UNIT – IV

Functions: Introduction to Functions, Function Declaration and Definition, Function call Return Types and Arguments, arrays as parameters, Scope and Lifetime of Variables, storage class, recursion, functions & pointers, function, and arrays.

Functions

Function: A function is a self-contained block of program statements that performs a particular task.

- We have written 'C' programs using three functions namely, main, printf() and scanf().
- Every program must have a main function to indicate where the execution of the program begins and ends.
- If the program is large and if we write all the statements of that large program in main itself, the program may become too complex and large.
- As a result, the task of debugging, testing and even understanding will become very difficult.
- So, the best way to develop a large program is to construct it from smaller pieces (or) modules.
- A module can be a single function (or) a group of related functions carrying out a specific task.
- Usually, it is easier to break down a difficult task into a series of smaller tasks and then to solve those subtasks individually and later combined into a single unit.
- These subtasks are called **User-defined functions.**

Advantages:

Modularity: Divides a program into smaller, manageable units.
Code Reusability: Write once, use multiple times.
Easier Debugging: Troubleshoot functions independently.
Better Readability: Makes code easier to understand.
Reduced Code Redundancy: Avoid repetition.

General form of a function:

```
return_type function-name(argument list)          ----->function header.
{
        local variable declaration;
        statement 1;
        statement 2;                              ------->function body
        ..........
        ..........
        return(expression);
}
```

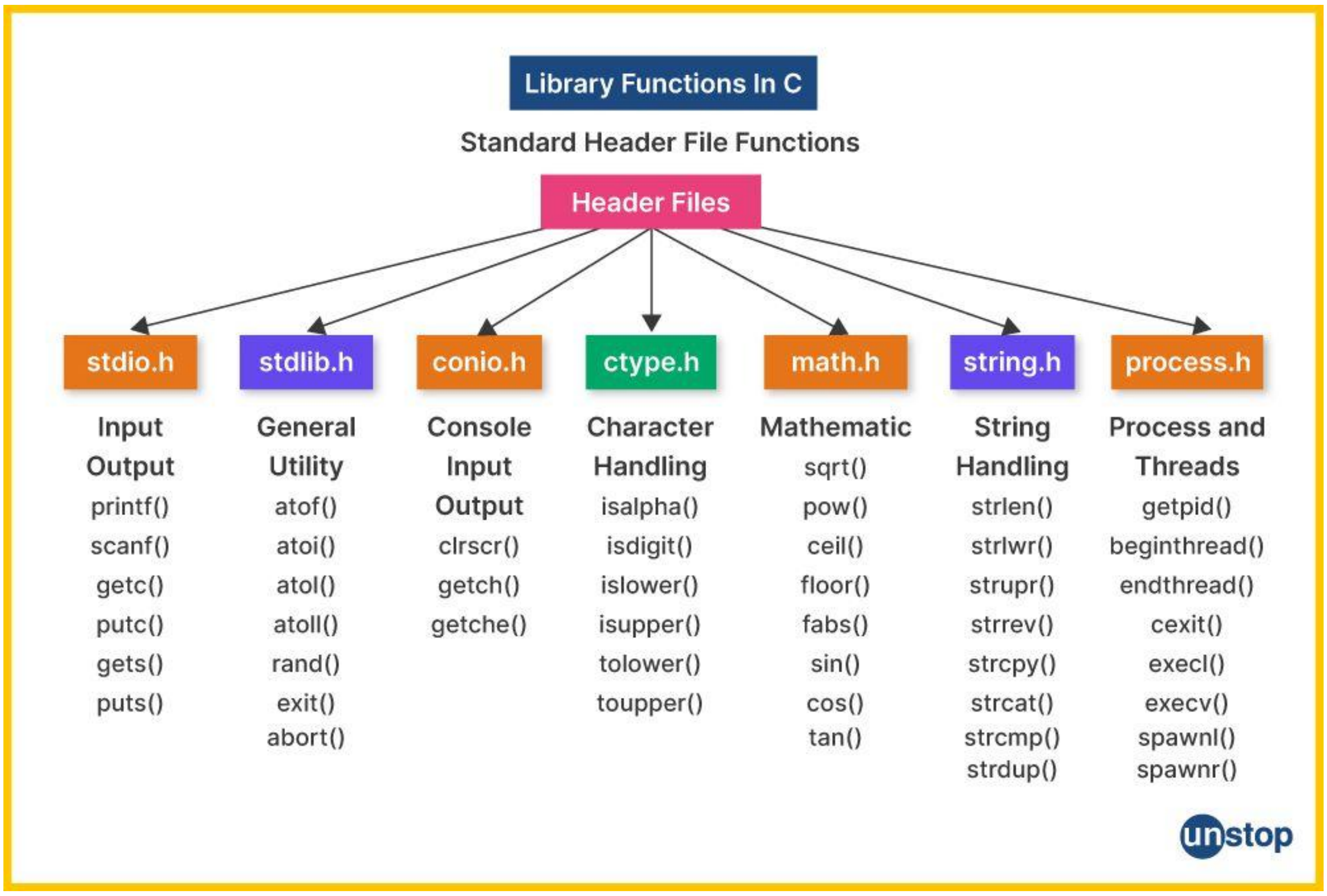
- Here, type specifies the type of data that the function returns.
- The type and argument list are optional.
- An unspecified type is always assumed by the compiler to int. int is the default type when no type specifier is present.
- A function returns a type other than int, it must be explicitly declared.
- The function-name is any valid identifier.

Characteristics of functions:

- Any function can return only one value.
- Parameter argument list is optional.
- Return statement indicates exit from the function and return to the point from where the function was invoked.
- A function can call any number of times.
- A call to the function must end with a semicolon.
- Any ‘C’ function cannot be defined in other function.
- When a function is not returning any value, void type can be used as return type.
- ‘C’ allows recursion i.e..., a function can call itself.

Types of Functions:

1) Library Functions: Predefined functions provided by C libraries.
Examples: printf(), scanf(), strlen(), sqrt(), etc.



2) User-Defined Functions: Functions created by the programmer to perform specific tasks.
Ex:

```
int add(int a, int b) {
        return a + b;
}
```

User-defined functions in C include several components that collectively define their structure and behavior. Below are the key components of a user-defined function in C:
1) Function Declaration or Prototype
2) Function Definition
3) Function Call

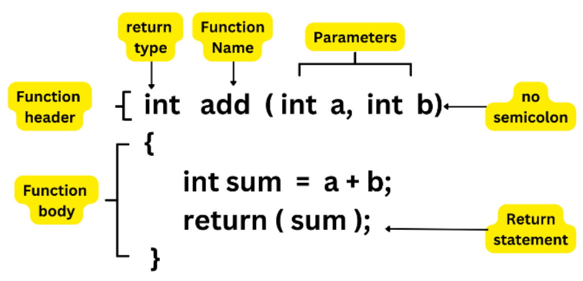
Function declaration: A function declaration tells the compiler about a function's name, return type, and parameters.
Syntax: return_type function_name(parameters types..);
Ex: int add(int,int);

Function definition: A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and some of the programs can define additional functions.

Syntax: return_type function_name(arguments list)

```
{
    statements.......
}
```

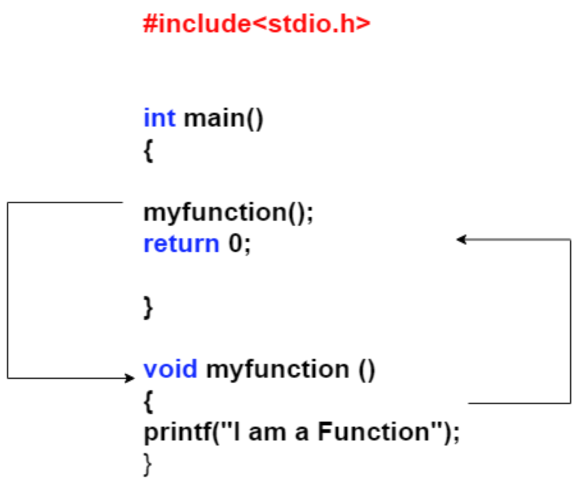
Function Definition



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

Function call: A function can be called by specifying function name, followed by a list of arguments enclosed in parenthesis and separated by commas. If the function call does not require any arguments, an empty pair of parenthesis must follow the function name.
Syntax: fun-name(arg1,arg2,....);

WORKING OF A FUNCTION

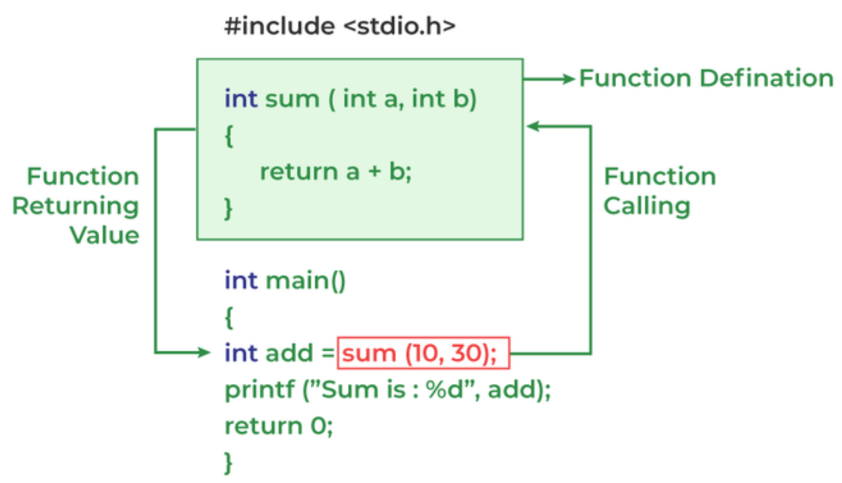


```
#include<stdio.h>

int main()
{
myfunction();
return 0;

}

void myfunction ()
{
printf("I am a Function");
}
```

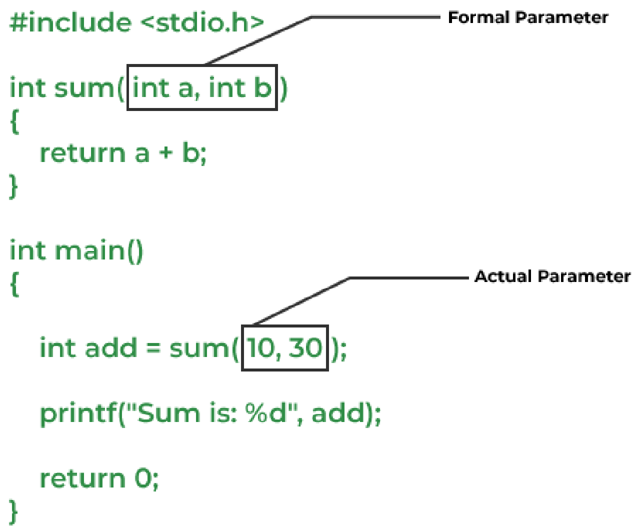
Parameters (Optional)

- Variables listed in the function declaration and definition to receive input values when the function is called.
- **Formal Parameters**: Parameters defined in the function header.
  - Example: `int add(int a, int b)` → `a` and `b` are formal parameters.
- **Actual Parameters**: Arguments passed during the function call.
  - Example: `add(5, 3)` → `5` and `3` are actual parameters.

Working of Function in C



```
#include <stdio.h>
int sum ( int a, int b)
{
    return a + b;
}
int main()
{
int add = sum (10, 30);
printf ("Sum is : %d", add);
return 0;
}
```



```
#include <stdio.h>

int sum(int a, int b)
{
   return a + b;
}

int main()
{

   int add = sum(10, 30);

   printf("Sum is: %d", add);

   return 0;
}
```

Working with functions in C Programming:

1) Program to demonstrate working with functions in C.

```
#include<stdio.h>
void wish();                                  //function prototype
int main()
{
        printf("Before Wish Call....\n");
        wish();                               //function call
        printf("After Wish Call....\n");
}
void wish()                                   //function definition
{
        printf("Good Afternoon\n");
}
```

Output:

```
Before Wish Call....
Good Afternoon
After Wish Call....
```

2) Program to practice the function calls in C.

```
#include<stdio.h>
void wish();
int main()
{
        printf("Before Wish Call....\n");
        wish();
        printf("After Wish Call....\n");
}
void wish()
{
        printf("Good Afternoon\n");
        displayYourName();
}
void displayYourName()
{
        printf("Rajesh B\n");
}
```

Output:

```
Before Wish Call....
Good Afternoon
Rajesh B
After Wish Call....
```

3) Program to perform arithmetic operations in C using functions.

```
#include<stdio.h>
int add(int,int);
int subtract(int,int);
int main()
{
        int x,y;
        scanf("%d%d",&x,&y);
        int sum=add(x,y);
        int diff=subtract(x,y);
        printf("Addition of %d and %d is: %d\n",x,y,sum);
        printf("Difference of %d and %d is: %d\n",x,y,diff);
        return 0;
}
int add(int a,int b)
{
        return a+b;
}
int subtract(int a,int b)
{
        return a-b;
}
```

4) Program to find the factorial of a give number.

```
#include<stdio.h>
int factorial(int);
int main()
{
   int n,fact;
   scanf("%d",&n);
   fact=factorial(n);
   printf("Factorial of %d is %d\n",n,fact);
   return 0;
}
int factorial(int n)
{
    //write your logic here.
          int f=1,i;
          for(i=n;i>=2;i--)
           {
              f=f*i;
           }
          return f;
}
```

5) Program to display the Fibonacci series up to given value n.

Fibonacci Sequence: Every element in the sequence is generated by the sum of its two previous elements.

Ex: 0 1 1 2 3 5 8 13 21 ....

Input:

```
30
```

Output:

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

```
#include<stdio.h>
void printFibonacci(int);
int main()
{
          int range;
          scanf("%d",&range);
          printFibonacci(range);
    return 0;
}

void printFibonacci(int n)
{
          int a=0,b=1,c;
          printf("%d %d ",a,b);
          c=a+b;
```

```
        while(c<=n)
        {
      printf("%d ",c);
                a=b;
                b=c;
                c=a+b;
        }
}
```

6) Program to find the sum of elements of an array using functions.

```
#include<stdio.h>
int arraySum(int[],int);
int main()
{
        int n,i;
        scanf("%d",&n);
        int arr[n];
        for(i=0;i<n;i++)
        {
                scanf("%d",&arr[i]);
        }
        int result=arraySum(arr,n);
        printf("Sum of array elements is: %d\n",result);
        return 0;
}
int arraySum(int X[],int size)
{
        int sum=0,i;
        for(i=0;i<size;i++)
   {
        sum=sum+X[i];
        }
        return sum;
}
```

Input:

```
3
11 22 33
```

Output:

7) Program to read your name and time as an input and display the wish message.

Sample Input:

```
Aditya Ram
14
```

Sample Output:

```
Hi Aditya Ram!
Good Afternoon.
```

```
#include<stdio.h>
int main()
{
        char name[30];
        int time;
        scanf("%[^\n]s",name);
        scanf("%d",&time);
        wish(name,time);
        return 0;
}
void wish(char word[],int time)
{
        printf("Hi %s\n",word);
        if(time>=0 && time<12)
          printf("Good Morning.\n");
        else if(time>=12 && time <16)
          printf("Good Afternoon\n");
        else
          printf("Good Evening\n");
}
```

Categories of function:

According to the arguments and return values, there are four types of user defined functions.

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.
4. Functions with no arguments and return values.

1) Functions with no arguments and no return values:

In this type of function, the called function doesn't take arguments from the calling function and it will not give any return value to the calling function.

2) Functions with arguments and no return values:

In this type of functions, the called function can take the arguments from the calling function but it will return any values to the calling function.

3) Functions with arguments and return values:

In this type of functions, the called function can take the arguments from the calling function and it will give some return values to the calling function.

4) Functions with no arguments and return values:

In this type of functions, the called function returns some value to the calling function. But it doesn't take any arguments from the calling function.

Examples:

```
//No arguments and
//No return value
#include<stdio.h>
void sum();
int main()
{
     sum();
     return 0;
}
void sum()
{
    int x,y;
    scanf("%d%d",&x,&y);
    printf("sum=%d",x+y);
}
```

```
//With arguments and
//No return value
#include<stdio.h>
void sum(int,int);
int main()
{
     int x,y;
     scanf("%d%d",&x,&y);
     sum(x,y);
     return 0;
}
void sum(int p,int q)
{
  printf("sum=%d",p+q);
}
```

```
//With arguments and
//With return value
#include<stdio.h>
int sum(int,int);
int main()
{
     int x,y,z;
     scanf("%d%d",&x,&y);
     z=sum(x,y);
     printf("sum=%d",z);
     return 0;
}
int sum(int p,int q)
{
  return (p+q);
}
```

```
//Without arguments and
//With return value
#include<stdio.h>
int sum();
int main()
{    int z;
     z=sum();
     printf("sum=%d",z);
     return 0;
}
int sum()
{
     int p,q;
     scanf("%d%d",&p,&q);
     return (p+q);
}
```

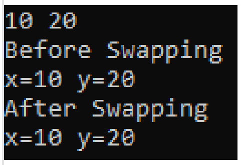
Parameter passing techniques: Arguments to a function are usually passed in two ways.
1. Call by value   2. Call by reference

Call by value: The call by value is a method of passing arguments to a function and the values of the actual parameters copy to the formal parameters of the function. In this case, changes made to the parameter inside the function have no effect on the actual arguments. By default, C programming uses call by value to pass arguments.

Call by address: The call by Address is a method of passing arguments to a function and copies the address of actual argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. In this case, changes made to the parameters inside the function has effected on the actual arguments.

```c
#include<stdio.h>
void swap(int,int);
int main()
{    int x,y;
     scanf("%d%d",&x,&y);
     printf("Before Swapping\n");
     printf("x=%d y=%d\n",x,y);
     swap(x,y);
     printf("After Swapping\n");
     printf("x=%d y=%d\n",x,y);
     return 0;
}
void swap(int p,int q)
{
     int r;
     r=p;
     p=q;
     q=r;
}
```

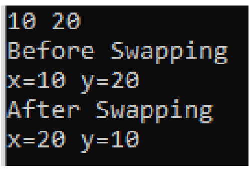
Output:



```
10 20
Before Swapping
x=10 y=20
After Swapping
x=10 y=20
```

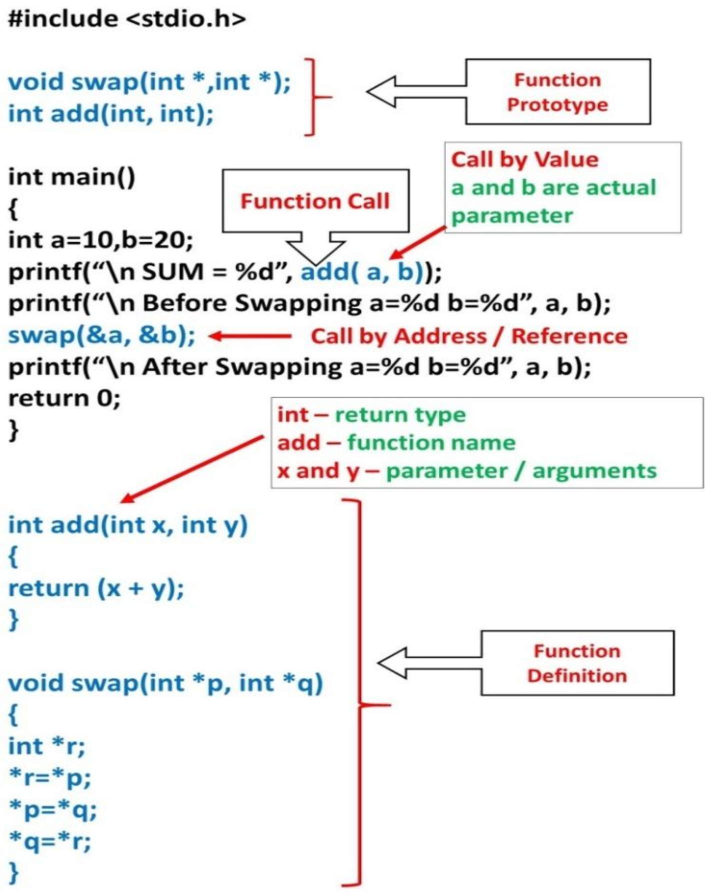
```c
#include<stdio.h>
void swap(int*,int*);
int main()
{    int x,y;
     scanf("%d%d",&x,&y);
     printf("Before Swapping\n");
     printf("x=%d y=%d\n",x,y);
     swap(&x,&y);
     printf("After Swapping\n");
     printf("x=%d y=%d\n",x,y);
     return 0;
}
void swap(int *p,int *q)
{
     int r;
     r=*p;
     *p=*q;
     *q=r;
}
```

Output:



```
10 20
Before Swapping
x=10 y=20
After Swapping
x=20 y=10
```

Call By Value	Call By Reference
While calling a function, we pass values of variables to it. Such functions are known as **"Call By Values"**.	While calling a function, instead of passing the values of variables, we pass address of variables(location of variables) to the function known as **"Call By References"**.
In this method, the value of each variable in calling function is copied into corresponding dummy variables of the called function.	In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function.
With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.	With this method, using addresses we would have an access to the actual variables and hence we would be able to manipulate them.
Thus actual values of a and b remain unchanged even after exchanging the values of x and y.	Thus actual values of a and b get changed after exchanging values of x and y.
In call by values we cannot alter the values of actual variables through function calls.	In call by reference we can alter the values of variables through function calls.
Values of variables are passes by Simple technique.	Pointer variables are necessary to define to store the address values of variables.



```
#include <stdio.h>

void swap(int *,int *);
int add(int, int);

int main()
{
int a=10,b=20;
printf("\n SUM = %d", add( a, b));
printf("\n Before Swapping a=%d b=%d", a, b);
swap(&a, &b);
printf("\n After Swapping a=%d b=%d", a, b);
return 0;
}

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

void swap(int *p, int *q)
{
int *r;
*r=*p;
*p=*q;
*q=*r;
}
```

Recursion: A function, which invokes itself repeatedly until some condition is satisfied, is called a recursive function.

- The normal function will be called by main function whenever the function name is used.
- On the other hand, the recursive function will be called by itself repeatedly, until some specified condition has been satisfied.
- This technique is used to solve problems whose solution is expressed in terms of successively applying the same set of steps.

Components of Recursion

1. **Base Case**: A condition that stops the recursion. Without it, the recursion would continue indefinitely (leading to a stack overflow error).
2. **Recursive Case**: The part where the function calls itself to break the problem into smaller sub-problems.

For example:

the factorial of a positive integer number 'n' is defined as **n!=n(n-1)(n-2).....1 with 1!=1, 0!=1.**
The above formula can also be written as **n!=n(n-1)!**

For example 5!=(5)(4!)
=5*4*3!
=5*4*3*2!
=5*4*3*2*1!
=5*4*3*2*1
= 120.

In 'C', a recursive function that calculates n! can be written as follows.

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

/*Program to find the factorial of a given number*/

Without Recursion

```
#include<stdio.h>
#include<conio.h>
int fact(int);
void main( )
{
        int n,f;
        clrscr( );
        printf("\n Enter n value");
        scanf("%d",&n);
        f=fact(n);
        printf("\n The factorial of %d is %d", n, f);
        getch( );
}
int fact(int x)
{
        int f;
        for(i=0; i<=x; i++)
                f=f * i;
        return f;
}
```

With Recursion

```
#include<stdio.h>
#include<conio.h>
int fact(int);
void main( )
{
        int n,f;
        clrscr( );
        printf("\n Enter n value");
        scanf("%d",&n);
        f=fact(n);
        printf("\n The factorial of %d is %d", n, f);
        getch( );
}
int fact(int x)
{
        if(x==0)
            return 1;
        else if(x==1)
            return 1;
        else
            return(x*fact(x-1));
}
```

/*C program for generating Fibonacci sequence with recursion*/

```
#include<stdio.h>
#include<conio.h>
int fib(int);
void main( )
{
        int n, f;
        clrscr( );
        printf("\n Enter n value");
        scanf("%d",&n);
        for(i=1; i<=n; i++)
        {
                f=fib(i);
                printf("%d \t",f);
         }
        getch( );
}
```

```
int fib(int x)
{
        if(x==1)
            return 0;
        else if(x==2)
            return 1;
        else
            return fib(x-1) + fib(x-2);
}
```

Storage classes:

Storage classes in C define the scope, lifetime, visibility, and default initial value of variables. The main storage classes in C are:

1) Automatic (auto)
2) Register (register)
3) Static (static)
4) External (extern)

1. Automatic Storage Class (auto): A variable defined within a function or block with auto specifier belongs to automatic storage class.

Scope: Local to the block where it is defined.
Lifetime: Until the block/function ends.
Default Value: Garbage value.
Keyword: auto (optional, as it is the default for local variables).
Example:

```
#include <stdio.h>
void autoExample()
{
   auto int x = 10;                  // 'auto' is optional
   printf("Automatic variable x = %d\n", x);
}
int main()
{
   autoExample();
   return 0;
}
```

Output:

```
Automatic variable x = 10
```

2. Register Storage Class (register): The register specifier declares a variable of register storage class. Variables belonging to register storage class are local to the block which they are defined in and get destroyed on exit from the block.

Scope: Local to the block where it is defined.
Lifetime: Until the block/function ends.
Default Value: Garbage value.
Keyword: register
Purpose: Suggests storing the variable in a CPU register for faster access.
Note: The compiler may ignore this request if registers are not available.
Example:

```
#include <stdio.h>
void registerExample()
{
   register int x = 5; /            / Suggest to store 'x' in a CPU register
   printf("Register variable x = %d\n", x);
}
int main()
{
   registerExample();
   return 0;
}
```

Output:

```
Register variable x = 5
```

3. Static Storage Class (static): The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

Scope: Local to the block where it is defined (for local variables) or global (for global variables).
Lifetime: Throughout the program's execution.
Default Value: 0 for numeric types.
Keyword: static.
Purpose: Retains its value between function calls.
Example:

```
#include <stdio.h>
void staticExample()
{
    static int x = 0;                    // Static variable retains value between calls
    x++;
    printf("Static variable x = %d\n", x);
}
int main()
{
    staticExample();
    staticExample();
    staticExample();
    return 0;
}
```

Output:

```
Static variable x = 1
Static variable x = 2
Static variable x = 3
```

4. External Storage Class (extern): The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. The variables declared as extern is to specify that the variable is defined elsewhere in the program. Default initial value of the extern integral variable is 0 otherwise null.

Scope: Global (accessible across multiple files).
Lifetime: Throughout the program's execution.
Default Value: 0 for numeric types.
Keyword: extern.
Purpose: Declare a global variable or function defined in another file.
Example (with two files):

File1.c:

```
#include <stdio.h>
int x = 10;                    // Global variable
void display() {
   printf("Value of x = %d\n", x);
}
```

File2.c:

```
#include <stdio.h>
#include "File1.c"
extern int x;                         // Declare global variable
void display();
int main()
{
   printf("Accessing x in another file: %d\n", x);
   display();
   return 0;
}
```

Output:

```
Accessing x in another file: 10
Value of x = 10
```

Practice Programs:

1) Program to find the transpose of a matrix using functions in C.

```
#include<stdio.h>
void readMatrix(int,int,int[][]);
void displayMatrix(int,int,int[][]);
void transposeMatrix(int,int,int[][],int[][]);
// Function to read a matrix
void readMatrix(int rows, int cols, int matrix[rows][cols]) {
   printf("Enter elements of the matrix (%d x %d):\n", rows, cols);
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         scanf("%d", &matrix[i][j]);
      }
   }
}
// Function to display a matrix
void displayMatrix(int rows, int cols, int matrix[rows][cols]) {
   printf("Matrix (%d x %d):\n", rows, cols);
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         printf("%d ", matrix[i][j]);
      }
      printf("\n");
```

```
    }
}
// Function to find the transpose of a matrix
void transposeMatrix(int rows, int cols, int matrix[rows][cols], int result[cols][rows]) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            result[j][i] = matrix[i][j];
        }
    }
}
int main()
{
    int rows, cols;
    // Input rows and columns of the matrix
    printf("Enter the number of rows and columns of the matrix: ");
    scanf("%d %d", &rows, &cols);
    int matrix[rows][cols], transposed[cols][rows];
    // Read the original matrix
    readMatrix(rows, cols, matrix);
    // Display the original matrix
    printf("\nOriginal ");
    displayMatrix(rows, cols, matrix);

    // Find the transpose of the matrix
    transposeMatrix(rows, cols, matrix, transposed);
    // Display the transposed matrix
    printf("\nTransposed ");
    displayMatrix(cols, rows, transposed);
    return 0;
}
```

Sample Input:

```
Enter the number of rows and columns of the matrix: 2 3
Enter elements of the matrix (2 x 3):
1 2 3
4 5 6
```

Sample Output:

```
Original Matrix (2 x 3):
1 2 3
4 5 6
Transposed Matrix (3 x 2):
1 4
2 5
3 6
```

2) Program to find the maximum element in the given matrix.

```c
#include <stdio.h>
void readMatrix(int,int,int[][]);
void displayMatrix(int,int,int[][]);
void findMaxElement (int,int,int[][]);
// Function to read a matrix
void readMatrix(int rows, int cols, int matrix[rows][cols]) {
   printf("Enter elements of the matrix (%d x %d):\n", rows, cols);
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         scanf("%d", &matrix[i][j]);
      }
   }
}
// Function to display a matrix
void displayMatrix(int rows, int cols, int matrix[rows][cols]) {
   printf("Matrix (%d x %d):\n", rows, cols);
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         printf("%d ", matrix[i][j]);
      }
      printf("\n");
   }
}
// Function to find the maximum element in a matrix
int findMaxElement(int rows, int cols, int matrix[rows][cols]) {
   int max = matrix[0][0]; // Assume the first element is the maximum
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         if (matrix[i][j] > max) {
            max = matrix[i][j];
         }
      }
   }
   return max;
}
int main()
{
   int rows, cols;
   // Input rows and columns of the matrix
   printf("Enter the number of rows and columns of the matrix: ");
   scanf("%d %d", &rows, &cols);

   int matrix[rows][cols];
```

```
    // Read the matrix
    readMatrix(rows, cols, matrix);

    // Display the matrix
    printf("\n");
    displayMatrix(rows, cols, matrix);

    // Find and display the maximum element
    int max = findMaxElement(rows, cols, matrix);
    printf("\nThe maximum element in the matrix is: %d\n", max);

    return 0;
}
```

MCQ Questions with Answers

1. What is the correct syntax to declare a function in C?

A. function int myFunc(int x); B. int myFunc(int x);

C. int myFunc(x int); D. declare int myFunc(x);

2. What is the default return type of a function in C if not specified?

A. int B. void C. float D. Undefined

3. Which of the following is true about functions in C?

A. A function can return multiple values. B. A function must always have a return statement.

C. A function can be called multiple times. D. A function cannot be recursive.

4. What is the purpose of the void keyword in a function declaration?

A. To indicate no parameters. B. To indicate no return value.

C. To declare an infinite function. D. To declare a global function.

5. How do you pass an array to a function in C?

A. By value B. By reference C. By pointer D. By memory

6. Which of the following is true about recursion in C?

A. Every recursive function must have a base case. B. Recursion is faster than iteration.

C. Recursion uses less memory than iteration.

D. Recursion is allowed only for mathematical functions.

7. What does the following code print?

```
void myFunc() {
    printf("Hello, World!");
}
int main() {
    myFunc();
    return 0;
}
```

A. Hello, World! B. Nothing C. Compile-time error D. Run-time error

8. What happens if a function does not have a return statement in C?

A. Compile-time error

B. Garbage value is returned if int return type is used.

C. It always returns 0.
D. Function execution fails.

9. What is the output of the following code?

```
int add(int a, int b) {
    return a + b;
}
int main() {
    printf("%d", add(2, 3));
    return 0;
}
```

A. 2 B. 3 C. 5 D. Compile-time error

10. Which storage class is used to define global functions?

A. auto B. register C. extern D. static

11. What is the maximum number of arguments a function in C can have?

A. 127 B. 255 C. Compiler-dependent D. Infinite

12. What happens if a function is called before its declaration in C?

A. Compile-time error B. The function executes normally.
C. Undefined behavior D. The program doesn't compile.
Answer: A

13. Which of the following is used to stop function execution and return a value?

A. exit B. break C. return D. continue

14. In which section of a program is a user-defined function defined?

A. Preprocessor section B. Main function C. Function definition section D. Library section

15. What does a function prototype specify?

A. The memory layout of a function B. The return type and parameter list of a function
C. The actual implementation of the function D. The scope of the function

16. What is the output of the following code?

```
int foo(int x) {
    return x * 2;
}
int main() {
    printf("%d", foo(4));
    return 0;
}
```

A. 2 B. 4 C. 8 D. Compile-time error

17. What happens when a void function tries to return a value?

A. Compile-time error B. Returns a garbage value
C. Returns 0 D. Function execution continues normally

18. What is the correct syntax to define a recursive function in C?

A. A function calling another function B. A function calling itself
C. A function defined inside another function D. A function with a static keyword

19. What is a library function in C?

A. A user-defined function stored in a library. B. A function available in C's standard library.
C. A global function. D. A private function.

20. What is the purpose of the main function in a C program?

A. To include all header files.

B. To serve as the entry point of the program.

C. To declare global variables.

D. To initialize the system.

Important Questions

1) Define a function. Explain its advantages with examples.
2) Differentiate between built-in and user-defined functions with examples.
3) Describe function prototype, function definition and function call with an example.
4) Differentiate between call-by-value and call-by-reference with examples.
5) How are arrays passed to functions in C and write a C program to find the largest element in an array using functions.
6) Differentiate between local and global variables with examples.
7) List and explain the different storage classes in C with examples.
8) Define recursion. What are the key components of a recursive function? Write a program to find the Fibonacci series using recursion.