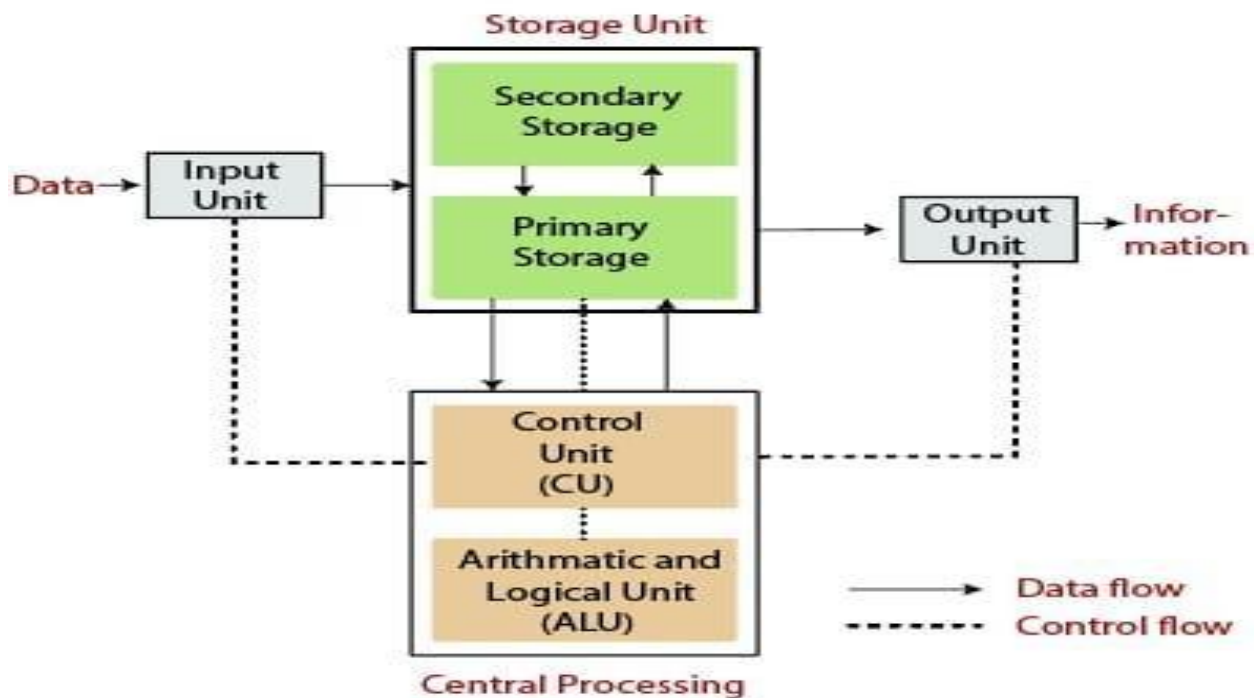


**UNIT-1: Introduction to Programming****Block diagram of Computer**

The computer performs basically five major operations of functions. These are

- 1) it accepts data or instruction by way of input.
- 2) it stores the data and instructions.
- 3) it can process data as required by the user.
- 4) it gives results in the form of output.
- 5) it controls all operations inside a computer.

1. Input: This is the process of entering data and programs into the computer system.

2. Central Processing Unit (CPU): The ALU and the CU of a computer system are jointly known as the central processing unit (CPU).

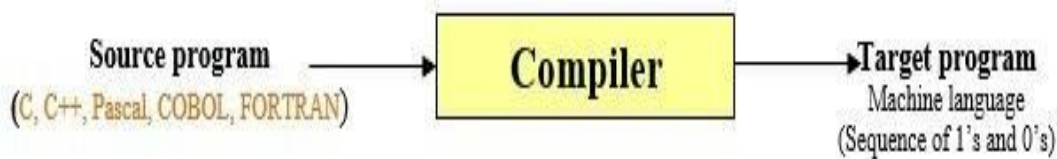
3. Control Unit (CU): The process of input, output, processing and storage is performed under the supervision of a unit called 'Control Unit'. It decides when to start receiving data, when to stop it, where to store the data etc. It takes care of step-by-step processing of all operations inside the computer.

4. Arithmetic Logic Unit (ALU): The major operations performed by the ALU are addition, subtraction, multiplication, division, logic and comparison etc.

5. Memory Unit: Memory unit is used to store data and instructions.

6. Output: This is the process of producing results from the data for getting useful information.

**Compiler:**



Compiler is a software that translates the program written in HLL to machine language. The program written in HLL is referred to as the source code and compiled program is referred to as object code. Machine language and is also known as object code. During this translation, compiler notifies if there are any ERRORS in the program. Each programming language has its own compiler.

Examples of compilers are: Turbo C, C++, Borland C, C++ compilers.

A C program has to pass through many phases for its successful execution and to achieve the desired output.

The various steps involved in the program development are as follows:

1. Editing phase
2. Compilation phase
3. Linking phase
4. Execution phase

### 1. Editing Phase:

In the editing phase, the C program is entered into a file through a text editor. The file is saved on the disk with an extension of 'C'. Corrections in the program at later stages are done through these editors. Once the program has been written, it requires to be translated into machine language.

### 2. Compilation Phase:

This phase is carried out by a program called as compiler. Compiler translates the source code into the object code. The compilation phase cannot proceed successfully until and unless the source code is error-free. The compiler generates messages if it encounters syntactic errors in the source code. The error-free source code is translated into object code and stored in a separate file with an extension '.obj'.

### 3. Linking Phase:

In this phase, the linker links the files and functions with the object code under execution. The result of this linking process produces an executable file with an extension of '.exe'. Now the object code is ready for next phase.

### 4. Execution Phase:

In this phase, the executable object code is loaded into the memory and the program execution begins. We may encounter errors during the execution phase even though compilation phase is successful. The errors may be run-time errors and logical errors.

**Algorithm:**

An algorithm is a step by step process to solve a problem.

The algorithm which is written in English is known as pseudo code.

**Properties of an Algorithm:**

Every algorithm should have the following 5 properties.

1. Input: An algorithm may take one or more inputs.
2. Output: An algorithm should produce at least one output.
3. Finiteness: An algorithm should end after a fixed number of steps.
4. Definiteness: Each instruction of an algorithm must be clear and well defined.
5. Effectiveness: The operations must be simple and each step of an algorithm must be easily convertible into program statement.

**Example:** Algorithm to find the largest number among 3 numbers.

Step 1: Start

Step 2: Declare the variables A, B, C

Step 3: Read/Input the values of A, B, C

Step 4: Check if  $(A > B)$  then goto step 5, otherwise goto step 6

Step 5: Check if  $(A > C)$  then goto step 7, otherwise goto step 9

Step 6: Check if  $(B > C)$  then goto step 8, otherwise goto step 9

Step 7: Output "A is the largest", then goto step 10

Step 8: Output "B is the largest", then goto step 10

Step 9: Output "C is the largest", then goto step 10

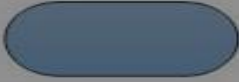

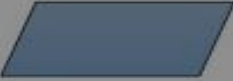
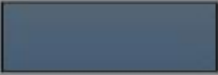

Step 10: Stop

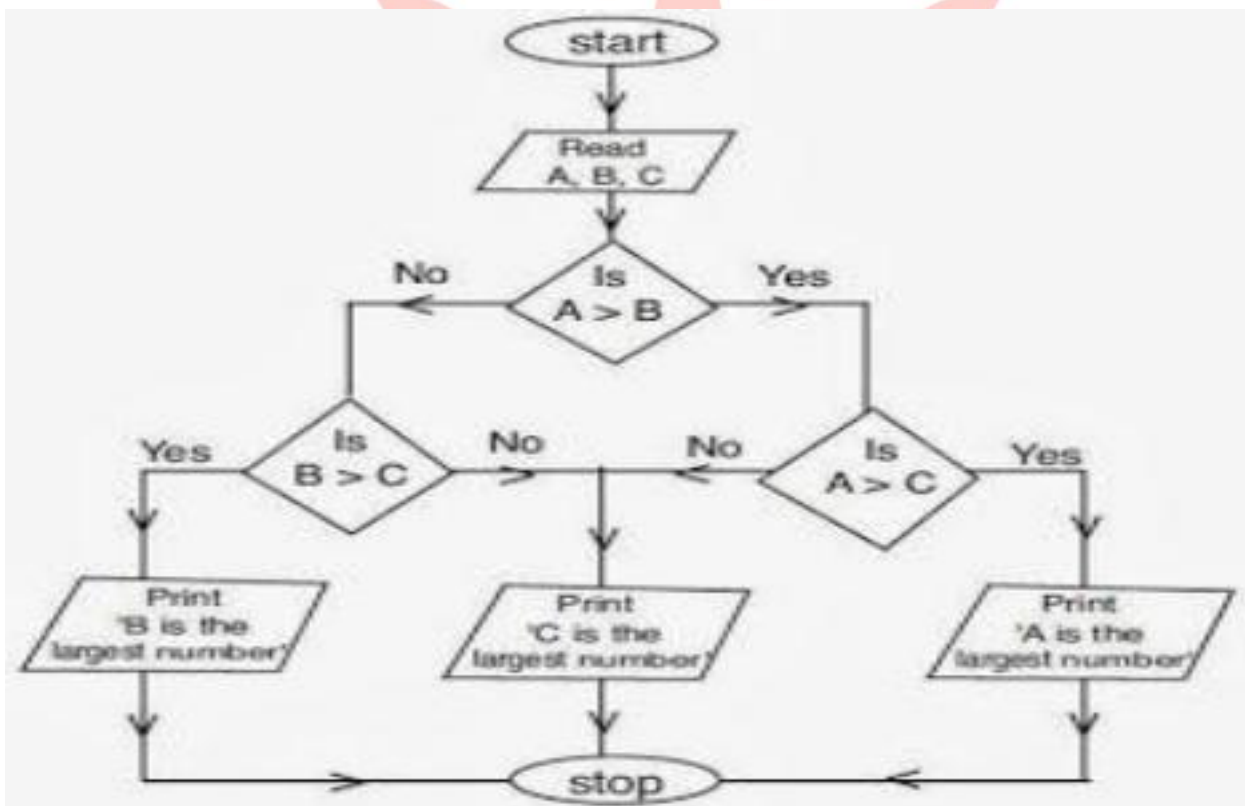
**Flowchart:**

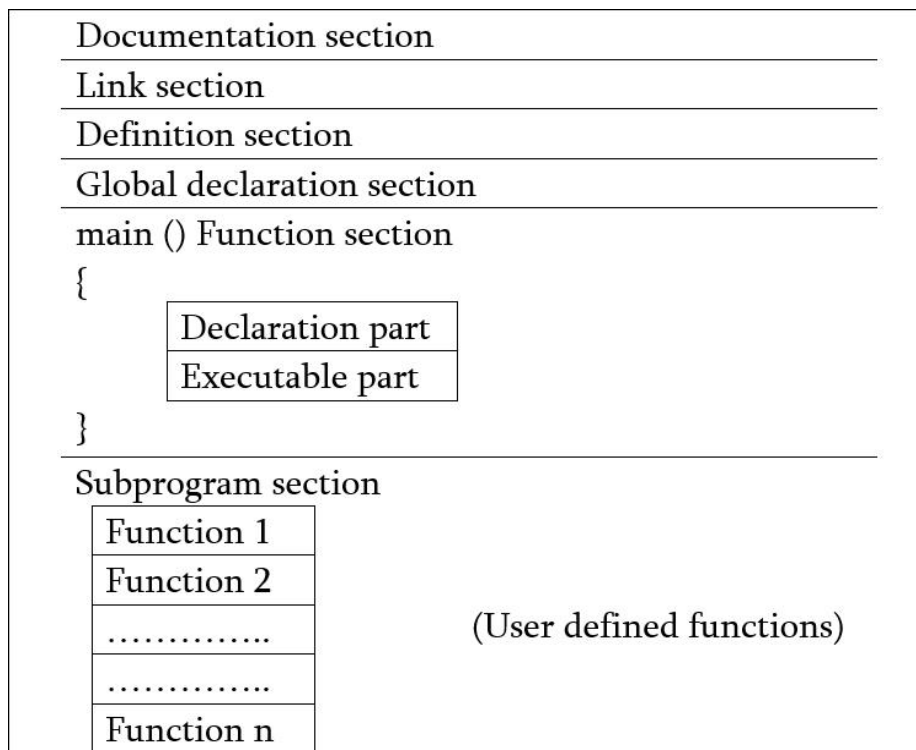
A graphical/pictorial representation of an algorithm is called flowchart. The following symbols are used to construct a flow chart.

your roots to success...

**Example:** Flowchart to find the largest number among 3 numbers.

Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision



**Structure of a C program:****1. Documentation section:**

It is a set of comment lines that describes the name of the program.

Examples: // Write a C program to find the sum of 2 numbers  
 /\* Write a C program to find the simple interest\*/

**2. Link section:**

It tells the compiler to link functions from the system library.

Examples:

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

```
#include<conio.h>
```

```
#include<math.h>
```

#include is a pre-processor directive command.

stdio is a standard input & output.

.h is an extension of header file.

**3. Definition section:**

It defines all the symbolic constants and assigns them some value.

Example: #define PI 3.14

Here #define is a pre-processor directive command which tells the compiler whenever PI is found in the program replace it with 3.14.

**4. Global declaration section:**

There are some variables used more than one function such variables are called global variables and are declared

outside of all the functions.

### **5. main() function section:**

Every C program should contain one main() function.

Every C program execution starts with main( ).

This section contains two parts.

#### **1. Declaration part:**

This part declares all the variables which are used in the executable part.

Examples:

```
int a, b, c;  
float x, y, z;
```

```
char name, branch;
```

#### **2. Executable part:**

This part contains atleast one statement.

All the statements in the declaration and executable parts should be end with semicolon (;). The program execution begins at the opening brace and ends with closing brace.

### **6. Subprogram section/user defined function section:**

It contains all the user defined functions that are called in the main() function section. All the user defined functions are placed before or after the main() function section.

### **C Tokens:**

The basic or smallest units in C program are called C Tokens. They are 6 types of tokens in C language. C programs are written using these tokens and syntax of the language.

5. Keywords/Reserved words
6. Identifiers
7. Constants/Literals
8. Strings
9. Special symbols
10. Operators

**1. Keywords:** Keywords are fixed and predefined meaning in C language. There are 32-Keywords.

## 32 Keywords in C Programming Language

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

### Rules for declaring keywords:-

1. Keywords must be written in the lower case letters.
2. Keywords should not be declared as variables or identifiers.

**2. Identifiers:** Identifiers are names given to the variables, functions and arrays defined by the user. Identifier is a sequence of letters, digits and one special character i.e., \_ (Underscore).

Ex:- a, b, abc , abc \_123,a123,\_xyz etc.

### Rules for declaring Identifiers:-

1. Identifiers must be start with a letter or \_ (Underscore).
2. Identifiers should not start with a digit.
3. No special symbols other than \_ (Underscore) are allowed within the identifiers.
4. Identifiers are case sensitive i.e., the uppercase and lowercase letters are different.
5. Identifiers should not be declared as keywords.
6. Identifiers can be a length of first 31 characters.

### 3.Constant/Literals:-

Constants are the fixed values that do not change during the execution of a program. Constant are divided into two types.

1. Numeric constants
2. Non-Numeric /Character constants

1.Numeric Constants: There are two types of Numeric constants

1. Integer constants
2. Real or floating point constants

1.Integer constants:- It is a sequence of digits without decimal points.

Ex: - 10 ,20 ,30 etc.

Rules for declaring Integer constants:-

1. It must contain at least one digit.
2. It should not contain any decimal point.
3. It could be either positive integer or negative integer constant.
4. If no sign precedes an integer constant, it is assumed to be positive.

2. Real/Floating point constants: It is a sequence of digits with decimal points.

Ex:- 10.32 , 2.02 etc.

Rules for declaring Real constants:-

1. It must contain at least one digit.
2. It should contain decimal point.
3. It could be either positive or negative real constant.
4. If no sign precedes a real constant, it is assumed to be positive.

2. Non-Numeric Constants/Character constant:- There are two types of character constants

1. Single character constants
2. String character constants

1. Single character constants:- Any single letter or digit enclosed within single quotation marks is called single character constant.

Ex:- 'A', '5'

2. String character constant:- String is a group of characters enclosed within the double quotation marks is called string character constants.

Ex:- "MRCET", "CSEA", "AIML"

4. Strings:- A string in C is merely an array of characters. The length of a string is determined by a terminating null character: '\0'. So, a string with the contents, say, "abc" has four characters: 'a', 'b', 'c', and the terminating null ('\0') character. The terminating null character has the value zero.

5. Special symbols :- {}, (), [], \$, #, @ are the special symbols.

6. Operators :- Operator is a symbol which specifies an operation to be performed on the operands.

Ex:- Arithmetic operators, Logical operators, Relational operators, Increment and Decrement operators etc.

Variable:

1. Variable is a data name which can be used to store a data value.
2. Variables can be changed because it takes different values at different times during the execution of a program.



**Declaration/Syntax of a variable** :- datatype variable1,variable 2, ,variable n;

Ex1:- int a, b, c;

Ex2:- float x, y,z;

**Assigning values to variables:-** Values can be assigned to a variable using assignment operator(=) in C language.

**Declaration/Syntax :-**

datatype variable =constant;

Ex1:- int a=10,b=34,c=43; Ex2:- float x=10.3,y=34.15;

**Rules for declaring variables :-**

1. Variable is a sequence of letters, digits, special character ‘\_’ (Underscore).
2. Variables must be start with a letter or \_ (Underscore).
3. Variables should not start with a digit.
4. No special symbols other than \_ (Underscore) is allowed within the variables.
5. Variables can be a length of first 31 characters.
6. Variables are case sensitive i.e., the upper case and lower case letters are different.
7. Variables should not be declared as keywords.

**Data types:**

3. Data type indicates what kind of data can be stored in the variables and identifiers.
4. It can be used for storage representations.
5. Data types are fixed and predefined meaning in C language.
6. Data types can be divided into four types: -
  - i) Primary / fundamental data types  
Ex: - int,short int,long int,float,double,long double,char.
  - ii) Derived data types  
Ex: - arrays,pointers,functions,strings,structures,unions
  - iii) User defined data types  
Ex: - typedef,enum.
  - iv) Empty data type  
Ex: - void.

**I. Primary/Fundamental data types :**

There are three types:-

1. Integer Data types

2. Character Data types
3. Real/Floating point Data types

### 1.Integer Data type: -

It can be used to represent/store integer constants.

There are 3 types.

- 1.int
- 2.short int
- 3.long int

#### 1.int: -

1. It is indicated by the keyword 'int'.
2. The control string of 'int' is %d.
3. It occupies of 2 bytes of memory location i.e., 16bits.
4. The range of int is -32768 to 32767.

Ex: - int x,y;  
scanf ("%d%d",&x,&y);

#### 2.short int: -

1. It is indicated by the keyword 'short'.
2. When the given value is less than the int range then short int is used.
3. The control string of short int is %d.
4. It occupies 1byte of memory location i.e., 8bits.
5. The range of short int is -128 to 127.

Ex: - short int x,y; or short x,y;  
scanf ("%d%d",&x,&y);

#### 3.long int: -

- 1.It is indicated by the keyword 'long'.
- 2.When the given value is greater than the 'int' range then long int is used
- 3.The control string of long int is "%ld"
- 4.It occupies 4bytes of memory location i.e., 32bits.
- 5.The range of long int is -2,147,483,648 to 2,147,483,647.

Ex: - long int x,y; or long x,y;  
scanf ("%ld%ld",&x,&y);

### 2.Character Data types : -

It can be used to represent/store either single character constants or string character constants.

- 1.It is indicated by the keyword 'char'.

2. The control string char is '%c' for single character constants and '%s' for string character constants.

3. It occupies 1 byte of memory location i.e., 8 bits.

4. The range of char is -128 to 127.

Ex1: char x,y;

```
scanf("%c%c",&x,&y);
```

Ex2: char name, college;

```
scanf("%s%s",&name,&college);
```

### 3. Real/Floating point data types : -

It can be used to represent store Real/Floating point constants. There are 3 types.

1. float

2. double

3. long double

1. float: -

- 1) It is indicated by the keyword 'float'.
- 2) The control string of float is "%f".
- 3) It occupies 4 bytes of memory location. i.e., 32 bits.
- 4) The range of float is  $3.4E-38$  to  $3.4E+38$ .

Ex: - float a,b;

```
scanf("%f%f",&a,&b);
```

2. double: -

- 1) It is indicated by the keyword 'double'.
- 2) When the given value is greater than the float range then double is used.
- 3) The control string of double is "%lf".
- 4) It occupies 8 bytes of memory location i.e., 64 bits.
- 5) The range of double is  $1.7E-308$  to  $1.7E+308$ .

Ex: - double a,b;

```
scanf("%lf%lf",&a,&b);
```

3. long double: -

- 1) When the given value is greater than the double range then long double is used.
- 2) The control string of long double is "%Lf".
- 3) It occupies 10 bytes of the memory location i.e., 80 bits.
- 4) The range of long double is  $3.4E-4932$  to  $1.1E+4932$ .

Ex: long double a,b;

```
scanf("%Lf%Lf",&a,&b);
```

### Operator:

Operator is a symbol which indicates an operation to be performed between the operands (Variables or Values).

1. Arithmetic operators: +, -, \*, /, %
2. Relational operators: >, <, <=, >=, ==, !=
3. Logical operators: &&, ||, !
4. Assignment operators: =
5. Conditional/Ternary operator: ? :
6. Increment and Decrement operators: ++, --
7. Bitwise operators: &, |, ^, <<, >>, ~
8. sizeof() operator: sizeof()
9. Special operators: Comma operator(,), dot/member operator(.), address operator(&), value at address operator (\*), arrow/selection operator(->)

#### 1. Arithmetic operator :-

1. Arithmetic operators are used to perform basic arithmetic operations between the operands.
2. There are 5 types of Arithmetic operators in C language they are +, -, \*, /, %.
3. Division operator(/) gives the result in quotient.
4. Modulus operator(%) gives the result in remainder.
5. % operator cannot be worked with the real constants.

#### 2. Relational operators :-

1. Relational operators are used to compare the relation between the operands.
2. There are 6 types of relational operators in C language they are <, >, <=, >=, !=, ==.
3. Relational operator gives the result either in 1 (true) or 0 (false).

#### 3. Logical operators :-

1. Logical operators are used to combine two or more relational conditions.
2. Logical operators gives the result either in 1 (true) or 0 (false).
3. There are 3 types of Logical operators in c language
  1. Logical AND (&&).
  2. Logical OR (||).
  3. Logical NOT (!).

**4. sizeof operator :-** This operator can be used to find the number of bytes occupied by a variable or data type.

Syntax:-        sizeof(operand);

**5. Assignment operator :-** Values can be assigned to a variables using assignment operator in c language. The symbol of assignment operator is '='.

Assignment operator supports 5short hand assignment operators in C language they are += , - = , \* = , /= , %=

**6. Conditional operator/Ternary operator :-**

In C language '?' and ':' are called conditional operator or ternary operators in C language.

Syntax:        Condition ? TRUE Part : FALSE Part;

Here first the condition is checked, if it is true then TRUE Part is executed otherwise FALSE Part is executed.

**7. Increment and Decrement operators :-**

1. These operators are used to increment or decrement the value of the variable by 1.
2. '++' is the increment operator in C language.
3. '--' is the decrement operator in C language.
4. There are 4 types of operators in C language.
  1. Pre-increment operator
  2. Post-increment operator
  3. Pre-decrement operator
  4. Post-decrement operator

Pre-increment operator:

1. ++ operator is placed before the operand is called pre-increment operator. Ex:  
++x, ++a
  - a. Pre-increment operator specifies first increment the value of the variable by 1 and then assigns the incremented value to other variable.

2. Post-increment operator:

1. ++ operator is placed after the operand is called post-increment operator. Ex:  
x++, a++
2. Post-increment operator specifies first assigns the value of the variable to other variable and then increments the value of the variable by 1.

3. Pre-decrement operator:

1. -- operator is placed before the operand is called pre-decrement operator. Ex: -- x, --a
2. Pre-decrement operator specifies first decrement the value of the variable by 1 and then assigns the decremented value to other variable.

4. Post-decrement operator:

1. -- operator is placed after the operand is called post-decrement operator. Ex: x--  
,a--
2. Post-decrement operator specifies first assigns the value of the variable to other variable and then decrements the value of the variable by 1.

**8. Bitwise operators:** The data stored in a computer memory as a sequence of 0's and 1's. There are some operators work with 0's and 1's are called bitwise operators. Bitwise operators cannot be worked with real constants.

There are 6 types of operators in C language.

1. Bitwise AND (&)
2. Bitwise OR (|)
3. Bitwise X-OR (^)
4. Bitwise left shift (<<)
5. Bitwise right shift (>>)
6. Bitwise 1's complement (~)

1. Bitwise AND (&): The result of the bitwise AND is 1 when both the bits are 1 otherwise 0.

2. Bitwise OR (|): The result of the bitwise OR is 0 when both the bits are 0 otherwise 1.

3. Bitwise X-OR (^): The result of the bitwise X-OR is 1 when one bit is 0 and other bit is 1 otherwise 0.

4. Bitwise left shift (<<): This operator can be used to shift the bit positions to the left by 'n' positions.

5. Bitwise right shift (>>): This operator can be used to shift the bit positions to the right by 'n' positions.

6. Bitwise 1's complement (~): This operator can be used to reverse the bit i.e., it changes from 0 to 1 and 1 to 0.

### Operator Precedence:

Operator precedence is used to determine the order of operators evaluated in an expression. In C programming language every operator has precedence (priority). When there is more than one operator in an expression the operator with higher precedence is evaluated first and the operator with the least precedence is evaluated last. This rule of priority of operators is called 'operator precedence'.

The following table shows the precedence of operators:

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
? :	right to left
= += -= *= /= %= &= ^=  = <<= >>=	right to left
,	left to right

To understand expression evaluation in C, let us consider the following simple example expression.

**10 + 4 \* 3 / 2**

In the above expression, there are three operators +, \* and /. Among these three operators, both multiplication and division have the same higher precedence and addition has lower precedence. So, according to the operator precedence both multiplication and division are evaluated first and then the addition is evaluated. As multiplication and division have the same precedence they are evaluated based on the associativity. Here, the associativity of multiplication and division is **left to right**. So, multiplication is performed first, then division and finally addition. So, the above expression is evaluated in the order of \* / **and** +. It is evaluated as follows...

4	*	3	====>	12
12	/	2	====>	6
10	+	6	====>	16

The expression is evaluated to **16**.

### Storage Classes:

Storage class of a variable defines the following terms:

1. Where the variable would be stored.
2. What is the scope of the variable.
3. What is the default value of the variable.
4. What is the life time of the variable.

Based on the above terms storage classes are of four types:

1. Automatic storage class
2. Register storage class
3. Static storage class
4. External storage class

#### **1. Automatic storage class:**

1. It is indicated by the keyword 'auto'.
2. It is stored in main memory.
3. The scope of auto storage class is local.
4. The default value of auto storage class is garbage value.
5. Lifetime of auto storage class is till the control remains within the function.
6. Syntax: auto datatype variable;
7. Example: auto int a;

#### **2. Register storage class:**

1. It is indicated by the keyword 'register'.
2. It is stored in CPU registers.
3. The scope of register storage class is local.
4. The default value of register storage class is garbage value.
5. Lifetime of register storage class is till the control remains within the function.
6. Syntax: register datatype variable;
7. Example: register int a;

#### **3. Static storage class:**

1. It is indicated by the keyword 'static'.
2. It is stored in main memory.
3. The scope of static storage class is local.
4. The default value of static storage class is garbage value.
5. Lifetime of static storage class is till the control remains within the function.
6. Syntax: static datatype variable;
7. Example: static int a;

#### **4. External storage class:**

1. It is indicated by the keyword 'extern'.
2. It is stored in main memory.
3. The scope of extern storage class is global.
4. The default value of extern storage class is zero.



5. Lifetime of extern storage class is till the program terminates.
6. Syntax: extern datatype variable;
7. Example: extern int a;

In a C programming language, the data conversion is performed in two different methods as follows.

### 7. Type Conversion

### 8. Type Casting

## 1. Type Conversion

The type conversion is the process of converting a data value from one data type to another data type automatically by the compiler. Sometimes type conversion is also called implicit type conversion. The implicit type conversion is automatically performed by the compiler.

**For example**, in C programming language, when we assign an integer value to a float variable the integer value automatically gets converted to float value by adding decimal value 0. And when a float value is assigned to an integer variable the float value automatically gets converted to an integer value by removing the decimal value.

For example,

```
int i=10; float
```

```
x=15.5; char
```

```
ch='A';
```

`i = x ;` =====> x value 15.5 is converted as 15 and assigned to variable i

`x = i ;` =====> Here i value 10 is converted as 10.000000 and assigned to variable x

`i = ch ;` =====> Here the ASCII value of A (65) is assigned to i

## 2. Type Casting

It is also called explicit type conversion and it is used to convert from one data type to another data type. To convert data from one type to another, we specify the new type in parentheses before the value we want converted.

For example, to convert an integer a to float, we code the expression like `int a;`

```
(float) a;
```

### Command line arguments:

The most important function of C/C++ is `main( )` function. It is mostly defined with a return type of `int` and without parameters.

```
int main()
```

```
{
```

```
/* ... */
```

```
}
```

The arguments can be passed from operating system command prompt to `main( )`. They are parameters/arguments supplied to the program when it is invoked.

To pass command line arguments, we typically define `main()` with two arguments : first argument is an integer value, that specify “the number of command line arguments” and second is “list of command-line arguments”. Therefore they are called as Command line arguments.

### Command line arguments are two types.

They are

1. `argc`
2. `argv`

1. `argc`- is an argument count that specifies number of arguments in the command line including program name.

2. `argv`- is an argument vector that specifies array of strings.

Syntax:

```
int main(int argc, char *argv[])
{
}
```

**Example:** To find the sum of two integer numbers using command line arguments in C. `#include <stdio.h>`

```
int main(int argc, char *argv[])
{
    int a,b,sum; if(argc!=3)
    {
        printf("please use in specified format\n"); printf("please
        use \"prg_name value1 value2 \\\"n\""); exit(1); }
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    sum = a+b;
    printf("Sum of %d, %d is: %d\n",a,b,sum);
}
```

your roots to success...

### Conditional/Decision making statements:

There are 5 types of Conditional/Decision making statements in C language.

They are

9. `_if` statement
10. `_if-else` statement
11. `_nested if-else` statement
12. `_if-else-if ladder` statement
13. `_switch` statement

**1.if statement:** It can be used to execute a single statement or a group of statements if and only if the specified condition is true.

It is a one way branching statement in C language.

**Syntax:**

```
if (condition)
{
Statements;
}
next statement;
```

**Operation:** First the condition is checked if it is true then the statements will be executed and then control is transferred to the next statement in the program.

if the condition is false, control skips the statements and then control is transferred to the next statement in the program.

**Ex:** Write a C program to check whether the given number is greater than 25. #include<stdio.h>

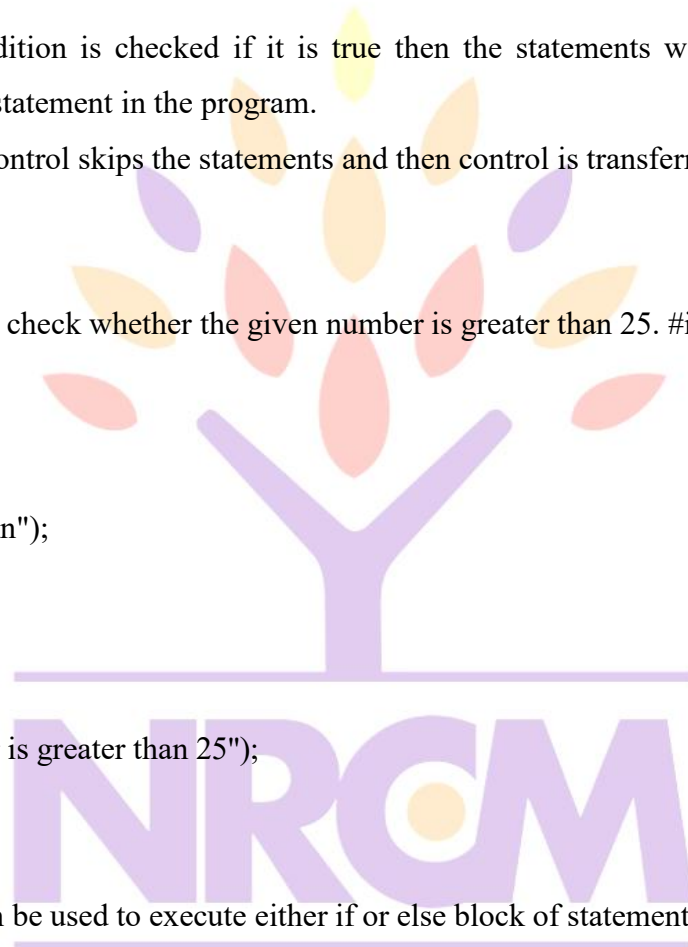
```
int main()
{
int n;
printf("Enter any number\n");
scanf("%d",&n);
if (n>25)

printf("The given number is greater than 25");
}
```

**2.if-else statement:** It can be used to execute either if or else block of statements based on the condition. It is a two way branching statement in C language.

**Syntax:**

```
if (condition)
{
statement1;
}
else
{
statement2;
}
next statement;
```



Operation: First the condition is checked if the condition is true then statement1 will be executed and it skips statement2 and then control is transferred to the next statement in the program.

if the condition is false then statement1 is skipped and statement2 will be executed and then control is transferred to the next statement in the program.

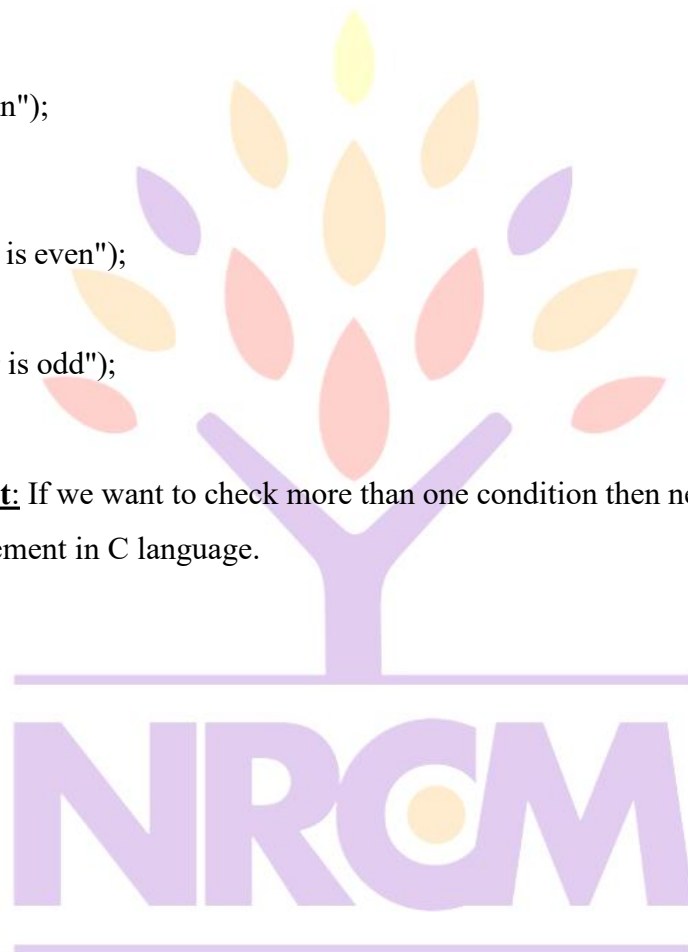
**Ex:** Write a C program to check whether the given number is even or odd.

```
#include<stdio.h>
int main()
{
int n;
printf("Enter any number\n");
scanf("%d",&n);
if (n%2==0)
printf("The given number is even");
else
printf("The given number is odd");
}
```

**3.nested if-else statement:** If we want to check more than one condition then nested if else is used. It is multi way branching statement in C language.

**Syntax:**

```
if (condition1)
{
if (condition2)
{
statement1;
}
else
{
statement2;
}
}
else
{
statement3;
}
next statement;
```



your roots to success...

Operation: First the condition1 is checked if it is false then statement3 will be executed and then control is

transferred to the next statement in the program.

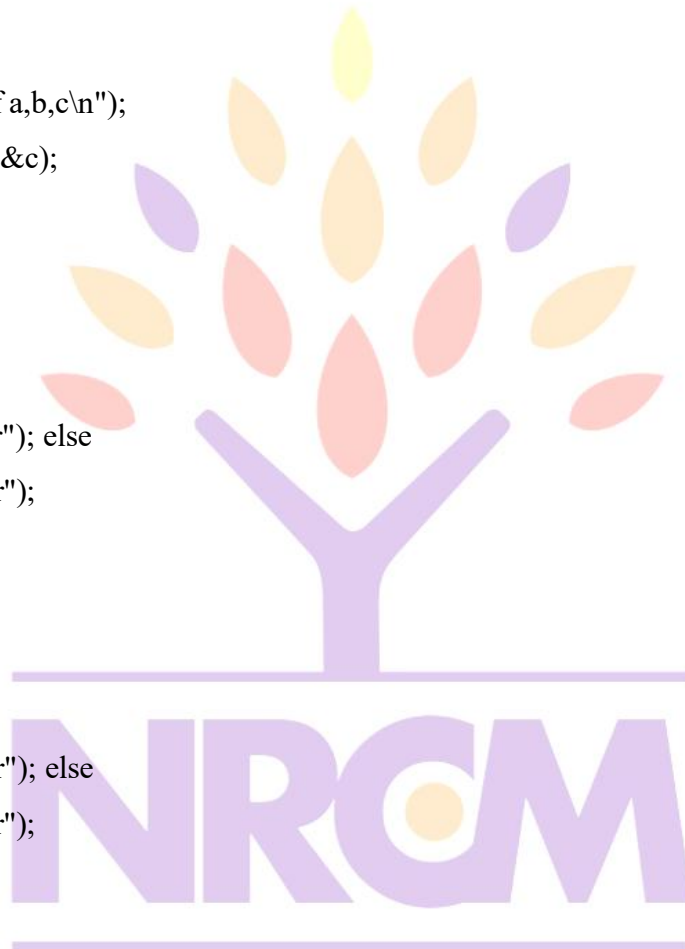
If the condition1 is true then condition2 is checked, if it is true then statement1 will be executed and then control is transferred to the next statement in the program.

If the condition2 is false then statement2 will be executed and then control is transferred to the next statement in the program.

**Ex:** Write a C program to find the largest number among three numbers using nested if else. #include<stdio.h>

```
int main()
{
int a,b,c;
printf("Enter the values of a,b,c\n");
scanf("%d%d%d",&a,&b,&c);
if (a>b)
{
if (a>c)

printf("a is largest number"); else
printf("c is largest number");
}
else
{
if(b>c)
printf("b is largest number"); else
printf("c is largest number");
} }
}
```



**4. if-else-if ladder statement:** If we want to check the multiple conditions then if-else-if ladder statement is used.

It is multi way branching statement in C language.

**Syntax:**

```
if(condition1)
{
statement1;
}
else if(condition2)
{
statement2;
```

```

}
else if(condition3)
{
    statement3;
}
else if(condition n)
{
    statement n;
}
else
{
    default statement;
}
next statement;

```

Operation: This type of structure is known as the else-if ladder. This chain generally looks like a ladder hence it is also called as an else-if ladder. The test-expressions are evaluated from top to bottom. Whenever a true test-expression is found, statement associated with it is executed. When all the n test-expressions become false, then the default else statement is executed.

**Ex:** Write a C program to calculate percentage and grade based on input marks of five subjects. #include

```
<stdio.h>
```

```
int main()
```

```
{
```

```
    int phy,chem,eng,math,comp;
```

```
    float per;
```

```
    printf("Enter five subjects marks of a student\n");
```

```
    scanf("%d%d%d%d%d",&phy,&chem,&eng,&math,&comp);
```

```
    per=(phy+chem+eng+math+comp)/5.0;
```

```
    printf("Percentage=%f\n",per);
```

```
    if(per>=70)
```

```
    {
```

```
        printf("Distinction");
```

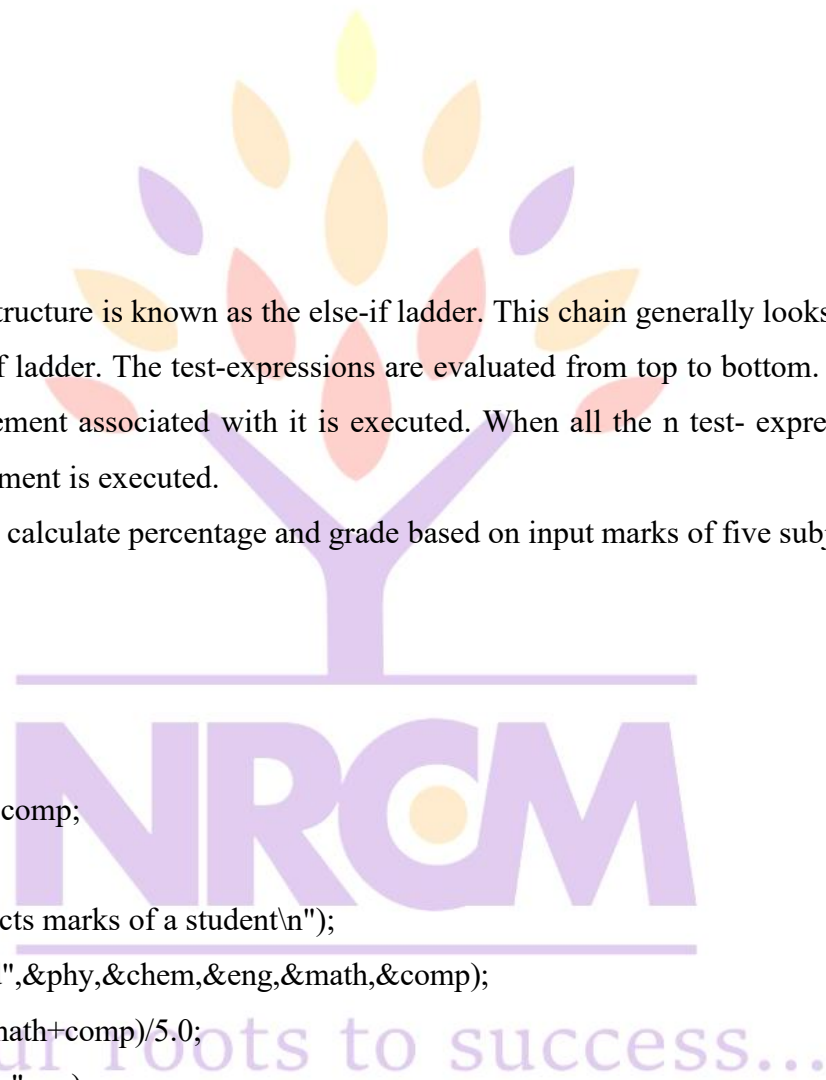
```
    }
```

```
    else if(per>=60)
```

```
    {
```

```
        printf("First Class");
```

```
    }
```



```

else if(per>=40)
{
    printf("Second Class");
}
else
{
    printf("Fail");
}
}

```

**5. switch statement:** If we want to select one statement from more number of statements then switch statement is used.

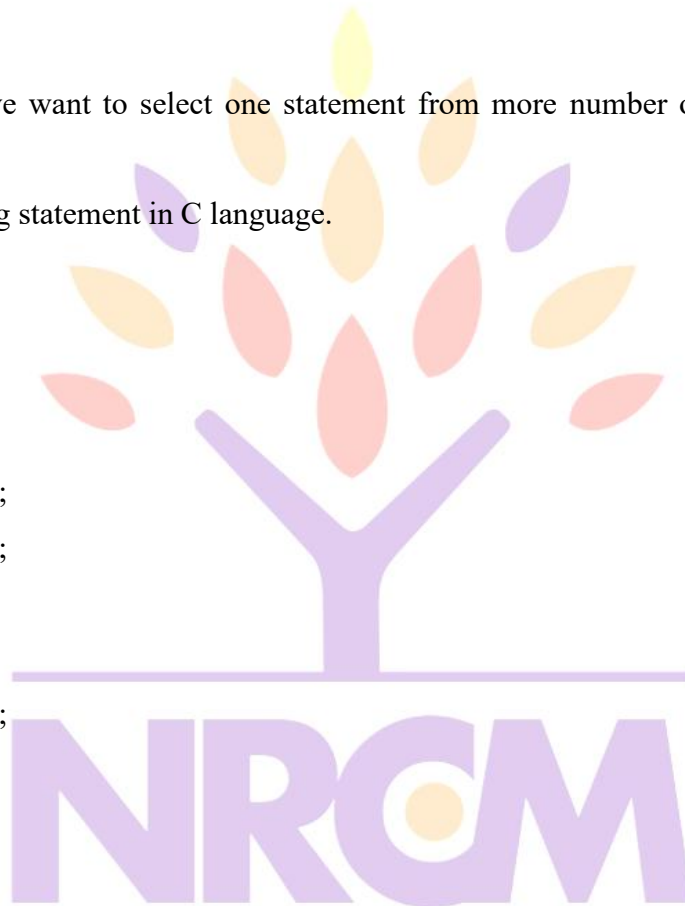
It is a multi way branching statement in C language.

**Syntax:**

```

switch(expression)
{
case value1:block1; break;
case value2:block2; break;
-----
-----
case valuen:blockn; break;
default :default block;
}
next statement;

```



**Operation:** First the expression value (integer constant/character constant) is compared with all the case values in the switch. if it is matched with any case value then the particular case block will be executed and then control is transferred to the next statement in the program.

If the expression value is not matched with any case value then default block will be executed and then control is transferred to the next statement in the program.

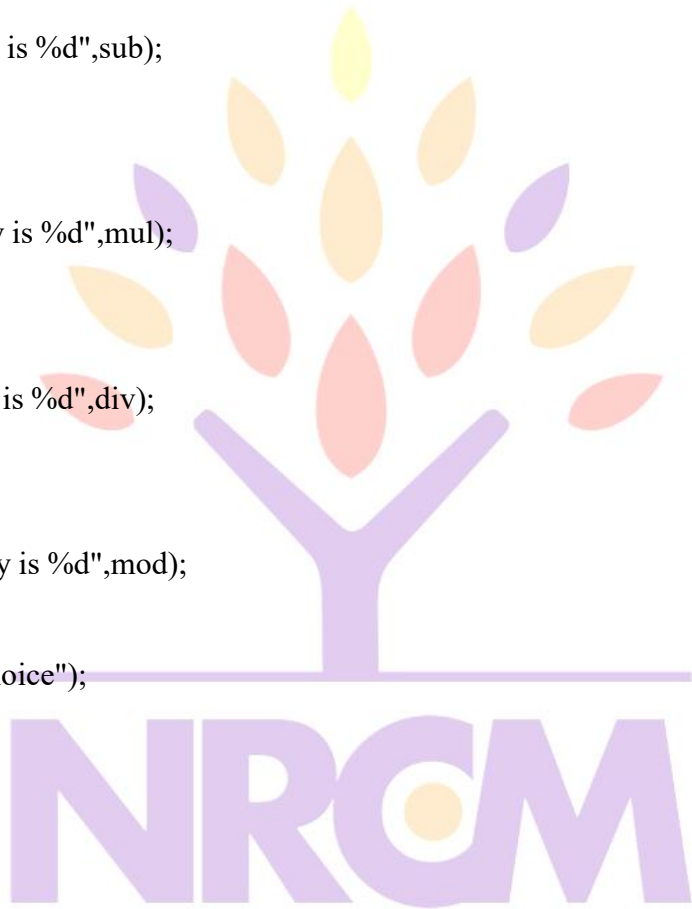
**Ex:** Write a C program to perform all arithmetic operations between 2 operands using switch statement.

```

#include<stdio.h>
int main()
{
int x,y,sum,sub,mul,div,mod,choice;
printf("Enter the value of x and y\n");

```

```
scanf("%d%d",&x,&y);
printf("Enter your choice 1.Addition\n2.Subtraction\n3.Product\n4.Quotient\n5.Remainder\n");
scanf("%d",&choice);
switch(choice)
{
case 1:sum=x+y;
printf("The sum of x and y is %d",sum);
break;
case 2:sub=x-y;
printf("The sub of x and y is %d",sub);
break;
case 3:mul=x*y;
printf("The mul of x and y is %d",mul);
break;
case 4:div=x/y;
printf("The div of x and y is %d",div);
break;
case 5:mod=x%y;
printf("The mod of x and y is %d",mod);
break;
default :printf("Invalid Choice");
break;
}
}
```



### Loops or Repetition statements:

There are 3 types of Loops/Repetition statements in C language. They are

14. while

15. do while

16. for

**1. while loop statement:** It is used when a group of statements are executed repeatedly until the specified condition is true.

It is also called entry controlled loop.

The minimum number of execution takes place in while loop is 0.

### Syntax:



```
while(condition)
{
body of while loop;
}
next statement;
```

Operation: First the condition is checked if the condition is true then control enters into the body of while loop to execute the statements repeatedly until the specified condition is true.

if the condition is false, the body of while loop is skipped and control comes out of the loop and continues with the next statement in the program.

**Ex:** Write a C program to print the numbers from 1 to 5 using while.

```
#include<stdio.h>
int main()
{
int i=1;
printf("The numbers from 1 to 5 are\n");
while(i<=5)
{
printf("%d\n ",i);
i++;
}
}
```

## 2. do while loop statement:-

It is used when a group of statements are executed repeatedly until the specified condition is true. It is also called exit controlled loop.

The minimum number of execution takes place in do while loop is 1.

### Syntax:-

```
do
{
body of do while loop;
}
while(condition);
next statement;
```

In do-while loop condition should be end with semicolon (;).

Operation: In do-while loop the condition is checked at the end i.e., the control first enters into the body of do while loop to execute the statements. After the execution of statements it checks for the condition. if the

condition is true then the control again enters into the body of do while loop to execute the statements repeatedly until the specified condition is true.

if the condition is false then control continues with the next statement in the program.

**Ex:** Write a C program to print the numbers from 1 to 5 using do while.

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

```
int main()
```

```
{
```

```
int i=1;
```

```
printf("The numbers from 1 to 5 are\n"); do
```

```
{
```

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

```
}
```

```
while(i<=5);
```

```
}
```

**3. for loop statement:-** It is used when a group of statements are executed repeatedly until the specified condition is true.

It is also called entry controlled loop.

The minimum number of execution takes place in for loop is 0.

**Syntax:-**

```
for (initialization;condition;increment/decrement )
```

```
{
```

```
body of for loop;
```

```
}
```

```
next statement;
```

**Operation:** First initial value will be assigned. Next condition is checked if the condition is true then control enters into the body of for loop to execute the statements. After the execution of statements the initial value will be incremented/decremented. After initial value will be incremented/decremented the control again checks for the condition. If the condition is true then the control is again enters into the body of for loop to execute the statements repeatedly until the specified condition is true.

if the condition is false, the body of for loop is skipped and control comes out of the loop and continues with the next statement in the program.

**Ex:** Write a C program to print the numbers from 1 to 5 using for.

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

```
int main()
{
int i;
printf("The numbers from 1 to 5 are\n");
for(i=1;i<=5;i++)
{
printf("%d\n",i);
}
}
```

### Unconditional statements:

Normal flow of control can be transferred from one place to another place in the program. This can be done by using the following unconditional statements in the C language.

There are 3 types of Unconditional statements in C language.

They are

17. goto

18. break

19. continue

#### **1. goto statement:**

goto is an unconditional statement used to transfer the control from one statement to another statement in the program.

#### **Syntax:-**

goto label;

label:

Statements;

: **Ex** Write a C program to check whether the given number is even or odd using goto statement.

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

```
int main()
```

```
{
```

```
int n;
```

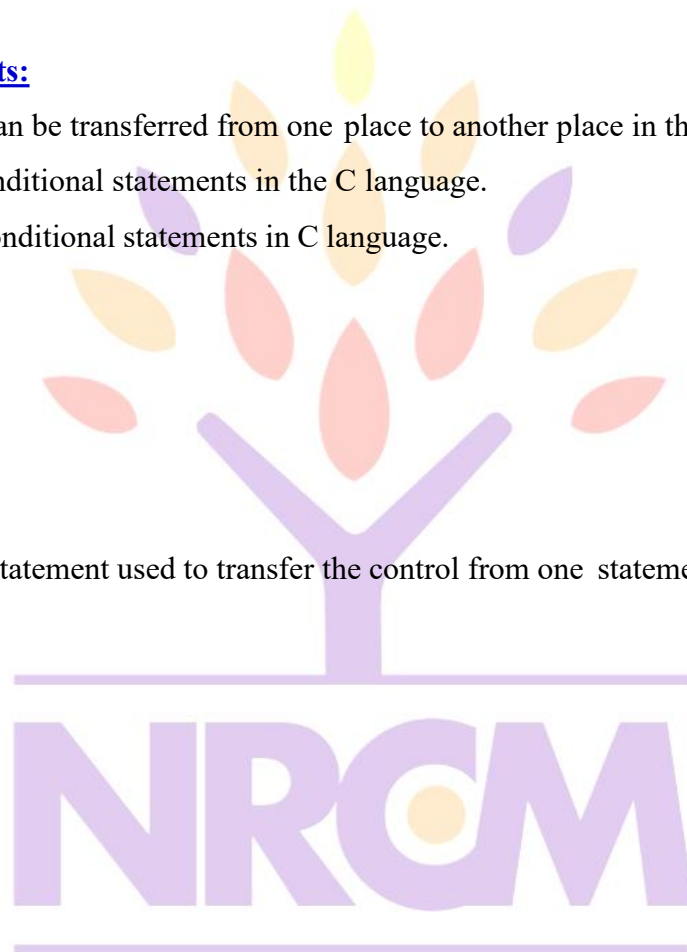
```
printf("Enter any number\n");
```

```
scanf("%d",&n);
```

```
if(n%2==0)
```

```
goto even;
```

```
else
```



```

goto odd;

even:

printf("The given number is even"); goto
end;

odd:

printf("The given number is odd");
goto end;

end:
}

```

## 2. break statement:

break is an unconditional statement used to terminate the loops or switch statement.

When it is used in loops (while,do while,for) control comes out of the loop and continues with the next statement in the program.

When it is used in switch statement to terminate the particular case block and then control is transferred to the next statement in the program.

**Syntax:-** statement; break;

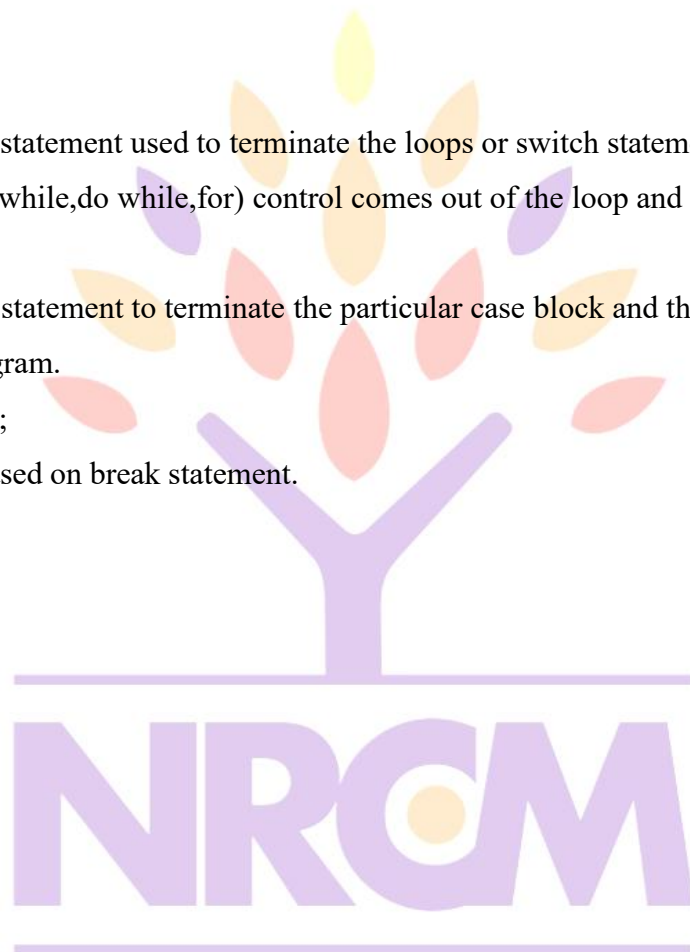
Ex: Write a C program based on break statement.

```

#include<stdio.h>

int main()
{
int i;
for(i=1;i<=5;i++)
{
if(i==4)
break;
printf("%d\t",i);
}
}

```



your roots to success...

3. **continue statement**:- continue is an unconditional statement used to control to be transferred to the beginning of the loop for the next iteration without executing the remaining statements in the program. **Syntax:-**

statement;

continue;

**Ex:** Write a C program based on continue statement.

```

#include<stdio.h>
int main()
{
int i;
for(i=1;i<=5;i++)

```

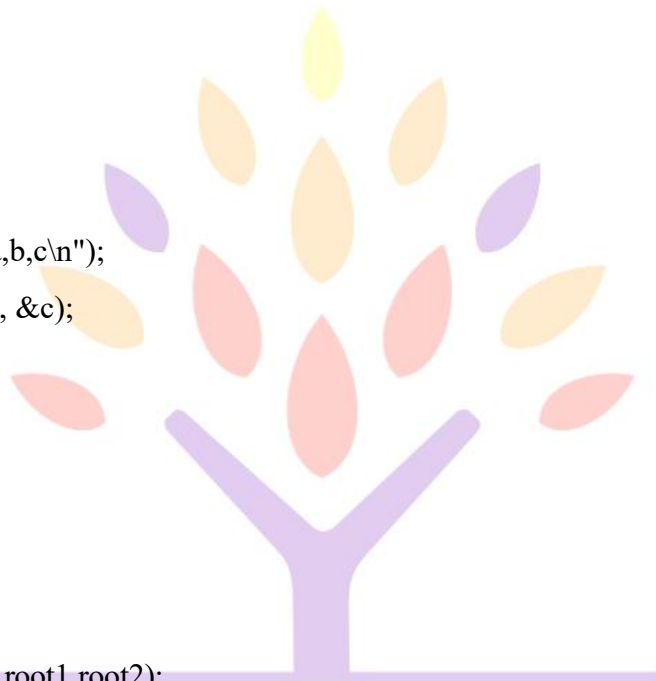
```
{
if(i==4)
continue;
printf("%d\t",i);
}
}
```

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

```
#include<math.h>
```

```
int main()
{
float a,b,c,d,root1,root2;
printf(" Enter the values of a,b,c\n");
scanf(" %f %f %f ", &a, &b, &c);
d= (b * b) - (4 * a * c);
if(d >=0)
{
root1=-b+sqrt(d)/(2*a);

root2=-b-sqrt(d)/(2*a);
printf("root1=%f,root2=%f",root1,root2);
}
else
printf("Roots are imaginary");
}
```



**NRCM**

your roots to success...

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

```
int main()
{
int n,i=1,sum=0;
printf("Enter any number\n");
scanf("%d",&n);
while(i<=n)
{
sum=sum+i;
i++;
}
```

```

}
printf("The sum of n natural numbers is %d",sum);
}

```

```

#include<stdio.h>

int main()
{
int digit,n,sum=0; printf("Enter any number\n");
scanf("%d",&n);
while(n!=0)
{
digit=n%10;
sum=sum+digit;
n=n/10;
}
printf("The sum of individual digits of a given number is %d",sum);
}

```

```

#include<stdio.h> int
main()
{
int digit,n,reverse=0; printf("Enter
any number\n"); scanf("%d",&n);
while(n!=0)
{
digit=n%10;
reverse=reverse*10+digit;
n=n/10;
}
printf("The reverse of a given number is %d",reverse);
}

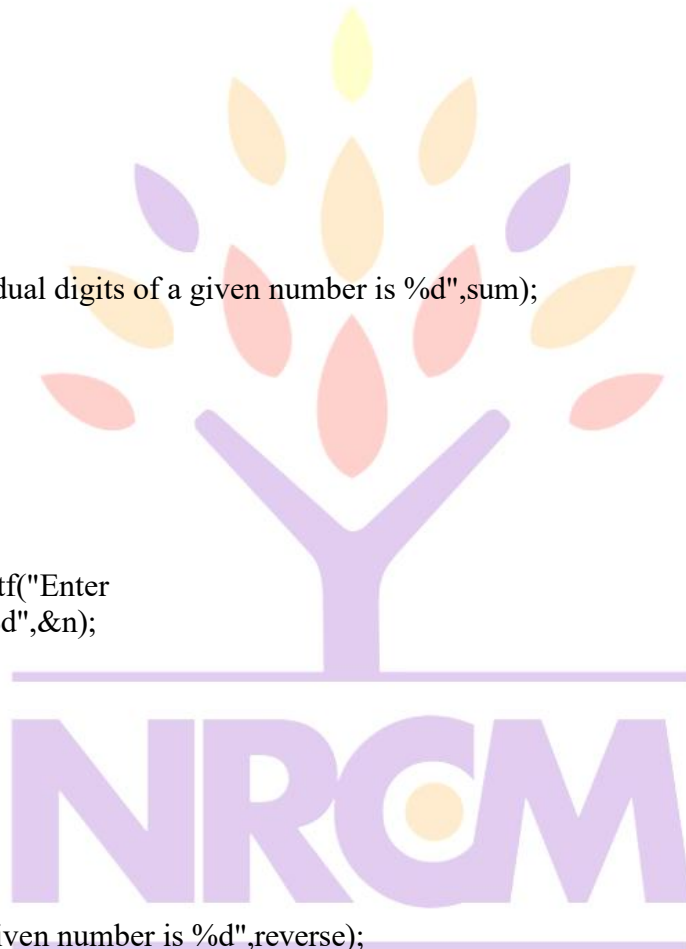
```

```

#include<stdio.h>

int main()
{
int digit,n,reverse=0,temp; printf("Enter
any number\n"); scanf("%d",&n);
temp=n; while(n!=0)
{
digit=n%10;
reverse=reverse*10+digit;
n=n/10;
}
if(temp==reverse)

```



```
printf("The given number is Palindrome");
else
printf("The given number is not a Palindrome");
}
```

```
#include<stdio.h>

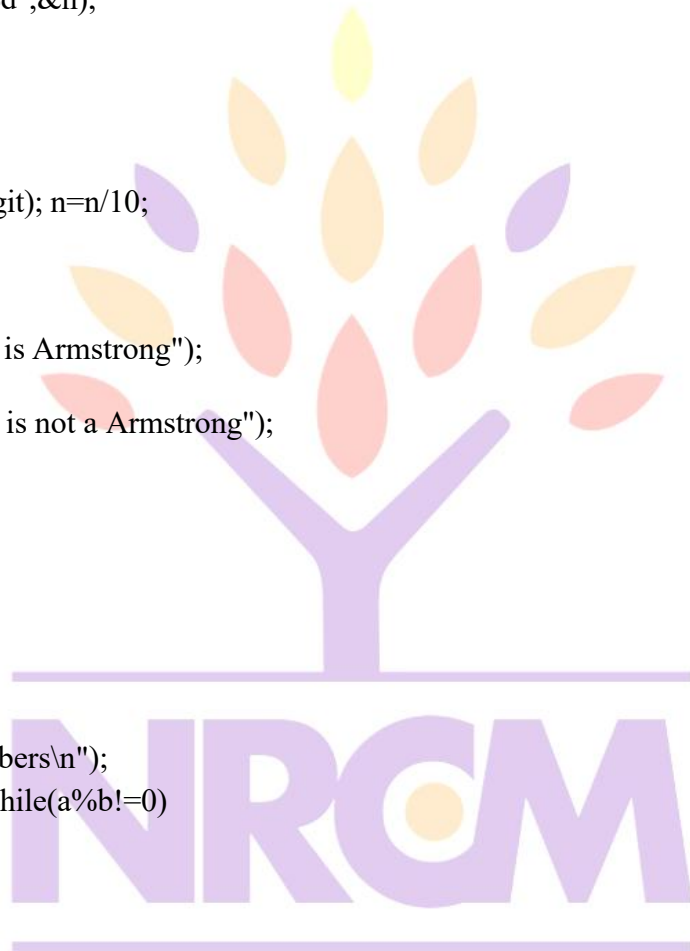
int main()
{
int digit,n,sum=0,temp; printf("Enter
any number\n"); scanf("%d",&n);
temp=n;
while(n!=0)
{
digit=n%10;
sum=sum+(digit*digit*digit); n=n/10;
}
if(temp==sum)
printf("The given number is Armstrong");
else
printf("The given number is not a Armstrong");
}
```

```
#include<stdio.h>
int main()
{
int a,b,c;
printf("Enter the two numbers\n");
scanf("%d%d",&a,&b); while(a%b!=0)
{
c=a%b;
a=b;
b=c;
}
printf("The GCD of a,b is %d",b);
}
```

```
#include <stdio.h>

int main()
{
int i,n,first=0,second=1,nextterm;

printf("Enter the number of terms\n"); scanf("%d",&n);
printf("Fibonacci Series is\n ");
for (i=1;i<=n;i++)
{
```



your roots to success...

```
printf("%d\t",first);
nextterm=first+second;
first=second; second=nextterm;
}
}
```

### stdin, stdout and stderr:

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

There are 3 special FILE pointers that are always defined for a program. They are stdin (standard input), stdout (standard output) and stderr (standard error).

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Your screen

#### 1. Standard Input(stdin)

Standard input is where things come from when you use scanf(). In other words, scanf("%d", &val); is equivalent to the following fscanf():

```
fscanf(stdin, "%d",&val);
```

#### 2. Standard Output(stdout)

Similarly, standard output is exactly where things go when you use printf(). In other words, printf("Value = %d\n", val):

is equivalent to the following fprintf():

```
fprintf(stdout, "Value = %d\n", val);
```

#### 3. Standard Error(stderr)

Standard error is where you should display error messages. We have already done that above: fprintf(stderr, "Can't open input file in the list!\n");

Standard error is normally associated with the same place as standard output.

### UNIT-2: Arrays, Strings, Structures and Pointers



**Array:**

1. Array is a collection of elements of the same data type.
2. Arrays can be used to store multiple values.
3. Arrays are the derived data type in C which can store the primary type of data such as int, char, double, float, etc.
4. Array range starts from 0 to n-1.
5. Array elements are stored in contiguous memory locations or consecutive memory locations or successive memory locations.

**1. One Dimensional arrays (1DA) :-**

If an array contains one subscript then it is called one dimensional array.

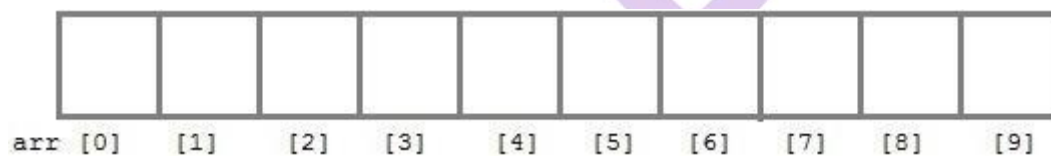
**Declaration/Syntax of 1DA:**

```
datatype arrayname[size];
```

In the above syntax data type maybe any basic data type such as int,float, etc. arrayname is any valid identifier.

size indicates number of elements can be stored in an array name.

**Example:**                   int arr[10];



Here int is the data type, arr is the name of the array and 10 is the size of array. It means array arr can only contain 10 elements of int type. Index of an array starts from 0 to size-1 i.e first element of arr array will be stored at arr[0] address and last element will occupy arr[9].

**Initialization (or) Assigning values to one dimensional arrays (1DA) :****Syntax :**

```
datatype arrayname[size] = { value1, value2,....., value n};
```

**Example:**

```
int x[5]={10, 20, 30, 40, 50};
```

In the above example five integer elements can be stored in the array name 'x'

Here x[0] refers to the 1<sup>st</sup> element stored in the array i.e.10 x[1]

refers to the 2<sup>nd</sup> element stored in the array i.e.20

x[2] refers to the 3<sup>rd</sup> element stored in the array i.e.30

x[3] refers to the 4<sup>th</sup> element stored in the array i.e.40

x[4] refers to the 5<sup>th</sup> element stored in the array i.e.50

**Example:** Program to Reading and Storing 5 elements in one dimensional array.

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

```
int main()
```

```
{
```

```
int a[10],i;
```

```
printf("Enter 10 integers\n");
```

```
for(i=0;i<10;i++)
```

```
{
```

```
scanf("%d",&a[i]);
```

```
}
```

```
printf("Displaying integers\n");
```

```
for(i=0;i<10;i++)
```

```
{
```

```
printf("%d\n",a[i]);
```

```
}
```

```
}
```

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

```
int main()
```

```
{
```

```
int x[5],i,large,small;
```

```
printf(" Enter any five integer array elements\n");
```

```
for(i=0;i<5;i++)
```

```
{
```

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

```
}
```

```
large=x[0];
```

```
small=x[0];
```

```
for(i=1;i<5;i++)
```

```
{
```

```
if(x[i]>large)
```

```
large=x[i];
```

```
if(x[i]<small)
```

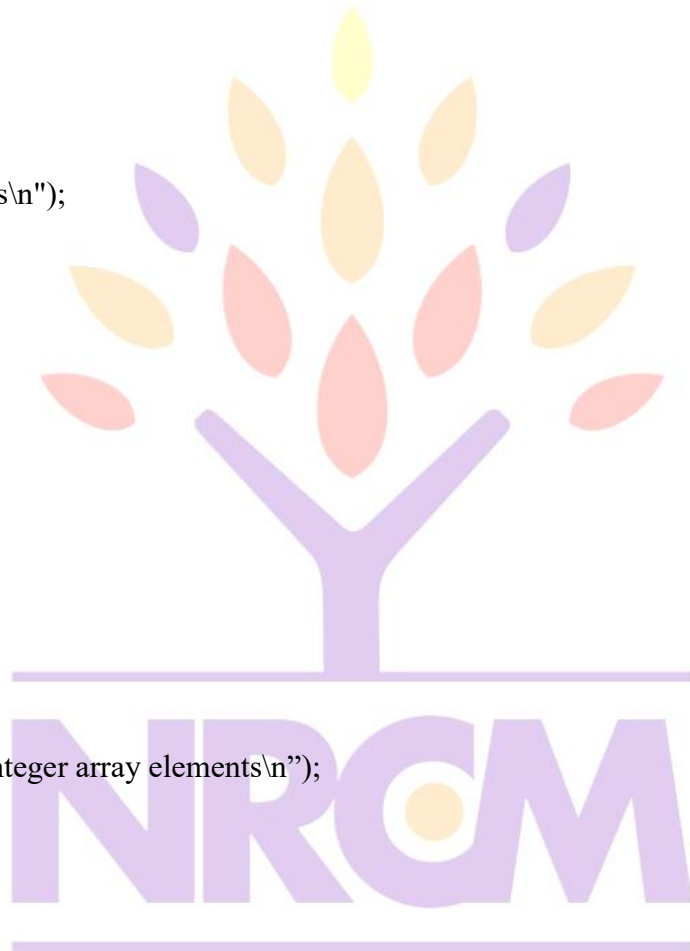
```
small=x[i];
```

```
}
```

```
printf("The largest element from the given array is %d",large);
```

```
printf("The smallest element from the given array is %d",small);
```

```
}
```



your roots to success...

```
#include<stdio.h>

int main ()
{
    int a[100],i,j,n,temp; printf("Enter
    size of an array\n"); scanf("%d",&n);
    printf("Enter the array elements\n"); for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    for(i=0;i<n;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(a[i]>a[j])
            {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
        }
    }
    printf("After sorting array elements are\n");
    for(i=0;i<n;i++)
    {
        printf("%d\t",a[i]);
    }
}
```



your roots to success...

### **Two Dimensional Arrays (2DA):**

If an array contains two subscripts then it is called two dimensional arrays. It is also called multi dimensional arrays.

Two dimensional array elements are stored in consecutive memory locations.

Two dimensional arrays are mainly used for performing matrix operations like matrix addition,

subtraction,multiplication,transpose,etc.

### Declaration/Syntax of Two Dimensional Arrays (2DA):

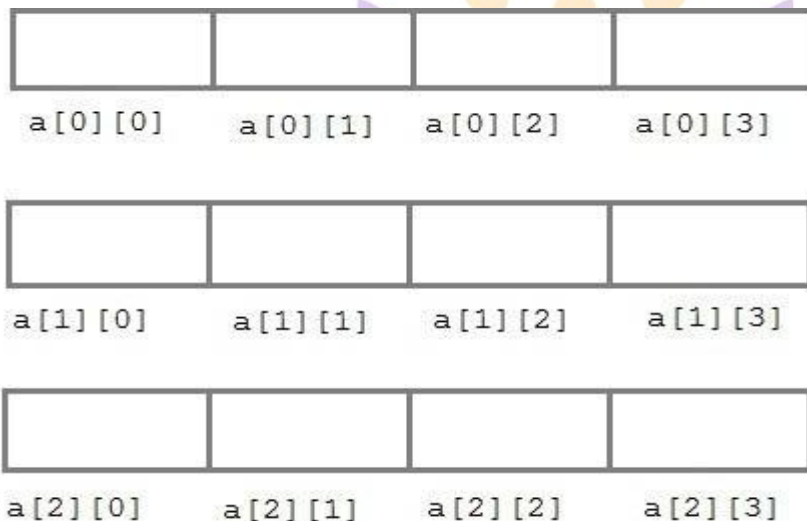
```
datatype arrayname[size1][size2];
```

In the above syntax data type maybe any basic data type such as int, float, etc. arrayname is any valid identifier.

size1 indicates no. of rows and size2 indicates no. of columns stored in an arrayname.

### Example:

```
int a [3][4];
```



### Initialization(or)Assigning values to two dimensional arrays (2DA) :

#### Syntax :

```
datatype arrayname[size1][size2] = { value1, value2,....., value n};
```

#### Example:

```
int x[2][2] = {10,20,30,40};
```

In the above example 2 rows and 2 columns can be stored in the array name 'x'

Here `x[0][0]` refers to the 1<sup>st</sup> element stored in the array i.e.10

`x[0][1]` refers to the 2<sup>nd</sup> element stored in the array i.e.20 `x[1][0]`

refers to the 3<sup>rd</sup> element stored in the array i.e.30 `x[1][1]` refers to

the 4<sup>th</sup> element stored in the array i.e.40

**Example:** Program to Reading and Storing elements in a matrix and printing it. `#include<stdio.h>`

```
int main()
```

```

{
int a[3][3],i,j;
printf("Enter the elements of matrix a\n");
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
scanf("%d",&a[i][j]);
}
}
printf("Printing the elements\n");
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
printf("%d\t",a[i][j]);
}
printf("\n");
}
}

```

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

```
int main()
{
```

```
int A[3][3],Transpose[3][3],i,j;
```

```
printf("Enter the values of elements of matrix A\n"); for(i=0;i<3;i++)
```

```
{
```

```
for(j=0;j<3;j++)
```

```
{
```

```
scanf("%d",&A[i][j]);
```

```
}
```

```
}
```

```
for(i=0;i<3;i++)
```

```
{
```

```
for(j=0;j<3;j++)
```

```
{
```



your roots to success...

```

Transpose[i][j]=A[j][i];
}
}
printf("The Transpose of matrix A is\n"); for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
printf("%d\t",Transpose[i][j]);
}
printf("\n");
}
}

```

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

```
int main()
```

```
{
```

```
int A[3][3], B[3][3], C[3][3],i,j,k;
```

```
printf("Enter the elements of matrix A\n");
```

```
for(i=0;i<3;i++)
```

```
{
```

```
for(j=0;j<3;j++)
```

```
{
```

```
scanf("%d",&A[i][j]);
```

```
}
```

```
}
```

```
printf("Enter the elements of matrix B\n");
```

```
for(i=0;i<3;i++)
```

```
{
```

```
for(j=0;j<3;j++)
```

```
{
```

your roots to success...

```
scanf("%d",&B[i][j]);
```

```
}
```

```
}
```

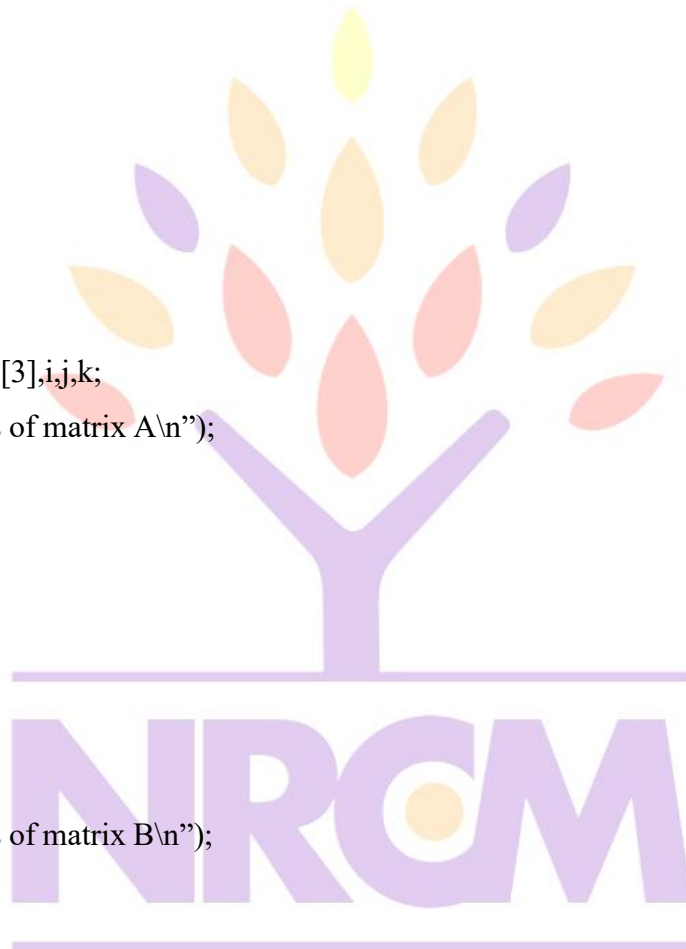
```
for(i =0;i <3;i++)
```

```
{
```

```
for(j= 0;j <3;j++)
```

```
{
```

```
C[i][j]=0;
```



```

for(k= 0;k<3; k++ )
{ C[i][j]=C[i][j]+(A[i][k]*B[k][j])
;
}
}
}

printf("The result in C matrix is \n");
for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
printf("%d\t",C[i][j]);
}
printf("\n");
}
}

```

### String:

1. String is a group of characters enclosed within double quotation marks.
2. String is also called array of characters or character arrays.
3. Character Array elements are stored in contiguous memory locations or consecutive memory locations or successive memory locations.
4. The string should end with '\0' (null character) in C language.
5. The size of the string is equal to no.of characters in the string + 1 (required for null character('\0')).

### One Dimensional Character arrays (1DA)/One Dimensional Strings :-

#### Declaration/Syntax of strings/character arrays:

```
datatype arrayname[size];
```

In the above syntax data type must be 'char' data type. array name is any valid identifier

size indicates no.of characters in the string + 1 (required for null character('\0')).

**Example:** char name[20];

#### Initialization of strings/character arrays:

**Syntax:** datatype arrayname[size]={character1,character2,...,character n};

#### **Example:**

```
char city[6]="DELHI";
```

(or)

```
char city[6]={'D','E','L','H','I','\0'};
```

city[0] refers to 1<sup>st</sup> character in string i.e. D city[1]

refers to 2<sup>nd</sup> character in string i.e. E city[2] refers

to 3<sup>rd</sup> character in string i.e. L city[3] refers to

4<sup>th</sup> character in string i.e. H city[4] refers to

5<sup>th</sup> character in string i.e. I city[5] refers to

6<sup>th</sup> character in string i.e. '\0'.

### **Example:**

```
#include<stdio.h>
int main()
{
    char name[50];
    printf("Enter your name\n "); gets(name);
    printf("Your name is\n ");
    puts(name);
}
```

### **String input and output functions in C:**

#### **1. gets():**

gets() is a input function in strings and it is used to read a string from the user. **Syntax:**

```
gets(arrayname);
```

#### **2. puts():**

puts() is a output function in strings used to print the string on the monitor. **Syntax:**

```
puts(arrayname);
```

### **Example Program:**

```
#include <stdio.h> int
main()
{
    char name[30]; printf("Enter
any name:"); gets(name); //
read string printf("Name: ");
puts(name); // display string
}
```

### **String Manipulation Functions/String Handling Functions:**



All the string handling functions are pre-defined in the system library i.e., in the header file #

include<string.h>

The following are commonly used string handling functions in C are

6. strcpy( )
7. strcat( )
8. strrev( )
9. strlen( )
10. strlwr( )
- 11.strupr( )
12. strcmp( )

### 1. strcpy():-

This function is used to copy one string into another string.

**Syntax:** strcpy(string1,string 2);

Here, string 2 is copied into string 1 and after copying both the contents of string 1 and string 2 are same.

**Example:** C program to copy one string into another string using strcpy().

```
#include<stdio.h>    #include<conio.h>
#include<string.h> int main()
{
char name1[20],name2[20]; printf("Enter
the first name\n"); gets(name1);
printf("Enter the second name\n");
gets(name2); strcpy(name1,name2);
printf("After copying the name1=%s,name2=%s",name1,name2);
}
```

### 2. strcat():-

This function is used to combine two strings.

**Syntax:** strcat (string1,string2);

Here, string2 is added to the end of the string1.

**Example :** C program to combine two strings using strcat().

```
#include<stdio.h>
#include<string.h>
int main()
{
char name1[20],name2[20]; printf("Enter
the first name\n"); gets(name1);
```

```
printf("Enter the second name\n");
gets(name2); strcat(name1,name2);
printf("After concatenation the name is %s",name1);
}
```

### 3.strrev() :-

This function is used to reverse a given string.

Syntax: strrev(string);

**Example:** C program to find the reverse of given string using strrev().

```
#include<stdio.h>
#include<string.h>
int main()
{
char name[20];
printf("Enter any name\n"); gets(name);
strrev(name);
printf("The reverse of a given name is %s",name);
}
```

### 4.strlen():-

This function is used to find the length of the given string.

This function does not include the '\0'(null character).

**Syntax:** strlen(string);

**Example:** C program to find the length of a given string using strlen().

```
#include<stdio.h>
#include<string.h>
int main()
{
char name[20]; int
x;
printf("Enter any name\n");
gets(name); x=strlen(name);
printf("The length of a given string is %d",x);
}
```

### 5.strlwr():-

This function is used to convert any uppercase letters into lowercase letters. **Syntax:**

```
strlwr(string);
```

**Example:** C program to convert any uppercase letters into lowercase letters using strlwr().

```
#include<stdio.h>
#include<string.h> int
main()
{
char name [20];
printf("Enter any name\n"); gets(name);
strlwr(name);
printf("The converted name is %s",name);
}
```

### 6.strupr():-

This function is used to convert any lowercase letters into uppercase letters. **Syntax:**

```
strupr(string);
```

**Example:** C program to convert any lowercase letters into letters uppercase usingstrupr(). #include<stdio.h>

```
#include<string.h> int
main()
{
char name [20];
printf("Enter any name\n"); gets(name);
strupr(name);
printf("The converted name is %s",name);
}
```

### 7.strcmp():-

This function is used to compare two strings character by character based on ASCII values (American Standard Code for Information Interchange) and returns zero when both the strings are equal.

**Syntax:** strcmp(string1,string2);

**Example :** C program to check whether the two strings are equal or not.

```
#include<stdio.h>
#include<string.h>
int main()
{
char name1[20],name2[20]; printf("Enter
```

```

the first name\n"); gets(name1);
printf("Enter the second name\n");
gets(name2); if(strcmp(name1,name2)=
=0)
printf("Both the given strings are equal");
else
printf("Both the given strings are not equal");
}

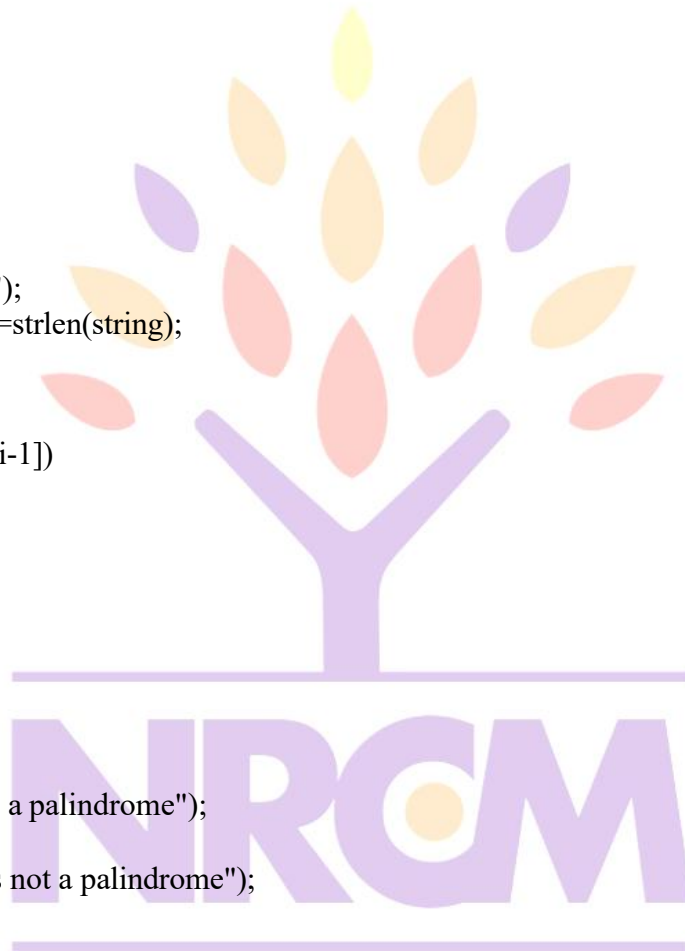
```

```

#include<stdio.h>
#include<string.h>

int main()
{
char string[20]; int
i,length,flag=1;
printf("Enter any string\n");
scanf("%s",string); length=strlen(string);
for(i=0;i<length;i++)
{
if(string[i]!=string[length-i-1])
{
flag=0;
break;
}
}
if(flag==1)
printf("The given string is a palindrome");
else
printf("The given string is not a palindrome");
}

```



### Two dimensional strings/Array of strings/ Two dimensional character arrays:

Using one dimensional string only a single string was accepted and processed. Sometimes, it becomes necessary to process a group of strings. In such situations we need a two dimensional character arrays. **Declaration of Two dimensional strings:**

#### **Syntax:**

```
datatype arrayname[size1][size2];
```

In the above syntax data type must be 'char' data type. array name is any valid identifier.

size1 indicates no. of strings and size2 indicates no. of characters of each string. **Example:**

```
char name[5][20];
```

**Initialization of Two dimensional strings:**

**Syntax:** datatype arrayname[size1][size2]={string1,string2,...,string n};

**Example:**

```
char name[5][20]={“dileep”,“sravani”,“mehta”,“likith”,“siddarath”};
```

name[0] refers to 1<sup>st</sup> string i.e.dileep name[1]

refers to 2<sup>nd</sup> string i.e.sravani name[2] refers

to 3<sup>rd</sup> string i.e.mehta name[3] refers to 4<sup>th</sup>

string i.e.likith name[4] refers to 5<sup>th</sup> string

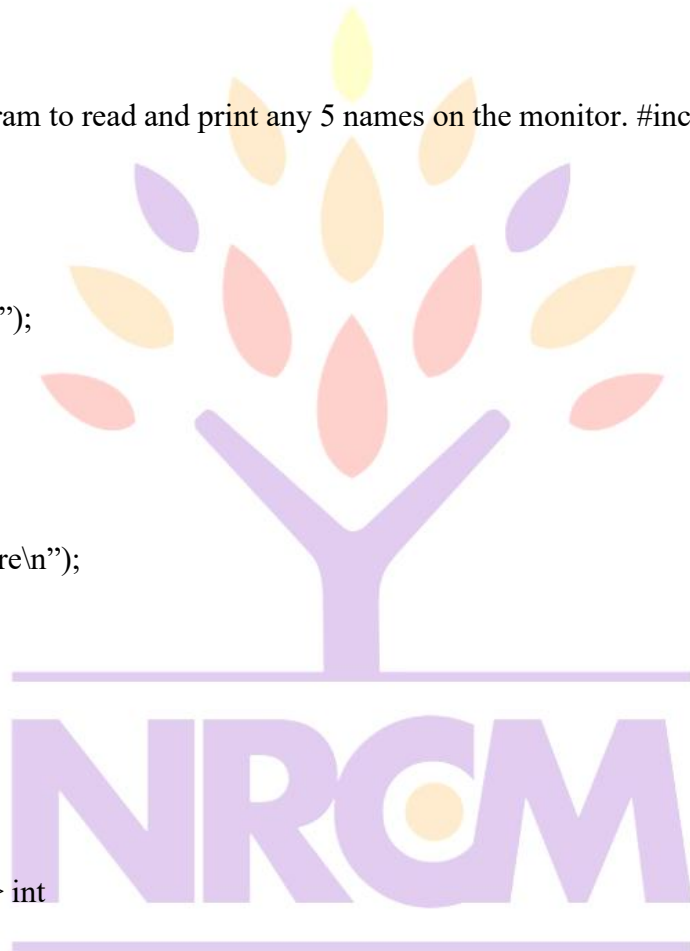
i.e.siddarath

**Example:** Write a c program to read and print any 5 names on the monitor. #include<stdio.h>

```
int main()
{
char name[5][20]; int
i;
printf(“Enter any 5 names”);
for(i=0;i<5;i++)
{
scanf(“%s”,name[i]);
}
printf(“Entered 5 names are\n”);
for(i=0;i<5;i++)
{
printf(“%s\n”,name[i]);
}
}
```

```
#include<stdio.h>
#include<string.h> int
main()
{
```

```
char name[10][20],temp[20];
int n,i,j;
printf(“Enter the number of strings\n”);
scanf(“%d",&n);
printf(“Enter the strings\n”);
for(i=0;i<n;i++)
{
scanf(“%s”,name[i]);
}
for(i=0;i<n-1;i++)
{
for(j=0;j<n-1;j++)
```



```

{
if(strcmp(name[j],name[j+1])>0)
{
strcpy(temp,name[j]);
strcpy(name[j],name[j+1]);
strcpy(name[j+1],temp);
}
}
}

printf("The sorted order of strings are\n") ;
for(i=0;i<n;i++)
{
printf("%s\n",name[i]);
}
}

```

**Pointer:**

Pointer is a variable that stores the address of another variable. Pointer is used to allocate the memory dynamically at the run time.

**Declaration of a pointer:**

```
datatype *variable;
```

**Example1:** int \*p;

Here p is a pointer variable which always points to integer data type.

**Example2:** float \*p;

Here p is a pointer variable which always points to real or floating point data type.

**Example3:** char \*p;

Here p is a pointer variable which always points to character data type.

**Initialization of a pointer:**

The way of allocating the address of a variable to a pointer variable is known as pointer initialization.

**Syntax:**

```
pointerVariableName = &variableName ;
```

**Example:**

```
int *p,x;
```

```
p=&x;
```

**Reference operator (&):**

& is called reference operator/address operator which specifies where the value would be stored. **Dereference**

### **operator/Indirection operator (\*):**

\* is called indirection operator or dereferencing operator which indicates value at address.

**Example:** C program to print the values and its corresponding addresses using pointers. #include<stdio.h>

```
main()
{
int *p,x=10;
p=&x;
printf("the address of p is %u\n",&p);
printf("the address of x is %u\n",&x);
printf("the value of p is %u\n",p);
printf("the value of x is %d\n",x);
printf("the value at p is %d\n",*p);
printf("the value at x is %d\n",*(&x));
}
```

### **Pointers and Arrays:**

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. The base address is the location of the first element (index 0) of the array.

### **Declaration of pointers and arrays:**

**Syntax:** datatype arrayname[size],\*variable;

**Example:** Suppose we declare an array arr, int

```
arr[5]={ 1, 2, 3, 4, 5 };
```

Assuming that the base address of arr is 1000 and each integer requires two bytes, the five elements will be stored as follows

element	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
Address	1000	1002	1004	1006	1008

Here variable arr will give the base address, which is a constant pointer pointing to the element, arr[0]. Therefore arr is containing the address of arr[0] i.e 1000.

We can declare a pointer of type int to point to the array arr.

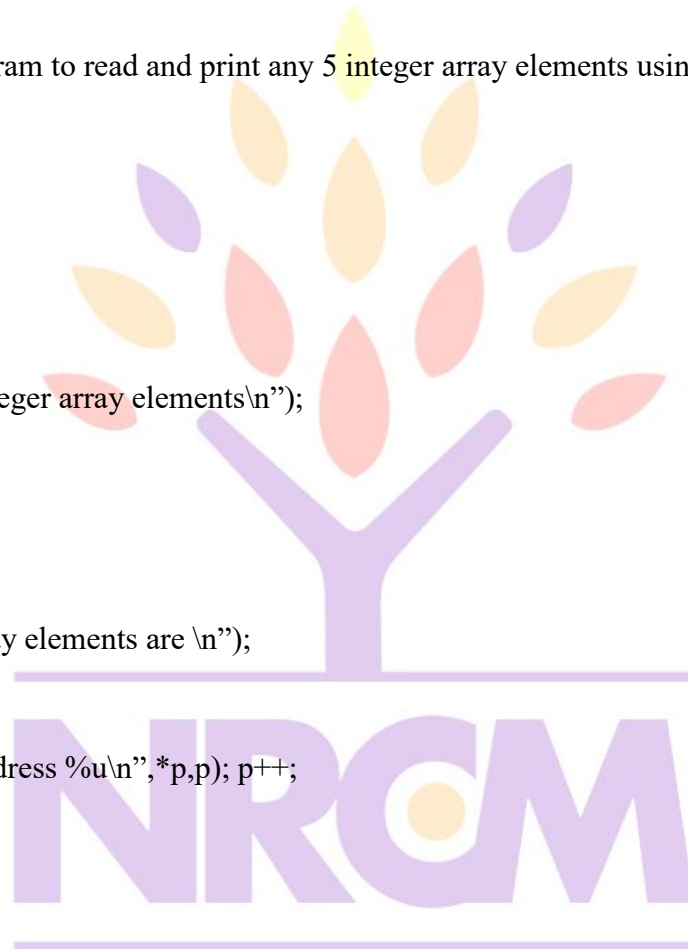
```
int *p;
p = &arr[0];
```

Now we can access every element of array arr using p++ to move from one element to another.

**Example:** Write a c program to read and print any 5 integer array elements using pointers.

```
#include<stdio.h>
int main()
{
int x[5],*p,i;
p=&x[0];
printf(" Enter any five integer array elements\n");
for(i=0;i<5;i++)
{
scanf("%d",&x[i]);
}
printf(" The 5 integer array elements are \n");
for(i=0;i<5;i++)
{
printf("%d is stored at address %u\n",*p,p); p++;
}
}
```

```
#include<stdio.h> int
main()
{
int x[5],*p,i,sum=0;
p=&x[0];
printf(" Enter any 5 integer array elements\n");
for(i=0;i<5;i++)
{
scanf("%d",&x[i]);
}
for(i=0;i<5;i++)
{
sum=sum+*p;
p++;
}
```



your roots to success...



```

}
printf(" The sum of array elements using pointers is %d",sum);
}

```

### Pointers to Functions:

Pointers can be passed as arguments to a called function.

Passing pointers to a function is called call by reference/call by address.

Pointer arguments are useful in functions because they allow to accessing the original data in the calling function.

### Example:

```

#include<stdio.h>
int swap(int *x, int *y);
int main()
{
int a =10,b=20;
printf("Before swapping a=%d, b=%d \n",a,b);
swap(&a,&b);
printf(" After swapping a=%d, b=%d",a,b);
}
int swap(int *x, int *y)
{
int temp;
temp = *x;
*x = *y;
*y = temp;
}

```



### Pointers to Structures:

A pointer variable which points to a structure is called structure pointers/pointers to structures. **Syntax:**

```
struct tagname *variable;
```

### Selection operator/arrow operator (->)

To access the each member of a structure using pointer variable we have use arrow operator ( -> ) in c language.

### Syntax:

Pointer variable ->structure member

**Example:** Define a structure type student that would contain roll number,name,branch and marks.Write a c program to read this information from keyboard and print the same on monitor using structure pointer or ( -> )operator.

### Example:

```

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

```

```

char branch[20];
float marks;
};
int main()
{
struct student s,*p;

p=&s;
printf("Enter the student rno,name,branch and marks\n");
scanf("%d%s%s%f",&s.rno,s.name,s.branch,&s.marks);
printf("%d\t%s\t%s\t%f",p->rno,p->name,p->branch,p->marks);
}

```

### Self referential structures:

A structure which contains atleast one member as a pointer to the same structure is known as Self referential structures.

Self referential structures are mainly used in the implementation of data structures like linked lists and trees.

### **Example:**

```

struct student
{
char name[20]; int
rollno;
struct student *ptr;
};

```

### enum(enumerated data type):

enum is an user defined data type and it is used to create a new data type and declaring enumeration types.

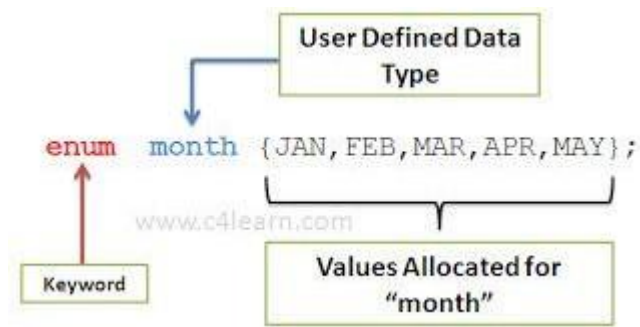
### Syntax:

```
enum enumerartiontype { identifier1,identifier2,.....,identifier n };
```

**Example:** Consider 12 months of a year.

```
enum month { jan,feb,mar,.....dec };
```

In the above example month is a user defined data type or enumeration type and jan,feb,.....dec are the Identifiers and their values starts from zero(0) .so jan refers to 0,feb refers to 1,mar refers to 2,...dec refers to 11.

**Example:**

```
enum year { Jan, Feb, Mar, Apr, May, Jun, Jul,
           Aug, Sep, Oct, Nov, Dec};
int main()
{
    int i;
    for(i=Jan; i<=Dec; i++)
    {
        printf("%d ", i);
    }
}
```

**Structure:**

Structure is a collection of elements of different data types.

**Declaration of a Structure:-****Syntax:**

```
struct tagname
{
```

```
    datatype member1;
    datatype member2;
    datatype member3;
    .
    .
    datatype member n;
```

```
};
```

In the above syntax struct is a keyword that declares/stores the members or fields of a structure. tag name is name of the structure.

datatype maybe any basic datatype such as int,float,char etc.

member1,member2,.....,member n are structure members or fields of a structure.

The body of a structure should be end with semicolon(;). **Example:**

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

### **Declaration of a structure variable:-**

**Syntax :-**  
struct tagname variable;

**Example:**  
struct student s;

### **Initialization of a structure variable :-**

**Syntax:-**  
struct tagname variable={member1,member2,...member n}; **Example:**

```
struct student s={501,"Dinesh","cse",100.5};
```

### **Member operator or dot operator(.):-**

To access the each member of a structure using the structure variable we have to use dot operator ( . ) in c language.

**Syntax :-**

Structure variable.Structure member

### **Example Program:**

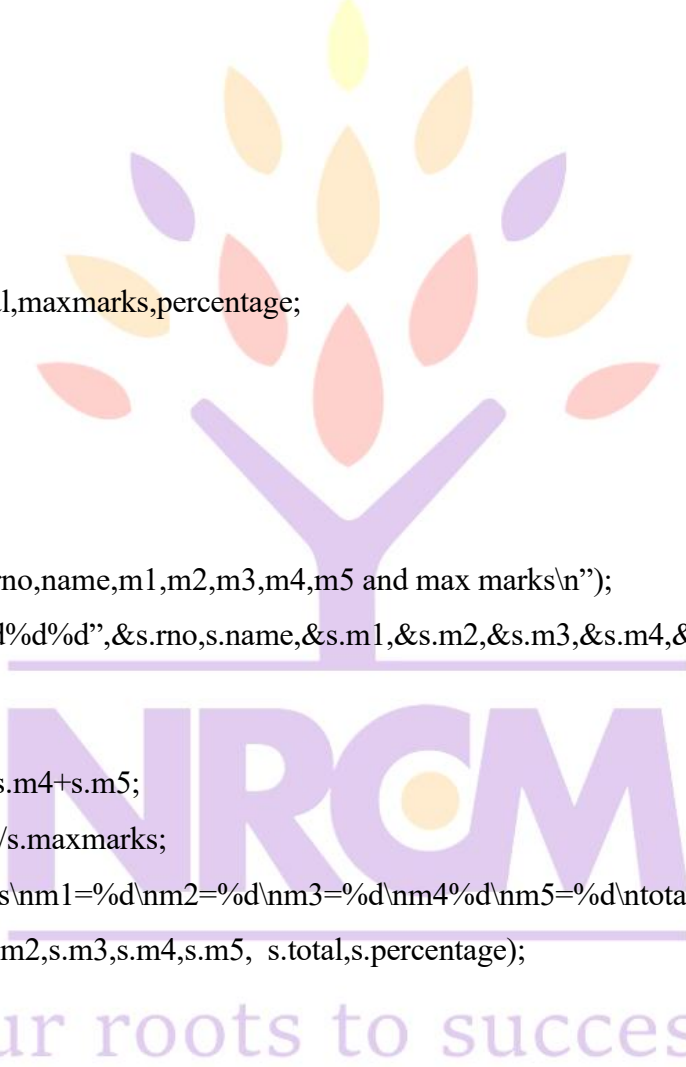
Example: Define a structure type student that would contain roll number,name,branch and marks. Write a C program to read this information from the keyboard and print the same on the monitor.

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

```
int main()
{
struct student s;
printf("Enter the student rno,name,branch and marks\n");
scanf("%d%s%s%f",&s.rno,s.name,s.branch,&s.marks);
printf("the student details are\n");
printf("%d\t%s\t%s\t%f",s.rno,s.name,s.branch,s.marks);
}
```

```
#include<stdio.h> struct
student
{
int rno;
char name[20];
int m1,m2,m3,m4,m5,total,maxmarks,percentage;
};
int main()
{
struct student s;
printf("Enter the student rno,name,m1,m2,m3,m4,m5 and max marks\n");
scanf("%d%s%d%d%d%d%d",&s.rno,s.name,&s.m1,&s.m2,&s.m3,&s.m4,&s.m5,&s.maxmarks);

s.total=s.m1+s.m2+s.m3+s.m4+s.m5;
s.percentage=(s.total*100)/s.maxmarks;
printf("rno=%d\nname=%s\nm1=%d\nm2=%d\nm3=%d\nm4=%d\nm5=%d\ntotal=%d\npercentage
=%d",s.rno,s.name,s.m1,s.m2,s.m3,s.m4,s.m5, s.total,s.percentage);
}
```



NRCM  
your roots to success...

### Array of Structures:

The same structure is applied to a group of people or group of items then array of structures is used. An array of structures in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.

### Syntax:

```
struct tagname arrayname[size];
```

**Example:** Define a structure type student that would contain roll number,name,branch and marks. Write a C program to read this information for 5 students from keyboard and print the same on monitor.

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

int main()
{
struct student s[5];
int i;
printf("Enter any 5 student details \n"); for(i=0;i<5;i++)
{
printf("Enter the student rno,name,branch,marks\n");
scanf("%d%s%s%f",&s[i].rno,s[i].name,s[i].branch,&s[i].marks);
}
printf("the 5 students details are\n");
for(i=0;i<5;i++)
{
printf("%d\t%s\t%s\t%f\n",s[i].rno,s[i].name,s[i].branch,s[i].marks);
}
}
```

**union:**

union is a collection of elements of different data types.

**Declaration of a union:-****Syntax:**

```
union tagname
```

```
{
```

```
datatype member1;
```

```
datatype member2;
```

```
datatype member3;
```

```
.
```

```
.
```

```
.
```

```
datatype member n;
```

```
};
```

In the above syntax union is a keyword that declares/stores the members or fields of a union. tag

name is name of the union.

datatype maybe any basic datatype such as int,float,char etc. member1,member2,  
..... ,member n are union members or fields of a union.

The body of a union should be end with semicolon(;).

### **Example:**

```
union student
{
int rno;
char name[20];
char branch[20];
float marks;
};
```

### **Declaration of a union variable:-**

#### **Syntax:-**

union tagname variable;

#### **Example:**

union student s;

### **Initialization of a union variable:-**

#### **Syntax:-**

union tagname variable={member1,member2,...member n}; Example:

union student s={501,"Dinesh","cse",100.5};

### **Member operator or dot operator ( . ) :-**

To access the each member of a union using the union variable we have to use dot operator ( . ) in c language.

#### **Syntax:-**

union variable. union member

### **Example Program:**

```
#include<stdio.h> union student
{
int rno;
char name[20];
float marks;
```

```

};
int main()
{
union student s;
printf("enter the student rollno\n");
scanf("%d",&s.rno);
printf("the rno=%d",s.rno);
printf("enter the student name\n");
scanf("%s",s.name);
printf("name of the student is %s",s.name);
printf("enter the marks of the student\n");
scanf("%f",&s.marks);
printf("marks of the student is %f",s.marks);
}

```

<u>Structure</u>	<u>Union</u>
<p>1. Structure is a collection of elements of different data types.</p> <p>2. Declaration of a structure:- struct tagname { datatype member1; datatype member2; datatype member3; . . datatype member n; };</p> <p>3. struct is a keyword that declares/stores the members or fields of a structure.</p> <p>4. The body of a structure should be end with semicolon(;).</p> <p>5. Example:- struct student { int rollno; char name[20]; char branch[20]; float marks; };</p>	<p>1. union is a collection of elements of different data types.</p> <p>2. Declaration of a union:- union tagname { datatype member1; datatype member2; datatype member3; . . datatype member n; };</p> <p>3. union is a keyword that declares/stores the members or fields of a union.</p> <p>4. The body of a union should be end with semicolon(;).</p> <p>5. Example:- union student { int rollno; char name[20]; char branch[20]; float</p>



6. Declaration of a structure variable:-

```
struct tagname variable;
```

Example:

```
struct student s;
```

7. Member operator/Dot operator (.):- To access the each member of a structure using the structure variable we have to use dot operator (.) in C language.

Syntax:- structurevariable.structuremember

8. In structures each member has its own storage allocation.

9. For example struct student

```
{
int rollno;
char name[20]; float
marks;
};
```

The total memory required to store a structure variable is equal to the sum of size of all the members.

In the above case 26 bytes(2+20+4) will be required to store structure variable.

11. Structure can contain any numbers of members and it can handle all the members at the same time.

```
marks;
```

```
};
```

6. Declaration of a union variable:-

```
union tagname variable;
```

Example:

```
union student s;
```

7. Member operator/Dot operator (.):- To access the each member of a union using the union variable we have to use dot operator (.) in C language.

Syntax:- unionvariable.unionmember

8. In unions all the members use the same memory allocation i.e., the highest storage allocation.

9. For example union student

```
{
int rollno;
char name[20]; float marks;
};
```

The total memory required to store a union variable is equal to the member with largest size.

In the above case 20 bytes will be required to store union variable.

11. Union can contain any numbers of members but it can handle one member at a time.

your roots to success...

**UNIT-3: Functions and Dynamic Memory Allocation****Function:**

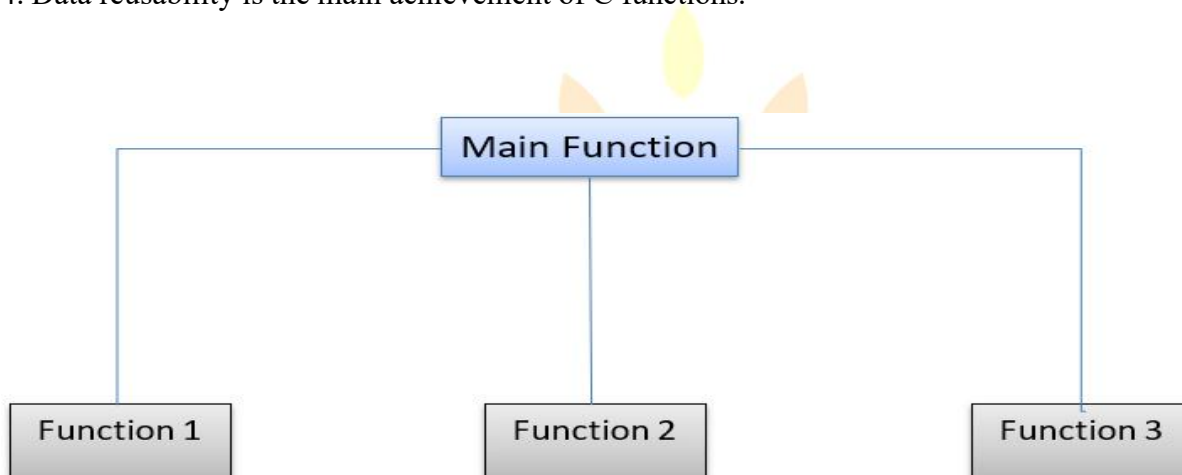
1. A Function is a sub program/self-contained block of one or more statements that performs a special task when called.

1. Every C program execution starts with main().

2. main() calls other functions to share the work.

3. The large programs can be divided into small programs using functions. Hence, a large project can be divided among many programmers.

4. Data reusability is the main achievement of C functions.



How function works: Whenever a function is called, control passes to the called function and working of the calling function is temporarily stopped. When the execution of the called function is completed, then control return back to the calling function and executes the next statements in a program.

**Types of functions:**

There are two types:

1) Library functions/Standard functions

2) User defined functions

1. Library functions: Library functions are fixed and pre defined meaning in C language. Ex:

printf(),scanf(),sqrt(),pow() etc.

2. User defined functions: These functions are developed by the user at the time of writing a program. Eg:

main(),MRCET(),add(),factorial() etc.

Every function in C has the following 3 elements

1. Function Declaration/Function Prototype
2. Function Call
3. Function Definition

#### 1. Function Declaration (Function Prototype):

The function declaration tells the compiler about function name, the data type of the return value and parameters.

##### Syntax:

```
returntype functionname(parameters list);
```

#### 2. Function Call:

The function call tells the compiler when to execute the function definition. When a function call is executed, the execution control jumps to the function definition where the actual code gets executed and return to the same function call once the execution completes.

##### Syntax:

```
functionname(parameters list);
```

#### 3. Function Definition:

The function definition provides the actual code of that function. The function definition is also known as the body of the function. The actual task of the function is implemented in the function definition. The function definition is performed before the main function or after the main function.

##### Syntax:

```
returntype functionname(datatype argument1,datatype argument2,...datatype argument n )  
{  
body of a function;  
return(expression);  
}
```

In the above syntax returntype may be any basic data type such as int, float, char etc. return statement is used to send a value back to the calling function.

##### Example:

```
main() //calling function
```

```

{
MRCET(A,B,C); //called function
}
MRCET(X,Y,
Z)
{
body of a function;
}

```

Actual arguments/Actual parameters: The arguments of the calling function are called actual arguments. In the above example A,B,C are actual arguments.

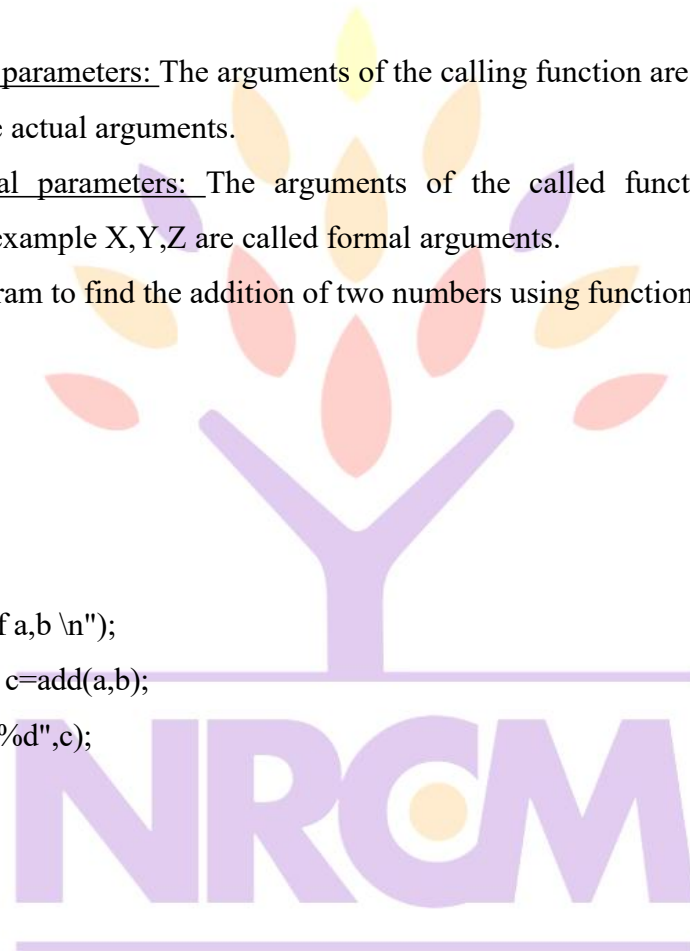
Formal arguments/Formal parameters: The arguments of the called function are called formal arguments. In the above example X,Y,Z are called formal arguments.

Example: Write a C program to find the addition of two numbers using functions.

```

#include<stdio.h>
int add(int x, int y);
int main()
{
int a,b,c;
printf("Enter the values of a,b \n");
scanf(" %d%d",&a,&b); c=add(a,b);
printf("The sum of a,b is %d",c);
}
int add(int x, int y)
{
return (x+y);
}

```



your roots to success...

### Function Categories based on Arguments and Return Values:

There are 4 types

1. Functions with arguments and functions with return values.
2. Functions without arguments and functions with no return values.
3. Functions with no arguments and functions with return value.

#### 4. Functions with arguments and functions with no return values.

##### 1. Functions with arguments and functions with return values

Functions with arguments i.e. the called function receive the data from the calling function.

Function with return values i.e. calling function receives the value from the called function. In effect there is a data transfer between calling and called function.

##### Example program

```
#include<stdio.h> int
add(int x, int y); int
main()
{
int a,b,c;
printf("enter the values of a,b\n");
scanf("%d%d",&a&&b);
c=add(a,b);
printf("the sum is %d",c);
}
int add(int x , int y)
{
return(x+y);
}
```



##### 2. Functions with no arguments and functions with no return values.

Functions with no arguments i.e. the called function does not receive any data from calling function. Functions

with no return values i.e. the calling function does not receive any value from the called function.

In effect there no data transfer between the calling and called function.

##### Example program

```
#include<stdio.h>
void add();
int main()
```

```

{
add();
}
void add()
{
int a,b;
printf("Enter the values of a,b\n");
scanf("%d%d",&a&&b);
printf("the sum is %d",a+b);
}

```

### 3. Functions with no arguments and functions with return values

Functions with no arguments i.e. the called function does not receive any data from the calling function.

Functions with return values i.e. the calling function receives the value from the called function.

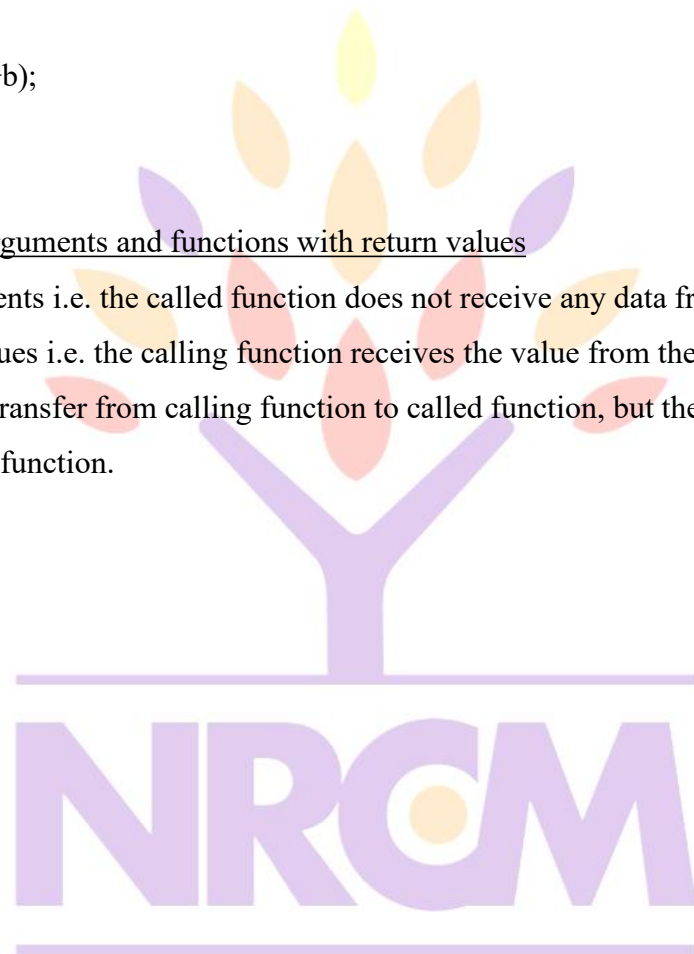
In effect there is no data transfer from calling function to called function, but there is a data transfer from the called function to calling function.

#### Example program

```

#include<stdio.h>
int add();
int main()
{
int c;
c=add();
printf("the sum is %d",c);
}
int add()
{
int a,b;
printf("Enter the values of a,b\n");
scanf("%d%d",&a,&b); return(a+b);
}

```



your roots to success...

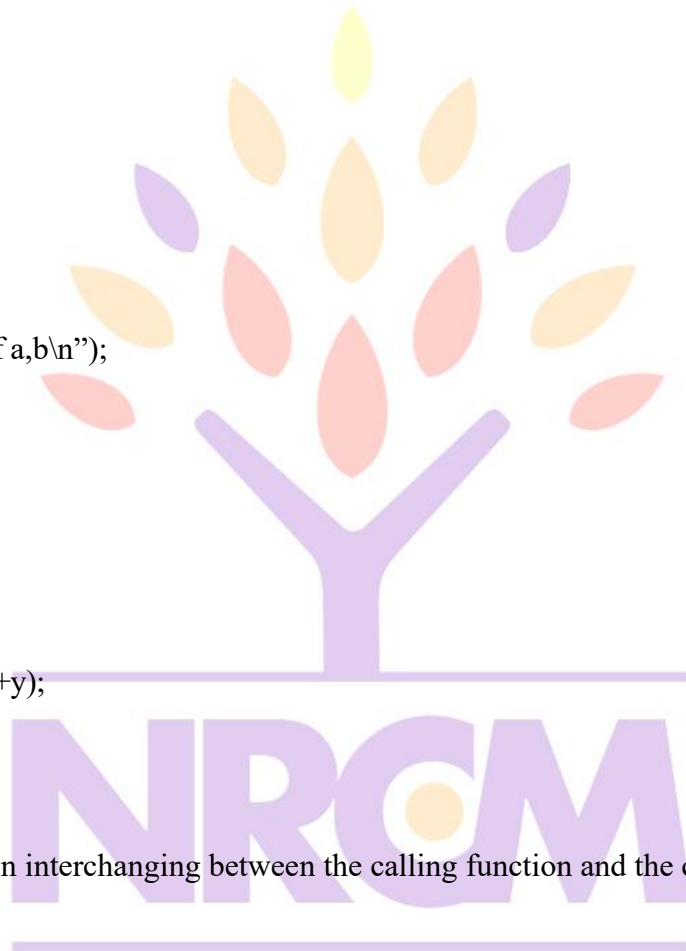
#### 4.Functions with arguments and functions with no return values

Functions with arguments i.e. the called function receives the data from the calling function. Functions with no return values i.e. the calling function does not receives the value from the called function.

In effect there is a data transfer from calling function to called function, but there is no data transfer from the called function to calling function.

#### Example program

```
#include<stdio.h> void
add(int x, int y); int
main()
{
int a,b;
printf("Enter the values of a,b\n");
scanf("%d%d",&a&&b);
add(a,b);
}
void add(int x,int y)
{
printf("The sum is %d",x+y);
}
```



#### Parameter passing:

The process of information interchanging between the calling function and the called function is known as parameter parsing.

There are two methods

- 1) Call by value/Pass by value
- 2) Call by address/Call by reference/Pass by address

#### 1.Call by value:

Sending or passing the values as actual arguments to the called function is known as call by value.

In this method the changes are made in formal arguments of the called function does not effect the actual arguments of the calling function.

Ex: Write a C program to interchange the values of two variables using call by value.

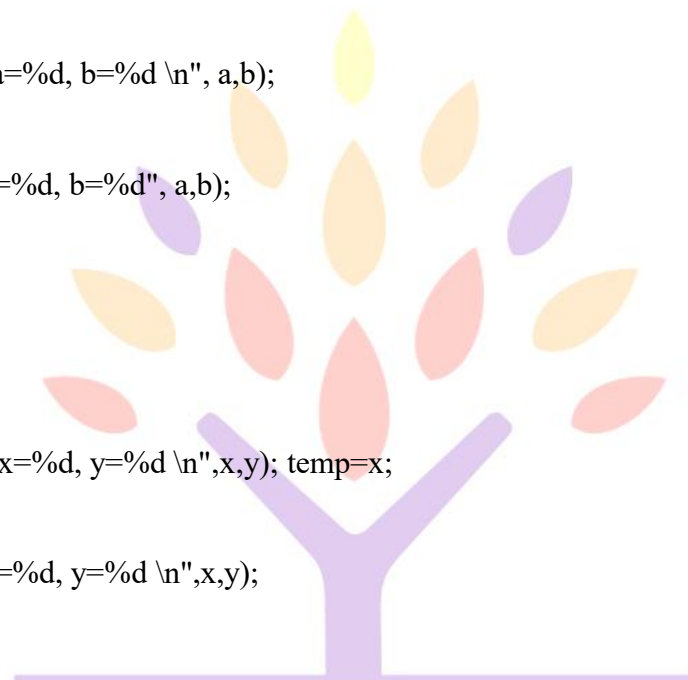
```
#include<stdio.h>
int swap(int x, int y);
int main()
{
int a=10,b=20;
printf("Before swapping a=%d, b=%d \n", a,b);
swap(a,b);
printf(" After swapping a=%d, b=%d", a,b);
}
int swap(int x, int y)
{
int temp;
printf(" Before swapping x=%d, y=%d \n",x,y); temp=x;
x=y; y=temp;
printf(" After swapping x=%d, y=%d \n",x,y);
}
```

## 2.Call by address:

Sending or passing addresses as an actual arguments to the called function is known as call by address. In this method changes are made in formal arguments of the called function effect the actual arguments of the calling function.

Ex: Write a C program to interchange the value of two variable using call by address.

```
#include<stdio.h>
int swap(int *x, int *y);
int main()
{
int a =10,b=20;
printf("Before swapping a=%d, b=%d \n",a,b);
```



**NIRCM**

your roots to success...



```

swap(&a,&b);
printf(" After swapping a=%d, b=%d",a,b);
}
int swap(int *x, int *y)
{
int temp;
temp = *x;
*x = *y;
*y = temp;
}

```

In C programming, we can pass an entire array as parameter to functions. To pass an entire array to a function, only the name of the array is passed as an argument.

#### Syntax:

```

returntype functionname(datatype arrayname[])
{
Body of a function;
}

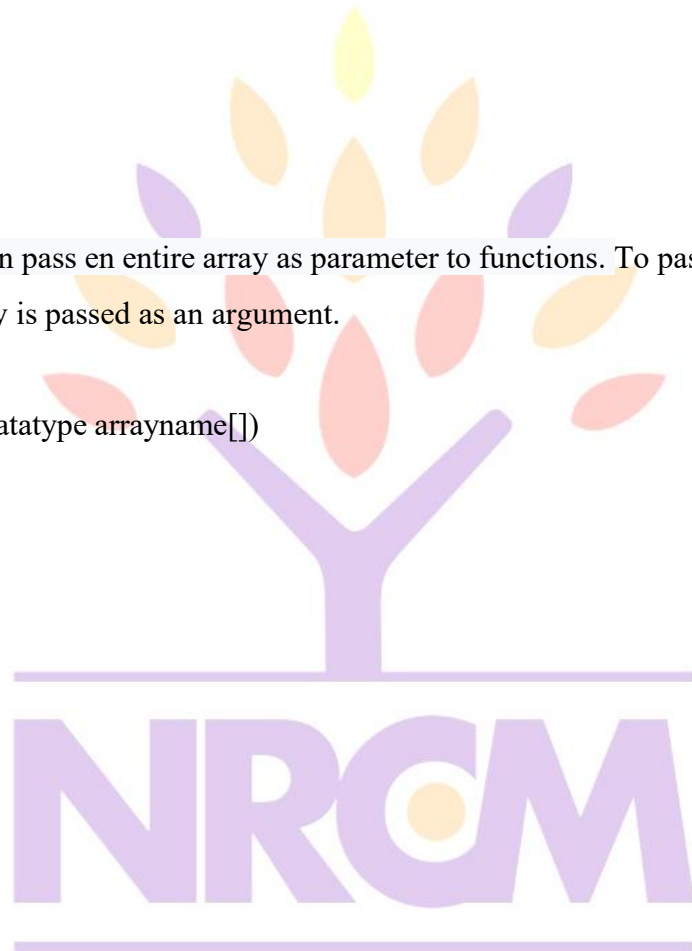
```

#### Example:

```

#include<stdio.h>
int display(int x[]);
int main()
{
int a[5],i;
printf("Enter 5 integers\n");
for(i=0;i<5;i++)
{
scanf("%d",&a[i]);
}
display(a);
}
int display(int x[])

```



your roots to success...

```

{
int i;
printf("Displaying integers\n");
for(i=0;i<5;i++)
{
printf("%d\n",x[i]);
}
}

```

### Recursion:

When a called function in turn calls the same function then chaining occurs. Recursion is a special case of this process or a repetitive process where a function calls itself. Any function which calls itself in the function definition part then it is called recursive function, and such function calls are called recursive calls. Recursive functions are very useful to solve many mathematical problems in a simpler way.

#### Example:

```

main()
{
recursion();
}
recursion()
{
recursion();
}

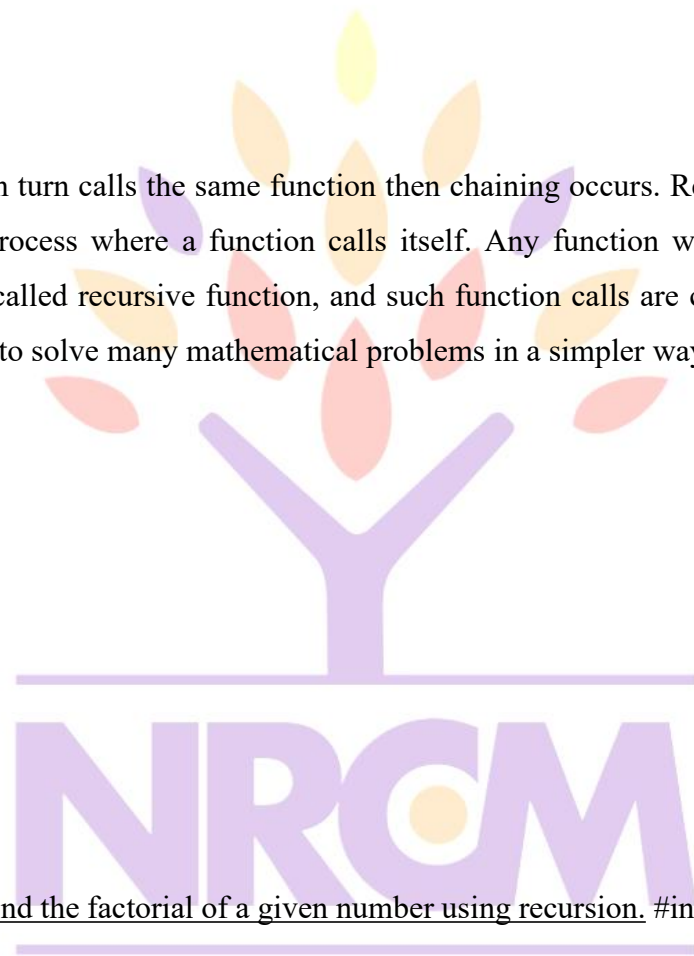
```

Example: C program to find the factorial of a given number using recursion. #include<stdio.h>

```

int factorial(int x); int
main()
{
int r,n;
printf("Enter any number\n");
scanf("%d",&n); r=factorial(n);
printf("the factorial of the given number is %d",r);
}
int factorial(int x)
{
int fact;

```



your roots to success...

```

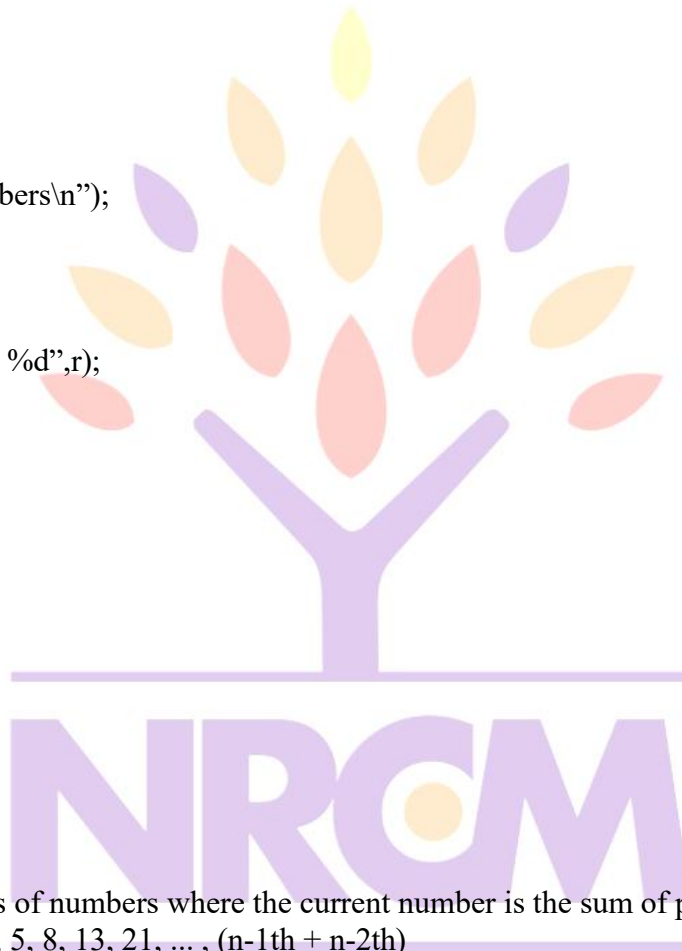
if(x==0)
return(1);
else
fact=x*factorial(x-1);
return(fact);
}

```

```

#include<stdio.h>
int gcd(int x,int y);
int main()
{
int a,b,r;
printf("Enter the two numbers\n");
scanf("%d%d",&a,&b);
r=gcd(a,b);
printf("The GCD of a,b is %d",r);
}
int gcd(int x,int y)
{
if(y==0)
return(x);
else
return gcd(y,x%y);
}

```



Fibonacci series is a series of numbers where the current number is the sum of previous two terms. For Example: 0, 1, 1, 2, 3, 5, 8, 13, 21, ... , (n-1th + n-2th)

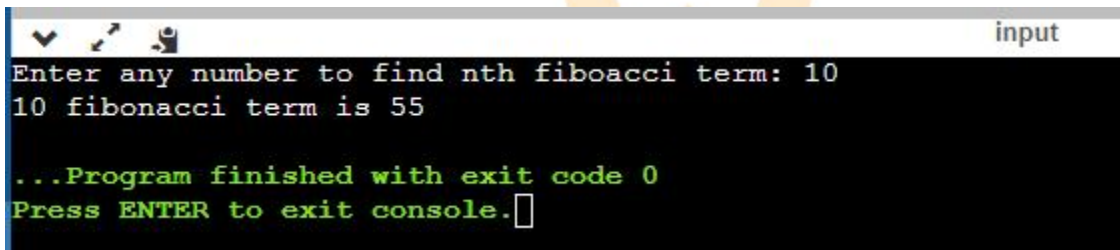
### Program to find nth Fibonacci term using recursion

```

#include <stdio.h>
unsigned long fibo(int num); int
main()
{
int num;
unsigned long fibonacci;
printf("Enter any number to find nth fiboacci term: "); scanf("%d",
&num);
fibonacci=fibo(num);

```

```
printf("%d fibonacci term is %lu",num, fibonacci);
}
unsigned long fibo(int num)
{
if(num==0)
return 0;
else if(num==1)
return 1;
else
return fibo(num-1)+fibo(num-2);
}
```



```
input
Enter any number to find nth fibonacci term: 10
10 fibonacci term is 55
...Program finished with exit code 0
Press ENTER to exit console. □
```

## Dynamic memory allocation in C

### Dynamic Memory Allocation:

The process of allocating memory during program execution is called dynamic memory allocation. All the memory management functions are found in standard library file (stdlib.h).

C language offers 4 dynamic memory allocation functions. They are,

1. malloc()
2. calloc()
3. realloc()
4. free()

### 1. malloc() function

malloc() is the short name for memory allocation and is used to dynamically allocate a single large block of contiguous memory according to the size specified.

It does not initialize the memory allocated during execution. It carries a garbage value. **Syntax:**

```
ptr=(void *)malloc(number*sizeof(int));
```

### Example program using malloc():

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

```
#include<stdlib.h>
int main()
{
int i,n,*ptr;
printf("Enter the no. of integers:");
scanf("%d",&n);
ptr=(int *)malloc(n*sizeof(int));
if(ptr==NULL)
{
printf("Memory is not available"); exit(1);
}
for(i=0;i<n;i++)
{
printf("Enter an integer:");
scanf("%d",ptr+i);
}
for(i=0;i<n;i++)
{
printf("%d\t",*(ptr+i));
}
}
```

### 1. calloc() function

calloc() is the short name for contiguous allocation and is used to dynamically allocate multiple blocks of memory according to the size specified.

It initializes the memory allocated during execution to zero.

#### Syntax:

```
ptr=(void *)calloc(number,sizeof(int));
```

#### Example program using malloc():

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int i,n,*ptr;
printf("Enter the no. of integers:");
scanf("%d",&n);
ptr=(int *)calloc(n,sizeof(int));
```



**NRCM**

your roots to success...

```

if(ptr==NULL)
{
printf("Memory is not available"); exit(1);
}
for(i=0;i<n;i++)
{
printf("Enter an integer:");
scanf("%d",ptr+i);
}
for(i=0;i<n;i++)
{
printf("%d\t",*(ptr+i));
}
}

```



### **3.realloc() function**

realloc () function modifies the allocated memory size by malloc () and calloc () functions to new size. If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

#### **Syntax**

realloc ()	realloc (pointer_name, number * sizeof(int));
------------	---

### **4.free() function**

free () function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system.

#### **Syntax**

free ()	free (pointer_name);
---------	----------------------

**UNIT-4: SEARCHING AND SORTING**

Step 1. Start

Step 2. Read the coefficients of the equation, a, b and c from the user. Step 3.

Calculate discriminant =  $(b * b) - (4 * a * c)$

Step 4. If (discriminant > 0) then

4.1: Calculate root1 =  $(-b + \text{sqrt}(\text{discriminant})) / (2 * a)$

4.2: Calculate root2 =  $(-b - \text{sqrt}(\text{discriminant})) / (2 * a)$

4.3: Display "Roots are real and different"

4.4: Display root1 and root2

Step 5: Otherwise if (discriminant = 0)

5.1: Calculate root1 =  $-b / (2 * a)$

5.2: Calculate root2 =  $-b / (2 * a)$

5.3: Display "Roots are real and equal"

5.4: Display root1 and root2

Step 6. Otherwise

6.1: Display "Roots are imaginary"

Step 7. Stop

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

```
int minmaxarray(int a[],int n); int
```

```
main()
```

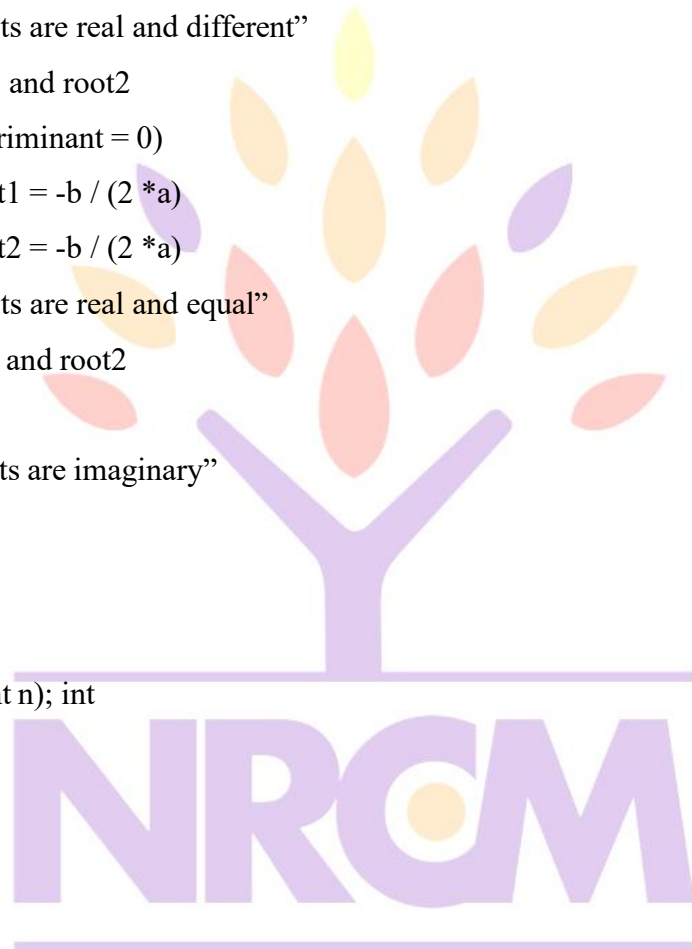
```
{
```

```
int a[100],i,n,sum;
```

```
printf("Enter size of the array\n");
```

```
scanf("%d", &n);
```

```
printf("Enter elements in array\n");
```



your roots to success...

```

for(i=0;i<n;i++)
{
    scanf("%d",&a[i]);
}

minmaxarray(a,n);
}

int minmaxarray(int a[],int n)
{
    int min,max,i;
    min=max=a[0];

    for(i=1;i<n;i++)
    {
        if(min>a[i])
            min=a[i];
        if(max<a[i])
            max=a[i];
    }

    printf("minimum of array is %d\n",min);
    printf("maximum of array is %d",max);
}

```



A prime number is a positive integer that is divisible only by 1 and itself.

For example: 2, 3, 5, 7, 11, 13, 17.

```

#include
<stdio.h> int
main()
{
    int n,i,c=0;
    printf("Enter a positive integer\n");
    scanf("%d", &n);

```



```

for(i=1;i<=n;i++)
{
if(n%i==0)
{

c
+
+
;
}
}
if(c==2)
printf("%d is a prime number",n);
else
printf("%d is not a prime number",n);
}

```

**Output:**

Enter a positive  
integer 7  
7 is a prime number

**1. Searching:**

Searching is a technique to find the particular element is present or not in the given list. There are two types of searching –

1. Linear Search
2. Binary Search

Both techniques are widely used to search an element in the given list.

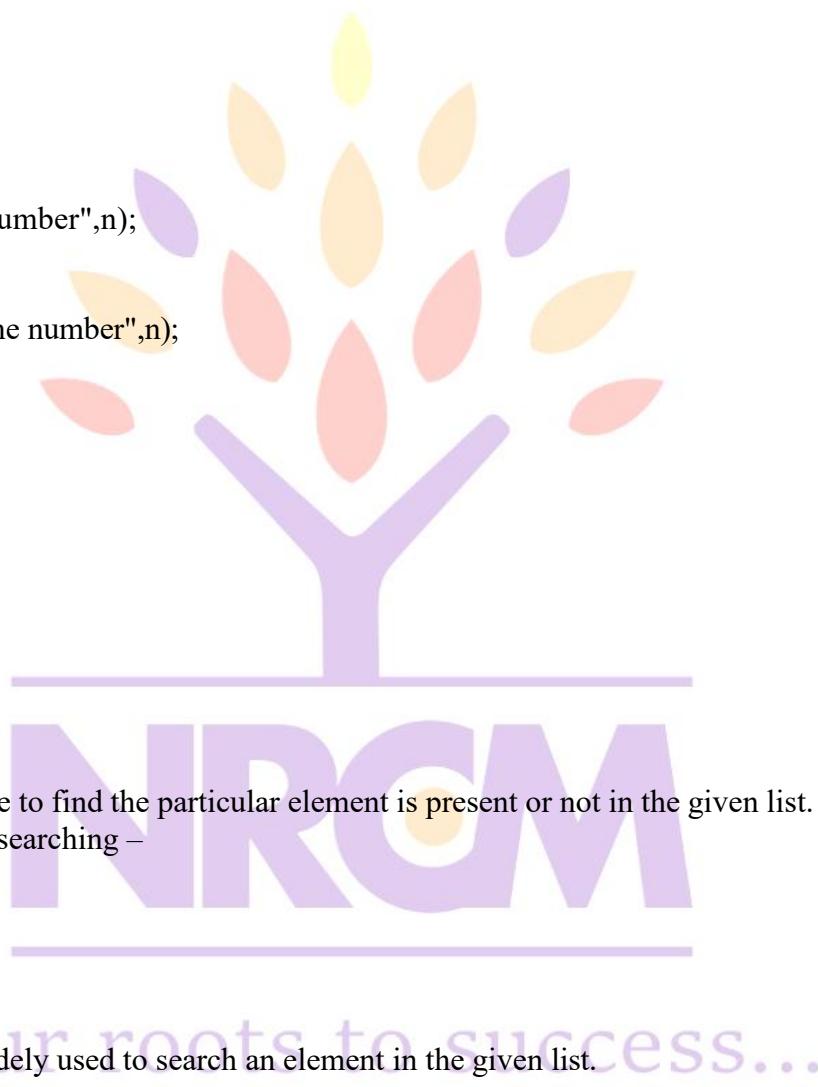
**Linear Search**

Linear search is a method of finding elements within a list. It is also called a sequential search. It is the simplest searching algorithm because it searches the desired element in a sequential manner.

It compares each and every element with the value that we are searching for. If both are matched, the element is found, and the algorithm returns the key's index position.

**Linear Search Example:**

Consider-



- We are given the following linear array.
- Element 15 has to be searched in it using Linear Search Algorithm.

92	87	53	10	15	23	67
0	1	2	3	4	5	6

Linear Search Example

Now,

- Linear Search algorithm compares element 15 with all the elements of the array one by one.
- It continues searching until either the element 15 is found or all the elements are searched. Linear Search Algorithm works in the following steps-

#### Step-01:

- It compares element 15 with the 1<sup>st</sup> element 92.
- Since  $15 \neq 92$ , so required element is not found.
- So, it moves to the next element.

#### Step-02:

- It compares element 15 with the 2<sup>nd</sup> element 87.
- Since  $15 \neq 87$ , so required element is not found.
- So, it moves to the next element.

#### Step-03:

- It compares element 15 with the 3<sup>rd</sup> element 53.
- Since  $15 \neq 53$ , so required element is not found.
- So, it moves to the next element.

#### Step-04:

- It compares element 15 with the 4<sup>th</sup> element 10.
- Since  $15 \neq 10$ , so required element is not found.
- So, it moves to the next element.

#### Step-05:

- It compares element 15 with the 5<sup>th</sup> element 15.
- Since  $15 = 15$ , so required element is found.
- Now, it stops the comparison and returns index 4 at which element 15 is present.

#### Binary Search:

- Binary Search is one of the fastest searching algorithms.
- It is used for finding the location of an element in a linear array.
- It works on the principle of divide and conquer technique.

Binary Search Algorithm can be applied only on **Sorted arrays**.

**Binary Search Example-**

Consider-

- We are given the following sorted linear array.
- Element 15 has to be searched in it using Binary Search

Algorithm. Binary Search Algorithm works in the following steps-

**Step-01:**

- To begin with, we take  $beg=0$  and  $end=6$ .
- We compute location of the middle element as-  $mid = (beg + end) / 2$

$$= (0 + 6) / 2$$

$$= 3$$

- Here,  $a[mid] = a[3] = 20 \neq 15$  and  $beg < end$ .
- So, we start next iteration.

**Step-02:**

- Since  $a[mid] = 20 > 15$ , so we take  $end = mid - 1 = 3 - 1 = 2$  whereas  $beg$  remains unchanged.

- We compute location of the middle element as-  $mid = (beg + end) / 2$

$$= (0 + 2) / 2$$

$$= 1$$

- Here,  $a[mid] = a[1] = 10 \neq 15$  and  $beg < end$ .
- So, we start next iteration.

**Step-03:**

- Since  $a[mid] = 10 < 15$ , so we take  $beg = mid + 1 = 1 + 1 = 2$  whereas  $end$  remains unchanged.

- We compute location of the middle element as-  $mid = (beg + end) / 2$

$$= (2 + 2) / 2$$

$$= 2$$

- Here,  $a[mid] = a[2] = 15$  which matches to the element being searched.
- So, our search terminates in success and index 2 is returned.

```
#include<stdio.h
```

```
> int main()
```

```
{
```

```
int a[100],i,n,flag=0,key;
```

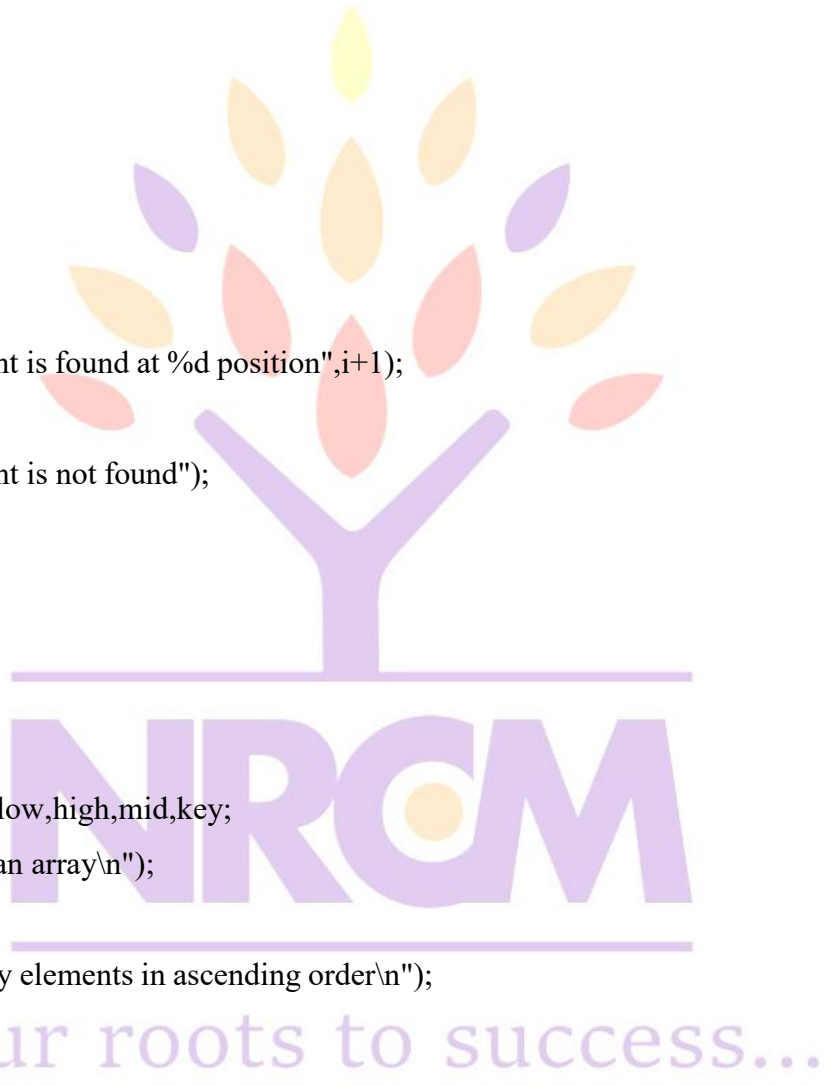
```
printf("Enter size of an array\n");
```

```
scanf("%d",&n);
```

```
printf("Enter the array elements\n");
```

```
for(i=0;i<n;i++)
{
    scanf("%d",&a[i]);
}
printf("Enter the key element\n");
scanf("%d",&key);
for(i=0;i<n;i++)
{
    if(key==a[i])
    {
        flag=1;
        break;
    }
}
if(flag==1)
    printf("Key Element is found at %d position",i+1);
else
    printf("Key Element is not found");
}

#include<stdio.h
> int main()
{
    int a[100],i,n,flag=0,low,high,mid,key;
    printf("Enter size of an array\n");
    scanf("%d",&n);
    printf("Enter the array elements in ascending order\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("Enter the key element\n");
    scanf("%d",&key);
    low=
    0;
    high
    =n-1;
```

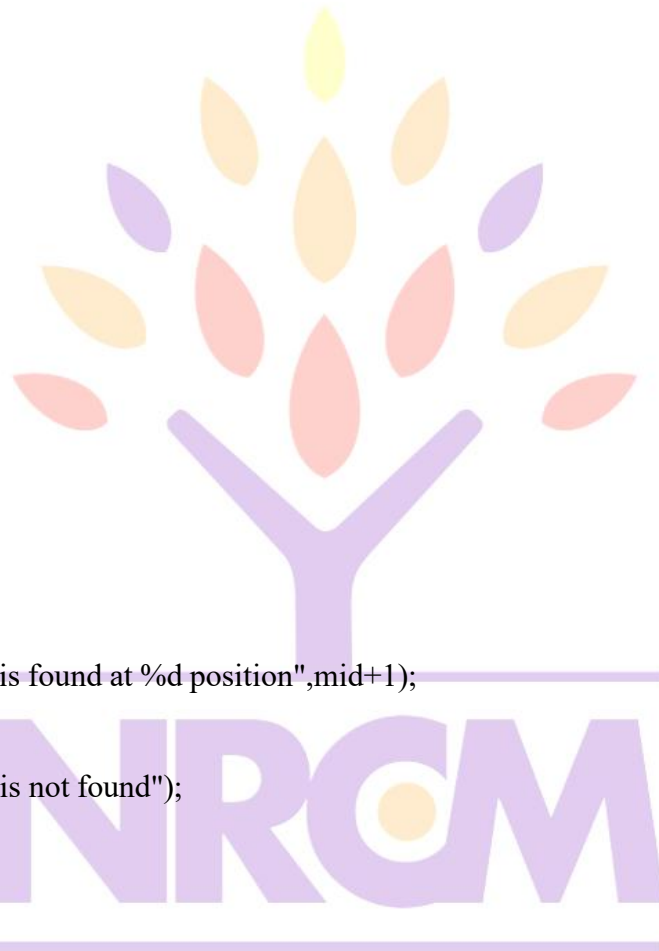


```

while(low<=high)
{
    mid=(low+high)/2;
    if(key==a[mid])

    {
        flag
        =1;
        brea
        k;
    }
    else
    {
        if(key>a[mid])
            low=mid+1;
        else
            high=mid-1;
    }
}
if(flag==1)
    printf("Key Element is found at %d position",mid+1);
else
    printf("Key Element is not found");
}

```



### Sorting:

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios –

- Telephone Directory – The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- Dictionary – The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

**Bubble Sort Algorithm**

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

**How Bubble Sort Works?**

We take an unsorted array for our example.



Bubble sort starts with very first two elements, comparing them to check which one is greater.



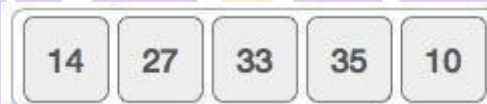
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this –



Next we compare 33 and 35. We find that both are in already sorted positions.



Then we move to the next two values, 35 and 10.



We know then that 10 is smaller 35. Hence they are not sorted.



We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –



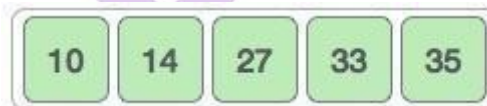
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sort learns that an array is completely sorted.



Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

### **How Selection Sort Works?**

Consider the following depicted array as an example.



For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



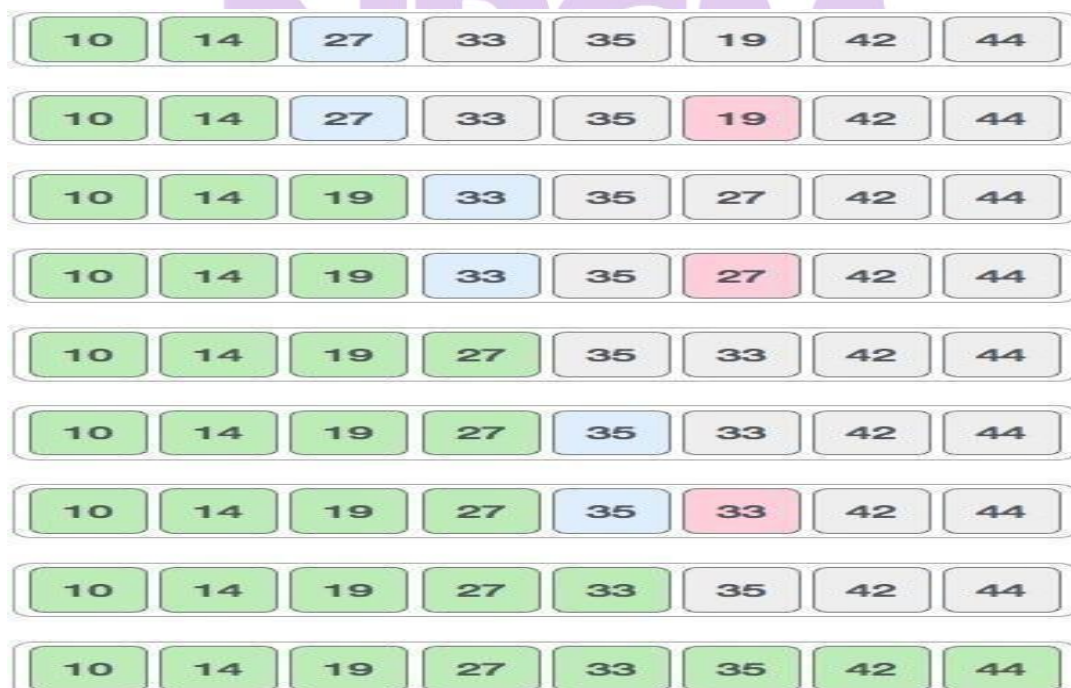
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array. Following is a pictorial depiction of the entire sorting process





This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array).

### How Insertion Sort Works?

We take an unsorted array for our example.



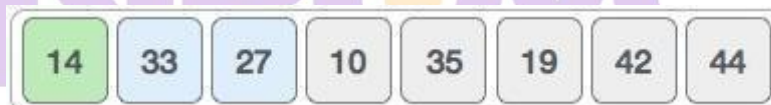
Insertion sort compares the first two elements.



It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in the correct position.



It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



These values are not in a sorted order.



So we swap them.



However, swapping makes 27 and 10 unsorted.



Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list.

```
#include<stdio.h> int
main ()
{
  int a[100],i,j,n,temp; printf("Enter
  size of an array\n"); scanf("%d",&n);
  printf("Enter the array elements\n"); for(i=0;i<n;i++)
  {
    scanf("%d",&a[i]);
  }
  for(i=0;i<n;i++)
  {
    for(j=i+1;j<n;j++)
    {
      if(a[i]>a[j])
```

```

        {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }

    }

}

printf("After sorting array elements are\n");
for(i=0;i<n;i++)
{
    printf("%d\t",a[i]);
}
}

```

```

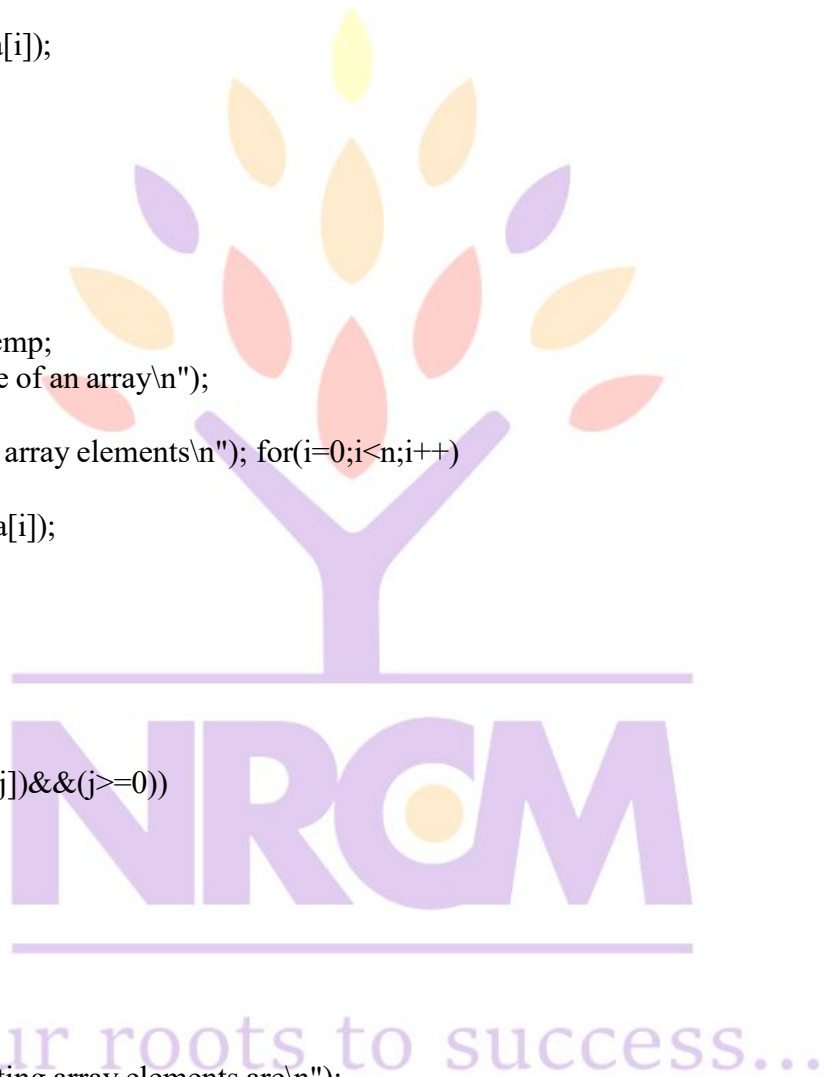
#include<stdio.h>
int main()
{
    int a[100],n,i,j,temp;
    printf("Enter size of an array\n");
    scanf("%d",&n);
    printf("Enter the array elements\n"); for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    for(i=1;i<n;i++)
    {
        temp=a[i];
        j=i-1;
        while((temp<a[j])&&(j>=0))
        {
            a[j+1]=a[j];
            j=j-1;
        }
        a[j+1]=temp;
    }
    printf("After sorting array elements are\n");
    for(i=0;i<n;i++)
    {
        printf("%d\t",a[i]);
    }
}

```

```

#include<stdio.h> int
main ()
{
    int a[100],i,j,n,temp,small;

```



```

printf("Enter size of an array\n");
scanf("%d",&n);
printf("Enter the array elements\n"); for(i=0;i<n;i++)
{
    scanf("%d",&a[i]);
}
for(i=0;i<n-1;i++)
{
    small=i; for(j=i+1;j<n;j++)
    {
        if(a[j]<a[small])
            small=j;
    }
    temp=a[small];
    a[small]=a[i]; a[i]=temp;
}
}
printf("After sorting array elements are\n");
for(i=0;i<n;i++)
{
    printf("%d\t",a[i]);
}
}

```

### **Time Complexity:**

Time Complexity is defined as the number of times a particular instruction set is executed rather than the total time is taken. It is because the total time taken also depends on some external factors like the compiler used, processor's speed, etc.

### **Space Complexity:**

Space Complexity is the total memory space required by the program for its execution. Both are calculated as the function of input size(n). One important thing here is that in spite of these parameters the efficiency of an algorithm also depends upon

the nature and size of the input.

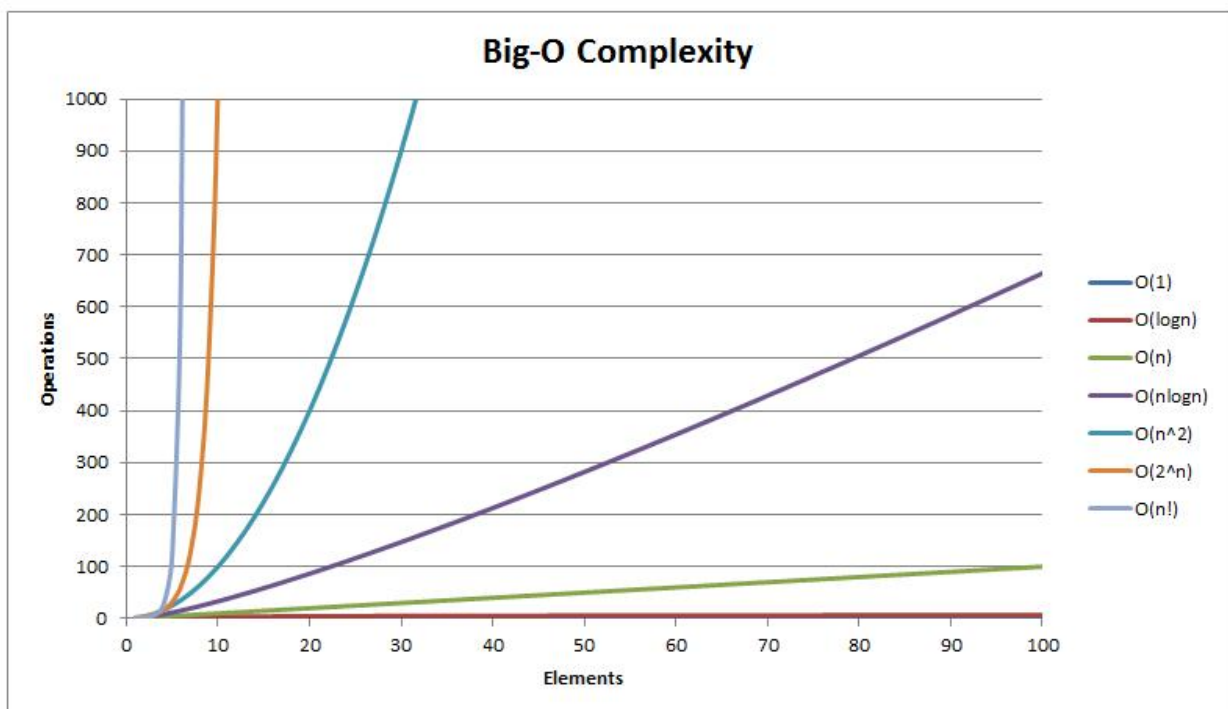
In this concept best, worst, and average cases of a given algorithm express what the resource usage is at least, at most and on average, respectively. Usually the resource being considered is running time, i.e. time complexity, but could also be memory or other resource.

Best case is the function which performs the minimum number of steps on input data of n elements.

Worst case is the function which performs the maximum number of steps on input data of size n.

Average case is the function which performs an average number of steps on input data of n elements.

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Bucket Sort	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$
Tim Sort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Shell Sort	$O(n)$	$O((n \log(n))^2)$	$O((n \log(n))^2)$	$O(1)$



## ANALYSIS OF SORTING ALGORITHMS

Name	Average	Worst	Memory	Stable	Method	Other notes
Bubble Sort	-	$O(n^2)$	$O(1)$	Yes	Exchanging	Oldest sort
Cocktail Sort	-	$O(n^2)$	$O(1)$	Yes	Exchanging	Variation of bubble sort
Comb Sort	-	-	$O(1)$	No	Exchanging	Small code size
Gnome Sort	-	$O(n^2)$	$O(1)$	Yes	Exchanging	Tiny code size
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$	No	Selection sort	Can be implemented as a stable sort

Insertion Sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Selection sort	Average case in $O(n+d)$ , where $d$ is the number of inversion
Shell Sort	-	$O(n \log^2 n)$	$O(1)$	No	Insertion	No extra memory required
Binary Tree Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	No	Insertion	When using a self-balancing binary search tree
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Merging	Recursive nature, extra memory required
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	Selection	Recursive, extra memory required
Quick Sort	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No	Partitioning	Recursive, based on divide conquer technique

Table: Comparison of sorting algorithms

## ANALYSIS OF SEARCHING ALGORITHMS

Name	Average	Worst	Space	Other Notes
Linear Search	$O(N)$	$O(N)$	$O(N)$	Suitable for small data
Binary Search	$O(\log N)$	$O(\log N)$	$O(N)$	Suitable for mid sized data
Hash Search	$O(1)$	$O(1)$	$O(N)$	Suitable for large data
Interpolation Search	$O(\log(\log N))$	$O(N)$		
Exponential search	$O(N)$	$O(N)$	$O(N)$	Suitable for large data
Ternary Search	$O(\log N)$	$O(N)$		Suitable for large data sets

Table: Comparison of searching algorithms



The logo for NIRCM (National Institute of Research in Computer Graphics and Multimedia) features the acronym 'NIRCM' in a large, bold, purple font. The letter 'I' is stylized with a yellow circle in the center, resembling a sun or a root. The logo is set against a background of stylized purple and yellow leaves.

your roots to success...

## UNIT-5: Preprocessor and File handling in C

### Preprocessors:

The C Preprocessor is not part of the compiler but is a separate step in the compilation process. In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation. We'll refer to the C Preprocessor as the CPP. All preprocessor commands begin with a pound symbol #. It must be the first nonblank character, and for readability, a preprocessor directive should begin in first column.

### List of Preprocessor Directives

To execute a preprocessor program on a certain statement, some of the preprocessor directives types are:

#define: It substitutes a preprocessor using macro.

#include: It helps to insert a certain header from another file.

#undef: It undefines a certain preprocessor macro.

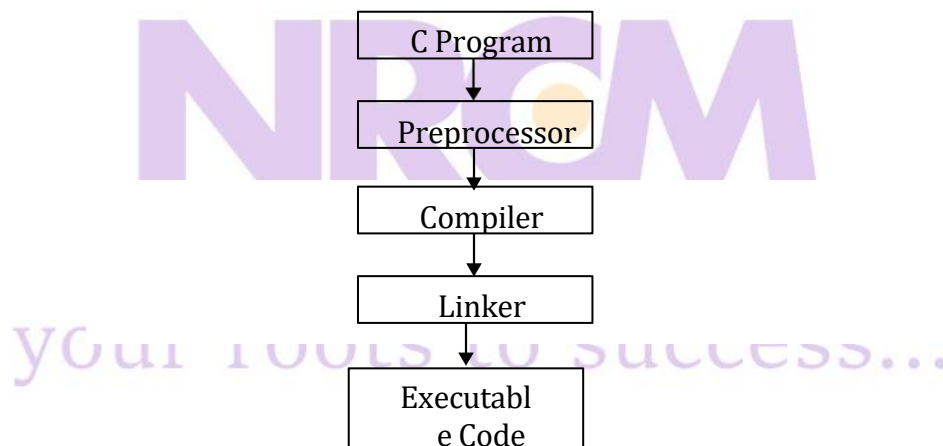
#ifdef: It returns true if a certain macro is defined.

#ifndef: It returns true if a certain macro is not defined.

#if, #elif, #else, and #endif: It tests the program using a certain condition; these directives can be nested too.

### PREPROCESSOR DIRECTIVES

The C Preprocessor is not part of the compiler but it extends the power of C programming language. . The preprocessor provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control. The preprocessor's functionality comes before compilation of source code and it instruct the compiler to do required pre- processing before actual compilation. Working procedure of C program is shown in Fig. 2.8. In general, preprocessor directives begin with a # symbol do not end with semicolon are processed before compilation of source code



**Fig. 2.8 Working Procedure of C Program**

There are four types of Preprocessor Directives supported by C language. They are:

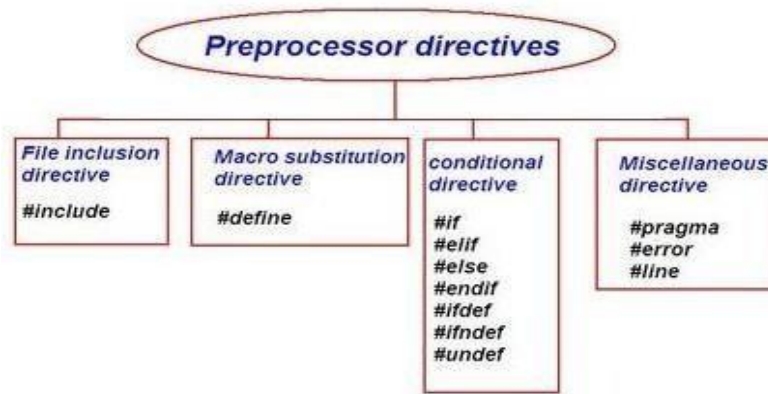
File Inclusion directive

Macro Substitution directive

Conditional directive

Miscellaneous directive

List of all possible directives belong to each of the above is listed in Fig 2.9.



**Fig Preprocessor Directives**

The details of above listed preprocessor directives are narrated in Table.

Table Preprocessor directives and their description

Directive	Description
#include	It includes header file inside a C Program.
#define	It is substitution macro. It substitutes a constant with an expression.
#if	It includes a block of code depending upon the result of conditional expression.
#else	It is a complement of #if
#elif	#else and #if in one statement. It is similar to if else ladder.
#endif	It flags the end of conditional directives like #if, #elif etc.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true If constant is defined earlier using #define.
#ifndef	Returns true If constant is not defined earlier using #define.
#pragma	Issues special commands to the compiler.
#error	Prints error message on stderr.

### File Inclusion directive

#### #include

It is used to include header file inside C Program. It checks for header file in current directory, if path is not mentioned. To include user defined header file double quote is used (") instead of using triangular bracket (<>).

Example:

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

```
// Standard Header File #include "big.h"
```

```
// User Defined Header File
```

Preprocessor replaces #include <stdio.h> with the content of stdio.h header file. #include "Sample.h" instructs the preprocessor to get Sample.h from the current directory and add the content of Sample.h file.

### Macro Substitution directive

#### #define

It is a simple substitution macro. It substitutes all occurrences of the constant and replace them with an expression. There are two types of macro supported by C. They are:

Simple macro

macro with arguments

Simple macro Syntax:



```
#define identifier value
```

Where

#define- is a preprocessor directive used for text substitution.

identifier - is an identifier used in program which will be replaced by value.(In general the identifiers are represented in capital letters in order to differentiate them from variable)

value -It is the value to be substituted for identifier.

Example:

```
#define PI 3.14
```

```
#define NULL 0
```

Example:

```
//Program to find the area of a circle using simple macro #include <stdio.h>
```

```
#define PI 3.14 int main()
```

```
{
int radius; float area;
printf("Enter the radius of circle \n"); scanf("%d", &radius);
area= PI * radius * radius; printf("Area of Circle=%f", radius);
}
```

Output

```
Enter the radius of circle 10
Area of Circle = 314.000000
```

macro with arguments

#define Preprocessing directive can be used to write macro definitions with parameters. Whenever a macro identifier is encountered, the arguments are substituted by the actual arguments from the C program.

Data type definition is not necessary for macro arguments. Any numeric values like int, float etc can be passed as a macro argument . Specifically, argument macro is not case sensitive.

Example:

```
#define area(r) (3.14*r*r)
```

Example:

```
//Program to find the area of a circle using macro with arguments #include <stdio.h>
```

```
#define area(r) (3.14*r*r) int main()
```

```
{
int radius; float a;
printf("Enter the radius of circle \n"); scanf("%d", &radius);
a= area(radius);
printf("Area of Circle=%f", a);
}
```

Output

```
Enter the radius of circle 10
Area of Circle = 314.000000
```

Predefined Macros in C Language

C Programming language defines a number of macros. Table 2.8 is the list of some commonly used macros in C

Table 2.8 Predefined macros in C

Macro	Description
NULL	Value of a null pointer constant.
EXIT_SUCCESS	Value for the exit function to return in case of successful completion of program.
EXIT_FAILURE	Value for the exit function to return in case of program termination due to failure.
RAND_MAX	Maximum value returned by the rand function.

FILE	Contains the current filename as a string.
LINE	Contains the current line number as a integer constant.
DATE	Contains current date in "MMM DD YYYY" format.
TIME	Contains current time in "HH:MM:SS" format.

Example:

```
// Program to print the values of Predefined macros
#include <stdio.h>
#include <stdlib.h>
int main()
{
printf("NULL : %d\n", NULL );
printf("EXIT_SUCCESS : %d\n", EXIT_SUCCESS );
printf("EXIT_FAILURE : %d\n", EXIT_FAILURE );
printf("RAND_MAX : %d\n", RAND_MAX );
printf("File Name : %s\n", FILE );
printf("DATE : %s\n", DATE);
printf("Line : %d\n", LINE);
return 0;
}
```

Output:

```
NULL : 0
EXIT_SUCCESS : 0
EXIT_FAILURE : 1
RAND_MAX : 32767
File Name : BuiltinMacro.c DATE : Aug 16 2017
Line : 12
```

Conditional directive

#if, #elif, #else and #endif

The Conditional directives permit to include a block of code based on the result of conditional expression.

Syntax:

```
#if <expression>
statements; #elif <expression>
statements;
```

your roots to success...

Expression represents a condition which produces a boolean value as a result.

Conditional directive is similar to if else condition but it is executed before compilation.

Condition\_Expression must be only constant expression.

Example:

```
//Program to illustrate the conditional directives #include <stdio.h>
#define A 10 int main()
{
#if (A>5)
printf("A=%d", X); #elif (A<5)
printf("A=%d", 4);
#else
printf("A=%d", 0);
```

```
#endif  
return 0;  
}  
Output X=10
```

```
#undef
```

The #undef directive undefines a constant or preprocessor macro defined previously using #define.

Syntax:

```
#undef <Constant>
```

Example:

```
#include<stdio.h> #define P 100 #ifdef P  
#undef P #define P 30  
#else  
#define P 100  
#endif  
int main()  
{  
    printf("%d",P); return 0;  
}
```

Output:  
30



your roots to success...

```
#ifndef #ifdef, #ifndef
```

```
#ifdef
```

#ifdef directive is used to check whether the identifier is currently defined.

Identifiers can be defined by a #define directive or on the command line. #ifndef

#ifndef directive is used to check whether the identifier is not currently defined.

Example:

```
#ifdef PI
```

```
printf( "Defined \n" );
```

```
#endif #ifndef PI
```

```
printf( "First define PI\n" );
```

```
        #endif
```

Output:

```
First define PI Miscellaneous directive
```

The pragma directive is used to access compiler-specific preprocessor extensions. Each pragma directive has different implementation rule and use . There are many type of pragma directive and varies from one compiler to another compiler .If compiler does not recognize particular pragma then it ignores the pragma statement without showing any error or warning message.

Example: #pragma sample int main()

```
{
```

```
printf("Pragma verification "); return 0;
```

```
}
```

Since #pragma sample is unknown for Turbo c compiler, it ignores sample directive without showing error or warning message and execute the whole program

assuming #pragma sample statement is not present. The following are the list of possible

#pragma directives supported by C.

```
#pragma startup
```

```
#pragma exit
```

```
pragma warn
```

```
#pragma option
```

```
#pragma inline
```

```
#pragma argsused
```

```
#pragma hdrfile
```

```
#pragma hdrstop
```

```
#pragma saveregs
```

```
#error
```

The #error directive causes the preprocessor to emit an error message. #error directive is used to prevent compilation if a known condition that would cause the program not to function properly.

Syntax:

```
#error "message"
```

Example: int main()

```
{
```

```
#ifndef PI
```

```
#error "Include PI" #endif
```

```
return 0;
```

```
}
```

Output

```
compiler error --> Error directive : Include PI
```

```
#line
```

It tells the compiler that next line of source code is at the line number which has been specified by constant in #line directive

Syntax:

#line <line number> [File Name] Where

File Name is optional

Example:

```
int main()
```

```
{
```

```
#line 700
```

```
printf("Line Number %d", LINE ); printf("Line Number %d", LINE ); printf("Line Number %d", LINE ); return 0;
```

```
}
```

**Output**

**t 700**

**701**

**702**

Files in C:

A File is collection of related data stored permanently in computer. Using files we can store our data in Secondary memory (Hard disk).

Types of Files

Text files

Binary files

Text files

Text files are the normal .txt files that you can easily create using notepad or any simple text editors. When you open those files, you will see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.

Binary files

Binary files are mostly the .bin files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold higher amount of data, are not readable easily and provides a better security than text files.

File Operations in C:

There are 5 basic operations on files, They are

Naming a file

Opening a file

Reading data from a file

Writing data to a file

Closing a file Defining and Opening a file

Data structure of file is defined as FILE in the standard I/O function. So all files should be declared as type FILE.

Declaration/Syntax of a file:

```
FILE *fp;
```

```
fp=fopen("filename", "mode");
```

Here fp declare a variable as a pointer to the data type FILE. In the above syntax mode specifies purpose of opening a file.

File Opening modes

S.No Mode Meaning

1 r Reading

Purpose

Open the file for reading only.

2	w	Writing	Open the file for writing only.
3	a	Appending	Open the file for appending (or adding) data to it.
4	r+	Reading + Writing	New data is written at the beginning override existing data.
5	w+	Writing + Reading	Override existing data.
6	a+	Reading + Appending	To new data is appended at the end of file.

### Closing a File

A file must be close after completion of all operations related to a file. For closing a file we need `fclose()` function.

Syntax:

```
fclose(filepointer);
```

File handling functions:

There are 8 types of file handling functions:

```
fopen()
fclose()
getc()/fgetc()
putc()/fputc()
getw()
putw()
fscanf()
fprintf()
```

`fopen()`: This can be used to create a new file or open an existing file.

Syntax: `fp=fopen("filename","mode");`

`fclose()`: A file must be closed after all operations completed on it.

Syntax: `fclose(fp);`

`getc()/fgetc()`: This function can be used to read a character from a file.

Syntax: `c=getc(fp);`

`putc()/fputc()`: This function can be used to write a character to a file.

Syntax: `putc(c,fp);`

`getw()`: This can be used to read an integer from a file.

Syntax: `n=getw(fp);`

`putw()`: This can be used to write an integer to a file.

Syntax: `putw(n,fp);`

fscanf( ): This can be used to read a set of data values from a file.

Syntax: fscanf(fp,"control string",&variablelist);

fprintf( ): This can be used to write a set of data values to a file.

Syntax: fprintf(fp,"control string",variablelist);

```
#include<stdio.h> int main( )
{
FILE *f1;
char ch;
f1=fopen("INPUT","w");
printf("Enter the data to the INPUT file and press ctrl Z to stop\n"); while((ch=getchar())!=EOF)
{
putc(ch,f1);
}
fclose(f1); f1=fopen("INPUT","r"); while((ch=getc(fp))!=EOF)
{
putchar(ch);
}
fclose(f1);
}
```

```
#include<stdio.h> int main( )
{
FILE *f1,*f2; char ch;
f1=fopen("source.txt","w");
printf("Enter the data to the source.txt file and press ctrl Z to stop\n");
while((ch=getchar())!=EOF)
{
putc(ch,f1);
}
fclose(f1); f1=fopen("source.txt","r");
f2=fopen("target.txt","w"); while((ch=getc(f1))!=EOF)
{
putc(ch,f2);
}
fclose(f1); fclose(f2);
printf("After copying the data in target.txt file is\n"); f2=fopen("target.txt","r");
while((ch=getc(f2))!=EOF)
{
putchar(ch);
}
fclose(f2);
}
```

```
#include<stdio.h> int main( )
{
FILE *f1,*f2,f3; char ch;
f1=fopen("file1.txt","w");
```

```

printf("Enter the data to the file1.txt file and press ctrl Z to stop\n"); while((ch=getchar())!=EOF)
{
putc(ch,f1);
}
fclose(f1); f1=fopen("file1.txt","r");
f3=fopen("file3.txt","w"); while((ch=getc(f1))!=EOF)
{
putc(ch,f3);
}
fclose(f1); fclose(f3);
f2=fopen("file2.txt","w");
printf("Enter the data to the file2.txt file and press ctrl Z to stop\n"); while((ch=getchar())!=EOF)
{
putc(ch,f2);
}
fclose(f2);

f2=fopen("file2.txt","r");
f3=fopen("file3.txt","w"); while((ch=getc(f2))!=EOF)
{
putc(ch,f3);
}
fclose(f2); fclose(f3);
printf("After merging two files into third file is\n"); f3=fopen("file3.txt","r");
while((ch=getc(f3))!=EOF)
{
putchar(ch);
}
fclose(f3);
}

```

fseek()

This function is used to set the cursor position to the specific position. Using this function we can set the cursor position from three different position they are as follows.

from beginning of the file (indicated with 0)

from current cursor position (indicated with 1)

from ending of the file (indicated with 2)

Syntax:

```
fseek(fp,offset,position);
```

Example Program:

```

#include<stdio.h>
int main()
{
FILE *fp;
int position; clrscr();
fp = fopen ("file.txt", "r"); position = ftell(fp);
printf("Cursor position = %d\n",position); fseek(fp,5,0);

```



```
position = ftell(fp);  
printf("Cursor position = %d\n", position); rewind(fp);  
position = ftell(fp);  
printf("Cursor position = %d", position); fclose(fp);  
}
```

**ftell()**

This function can be used to gives the current position in the file (in terms of bytes from the start).

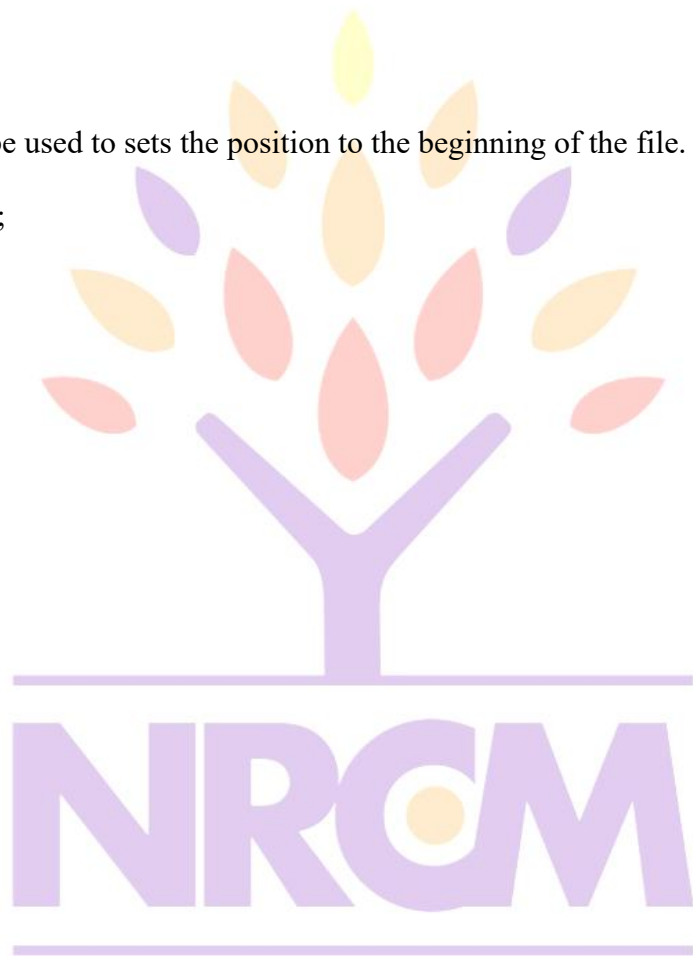
**Syntax:**

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

**rewind()**

This function can be used to sets the position to the beginning of the file.

**Syntax:** rewind(fp);



your roots to success...