SYLLABUS --UNIT II-FUNCTIONS, POINTERS, STRUCTURES AND UNIONS

Functions – Pass by value – Pass by reference – Recursion – Pointers - Definition – Initialization – Pointers arithmetic. Structures and unions - definition – Structure within a structure - Union - Programs using structures and Unions – Storage classes, Pre-processor directives.

S.No	Functions		
1	Functions		
		Introduction	
		Types of Function Implementation	
		Passing Arguments(Parameters) To Function	
			Call by Value
			Call by Reference
		Recursive Functions	
		Example Programs using functions	
2	Pointers		
		Introduction --Definition --Declaration	
		Pointer Arithmetic	
		Pointers and arrays	
		Pointers and Strings	
		Pointers and Functions	
3	Structures		
		Intro	
		Declaration—Initialization --Accessing	
		Programs using structures	
		Structure within Structure[Nested Structure]	
		Arrays and Structures	
		Structure and functions	
		Pointers and structure	
4	Union		
5	Union using Structure		
6	Storage classes		
7	Pre-processor directives.		

FUNCTION

A function is self contained program segment that carries out some specific,well defined task.

Need For Functions:

Many programs require a set of instructions to be executed repeatedly from several places in a program, then the repeated code can be placed within a single function, which can be accessed whenever it is required.

→Dividing the program into separate well-defined functions facilitates each function to be written and tested separately.
→Understanding, coding, and testing multiple separate functions is easier than doing the same for one big function main function.
→ A big program is broken into comparatively smaller functions.

Advantages of Functions

-Improves readability of code.
-Divides a complex problem into simpler ones
- Improves reuseability of code.
-Debugging errors is easier. -
modifying a program is easier.

Terms

→A function *f* that uses another function *g* is known as the *calling function*, and *g* is known as the *called function.*
→The inputs that a function takes are known as *arguments.*
→When a called function returns some result back to the calling function, it is said to *return* that result.
→The calling function may or may not function,which can theot pass *parameters* to the called function.

→Function Declaration
→Function Definition
→Function Call

Function Definition

When a function is defined, space is allocated for that function in the memory. A function definition comprises of two parts:

- Function header
- Function body

The syntax of a function definition can be given as:

```
return_data_type function_name(data_type variable1, data_type variable2,..)
{
statements
return(variable);
}
```

```
void sum(int a,int b)
{
Int c;
c=a+b;
return(c);
}
```

```
int sum(int a,int b)
{
Int c;
c=a+b;
return(c);
}
```

Note that the number of arguments and the order of arguments in the function header must be the same as that given in the function declaration statement.

Function Call

The function call statement invokes the function. When a function is invoked, the compiler jumps to the called function to execute the statements that are a part of that function. Once the called function is executed, the program control passes back to the calling function. A function call statement has the following

Syntax-1:

```
function_name(variable1, variable2, ...);
```

```
function_name(argument1, argument2, ...);
```

If the return type of the function is not void then the following syntax can be used. Syntax-2

```
variable_name = function_name(variable1, variable2, ...);
```

```
variable_name = function_name(parameter1, parameter2, ...);
```

The following points are to be noted while calling a function:

→Function name and the number and the type of arguments in the function call must be same as that given in the function declaration and the function header of the function definition.

→Names (and not the types) of variables in function declaration, function call, and header of function definition may vary.

Function Declaration

It is also called as function prototype. A function prototype consists of 4 parts

(1) Return type
(2) function name
(3) parameter list
(4) terminating (;) semicolin

```
data_type function_name(data type1 variable1, ...,data_type n variable n);
```

A function prototype tells the compiler that the function may later be used in the program. The function prototype is not needed if function is placed before main function code.

Example program

```
#include <stdio.h>
void sum(int a ,int b);   //FUNCTION DECLARATION
int main()
{   int a,b;
    scanf("%d%d",&a,&b);//get input values for a and b
    sum(a,b);   //FUNCTION CALL
}
void sum(int x,int y)   // FUNCTION DEFINITION
{
  int c;
 c=x+y;
 printf("c=%d",c);
}
```

o/p
2
3
c=5

void data type indicates that the function is returning nothing .(i.e) if the function is not returning anything then its datatype is as specified as void.

Example program without function prototype(function declaration)

```
#include <stdio.h>
void sum(int x,int y) // FUNCTION DEFINITION {
int c;
  c=x+y;
 printf("c=%d",c);
}
int main()
{   int a,b;
    scanf("%d%d",&a,&b); //get input values for a and b
    sum(a,b); //FUNCTION CALL
}
```

o/p
2
3
c=5

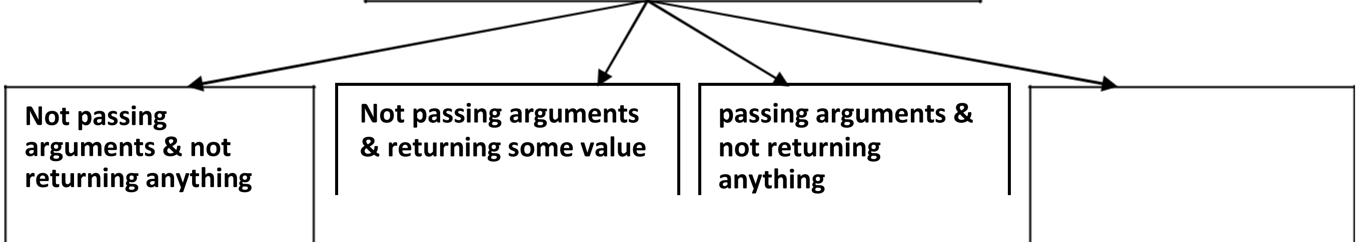
Eg-Write a program to find whether a number is even or odd using functions.

```
#include <stdio.h>
int evenodd(int); //FUNCTION DECLARATION
int main()
{
    int num, flag;
    printf("\n Enter the number : ");
    scanf("%d", &num);
    flag = evenodd(num); //FUNCTION CALL
     if (flag == 1)

     printf("\n %d is EVEN", num);
    else
    printf("\n %d is ODD", num);
    return 0;
}
int evenodd(int a)   // FUNCTION DEFINITION
{
 if(a%2 == 0)
return 1; else

retun 0;
}
```

Output
Enter the number :
78 78 is EVEN



passing arguments & returning some value

CASE-1 NOT PASSING ARGUMENTS & NOT RETURNING SOME VALUE

```
#include <stdio.h>
int sum( );    //FUNCTION DECLARATION
int main( )
{
   sum( );     //FUNCTION CALL
}
int sum( )   // FUNCTION DEFINITION
{
  int a,b,c;
 scanf("%d%d",&a,&b); // get input values for a and b
 c=x+y;
 printf("c=%d",c);
}
```

o/p
2
3
c=5

CASE-2 NOT PASSING ARGUMENTS & RETURNING SOME VALUE

```
#include <stdio.h>
int sum( );    //FUNCTION DECLARATION
int main( )
{   sum( );      //FUNCTION CALL
  printf("c=%d",c);
}
int sum( )   // FUNCTION DEFINITION
{
  int a,b,c;
scanf("%d%d",&a,&b); // get input values for a and b
 c=x+y;

return c;
 }
```

o/p
2
3
c=5

CASE-3 PASSING ARGUMENTS & NOT RETURNING ANYTHING

```
#include <stdio.h>
int sum( int a,int b);    //FUNCTION DECLARATION
int main( )
{   int a,b;
  scanf("%d%d",&a,&b); // get input values for a and b
   sum(int a,int b ); //FUNCTION CALL
   }
int sum( int x,int y)    // FUNCTION DEFINITION
{
  int c;
  c=x+y;
  printf("c=%d",c);
}
```

o/p
2
3
c=5

CASE-4 NOT PASSING ARGUMENTS & RETURNING SOME VALUE

```
#include <stdio.h>
int sum( int a,int b); //FUNCTION DECLARATION
int main( )
{    int a,b,c;
    scanf("%d%d",&a,&b); // get input values for a and b
  c=sum(int a,int b ); //FUNCTION CALL
  printf("c=%d",c);
return 0;
}
int sum( int x,int y)    // FUNCTION DEFINITION
{ int result
  result=x+y;
 return result; }
```

o/p
2
3
c=5

PASSING ARGUMENTS(PARAMETERS) TO FUNCTION

In programming function argument is commonly referred as **actual parameter** and function parameter is referred as **formal parameter**.

```
                         // Formal parameters
void add(int num1, int num2) // Function definition
{
    // Function body
}


int main()
{
                  // Actual parameters
    add(10, 20); // Function call

    return 0;
}
```

There are two ways by which parameters can be passed to a function

1. Call by Value
2. Call by Reference

Call by value

In Call by value, during function call actual parameter value is copied and passed to formal parameter. Changes made to the formal parameters does not affect the actual parameter.

Eg Program - C program to swap two numbers using call by value

```
#include<stdio.h>
int main()
{
   int n1, n2;
   printf("Enter two numbers: ");
   scanf("%d%d", &n1, &n2);
   printf("In Main values before swapping: %d %d\n\n", n1, n2);
   swap(n1, n2);
   printf("In Main values after swapping: %d %d", n1,
   n2); return 0;
}
void swap(int num1, int num2)
{
   int temp;
   printf("In Function values before swapping: %d %d\n", num1, num2);
   temp = num1;
   num1 = num2;
   num2 = temp;
   printf("In Function values after swapping: %d %d\n\n", num1, num2);
}
```

Output -

Enter two numbers: 10 20
In Main values before swapping: 10 20
In Function values before swapping: 10 20
In Function values after swapping: 20 10
In Main values after swapping: 10 20

NOTE : In the above program swap() function does not alter actual parameter value. Before passing the value of *n1* and *n2* to the swap() function, the C runtime copies the value of actual parameter *n1* and *n2* to a temporary variable and passes copy of actual parameter. Therefore inside the swap() function values has been swapped, however original value of *n1* and *n2* in main() function remains unchanged.

Call by reference

In Call by reference we pass memory location (reference) of actual parameter to formal parameter. It uses pointers to pass reference of an actual parameter to formal parameter. Changes made to the formal parameter immediately reflects to actual parameter.

Eg Program - C program to swap two numbers using call by reference

```
#include <stdio.h>
int main()
{   int n1, n2;
   printf("Enter two numbers: ");
   scanf("%d%d", &n1, &n2);
   printf("In Main values before swapping: %d %d\n\n", n1,
   n2); swap(&n1, &n2);
   printf("In Main values after swapping: %d %d", n1,
   n2); return 0;
}
void swap(int * num1, int * num2)
{
   int temp;
   printf("In Function values before swapping: %d %d\n", *num1,
   *num2); temp = *num1;
   *num1 = *num2;
   *num2 = temp;
   printf("In Function values after swapping: %d %d\n\n", *num1, *num2);
}
```

Output :-

Enter two numbers: 10 20
In Main values before swapping: 10 20
In Function values before swapping: 10 20
In Function values after swapping: 20 10
In Main values after swapping: 20 10

In above example instead of passing a copy of *n1* and *n2*, to swap() function. Operations performed on formal parameter is reflected to actual parameter (original value). Hence, actual swapping is performed inside swap() as well as main() function.

WRITE THESE SWAP PROGRAM(S) FOR YOUR EXAMS

Call by value

```
#include<stdio.h>
Void swap(int x,int y);
int main( )
{
   int a = 10, b = 20 ;
   swap (a,b) ; // calling by value
   printf ( "\n Before swapping x and y)
   ; printf ( "\na = %d b = %d", a, b ) ;
   return 0;
}
void swap( int x, int y )
{
   int t ;
   t = x ;
   x = y ;
   y = t ;
   printf ( "\n After swapping x and y) ;
   printf( "\nx = %d y = %d", *x,*y);
}
```

```
Before swapping x and y
10 20
After swapping x and
y 20 10
```

Call by reference

```
#include<stdio.h>
Void swap(int *x,int *y);
int main( )
{
   int a = 10, b = 20 ;
   swap ( &a, &b ) ; // calling by
   reference printf ( "\n Before swapping x
   and y) ; printf ( "\na = %d b = %d", a, b )
   ; return 0;
}
void swap( int *x, int *y )
{
   int t ;
   t = *x ;
   *x = *y ;
   *y = t ;
   printf ( "\n After swapping x and y) ;
   printf( "\nx = %d y = %d", *x,*y);
}
```

```
Before swapping x and y
10 20
After swapping x and
y 20 10
```

Call by value	**Call by reference**
When a function is called the actual values are passed	When a function is called the address of values(arguments) are passed
The parameters passed are normal variables	The parameters passed are pointer variables
In call by value, actual arguments cannot be modified.	Modification to actual arguments is possible within from called function.
Actual arguments remain preserved and no chance of modification accidentally.	Actual arguments will not be preserved.
Calling Parameters: swap(a,b)	Calling Parameters: swap(&a,&b)
Receiving parameters: void swap(int x, int y)	Receiving parameters: void swap(int *x, int *y)

PASSING ARRAYS TO A FUNCTION

Passing Array Element to a Function

```
#include <stdio.h>
void display(int a);
int main()
{
int age[] = { 21, 31, 41 };
 display(age[2]); //Passing array element age[2] only.
return 0;
}
void display(int age)
 {
 printf("%d", age);
 }
```

output

41

Passing Entire Array(1-Dimentional array) Element to a Function

```
#include <stdio.h>
float average(float age[]);
int main()
 {
float avg, age[] = { 30.5,40.5,60.5,80.5};
avg = average(age);
printf("Average age=%.2f",
 avg); return 0;
 }
 float average(float age[])
 {
    int i;
    float avg, sum = 0.0;
    for (i = 0; i < 4; ++i)
            {
        sum += age[i];
            } avg = (sum /
4); return avg;
 }
```

Output

53.000000

Passing Entire Array(2-Dimentional array) Element to a Function

```
#include <stdio.h>
void displaynumbers(int num[2][2]);
int main( )
{
  Int num[2][2],i,j;
   Printf("enter numbers");
for(i=0;i<2;i++)
for(i=0;i<2;i++)
      scanf("%d",&num[i][j]);
displaynumbers(num)   // calling function
return 0;
}
void displaynumbers(int num[2][2])  / / function declaration
{
  int i,j;
  Printf("display numbers");
  for(i=0;i<2;i++)
  for(i=0;i<2;i++)
    printf("%d",num[i][j]);
}
```

Output

Enter numbers
10
20
30
40
Display numbers
10
20
30
40

RECURSION FUNCTION

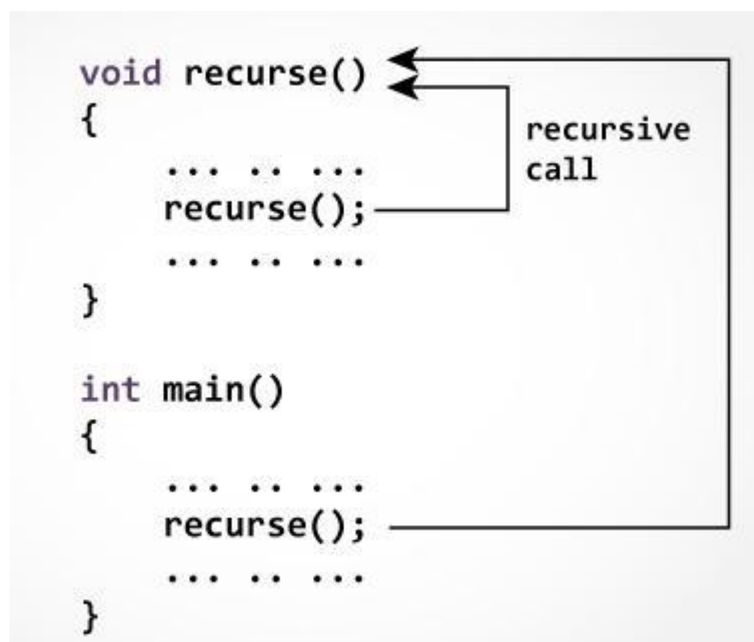
A function that calls itself is known as a recursive function. And, this technique is known as recursion.

Advantage of Recursion

- Function calling related information will be maintained by recursion.
- Stack evaluation will be take place by using recursion.
- In fix prefix, post-fix notation will be evaluated by using recursion.

Disadvantage of Recursion

- It is a very slow process due to stack overlapping.
- Recursive programs can create stack overflow.
- Recursive functions can create as loops.

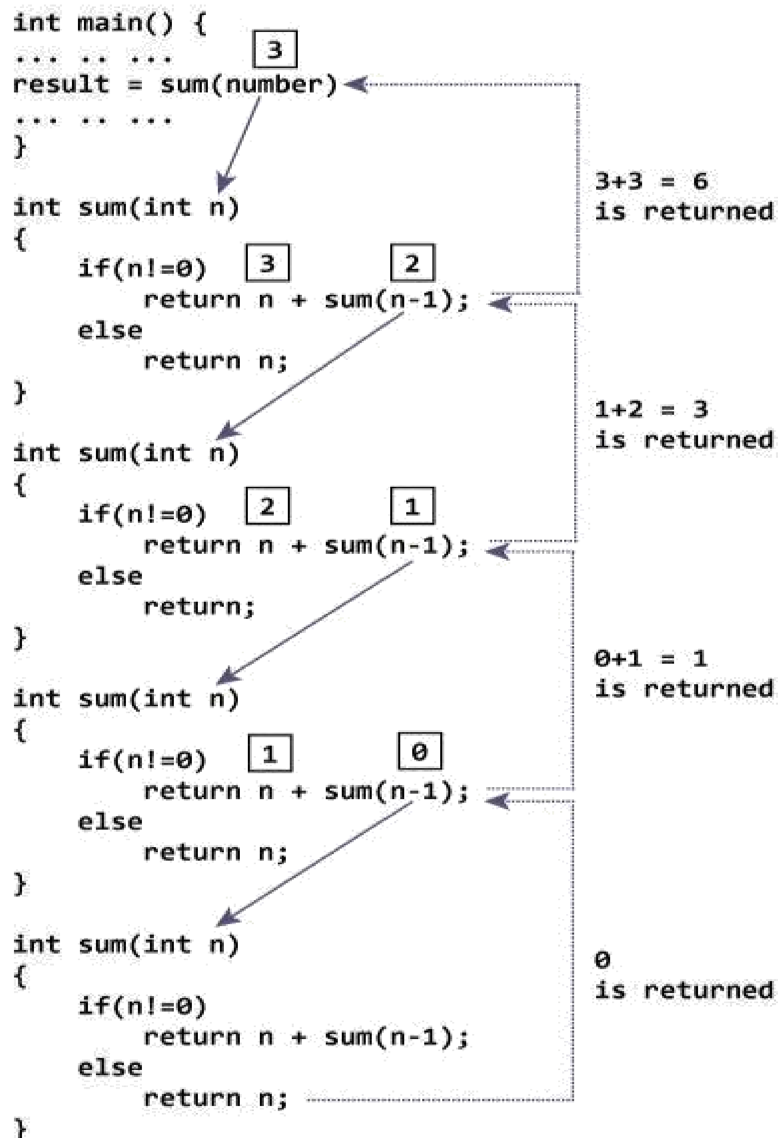


Example: Sum of Natural Numbers Using Recursion

```
#include <stdio.h>
int sum(int n);
int main()
{
    int number, result;
    printf("Enter a positive integer: ");
    scanf("%d", &number);
    result = sum(number);
    printf("sum=%d", result);
}
int sum(int num)
{
    if (num!=0)
        return num + sum(num-1); // sum() function calls
    itself else
        return num;
}
```

Output

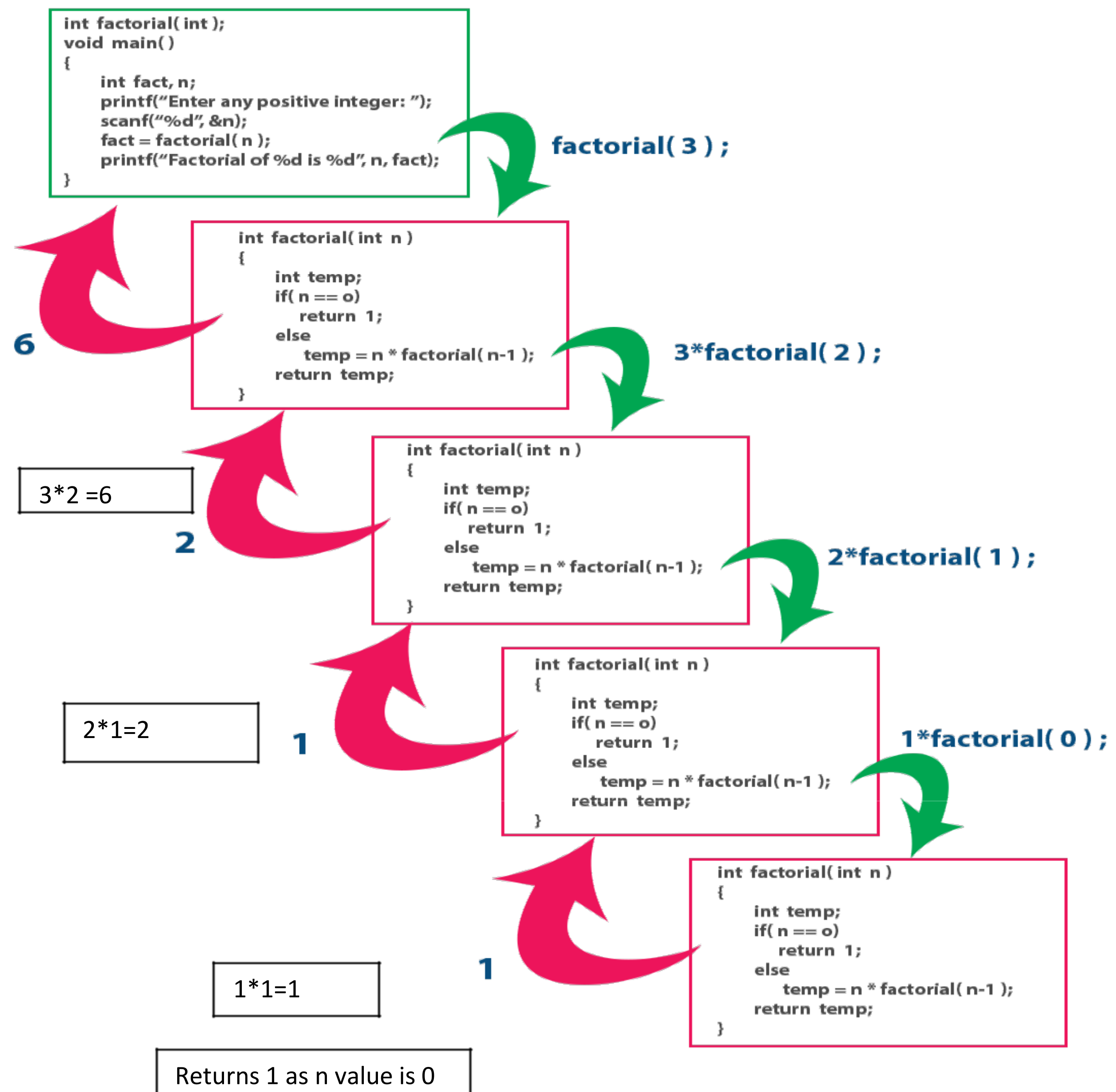
```
Enter a positive integer:3
6
```



FACTORIAL OF A NUMBER USING RECURSIVE FUNCTION

```
#include <stdio.h>
int factorial( int ) ;
void main( )
{
        int fact, n ;
        printf("Enter any positive integer: ") ;
        scanf("%d", &n) ;
        fact = factorial( n ) ;
        printf("Factorial of %d is %d", n, fact) ;
}
int factorial( int n )
{
        int temp ;
        if( n == o)
            return 1 ;
        else
            temp = n * factorial( n-1 ) ; // recursive function call
        return temp ;
}
```

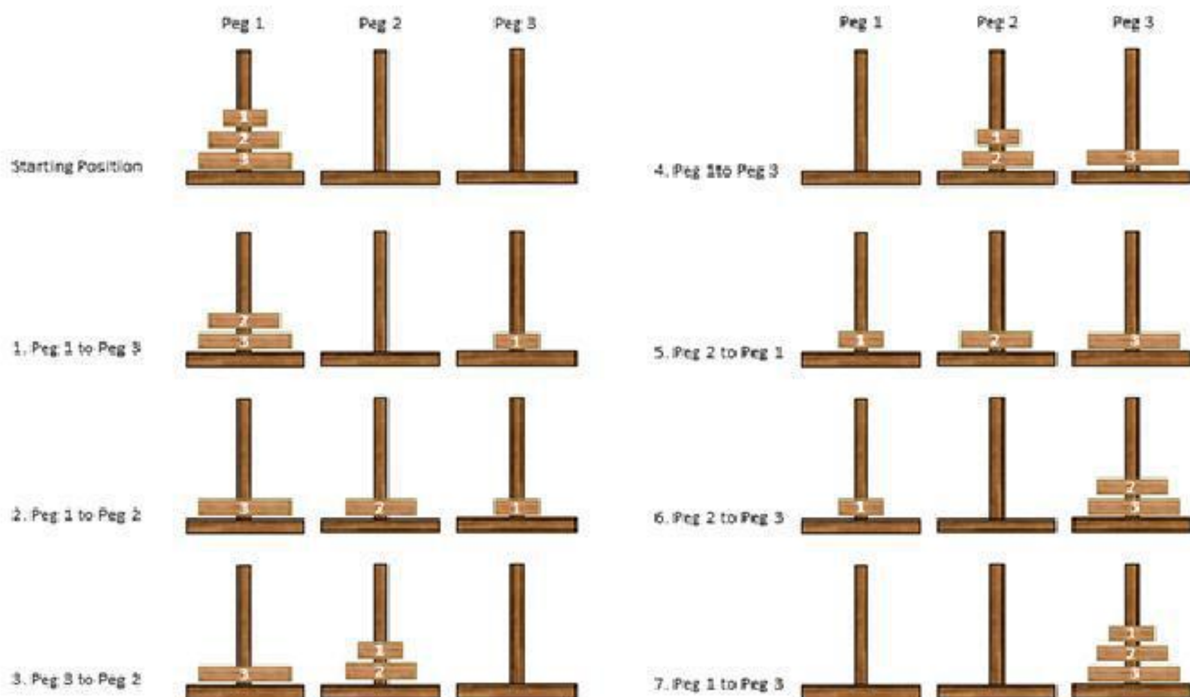
Enter any positive integer: 3 Factorial of 3 is 6



TOWER OF HANOI OF A NUMBER USING RECURSIVE FUNCTION

```
#include <stdio.h>
void hanoi(int n, char from, char to, char temp)
{
   if (n == 1)
   {
      printf("\n Move Disk 1 from Peg %c to %c", from, to);
      return;
   }
   hanoi(n-1, from, temp, to);
   printf("\n Move disk %d from rod %c to rod %c", n, fr,
   tr); hanoi(n-1, temp, to, from);
}
 int main()
{
   Printf(" Towers of Honoi");
   int n;
   printf("\n Enter number of Disks");
   scanf(" %d ",&n); // n implies the number of discs
   hanoifun(n, 'A', 'C', 'B'); // A, B and C are the name of Peg
   return 0;
}
```

To Transfer from disks from peg A to C taking help of B



Output

- Move Disk 1 from Peg 1 to Peg 3.
- Move Disk 2 from Peg 1 to Peg 2.
- Move Disk 1 from Peg 3 to Peg 2.
- Move Disk 3 from Peg 1 to Peg 3.
- Move Disk 1 from Peg 2 to Peg 1.
- Move Disk 2 from Peg 2 to Peg 3.
- Move Disk 1 from Peg 1 to Peg 3.

Example: GCD of Two Numbers using Recursion

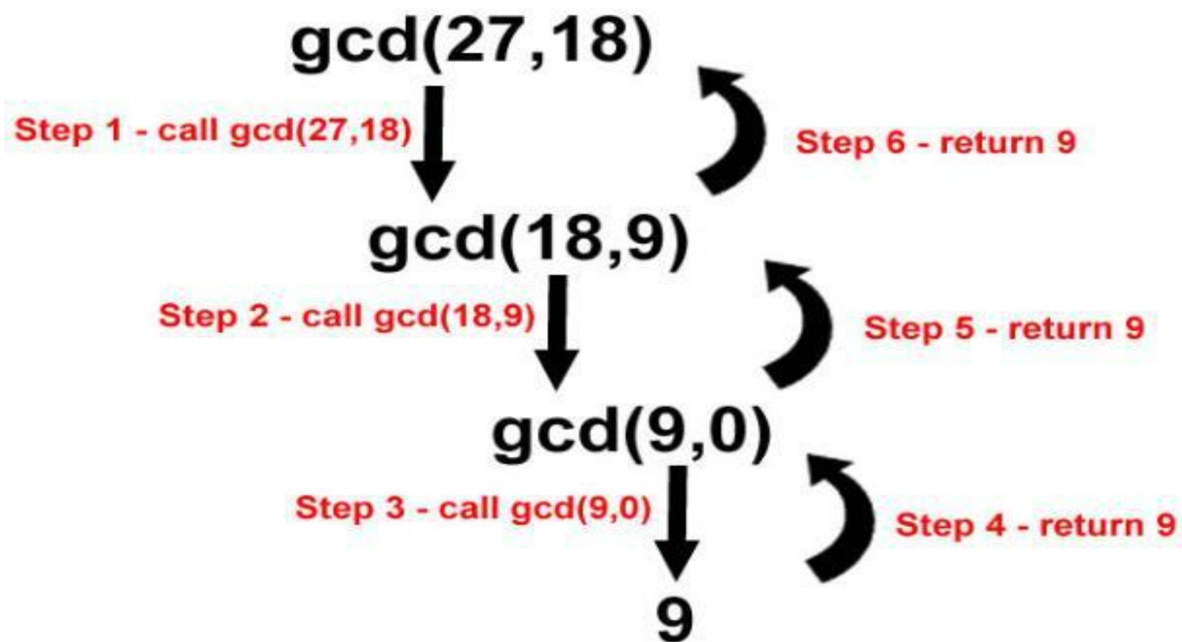
```
#include <stdio.h>
int hcf(int n1, int n2);
int main()
{
   int n1, n2;
   printf("Enter two positive integers:
   "); scanf("%d %d", &n1, &n2);

   printf("G.C.D of %d and %d is %d.", n1, n2,
   hcf(n1,n2)); return 0;
}

int hcf(int n1, int n2)
{
    if (n2 != 0)
       return hcf(n2,
    n1%n2); else
       return n1;
}
```

Output

```
Enter two positive integers: 27 18
G.C.D of 27 and 8 is 9.
```



Example: Fibonacci Series using Recursion

```
#include <stdio.h>
int fibonacci(int i)
{
   if(i == 0)
   {
      return 0;
   }
   if(i == 1)
   {
      return 1;
   }
   return fibonacci(i-1) + fibonacci(i-2);
}

int  main()
{
   int i,f;
   for (i = 0; i < 10; i++)
   {
     f= fibonacci(i);
     printf("%d \n ",f);
   }
   return 0;
}
```

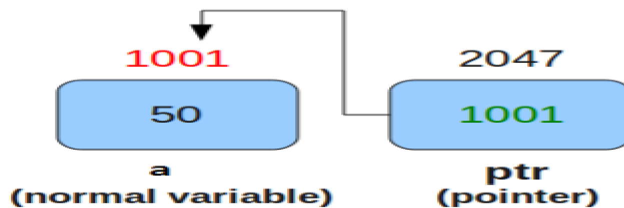
Output

```
0
1
1
2
3
5
8
13
21
34
```

Pointers in C Programming

A pointer is variable which stores address of another variable.

Pointers can only store addresses of other variable.




```
int x=10, y=20;
printf("%u %u", &x, &y);
```

Note :Here **%u** is a format specifier. It stands for unsigned, so it will only display positive values.

You will get output of the above program like below.
605893 605897

&-address of operator.

& is the "address of" operator. It is used to tell the C compiler to refer to the address of variables. Address of any variable can't be negative. This is the reason %u format specifier is used to print the address of variables on the screen.

value at address (*) Operator

This is the second operator used for pointers. It is used to access the value present at some address. And it is used to declare a pointer.

Declaration and initialization of pointers

```
int x=10;
int *ptr;  // Declaration of Pointer variable
ptr=&x; // Storing address of x variable in y pointer variable
```

Example program-1

```
#include<stdio.h>
 void main()
{
int a=6,b=12;
int *x,*y;
x=&a;
y=&b;
printf("%d t %d n",a,b);
printf("%u t %u n",&a,&b);
printf("%u t %u n",x,y);
printf("%d t %d n",*x,*y);
printf("%d t %d",(&a),(&b));
printf("%d t %d",*(&a),*(&b));
}
```

```
 6              12
6552465522
65524           65522
6               12
65524           65522
6               12
```

```
#include <stdio.h>
int main ()
{
   int  var = 20;                   /* actual variable declaration */
   int  *ip;                      /* pointer variable declaration */
   ip = &var;                    /* store address of var in pointer variable*/
   printf("Address of var variable: %x\n", &var  );
                              /*address stored in pointer variable*/
   printf("Address stored in ip variable: %x\n", ip );
                              /*access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );
   return 0;
}
```

A **pointer** is a variable whose value is the memory address of another variable

syntax

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable.

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

NULL Pointers

A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero.

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned.

```
#include <stdio.h>
int main ()
{
   int  *ptr = NULL;
   printf("The value of ptr is : %x\n", ptr);
    return 0;
}

output
The value of ptr is 0
```

Incrementing a Pointer(32-bit)

```
#include <stdio.h>
const int MAX = 3;
int main () {
   int  var[] = {10, 100, 200};
   int  i, *ptr;
                           /* let us have array address in pointer */
   ptr = var;
      for ( i = 0; i < MAX; i++) {
      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );
                           /* move to the next location */
      ptr++;
   }

   return 0;
}
```

output

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Decrementing a Pointer(32-bit machine)

decreases its value by the number of bytes of its data type.

```
#include <stdio.h>
const int MAX = 3;
int main () {
   int  var[] = {10, 100, 200};
   int  i, *ptr;
                        /* let us have array address in pointer */
   ptr = &var[MAX-1];
    for ( i = MAX; i > 0; i--) {
      printf("Address of var[%d] = %x\n", i-1, ptr );
      printf("Value of var[%d] = %d\n", i-1, *ptr );
                      /* move to the previous location */
      ptr--;
   }

   return 0;
}
```

output

```
Address of var[2] = bfedbcd8
Value of var[2] = 200
Address of var[1] = bfedbcd4
Value of var[1] = 100
Address of var[0] = bfedbcd0
Value of var[0] = 10
```

Program for pointer arithmetic(32-bit machine)

```
#include <stdio.h>
int main()
{
    int m = 5, n = 10, val = 0;
    int *p1;
    int *p2;
    int *p3;

    p1  = &m;    //printing the address of m
    p2  = &n;    //printing the address of n

    printf("p1 = %d\n", p1);
    printf("p2 = %d\n", p2);

    printf(" *p1 = %d\n", *p1);
    printf(" *p2 = %d\n", *p2);


    val = *p1+*p2;
    printf("*p1+*p2 = %d\n", val);//point 1

    p3 = p1-p2;
    printf("p1 - p2 = %d\n", p3); //point 2
```

```
    p1++;
    printf("p1++ = %d\n", p1); //point 3

    p2--;
    printf("p2-- = %d\n", p2); //point 4


    return 0;
}
```

OUTPUT

```
p1 = 2680016
p2 = 2680012

*p1=5;
*p2=10;

*p1+*p2 = 15


p1-p2 = 1
p1++ = 2680020
p2-- = 2680008
```

STRUCTURES

Structure is a user-defined data type that can store related information (of different data types) together. The major difference between a structure and an array is that an array can store only information of same data type. A structure is therefore a collection of variables under a single name. The variables within a structure are of different data types.

- A structure is a collection of variables under a single name. These variables can be of different types, Therefore, a structure is a convenient way of grouping together several pieces of related information.
- Complex hierarchies can be created by nesting structures.

- Declaration/Initializing/Accessing
- Nesting of Structures
- Arrays of Structures
- Structures and Pointers
- Structures and Functions

8.2.1 DECLARING STRUCTURES AND STRUCTURE VARIABLES

A structure is declared by using the **keyword struct** followed by an **optional structure tag** followed by the **body of the structure**. The ***variables* or *members*** of the structure are declared within the body.

The general format of declaring a simple structure is given as follows.

```
struct <structure_tag_name >{
<data_type member_name1>;
<data_type member_name2>;
      :   .
} <structure_variable1>,<structure_variable2>,...;
```

Keyword

Basic data type

There are three different ways to declare and/or define a structure. These are

- Variable structure
- Tagged structure
- Type-defined structure

1. A variable structure may be defined as follows.

```
struct
{
member_list
}variable_identifier;
```

```
struct
{
   int x;
   int y;
}a;
```

No tag name exist

members

variable identifier

```
struct
{
int r_no;
char name[20];
char course[20];
float fees;
} student1;
```

where..... a,student1-are called as structure variables

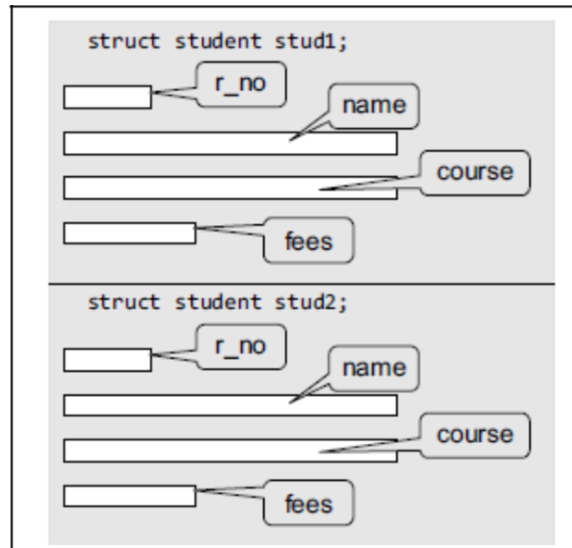
...

2.A <u>tagged structure</u> has been described earlier. It has the following format:

```
struct tag_name
{
member_list
}variable_identifier;
```

```
struct coordinate   <-- tag name
{
  int x;
  int y;
}a;
```

```
struct student
{
int r_no;
char name[20];
char course[20];
float fees;
} stud1, stud2;
```



Memory allocation

3.<u>Type-defined</u> structure declaration is as follows

- typedef keyword enables the programmer to create a new data type name by using an existing data type.

typedef existing_data_type new_data_type;

```
typedef struct newdatatype
{
member_list;

} newdatatype variable_identifier;
```

```
typedef struct student
{
int r_no;
char name[20];
char course[20];
float fees;
} student stud1;
```

When we precede a struct name with the **typedef** keyword, then the **struct** becomes a new type. **student** becomes a new data type. To declare a variable of structure student, you may write **student stud1;**

Note that we have not written **struct student stud1**.

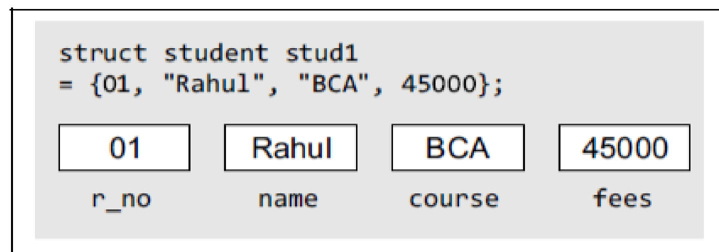
INITIALIZATION OF STRUCTURES

A structure can be initialized in the same way as other data types are initialized. Initializing a structure means assigning some constants to the members of the structure.

```
struct struct_name
{
data_type member_name1;
data_type member_name2;
data_type member_name3;
.......................
}struct_var = {constant1, constant2, constant3,...};
```

Example-1

```
struct student
{
int r_no;
char name[20];
char course[20];
float fees;
}stud1 = {01, "Rahul", "BCA", 45000};
```



ACCESSING THE MEMBERS OF A STRUCTURE

The members are accessed by relating them to the structure variable with a dot operator. The general form of the statement for accessing a member of a structure is as follows.

```
< structure_variable >.< member_name > ;
```

```
stud1.r_no
stud1.name
stud1.course
```

```
Stud2.r_no
Stud2.name
Stud2.course
```

Each member of a structure can be used just like a normal variable, but its name will be a bit longer. A structure member variable is generally accessed using a '.' (dot) operator.

```
stud1.r_no = 01;
stud1.name = "Rahul";
stud1.course = "BCA";
stud1.fees = 45000;
```

To input values for data members of the structure variable stud1, we may write

```
scanf("%d", &stud1.r_no);
scanf("%s", stud1.name);
```

Similarly, to print the values of structure variable stud1, we may write

```
printf("%s", stud1.course);
printf("%f", stud1.fees);
```

Example-2 both initializing and accessing member data

```
struct {
float p, q,
int r;
} k = {k .p = 3.0, k.q = 7.9, k.r = 5};
```

Write a program using structures to read and display the information about a student

```
#include <stdio.h>
struct student
{
int roll_no;
char name[80];
float fees;
char DOB[80];
};

int main()
{
struct student stud1;

printf("\n Enter the roll number : ");
scanf("%d", &stud1.roll_no);

printf("\n Enter the name : ");
scanf("%s", stud1.name);

printf("\n Enter the fees : ");
scanf("%f", &stud1.fees);

printf("\n Enter the DOB : ");
scanf("%s", stud1.DOB);

printf("\n ********STUDENT'S DETAILS *******");
printf("\n ROLL No. = %d", stud1.roll_no);
printf("\n NAME = %s", stud1.name);
printf("\n FEES = %f", stud1.fees);
printf("\n DOB = %s", stud1.DOB);
getch();
return 0;
}
```

Output

```
Enter the roll number : 01
Enter the name : Rahul
Enter the fees : 45000
Enter the DOB : 25–09–1991
*******STUDENT’S DETAILS *******
ROLL No. = 01
NAME = Rahul
FEES = 45000.00
DOB = 25–09–1991
```

C program to read and print employee's record using structure

```
#include <stdio.h>

 struct employee
{
   char    name[30];
   int     empId;
   float   salary;
};

int main()
{
   struct employee emp;             //  declare structure variable
   printf("\nEnter details :\n");
   printf("\n ************");
   printf("Name ?:");
   scanf("%s",emp.name);
   printf("ID ?:");
   scanf("%d",&emp.empId);
   printf("Salary ?:");
   scanf("%f",&emp.salary);

   /*print employee details*/

   printf("\n Employee detail is:");
   printf("\n ************");
   printf("Name: %s"   ,emp.name);
   printf("Id: %d"     ,emp.empId);
   printf("Salary: %f\n",emp.salary);
   return 0;
}
```

Output

```
Enter details :
**********
Name ?:Mike
ID ?:1120
Salary ?:76543

Employee detail is:
***************
Name: Mike
ID: 1120
Salary: 76543.000000
```

Write a program to copy and compare structures for employee details

```
// A structure can be assigned to another structure of the same type. Here is an
example of assigning one structure to another.

#include <stdio.h>
struct employee
{
char grade;
int basic;
float allowance;
};

int main()
{
struct employee ramesh={‘B’, 6500, 812.5}; /* creating & initializing
member of employee */

struct employee vivek;                          /* creating another member of
employee */

vivek = ramesh;                                 /* copy respective members of
ramesh to vivek */

printf(“\n vivek’s grade is %c, vivek.grade);
printf(“\n vivek’s basic is Rs %d, vivek. Basic);
printf(“\n vivek’s allowance is Rs %f”,  vivek.allowance);
return 0;
}
```

Output:
vivek’s grade is B
vivek’s basic is Rs 6500
vivek’s allowance is Rs
812.500000

Using Typedef :- A program that prints the weight of various sizes of fruits.

```
#include <stdio.h>
typedef struct fruits
{
float big;
float medium;
float small;
}fruits weight;

int main()
{
weight apples={200.75,145.5,100.25};
weight pears={150.50,125,50};
weight mangoes={1000, 567.25, 360.25};

printf("\n\n apples: big size %f kg, medium size %f kg, small size %f kg", apples.big,
apples.medium, apples.small);

printf("\n\n pears: big size %f kg, medium size %f kg,small size %f kg", pears.big,
pears.medium, pears.small);

printf("\n\n mangoes: big size %f kg, medium size %f kg,small size %f kg", mangoes.big,
mangoes.medium, mangoes.small);
return 0;
}
```

Output:

apples: big 200.75kg, medium 145.50kg, small 100.25kg
pears: big 150.50kg, medium 125.00kg, small 50.00kg
mangoes: big 1000kg, medium 567.25kg, small 360.25kg

Nesting of Structures

A structure can be placed within another structure. In other words, structures can contain other structures as members. A structure within a structure means nesting of structures.

In such cases, the dot operator in conjunction with the structure variables are used to access the members of the innermost as well as the outermost structures.

```
typedef struct name
{
char  first_name[20];
char  mid_name[20];
char  last_name[20];
}NAME;

typedef  struct dob
{
int dd;
int mm;
int yy;
}DATE;

typedef struct student
{
int r_no;
NAME name;
char course[20];
DATE DOB;
float fees;
} STU;
```

assign values to the structure fields, we will write

```
student stud1;
stud1.r_no = 01;
stud1.name.first_name = "Janak";
stud1.name.mid_name = "Raj";
stud1.name.last_name = "Thareja";
stud1.course = "BCA";
stud1.DOB.dd = 15;
stud1.DOB.mm = 09;
stud1.DOB.yy = 1990;
stud1.fees = 45000;
```

Write a program to read and display the information of a student using a nested structure.

```
#include <stdio.h>
struct DOB
{
int day;
int month;
int year;
};

struct student
{
int roll_no;
char name[100];
float fees;
struct DOB date;
};

int main()
{
struct student stud1;

printf("\n Enter the roll number : ");
scanf("%d", &stud1.roll_no);

printf("\n Enter the name : ");
scanf("%s", stud1.name);

printf("\n Enter the fees : ");
scanf("%f", &stud1.fees);

printf("\n Enter the DOB : ");
scanf("%d %d %d", &stud1.date.day, &stud1.date.month, &stud1.date.year);

printf("\n ********STUDENT'S DETAILS *******");
printf("\n ROLL No. = %d", stud1.roll_no);
printf("\n NAME = %s", stud1.name);
printf("\n FEES = %f", stud1.fees);
printf("\n DOB = %d – %d – %d", stud1.date.day, stud1.date.month, stud1.date.year);
getch();
return 0;
}
```

Output

```
Enter the roll number : 01
Enter the name : Rahul
Enter the fees : 45000
Enter the DOB : 25 09 1991
*******STUDENT’S DETAILS *******
ROLL No. = 01
NAME = Rahul
FEES = 45000.00
DOB = 25 – 09 – 1991
```

Write A program to demonstrate nesting of structures and accessing structure members.

```
#include <stdio.h>
struct outer                          /* declaration of outer structure */
{
int out1;                             /* member of outer structure */
float out2;                           /* member of outer structure */

struct inner                          /* declaration of inner structure */
{
int in1;                              /* member of inner structure */
float in2;                            /* member of inner structure */
}invar;                    /* structure_variable of inner structure*/
};

int main()
{
struct outer outvar;                  /* declaring structure_variable of outer */
outvar.out1= 2;                       /* assigning values to members */
outvar.out2= 10.57;
outvar.invar.in1= 2* outvar.out1;              /* assigning values to members */
outvar.invar.in2= outvar.out2 + 3.65;

printf(" out1=%d, out2=%f, in1=%d, in2=%f",outvar.out1, outvar.out2,outvar.invar.in1,
outvar.invar.in2);
return 0;
}
```

Output:

out1=2, out2= 10.57, in1=4, in2= 14.22

Write a program to read, display, add, and subtract two complex numbers.

```
#include <stdio.h>
#include <conio.h>
int main()
{
typedef struct complex
{
int real;
int imag;
}COMPLEX;
COMPLEX c1, c2, sum_c, sub_c;

int option;

do
{
printf("\n ******** MAIN MENU *********");
printf("\n 1. Read the complex numbers");
printf("\n 2. Display the complex numbers");
printf("\n 3. Add the complex numbers");
printf("\n 4. Subtract the complex
numbers"); printf("\n 5. EXIT");
printf("\n Enter your option : ");

scanf("%d", &option);

switch(option)
{
case 1:
printf("\n Enter the real and imaginary parts of the first complex number :
"); scanf("%d %d", &c1.real, &c1.imag);
printf("\n Enter the real and imaginary parts of the second complex number : ");
scanf("%d %d", &c2.real, &c2.imag);
break;
case 2:
printf("\n The first complex number is : %d+%di", c1.real,c1.imag);
printf("\n The second complex number is : %d+%di",
c2.real,c2.imag); break;
case 3:
sum_c.real = c1.real + c2.real;
sum_c.imag = c1.imag + c2.imag;
printf("\n The sum of two complex numbers is : %d+%di",sum_c.real,
sum_c.imag); break;
case 4:
sub_c.real = c1.real – c2.real;
sub_c.imag = c1.imag – c2.imag;
printf("\n The difference between two complex
numbers is :%d+%di", sub_c.real, sub_c.imag);
break;
}
}while(option != 5);
return 0;
}
```

Output

```
******** MAIN MENU *********
1. Read the complex numbers
2. Display the complex numbers
3. Add the complex numbers
4. Subtract the complex numbers
5. EXIT

Enter your option : 1
Enter the real and imaginary parts of the first
complex number : 4 5
Enter the real and imaginary parts of the second
complex number : 2 3
Enter your option : 2
The first complex numbers is : 4+5i
The second complex numbers is : 2+3i
Enter your option : 3
The sum of two complex numbers is : 6+8i
Enter your option : 4
The difference between two complex numbers is : 2+2i

Enter your option : 5
```

ARRAYS OF STRUCTURES

Arrays of structures means that the structure variable would be an array of objects, each of which contains the member elements declared within the structure construct.

Why would need an array of structures

1. In a class, we do not have just one student. But there may be at least 60 students. So, the same definition of the structure can be used for all the 30 students. This would be possible when we make an array of structures.
2. Another example where an array of structures is desirable is in case of an organization. An organization has a number of employees. So, defining a separate structure for every employee is not a viable solution. So, here we can have a common structure definition for all the employees.

Syntax

```
struct struct_name
{
data_type  member_name1;
data_type  member_name2;
data_type member_name3;
.......................
};
struct struct_name struct_var[index];
```

```
struct student
{
int r_no;
char name[20];
char course[20];
float fees;
};
```

A student array can be declared by writing,

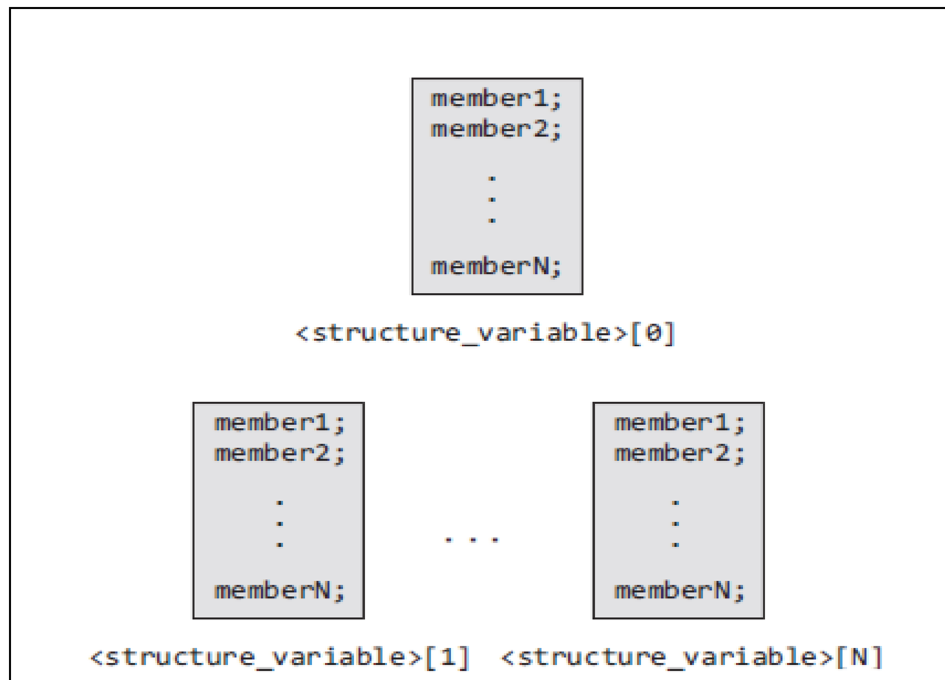
```
struct student stud[30];
```

Now, to assign values to the `ith` student of the class, we can write as

```
stud[i].r_no = 09;
stud[i].name = "RASHI";
stud[i].course = "MCA";
stud[i].fees = 60000;
```

In order to initialize the array of structure variables

```
struct student stud[3][4] = {{01, "Aman", "BCA", 45000},{02,
"Aryan", "BCA", 60000}, {03,"John", "BCA", 45000}};
```



Write a program to print the tickets of the boarders of a boat using array of structures with initialization in the program.

```
#include <stdio.h>

struct boat // declaration of structure // {

char name[20];
int seatnum;
float fare;
};

int main()
{
int i;
struct boat ticket[4][3]={{"Vikram",1,15.50},{"Krishna",2,15.50},
                          {"Ramu",3,25.50},{"Gouri",4, 25.50 } };
printf("\n passenger Ticket num. Fare");
for(i=0;i<=3;i++)
printf("\n %s %d %f ", ticket[i].name,ticket[i].seatnum,ticket[i].fare);
return 0;
}
```

Output:

Passenger	Ticket num.	Fare
Vikram	1	15.500000
Krishna	2	15.500000
Ramu	3	25.500000
Gouri	4	25.500000

C program to generate salary slip of employees using structures

```
#include<stdio.h>
struct emp
{
   int empno ;
   char name[10] ;
   int bpay, allow, ded, npay ;
} e[10] ;

void main()
{
   int i, n ;

   printf("Enter the number of employees : ") ;
   scanf("%d", &n) ;

    for(i = 0 ; i < n ; i++)
   {
      printf("\nEnter the employee number : ")
      ; scanf("%d", &e[i].empno) ;

      printf("\nEnter the name : ") ;
      scanf("%s", e[i].name) ;

      printf("\nEnter the basic pay, allowances & deductions : ") ;
      scanf("%d %d %d", &e[i].bpay, &e[i].allow, &e[i].ded) ;

      e[i].npay = e[i].bpay + e[i].allow - e[i].ded ;
   }
   printf("\nEmp. No.\t  Name \t  Salary \n") ;
   printf("\n *************************") ;
   for(i = 0 ; i < n ; i++)
   {
      printf("%d \t   %s \t  %f \t ", e[i].empno,e[i].name, e[i].npay) ;
   }
   return 0;
}
```

```
Enter the employee number  2

Enter the employee number: 1001
Enter the employee number: Rina
Enter the basic pay, allowances & deductions: 75000 10000 2000

Enter the employee number: 2001
Enter the employee number: Bina
Enter the basic pay, allowances & deductions: 80000 10000 3000

Emp.No.     Name        NetSalary
***************************
  1001       Rina          83000
  2001       Bina          87000
```

Write a program to read and display the information of all the students in a class. Then edit the details of the ith student and redisplay the entire information.

```
#include <stdio.h>
#include <string.h>
struct student
{
int roll_no;
char name[80];
int fees;
char DOB[80];
};
int main()
{
struct student stud[50];
int n, i, num, new_rollno;
int new_fees;
char new_DOB[80], new_name[80];
clrscr(); // clear screen
printf("\n Enter the number of students : ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n Enter the roll number : ");
scanf("%d", &stud[i].roll_no);

printf("\n Enter the name : ");
gets(stud[i].name);

printf("\n Enter the fees : ");
scanf("%d",&stud[i].fees);

printf("\n Enter the DOB : ");
gets(stud[i].DOB);
}

for(i=0;i<n;i++)
{
printf("\n ********DETAILS OF STUDENT %d*******",
i+1); printf("\n ROLL No. = %d", stud[i].roll_no);
printf("\n NAME = %s", stud[i].name);
printf("\n FEES = %d", stud[i].fees);
printf("\n DOB = %s", stud[i].DOB);
}

printf("\n Enter the student number whose record has to be edited :
"); scanf("%d", &num);

printf("\n Enter the new roll number : ");
scanf("%d", &new_rolno);

printf("\n Enter the new name : "):
gets(new_name);

printf("\n Enter the new fees : ");
scanf("%d", &new_fees);
```

```
printf("\n Enter the new DOB : ");
gets(new_DOB);

stud[num].roll_no = new_rollno;
strcpy(stud[num].name, new_name);
stud[num].fees = new_fees;
strcpy (stud[num].DOB, new_DOB);

for(i=0;i<n;i++)
{
printf("\n ********DETAILS OF STUDENT %d*******", i+1);
printf("\n ROLL No. = %d", stud[i].roll_no); printf("\n NAME =
%s", stud[i].name);
printf("\n FEES = %d", stud[i].fees);
printf("\n DOB = %s", stud[i].DOB);
}
getch();
return 0;
}
```

Output

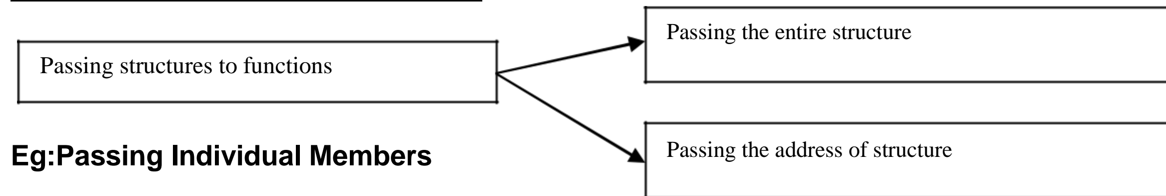
```
Enter the number of students : 2
Enter the roll number : 1
Enter the name : kirti
Enter the fees : 5678
Enter the DOB : 9- 9- 99

Enter the roll number : 2
Enter the name : kangana
Enter the fees : 5678
Enter the DOB : 27- 8- 99
********DETAILS OF STUDENT 1*******
ROLL No. = 1
NAME = kirti
FEES = 5678
DOB = 9- 9- 99
********DETAILS OF STUDENT 2*******
ROLL No. = 2
NAME = kangana
FEES = 5678
DOB = 27- 8 -99

Enter the student number whose record has to be edited : 2
Enter the new roll number : 2
Enter the new name : kangana khullar
Enter the new fees : 7000
Enter the new DOB : 27- 8 -92

********DETAILS OF STUDENT 1*******
ROLL No. = 1
NAME = kirti
FEES = 5678
DOB = 9 -9 -99
********DETAILS OF STUDENT 2*******
ROLL No. = 2
NAME = kangana khullar
FEES = 7000
DOB = 27- 8 -92
```

STRUCTURES AND FUNCTIONS



Eg:Passing Individual Members

```
#include <stdio.h>
typedef struct
{
  int x;
  int y;
}POINT;

void display(int, int);    // function declaration

int main()
{
POINT p1 = {2, 3};
display(p1.x, p1.y); // function call
return 0;
}

void display(int a, int b)    // function definition
{
printf(" The coordinates of the point are: %d %d", a, b);
}
```

Output
The coordinates of the point are: 2 3

Eg:Passing the Entire Structure

```
#include <stdio.h>
typedef struct
{
int x;
int y;
}POINT;

void display(POINT);
int main()
{
POINT p1 = {2, 3};
display(p1);
return 0;
}
void display(POINT p)
{
printf("The coordinates of the point are: %d %d", p.x, p.y);
}
```

- **Passing Structure by Value**
- **Passing Structure by Reference**

Passing Structure by Value

In this approach, the structure object is passed as function argument to the definition of function, here object is reperesenting the members of structure with their values.

Program

```
#include<stdio.h>

    struct Employee
    {
        int Id;
        char Name[25];
        int Age;
        long Salary;
    };

    void Display(struct Employee);

    void main()
    {
        struct Employee Emp = {1,"Kumar",29,45000};

        Display(Emp);

    }

    void Display(struct Employee E)
    {
            printf("\n\nEmployee Id : %d",E.Id);
            printf("\nEmployee Name : %s",E.Name);
            printf("\nEmployee Age : %d",E.Age);
            printf("\nEmployee Salary : %ld",E.Salary);
    }
```

Output :

```
Employee Id : 1
Employee Name : Kumar
Employee Age : 29
Employee Salary : 45000
```

Passing Structure by Reference

In this approach, the reference/address structure object is passed as function argument to the definition of function.

Program

```
#include<stdio.h>

    struct Employee
    {
        int Id;
        char Name[25];
        int Age;
        long Salary;
    };

    void Display(struct Employee*);
    void main()
    {
        struct Employee Emp = {1,"Kumar",29,45000};

        Display(&Emp);

    }

    void Display(struct Employee *E)
    {
            printf("\n\nEmployee Id : %d",E->Id);
            printf("\nEmployee Name : %s",E->Name);
            printf("\nEmployee Age : %d",E->Age);
            printf("\nEmployee Salary : %ld",E->Salary);
    }
```

Output :

```
        Employee Id : 1
        Employee Name : Kumar
        Employee Age : 29
        Employee Salary : 45000
```

Passing Structures through Pointers

```
struct struct_name
{
data_type member_name1;
data_type member_name2;
data_type member_name3;
......................
}*ptr;
```

or,

```
struct struct_name *ptr;
```

we can declare a pointer variable by writing

struct student *ptr_stud, stud;

The next thing to do is to assign the address of stud to the pointer using the address operator (&).

ptr_stud = &stud;

To access the members of a structure, we can write

(*ptr_stud).roll_no;

(->)This operator is known as ‘pointing-to’ operator

ptr_stud -> roll_no = 01;

Write a program to initialize the members of a structure by using a pointer to the structure.

```
#include <stdio.h>

struct student
{
int r_no;
char name[20];
char course[20];
int fees;
};

int main()
{
struct student stud1, *ptr_stud1;

ptr_stud1 = &stud1;

printf("\n Enter the details of the student :");
printf("\n *************************”);

printf("\n Enter the Roll Number =");
scanf("%d", &ptr_stud1 -> r_no);

printf("\n Enter the Name = );
gets(ptr_stud1 -> name);

printf("\n Enter the Course = ");
gets(ptr_stud1 -> course);

printf("\n Enter the Fees = ");
scanf("%d", &ptr_stud1 -> fees);

printf("\n DETAILS OF THE STUDENT");
printf("\n *************************”);

printf("\n ROLL NUMBER = %d", ptr_stud1 –> r_no);
printf("\n NAME = %s", ptr_stud1 –> name); printf("\n
COURSE = %s", ptr_stud1 –> course); printf("\n
FEES = %d", ptr_stud1 –> fees);
return 0;
}
```

Output

```
Enter the details of the student:
************************
Enter the Roll Number = 02
Enter the Name = Aditya
Enter the Course = MCA
Enter the Fees = 60000

DETAILS OF THE STUDENT
************************
ROLL NUMBER = 02
NAME = Aditya
COURSE = MCA
FEES = 60000
```

Storage Classes in C++ Programming

Storage class of a variable defines the **lifetime and visibility** of a variable. Lifetime means the duration till which the variable remains active and visibility defines in which module of the program the variable is accessible. They are:

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

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero
Register	register	Function Block	Local	Garbage

1. Automatic Storage Class

This variable is **visible** only within the function it is declared and its **lifetime** is same as the lifetime of the function as well. This is the default storage class we have been using so far. It applies to local variables only and the variable is visible only inside the function in which it is declared and it dies as soon as the function execution is over. If not initialized, variables of class auto contains garbage value.

- The value is lost after the execution of function.

Syntax: auto datatype var_name1 [= value];

Example:

```
int var; // by default, storage class is auto
auto int var;   // auto int j=10;
```

Example Program:

```
#include<stdio.h>

void display();
void main()
{
       auto int a=10;       //OR int a=10;
       printf("\n A1 : %d",a);
       display();
       printf("\n A3 : %d",a);
}
void display()
{
       int a=20;               //OR auto int a=20;
       printf("\n A2 : %d",a);
}
```

Output :

```
A1:10
A2:20
A3:10
```

2. External Storage Class

External storage class reference to a **global variable** declared outside the given program. *extern* keyword is used to declare external variables. They are **visible** throughout the program and its **lifetime** is same as the lifetime of the program where it is declared. This visible to all the functions present in the program.

```
Syntax : extern datatype var_name1;
 Example: extern float a;
```

The extern keyword is optional, there is no need to write it.

- The scope of external variable is the entire program.
- If not initialized external variable is assigned a zero value.
- The value is not lost after the execution of function.

Example Program:

```
#include<stdio.h>

void display();
extern int a=10; //global variable void main()

{
       printf("\nA : %d",a);
       increment();
       display();
       printf("\nA : %d",a);

}
void increment()
{
       a = 20;
}
void display()
{
       printf("\nA : %d",a);
}
```

Output :

```
A:10
A:20
A:20
```

3. Static Storage Class

Static storage class ensures a variable has the **visibility** mode of a local variable but **lifetime** of an external variable. It can be used only within the function where it is declared but destroyed only after the program execution has finished. The default initial value of static variable is **0**. The value of a static variable persists between function calls
.

Syntax: static datatype var_name1 [= value];
Example: static int x = 101;
static float sum;

During multiple calling static variables retains their previous value.

- We must declare variable as static.
- Static variables can't be accessed outside the function.
- If not initialized static variables have zero as initial value.

Example of static storage class

```
#include<stdio.h>
void display();
void main()
{
        display();
        display();
        display();
    }
void display()
{
        static int a=1;
        printf("\nA : %d",a);
        a++;
}
```

Output:

A : 1
A : 2
A : 3

In the above example, we does not use static keyword then the output will be :

```
Output :                                                    A : 1
                A : 1
                A : 1
```

4. Register Storage Class

Variables of class 'register' are stored in CPU registers instead of memory which allows faster access. It has its lifetime and visibility same as automatic variable. The scope of the variable is local to the function in which it is defined and it dies as soon as the function execution is over. It contains some garbage value if not initialized. The purpose of creating register variable is to increase access speed and makes program run faster. As register space is very limited, so only those variables which requires fast access should be made register It is declared as:

Syntax: register datatype var_name1 [= value];
Example: register int count
register char a;

C program to create automatic, global(extern) variables.

```
#include<stdio.h>
 void main()
  {
     register int a=10;
      printf("\nA : %d",a);
}
```

```
Output :

 A:10
```

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

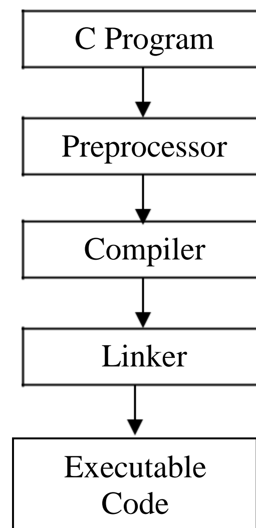


Fig. 2.8 Working Procedure of C Program

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.

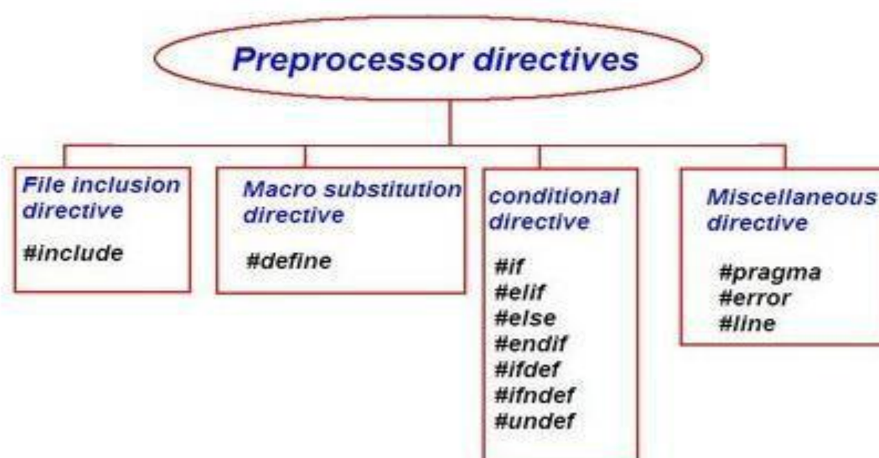


Fig Preprocessor Directives

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:

1. Simple macro
2. macro with arguments

Simple macro

Syntax:

#define identifier value

Where

#define	- is apreprocessor 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 captital 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;
#else
        statements;
#endif
```

Where

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

#ifdef #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;
}
```

Output

Pragma verification

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.

1. **#pragma startup**
2. **#pragma exit**
3. **pragma warn**
4. **#pragma option**
5. **#pragma inline**
6. **#pragma argsused**
7. **#pragma hdrfile**
8. **#pragma hdrstop**
9. **#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
700
701
702