FUNCTIONS UNIT - 3

→ Library Functions
→ Need for user-defined functions
→ A multi-function program
→ Elements of user-defined functions
→ Definition of functions
→ Return values and their types
→ Function calls
→ Function declaration
→ Category of functions
→ Nesting of functions
→ Recursion
→ Passing arrays to functions
→ The Scope, visibility and Lifetime of variables
→ Multifile programs
→ Prepnocesson commands

Functions:-

A function is a self-contained block of one or more statements that performs a particular task.

Example : scanf() → scanf() function reads the input
clrscr() function is used to clear the screen.

⇒ A 'C' program is a collection of functions.

⇒ Basically functions are divided into 2 types:

1. Library functions:

Library functions are predefined functions which are defined in C library. Examples: printf()
scanf()
clrscr()
pow()
sqrt() etc

→ Library functions are developed by the developers.
→ As a programmer we can't modify the library functions.
→ All the library functions are pre-defined in the headerfiles. If you are using any library functions, include the corresponding header file using "#include".

2. User-defined functions:-

The functions which are defined by the user or programmer are known as user-defined functions.

Example : main()

LIBRARY FUNCTIONS:

⇒ Library functions are pre-defined functions.

⇒ The main advantage is: code Reuse → Reusing the functions which are already been written and tested.

⇒ C allows reuse by providing many predefined functions that can be used to perform mathematical computations.

⇒ Some of the library functions are:

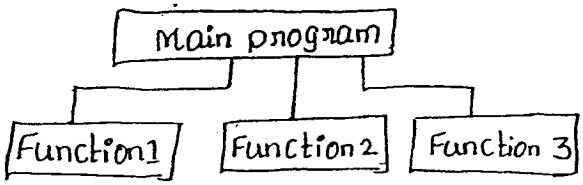
Function	Standard header file	Purpose	Example	Argu ty
abs(x)	<stdlib.h>	Returns the absolute value of its integer argument	If x = -5 abs(x) = 5	int
ceil(x)	<math.h>	Returns the smallest integral value that is not less than x	If x = 45.23 ceil(x) = 46.00	dou
floor(x)	<math.h>	Returns the largest integral value that is not greater than x.	If x = 45.23 floor(x) = 45.00	doul
sqrt(x)	<math.h>	Returns the non-negative square root of x (i.e. $\sqrt{x}$) for x ≥ 0	Ex: If x = 4.0 sqrt(x) = 2.0	doul
pow(x,y)	<math.h>	Return x^y	If x = 2, y = 3 pow (x,y) = 8	doub d
fabs(x)	<math.h>	Returns the absolute value of its type double argument	If x = -8.432 fabs(x) = 8.432	doul
exp(x)	<math.h>	Returns e^x, where e = 2.71828	If x = 1 exp(x) = 2.71828	dou
log(x)	<math.h>	Returns the natural logarithm of x for x > 0.	If x = 2.71828 log(x) = 1.0	dou
log10(x)	<math.h>	Returns the base-10 logarithm of x for x > 0	If x = 100.0 log10(x) = 2.0	dou
sin(x)	<math.h>	Returns the sine of angle x.	Note: The angle must expressed in radian	
cos(x)	<math.h>	Returns the cosine of angle x.		
tan(x)	<math.h>	Returns the tangent of angle x		

⇒ We already know that every C program must have a main function to indicate where the program has to begin its execution. It is possible to write any program using only main functio. But it leads to number of problems

- The program may become too large and complex
- The task of testing, debugging and maintaining becomes difficult

To avoid this, using functions large programs can be divided into subprograms. These subprograms are coded independently and later combine into a single unit. It is easy to understand, debug and test the subprograms

⇒ There are situations, where certain type of operations or calculation are repeated



⇒ There are situations, where certain type of operations or calculations are repeated at many points throughout a program.

For example consider, in a program we might use factorial of a numbe at several points in the program. we may repeat the program statements whenever they are needed. It increases program code.

To reduce the program code write the particular block of statements in a user-defined function and call the function whenever required. This reduces both program size and time.

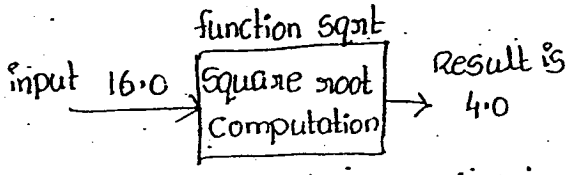
⇒ The main advantage is code Reusability. i.e. A function may be used by many other programs and any number of time.

⇒ It is easy to locate and isolate the faulty functions.

A MULTI-FUNCTION PROGRAM:

⇒ Once a function has been designed and packed, it can be treated as a 'black-box'.

Function takes some data from the main program and returns a value. The inner details of function are invisible to the rest of the program



All that the program knows about a function is: what goes in & what comes out.

⇒ Every C program is a collection of one or more functions.

⇒ For example, consider the following function printline():

```
void printline()
{
   int i;
   for(i=0; i<20; i++)
   printf("-");
   printf("\n");
}
```

The above function print a line of 20 character length. This function can be used in a program as follows:

```
main()
{
   void printline(); /* Function declaration */
   clrscr();
   printline(); /* Function call */
   printf("This shows the use of C functions\n");
   printline(); /* Function call */
   getch();
}
```

This program will print the following output:

```
--------------------
This shows the use of C functions
--------------------
```

The above program contains 2 user-defined functions: main() function
printline() function

We know that the program execution always begins with the main() functio During execution of the main, it calls printline function, which indicates that the function printline is to be executed. At this point, the working of calling function is stopped and the program control is transferred to the called function printline. After executing the printline function, the control is transferred back to the main and continues the execution. After execut the printf() function, again it calls printline(). Now the control is again transferred to the printline function for printing the line.

⇒ Any function can call any other function. A 'called function' can also call another function.

```
Example: void main()
         {
           void message();
           void message1();
           clrscr();
           message();
           getch();
         }
         void message()
         {
           printf("Hai");
           message1();
         }
         void message1()
         {
           printf("\n Hello");
         }
```

① call → void message()

② call → void message1()

③ Return

Sample output:

```
Hai
Hello
```

In the above program, main function calls message(), which calls message1 i.e. a called function can also call another function.

⇒ A function can be called more than once.

```
Example: main()
         {
           void message();
           clrscr();
           message(); /* Function call */
           message(); /* Function call */
           message(); /* Function call */
           getch();
         }
         void message()
         {
           printf("\n This is a message");
         }
```

Output : This is a message
This is a message
This is a message

⇒ A function can call itself.

Ex:
```
main()
{
  printf("Hai");
  main();   → Here the main function calls itself.
}
```

Important Note:

→ A function can be called from any other function, but a function can't be defined inside a another function.

Ex:
```
main()
{
  printf("Hai");
void message()
{
  printf("This is a message");
}
}
```
} You should n't define a function inside another function.

→ The order in which the functions are defined in a program and the order in which they get called need not be the same.

Ex:
```
main()
{
  message1();
  message2();
}
message2()
{
  printf("\n This is message2");
}
message1()
{
  printf("\n This is message1");
}
```

ELEMENTS OF USER-DEFINED FUNCTIONS:

⇒ Functions are classified as one of the derived data types in C.

⇒ Like variables functions should declared and defined before using them in a program.

Similarities between variables and functions in C:

1. Both function names and variable names must be valid identifiers. so they must follow the rules.
2. Like variables, functions also have types(such as int, char) associated with them
3. Like variables, function names and their types must be declared and defined before they are used in a program.

How to use User-defined functions:-

To make use of user-defined functions, we need to establish 3 elements related to functions: 1. Function definition.
2. Function call.
3. Function declaration

FUNCTION DEFINITION:-

Function definition is also known as function implementation. Function definition consists of set of statements that are specially written to implement a particular task.

A general format of function definition is:

Syntax:

```
function-type function_name(parameter list)   => Function header
{
    local variable declaration;
    executable statement1;
    executable statement2;
    ----------
    ----------
    return statement;
}
```

=> Function Body

Function Header:

The function header consists of 3 parts:

1. Function type (also known as return type)
2. Function name
3. Parameter list

1. Function type:

The function type specifies the type of value returned by the function (like int or float or double).

If the function is not returning anything, then we need to specify the return type as void.

If the return type is not explicitly specified, C assumes that by default all functions returns an integer value.

The value returned is the output produced by the function.

2. Function name:

The function name is a valid C identifier. Therefore while giving function names you should follow the rules. The function name should be related to the task performed by the function.

3. Parameter list:

The parameter list declares the variables that will receive the data sent by the calling program.

Parameters are also known as arguments. Parameters serve as input data to the function to carry out the specified task.

The parameter list contains declaration of variables seperated by con

Example: ① `int sum(int a, int b)`
`{`
`------`
`------`
`}`

The above function sum takes two integer values as input and returns the result.

② `float quadratic(int a, int b, int c)`
`{`
`------`
`------`
`}`

⇒ A function need not always receive values from the calling program. In such cases, the parameter list will be empty. To indicate that the parameter list is emptly, we use void keyword.

Ex: `void sum(void)` (or) `void sum()` → This value doesn't return any value

`{`
`------`
`}`

→ Empty. It doesn't take any input.

FUNCTION BODY:-

The function body contains the declarations and statements necessary for performing the required task.

The function body contains:

1. Local variable declarations:
   A local variable is a variable that is declared inside a function.
2. Executable statements:
   - Set of statements that performs the task of the function.
3. Return statement:
   A return statement that returns the value.

Example: ①
```
void display(void)
{
    printf("No type, no paramenters");   /* No local variables */
                                         /* No return statement
}
```

②
```
int sum(int a, int b)
{
    int c;  /* Local variable declaration */
    c = a+b; /* Executable statements */
    return c; /* Returns the result */
}
```

RETURN VALUES AND THEIR TYPES

A function may or may not send back any value to the calling function. If a function send any value, it done through the return statement.

The return statement can take one of the following forms.

1. return;
   The above statement doesn't return any value, it acts as the closing brace of the function.
2. return(expression);
   The above statement returns the value of the expression.

Ex:
```
int sum(int a, int b)
{
    int c;
    c = a+b;
    return c;
}
```
The above function returns the value of c

(OR)
```
int sum(int a, int b)
{
    return(a+b);
}
```
↳ This is also valid.

A function may have more than one return statements. This situation arises when the value returned is based on certain condition.

Example:
```
if(a>b)
    return a;
else
    return b;
```

⇒ When a function reaches its return statement, the control is transferred back to the calling function.

FUNCTION CALL:-

⇒ To use any user-defined function, we need to call the function at a required place in the program.

⇒ A function can be called by simply using the function name followed by a list of parameters if any, enclosed in parenthesis.

Example:
```
sum(2,3);
sum();
sum(a,b);
sum(10,b);
sum(10+5,12);
sum(sum(2,3),6);
```

Note: The parameters used in the function call may be values, variables or expressions.

⇒ When the compiler encounters a function call, the control is transferred to the called function. The called function is then executed line by line.

FUNCTION DECLARATION:-

⇒ Like variables, all functions in a C program must be declared before they are invoked.

⇒ Function declaration is also known as function prototype.

Syntax: function_type functionname(parameter list);

⇒ A function prototype tells the C compiler the following things:

1. The type of value returned by the function.
2. The function name
3. Information about the parameters or arguments.

Ex:
```
void addition(int x, int y); /*Function prototype*/
void mul(int m, int n);
int largest(int a, int b);
float Average(int a, int b, int c);
```

The following example shows the working of functions.

```
void main()
{
    int sum(int, int);  /* Function declaration */
    int x;
    x = sum(10,5);      /* Function call */
    printf("In Sum is: %d", x);
    getch();
}
int sum(int a, int b)   /* Function definition */
{
    int c;
    c = a+b;
    return c;
}
```


Explanation:-

Calling function:-

The function which is calling another function is known as calling function.

Called function:

The function which is invoked (i.e. called) by the calling function is known as called function.

In the above example main() is the calling function
sum() is the called function

Formal arguments (or) Formal parameters:-

The parameters used in function declaration and function definition are called formal parameters (or) formal arguments.

Actual arguments (or) Actual parameters:

The parameters used in function call are called actual parameters (or) actual arguments.

⇒ The formal and actual parameters must match exactly in type, order and number. Their names need not be same.

⇒ The values of actual arguments are assigned to the formal arguments on a one to one basis, starting with the first argument.

WORKING OF FUNCTIONS:-

We know that program execution always starts with a main() function During the execution of a program, whenever a function is called the working of the calling function is stopped and the control is transferred to the called function.

The called function is then executed line by line. When the execution c the called function is completed or when it reaches return statement, the control is transferred back to the calling function and continues the executi

The values of the actual arguments passed by the calling function are received by the formal arguments of the called function. The called function operates on formal arguments.

NOTE:-

1. If the actual arguments are more than the formal arguments, then the extra actual arguments are discarded.
2. If the actual arguments are less than the formal arguments, then the unmatched formal arguments are initialized to some garbage values.
3. Any mismatch in data type may also result in passing of garbage values.
   → No error message will be generated.
4. The formal arguments must be valid variable names. The actual argume may be variable names, expressions or constants.
5. The variables used in actual arguments must be assigned values befor they are passed.

CATEGORY OF FUNCTIONS:-

A function depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories:

1. Functions with no arguments and no return values
2. Functions with arguments and no return value
3. Functions with arguments and return value
4. Functions with no arguments but return value
5. Functions that return multiple values.

1. FUNCTIONS WITH NO ARGUMENTS AND NO RETURN VALUE:-

⇒ When a function has no arguments, it doesn't receive any data from the calling function.

⇒ Similarly, when the called function doesn't return any value, the calling function doesn't receive any data from the called function.

⇒ There is no data transfer between the calling function and the called function.

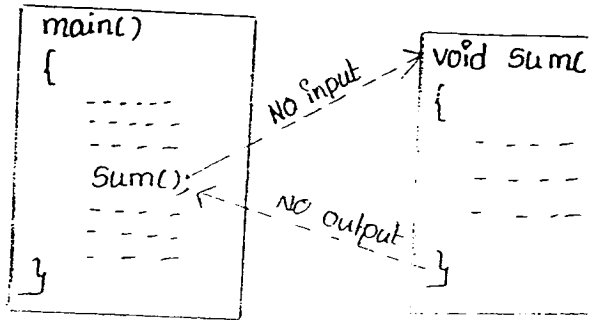


Figure: No data communication between functions

Example:- Program to find Sum of two numbers using functions with no arguments and no return value.

```
#include<stdio.h>
Void main()
{
    Void sum();
    clrscr();
    sum();
    getch();
}
Void sum()
{
    int a, b, c;
    Printf("Enter a, b values:");
    Scanf("%d %d", &a, &b);
    c=a+b;
    Printf("Sum is: %d", c);
}
```

Explanation:

→ In this program, the function sum() has no arguments. So it doesn't receive any data from the main function.

→ The function sum() doesn't return any value to the main() function. So return type is specified as void.

```
Void sum();
     └→ No arguments
└→ No return value
```

3. FUNCTIONS WITH ARGUMENTS BUT NO RETURN VALUE:-

⇒ In this case, the data are transferred from calling function to the called function.

⇒ The called function receives some data from the calling function. But it doesn't return any value back to the calling function.

Ex: void sum(int x, int y) → Receives the data (or) input from calling func
↓
Doesn't return any value.

⇒ The nature of communication between the calling function and the called function is as follows:

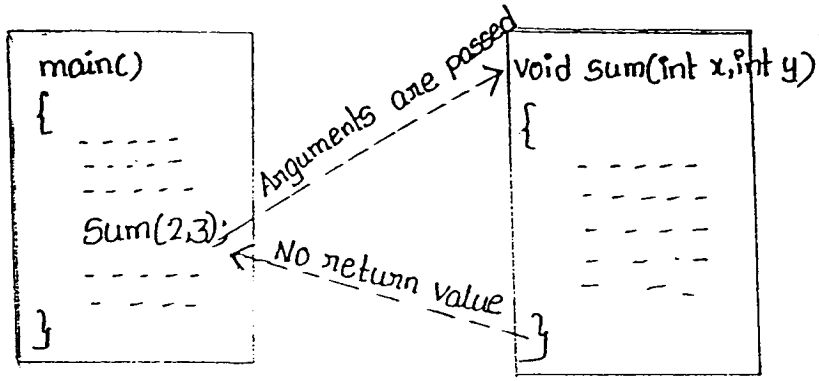


Figure: One-way data communication

Example:

```
/* Program to find the sum of two integers using function with arguments
   and no return value */
#include<stdio.h>
Void sum(int a, int b);
Void main()
{
   int x,y;
   clrscr();
   printf("Enter x,y values:");
   scanf("%d %d",&x,&y);
   sum(x,y);
   getch();
}
Void sum(int a, int b)
{
   int c;
   c=a+b;
   printf("Sum is: %d", c);
}
```

Explanation:-

→ In this program the function sum() receives 2 x,y values as input frc the main() function.

→ But the main() function doesn't receive any value from the sum();

FUNCTIONS WITH ARGUMENTS AND WITH RETURN VALUES:-

⇒ In this case, the data is transferred between the calling function and the called function.

⇒ The called function receives some data from the calling function and returns a value back to the calling function. There is a two-way data communication between the functions.

Ex: The function definition like as follows:

```
int sum(int x, int y)
```
→ Receives data from the calling function

↓

Returns a value back to the calling function

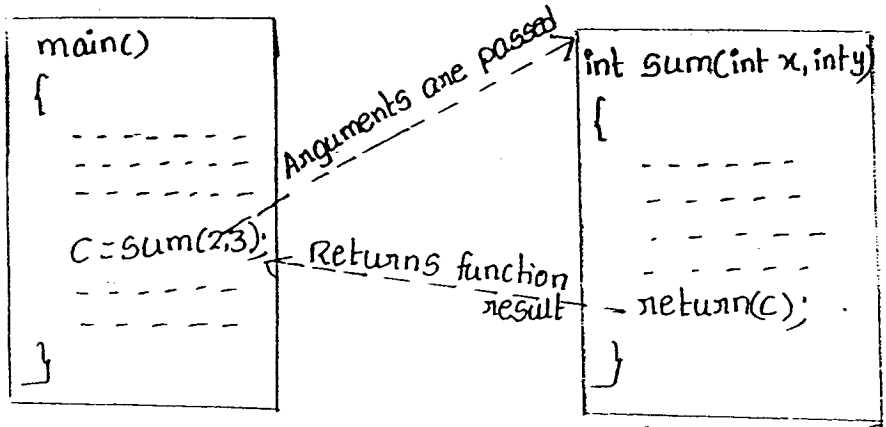


Figure:- TWO-WAY DATA COMMUNICATION BETWEEN FUNCTIONS

Example:

/* Program to calculate sum of two integers using functions with arguments and with return value */

```
#include<stdio.h>
Void main()
{
  int sum(int a,int b);
  int x, y, z;
  clrscr();
  printf("Enter x,y values:");
  scanf("%d %d",&x, &y);
  z = sum(x,y);
  printf("Sum is: %d", z);
  getch();
}
int sum(int a, int b)
{
    int c;
    c=a+b;
    return c;
}
```

Explanation:-

⇒ In this program, the function sum() receives two integer values as input from the main() function.

⇒ The function sum() calculates the sum of two integer values and returns result back to the main() function.

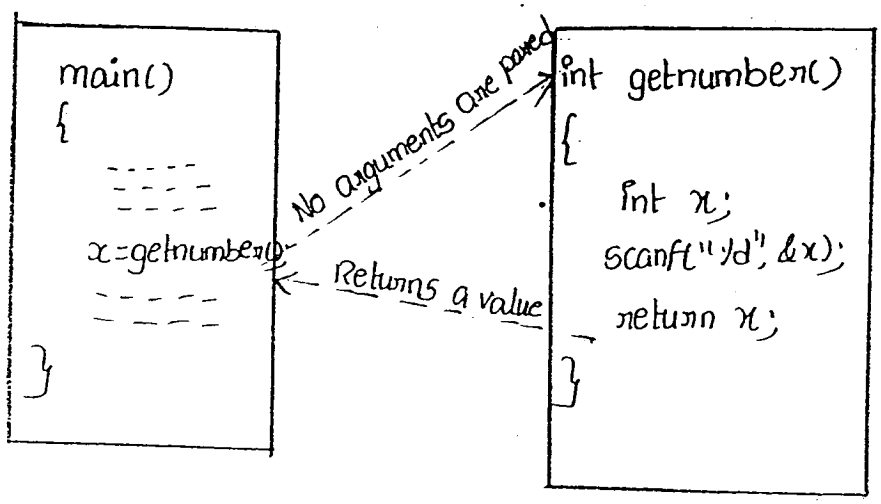
⇒ The main() function receives the returned value and that value is assigned to variable 'z'.

4. FUNCTIONS WITH NO ARGUMENTS BUT RETURN A VALUE:-

⇒ In this case, the called function doesn't take any arguments from the calling function, but returns a value to the calling function.

Ex: getchar() → getchar() function has no arguments but it returns a character.

Ex: int getnumber();
↓ → The function has no arguments
Function returns an integer value to the calling function.



Example program:-

```
#include<stdio.h>
void main()
{
   int getnumber();
   int a;
   a=getnumber();
   printf("a=%d", a);
   getch();
}
int getnumber()
{
    int x;
    printf("Enter a value:");
    scanf("%d", &x);
    return x;
}
```

NESTING OF FUNCTIONS:-

⇒ C allows nesting of functions.
⇒ Nesting of functions means calling one function inside another function

```
main()
{
    --------
    --------
    function1();
    --------
    --------
}
function2()
{
    --------
    --------
    function2();
    --------
    --------
}
function2()
{
    --------
    --------
    --------
}
```

⇒ Nesting of functions. function1() calls function2().

Figure: Nesting of functions

As shown in the above figure main function calls function1, which calls function2. There is no limit. Any function can call any another function.

Example:

Program to calculate the Ratio of $\frac{a}{b-c}$. The ratio cannot be evaluated if (b-c)=0.

```
float ratio(int x, int y, int z);
int difference(int x, int y);
void main()
{
    int a,b,c;
    clrscr();
    printf("Enter a,b,c values:");
    scanf("%d %d %d", &a, &b, &c);
    printf("Ratio is: %f", ratio(a,b,c));
}
```

```
float ratio(int x, int y, int z)
{
    if (difference(y,z)!=0)
        return (x/(y-z));
    else
        return (0.0);
}
int difference(int y, int z)
{
    if((y-z)!=0)
        return (1);
    else
        return (0);
}
```

Explanation:

The above program contains 3 functions : main()
ratio()
difference().

→ main() reads the values of a,b,c and calls the function ratio to calculate the value a/(b-c).

→ The ratio can't be evaluated if (b-c)=0. So ratio calls another function difference to test whether the difference (b-c) is zero or not.

→ The function difference return 1 if the difference (b-c) is not equal to zero, otherwise return zero to the function ratio.

→ The ratio calculates the ratio a/(b-c) value and returns the result if it receives 1. If the function receives '0' it returns 0.0 to main function.

Nesting of function calls:-

Nesting of function calls is also possible.

Example: `x = sum(sum(2,3),6);`

The above statement represents two sequential function calls. The inner function call is evaluated first (i.e. first it evaluates sum(2,3)) and the returned value is again used as actual argument in the outer function call. (ie. x=sum(5,6) = 11).

RECURSION:-

⇒A function that calls itself is known as recursive function and the process of calling function itself is known as recursion.

Ex: #include<stdio.h>
```
void main()
{
   printf("This is an example of recursion");
   main();
}
```

In the above program, main() function calls itself repeatedly infinite number of times. The user has to stop the execution abnormally (or) there must be some condition for stopping the execution -

⇒A useful example of recursion is the evaluation of factorial of a number.

Program:-

```
/* Factorial of a given number using recursion */
#include<stdio.h>
void main()
{
   int n;
   int factorial(int);
   clrscr();
   printf("Enter a number:");
   scanf("%d",&n);
   printf("The factorial of a given number is: %d", factorial(n));
   getch();
}
int factorial(int x)
{
   int fact;
   if(x==1)
      return 1;
   else
      fact = x*factorial(x-1);
   return fact;
}
```
→ /* Recursion */

Let us assume n=3.

Since n≠1 the statement fact=3*factorial(3-1) will be executed.

Again it includes a function call with n=2. This call will return 2*factoria

Once again the factorial is called with n=1. This time the function returns

⇒ The sequence of operations are:

fact = 3 * factorial(2)

= 3 * 2 * factorial(1)

= 3*2*1

= 6.

Advantages:-

1. Extremely useful when applying the same solution to the subsets of the problem.

~~3. Length~~

Disadvantages:-

1. Recursive functions can create infinite loops.
2. It requires extra storage space.
3. Recursive functions can create stack overflow.
4. If the programmer forgets to specify the exit condition in the recursive function, then the program will execute infinite number of time
5. It is difficult to trace the logic of the program.
6. It is difficult to debug the code containing recursion.
7. Often confusing.

PASSING ARRAYS TO FUNCTIONS:-

Like the values of simple variables, it is also possible to pass the elements of an array to a function.

Passing One-dimensional Arrays to functions:-

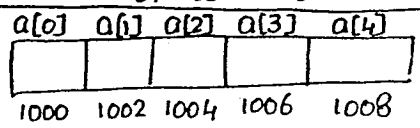
To pass a one-dimensional array to a called function, it is sufficient to pass the array name without any subscripts, and the size of the array as arguments.

Example: `int a[5] = {1, 2, 3, 4, 5};`

`largest(a,5);` → will pass the whole array a to the largest function
- 5 → size of the array
- a → Name of the array

⇒ In C, the name of the array represents the address of its first element.

Ex: `int a[5];`



a = &a[0] = 1000
a+1 = &a[1] = 1002
a+2 = &a[2] = 1004
a+3 = &a[3] = 1006
a+4 = &a[4] = 1008

→ By passing the array name, we are passing the address of the array to the called function. Any changes in the array in the called function will reflect the original array.

RULES:-

1. The function must be called by passing only the name of the array and its size.

   Ex: `largest(a, n);`

2. The function declaration must show that the argument is an array.

   Ex: `int largest(int a[], int n);`

3. In the function definition, the formal parameter must be an array type. The size of the array need not be specified in the square brackets.

   Ex:
   ```
   int largest(int array[], int n)
   {
   - - - - - -
   - - - - - -
   - - - - - -
   }
   ```

Example:-

```
/*Program to find largest element in an array */

#include<stdio.h>
Void main()
{
    int largest(int a[],int n); /*Function declaration */
    int value[5] = { 10, 15, 7, 25, 13};
    int large;
    Clrscn();
    large = largest(value, 5);
    printf("\n The largest element is: %d", large);
    getch();
}
int largest(int a[], int n)
{
    int i, max;
    max=a[0];
    for(i=1;i<n;i++)
    {
        if(max<a[i])
            max=a[i];
    }
    return max;
}
```

Explanation:-

When a function largest(value,5) is called, all elements of array 'value become the corresponding elements of array 'a' in the called function. The largest function finds the largest value in the array and returns the result to the main function.

TWO DIMENSIONAL ARRAYS TO FUNCTIONS:-

⇒Like one-dimensional arrays, it is also possible to pass 2D-arrays to Functions

RULES:-

1. In the function definition, we must indicate that the array has two-dimensions by including two sets of brackets. The size of second dimension must be specifi

Ex: int SumOfMatrix(int x[][N], int M, int N)
{
- - - - - -
- - - - - -
}

2. The function declaration should be similar to function header.

Ex: int SumofMatrix(int x[][N], int M, int N);

3. The function must be called by passing only the array name and the size of two dimensions.

Ex: int a[2][2] = {{1,2},{3,4}};
Sum = SumofMatrix(a, 2, 2);

Example program:-

```
/*Program to calculate Sum of all elements in a matrix*/
#include<stdio.h>
Void main()
{
    int SumofMatrix(int x[][N], int M, int N);
    int a[2][2] = {{1,3},{4,6}};
    int sum;
    clrscr();
    sum = SumofMatrix(a, 2, 2);
    printf("\n Sum of all elements in a matrix is : %d", sum);
    getch();
}
int SumofMatrix(int x[][N], int M, int N)
{
    int i, j, S=0;
    for(i=0; i<M; i++)
    {
        for(j=0; j<N; j++)
        {
            S = S + x[i][j];
        }
    }
    return S;
}
```

OUTPUT:-

Sum of all elements in a matrix is: 14

Example:-

```
/* Program to arrange array elements in ascending order */
#include<stdio.h>
void main()
{
  void sort(int x[], int m);
  int a[5] = {40, 90, 73, 81, 35};
  int i;
  clrscr();
  printf("\n Marks before sorting:");
  for(i=0; i<5; i++)
    printf("\t %d", a[i]);
  sort(a, 5); /* Function call (we are passing array name and its size */
  printf("\n Marks after sorting:");
  for(i=0; i<5; i++)
    printf("\t %d", a[i]);
  getch();
}
void sort(int x[], int m)
{
    int i, j, temp;
    for(i=0; i<m; i++)
    {
      for(j=i+1; j<m; j++)
      {
        if(x[i] > x[j])
        {
          temp = x[i];
          x[i] = x[j];
          x[j] = temp;
        }
      }
    }
}
```

OUTPUT:-

```
Marks before sorting:
  40  90  73  81  35
Marks after sorting:
  35  40  73  81  90
```

⇒ From the above program it is clear that, if a function changes the values of an array, then these changes will be made to the original array that passed to the function.

In C, array name represents the address of its first element. When an entire array is passed as an argument, the information about the addresses of array elements are passed to the called function. Therefore the called function refers to the original array stored in the memory. Therefore any changes made to the array elements will be reflected in the original array.

PASSING PARAMETERS TO FUNCTIONS

⇒ Parameters refer to the input given to a function.

⇒ The technique used to pass data from one function to another function is known as 'parameter passing'.

⇒ There are 2 ways to pass the parameters:

1. Pass by value (or) Call by value
2. Pass by address (or) Call by address (or) Call by reference (or) Pass by reference (or) Pass by pointers

1. Call by value (or) Pass by value:-

⇒ In call by value, the values of actual parameters are copied to the formal parameter list of the called function.

⇒ The called function works on the copy (i.e. on formal parameter list), but not on the original values of the actual parameters.

⇒ Any changes made to the formal parameters doesn't effect the actual parameters. This ensures that the original data in the calling function can't be changed accidently.

⇒ Changes made in the formal parameters are local to the block of called function.

Example:- Swapping of two values using call by value

```
#include<stdio.h>
Void main()
{
   void swap(int, int);
   int x, y;
   printf("Enter the values of x and y:");
   scanf("%d %d", &x, &y);
   swap(x,y); /* Calling the function by passing x,y values */
   printf("In main() function x=%d, y=%d", x, y);
   getch();
}
Void swap(int a, int b)
{
    int c;
    c=a;
    a=b;
    b=c;
    printf("In swap() function x=%d, y=%d", a, b);
}
```

Output:-

```
Enter the values of x and y: 5  10
In swap() function x=10, y=5
In main() function x=5, y=10
```

2. Pass by address (or) Call by address:-

⇒ In call by address, the memory addresses of the variables are passed to the called function. Function operates on addresses rather than values.

⇒ The called function directly works on the original data in the calling function

⇒ Any changes made in the formal parameters effect the actual parameters.

⇒ Here formal parameters are pointers to the actual parameters.

RULES FOR CALL BY ADDRESS:-

1. In call by address, the actual arguments in the function call must be the addresses of variables. Ex: swap(&x, &y) → address of variables
2. The formal arguments in the function header and function prototype must be prefixed by *. Ex: void swap(int *a, int *b);
3. To access the value of an actual argument in the called function, we must use the corresponding formal argument prefixed by *.

Example:

```
/* Swapping of two values using call by address */
#include<stdio.h>
Void main()
{
    void swap(int *, int *);
    int x,y;
    clrscr();
    printf("Enter values of x & y:");
    scanf("%d %d", &x, &y); /* addresses are passed */
    swap(&x, &y);
    printf("In main() function x=%d, y=%d", x, y);
    getch();
}
Void swap(int *a, int *b)
{
    int *c;
    *c = *a;
    *a = *b;
    *b = *c;
    printf("In swap function x=%d, y=%d", a, b);
}
```

Output:

```
Enter values of x &y: 10 5
In swap() x=5, y=10
In main() x=5, y=10
```

⇒ Any changes made in the arguments are permanent.

THE SCOPE, VISIBILITY AND LIFETIME OF VARIABLES:-

STORAGE CLASSES:-

⇒ In C, all the variables have a data type and a storage class.

⇒ The storage class of a variable gives the information about:

1. The location of the variable in which it is stored
2. The default initial value of the variable, if the initial value is not specifically assigned.
3. Scope of the variable i.e. in which block the variable is available (or) visible
4. Life time of the variable i.e. how long the variable would be in active mode.

⇒ There are 4 types of storage classes:-

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

1. AUTOMATIC STORAGE CLASS:-

⇒ Automatic variables are declared inside a function.

⇒ They are created when the function is called and destroyed automatically when the function is exited, hence the name automatic.

⇒ Automatic variables are local to the function in which they are declared. Because of this, automatic variables are also called local variables (or) internal variables.

⇒ To define a variable as automatic, the keyword auto is used.

Ex:
```
main()
{
  auto int m;
  auto float avg;
  .....
  .....
}
```

⇒ A variable declared inside a function, without any storage class is, by default an automatic variable.

Ex:
```
main()
{
  int m; /* Automatic variable */
  .....
  .....
}
```

⇒ By defining a variable as automatic storage class,

→ It is stored in memory.

→ The default value of the variable will be ganbage value.

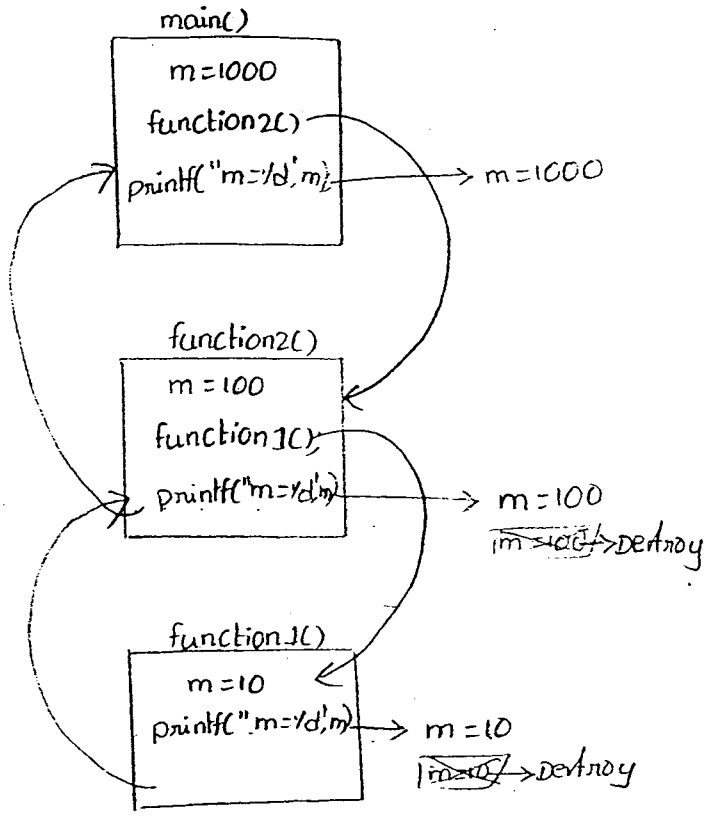
→ Scope of the variable is only within the function or block in which they are declared.

↳ The lifetime of the variable is until end of the function (or) block.

⇒ We may use the same variable name in different functions in the same progra without causing any confusion to the compiler.

Example program:-

```
void function1();
void function2();
main()
{
    int m=1000;
    function2();
    printf("m=%d",m);
}
function1()
{
    int m=10;
    printf("m=%d",m);
}
function2()
{
    int m=100;
    function1();
    printf("m=%d",m);
}
```



output: m=10
m=100
m=1000

Explanation:-

⇒ The main() function calls function2(), which in turn calls function1().

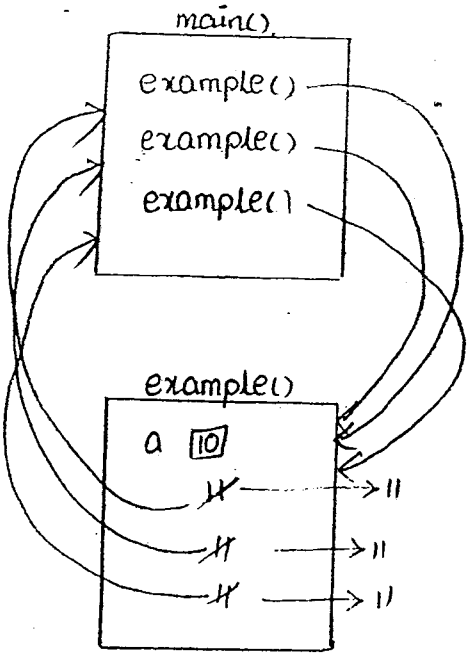
⇒ In function1() m=10, and this value is destroyed when it leaves the functio

⇒ In function2() m=100, and this value is destroyed when it leaves the functic

⇒ In main() function m=1000.

From this, it is clear that, the value of m is active when the control enters into the function, and destroyed automatically when the control leaves the function.

```
void example();
void main()
{
    example();
    example();
    example();
}
void example()
{
    auto int a;
    a=10;
    ++a;
    printf("\n %d", a);
}
```



OUTPUT:-
11
11
11

Explanation:-

⇨ The lifetime of the variable is only inside the function in which they are declared.

→ The main() function calls example()
→ When the function example() is called first time, the control goes to the example function a is initialized to '0'. Next 'a' is incremented by 1 (now a=11). This value is destroyed when the control leaves the function.
→ When example() is called second time, the control goes to the function definition initialize a=10 and increments its value by 1 (i.e. a=11) and this value is destroyed automatically when the control leaves the function

Example 3:-

```
void abc();
void main()
{
    abc();
    abc();
    printf("%d", a);
}
void abc()
{
    auto int a;
    a=30;
    ++a;
}
```

↳ Here 'a' is not accessible. Since the scope of the variable is only inside the function in which they are declared.

Output: Error → Undefined symbol 'a'.

REGISTER STORAGE CLASS:-

⇒ It is also possible to store variables in one of the machine's registers, instead of keeping them in the memory.

⇒ Register access is much faster than a memory access. So keeping the frequently accessed variables in the register will lead to faster execution of programs.

⇒ To define a variable as register storage class, the keyword register is used.

Example: register int x;
register char y;

⇒ When a variable is declared as a register variable,

→ It is stored in CPU Register.
→ The default value of the variable will be garbage value.
→ Scope of the variable is only within the function (or) block in which they are defined.
→ The lifetime of the variable is until end of function (or) block.

⇒ Only few variables can be placed in the registers. Most compilers allow only int or char variables to be placed in the register.

⇒ C will automatically convert register variables into non-register variable once the limit is reached.

Example program.

```
#include<stdio.h>
void main()
{
   register int i=1; /* i is declared as register variable */
   while(i<=10)
   {
     printf("\n%d",i);
     i++;
   }
   getch();
}
```

⇒ External variables are declared outside of all functions in the program.

⇒ External variables are also known as Global variables.

⇒ Global variables can be accessed by any function in the program.

⇒ When a variable is declared as global (or) extern

1. It is stored in memory.
2. The default value is initialized to zero
3. The scope of the variable is global.
4. The lifetime of the variable is until the program execution comes to an end.

Ex①:
```
int number;
float length;
main()
{
   ---------
   ---------
   number=10;
   length=1.5;
   ---------
}
function1()
{
   printf("%d", number);
}
function2()
{
   printf("%f", length);
}
```

Note: All the 3 functions can access the variables number & length.

⇒ If a local variable, and a global variable have the same name, the local variable will have highest precedence over the global one in which it is declared.

Ex②:-
```
int a=10;
void main()
{
   int a;
   a=1;
   printf("a=%d", a);
}
```

Output: a=1

⇒ When main() references the variable a, it will reference only its local variable, not the global one. So in main() function a=1.

⇒ Once a variable has been declared as global, any function can use it and change its value. Then the subsequent functions can reference only that new value.

Ex ③:-

```
void function1();
void function2();
void function3();
int x;
void main()
{
    x=10;
    printf("x=%d", x);
    function1();
    function2();
    function3();
}
void function1()
{
    x=x+10;
    printf("x=%d", x);
}
void function2()
{
    int x;
    x=1;
    printf("\n x=%d", x);
}
void function3()
{
        x=x+20;
        printf("\n x=%d", x);
}
```

OUTPUT:-

x=10
x=20
x=1
x=40

⇒ Another property of a global variable is that it is available only from the point of declarat to the end of the program.

Ex:

```
main()
{
    y=5;
    ----
    ----
    ----
}
int y; ⟶ /* Global variable */
function1()
{
    y=y+1;
}
```

Output: Error. The variable y is declared after th main() function. So it is not available in the main function

⇒ To avoid this, we have to specify y is a global variable explicitly by using the keywond extern

```
main()
{
    extern int y; ⟶ /* External declaration
    y=5;
    printf("y=%d", y);
    ----
}
int y; ⟶ /* Definition */
function1()
{
    y=y+1;
}
```

→ Although the variable y is defined after the ma function, the external declaration of y inside the function informs the compiler that y is a external variable.

Note: External declaration doesn't allocate storag space (i.e. memory) for variables.

⇒ A variable can be declared as static variable by using the keyword static.

Ex: static int x;
    static float y;

⇒ When a variable is declared as static,

1. It is stored in memory
2. The default value of the variable will be zero.
3. The scope of the variable is inside the function (or) block where it is defined.
4. The lifetime of the variable persists between function calls.

⇒ A static variable is initialized only once, when the program is compiled. It can't be reinitialized.

Example1:
```
#include<stdio.h>
void main()
{
   static int b;
   printf(" b=%d",b);
}
```
output: b=0. (∵ The default value of static variable is zero.)

Example 2:-
```
void start();
void main()
{
    start();
    start();
    start();
}
void start()
{
    static int x;
    x=x+1;
    printf("\n x=%d", x);
}
```
output:-
x=1
x=2
x=3

⇒ A static variable may be either an internal type (or) an external type depending on the place of declaration.

⇒ Internal static variables are those which are declared inside a function. The scope of internal static variables is up to the end of the function where it is defined.

⇒ An external static variable is declared outside of all functions and is available to all the functions in that program.

⇒ NOTE:-

Difference between a static external variable and a simple external variable is that static external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.

NESTED BLOCKS:-

⇒ A set of statements enclosed in a set of braces is known as a block or Compound statement.

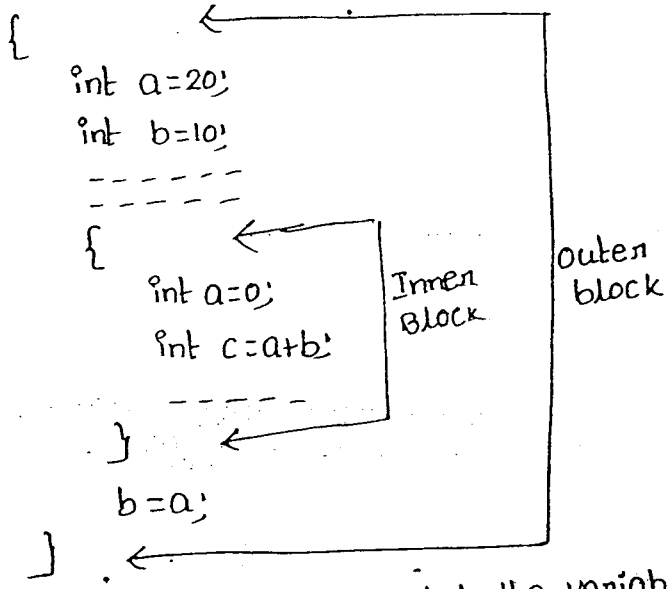
⇒ All the functions including the main() can use block of statements.

⇒ A block can have its own declarations and other statements.

⇒ It is also possible to have a block of statements inside the body of a function (or) another block.

Ex:

```
main()
{
    int a=20;
    int b=10;
    ------
    ------
    {
        int a=0;
        int c=a+b;
        -----
    }
    b=a;
}
```



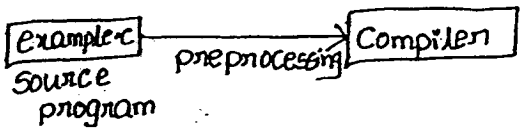
Note: When this program is executed, the variable c will be 10, not 30. The statement b=a; assigns b to 20, not zero.

PREPROCESSOR COMMANDS (OR) PREPROCESSOR DIRECTIVES:-

PREPROCESSOR COMMANDS:-

⇒ Preprocessor commands are the instructions that are executed before the source code passes through the compiler.

⇒ The program that process the preprocessor commands (or) directives is called a preprocessor.



⇒ Preprocessor directives are placed in the source program before the main() function.

⇒ Before the source code passes through the compiler, it is examined by the preprocessor for any preprocessor directives. If there are any preprocessor commands, appropriate actions are taken.

⇒ Each of the preprocessor directive begins with the symbol # and don't require a semicolon at the end.

⇒ Preprocessor commands can be placed anywhere in the program, but it is good practice to place them at the beggining.

Directive	Function
#define	– Defines a macro substitution
#undef	– Undefines a macro
#include	– Specifies the files to be included
#ifdef	– Test for a macro definition
#endif	– Specifies the end of #if
#ifndef	– Tests whether a macro is not defined.
#if	– Tests a compile-time condition
#else	– Specifies alternatives when #if test fails.

⇒ The preprocessor directives are broadly classified into 3 categories.

1. Macro substitution
2. File inclusion
3. Conditional compilation.

1. MACRO SUBSTITUTION DIRECTIVES:-

⇒ Macro substitution is a process where an identifier in a program is replaced by a predefined string

⇒ #define directive is used for this purpose.

Syntax: `#define identifier string`

If the above statement is included in the program, then the preprocessor replaces every occurance of the identifier in the source code by the string.

Identifier must be a valid name. String may be any text.

⇒ There are different forms of macro substitution:

1. Simple macro substitution
2. Argumented macro substitution
3. Nested macro substitution.

1. Simple macro substitution:

⇒ #define directive is used to define constant macros.

Ex: `#define PI 3.14`

During the preprocessing the preprocessor replaces every occurance of PI with 3.14

Ex:
```
#define  COUNT  100
#define  FALSE  0
#define  SIZE   50
```

Example program:-

```
#define A 10
void main()
{
    int i;
    i=A;
    printf("i=%d",i);
}
```

output: i=10

⇒ A macro definition can include more than one simple constant.

Ex:
```
#define AREA 5*25
#define SIZE 5+7
#define MAX sizeof(int)*4
```

Syntax: `#define identifier(f1,f2,....fn) String`

→ macro with arguments is known as macro call, which is similar to function call.

Ex: `#define CUBE(x) x*x*x`

→ Suppose if the following statement appears in the program

```
Volume = CUBE(3);
```

Then the preprocessor will expand this statement to volume = 3*3*3.

Example program:-

```
#include<stdio.h>
#define SQUARE(x) x*x
#define CUBE(x) x*x*x
void main()
{
    int x;
    x=SQUARE(2); /* x=2*2 */
    printf("Square of 2 is: %d", x);
    x=CUBE(2); /* x=2*2*2 */
    printf("Cube of 3 is: %d", x);
    getch();
}
```

3. Nested macro substitution:-

⇒ We can also use one macro in the definition definition of another macro.

Ex:
```
#define M 5
#define N M+1
```

Ex:
```
#define SQUARE(x) (x*x)
#define CUBE(x) (SQUARE(x)*x)
#define SIXTH(x) (CUBE(x)*CUBE(x))
```

⇒ For example the statement CUBE(x) #define SIXTH(x) is expanded to

```
(SQUARE(x)*x) * (SQUARE(x)*x)
```

Since SQUARE(x) is still a macro, it is further expanded into.

`((x*x)*x)*((x*x)*x)`, which finally evaluates x^6.

#undef:-

⇒A macro defined with #define directive can be undefined with #undef directive.

Syntax: #undef identifier

⇒It is useful when we don't want to allow the use of macros in any portion of the program.

Ex:
```
#include<stdio.h>
#define SIZE 100
void main()
{
   #undef SIZE
   int a[SIZE];
   int i;
   clrscr();
   for(i=1;i<SIZE;i++)
   {
     scanf("%d",&a[i]);
   }
}
```

FILE INCLUSION DIRECTIVE:-

⇒By using #include directive, we can include the files containing functions or macro definitions in the program so that no need to rewrite those function or macro definitions.

Syntax:
```
#include "filename"
   (OR)
#include <filename>
```

where filename is the name of the file containing the required functions.

→ The preprocessor inserts the entire contents of filename into the source code of the program.

⇒ When the filename is included within the double quotation marks, the search for the file is made in the current directory and then in the standard directories.

⇒ When the filename is included without double quotation marks, then the search for the file is made only in the standard directories.

```
#include<stdio.h>
#include <conio.h>
#include<math.h>
#include<stdlib.h>
#include<process.h>
#include<ctype.h>
#include<string.h>
```

CONDITIONAL INCLUSION DIRECTIVES:-

Some directives are used to compile a part of the program based on certain conditions.

The most commonly used conditional inclusion directives are:

1. #if
2. #else
3. #ifdef
4. #ifndef
5. #endif
6. #elif

1. #if :-

#if directive is used to check whether the result of a given expression is zero or not. If the result is not zero, then the statements after #if are compiled, otherwise the statements after #else are compiled.

Syntax:

```
#if expression
{
   statement1;
   statement2;
}
#else
{
   statement3;
   statement4;
}
#endif
```

Example:

```
#include<stdio.h>
void main()
{
  printf("Welcome to");
  #if 5>6
   printf("SITAMS");
  #else
   printf("CSE");
  #endif;
  getch();
}
```

#elif:-

```
#include<stdio.h>
#define AGE 10
void main()
{
   clrscr();
   #if AGE<=10
   printf("Child");
   #elif AGE>10 && AGE<=30
   printf("Youth");
   #elif AGE>30 && AGE<=60
   printf("Middle aged");
   #else
   printf("Old");
  #endif
}
```

#ifdef directive:-

#ifdef directive compile a part of the program only if the macro is defined as a parameter irrespective of its value.

Syntax:
```
#ifdef identifier
    statements1;
#else
    statement2;
#endif
```

Explanation: #ifdef directive checks whether the identifier is defined or not. If defined then the statements after #ifdef are compiled and executed otherwise the statements after #else are compiled and executed.

Example:

```
#include<stdio.h>
#define SIZE
void main()
{
   printf("Welcome to");
   #ifdef SIZE
   printf("Java");
   printf("C++");
   #else
   printf("C");
   #endif
}
```

output: Welcome to Java C++

```
#include<stdio.h>
void main()
{
   printf("Welcome to");
   #ifdef SIZE
   printf("C++");
   printf("Java");
   #else
   printf("C");
   #endif
   printf("Have a cool day");
}
```

output: Welcome to C. Have a cool day.

#ifndef directive:

⇒ #ifndef works exactly opposite to #ifdef directive.

⇒ This directive tests whether the identifier is defined or not. The statements after #ifndef are compiled and executed if identifier is not defined. If the the identifier is defined the statements after #else are compiled.

Example:

```
#include<stdio.h>
void main()
{
  clrscr();
  #ifndef T
  printf("macro is not defined");
  #else
   printf("macro is defined");
  #endif
  getch();
}
```

Output: macro is not defined.

MULTI FILE PROGRAMS:-

⇒ In real-time a program may use more than one source file which may be compiled seperately and linked later to form an executable file.

⇒ Multiple source files can share a variable.

⇒ Variables that are shared by two or more files are global variables. Therefore we must declare them as global variable in one file and then explicitly define them with extern in other files.

Example:-

file1.C

```
int m; /* global variable */
main()
{
  int i;
  m=10;
  ----
}
function1()
{
  int j;
  m=m+20;
  -----
  ---
}
```

file2.C

```
function2()
{
  extern int m;
  int j;
  -------
  -------
}
function3()
{
  int count;
  -----
  -----
}
```

⇒ The function2() in file2 can reference the variable m which is declared as global in file1. function3() cann't access variable m. However if the statement extern int m; is placed before all functions, then both the functions can refer to m.

⇒ The extern keyword tells the compiler that the variable types and names have already been declared somewhere else and no need to create storage space for them.

STRINGS

→ INTRODUCTION
→ DECLARAING AND INITIALIZING STRING VARIABLES
→ READING STRING FROM TERMINAL
→ WRITING STRINGS TO THE SCREEN
→ ARITHMETIC OPERATIONS ON CHARACTERS
→ PUTTING STRINGS TOGETHER
→ COMPARISION OF TWO STRINGS
→ STRING HANDLING FUNCTIONS
→ TABLE OF STRINGS
→ STRING/DATA CONVERSION

1. INTRODUCTION:-

String:- A string is a sequence of characters that is treated as a single item.

⇒ Any group of characters defined between double quotation marks is a string constant.

Ex: "Welcome"
"Well Done!"
"Hai"

⇒ The common operations performed on character strings include:

1. Reading and writing strings
2. Combining strings together
3. Copying one string to another
4. Comparing strings for equality
5. Extracting a portion of a string.

2. DECLARING AND INITIALIZING STRING VARIABLES:-

DECLARATION:-

⇒ String is a collection of characters. So strings can be declared as character arrays.

Syntax: `Char string_name[size];`

Where Char → is a data type.

String_name must be a valid identifier.

Size determines the number of characters in the string.

Ex: Char city[10];
Char name[30];
Char str[15];

⇒ The size should be equal to maximum number of characters plus 0' Because, the compiler automatically supplies a null character ('\0') at the end of the string.

INITIALIZATION:-

⇒ Character arrays may be initialized when they are declared.

⇒ In C, Strings may be initialized in either of the following 2 forms:

1. Char city[9] = "NEW YORK";

(OR)

2. Char city[9] = {'N','E','W',' ','Y','O','R','K'};

⇒ We can also initialize a character array without specifying the num of characters.

Ex: Char string[ ] = {'G','O','O','D','\0'};

In this case size of the array will be determined automatically b on the number of characters initialized. In the above example size the string is 5.

⇒ We can also declare the size much langer than the string size.

Ex: Char str[10] = "GOOD";

In this case, the computer creates a character array of size 10, places "GOOD" in it and initializes all other elements to NULL.

G	O	O	D	\0	\0	\0	\0	\0	\0

⇒ The size should be not greater less than the number of characters.

Ex: Char S[3] = "GOOD"; → is wrong, it will show a compile time er

⇒ We can't seperate the initialization from declaration:

Ex: Char S[5];
S = "Good"; is not allowed.

3. READING STRINGS FROM TERMINAL:-

⇒ There are 3 ways to read strings from terminal:

1. Using scanf() function
2. Using gets() function
3. Using getchar() function

1. Using scanf() function:

⇒ The input function scanf can be used with %s format specification to read in a string of characters.

EXAMPLE:- Char city[10];
scanf("%s", city);

EXAMPLE:- Char S1[10], S2[10];
scanf("%s %s", S1, S2);

→ The scanf function automatically terminates the string that is read with null character.

NOTE: While reading strings using scanf() function, the & is not required before the variable name.

→ PROBLEM WITH SCANF():-

The problem with the scanf function is that it terminates its reading on the first white space it finds.

Ex: Char city[10];
scanf("%s", city);

For example, if the following line of text is entered

NEW YORK

then only the string "NEW" will be read into the string city, since the white sp after the string 'NEW' will terminate the reading of string

EXAMPLE PROGRAM:-

⇒ Write a program to read a series of words from terminal using scanf func

```
#include<stdio.h>
#include<conio.h>
void main()
{
  char word1[40], word2[20], word3[10];
  clrscr();
  printf("Enter 3 words:");
  scanf("%s", word1);
  scanf("%s %s", word2, word3);
  printf("\n word1 = %s \n word2 = %s \n word3 = %s", word1, word2, word3);
  getch();
}
```

OUTPUT: Enter 3 words: Oxford Road London

word1 = Oxford
word2 = Road
word3 = London

⇒ We can also specify the field width %ws in the scanf function for reading a spec number of characters from the input string.

EXAMPLE:- scanf("%ws", name);
↳ specifies number of characters to be read from the inpu str

Ex: char name[10];
scanf("%5s", name);

If the entered input string is "KRISHNAKUMAR" then the scanf function reads only 5 characters i.e. KRISH (∵ Because %5s → means it reads only 5 characte

READING A LINE OF TEXT USING SCANF FUNCTION:-

⇒ scanf with %s specification can't be used for reading more than one wor

Ex: char line[80];
scanf("%s", line); → Reads only a single word.

⇒ C supports a format specification known as edit set conversion code %[..], whic be used to read a line containing a variety of characters including whitesp

Ex: char line[80];
scanf("%[^\n]", line);
↳ Read a line of input from the keyboard.
printf("%s", line);

2. Using gets() function:-

⇒ gets() function can be used to read a line of text containing white spac

Syntax: `gets(string);`

gets() function reads characters into string from the keyboard until a new-line character (Enter button) is entered and adds a null character at end of the string.

Ex: char line[80];
gets(line);

EXAMPLE PROGRAM:-

⇒ Write a program to read a line of text using gets() function.

```
#include<stdio.h>
Void main()
{
    char name[30];
    clrscr();
    printf("Enter a string:");
    gets(name);
    printf("The string is: %s",name);
    getch();
}
```

INPUT:
Enter a string: C programming
The string is: C programming an

3. Using getchar() function:-

⇒ getchar() function reads a single character from the keyboard.

Ex: char ch;
ch=getchar();

⇒ We can use this function repeatedly to read a collection of characters from the terminal and store them into a character array. Thus a line of text can be read and stored in an array.

⇒ The reading is terminated when '\n' is entered and the null character is inserted at the end of the string.

```
Ex: char s[100],ch;
    int i=0;
    while((ch=getchar())!='\n')
    {
        s[i]=ch;
        i=i+1;
    }
    s[i]='\0';
```

EXAMPLE PROGRAM:-

⇒Write a program to read a line of text using getchar() function and display the

```
#include<stdio.h>
#include<conio.h>
Void main()
{
   Char Str[30],ch;
   int i=0;
   clrscr();
   printf("Enter a line of text:");
   while((ch=getchar())!='\n')
   {
      Str[i]=ch;
      i=i+1;
   }
   Str[i]='\0';
   printf("The entered text is: %s",Str);
   getch();
}
```

COPYING ONE STRING INTO ANOTHER STRING:-

⇒We cann't copy one string to another string directly.

Ex: Char String1[10], String2[10];

String1="ABC";
String2=String1 } are invalid.

⇒If we want to copy the string characters in string2 into String1, then we ca copy character-by-character basis.

EXAMPLE PROGRAM:

⇒Write a program to copy one String into another String.

```
#include<stdio.h>
Void main()
{
   Char String1[30],String2[30];
   int i;
   clrscr();
   printf("Enter a string:");
   scanf("%s",String1);
   for(i=0; String1[i]!='\0'; i++)
   {
      String2[i]=String1[i]; /*copying character by character */
   }
   String2[i]='\0';
   printf("After copying String2 is: %s",String2);
}
```

INPUT:-
Enter a string: Good

OUTPUT:-
After copying String2 is: Good.

WRITING STRINGS TO THE SCREEN:-

↳ Printing (or) Displaying strings

⇒ There are 3 ways to display strings on the screen.

1. Using printf() function
2. Using puts() function
3. Using putchar() function.

1. Using printf Function:-

⇒ The printf function with %s format can be used to print strings to the screen

Syntax: printf("%s", string_name);

The above function display the entire contents of string.

Ex: char city[10] = "NEW YORK";
printf("%s", city); → Displays "NEW YORK

EXAMPLE PROGRAM:-

⇒ Write a program to read a string using scanf function and display the same using printf function.

```
#include<stdio.h>
Void main()
{
    char str[30];
    clrscr();
    printf("Enter a string:");
    scanf("%s", str);
    printf("The entered string is: %s", str);
    getch();
}
```

INPUT:-
Enter a string: India
OUTPUT:-
The entered string is: India

⇒ We can also specify the precision with which array is displayed.

Syntax: printf("%w.ns", string);
↳ Number of characters to be displayed.
↳ Field width.

Ex: char name[15] = "United kingdom";

printf("%10.4s", name); → Display first 4 characters in the field width of 10 columns.
U n i t

printf("%-10.4s", name); → String will be printed left-justified
U n i t

printf("%15.0s", name); → Nothing will be printed
printf("%.3s", name); → Displays Uni
printf("%s", name); → Displays United kingdom

⇒ `printf("%*.*s", w, n, string);` → Display the first n characters of the string in the field width of w.

EXAMPLE PROGRAM:-

⇒ Write a program to print the following input

```
C
CP
CPR
CPRO
CPROG
CPROGR
CPROGRA
CPROGRAM
CPROGRA
CPROGR
CPROG
CPRO
CPR
CP
C
```

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char string[10] = "CPROGRAM";
    int i, n=0;
    clrscr();
    for(i=0; string[i]!='\0'; i++)
    {
        n=i+1;
        printf("%.*s", n, string);
    }
    for(i=7; i>=0; i--)
    {
        n=i-1;
        printf("%.*s", n, string);
    }
    getch();
}
```

2. Using puts() function:-

⇒ puts function can be used to print strings to the screen.

Syntax: `puts(string_name);`

The puts function displays the contents of the string and then moves the cursor to the beginning of the next line on the screen.

Example:- `Char city[10]="PARIS";`

`puts(city);` → Display PARIS on the screen.

EXAMPLE PROGRAM:-

⇒ Write a program to read a line of text using gets() function and displa the same using puts() function.

```
#include<stdio.h>
#include<conio.h>
Void main()
{
  Char book[50];
  Clrscr();
  printf("Enter a book name:");
  gets(book); /*Reading using gets */
  puts(book); /*Displaying using puts*/
  getch();
}
```

INPUT:
Enter a book: C and Data Structure
OUTPUT: C and Data Structures

3. Using putchar() function:-

⇒ putchar function displays one character at a time on the screen.

Ex: `Char c='A';`
`putchar(c);` → Display A

⇒ We can use this function repeatedly to display a line of text stored in a character array using a loop.

Syntax:
```
Char city[10]="NEW YORK";
int i;
for(i=0;city[i]!='\0';i++)
{
  putchar(city[i]);
}
```

(OR)
```
Char city[10]="NEW YORK";
int i=0;
while(city[i]!='\0')
{ putchar(city[i]);
  i++;
}
```

↳ Displays one character at a time. i.e. it display the string character by characte

EXAMPE PROGRAM:

⇒ Write a program to display a line of text using getchar function and display th same using putchar function.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   char str[50],ch;
   int i=0;
   clrscr();
   printf("Enter a string:");
   while((ch=getchar())!='\n')
   {
       str[i]=ch;
       i=i+1;
   }
   str[i]='\0';
  printf("The entered string is:");
   i=0;
   while(str[i]!='\0')
   {
      putchar(str[i]);
      i=i+1;
   }
   getch();
}
```

ARITHMETIC OPERATIONS ON CHARACTERS:-

⇒ Whenever a character variable is used in an expression, it is automc converted into an integer value (i.e. the equivalent ASCII value) by the system.

EXAMPE: int x;

x='a'; → convents 'a' into 97 i.e. ASCII value of a is 97

printf("%d", x); → Display 97 on the screen.

⇒ It is also possible to perform arithmetic operations on character constants and variables.

Ex: x = 'z'-1 = 122-1 = 121

y = 'u'+10 = 117+10 = 127

z = 'a'*2 = 97*2 = 194

a = 'y'+'z' = 121+122 = 243

ALPHABETS	ASCII VALUES
a-z	97-122
A-Z	65-90
0-9	48-57

⇒ Write a program to print the alphabet set a to z and A to Z in character and decimal form. i.e.

A-65 a-97
B-66 b-98
C-67 C-99
⋮ ⋮
Z-90 z=122

```
#include<stdio.h>
#include<conio.h>
void main()
{
    Char c;
    for(c=65; c<=90; c++)
    printf("\t %c - %d", c, c);
    printf("\n");
    for(c=97; c<=122; c++)
    printf("\t %c - %d", c, c);
    getch();
}
```

⇒ We may also use character constants in relational expressions.

Ex: Ch >= 'A' && Ch <= 'Z'

EXAMPLE PROGRAM:-

⇒ Write a program to check whether the given character is an uppercase letter or not.

```
#include<stdio.h>
void main()
{
    char ch;
    clrscr();
    printf("Enter a character:");
    scanf("%c", &ch);
    if( ch>='A' && ch<='Z')
     printf("The given character is an uppercase letter");
    else
     printf("The given character is not an uppercase letter");
    getch();
}
```

⇒ We can convert a character digit to its equivalent integer valu by using the following formula:

$$x = \text{character digit} - '0'$$

Ex: To convert '7' to its equivalent integer value

$x = '7' - '0'$

= ASCII value of '7' − ASCII value of '0'

= 55 − 48

= 7.

⇒ We can convert a string of digits into their integer value using function.

Syntax: atoi(string);

Ex:
```
char number[5] = "1988";
int year;
year = atoi(number);
year = 1988
```
→ converts "1988" (string) into 1988 →

6. PUTTING STRINGS TOGETHER:-

⇒ It is not possible to join (or) combine (or) add two strings directly by the simple arithmetic addition.

i.e. string3 = string1 + string2 ; → is not valid.

⇒ To combine string1 and string2, the characters from string1 and string2 should be copied into string3 one after the other.

⇒ The process of combining two strings together is called concatenation.

EXAMPLE PROGRAM:-

⇒ Consider the name of a person are stored in 3 arrays, namely first_name, second_name and last_name. Write a program to concatenate the 3 parts into one string called name.

```
#include<stdio.h>
void main()
{
  char first_name[10]="Viswanath";
  char second_name[10]="Pratap";
  char last_name[10]="Singh"; char name[30];
  int i,j,k;
  clrscr();
  for(i=0; first_name[i]!='\0'; i++)
  {
     name[i]=first_name[i]; → /* copying first_name into name */
  }
  name[i]=' ';
  for(j=0; second_name[j]!='\0'; j++)
  {
     name[i+j+1]=second_name[j]; → /*copying second_name into name*/
  }
  name[i+j+1]=' ';
  for(k=0; last_name[k]!='\0'; k++)
  {
     name[i+j+k+2]=last_name[k]; → /*copying last_name into name*/
  }
  name[i+j+k+2]='\0';
  printf("Full name is: %s", name); → /*Displaying full name i.e. name*/
}
```

7. COMPARISION OF TWO STRINGS:-

⇒ C doesn't support comparision of two strings directly.

ie. if (string1 == string2) is not permitted.

⇒ If you want to compare any two strings, then compare the two strings character by character.

⇒ The comparision is done until there is a mismatch on one of the strin terminates into a NULL character(\0).

EXAMPLE PROGRAM:-

⇒ Write a program to compare two strings. If both strings are equal, display th message "strings are equal". Otherwise display "strings are not equal.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   char str1[10], str2[10];
   int i=0;
   clrscr();
   printf("Enter string1:");
   gets(str1);
   printf("Enter string2:");
   gets(str2);
   while(str1[i]==str2[i] && str1[i]!='\0' && str2[i]!='\0')
   {
      i=i+1;
   }
   if(str1[i]=='\0' && str2[i]=='\0')
   printf("Strings are equal");
   else
   printf("Strings are not equal");
   getch();
}
```

INPUT:

Enter string1 : Hello

Enter string2 : Hello

OUTPUT:

Strings are equal.

8. # STRING-HANDLING FUNCTIONS

⇨C-library supports a large number of string-handling functions that can be used to perform the following operations on strings:

- → Combining strings together
- → Copying one string to another
- → Comparing strings for equality
- → Finding length of a string
- → Reversing a string

1. Strcat() Function:-

⇨The strcat function joins two strings together.

Syntax: `Strcat(string1, string2);`

→ string1 and string2 are character arrays

→ Strcat function appends (i.e. adds) string2 to the end of string1.

→ The size of string1 must be large to hold the final string.

Ex:
```
Char Str1[10] = "good";
Char Str2[10] = "news";
Strcat(Str1, Str2);
```

The above function adds Str2 to the end of Str1.

∴ The content of Str1 is: goodnews

EXAMPLE PROGRAM:-

```
#include<stdio.h>
void main()
{
   Char Str1[10] = "very";
   Char Str2[10] = "good";
   clrscr();
   Strcat(Str1, Str2);
   printf("After concatenation string1 is: %s", Str1);
   getch();
}
```

OUTPUT:

After concatenation string1 is: Verygg

Ex:
```
Char Str1[20] = "good";
Strcat(Str1, "Morning");
```
→ Str1: goodmorning

⇒ Strcat function can be nested.

Ex: Strcat(Strcat(String1, String2), String3);

The above function concatenates all the 3 strings together and the resultant string is stored in string1.

EXAMPLE PROGRAM:-

```
#include<Stdio.h>
Void main()
{
  Char S1[30]="Very", S2[10]="Good", S3[10]="News";
  ClrScr();
  Strcat(Strcat(S1,S2),S3);
  printf("S1 is: %S", S1);
  getch();
}
```

OUTPUT:-
S1 is: VeryGoodNews

2. Strncat() Function:-

⇒ The strncat function concatenate the left-most n characters of String2 to the end of String1.

Syntax: `Strncat(String1, String2, n);`

Ex: Char S1[10]="Bala", S2[10]="Gurusamy";
Strncat(S1, S2, 4);
The above function adds the first 4 characters of S2 to the end of S
∴ S1 is: BalaGuru

EXAMPLE PROGRAM:-

```
#include<Stdio.h>
Void main()
{
  Char S1[30]="Programming", S2[10]="In C and Java";
  ClrScr();
  Strncat(S1,S2,2);
  printf("After Concatenation S1 is: %S", S1);
  getch();
}
```

OUTPUT:- After Concatenation S1 is: ProgrammingIn

3. Strcmp() Function:-

⇒ The strcmp() function compares two strings.

If strings are equal it returns 0.

If strings are not equal, it returns the numeric difference between the first non-matching characters in the strings.

Syntax: `Strcmp(String1,String2);`

Here String1 and String2 may be string variables or string constants.

Ex: Char S1[10]="Hello", S2[10]="Hello";

```
Strcmp(S1,S2);
Strcmp(S1,"Hello");
Strcmp("Their","There");
```

EXAMPLE PROGRAM:-

⇒ Write a program to check whether the given strings are equal or not.

```
#include<stdio.h>
void main()
{
  char Str1[10], Str2[10];
  int n;
  printf("Enter String1:");
  gets(Str1);
  printf("Enter String2:");
  gets(Str2);
  n=Strcmp(Str1,Str2);
  printf(".");          ↳ Comparing
  if(n==0)
   printf("Both strings are equal");
  else
   printf("Strings are not equal");
}
```

INPUT:-

Enter String1 : Good morning

Enter String2 : Good morning

OUTPUT:-

Both Strings are equal.

4. Strncmp() Function:-

⇒ The strncmp() function compares the left-most n characters of S1 to S1 and returns : 1. 0 if they are equal

2. Negative number if String1 is less than String2

3. Positive number if String1 is greater than String2

Syntax: `Strncmp(String1,String2,n);`

EX: strncmp(S1,S2,4); → Compares first 4 characters of S1 to S2.

PROGRAM:-
```
void main()
{
  char S1[15]="good morning", S2[15]="good Evening";
  int n;
  clrscr();
  n=strncmp(S1,S2,4);
  if(n==0)
    printf("First 4 characters are equal");
  else
    printf("First 4 characters are not equal");
  getch();
}
```

OUTPUT:
First 4 characters are equ

5. strcpy() Function:-

⇒ The strcpy function copies the contents of one string to another.

Syntax: `strcpy(string1,string2);`

→ The strcpy function copies contents of string2 to string1.

→ String2 may be a character array or a string constant

EX1: `char city1[10], city2[10]="Delhi";`

`strcpy(city1,city2);` → Copies contents of city2 to city1.
∴ city1 contains Delhi.

EX2: `strcpy(city1,"Hyderabad");` → Copies "Hyderabad" to city1.

PROGRAM:-
```
#include<stdio.h>
void main()
{
  char city1[10], city2[10]="DELHI";
  clrscr();
  strcpy(city1,city2);
  printf("After copying city1 is: %s",city1);
  getch();
}
```

OUTPUT:-
After copying city1 is: DELHI

8. Strrev() Function:-

⇒ The strrev function reverse a given string.

Syntax: `strrev(string);`

Ex: `char *str="good";`

`strrev(str);` → Reverse the string str. i.e. doog

PROGRAM:-

⇒ Write a program to reverse a given string.

```
#include<stdio.h>
void main()
{
    char *city="Delhi";
    clrscr();
    printf("Before reverse city is: %s", city);
    strrev(city);
    printf("After reversing city is: %s", city);
    getch();
}
```

OUTPUT:-

Before reverse city is: De
After reversing city is: ih

9. Strstr() Function:-

⇒ The strstr() function can be used to locate a sub-string in a main st

Syntax: ~~strstr(main,sub);~~

`strstr(main-string, sub-string);`

The above function searches the main string to check whether the sub is present or not. If the sub string is present in the main string, it re the first occurance of the sub-string. Otherwise it returns a NULL poin

EXAMPLE PROGRAM:-

⇒ Write a program to see whether the sub string is present in the main str If it present display the position of sub string.

```
#include<stdio.h>
void main()
{
    char m[20], s[10], *x;
    clrscr();
    printf("Enter the main string:");
    scanf("%s", m);
    printf("Enter the sub string:");
    scanf("%s", n);
    x=strstr(m,s);
    if(x==NULL)
    printf("Sub string is not present
                                the mai
    else
    printf("Position of substring in
            main string is: %d", x
    getch();
}
```

6. Strncpy() Function:-

⇨ The strncpy function copies only the left-most n characters of the source string to the target string variable.

Syntax: `strncpy(String1, String2, n);`

The above function copies the first n characters of string2 into str

EX: `strncpy(S1, S2, 5);` → Copies first 5 characters of S2 into S1.

`S1[6] = '\0';` → We have to explicitly include the NULL character.

PROGRAM:-

```
#include<stdio.h>
void main()
{
    char S1[10], S2[20] = "Programming";
    clrscr();
    strncpy(S1, S2, 7);        → copies first 7 characters of S2 to S1
                                  S1 contains Program
S1[7] = '\0';
    printf("String1 is: %s", S1);
    getch();
}
```

OUTPUT:
String1 is: Program

7. Strlen() Function:-

⇨ The strlen function returns the number of characters in a string.

Syntax: `n = strlen(string);`

Where n is an integer variable, which receives the length of the string

EX: `char str[5] = "good";`
`int n;`
`n = strlen(str);` → n = 4, Since str contains 4 characters.

PROGRAM:-

```
#include<stdio.h>
void main()
{
    char str[20] = "Programming";
    int n;
    clrscr();
    n = strlen(str);
    printf("The length of the string is: %d", n);
    getch();
}
```

OUTPUT:-
The length of the string is: 11

INPUT:-
OUTPUT:-

Enter the main string : Welcome
Enter the sub string : come

OUTPUT:

The position of sub string in main string is : 4

String Palindrome:-

⇒Write a program to check whether the given string is palindrome or not.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   char str[30];
   int i,j,n,c=0;
   clrscr();
   printf("Enter a string:");
   scanf("%s",str);
   n=strlen(str);
   for(i=0,j=n-1;str[i]!='\0';i++,j--)
   {
       if(str[i]==str[j])
       continue;
       else
       {
          c=1;
          break;
       }
   }
   if(c==0)
   printf("%s is a palindrome",str);
   else
   printf("%s is not a palindrome",str);
   getch();
}
```

TABLE OF STRINGS:-

⇨ Suppose consider, we want to store a list of the names of students i a class, list of the names of employees in a company, list of places

⇒ A list of names can be treated as a table of strings.

⇒ A two-dimensional character array can be used to store a table of stri

Ex: student[30][15] may be used to store a list of 30 names whos length is not more than 15 characters.

⇒ Consider the following table of strings:

B	a	n	g	l	o	r	e
B	o	m	b	a	y		
m	a	d	r	a	s		
I	n	d	i	a			
C	h	e	n	n	a	i	

⇒ The above table can be stored as follows:

```
Char city[5][15] = { "Banglore", "Bombay", "madras", "India", "Chen
```

→ city[i] denotes the name of ith city.

For example city[0] denotes "Banglore", city[1] denotes "Bombay" so on.

EXAMPLE PROGRAM:-

⇒ Write a program to read and display a table of strings.-

```
#include<stdio.h>
void main()
{
  Char student[30][15];
  int i;
  clrscr();
  printf("Enter 30 students names:");
  for(i=0; i<30; i++)
  {
    scanf("%s", student[i]);
  }
  printf("Students names are:");
  for(i=0; i<30; i++)
  {
    printf("\n %s", student[i]);
  }
  getch();
}
```

Strstr():

```
#include<stdio.h>
#include<conio.h>
void main()
{
char main[10]="usha rani",sub[5]="rani",*x;
clrscr();
x=strstr(main,sub);
if(x==NULL)
printf("The sub string is not present in main
string");
else
printf("\nPosition of substring in main string
is:%d",(x-main));
getch();
}
```

Strrev():

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char *s1="sitams";
clrscr();
printf("Before reverse s1 is:%s",s1);
strrev(s1);
printf("\nAfter reverse s1 is:%s",s1);
getch();
}
```

STRING PALINDROME (USING STRING H/ FUNC

```
void main()
{
  Char Str1[30], Str2[30];
  int n;
  Clrscr();
  Printf("Enter a string:");
  Scanf("%s", Str1);
  StrCpy(Str2, Str1);
  n = StrCmp (Str1, Strrev(Str2));
  if(n==0)
  Printf("%s is palindrome", Str1
  else
  Printf("%s is not a palindrome
  getch();
}
```

Without using string handling func.

```
main()
{
  Char Str[30];
  int i,j,n, c=0;
  Clrscr();
  Printf("Enter a string:");
  Scanf("%s", Str);
  n = Strlen(Str);
  for(i=0,j=n-1; Str[i]!='\0'; i++,
  {
      if(Str[i]==Str[j])
      continue;
      else
      {
        c=1;
        break;
      }
  }
  if(c==0)
  Printf("%s is palindrome", St
  else
  Printf("%s is not a palindrom
  getch()
}
```

⇒ WAP using pointer to print array of 5 strings.

```
#include<stdio.h>
#include<conio.h>
void main()
{
  char *name[5]={"India","Hydenabad","Chennai","Chittoor","Banglore"};
   int i;
   clrscr();
   for(i=0;i<5;i++)
   printf("\n%s",name[i]);
   getch();
}
```

⇒ Write an example program to return multiple values in a function?

```
main()
{
   void change(int *x,int *y,int *z);
    int a=10, b=20, c=30;
    clrscr();
    change(&a,&b,&c);
    printf("a=%d \n b=%d \n c=%d", a, b, c);
    getch();
}
void change(int *x,int *y, int *z)
{
    x=x+1;
    y=y+1;
    z=z+1;
}
```

String comparison without using string handling functions:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str[10],str1[10];
int i=0;
clrscr();
printf("Enter string1: ");
gets(str);
printf("Enter string2:");
gets(str1);
while(str[i]==str1[i]) /*&& str[i]!='\0' &&
str1[i]!='\0')*/
i++;
if(str[i]=='\0' && str1[i]=='\0')
printf("Strings are equal");
else
printf("Strings are not equal");
getch();
}
```

String comparison using string handling functions:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[10],str2[10];
int x;
clrscr();
printf("Enter sring1:");
gets(str1);
printf("Enter string2:");
gets(str2);
x=strcmp(str1,str2);
if(x==0)
printf("Both strings are equal");
else
printf("Strings are not equal");
getch();
}
```

String Concatenation using string handling functions:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[10]="very";
char str2[10]="good";
clrscr();
strcat(str1,str2);
printf("\nAfter concatenation String1
is:%s",str1);
getch();
}
```

Example2:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[10]="very";
clrscr();
strcat(str1,"bad");
printf("\nAfter concatenation String1
is:%s",str1);
getch();
}
```

Example 3:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char
str1[20]="Very",str2[5]="good",str3[5]="news";
clrscr();
strcat(strcat(str1,str2),str3);
printf("After concatenation string1 is:%s",str1);
getch();
}
```

Example 4:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[10],str2[5];
clrscr();
printf("Enter string1:");
gets(str1);
printf("Enter string2:");
gets(str2);
strcat(str1,str2);
puts(str1);
puts(str2);
getch();
}
```

String Copy using string handling functions:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[10]="good",str2[10]="Morning";
clrscr();
strcpy(str1,str2);
printf("\n After String copy string1 is:%s",str1);
getch();
}
```

String Copy without using string handling functions:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[20]="good morning",str2[20];
int i;
clrscr();
for(i=0;str1[i]!='\0';i++)
str2[i]=str1[i];
str2[i]='\0';
printf("After copying string2 is:%s",str2);
getch();
}
```

String length using string handling functions:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str[10]="very good";
int l;
clrscr();
l=strlen(str);
printf("\n The given string is:%s",str);
printf("\nLength of the string is:%d",l);
getch();
}
```

String length without using string handling functions:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str[10]="Very good";
int i=0;
clrscr();
while(str[i]!='\0')
i=i+1;
printf("\n The length of the string is:%d",i);
getch();
}
```

Strncpy():

```
#include<stdio.h>
#include<conio.h>
void main()
{
char s1[10]="very good",s2[10];
clrscr();
strncpy(s2,s1,4);
s2[4]='\0';
printf("String2 is:%s",s2);
getch();
}
```

Strncmp()

```
#include<stdio.h>
#include<conio.h>
void main()
{
char s1[15]="sitams cse",s2[15]="sitams ece";
int x;
clrscr();
x=strncmp(s1,s2,5);
if(x==0)
printf("First 5 characters are equal");
else
printf("Not equal");
getch();
}
```

Strncat():

```
#include<stdio.h>
#include<conio.h>
void main()
{
char s1[20]="Bala",s2[10]="Gurusamy";
clrscr();
strncat(s1,s2,4);
printf("After concatenation s1 is:%s",s1);
getch();
}
```