# Computer Concept & Programming

I semester, BScCSIT

Ambition Guru College

**Compiled by :**
**Ankit Bhattarai**

# Unit V

(8 hrs.)

## Function

- Introduction, User Defined Functions, Return Statement, Function Call,
- Types of Functions based on their Return Type and Function Call,
- Inline Functions, Recursions,
- Arrays and Functions, Preprocessor Directives and Standard Library Functions

## Concept of Function

- Function is a self -contained block of code or statement that performs a particular task.
- Every program must contain one function named main() from where the program execution begin.
- Complex problem can be solved by breaking them into set of sub-problems, called modules or functions. This technique is called divide and conquer.
- Each module can be implemented independently and later can be combined into single unit.

## Concept of Function

**Advantages of function**

- It makes program significantly easier to understand and maintain by breaking up them into easily manageable chunks.
- A single function can be used multiple times which avoids rewriting of same code again and again.
- Different programmers working on one large project divide the workload by writing different functions.
- Program that use functions are easier to design debug and maintain.
- C functions can be used to build a customized library of frequently used routines

## Concept of Function

**Disadvantage**

- It increase the execution time because when function is called the control have to jump to function definition to perform the particular task ,and after completion of task , control again come back to the function call.

## Example of Function:

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

int sum(int,int); //function declaration

int main()
{
int a, b,result;
printf("Enter the first number\n");
scanf("%d",&a);
printf("Enter the second number\n");
scanf("%d",&b);
/* Calling the function here, the function return type is integer
so we need an integer variable to hold the returned value of this
function.*/

result =sum(a,b);
/*actual arguments a and b are passed in calling function*/

printf ("Sum of two numbers= %d, result);
return 0;
}
```

## Example of Function:

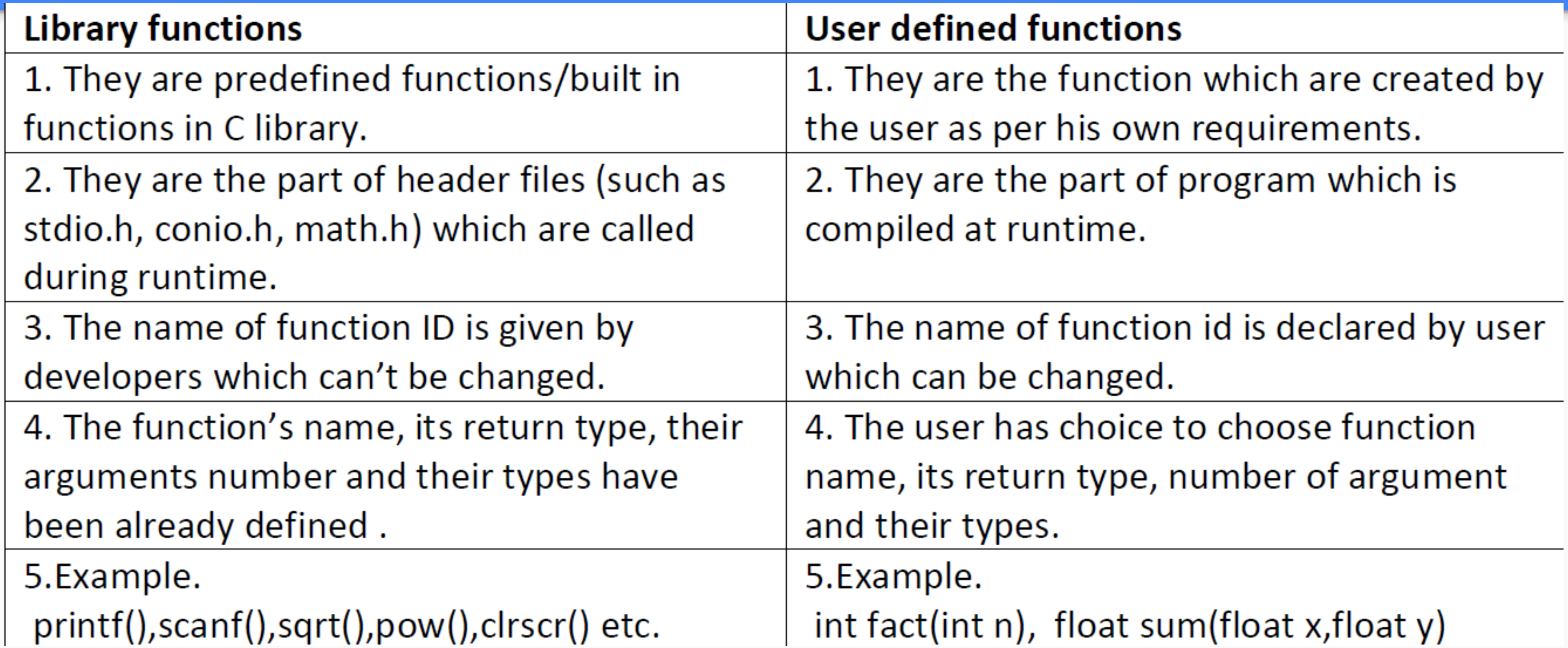
```
/*formal arguments x and y are passed in called
function*/

int sum(int x, int y) //function definition
{
   int r;
   r=x+y;

/* Function return type is integer so we are returning an
integer value, the sum of the passed numbers.*/
   return r;
}
```

## Types of Function

## Types of Function



| Library functions | User defined functions |
|---|---|
| 1. They are predefined functions/built in functions in C library. | 1. They are the function which are created by the user as per his own requirements. |
| 2. They are the part of header files (such as stdio.h, conio.h, math.h) which are called during runtime. | 2. They are the part of program which is compiled at runtime. |
| 3. The name of function ID is given by developers which can’t be changed. | 3. The name of function id is declared by user which can be changed. |
| 4. The function’s name, its return type, their arguments number and their types have been already defined . | 4. The user has choice to choose function name, its return type, number of argument and their types. |
| 5.Example. printf(),scanf(),sqrt(),pow(),clrscr() etc. | 5.Example. int fact(int n), float sum(float x,float y) |

## Types of Function

- Program to illustrate use of user defined function and library functions.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int square(int);
int main()
{
int num=5,result;
result=pow(num,2);
printf("Square of number using library function = %d",
result);
result= square(num);
printf("\nSquare of number using user defined function=%d"
,result);
return 0;
}
int square(int x)
{
           return (x * x);
}
```

## Components associated with function

## Components associated with function : Function Definition

### Function definition

The collection of program statements that describes the specific task to be done by the function is called function definition. It consists of:

**Function header:** which defines function's name, its return type and its argument list

**Function body :** which is a block of code enclosed in parenthesis.

**Syntax:**

```
return_type function_name(data_type variable1,data_type variable2,…..,data_type variable n)
{
statements ;
…………………..
……………………
return value;
}
```

## Components associated with function : Function Definition

### Function definition

**Note**: If a function doesn't return any value , then its return type is void

**Syntax**:

```
void function_name (data_type varibale1,data_type variable2,…..,data_type
variable n)
{
statements ;
………………..
……………………
}
```

## Components associated with function : Function Prototype/ Function Declaration

### Function Prototype / Function Declaration

The function declaration or prototype is a model or blueprint of a function. If a function is used before it is defined in a program, then function declaration or prototype is needed to provide the following information to the compiler.

- The name of function
- The type of the value returned by the function
- The number and type of arguments that must be supplied while calling the function

The syntax for function declaration is :

**return_type function_name(type1, type2 ,type3 ……… type n);**

Here, return_type specifies the data type of the value returned by the function. A function can return value of any data type.If there is no return value, the keyword void is used.

## Components associated with function : Function Prototype/ Function Declaration

### Function Prototype / Function Declaration

The function declaration and declarator or header in function definition must use the same function name, number of arguments, argument types and return types. Some examples of

function prototype are:

int add(int a,int b);

int factorial(int n);

## Components associated with function : Accessing/calling a function

- When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.
- A function can be called or accessed by specifying its name, followed by a list of arguments enclosed in parentheses and separated by commas. **For example, the function sum() with two arguments is called by sum(a,b) to add two numbers.**
- During function call, function name must match with function prototype name, the type of return type, the number of arguments and order of arguments.

## Components associated with function : Return Statement

The job of return statement is to hand over some value given by function body to the point from where the call was made. The function defined with its return type must return a value of that type.

For example: If a function has return type int, it returns an integer value from the called function to the calling function.

The main functions return statement are:

- It immediately transfers the control back to the calling program after execution of return statement (i.e. no statement within the function body is executed after the return statement.)
- It returns value to the calling function.

## Components associated with function : Function Parameter (Arguments)

Parameters provide the data communication between calling function and called function. They can be classified into actual parameters and formal parameters.

**Actual parameters:**

- These are the parameters that transfer data from the calling function to the called function.
- This type of parameter is used in function call.

**Formal parameters:**

- These parameters used to receive the data sent by the calling function. These are also used to send data from calling function to the called function.
- These parameters used in function definition.

In given example the parameter a and b are actual parameter which are used in calling function and x and y are formal parameters which are used in called function.

## Types of Function

- Program to illustrate use of user defined function and library functions.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int square(int);
int main()
{
int num=5,result;
result=pow(num,2);
printf("Square of number using library function = %d",
result);
result= square(num);
printf("\nSquare of number using user defined function=%d"
,result);
return 0;
}
int square(int x)
{
          return (x * x);
}
```

## Classwork

WAP to calculate the area and perimeter of rectangle using function

```
#include<stdio.h>
#include<conio.h>
void area(int,int);
void perimeter(int,int);
int main()
{
int l,b;
printf("Enter the length and breadth of rectangle\n");
scanf("%d%d",&l,&b);
area(l,b);
perimeter(l,b);
getch();
return 0;
}
void area(int x,int y)
{
          int area;
          area=x*y;
          printf("area=%d",area);
}
void perimeter(int x,int y)
{
          int peri;
          peri=2*(x+y);
          printf("Perimeter=%d",peri) ;
}
```

## Classwork

WAP to calculate the area of two circles having different radius using the same function area.

```
#include<stdio.h>
#include<conio.h>
float area(float r);
int main()
{
float r1,r2,a1,a2;
printf("Enter the radious of first circle");
scanf("%f",&r1);
a1=area(r1);
printf("Enter the radious of second circle");
scanf("%f",&r2) ;
a2=area(r2);
printf("Area of first circle=%f",a1);
printf("\nArea of second circle=%f",a2);
getch();
}
float area(float r)
{
             float a;
             a=3.14*r*r;
             return a;
}
```

## Classwork

**WAP to check given number is palindrome or not using function**

```
#include<stdio.h>
#include<conio.h>
void palindrome(int n);
int main()
{
int num;
printf("Enter the number you want to check\n");
scanf("%d",&num);
palindrome(num);
getch();
return 0;
}
```

## Classwork

**WAP to check given number is palindrome or not using function**

```
void palindrome(int n)
{
int rem,rev=0,a;
a=n;
while(n!=0)
{
            rem=n%10;
            rev=rev*10+rem;
            n=n/10;
}
if(a==rev)
            {
            printf("Number is palindrome");
            }
else
            {
            printf("Number is not palindrome");
            }
}
```

## Classwork

**WAP to check given number is palindrome or not using function**

```
void palindrome(int n)
{
int rem,rev=0,a;
a=n;
while(n!=0)
{
        rem=n%10;
        rev=rev*10+rem;
        n=n/10;
}
if(a==rev)
        {
        printf("Number is palindrome");
        }
else
        {
        printf("Number is not palindrome");
        }
}
```

## Global Vs Local Variable

**Global variables:**

Global variables are accessible to all functions defined in the program. Global variables are declared outside any function, generally at the top of program after preprocessor directives. It is useful to declare variable global if it is to be used by many functions in the program. The default initial values for this variable is zero. The lifetime is as long as the program execution does not come to an end.

**Local variables:**

The variables that are defined within the body of a function or block and local to that function or block only are known as local variables. The name and value of local variables are valid within the function in which it is declared. They are unknown to other. The local variables are created when the function is called and destroyed automatically when function is exited function.

## Global Vs Local Variable

```
#include<stdio.h>
#include<conio.h>
void fun();
int x=5; //global declaration
int main()
{
int a=10; //local declaration
printf("value of x=%d,a=%d",x,a);
fun();
getch();
return 0;
}

void fun()
{
int b=20; //local declaration
printf("\nvalue of x=%d,b=%d",x,b);
}
```

**Output**

```
Value of x=5, a=10
Value of x=5, b=20
```

## Different ways of using function

# Different ways of using function

Function can be categorized in four types, on the basis of arguments and return value.

1.Function with no arguments and no return values

2.Function with arguments but no return values

3.Function with no arguments but with return values

4.Function with arguments and return values

**1. Function with no arguments and no return values**

When a function has no arguments, it does not receive any data from the calling function. Similarly, when it doesn't return a value, the calling function does not receive any data from the called function. Thus in such type of functions, there is no transfer between the calling function and the called function. This type of function is defined as.

```
void function_name( )
{
          /*body of the function*/
}
```

## Different ways of using function

**1. Function with no arguments and no return values**

- The keyword **void** means the function does not return any value.
- There is no arguments within parenthesis which implies function has no argument and it does receive any data from the called function.

***Program to illustrate the "function with no arguments and no return values"***

```
#include<stdio.h>
#include<conio.h>
void sum();
int main()
{
sum();
getch();
return 0;
}

void sum()
{
          int x,y,r;
          printf("Enter the two numbers\n");
          scanf("%d%d",&x,&y);
          r=x+y;
          printf("The sum of numbers=%d",r);
}
```

# Different ways of using function

2. **Function with arguments but no return value**

- These type of function has arguments and receives the data from the calling function.
- The function completes the task and does not return any values to the calling function. Such type of functions are defined as

```
void function_name(argument list)
{
/*body of function*/
}
```

***Program to illustrate the "function with arguments but no return values"***

```
#include<stdio.h>
#include<conio.h>
void sum(int,int);
int main()
{
int a,b;
printf("Enter the two numbers\n");
scanf("%d%d",&a,&b);
sum(a,b);
getch();
return 0;
}

void sum(int x,int y)
{
          int r;
          r=x+y;
          printf("The sum of numbers=%d",r);
}
```

## Different ways of using function

3. **Function with no arguments but with return values**

- These type of function does not receive any arguments but the function return values to the calling function. Such type of functions are defined as

```
return_type function_name( )
{
/*body of function*/
}
```

***Program to illustrate the "function with no arguments and with return values"***

```
#include<stdio.h>
#include<conio.h>
int sum();
int main()
{
int result;
result=sum();
printf("The sum of numbers=%d",result);
getch();
return 0;
}

int sum()
{
          int x,y,r;
          printf("Enter the two numbers\n");
          scanf("%d%d",&x,&y);
          r=x+y;
          return r;
}
```

# Different ways of using function

**4. Function with arguments and return value**

- The function of this category has arguments and receives the data from the calling function.
- After completing its task ,it returns the result to the calling function through return statement.
- Thus there is data transfer between called function and calling function using return values and arguments. These type of functions are defined as

```
return_type function_name(argument

list)

{

/*body of the function*/

}
```

***Program to illustrate the "function with arguments and return values"***

```
#include<stdio.h>
#include<conio.h>
int sum(int,int);
int main()
{
int a,b,result;
printf("Enter the two numbers\n");
scanf("%d%d",&a,&b);
result=sum(a,b);
printf("The sum of numbers=%d",result);
getch();
return 0;
}

int sum(int x,int y)
{
          int r;
          r=x+y;
          return r;
}
```

## Call by Value

## Call by Value

The arguments in function can be passed in two ways, namely **call by value** and **call by reference.**

**Function call by value (or pass by value)**

- In this method the value of each actual arguments in the calling function is copied into corresponding formal arguments of the called function.
- With this method the changes made to the formal arguments in the called function have no effect on the values the actual argument in the calling function.

## Call by Value

**Explanation:**
*In this example, the values of a and b are passed in function swap() by value. The value of a is copied into formal argument x and value of b is copied into formal argument y. The copy of value of a and b to x and y means the original values of a and b remain same and these values are copied into the variables x and y also. Thus when x and y are changed within the function , the values of the variables x and y are changed but the original value of a and b remains same.*

```
#include<stdio.h>
#include<conio.h>
void swap(int x,int y);
int main()
{
int a,b;
a=10;
b=20;
printf("value before swapping a=%d and b=%d",a,b);
swap(a,b);
printf("\nvalue after swapping a=%d and b=%d",a,b);
getch();
return 0;
}

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

**Output:**

```
Value before swapping a=10 and b=20
Value after swapping a=10 and b=20
```

## Call by Reference

## Call by Reference

**Function call by reference (Or pass by Reference)**

- In this method the address of actual arguments in the calling function are copied into formal arguments of the called function .This means that using these addresses the actual arguments can be accessed.
- In call by reference since the address of the value is passed any changes made to the value reflects in the calling function.

## Call by Reference

**Explanation:** *In this example ,the address of variables (i.e. &a and &b) are passed in function. These addresses are received by corresponding formal arguments in function definition. To receive addresses, the formal arguments must be of type pointers which stores the address of variables. The address of a and b are copied to the pointer variable x and y in function definition.*

*When the values in addresses pointed by these pointer variables are altered, the values of original variables are also changed as the content of their addresses have been changed .Thus, while arguments are passed in function by reference ,the original values are changed if they are changed within function.*

```
#include<stdio.h>
#include<conio.h>
void swap(int *x,int *y);
int main()
{
int a,b;
a=10;
b=20;
printf("The value before swapping are a=%d and
b=%d",a,b);
swap(&a,&b);
printf("\nThe values after swapping are a=%d
and b=%d",a,b);
getch();
return 0;
}

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

**Output:**

```
Value before swapping a=10 and b=20
Value after swapping a=20 and b=10
```

## Question:

**WAP to check the given number is prime or not using user defined function.**

```
#include<stdio.h>
#include<conio.h>
void prime(int n);
int main()
{
  int num;
  printf("Enter the number you want to check\n");
  scanf("%d",&num);
  prime(num);
  getch();
  return 0;
}

void prime(int n)
{
  int i;
  for(i=2;i<n;i++)
  {
      if(n%i==0)
      {
          printf("Number is not prime");
          break;
      }
}

if(i==n)
  {
   printf("Number is prime");
  }
}
```

## Question 2:

**WAP to generate the Fibonacci series up to nth term when initial value is given by user.**

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

void fibo(int n,int x,int y);

int main()
{
 int a,b,num;
 printf("Enter the number of terms you want to
generate\n");
 scanf("%d",&num);
 printf("Enter the two inital values\n");
 scanf("%d%d",&a,&b);
 fibo(num,a,b);
 getch();
 return 0;
}
```

## Question 2:

**WAP to generate the Fibonacci series up to nth term when initial value is given by user.**

```
void fibo(int n,int x,int y)
{
   int i,c;
   printf("%d\t%d",x,y);

   for(i=1;i<=n-2;i++)
   {
      c=x+y;
      printf("\t%d",c);
      x=y;
      y=c;
}
```

## Recursion

## Recursion

A function that calls itself is known as a recursive function. Recursion is a process by which function calls itself repeatedly until some specified condition will be satisfied.

To solve a problem using recursive method, two conditions must be satisfied .They are

- Problem could be written or defined in terms of previous result
- Problem statement must include stopping condition

**Advantages**

- Avoid unnecessary calling of functions.
- Through Recursion one can solve problems in easy way while its iterative solution is very big and complex
- Reduces time complexity.
- Using recursion, the length of the program can be reduced.
- Extremely useful when applying the same solution repeatedly.

## Recursion

**Disadvantages**

- Recursive solution is always logical and it is very difficult to trace. (debug and understand).
- For each step we make a recursive call to a function. For which it occupies significant amount of stack memory with each step.
- It is usually slower due to the overhead of maintaining the stack.
- May cause stack-overflow if the recursion goes too deep to solve the problem
- If the programmer forgets to specify the exit condition in the recursive function, the program will execute out of memory.

## Recursion

- WAP to find the factorial of a given number using recursive function

```
#include<stdio.h>
#include<conio.h>
long int fact(int n);
int main()
{
int num;
long int f;
printf("Enter the number\n");
scanf("%d",&num);
f=fact(num);
printf("The factorial of a given number is %ld",f);
getch();
return 0;
}

long int fact(int n)
{
          if(n==1||n==0)
          return 1;
          else
          return n*fact(n-1);
}
```

## Recursion

- Write a program to find the sum of first n natural number using recursive function

```
#include<stdio.h>
#include<conio.h>
int snatural(int n);
int main()
{
int num,s;
printf("Enter the value of n \n");
scanf("%d",&num);
s=snatural(num);
printf("The sum of n natrual number is %d",s);
getch();
return 0;
}

int snatural(int n)
{
        if(n==1)
        return 1;
        else
        return n + snatural(n-1);
}
```

## Recursion Vs Iteration

| Recursion | Iteration |
|---|---|
| The statement in a body of function calls the function itself. | Allows the set of instructions to be repeatedly executed. |
| Recursion is always applied to functions. | Iteration is applied to iteration statements or "loops". |
| In recursive function, only termination condition (base case) is specified | Iteration includes initialization, condition, execution of statement within loop and updating (increments/decrements) the control Variable |
| Infinite recursion can crash the system. | Infinite loop uses CPU cyclesrepeatedly. |
| Slow in execution | Fast in execution |

## Inline Function in C

## Inline Function

- An **inline function** in C is a function that the compiler expands at the point where it is called instead of performing a regular function call.
- This eliminates the overhead of function calls, making execution faster for small, frequently used functions.

**Syntax:**

```
inline return_type function_name(parameters)
{
        // Function body
}
```

## Inline Function

**Example:**

```
#include <stdio.h>
inline int square(int x) {
    return x * x;
}

int main() {
    int num = 5;
    printf("Square of %d is %d\n", num, square(num));
    return 0;
}
```

## Inline Function

**When to Use Inline Functions**

- **Use them** for small, frequently called functions where the performance gain outweighs the code size increase (e.g., math utilities like min() or max()).
- **Avoid them** for large functions, recursive functions (which can't be inlined effectively), or when code size is a bigger concern than execution speed (e.g., in embedded systems with limited memory).

## Inline Function

**Pros:**

- **Improved Performance**: Inline functions can reduce function call overhead (e.g., pushing/popping stack), as the compiler may replace the function call with the function body directly.
- **Faster Execution**: By avoiding the overhead of a function call, inline functions can lead to quicker execution, especially for small, frequently called functions.
- **Optimization Opportunity**: The compiler can optimize the inlined code more effectively within the calling context, potentially leading to better machine code generation.
- **No Stack Overhead**: Since the function body is inserted at the call site, there's no need to manage stack frames for small functions.
- **Useful for Small Functions**: Ideal for simple, short functions where the overhead of a call outweighs the execution time of the function itself.

## Inline Function

**Cons:**

- **Increased Binary Size**: Inlining duplicates the function's code at every call site, which can significantly increase the size of the compiled binary.
- **Limited Compiler Control**: The inline keyword is only a suggestion in C; the compiler may ignore it if it deems inlining unsuitable, reducing portability of expected behavior.
- **Cache Inefficiency**: Larger binaries from excessive inlining can lead to poor instruction cache performance, slowing down execution in some cases.
- **Not Suitable for Large Functions**: Inlining large functions can bloat the code unnecessarily, negating performance benefits and making the program harder to manage.
- **Increased Compile Time**: More code to process (due to duplication) can slow down the compilation process.
- **Debugging Difficulty**: Inlined functions may not appear in stack traces or debuggers as separate entities, complicating debugging efforts.

## Preprocessor Directives

## Preprocessor Directives

- The preprocessor is a program that processes the source code before it is passed to the compiler. It operates under the control of preprocessor directives, which are special instructions used to modify the code before compilation.
- Preprocessor directives always begin with a hash (#) sign and are typically placed at the beginning of a program, before the main function. Unlike regular statements, preprocessor directives do not require a semicolon (;) at the end.

**Example:**

```
#include
#define
#ifdef
#undef
#if
#else
#endif
```

## Preprocessor Directives : Macros

- Macros are a piece of code in a program which is given some name. Whenever this name is encountered by the compiler the compiler replaces the name with the actual piece of code.

**Syntax:**

**#define macro_name macro_expansion**

- macro_name is any valid C identifier, and it is generally in capital letters to distinguish it from other variables.
- macro_expansion can be any text

The ‘#define’ directive is used to define a macro.

Eg. `#define PI 3.1415`

## Preprocessor Directives : Macros

***Program to find the square of a number by using both macro and function***

```
#include<stdio.h>
#include<conio.h>
#define NUM 5
#define square(num) (num*num)
void func();

int main()
{
  printf("Square of a number using macro=%d",square(NUM));
  func();
  return 0;
}

void func()
{
   int result,a=5;
   result=a*a;
   printf("\nSquare of number using function=%d",result);
}
```

**Output:**

```
Square of number using macro=25
Square of number using function=25
```

## Standard Library Functions

## Standard Library Functions

- Header files are the special files which stores the predefined library functions.

| Header file | function |
|---|---|
| #include<stdio.h> | Used to perform input and output operations in C like scanf() and printf(). |
| #include<string.h> | Perform string manipulation operations like strlen and strcpy |
| #include<conio.h> | Perform console input and console output operations like clrscr() to clear the screen and getch() to get the character from the keyboard. |
| #include<stdlib.h> | Perform standard utility functions like dynamic memory allocation, using functions such as malloc() and calloc(). |
| #include<math.h> | Perform mathematical operations like sqrt() and pow(). To obtain the square root and the power of a number respectively. |
| #include<signal.h> | Perform signal handling functions like signal() and raise(). To install signal handler and to raise the signal in the program respectively |
| #include<errno.h> | Used to perform error handling operations like errno(). |

## Standard Library Functions

The C Standard Library provides built-in functions categorized under various headers.

**<stdio.h> (Standard Input/Output):**

- printf(): Prints formatted output to the standard output (console).
- scanf(): Reads formatted input from the standard input (keyboard).
- fprintf(), fscanf(): file input and output.
- fopen(), fclose(): file operations.
- getchar(), putchar(): character input and output.

**<stdlib.h> (Standard Library):**

- malloc(), calloc(), realloc(), free(): Dynamic memory allocation.
- atoi(), atof(), atol(): String to integer/float/long conversions.
- rand(), srand(): Random number generation.
- exit(): Terminates the program.
- abs(): absolute value.
- system(): executes a system command.

## Standard Library Functions

**<string.h> (String Handling):**

- strcpy(), strncpy(): String copying.
- strcat(), strncat(): String concatenation.
- strcmp(), strncmp(): String comparison.
- strlen(): String length.
- strchr(), strstr(): String searching.

**<math.h> (Mathematics):**

- sin(), cos(), tan(): Trigonometric functions.
- sqrt(): Square root.
- pow(): Power.
- exp(): exponential function.
- log(), log10(): logarithmic functions.

## Standard Library Functions : Example

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

#define PI 3.14159
int main() {
    char str1[20] = "Hello";
    char str2[20] = " World";

    strcat(str1, str2);
    printf("Concatenated String: %s\n", str1);

    double radius = 5.0;
    double area = PI * pow(radius, 2);
    printf("Area of Circle: %.2f\n", area);

    return 0;
}
```

## Arrays and Functions

## Passing Array to Function

## Passing Array to Function

### Passing one dimensional Array to function

In C, arrays are automatically passed by reference to a function. The name of the array represents the address of its first element. By passing the array name, we are in fact, passing the address of the array to the called function. The array in the called function now refers to same array stored in the memory. Therefore any changes in the array in the called function will be reflected in the original array.

- The corresponding formal argument in function definition is written in the same manner, though it must be declared as an array.
- When declaring a one dimensional array as a formal argument, the array name is written with a pair of empty square brackets. The size of the array is not specified within the formal argument declaration.
- To pass an array as an argument to function, the array name must be specified, without brackets or subscripts, and the size of an array as arguments.

## Passing Array to Function

**Syntax for function declaration that receives array as argument**

```
return_type function_name(data_type array_name[ ] );
```

**Syntax to pass array to the function**.

```
function_name(arrayname);
```

**Note:** We cannot pass a whole array by value as we did in the case of ordinary variables

## Passing Array to Function

**Program to illustrate passing array to a function one element at a time.**

```
#include<stdio.h>
#include<conio.h>
void display(int x);
int main()
{
    int i;
    int a[5]={5,10,15,20,25};
    for(i=0;i<5;i++)
    {
        display(a[i]);
    }
    getch();
    return 0;
}
```

```
void display(int x)
{
    printf("%d\t",x);
}
```

## Passing Array to Function

**Write a program to pass one dimensional array to a function and display that array in that called function.**

```
#include<stdio.h>
#include<conio.h>
void display(int x[],int n);
int main()
{
   int i;
   int a[5]={5,10,15,20,25};
   display(a,5);
   getch();
   return 0;
}

void display(int x[],int n)
{
    int i;
    for(i=0;i<n;i++)
    {
      printf("%d\t",x[i]);
    }
}
```

## Passing Array to Function

**Write a program to read n number in an array and sort them in ascending order using function.**

```
#include<stdio.h>
#include<conio.h>
void sort(int x[],int n);
void disp(int d[],int m);
int main()
{
 int a[100],n,i;
 printf("Enter how many elements\n");
 scanf("%d",&n);
 printf("Enter the array elements\n");
 for(i=0;i<n;i++)
 {
   scanf("%d",&a[i]);
 }
 printf("\nArray elements before sorting are\n");
 disp(a,n);
 sort(a,n);
 printf("\nArray elements after sorting are\n");
 disp(a,n);
 getch();
 return 0;
}
```

## Passing Array to Function

**Write a program to read n number in an array and sort them in ascending order using function.**

```
void sort(int x[],int n)
{
    int i,j,temp;
    for(i=0;i<n-1;i++)
   {
       for(j=0;j<n-1-i;j++)
            {
                if(x[j]>x[j+1])
                    {
                     temp=x[j];
                      x[j]=x[j+1];
                      x[j+1]=temp;
                     }
            }
     }
}
```

```
void disp(int d[],int m)
{
           int i;
           for(i=0;i<m;i++)
           {
           printf("%d\n",d[i]) ;
           }
}
```

## Some Questions to Work On

- WAP to read n numbers in an array find the largest number among them.
- WAP to input n number in an array and sort them in Ascending order
- Write a program to add two 3X3 matrix. Display the sum stored in third matrix
- WAP to check whether the given number is present in a array or not.
- Write a program to enter values in 3*3 order matrix and compute the sum of odd elements.

## Passing 2D Array to Function

### Passing two dimensional Array to function

It is similar as in 1 dimensional array, when two dimensional array are passed as a parameter,

the base address of the actual array is sent to function.

- Any change made to element inside function will carry over to the original location of array that is passed to function.
- The size of all dimensions except the first must included in the function prototype (declaration) and in function definition.

Syntax:

```
return_type function_name (array_name, size, size);
```

Eg. `void display(int[][10],int,int);`

is a valid declaration.

## Passing  2D Array to Function

**Write a program to show addition of two m*n order matrix using function.**

```
#include<stdio.h>
#include<conio.h>
void input(int a[][20],int x,int y);
void disp(int d[][20],int m,int n);
void addition(int a1[][20],int a2[][20],int p,int q);
int i,j;
int main()
{
            int matrix1[20][20],matrix2[20][20],r,c;
            printf("Enter the row and column size of matrix\n");
            scanf("%d%d",&r,&c);
            printf("Enter the first matrix");
            input(matrix1,r,c);
            printf("Enter the second matrix");
            input(matrix2,r,c);
            printf("The first matrix is\n");
            disp(matrix1,r,c);
            printf("The second matrix is\n");
            disp(matrix2,r,c);
            addition(matrix1,matrix2,r,c);
            getch();
            return 0;
}
```

## Passing 2D Array to Function

**Write a program to show addition of two m*n order matrix using function.**

```
void input(int a[][20],int x,int y)
{
            for(i=0;i<x;i++)
            {
                        for(j=0;j<y;j++)
                        {
                           scanf("%d",&a[i][j]);
                        }
            }
}

void disp(int d[][20],int m,int n)
{
            for(i=0;i<m;i++)
            {
                        for(j=0;j<n;j++)
                        {
                        printf("%d\t",d[i][j]);
                        }
            printf("\n");
}
}
```

## Passing  2D Array to Function

**Write a program to show addition of two m*n order matrix using function.**

```
void addition(int a1[][20],int a2[][20],int p,int q)
{
        int sum[20][20];
        for(i=0;i<p;i++)
        {
                for(j=0;j<q;j++)
                {
                sum[i][j]=a1[i][j]+a2[i][j] ;
                }
        }
        printf("Resultant matrix is\n");
        disp(sum,p,q);
}
```

## Question 1:

- WAP that counts the number of even elements in an array using function.

```
#include <stdio.h>
int cEven(int arr[], int size);

int main() {
    int size;
    int csize;
    printf("Enter size of array: ");
    scanf("%d", &size);

    int arr[size];

    printf("Enter elements:\n");
    for(int i = 0; i < size; i++)
        {
        scanf("%d", &arr[i]);
        }
    csize = cEven(arr, size);

    printf("Even count = %d\n", csize);
    return 0;
}

int cEven(int arr[], int size) {
    int count = 0;
    for(int i = 0; i < size; i++)
        if(arr[i] % 2 == 0)
            {
            count++;
            }
    return count;
}
```

## Question 1:

- Multiply each element by 2.

```
#include <stdio.h>
void display(int arr[], int size);

int main() {
    int arr[100];
    int size,i;
    printf("Enter size of array: ");
    scanf("%d", &size);

    printf("Enter elements:\n");
    for(i = 0; i < size; i++)
        {
        scanf("%d", &arr[i]);
        }

    display(arr, size);
    return 0;
}

void display(int arr[], int size){
            int i;
            for(i=0; i <size; i++)
            {
                        arr[i] =  arr[i] * 2;
            }
printf("After multiplication \n");
   for(i =0; i < size; i++)
            {
            printf("%d", arr[i]);
              }
}
```

## Passing Strings to Function

## Passing Strings to Function

In C, we can pass a string to a function using a character array or a pointer to a character. Since strings in C are stored as arrays of characters and are null-terminated (\0), passing a string essentially means passing a pointer to its first character.

Two approaches:

1. Passing String as a Character Array
2. Passing string as Pointer

**Passing String as a Character Array**

- A string in C is represented as a character array (char array[]).
- When we pass an array to a function, we are actually passing the base address of the array.

## Passing Strings to Function

```
#include<stdio.h>
void printString(char str[]);

int main() {
    char myString[] = "Hello, C!";

    // Passing the string to function
    printString(myString);

    return 0;
}

// Function to print a string
void printString(char str[]) {
    printf("The string is: %s\n", str);
}
```

THANK YOU

## Any Queries ?