MODULE 3

1. Define function. Why are functions needed? (What are the advantages of functions?)

A function is a block of a code that performs a specific task.

C enables programmers to break up a program into segments known as **functions**, each of which can be written more or less independently of the others.

- ➢ Dividing a program into separate functions facilitates each function to be written and tested separately. This simplifies the process of getting the total program to work.
- ➢ Understanding, coding, and testing multiple separate functions is easier than doing it for one big function.
- ➢ If a big program has to be developed without the use of any function other than main() function, then there will be countless lines in the main() function and maintaining this program will be very difficult.
- ➢ All the libraries in C contain a set of functions that the programmers are free to use in their programs. Programmers can use them without worrying about their code details.
- ➢ When a big program is broken into smaller functions, then different programmers working on that project can divide the workload by writing different functions.
- ➢ Like C libraries, programmers can also write their functions and use them at different points in the main program or in any other program that needs its functionalities.

2. Discuss the various ways of passing parameter to the functions. (Distinguish between Call by Value and Call by Reference using suitable example).

There are two ways in which arguments or parameters can be passed to the called function.

- **Call by value** in which values of variables are passed by the calling function to the called function.
- **Call by reference** in which address of variables are passed by the calling function to the called function.

Call by Value: In the call-by-value method, the called function creates new variables to store the value of the arguments passed to it. Therefore, the called function uses a copy of the actual arguments to perform its intended task.

If the called function is supposed to modify the value of the parameters passed to it, then the change will be reflected only in the called function. In the calling function no change will be made to the value of the variables.

Consider the code given below. The add( ) function accepts an integer variable num and adds 10 to it. In the calling function, the value of num=2. In add( ), the value of num is modified to 12 but in the calling function the change is not reflected.

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

```
void add( int n );
int main ( )
{
int num = 2;
printf( "The value of num before calling the function = %d \n", num );
add ( num );
printf( "The value of num after calling the function = %d \n", num );
return 0;
}
void add ( int n )
{
n = n + 10;
printf( "The value of num in the called function = %d \n", n);
}
```

Output

```
The value of num before calling the function = 2
The value of num in the called function = 12
The value of num after calling the function = 2
```

Call by Reference: In call by reference, we declare the function parameters as references rather than normal variables. When this is done any changes made by the function to the arguments it receives are visible in the calling function.

To indicate that an argument is passed using call by reference, an asterisk (*) is placed after the type in the parameter list. This way, changes made to the parameter in the called function will then be reflected in the calling function.

Hence, in call-by-reference method, a function receives an implicit reference to the argument, rather than a copy of its value. Therefore, the function can modify the value of the variable and that change will be reflected in the calling function as well.

The following program uses this concept.

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

```
int main ( )
{
int num = 2;
printf( "The value of num before calling the function = %d \n", num );
add ( &num );
printf( "The value of num after calling the function = %d \n", num );
return 0;
}
void add ( int *n )
{
*n = *n + 10;
printf( "The value of num in the called function = %d \n", *n);
}
```

Output

```
The value of num before calling the function = 2
The value of num in the called function = 12
The value of num after calling the function = 12
```

3. Define storage class. Explain the different storage classes supported by C.

C supports four storage classes: automatic, register, external, and static. The general syntax for specifying the storage class of a variable can be given as:

<storage_class_specifier> <data type> <variable name>

auto Storage Class: The **auto** storage class specifier is used to explicitly declare a variable with automatic storage. It is the default storage class for variables declared inside a block. For example, if we write

auto int x;

then x is an integer that has automatic storage. It is deleted when the block in which x is declared is exited.

The auto storage class can be used to declare variables in a block or the names of function parameters.

register Storage Class: When a variable is declared using **register** as its storage class, it is stored in a CPU register instead of RAM. Since the variable is stored in a register, the maximum size of the variable is equal to the register size. A register variable is declared in the following manner:

register int x;

Register variables are used when quick access to the variable is needed. Like auto variables, register variables also have automatic storage duration.

extern Storage Class: The **extern** storage class is used to give a reference of a global variable that is visible to all the program files. To declare a variable x as extern write,

extern int x;

External variables may be declared outside any function in a source code file as any other variable is declared. But usually external variables are declared and defined at the beginning of a source file.

Memory is allocated for external variables when the program begins execution, and remains allocated until the program terminates. External variables have global scope, i.e., these variables are visible and accessible from all the functions in the program.

static Storage Class: While **auto** is the default storage class for all local variables, **static** is the default storage class for all global variables. Static variables have a lifetime over the entire program, i.e., memory for the static variables is allocated when the program begins running and is freed when the program terminates. To declare an integer x as static, write

static int x = 10;

Here x is a local static variable. Static local variables when defined within a function are initialized at the runtime.

4. What is recursion? Explain different types of recursion.

A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.

Recursion is defining a large and complex problem in terms of smaller and more easily solvable problems. Any recursive function can be characterized based on:

- whether the function calls itself directly or indirectly (direct or indirect recursion),
- whether any operation is pending at each recursive call (tail-recursive or not), and
- the structure of the calling pattern (linear or tree-recursive).

Direct Recursion: A function is said to be directly recursive if it explicitly calls itself.
For example, consider Figure.
Here, Func( ) calls itself for all positive values of n, so it is said to be a directly recursive function.

```
int Func(int n)
{
if (n==0)
return n;
else
return (Func(n-1));
}
```
Fig: Direct recursion

Indirect Recursion: A function is said to be indirectly recursive if it contains a call to another function which ultimately calls it.
Look at the functions given in Figure.
These functions are indirectly recursive as they both call each other.

```
int Func1( int n )
{
if ( n== 0 )
return n;
else
return Func2( n );
}
```

```
int Func2( int x )
{
return Func1( x-1 );
}
```
Fig: Indirect recursion

Tail Recursion: A recursive function is said to be tail recursive if no operations are pending to be performed when the recursive function returns to its caller. When the called function returns, the returned value is immediately returned from the calling function.
For example, the factorial function shown in figure is a non-tail-recursive function because there is a pending operation of multiplication to be performed on return from each recursive call.
However, the same factorial function can be written in a tail-recursive manner as shown in Figure.
Here, Fact1 function preserves the syntax of Fact (n).
Here the recursion occurs in the Fact1 function and not in Fact function.

```
int Fact(int n)
{
if ( n==1 )
return 1;
else
return (n* Fact(n-1));
}
```
Figure: Non-tail recursive function

```
int Fact( n)
{
retuen Fact1( n, 1 );
}
int Fact( int n, int res )
{
if ( n==1 )
return res;
else
return Fact1( n-1, n*res);
}
```
Figure: Tail-recursive function

5. Define array. Write the syntax for declaring and initializing 1D arrays with suitable example.

An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an index known as subscript.

Declaring arrays: Arrays are declared using the below syntax:

data_type name[size];

Here, data_type can be either int, float, double, char or any other valid data type. The size indicates the number of elements in the array and the name is any valid name given for identifying the array.

Example: int marks[10];

Initializing arrays:

Elements of the array can also be initialized at the time of declaration. Arrays are initialized as: type array_name [size] = { list of values };

The values are written within curly brackets and every value is separated by a comma.

Example: int marks[5] = { 90, 82, 78, 95, 88 };

Here, an array with name marks is declared that has enough space to store 5 elements. The first element, i.e., marks [0] is assigned with the value 90. Similarly, the second element of the array, i.e., marks [1] is assigned 82, and so on.

While initializing the array at the time of declaration, the programmer may omit the size of the array. For example: int marks[ ] = { 90, 80, 70, 60 };

Here, the compiler will allocate enough space for all initialized elements.

If the number of values provided is less than the number of elements in the array, the un-assigned elements are filled with zeroes. For example: int marks [5] = { 98, 90, 72};

All the elements in the array can be initialized with a single value as well. For example:

int marks [5] = { 0 }; Figure below illustrates initialization of arrays.

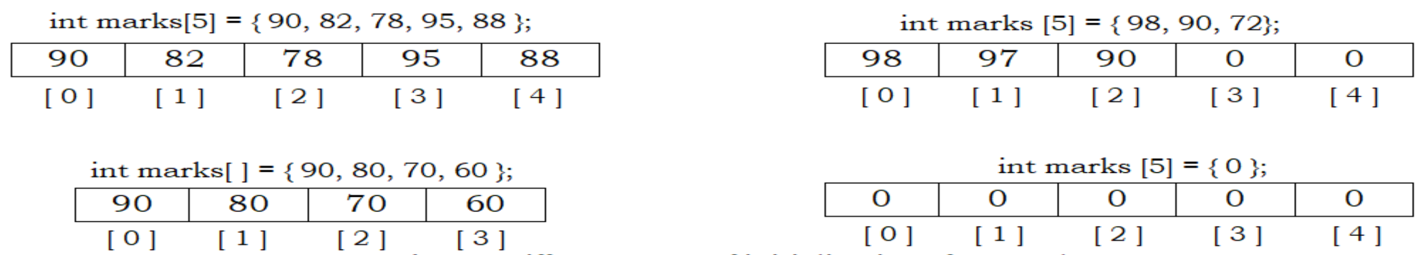


Figure: Different ways of initialization of array elements

6. How 1D integer array is represented in memory. With the help of suitable example demonstrate the initializing the element.

If we declare: int marks [7];

Then above statement declares 'marks' to be an array containing 7 elements. The array index starts from zero. The 1st element will be stored in marks [0], the 2nd element will be stored in marks [1] and so on. The memory representation is shown below:

1st element	2nd element	3rd element	4th element	5th element	6th element	7th element
marks [0]	marks [1]	marks [2]	marks [3]	marks [4]	marks [5]	marks [6]

Initializing Array Elements:

Elements of the array can also be initialized at the time of declaration. When an array is initialized, a value for every element in the array has to be given. Arrays are initialized as below:

type array_name [size] = { list of values };

The values are written within curly brackets and every value is separated by a comma.

Example: int marks[5] = { 90, 82, 78, 95, 88 };

Here, an array with name marks is declared that has enough space to store 5 elements. The first element, i.e., marks [0] is assigned with the value 90. Similarly, the second element of the array, i.e., marks [1] is assigned 82, and so on.

While initializing the array at the time of declaration, the programmer may omit the size of the array. For example: int marks[ ] = { 90, 80, 70, 60 };

Here, the compiler will allocate enough space for all initialized elements.

If the number of values provided is less than the number of elements in the array, the un-assigned elements are filled with zeroes. For example: int marks [5] = { 98, 90, 72};

All the elements in the array can be initialized with a single value as well. For example:

int marks [5] = { 0 };

Figure below illustrates initialization of arrays.

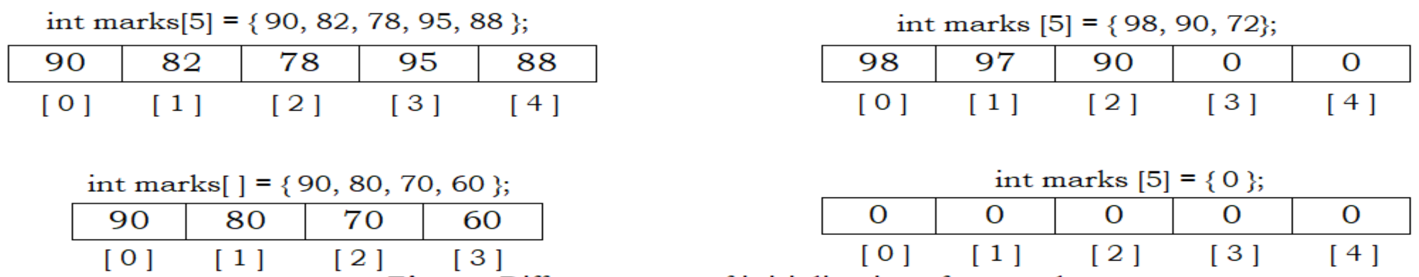


Figure: Different ways of initialization of array elements

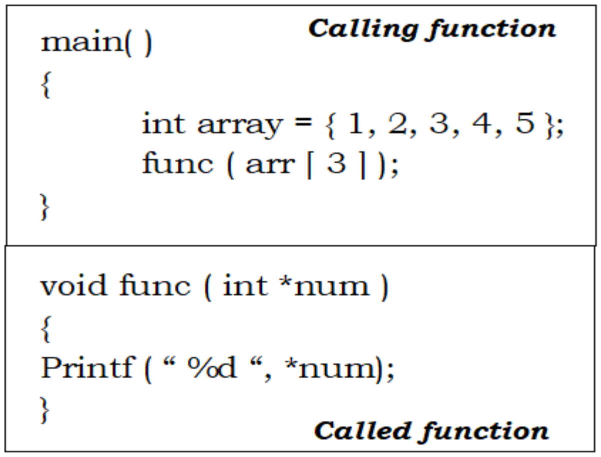
7. Explain different ways of passing arrays into a function with example.

An array can be passed to a function in different ways.

Passing Individual Elements: The individual elements of an array can be passed to a function by passing either their data values or their addresses.

Passing Data Values: The individual elements can be passed in the same manner as we pass variables of any other data type. The condition is just that the data type of the array element must match with the type of the function parameter. Figure below shows the code to pass an individual array element by passing data value.

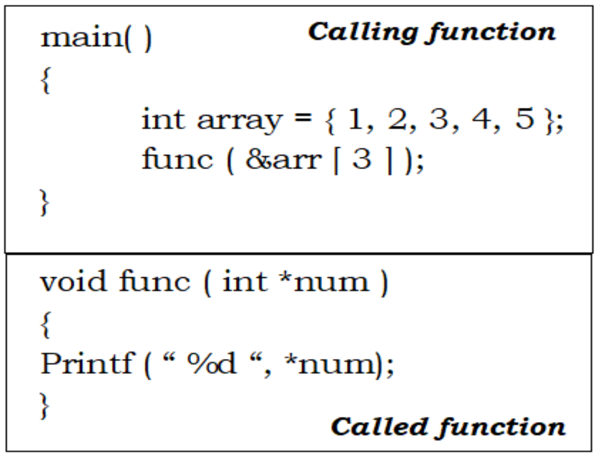
In the example given, only one element of the array is passed to the called function. This is done by using the index expression. So arr[3] actually evaluates to a single integer value.



```
main( )                          Calling function
{
        int array = { 1, 2, 3, 4, 5 };
        func ( arr [ 3 ] );
}
```

```
void func ( int *num )
{
Printf ( " %d ", *num);
}                                Called function
```

Figure: Passing value of individual array elements to a function



```
main( )                          Calling function
{
        int array = { 1, 2, 3, 4, 5 };
        func ( &arr [ 3 ] );
}
```

```
void func ( int *num )
{
Printf ( " %d ", *num);
}                                Called function
```

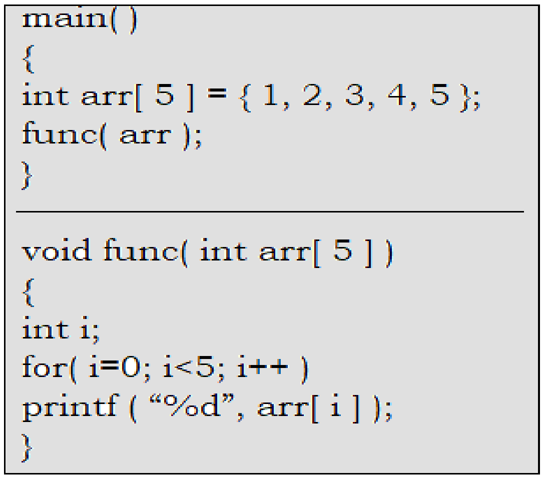
Figure: Passing address of individual array elements to function

Passing Addresses: Like ordinary variables, we can pass the address of an individual array element by preceding the indexed array element with the address operator (&). Therefore, to pass the address of the fourth element of the array to the called function, we will write &arr[3].

However, in the called function the value of the array element must be accessed using the indirection (*) operator as shown in above figure.

Passing the entire array: To pass an entire array to a function, simply pass the name of the array.

Figure illustrates the code which passes the entire array to the called function.



```
main( )
{
int arr[ 5 ] = { 1, 2, 3, 4, 5 };
func( arr );
}
```

```
void func( int arr[ 5 ] )
{
int i;
for( i=0; i<5; i++ )
printf ( "%d", arr[ i ] );
}
```

Fig: Passing entire array to the function

8. Write a C program to find factorial of a number using functions.

```
#include <stdio.h>
int fact( int n );                // Function Declaration
int main ( )
{
int m, factorial=0;
printf ( "Enter a positive integer: \n" );
scanf ( "%d", &n );
factorial = fact( m );            // Function Call
printf ( " Factorial of %d is = %d", m, factorial);
}
int fact( int n )                 // Function Definition : Function Header
{                                     // Function Body
int result;
if ( n==1 )
return 1;
else
result = n * fact( n-1);
return result;
}
```

9. Develop a C program to print Fibonacci series using recursion.

```
#include <stdio.h>
int fibonacci ( int i );          // Function Declaration
int main ( )
{
int n, i, result=0;
printf ( "Enter the number of terms: \n" );
scanf ( "%d", &n );
printf ( "Fibonacci series is : " );
for ( i=0; i<n; i++ )
{
result = fibonacci ( i );          // Function Call
printf ( "%d ", result);
}
}
int fibonacci ( int i )              // Function Definition : Function Header
{                                    // Function Body
int total;
if ( i==0 )
return 0;
else if ( i==1 )
return 1;
else
total = fibonacci(i-1) + fibonacci(i-2));
return total;
}
```

10. Develop a C program to add two integers using functions.

```
#include <stdio.h>
int sum ( int a, int b );          // Function Declaration
int main( )
{
int num1, num2, total = 0;
printf ( "Enter the first number: \n ");
scanf ( "%d", &num1 );
printf ( "Enter the second number: \n" );
scanf ( "%d", &num2 );
total = sum ( num1, num2 );          // Function Call
printf ( "Total = %d \n", total);
return 0;
}
int sum ( int a, int b )          // Function Definition : Function Header
{                                 // Function Body
int result;
result = a + b;
return result;
}
```

11. Write a program to calculate GCD using recursive function.

```
# #include <stdio.h>
int GCD (int x, int y);
int main()
{
    int n1, n2, res = 0;
    printf ( " Enter the two number \n ");
    scanf ("%d %d", &n1, &n2);
    res = GCD( n1, n2);
    printf ( " GCD of the two numbers is %d", res);
    }
int GCD (int x, int y)
{
    int rem, result;
    rem = x % y;
    if(rem==0)
    return y;
    else
    result= GCD(y, rem);
    return result;
}
```

12. Write a C program to implement linear search.

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

```
int main( )
{
int a[ 10 ], n, i, key, found=0;;
printf ( "Enter the number of elements in the array \n ");
scanf ( "%d", &n );
printf ( "Enter the elements: \n" );
for ( i = 0; i < n; i++ )
scanf ( "%d", &a[ i ] );
printf ( "Enter the number that has to be searched \n ");
scanf ( "%d", &key );
for ( i = 0; i < n; i++ )
{
if( a[ i ] == key )
{
found = 1;
break;
}
}
if( found == 1 )
printf( " The key number is found at position %d", i+1 );
else
printf( " The key number does not exist in the array" );
}
return 0;
}
```

13. Develop a C program to sort the given set of N numbers using selection sort.

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

```
int arr[100], n:
int i, j, temp, position;
printf( "Enter the number of elements in the array \n " );
scanf( "%d", &n );
printf("Enter array elements: \n");
for ( i = 0; i < n; i++ )
scanf ( "%d", &a[ i ] );
/*sort elements in Ascending Order*/
for(i=0; i<n; i++)
{
position=i;
for( j=i+1; j<n; j++ )
{
if( arr[ position ] > arr[ j ])
      {
        position = j;
      }
      if( position !=i )
      {
        temp = arr[ i ];
        arr[ i ] = arr[ position ];
        arr[ position ] = temp;
      }
    }
  }
printf("Array elements in Ascending Order:\n");
for(i=0; i<n; i++)
printf( "%d ",arr[ i ] );
printf( "\n" );
return 0;
}
```

14. Write a C program to implement binary search.

```
##include <stdio.h>
int main( )
{
int a[100], n:
int i, key, beg, end, mid;
printf( "Enter the number of elements in the array \n " );
scanf( "%d", &n );
printf("Enter array elements: \n");
for ( i = 0; i < n; i++ )
scanf ( "%d", &a[ i ] );
printf ( "Enter the number that has to be searched \n ");
scanf ( "%d", &key );
beg=0, end=n-1;
mid = ( beg + end ) / 2;
while( beg <= end )      {
if( a[ i ] < key )
beg = mid + 1;
else if( a[ i ] == key)      {
printf( “ %d is found in the array at position %d”, key, mid );
break;
}
else
end = mid - 1;
mid = ( beg + end ) / 2;
}
if( beg > last )
printf( “ Not found. %d is not present in the array”, key );
return 0;
}
```