Syntax:**

```
<type> <array_name>[<size>] = {<value1>,<value2>,,,,,<valuen>;}
```

**Example:**

```
int x[5] = {50,80,40,90,20};
```

(or)

```
int x[] = {50,80,40,90,20};
```

In C-Language, Array size must be given at the time of declaration. But, in case of initialization, we have no need to specify the size. Based on number of elements size will be taken internally.

### Printing Singl Dimensional Array:

To print first element on console screen, we write:

```
printf("%d",x[0]);
```

To print all elements on console screen, we write the loop. Loop variable represents array index.

To print all elements on console screen, we write:

```
for(i=0; i<5; i++)           // i => Array Index
    printf("%d\n",x[i]);
```

With above Loop, when i value 0 first element will be printed. When i value is 1 second element will be printed and so on.

**Reading Single Dimensional Array:**

To read first element from console screen, we write:

```
scanf("%d",&x[0]);
```

To read all elements from console screen, we write the Loop. Here, Loop variable represents Array Index.

To read all elements from console screen, we write:

```
printf("Enter x-array elements:");  
for(i=0;i<5;i++)           // i => Array Index  
    scanf("%d",&x[i]);
```

With above Loop, when i value is 0, it reads first element. When i value is 1, it reads second element and so on.

**Multi-Dimensional Array:**

Multi-Dimensional Array can represent both rows & columns. It has sub types. They are:

- Two-Dimensional Array => collection of 1D-Arrays
- Three-Dimensional Array => collection of 2D-Arrays
- Four-Dimensional Array => collection of 3D-Arrays
- .
- .
- .
- .
- N-Dimensional Array => collection of N-1 Dimensional Arrays

## Two-Dimensional Array:

- Two-Dimensional Array is a collection of single dimensional arrays.
- It can represent rows and columns.
- In C-Language, Row Indexing starts from zero and Column Indexing starts from zero.
- Like Single Dimensional Array, Double Dimensional Array elements are stored in sequential memory locations.

### Example:

`int x[2][3];` // For 6 elements, 24 Bytes memory will be allocated

		column index-0	column index-1	column index-2
		3200 <code>x[0][0]</code>	3204 <code>x[0][1]</code>	3208 <code>x[0][2]</code>
row index-0	<code>x[0]</code>	10	90	60
row index-1	<code>x[1]</code>	50	40	70
		<code>x[1][0]</code> 3212	<code>x[1][1]</code> 3216	<code>x[1][2]</code> 3220

**Two-Dimensional Array**

In above Example,

3200 => `&x[0][0]`

`x[0]` => 1<sup>st</sup> Row. It represents 3200. First row's First Element Address

`x[1]` => 2<sup>nd</sup> Row. It represents 3212. Second row's First Element Address

`x[0][0]` => 1<sup>st</sup> row 1<sup>st</sup> element

`x[1][0]` => 2<sup>nd</sup> row 1<sup>st</sup> element

---

## Declaring Double Dimensional Array:

### Syntax:

`<type> <array_name>[<row_size>][<column_size>;`

### Example-1:

```
int x[2][3]; // 24 Bytes Memory will be allocated
```

2 Single Dimensional Arrays Each can have 3 elements.

2\*3 = 6 elements. 6 is Maximum size

In above statement, 2 is the maximum row size. 3 is the maximum column size. 'x' array can hold maximum of 6 integer type elements. For every element 4 Bytes memory will be allocated. 24 Bytes of memory will be allocated for 6 elements. First element's address will be stored in array name 'x'. x[0] represents first row. It holds first row's first element address. x[1] represents second row. It holds second row's first element address.

## Initializing Double Dimensional Array:

```
int x[2][3] = { {10,90,60},  
               {50,40,70} };
```

Put Single Dimension Array elements in curly braces. Separate the single dimensional arrays using comma ( , ) and put all single dimensional arrays in curly braces.

(OR)

```
int x[][3] = { {10,90,60},  
              {50,40,70} };
```

We must specify the row size and column size when we declare double dimensional array. But In case of initialization, we have no need to specify row size. But we must specify column size. Based on column size, number of elements in each row will be identified here.

(OR)

```
int x[2][3] = {10,90,60,50,40,70};
```

We can initialize double dimensional array by separating all elements using comma ( , ). Based on column size, every 3 elements will be treated as one row.

### Printing Double Dimensional Array:

To access first element in Double Dimensional Array, we write:

```
printf("%d\n",x[0][0]); // x[0][0] => First Row First Element
```

To print first row elements of double-dimensional Array, we write:

```
for(j=0;j<3;j++)  
    printf("%d\t",x[0][j]);    // x[0] => First Row
```

To print rows and columns, we use nested loop. We write a loop to print a row. Write nested loop to print multiple rows.

```
for(i=0; i<2;i++)  
{  
    for(j=0; j<3; j++)  
        printf("%d\t",x[i][j]);  
    printf("\n");  
}
```

In above code:

```
i  => row index  
2 => number of rows  
j  => column index  
3 => number of columns
```

### Reading Double Dimensional Array:

To read First Row First Element we write:

```
scanf("%d",&x[0][0]);
```

To read first row elements we write:

```
for(j=0;j<3;j++)  
    scanf("%d",&x[0][j]);
```

To read one row use a loop. To read multiple rows use nested loop.

```
printf("Enter x-array elements:");
for(i=0;i<2;i++)
    for(j=0;j<3;j++)
        scanf("%d",&x[i][j]);
```

### Three-Dimensional Array:

Three-Dimensional Array is a collection of double dimensional arrays.

#### Example:

		Column Index-0 <code>x[0][0][0]</code>	Column Index-1 <code>x[0][0][1]</code>	Column Index-2 <code>x[0][0][2]</code>
2D- arrayay index- 0	Row Index-0	10	20	30
	Row Index-1	40	50	60
		<code>x[0][1][0]</code>	<code>x[0][1][1]</code>	<code>x[0][1][2]</code>
		<code>x[1][0][0]</code>	<code>x[1][0][1]</code>	<code>x[1][0][2]</code>
2D- Array index-1	Row Index-0	11	22	33
	Row Index-1	44	55	66
		<code>x[1][1][0]</code>	<code>x[1][1][1]</code>	<code>x[1][1][2]</code>
Three-Dimensional Array				

x[0] => First 2D- Array

x[1] => Second 2D-Array

x[0][0] => First 2D-Array's First Row

x[0][1] => First 2D-Array's Second row

x[1][0] => Second 2D-Array's First Row

x[1][1] => Second 2D-Array's Second Row

x[0][0][0] => First 2D-Array's First Row's First Element

x[1][0][0] => Second 2D-Array's First Row's First Element

### **Declaring Three-Dimensional Array:**

```
int x[2][2][3]; // 48 Bytes memory will be allocated
```

2 Double Dimensional Arrays. Each can have 2 rows and 3 columns  
 $2 * 2 * 3 = 12$  elements. 12 is maximum number of elements.

### **Initializing Three-Dimensional Array:**

```
int x[2][2][3] = { { {10,20,30}, {40,50,60} },  
                  { {11,22,33}, {44,55,66} }    };
```

(OR)

```
int x[][2][3] = { { {10,20,30}, {40,50,60} },  
                 { {11,22,33}, {44,55,66} }    };
```

(OR)

```
int x[2][2][3] = {10,20,30,40,50,60,11,22,33,44,55,66};
```



**Printing 3D-Array:**

```
for(i=0;i<2;i++)
{
    for(j=0;j<2;j++)
    {
        for(k=0;k<3;k++)
            printf("%d\t",x[i][j][k]);
        printf("\n");
    }
    printf("\n");
}
```

**Reading 3-D Array:**

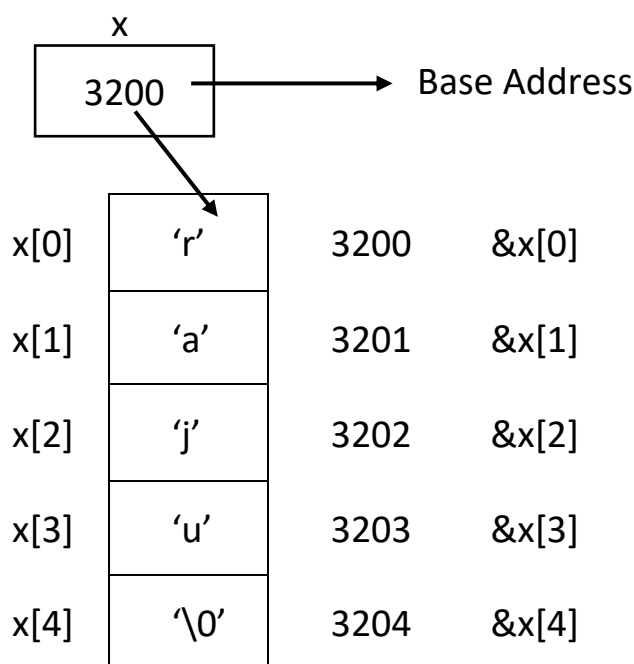
```
printf("Enter x-array elements:");
for(i=0; i<2; i++)
    for(j=0; j<2; j++)
        for(k=0; k<3; k++)
            scanf("%d",&x[i][j][k]);
```

## Strings

- String is a set of characters.
- String Constant must be enclosed in double quotes in C-Language.  
**Ex:** "raju"
- String is a single dimensional array of "char" type.  
**Ex:** char x[10] = "raju"; => x => single dimensional char array
- C-Compiler passes '\0' at end of the string.
- To identify where the string is ended, C-Compiler passes '\0'.
- '\0' is a null character. Its integer equivalent value is 0.
- Character elements are stored in sequential memory locations.
- String name holds first element's address.

### Example:

char x[5]; // x is array name. It holds first element's address



## Declaring a String:

### Syntax:

```
char <array_name>[<size>;
```

### Example:

```
char x[20]; //20 Bytes memory will be allocated
```

In above example, x is array name. 20 is the maximum size. In C-Language, 1 Byte memory will be allocated for 1 character. x-array [String] can hold maximum of 20 characters. So, 20 Bytes memory will be allocated.

## Initializing a String:

```
char x[10] = "raju";
```

(OR)

```
char x[10] = {'r', 'a', 'j', 'u'};
```

(OR)

```
char x[] = "raju";
```

## Printing a String:

```
printf("%s",x);
```

(OR)

```
printf(x);
```

(OR)

```
puts(x);
```

(OR)

```
for(i=0;x[i]!=0;i++)  
    printf("%c",x[i]);
```

## Reading a String:

```
printf("Enter a String:");  
scanf("%s",x); // No need to use &. "x" itself is address
```

(OR)

```
printf("Enter a String:");  
gets(x);
```

## String Functions in C-Language:

C-Language provides following string functions. All these string functions included in "string.h" Header File:

Function Name	Purpose	Example
strlen()	Used to find string length	n=strlen("sai"); printf("%d",n); => 3
strcpy()	Used to copy one string to another	strcpy(x,"sai"); printf("%s",x); => sai
strrev()	Used to reverse a String	char x[10]="sai"; strrev(x); printf("%s",x); => ias
strcat()	Used to concatenate (combine) two strings.	char x[10]="raj"; char y[10]="kumar"; strcat(x,y); printf("%s",x); => Rajkumar

strcmp()	Used to compare two strings. It compares ASCII values of characters internally.  It returns 0 if both are equal. It returns 1 if first string is greater. It returns -1 if second string is greater.	n = strcmp("raj","kumar"); printf("%d",n); => 1  n = strcmp("kumar", "raj"); printf("%d",n); => -1  n = strcmp("raj", "raj"); printf("%d",n); => 0
strlwr()	Used to convert a string to lower case	char x[10] = "RaJu"; strlwr(x); printf("%s",x); => raju
strupr()	Used to convert a string to upper case	char x[10] = "RaJu"; strupr(x); printf("%s",x); => RAJU

## Double-Dimensional Character Array:

Double-Dimensional character array is a set of strings.

### Example:

```
char x[3][10]; //Double Dimensional Char Array
```

Above Array can hold maximum of 3 strings. Each string can have maximum of 10 characters.

	X[0][0]	X[0][1]	X[0][2]	X[0][3]	
x[0]	'r'	'a'	'j'	'u'	'\0'
x[1]	's'	'a'	'i'	'\0'	
x[2]	'r'	'a'	'v'	'i'	'\0'

---

## Declaring Double-Dimensional Character Array:

### Syntax:

```
char <array_name>[<size1>][<size2>;
```

size1 => Maximum Number of Strings

size2 => Maximum Number of Characters

### Example-1:

```
char x[5][10]; // 50 Bytes Memory will be allocated
```

In above statement, x is double dimensional character array. It can hold maximum of 5 strings. Each string can have maximum of 10 characters.

## Initializing Diuble-Dimensional Character Array:

### Syntax:

```
char <array_name>[<size1>][<size2>] = {<string-1>,<str-2>,,,,,<str-n>;
```

### Example:

```
int x[5][10] = {"raju", "sai", "vijay", "kiran", "arun"};
```

(or)

```
int x[][10] = {"raju", "sai", "vijay", "kiran", "arun"};
```

No need to specify first size at the time of initialization. Based on number of strings compiler takes the size implicitly.

**Printing Double-Dimensional Character Array:**

To print first string of Double-Dimensional Character Array, we write:

```
printf("%s",x[0]);
```

To print first character in first string, we write:

```
printf("%c",x[0][0]);
```

To print all strings of Double-Dimensional Character Array, we write:

```
for(i=0; i<5;i++)  
    printf("%s\n",x[i]);
```

**Reading Double-Dimensional Character Array:**

To read a set of strings, we write:

```
printf("Enter 5 strings:");  
for(i=0;i<5;i++)  
    scanf("%d", x[i]);
```

---

## Pointers

- Pointer is a derived data type.
- Pointer is a variable that holds address of the variable of same data type.
- To declare pointer variable use \* [Pointer Operator].
- Pointing to an address gives value at that address.
- 8 Bytes memory will be allocated for pointer variable.

### **Advantages:**

- Pointers are used for fast accessing.
- provides Dynamic Memory Allocation.
- provides Call by Reference.

### **Disadvantages:**

- Security problems may occur due to pointer arithmetic.
- pointer requires extra memory.
- Programmer may feel difficult to write programs using pointers.

### **Note:**

There are 2 types of variables. They are:

- Data Variables
- Pointer Variables

Data Variable holds the data.

Pointer variable holds address of the variable.

### **Declaring a Pointer Variable:**

#### **Syntax:**

`<type> *<variable>;`

#### **Example:**

```
int *p1;    //8 Bytes memory will be allocated
```

p1 is pointer variable of integer type. It can hold Integer variable address

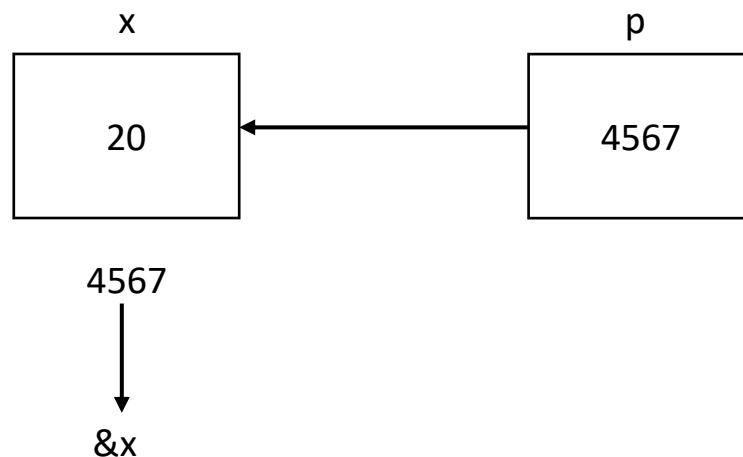
```
float *p2;  //8 Bytes memory will be allocated
```

p2 is the pointer variable of float type. It can hold float variable address



## Initializing a Pointer:

```
int x=20,*p;  
p=&x;    //Initializing a pointer
```



## Different ways of Printing addresses and values:

```
#include<stdio.h>  
main()  
{  
    double x=40,*p;  
  
    p=&x;  
  
    printf("value=%d\n",x);  
  
    printf("address=%d\n",&x);  
  
    printf("address=%d\n",p);  
  
    printf("value=%d\n",&x);  
  
    printf("value=%d\n",*p);  
  
    printf("size of p=%d",sizeof(p));  
}
```

### Output:

```
value=40  
address=6487572  
address=6487572  
value=40  
value=40  
size of p=8
```

## Pointer Arithmetic:

- Using operators on pointers is called pointer arithmetic.
- We can use four operators on pointers. They are:
  - +
  - ++
  - -
  - --
- We can increase address value or decrease address value. But we cannot multiply or divide them.
- We increase the address value to access next memory locations.
- We decrease the address value to access previous elements.
- It is used to access the array elements.

### Program to demonstrate pointer arithmetic:

```
#include<stdio.h>
main()
{
    short int x[] = {10,20,30,40,50},*p;

    p=x;

    printf("%d\n",*p); //10

    p=p+4;
    printf("%d\n",*p); //50

    p--;
    printf("%d\n",*p); //40

    p=p-2;
    printf("%d\n",*p); //20

    p++;
    printf("%d\n",*p); //30
}
```

**Output:**

```
10
50
40
20
30
```

## Arrays & Pointers:

- Pointer concept is suitable to access array elements. Because array elements are stored in sequential memory locations.
- Array name holds first element's address. So, an array name works like a pointer.

Hold first element address in pointer variable and increase the address to access next element as following:

```
int x={10,20,30,40,50},*p;  
  
p=x;    [or]    p=&x[0];  
  
printf("%d\n",*p); //prints first element 10  
  
p++;  
printf("%d\n",*p); //prints second element 20
```

### Accessing Single Dimensional Array using pointer:

To refer single dimensional array element, we write :  $*(x+i)$ .  $i$  is index.

$*(x+i) \Rightarrow$ equivalent to $x[i]$
---

### Accessing Double Dimensional Array using pointer:

To refer double dimensional array element, we write:  $*(*(x+i)+j)$ .  
 $i$  is row index.  $j$  is column index.

$*(*(x+i)+j) \Rightarrow$ equivalent to $x[i][j]$
---

### Accessing Three-Dimensional Array using pointer:

To refer three-dimensional array element, we write:  $*(*(x+i)+j)+k$ .  
i is 2d-array index. j is row index. k is column index

$*(*(x+i)+j)+k$  ) => equivalent to  $x[i][j][k]$

### Array of Pointers:

- Array of pointer means pointer array.
- “Array” holds set of values. Similarly, “Array of Pointers” hold set of addresses of variables of same data type.
- To hold addresses of set of strings or addresses of array elements (or) addresses of set of arrays, we can use it.

#### Example:

```
int x[3]={10,20,30},*p[3]; // *p[3] is an array of pointers
```

```
p[0] = &x[0];
p[1] = &x[1];
p[2] = &x[2];
```

```
printf("%d\n",*p[0]); //prints 10
printf("%d\n",*p[1]); //prints 20
printf("%d\n",*p[2]); //prints 30
```

x[3]			p[3]		
&x[0]	2000	10	x[0]	2000	p[0]
&x[1]	2004	20	x[1]	2004	p[1]
&x[2]	2008	30	x[2]	2008	p[2]

In above example,

x is array. It is holding set of values 10,20 and 30.

p is array of pointers. It is holding a set of addresses 2000, 2004 and 2008.

---

## **Call by Value & Call by Reference {Parameter Passing techniques}:**

There are two parameter passing technique in C-Language. They are:

- Call by Value
- Call by Reference [Address]

### **Call by Value:**

- In this mechanism, values of variables (or) values are passed as arguments to the Function.
- If we make any changes to the variables of called function, those changes will not be applied to the variables of calling function.

#### **Example:**

```
#include<stdio.h>
void swap(int,int);
void swap(int x,int y)
{
    int temp;
    temp=x;
    x=y;
    y=temp;
}
main()
{
    int a=20,b=30;
    swap(a,b);    // Call By Value
    printf("a=%d\tb=%d",a,b);
}
```

#### **Output:**

a=20 b=30

In the above example, a and b variables are not swapped.

## Call by Reference:

- In this mechanism, addresses of variables are passed as arguments to the Function.
- If we make any changes to the variables of called function, those changes will be applied to the variables of calling function.

### Example:

```
#include<stdio.h>
void swap(int*,int*);
void swap(int *x,int *y)
{
    int temp;
    temp=*x;
    *x=*y;
    *y=temp;
}
main()
{
    int a=20,b=30;

    swap(&a, &b);    //call by address [reference]

    printf("a=%d\tb=%d",a,b);
}
```

### Output:

a=30 b=20

In the above example, a and b variables are swapped from called function swap().

---

## Dynamic Memory Allocation:

- Allocating memory at runtime is called “Dynamic Memory Allocation”.
- In this mechanism, Memory will be allocated in Heap Memory Area.
- Advantage is we can expand the memory.
- C-Language provides three functions for Dynamic memory Allocation. They are: malloc() realloc() calloc().
- If memory is allocated using above functions, that memory must be destroyed by using free() function. It is responsibility of the programmer to free the dynamically allocated memory.

C-Language provides following functions for dynamic memory allocation and de-allocation. All these functions are available in “stdlib.h” and “alloc.h” header files:

- malloc()
- realloc()
- calloc()
- free()

### malloc():

- used to allocate specified number of bytes of memory at run time on heap memory.
- This function returns address of first byte & its return type is void. [void type address].
- Type conversion is required to store the data in this memory.

### Example:

```
int *p;  
p = (int*)malloc(50)    => allocates 50 bytes memory  
  
float *x;  
x = (float*)malloc(sizeof(float)) => allocates 4 bytes
```

**realloc():**

- used to reallocate the memory which is already allocated by malloc() function.
- This function returns address of first byte & its return type is void. [void type address].
- Type conversion is required to store the data in this memory.

**Example:**

```
p = (int*)realloc(p,100) => reallocates 100 bytes memory
```

**calloc():**

- used to allocate a group of blocks of memory.
- It is used to allocate the memory to hold a set of values.
- This function returns address of first byte & its return type is void. [void type address].
- Type conversion is required to store the data in this memory.

**Example:**

```
p = (int*)calloc(10,sizeof(int));
```

Above statement allocates 10 blocks of memory. Each memory block size is 4 bytes.  $10 * 4 = 40$  Bytes memory will be allocated.

**free():**

- used to destroy the memory which is allocated using malloc() or realloc() or calloc().
- It is the responsibility of the programmer to destroy the memory if memory allocated at run-time. Automatic memory management is not available for dynamic memory allocation in C-Language.

**Example:**

```
free(p);
```



## Pointer to Pointer:

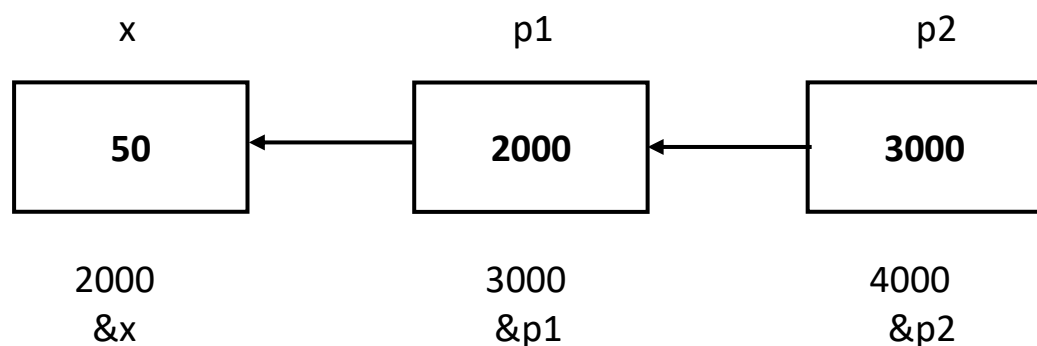
- Pointer to pointer is a variable that holds address of another pointer variable.
- To declare this variable we prefix variable name with “\*\*”.

### Example:

```
int x=50,*p1,**p2; // **p2 -> pointer to pointer
```

```
p1=&x;  
p2=&p1;
```

```
printf("%d\n",x);    //prints 50  
printf("%d\n",*p1); //prints 50  
printf("%d",**p2);  //prints 50
```



---

## **User-Defined Data Types** **[Structures, Unions & Enumerations]**

In C\_Language, We can define our own data types like int, float and char. These are called “User-Defined Data-Types”. User-Defined data types are:

- Structures
- Unions
- Enumerations

### **Structure:**

- Structure is a User-Defined Data Type.
- Structure is a collection of elements of different data types.
- Structure encapsulates [combines] different data type variables in one container. This container is called “Structure”.
- “struct” keyword is used to define the structure.
- To access structure elements, we use Dot Operator [ . ].
- Structure elements are stored in sequential memory locations.
- Every member of a structure has its own memory location.
- Sum of the sizes of the members is size of the structure variable.

### **Advantages:**

- We can define our own data type.
- We can hold different type elements in one container.
- In C-Language, a function can return one value only. But using structure we return group of values.
- Structure can contain another Structure. With this we can extend its properties of structure.

### **Defining a Structure:**

“struct” keyword is used to define a structure. Structure Definition will be terminated with semicolon [ ; ]. It can be defined inside of main() or outside of main().

**Syntax:**

```
struct <structure_name>
{
    <type-1>  <v1>[,<v2>,.....];
    <type-2>  <v1>[,<v2>,.....];
    <type-3>  <v1>[,<v2>,.....];
    .....
    .....
};
```

**Example:**

```
struct student
{
    int rno;
    char name[20];
    float avrg;
};
```

} Members (or)  
Member Variables (or)  
Properties (or)  
Attributes (or)  
Fields (or)  
Instance Variables

**Note:**

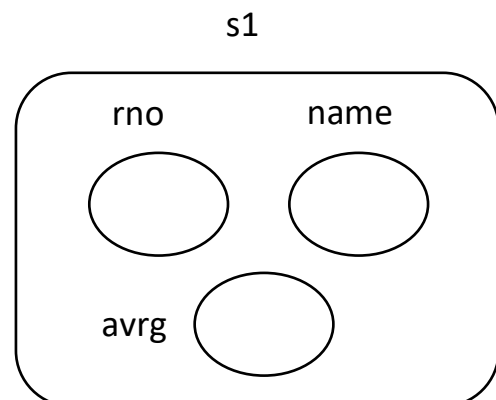
Memory will not be allocated when we define the structure. Memory will be allocated when the structure variable is declared.

**Declaring a Structure Variable:**

```
struct student s1;
```

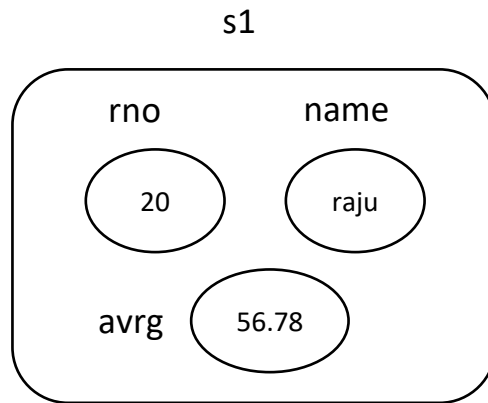
s1 size = rno size + name size + avrg size  
= 4 + 20 + 4  
= 28

28 Bytes memory will be allocated for s1



## Initializing a Structure Variable:

```
struct student s1 = {20, "raju", 56.78};
```



## Printing Structure Elements:

To access structure elements, use dot operator [ . ]. To print all elements we write:

```
printf("%d\t",s1.rno);  
printf("%d\t",s1.name);  
printf("%d",s1.avrg);
```

## Reading Structure Elements:

To read all structure elements we write:

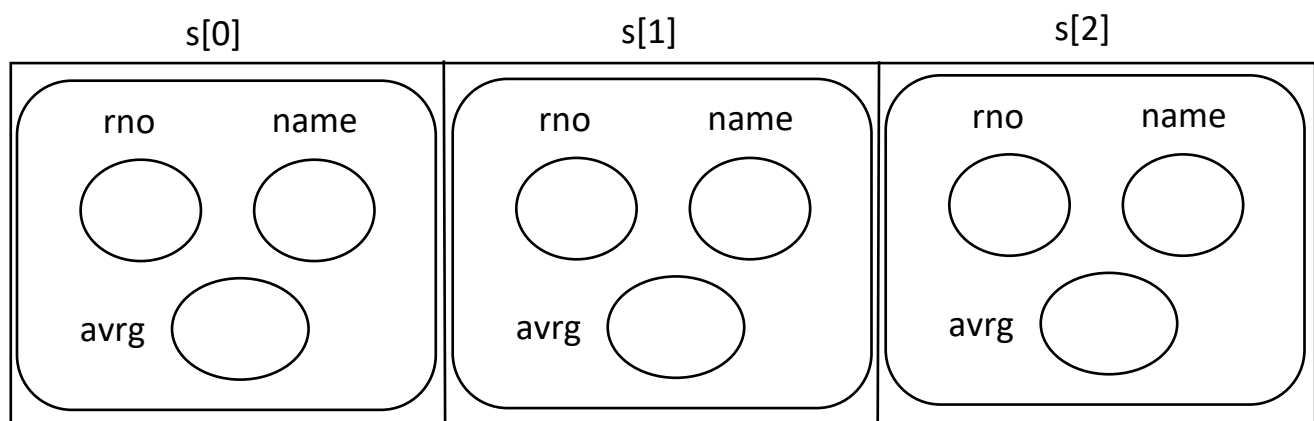
```
printf("Enter rno,name,average marks:");  
scanf("%d%s%f",&s1.rno,s1.name,&s1.avrg);
```

## Array of Structure:

Array of Structure can hold a set of structures. Array of structure is declared as following:

```
struct student s[3]; // Array of structure
```

‘s’ is an array of structure that can hold 3 students records.



In above example,

's' is the array of structure that can hold 3 students records.

s[0]       => First Student

s[0].rno   => First student's rno

s[0].name  => First Student's name

s[0].avrg   => First Student's Average marks

s[1]       => Second Student

s[1].rno   => First student's rno

s[1].name  => First Student's name

s[1].avrg   => First Student's Average marks

s[2]       => Third Student

s[2].rno   => First student's rno

s[2].name  => First Student's name

s[2].avrg   => First Student's Average marks

## Structure of Array:

- Defining array as a member in structure is called “Structure of Array”.
- To hold many members of same type, we can use Array as a member in structure.

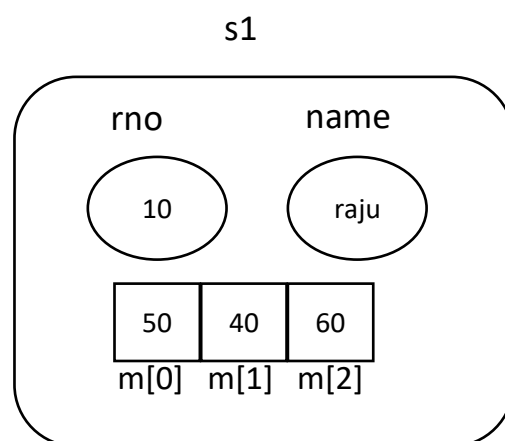
## Defining Structure of Array:

```
struct student
{
    int rno;
    char name[20];
    int m[3];      //Structure of Array
};
```

In the above example, m is the array that can hold 3 subjects marks.

## Initializing Structure variable:

```
struct student s1 = {10,"raju",{50,40,60}};
```

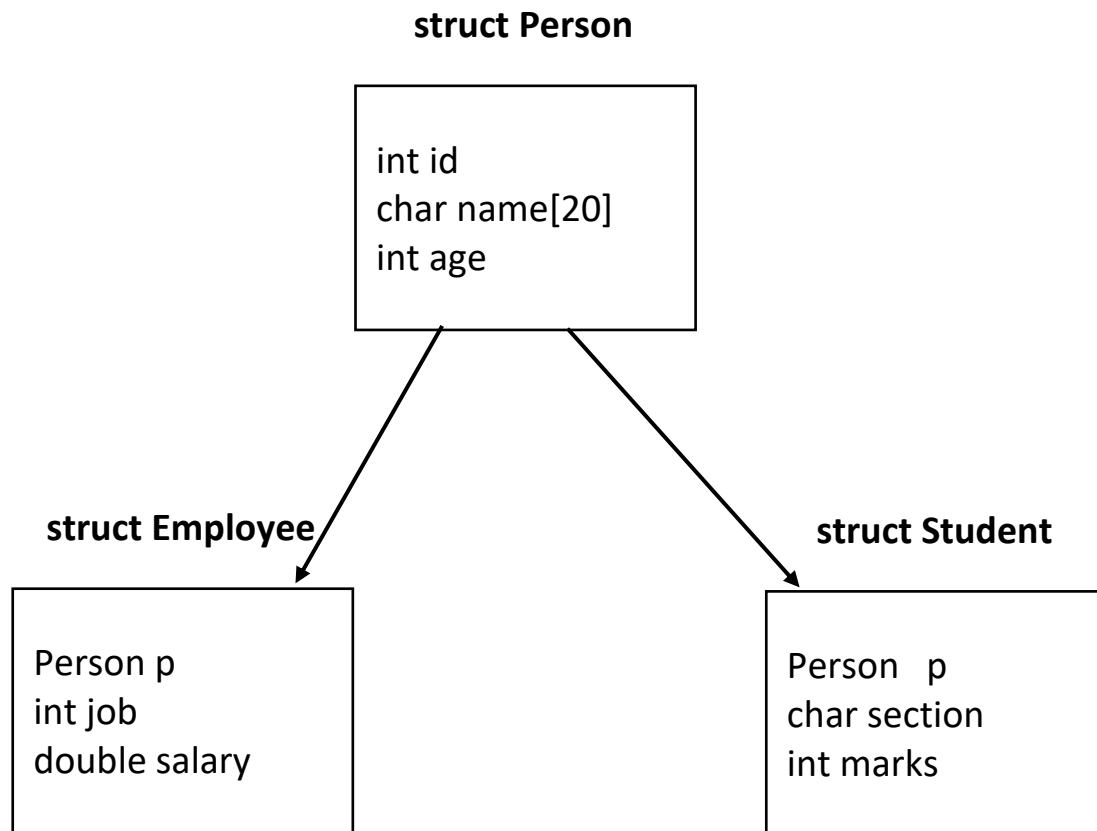


In the above example,

s1	=>	student1
s1.m[0]	=>	Student1's First Subject Marks

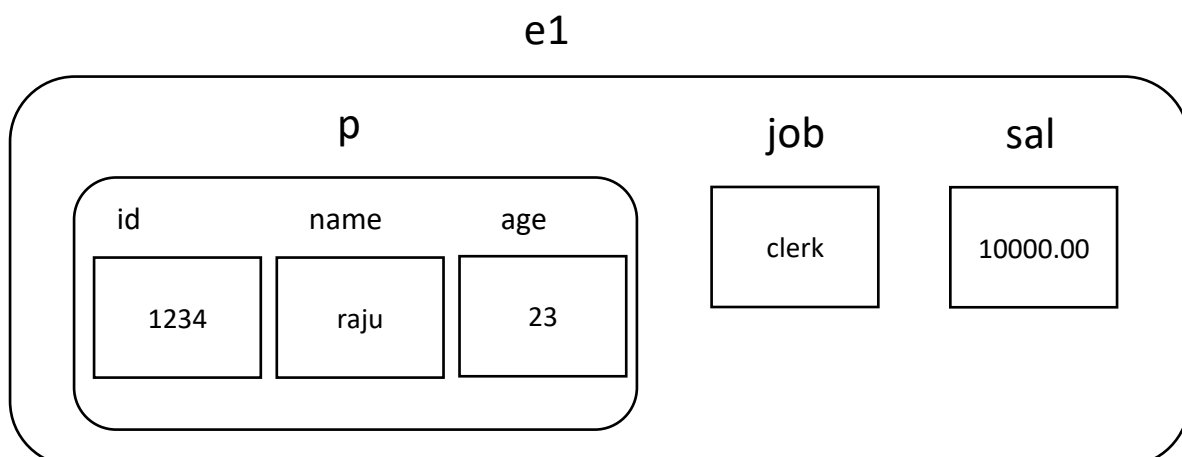
## Structure in Structure [Nested Structure]:

Defining structure as a member in another structure is called “Structure in Structure”.



In the above example,  
“p” is a member of “Person” type which is declared in another structure “Employee”. It is called “Structure in Structure”. “p” can hold three members.

```
struct Employee e1 = { 1234,"raju",23,"clerk",10000.00};
```



In the above example, e1 structure has p structure. It is called "Structure in Structure".

To Access Employee data which is in p variable we write:

```
e1.p.id  
e1.p.name  
e1.p.age
```

## Pointer to Structure:

A pointer variable which holds address of a structure is called pointer to structure.

For Example,  
Define a structure as following:

```
typedef struct student  
{  
    int rno;  
    float avrg;  
}std;
```

Declare structure variable, initialize it & declare a pointer variable to hold address of s1.

```
std s1,*p;  
p=&s1;
```

Now we can access structure elements using pointers, we write:

```
printf("%d\t%f\n", (*p).rno, (*p).avrg );
```

[OR]

We can use arrow operator ( -> ) to access structure elements using pointers as following:

```
printf("%d\t%f\n", p->rno, p->avrg);
```

<b>(*p).rno is equivalent to p-&gt;rno</b>
--



---

## Structure as Argument:

Like primitive data type variable, we can pass structure as argument to function. When we pass structure as actual argument, all structure elements will be copied into formal argument [Call by value].

### Program to demonstrate structure as argument:

```
#include<stdio.h>
typedef struct student
{
    int rno;
    char name[20];
    int marks;
}std;
void show(std);
void show(std s)
{
    printf("%d\t%s\t%d\n",s.rno,s.name,s.marks);
}
main()
{
    std s1;

    printf("Enter rno,name and marks:\n");
    scanf("%d%s%d",&s1.rno,s1.name,&s1.marks);

    show(s1);    //Passing Structure as Argument
}
```

### Output:

```
Enter rno,name and marks:
20 Sravan 500
20    Sravan    500
```

## Returning Structure:

In C-Language, a function can return only one value. By returning structure we can return set of values to another function.

### Program to demonstrate returning structure:

```
#include<stdio.h>
typedef struct student
{
    int rno;
    char name[20];
    int marks;
}std;

std f1();

std f1()
{
    std s={12,"Vijay",456};

    return s;
}
main()
{
    std s1;

    s1 = f1();

    printf("%d\t%s\t%d",s1.rno,s1.name,s1.marks);
}
```

### Output:

12 Vijay 456

In the above example, s returned to main() function. s values copied into s1.

## Union:

- Union is a User-Defined Data Type.
- Union is a collection of elements of different data types. But it can hold one value at a time.
- Union encapsulates [combines] different data type variables in one container. This container is called “Union”.
- “union” keyword is used to define the structure.
- To access union elements, we use Dot Operator [ . ].
- All Union members share common memory area.
- Maximum size required for individual members is size of the union variable.

## Defining a Union:

“union” keyword is used to define a union. Union Definition will be terminated with semicolon [ ; ]. It can be defined inside of main() or outside of main().

### Syntax:

```
union <union_name>
{
    <type-1>  <v1>[,<v2>,.....];
    <type-2>  <v1>[,<v2>,.....];
    <type-3>  <v1>[,<v2>,.....];
    .....
    .....
};
```

### Example:

```
union abc
{
    int x;
    double y;
    char z;
};
```

} Members (or)  
Member Variables (or)  
Properties (or)  
Attributes (or)  
Fields (or)  
Instance Variables

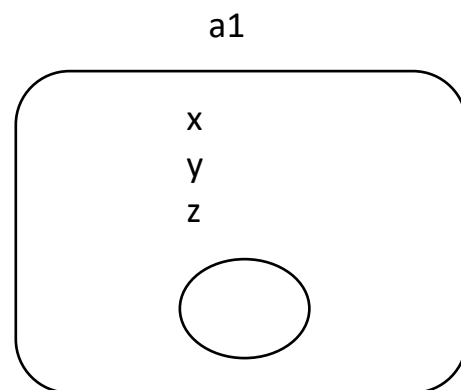
## Declaring a union Variable:

```
union abc a1;
```

a1 size = max size required for individual members

8 Bytes memory will be allocated for a1.

x, y and z members share same memory area. At a time it can hold one value only.



**Program to demonstrate union members share same memory location:**

```
union aaa
{
    int x;
    int y;
};
main()
{
    union aaa a1;

    a1.x = 20;
    printf("%d\n",a1.y); //20

    a1.y = 30;
    printf("%d",a1.x); //30
}
```

**Output:**

```
20
30
```

**Enumeration:**

- Enumeration is a user-defined data type.
- It is used to define string constants with integer equivalent values.
- “enum” keyword is used to define Enumeration.
- When we don't give integer values to string constants in enumeration definition, they take default values. This default numbering starts from zero.

**Syntax:**

```
enum <union_name>
{
    <String_Constant-1>=<value-1>,
    <String_Constant-2>=<value-2>,
    .....
};
```

**Example:**

```
enum colors {RED=10, BLUE=20, GREEN=30 };
```

**Program to demonstrate enumeration:**

```
#include<stdio.h>
enum colors { RED=10, BLUE=20, GREEN=30 };
typedef enum colors colors;
main()
{
    colors c1,c2,c3;
    c1=RED;
    c2=BLUE;
    c3=GREEN;
    printf("%d\t%d\t%d",c1,c2,c3);
}
```

**Output:**

```
10    20    30
```

## Files

- “File” is a collection of related data.
- “Files” concept is used to store the data permanently in a file.
- “File Management System” is a methodology (or) a way of organizing files. It allows us to perform file operations like:
  - Writing data into file
  - Reading data from file
  - Copying one file data to another
  - Renaming a File
  - Removing a File

### **Different modes to open a File:**

In C-Language, A file can be opened in following modes:

File Mode	Meaning
w	Write
a	Append
r	Read
w+	Write Read
r+	Read Write
a+	Append Read
wb	Write Binary
rb	Read Binary
ab	Append Binary
w+b	Write Read Binary
r+b	Read Write Binary
a+b	Append Read Binary

## Functions that are used to work with files:

Function	Purpose
fopen()	Used to open create a new file (or) open an existing file in particular mode
fclose()	Used to close a file
getc()	Used to read a character from file
putc()	Used to write a character into file
fprintf()	Used to write any type of data in file
fscanf()	Used to read any type of data from file
rewind()	Used to move the file pointer to BOF (Beginning of file)
fseek()	Used to move the file pointer to specific position For random accessing, we use it
ftell()	Used to know the file pointer position
fread()	Used to read data from binary file
fwrite()	Used to write the data into binary file

## Opening a File:

### fopen():

- "fopen()" function is used to create a new file or open an existing file in particular mode.
- This function returns address of the file. This address is "FILE" type address.

### Syntax:

```
<file_pointer> = fopen(<file_name>,<mode>);
```

### Example:

```
fp = fopen("demo.txt","w");  
fp = fopen("d:\\nareshit\\student.txt", "a");  
fp = fopen("demo.txt","r");
```

## Difference between “w” [write] mode and “a” [append] mode:

<b>fp= fopen(“demo.txt”,“w”);</b>	<b>fp= fopen(“demo.txt”,“a”);</b>
<p><b>First Time:</b> First, it creates a new file “demo.txt” &amp; opens in write mode.</p> <p><b>Second Time onwards:</b> Existing file will be opened without contents &amp; file pointer will be placed at Beginning Of File [BOF]. From BOF, it writes the data.</p>	<p><b>First Time:</b> First, it creates a new file “demo.txt” &amp; opens in append mode.</p> <p><b>Second Time onwards:</b> Existing file will be opened with contents safe &amp; file pointer will be placed at Ending Of File [EOF]. From EOF, it writes the the data. It means, it is appending [adding] data to existing data.</p>

## Closing a file:

### **fclose():**

“fclose()” function is used to close an opened file. After performing file input output operations, we must close the opened file. Otherwise, we may loss the data.

#### **Syntax:**

```
fclose(<file_pointer>);
```

#### **Example:**

```
fclose(fp);
```



## **getc() & putc() Functions:**

These functions are included in “stdio.h”.

### **putc():**

putc() function is used to write a character into file. To write multiple characters [Text], we can use Loop.

#### **Syntax:**

```
putc(<char>,<file_pointer>);
```

#### **Example:**

```
putc(ch,fp);
```

“ch” value will be written “fp” file with above code.

### **getc():**

getc() function is used to read a character from file. To read multiple characters [Text], we can use Loop.

#### **Syntax:**

```
<char> = getc(<file_pointer>);
```

#### **Example:**

```
ch = getc(fp);
```

reads a character from “fp” file & it will be stored in “ch” variable with above code.

---

## **fprintf() & fscanf() functions:**

These functions are included in “stdio.h” header file.

### **fprintf():**

- This function is used to write the data into file.
- Using this function, we can write any type of data into file like int, float & char ...etc.

#### **Syntax:**

```
fprintf(<file_pointer>,<format_string>,<argument_list>);
```

#### **Example:**

```
fprintf(fp, "%d\t%s", rno, name);
```

“rno” and “name” data will be written in “fp” file with above code.

### **fscanf():**

- This function is used to read the data from file.
- Using this function, we can read any type of data from file like int, float & char ...etc.

#### **Syntax:**

```
fscanf(<file_pointer>,<format_string>,<argument_list>);
```

#### **Example:**

```
fscanf(fp, "%d%s", &rno, name);
```

“rno” and “name” data will be read from “fp” file with above code.

**fseek & ftell():**

These functions are included in “stdio.h” header file.

**ftell():**

This function is used to get the position of file pointer.

**Syntax:**

```
ftell(<file_pointer>);
```

**Example:**

```
n = ftell(fp);
```

Above code returns file pointers position & it will be stored in “n”.

**fseek():**

This function is moved to move file pointer to specified number of bytes from specified position.

**Syntax:**

```
fseek(<file_pointer>,<number_of_bytes_to_move>,<from_where>);
```

String Constant [or] value	Meaning
SEEK_SET [or] 0	From Beginning of File
SEEK_CUR [or] 1	From Current Position
SEEK_END [or] 2	From End of File

**Example:**

fseek(fp,2,SEEK_SET)	Moves 2 bytes forward from beginning of file
fseek(fp,3,SEEK_CUR)	Moves 2 bytes forward from current position
fseek(fp,4,SEEK_END)	Moves 4 bytes backward from end of file

## Types of Files:

There are 2 types of files. They are:

- Text File
- Binary File

### Text File:

- Text File contains plain text.
- It contains data as we entered data through keyboard.
- **Example:** .txt files

### Binary File:

- Binary file contains data in the form of bits and bytes.
- It contains the data as the data stored in memory.
- **Example:** .bin files, .exe files., image files, audio files, video files

## fwrite() & fread() functions:

These are included in “stdio.h” header file.

### fwrite():

“fwrite()” function is used to write the binary data into file.

### Syntax:

```
fwrite (<address_of_structure_variable>, <size_of_structure_variable>,  
<number_of_records_to_write>, <file_pointer>);
```

### Example:

```
fwrite(&s1, sizeof(s1), 1, fp);
```

Above code writes s1 records into “fp” file”.

**fread():**

“fread()” function is used to read the binary data into file.

**Syntax:**

```
fwrite (<address_of_structure_variable>, <size_of_structure_variable>,  
<number_of_records_to_write>, <file_pointer>);
```

**Example:**

```
fread(&s1, sizeof(s1), 1, fp);
```

Above code reads a record & stores in “s1”

---

## Graphics

There are 2 types of applications. They are:

- CUI Applications [Character User Interface]
- GUI Applications [Graphical User Interface]

### **CUI Application:**

CUI Application can display text (or) characters only.

### **GUI Application:**

GUI Application can display text, colors, shapes, font styles and images.

### **Shapes:**

C-Language provides following functions to draw the shapes:

Function Name	Purpose
line()	Used to draw a line
rectangle()	Used to draw a rectangle
circle()	Used to draw a circle
ellipse()	Used to draw an ellipse (or) part pf the ellipse
arc()	Used to draw an arc or circle
pieslice()	Used to get part [slice] in a circle
putpixel()	Used to print a pixel [dot]
drawpoly()	Used to draw polygon

## initgraph() Function:

This function is used to switch the screen from text mode to graphics mode.

### Syntax:

```
Initgraph(address of graphics drivers, address of graphics mode,  
path of BGI Folder);
```

### Example:

```
int gd, gm;  
gd = DETECT; //automatically detects graphic drivers  
initgraph(&gd, &gm, "C:\\\\TURBOC3\\\\BGI");
```

Above code detects the graphics drivers & output screen will be switched to graphics mode.

## Color String Constants or Values:

In C-Language, 16 colors can be used in graphics applications. They are numbered from 0 to 15. "colors" enumeration defined in "conio.h" header file which has 16 color string constants. They are:

Color String Constant	Value
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15

**line() :**

line() function is used to draw a line.

**Syntax:**

```
line(x1, y1, x2, y2);
```

**Example:**

```
line(200,100,300,200);
```

Above code draws a line from (200,100) point to (300,200) point.

**rectangle() :**

rectangle() function is used to draw a box [rectangle (or) square].

**Syntax:**

```
rectangle(x1, y1, x2, y2);
```

**Example:**

```
rectangle(200,100,300,200);
```

Above code draws a rectangle from (200,100) point to (300,200) point.

**circle() :**

circle() function is used to draw a circle.

**Syntax:**

```
circle(x, y, radius);
```

**Example:**

```
circle(200,200,100);
```

Above code draws a circle at (200,200) point with 100 radius.



**putpixel() :**

putpixel() function is used to draw a pixel [dot].

**Syntax:**

```
putpixel(x, y, color);
```

**Example:**

```
circle(200,200,RED);
```

Above code draws a pixel at (200,200) point in RED color.

**drawpoy() :**

drawpoly() function is used to draw the polygon.

**Syntax:**

```
drawpoly(number_of_points, array);
```

**Example:**

```
int x[] = {100,100,200,200,300,100,400,200};  
drawpoly(4,x);
```

Above code draws a line from (100,100) to (200,200). Then (200,200) to (300,100). Then (300,100) to (400,200).

**ellipse() :**

ellipse() function is used to draw an ellipse.

**Syntax:**

```
ellipse(x, y, staring_angle, ending_angle,x-radius, y-radius);
```

**Example:**

```
ellipse(200, 200, 0, 360, 100, 50);
```

Above code draws an ellipse at (200,200) point from 0 degrees to 360 degrees angle with 100 x-axis radius and 50 y-axis radius.

**arc():**

arc() function is used to draw an arc.

**Syntax:**

```
arc(x, y, starting_angle, ending_angle, radius);
```

**Examples:**

arc(200,200,0,90,100); => 0 to 90 degrees arc will be drawn with 100 radius from the point (200,200).

arc(200,200,90,180,100); => 90 to 180 degrees arc will be drawn with 100 radius from the point (200,200).

arc(200,200,90,270,100); => 90 to 270 degrees arc will be drawn with 100 radius from the point (200,200).

arc(200,200,0,360,100); => 0 to 360 degrees arc will be drawn with 100 radius from the point (200,200). It means, it draws a circle.

**pieslice() :**

pieslice() function is used to a slice [part] of a circle. It is same as arc. But it is closed one.

**Syntax:**

```
pieslice(x, y, starting_angle, ending_angle, radius);
```

**Example:**

```
pieslice(200,200,0,90,100);
```

Above code draws a slice of circle at (200,200) point with 100 radius from 0 to 90 degrees.

**setfillstyle() and floodfill():**

These two functions are used to fill the colors in shapes.

**setfillstyle():**

This function is used to set fill pattern and fill color.

**Syntax:**

```
setfillstyle(pattern_constant/value, color_constant/value);
```

**Example:**

```
setfillstyle(1, 4); // 1 is pattern number. 4 is color number  
(or)  
setfillstyle(SOLID_FILL, RED);
```

color number ranges from 0 to 15 [16 colors].

pattern number ranges from 0 to 12 [13 patterns].

Pattern Constants & values are listed below:

Pattern String Constant	Value
EMPTY_FILL	0
SOLID_FILL	1
LINE_FILL	2
LTSLASH_FILL	3
SLASH_FILL	4
BKSLASH_FILL	5
LTBKSLASH_FILL	6
HATCH_FILL	7
XHATCH_FILL	8
INTERLEAVE_FILL	9
WIDE_DOT_FILL	10
CLOSE_DOT_FILL	11
USER_FILL	12

**floodfill():**

This function is used to fill the color in shape. It fills from specified point to till it reaches the specified color border.

**Syntax:**

```
floodfill(x, y, color);
```

**Example:**

```
floodfill(200, 200, BLUE);
```

**Example on floodstyle & floodfill():**

```
setcolor(BLUE);  
circle(200,200,100);  
setfillstyle(SOLID_FILL, GREEN);  
setfloodfill(200,200,BLUE);
```

above code fills GREEN color in circle. From (200,200) point GREEN color will be filled till it reaches BLUE color border.

**outtextxy() :**

outtextxy() function is used to draw the text at specified point in specified color.

**Syntax:**

```
outtextxy(x, y, color);
```

**Example:**

```
outtextxy(200, 200, "hello");
```

Above code draws "hello" at (200,200).

**settextstyle() :**

It is used to set text style, text direction and text size.

**Syntax:**

```
settextstyle(text_style_constant / value,  
text_direction_constant / value, text_size);
```

**Example:**

```
settextstyle(4,0,10);  
(or)  
settextstyle(GOTHIC_FONT, HORIZ_DIR, 10);
```

With above code, text will be displayed horizontally with font size 10 & in “GOTHIC\_FONT” style.

**Example on outtextxy() and settextstyle():**

```
settextstyle(GOTHIC_FONT, HORIZ_DIR, 10);  
outtextxy(200, 200, “hello”);
```

With above code, “hello” will be printed horizontally.

```
settextstyle(SANS_SARIF_FONT, VERT_DIR, 10);  
outtextxy(200, 200, “hi”);
```

With above code, “hi” will be printed vertically.

Text direction constants are listed below:

Text Direction String Constant	Values
HORIZ_DIR	0
VERT_DIR	1

Font Styles are listed below:

Font Style String Constant	Value
DEFAULT_FONT	0
TRIPLEX_FONT	1
SMALL_FONT	2
SANS_SARIF_FONT	3
GOTHIC_FONT	4