MODULE 3: Arrays, Functions and Strings

Contents covered in this module

I. Using an Array II.

II. Functions in C

III. Argument Passing

IV. Functions and Program Structure, locations of functions

V. Function Design VI. Recursion

VII. Multidimensional Arrays

VIII. Using Arrays with Functions

IX. Declaring, Initializing, printing and reading strings

X. String Input and Output Functions XI. String Manipulation Functions

XII. Array of Strings

XIII. Programming Examples

I. Using an Array

Array is collection of similar data items or elements. Array is a collection of elements of the same data type. All the elements of the array are stored in contiguous (sequential) memory locations.

Array is a sequential collection of elements of same type under a single name. Pictorial representation of an array is

Array Element	10	13	4	26	38
Index or Position or Subscript	0	1	2	3	4
Address (Assumed Values)	1000 (starting address)	1004	1008	**1012**	1016

In the above representation, an array is a collection of 5 integer elements.
The addresses are of each element of an array is contiguous in nature. Hence, the addresses are 1000, 1004, 1008, 1012 and 1016 since **size of each integer data is 4 bytes**.
Address of any element of an array can be calculated by using the following formula

Element Address = Starting Address + (sizeof(element) * index)

For instance, 4^{th} element index is 3 from the above, hence

4^{th} element address = 1000 + (4 * 3)
= 1000 +12
= **1012**

The position or index of first element of an array is always ZERO (0). The index or position of next elements is one more than the previous index or position. Hence, the indices in the representations are 0, 1, 2, 3, and 4.

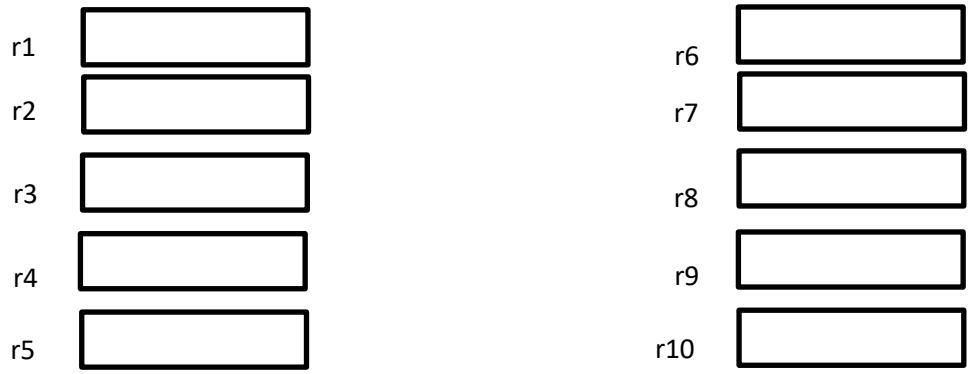
Why Arrays? or What is the need for arrays?

Imagine we have a problem that requires us to read, process and print 10 integers. To begin, we can declare and define 10 variables, each with a different name as shown below;

int r1, r2, r3, r4, r5, r6, r7, r8, r9, r10;

This creates 10 variables and allocates 10 different blocks of memory one for each variable as shown below.

Having 10 different names, how can we read 10 integers from the keyboard and store them? To read 10 integers from the keyboard, we need 10 read statements, each to a different variable. Furthermore, once we have them in memory, how can we print them? To print them we need 10 print statements.



For example, the following program illustrates how to read and print 10 integer numbers into 10 different names.

```
#include<stdio.h
> int main() {
        int     r1, r2, r3, r4, r5, r6, r7, r8, r9, r10;
        // read 10 integers into 10 different
names scanf("%d", &r1);       scanf("%d", &r2);
        scanf("%d", &r3); scanf("%d", &r4);
scanf("%d", &r5); scanf("%d", &r6);
        scanf("%d", &r7);       scanf("%d", &r8);
scanf("%d", &r9); scanf("%d", &r10);

        //print 10 integer numbers from 10 different
names printf("%d", r1);          printf("%d", r2);
        printf("%d", r3); printf("%d", r4); printf("%d",
r5); printf("%d", r6);   printf("%d", r7);
        printf("%d", r8);
        printf("%d", r9);
        printf("%d", r10);

        return 0;
}
```

Although this approach may be acceptable for 10 variables but how about 100 variables, 1000 variables or 10000 variables? It is definitely not acceptable.

Hence, we need more powerful data structure to process large amounts of data. Hence the array is used for processing large amounts of data of same type.
By using arrays the above program can be rewritten as follows

```
#include<stdio.h> int
main()
{
        int      r[10];
        int i;
        // read 10 integer numbers
        for(i=0;i<10;i++)
        scanf("%d" , &a[i]);
        //print 10 integer numbers
        for(i=0;i<10;i++)
                printf("%d" , a[i]);
        return 0;
}
```

This example program is much easier to read, understand and alter the number of elements to be read and print as per the requirement. Hence, the arrays are much better than variables for storing large amounts of same type of data. Hence the various applications and advantages are listed below.

Applications of Arrays

Various applications and advantages of arrays are

- Can store large elements of same type
- Can replace multiple different variable names of same type by single name
- Can be used for sorting and searching applications
- Can be used for representing matrix
- Can be used in recursion
- Can be used for implementing various data structures like stack, queue, etc.

Different Types of Arrays

In C, the arrays can be classified as two types based on the arrangement of data.

a. One-Dimensional Array

b. Multi-Dimensional Array

One-Dimensional Array:

An array is one in which elements are arranged linearly. If an array contains only one subscript then it is called as one-dimensional array.

What is Index or Subscript?

- Individual data items can be accessed by the name of the array and an integer enclosed in square bracket called subscript / index
- Subscripts helps us to identify the element number to be accessed in the contiguous memory

Declaration and definition of one-dimensional array

An array must be declared and defined before it can be used. Array declaration and definition tells the compiler

- Name of the array
- The size or the number of elements in the array
- The type of each element of the array
- Total amount of memory to be allocated to the array

The size of the array is a constant and must have a value at compilation time.

The declaration format or syntax of declaration is given below;

Type array_name [arraysize] ;

Where,

Type - refers to the type of each element of the arrays which can be int, float, char, double, short, long, etc.

array_name - refers to the name of the array to be specified with the help of rules of identifier

arraysize - refers to the size or number of elements of the array. This **arraysize** can be a symbolic constant, integer constant or integer arithmetic expression.

Examples:

a. declare and define an array by the name **rollno** which contains **50 integer** elements. In this example, **arraysize** is an **integer constant.** int rollno[50];

b. declare and define an array called **average** which contains 10 floating point elements. In this example, **arraysize** is an **integer constant**. float average[10];

c. #define MAX 1000 //defines a macro MAX with value 1000 char string [MAX]; // declares and defines an array by the name **string** that

can contain **1000 characters.** In this example**, arraysize** is **symbolic constant.**

d. double salary[10+5]; // declares and defines an array called **salary** which can contain 15 (result of 10 + 5) double precision floating point numbers. In this example, **arraysize** is **integer arithmetic expression.**

Accessing Elements in Arrays

C uses an **index** to access individual elements in an array. The index must be an integral value or an expression that evaluates to an integral value. For example, consider the following array

int marks[5];

To access **first element** of the array **marks**, we can use **marks[0],** since the index of first element is ZERO (0).

Similarly, second element of the array can be accessed by marks[1], third element by marks[2], etc.

Storing Values in Arrays

Declaration and definition only reserve space for the elements in the array. No values are stored.

If we want to store values in the array, we can use the following methods of initialization

- initialize the elements **(Initialization)**
- read values from the keyboard **(Inputting)**
- assign values to each individual element **(Assignment)**

Initialization of an array

Providing a value for each element of the array is called as initialization. The list of values must be enclosed in curly braces.

NOTE: It is a compile time error if we provide more values than the size of the array.

Examples:

a) basic initialization

```
int numbers[5] = { 3, 23, 17, 8, 19};
```

In this example, first element numbers[0] is initialized to 3
second element numbers[1] is initialized to 23
third element numbers[2] is initialized to 17
fourth element numbers[3] is initialized to 8
fifth element numbers[4] is initialized to 19

b) initialization without size of the array

```
int numbers[ ] = { 3, 7, 12, 29, 30 };
```

In this example, since the size of the array is not specified in brackets, size of array is decided based on number of elements in the curly braces. Hence the size of array is 5.

c) Partial initialization

```
int numbers[5] = {4, 9};
```

In this example, the array is partially initialized, since the number of values provided in curly braces are fewer than the size of the array. Therefore, it initializes first element with 4, second element with 9 and unassigned or uninitialized elements are filled with ZEROs automatically.

d) Initialization of all elements to ZEROs

```
int numbers[100] = {0};
```

In this example, all the elements are filled with ZEROs.

Inputting Values

Another way to fill the array is to read the values from the keyboard or file by using scanf statement or fscanf statement. For Example,

```
int numbers[5];
int i;
for(i=0;i<5;i++)
scanf("%d" , &numbers[i]);
```

In this example, scanf function is repetitively called for 5 times and reads the values for all elements of the array. Since the starting index of array is ZERO, so index **i** is initialized to 0 in for loop and the condition i<5 iterates loop for 5 times i.e., the number of elements in the array.

Assigning values

We can assign values to individual elements using the assignment operator. Any value that reduces to the proper type of array can be assigned to an individual array element. For Example,

```
                int     numbers[5];
        numbers[0]  =       10;     //assigns first element to 10
numbers[1]  =       4;      //assigns second element to 4
```

II. Functions in C

Function is an independent module which contains set of instructions to perform a particular task.

Why Functions? Or What is the need for functions in C?

Since C is modular programming language, the larger problem or complex problem can sub divided into smaller problems, each of these smaller problems can be developed independently by using functions. Hence, by using functions in C, we can have so many advantages listed below. The process of subdividing larger problem into smaller problem is called modular approach.

Advantages of Functions

- It is easy to read, understand, test, manage, and debug the programs.
- Reduces the development time
- Reusability of code can be done
- We can protect data, since one function cannot access local data of another function.
- Complexity of the program can be reduced.
- Users can create their own libraries i.e user-defined libraries.

Types of Functions in C

There are two types of functions in C.

a. Standard functions or library functions:

The functions which have pre-defined purpose and readily available for use in header files are called as standard functions or library functions. Examples:

sqrt() math.h - used to find square root of a given number. Available in header file.

sin() - used to find sin(x). Available in math.h header file.

printf() - used to display data in required form. Available in stdio.h header file.

scanf() - used to read data in required form. Available in stdio.h header file.

b. User-defined functions:

The functions which are defined by user to perform a specific task are called as user-defined functions.

Using User-defined functions in C

To understand how to use user-defined functions in C and how program is executed when functions are used in C, we will start with a simple program and its flow of execution. One

must understand different elements of functions and various statements to be used while designing functions in C program.

Explanation:

The below shown program illustrates finding a square of a given number using a user-defined function square(). It also illustrates flow of program execution and the three elements of function.

Understanding flow of program execution

a. Firstly Operating System will call main function.
b. When control comes inside main function, execution of main starts (i.e execution of C program starts) and executes Line 5 first
c. Consider Line 6 i.e., result = square(5),
   In this statement, a function call is made to the user-defined function **square**() which accepts one parameter i.e., 5 to find a square of 5. Hence, it halts or stops the execution of main function at line 6 and the control is transferred to the function definition i.e., line 10 as shown with arrow. The value of function parameter is copied to variable x.
d. Then function definition is executed line by line i.e., from line 10 to line 14.
e. Consider line 14 i.e. return y,
   In this statement the value of y i.e. 25 is returned to the calling function i.e. main function. Hence, the called function i.e., square() completes its execution and returns the control back to calling function i.e. main function to the line 6 as shown with arrow. The return value i.e. 25 is copied to variable result.
f. Then main function will resume its execution and executes its statements.
g. Lastly the main function terminates with return value 0 to the operating system. From the example, we can also define the following basic definitions related to functions of C.

Calling Function: The function which invokes another function to do something is known as calling function.

Called Function: The function which is invoked by another function is known as called function.

Actual Parameters: the parameters in a function call which are used to send one or more values to the called function are known as actual parameters.

For example:

If a function call is

```
addition(p,q);
```

Then in this function call the parameters listed in parenthesis namely variables p and q are actual parameters whose value is passed to the formal parameters.

Formal Parameters: the parameters in function declaration or definition which are used to receive one or more values from the calling function are known as formal parameters.

For example:

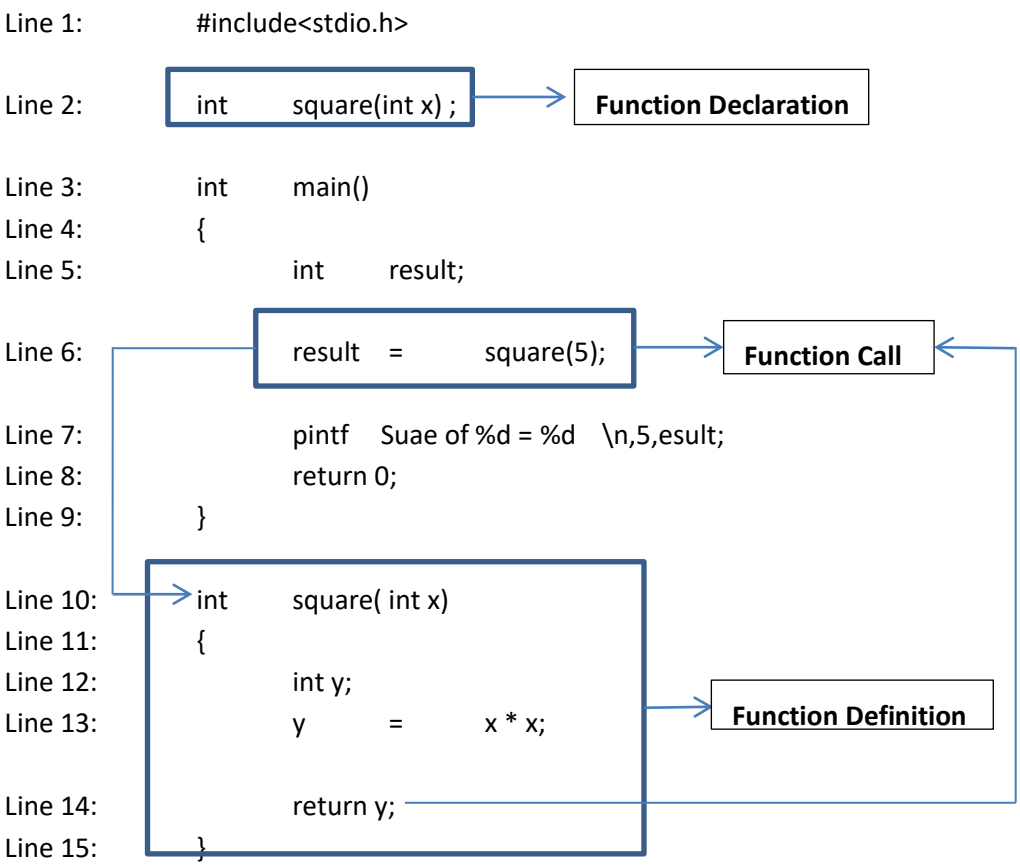
If a function definition is

```
void  addition(int x, int y)
{
        printf("sum = %d\n", x+y);
}
```

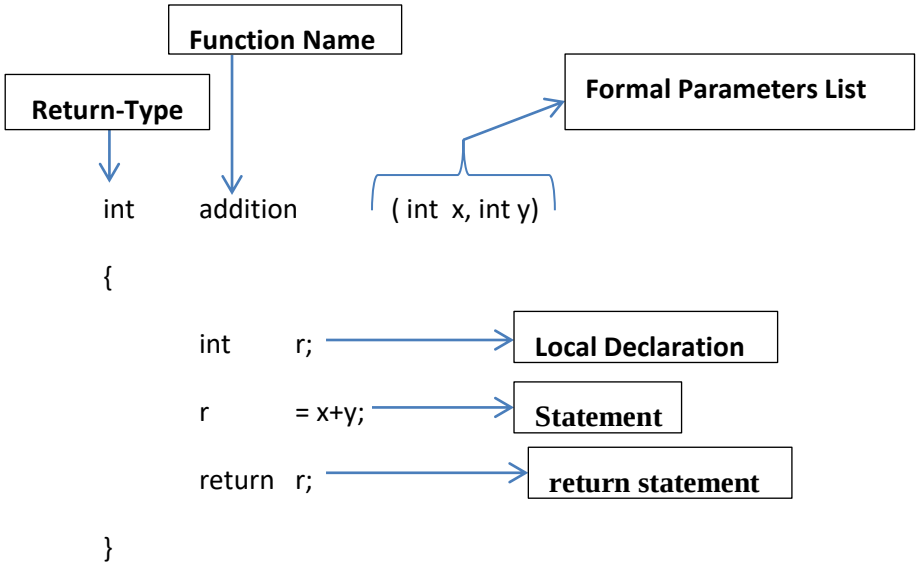
In this function definition, the parameters listed in function header namely x and y are formal parameters which will receive the copy of actual parameters.

Return statement: the statement which is used to return a value to the calling function from called function is known as return statement.

Function Name: the name given to the function which must be framed with the help of rules of identifier.



Pictorial view of all above definitions can be shown as given below



Elements of Function

From the above example, one can observe that to define user-defined functions in C program, the three elements of functions are very important and need to understand how to use and purpose of these elements.

Three elements of function are

1. Function Declaration or Function Prototype
2. Function call
3. Function Definition

1. Function Declaration or Function Prototype or Function Header

As we declare variables, arrays, etc, the function must be declared before we use it. The function declaration tells the compiler that we are going to define this function somewhere in the program. It gives prior information about the function to the compiler. It contains only a function header.

The declaration tells the compiler about

- Name of the function
- Return type of the function
- List of formal parameters and their types

Hence, the general format or syntax for declaring function is as given below. The declaration must end with semicolon. return-type Function_Name (List of formal parameters) ;

Where,

return-type can be

- void type when function doesn‟t return any value
- any of the standard data type or user-defined data type when function returns a value
  for example: int, float, double, char, short int, long int, etc

Function_Name can be any name framed by using the rules of identifier

List of formal parameters: refers to zero or more parameters to receive either value or address of actual parameters.

Format for writing list of formal parameters

Type1 param1, Type2 param2, Type3 param3, …, TypeN paramN

Where,

Type1,Type2, …, TypeN can be any valid data types of C

Param1, param2, …, paramN are names of formal parameters. All names must be unique. The names must be framed by using rules of identifier.

For example:

a. int addition(int x, int y); // here the function name is addition which accepts two parameters of type integer and returns an integer value.

b. void square(float a); // here function name is square which accepts one parameter of floating point type and doesn‟t return any value.

c. char str(char c, int a); // here function name is str which accepts two parameters one of type character and another of integer type and returns a character.

2. Function Call

Function call is an operator which is used to invoke a function with or without parameters or arguments known as actual parameters. The actual parameters identify the values or addresses that are to be sent to the called function. They can contain zero or more parameters.

They must match the functions formal parameters type and order in the parameter list.

Function call statement must end with semicolon.

General Syntax or format of function call is variable = Fucntion_Name (list of actual parameters) ;

Where,

Variable is one which receives the return value of function if the function return type is Non-Void. If the function return type is void then variable must be ignored.

Function_Name must match to name of the function which is declared before.

List of actual parameters can be zero or more names of variables to send either value or address.

For example,

a. sum = addition(p, q); // it invokes function addition with two actual parameters p and q and return value of function is stored into variable sum.

b. square(x); // invokes a function square with only one actual parameter x

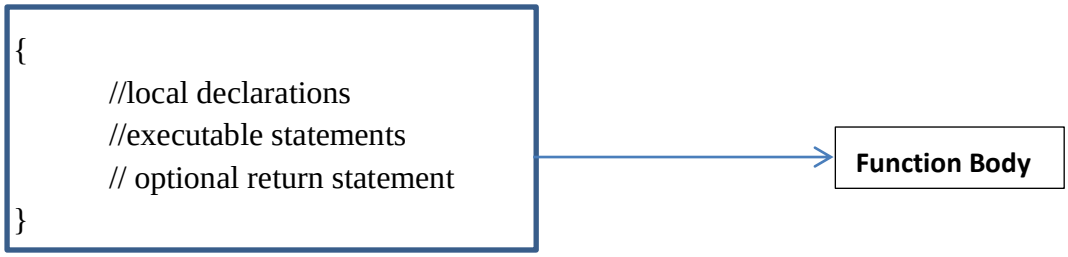
c. ch = str(ch1, x); // invokes a function str with two parameters ch1 and x and stores the return value in variable ch.

3. Function Definition

The function definition contains the code for a function. It is made up of two parts: the function header and the function body, which is a block of statements enclosed in pair of curly braces.

Syntax of function definition is

return-type Function_Name (list of formal parameters) → **Function Header**



A function header consists of three parts: return-type, the function name and list of formal parameters. A semicolon is not used at the end of the function header.
Function body contains local declarations and the set of executable statements to perform a particular task.
If the function return-type is void then function definition must not contain return statement or it can have empty return statement. If the function return-type is non-void then function definition must return with appropriate type of value using return statement.
For example,

a. int addition(int x, int y) {
int sum;
sum = x+y;
return sum;
}
In this function definition, x and y are formal parameters, sum is a local variable. The function returns value of sum using return statement.

b. void square(float x)
{ printf("square of %f = %f\n" , x, x*x);
}
In this function definition, x is a formal parameter of type float. This function prints square of x. Function doesn"t return any value since return type is void hence, it has no return statement.

III. Parameters Passing Mechanisms or Argument Passing Mechanisms

In C, we can two types of communications such as one-way communication and two-way communication. One-way communication may occur from calling function to called function or from called function to calling function. Two-way communication is also known as bidirectional communication which occurs between both calling and called function. The strategies used to implement inter-function communication between the functions are known as parameter passing mechanisms. There are three mechanisms to implement interfunction communications.

1. Call-by-Value or Pass-by-Value method
2. Call-by-Reference or Pass-by-Reference (call-by-address or pass-by-address) 3. Return statement

1. Call-by-Value or Pass-by-Value method

In this mechanism, the values of actual parameters in a function call are copied to the formal parameters.
Any modification to the formal parameters in the function definition will not affect actual parameters.
Default method of function call is call-by-value mechanism.

Example: Program to Swap two integer numbers

```
#include<stdio.h>

void  swap(int x, int y);

int main() {
      int   p,q;
      printf("Enter the value for p and q\n");
      scanf(„%d%d", &p, &q);
      printf("In Main: Before Function

                        swap(10,20)

      printf("In Main: After Function Call\n");
      printf("p=%d, q=%d\n", p, q);
      return 0;
}
void   swap(int x, int y)

      int temp;
Call\n");     printf("p=%d, q=%d\n", p, q);

swap(p,q);
```

Copies value of p=10 to formal parameter x and value of q=20 to formal parameter y and transfers control to function definition

```
{

 printf("In Swap: Before Exchange\n");
      printf("x=%d, y=%d\n", x, y);
      temp=x;          x=y;
      y=temp;
 printf("In Swap: After Exchange\n"); printf("x=%d,
      y=%d\n", x, y);
}
```

Output of the above program is

Enter the value for p and q
10 20
In Main: Before Function Call p=10, q=20
In Swap: Before Exchange x=10, y=20
In Swap: After Exchange
x=20, y=10
In Main: After Function Call p=10, q=20

From the above output and flow shown with arrow and text description, we can understand that the values of actual parameters p and q are copied to formal parameters x and y respectively. From the bold text of output we can understand that the modification that we have done for formal parameters in function definition have not affected actual parameters in calling function hence, the p and q value before function call and after function is same.

2. Call-by-reference or Pass-by-reference method

In this mechanism, the addresses of actual parameters in a function call are copied to the formal parameters.

Any modification to the formal parameters in the function definition will affect actual parameters.

Example: Program to Swap two integer numbers

```
// Pointer is a variable which contains address of another variable
#include<stdio.h> void swap(int *x, int *y); int
main() { int p, q; printf("Enter the value for p
and q\n"); scanf(„%d%d", &p, &q); printf("In
Main: Before Function Call\n"); printf("p=%d,
q=%d\n", p, q);
 swap(1000,2500) swap(&p,&q);

    printf("In Main: After Function Call\n");
    printf("p=%d, q=%d\n", p, q);
    return 0;
}
void  swap(int *x, int *y)
{
    int temp;
    printf("In Swap: Before Exchange\n");
    printf("x=%d, y=%d\n", *x, *y);
    temp=*x;
Enter the value for p and q
10 20
        *x=*y;
        *y=temp;
    printf("In Swap: After
        Exchange\n");
        printf("x=%d, y=%d\n",
        *x, *y);
}
```

Output of the above program is

Copies address of p (&p) assumed that it is at address 1000 to formal parameter *x and address of q (&q) assumed that it is at address 2500 to formal parameter *y and transfers
control to function definition

In Main: Before Function Call p=10,
q=20
In Swap: Before Exchange x=10,
y=20
In Swap: After Exchange x=20,
y=10
In Main: After Function Call p=20,
q=10
From the above output and flow shown with arrow and text description, we can understand that the address of actual parameters p and q are copied to formal parameters *x and *y respectively. From the bold text of output we can understand that the modification that we have done for formal parameters in function definition have affected(modified) actual parameters in calling function hence, the p and q value before function call and after function are different.

IV. Functions, Program Structure and Location of functions

When functions are used with C then general structure of the C program can be organized in two different ways based on **placement or location of the function definition**. C program can be designed in two different approaches namely; top-down approach and bottom-up approach. Hence the program with functions can be organized in two-different ways. In **top-down approach**, main function is designed and developed first then sub functions are designed and developed hence it is known as top-down approach. The program runs from top to bottom.
In **bottom-up approach**, the sub functions are designed and developed first then main function is designed and developed hence the name bottom-up. The program runs from bottom to up.
General Structure for Top-Down approach
Preprocessor Directives
Global Decalrations
User-Defined Functions Declarations
int main()
{
    Local Declarations
    Executable Statements
    Functions call
}
User-Defined Function Definitions
General Structure for Bottom-Up approach Preprocessor
Directives
Global Decalrations
User-Defined Functions defintions
int main()
{
    Local Declarations
    Executable Statements
    Functions call

}

Example for Top-Down Approach

//program to print a greeting message using top-down approach

```
#include<stdio.> void
greeting(void); int
main()
{
        greeting();
        return 0;
}
void  greeting(void)
{
        printf("Good Morning!!!Have a Good Day\n");
}
```

Here, in this program main function is designed first which invokes sub function greeting() to display a greeting message. So sub function is designed later. The program starts its execution from main function and flows through sub function greeting hence, it is top-down approach.

Example for Bottom-Up Approach

//program to print a greeting message using bottom-up approach

```
#include<stdio.> void
greeting(void)
{
        printf("Good Morning!!!Have a Good Day\n");
}       int
main()
{
        greeting();
        return 0;
}
```

Here, in this program sub function greeting is designed and developed first then main function is designed which invokes sub function greeting() to display a greeting message. The program starts its execution from main function (bottom) then flows through sub function greeting() (which is above main()) hence, it is bottom-up approach.

V. Function Design

The functions can be designed in four different forms to establish the communications such as one-way communication or two-way communication. These function designs are categorized based on return value of the function and parameters accepted by the function. So they are also called as categories of function designs.

The four different forms or categories of function design are:

1. Void function without parameters
2. Void function with parameters

3. Non-Void function without parameters
4. Non-Void function with parameters

1. Void function without parameters

In this design, the function doesn‟t return any value and doesn‟t accept any parameters so it is also called as function with no return value and no parameters. When function doesn‟t return value then return type must be void.

Consider the following example program which performs addition of two integer numbers with the help of function. #include<stdio.h> void addition(void); int main()

```
{
        addition();
        return 0;
}
void addition(void)
{
        int   p, q,sum;
printf(“Enter p and q\n”);
scanf(“%d%d”, &p, &q);       sum
= p + q;          printf(“sum =
%d\n”, sum);
}
```

In this program, the sub function addition() is not returning any value and not even taking any parameters. The full control is given to sub function which reads two integer numbers, performs addition and then prints the sum. **2. Void function with parameters**

In this design, the function doesn‟t return any value but accepts one or more parameters so it is also called as function with no return value but with parameters. When function doesn‟t return value then return type must be void.

Consider the following example program which performs addition of two integer numbers with the help of function. #include<stdio.h> void addition(int x, int y); int main()

```
{
        int   a, b;
printf(“Enter a and b\n”);
scanf(“%d%d”, &a, &b);
addition(a,b);
        return 0;
}
void addition(int x, int y)
{
        int sum;
        sum = x + y;
        printf(“sum = %d\n”, sum);
}
```

In this program, the sub function addition() is not returning any value but taking two parameters x and y which receives copy of a and b respectively. It is a type of down-ward one-way

communication since, main function i.e calling function is sending data to sub function addition i.e. called function.

3. Non-Void function without parameters

In this design, the function returns a value but doesn‟t accept any parameters so it is also called as function with return value and no parameters.

When function returns a value then return type must be non-void.

Consider the following example program which performs addition of two integer numbers with the help of function. #include<stdio.h> int addition(void); int main()

```
{
        int   sum;
        sum = addition();
        printf(“sum = %d\n”, sum);
        return 0;
}
int addition(void)
{
        int sum, a, b;
printf(“Enter a and b\n”);
scanf(“%d%d”, &a, &b);
        sum   =   x   +   y;
return sum;
}
```

In this program, the sub function addition() is returning an integer value i.e. sum but not taking any parameters. It is a type of up-ward one-way communication since sub function i.e called function is sending data to main function i.e. calling function.

4. Non-Void function with parameters

In this design, the function returns a value and also accepts one or more parameters so it is also called as function with return value and with parameters.

When function returns a value then return type must be non-void.

Consider the following example program which performs addition of two integer numbers with the help of function. #include<stdio.h> int addition(int x, int y);

```
int main()
{
        int   a, b, sum;
printf(“Enter a and b\n”);
scanf(“%d%d”, &a, &b);
        sum = addition(a,b);
        printf(“sum = %d\n”, sum);
        return 0;
}
int addition(int x, int y)
{
        int sum;
```

```
    sum = x + y;
    return sum;
}
```

In this program, the sub function addition() is returning an integer value i.e. sum and also taking two parameters x and y which receives copy of a and b respectively. It is a type of two-way communication since, main function i.e calling function is sending data to sub function addition i.e. called function as well as called function is also returning a value to calling function.

VI. Recursion

Recursion is a process in which a function calls itself. This enables the function to call several times or repeatedly to solve the given problem. The function which is called recursively by itself is known as recursive function. Since recursion initiates repetition of same function there must exist a stopping condition.

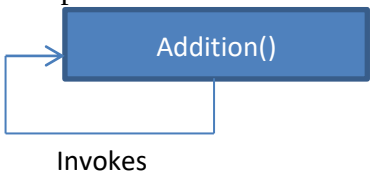
The various types of recursion are

a. Direct recursion
b. Indirect recursion

a. Direct recursion

A function that invokes itself is said to be direct recursion.

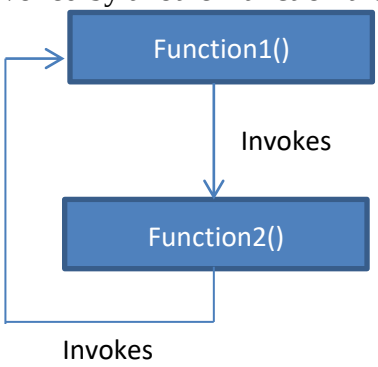
The following picture depicts the direct recursion



In this picture, a function known as Addition() is invoking Addition() recursively i.e., a function is called by itself hence direct recursion.

b. Indirect recursion

A function is recursively invoked by another function then it is known as indirect recursion.



In this picture, it illustrates that Function1 is invoking Function2 in turn Function2 is invoking Function1 hence it forms recursion. Here, Function1 is called recursively by Function2 and even Function2 is called recursively by Function1 hence, both Function1 and Function2 are indirect recursive functions.

Example: Program to find the factorial of a given number using recursion.

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

```
int     fact (int n)
{
        if( n ==0)
                return 1;
        return n * fact(n-1) ;
}       int
main()
{
        int N, result;    printf("Enter the value of
N\n");          scanf("%d", &N);       result =
fact(N);        printf("Factorial of %d = %d\n" , N,
result);        return 0;
}
```

VII. Multi-Dimensional Array

If an array contains more than one subscript then it is called as multi-dimensional array.

Two-Dimensional Arrays:

If an array contains two subscripts then it is called as two-dimensional array. The elements in two-dimensional array are arranged in matrix or table form.

Declaration and definition of Two-Dimensional array

An array must be declared and defined before it can be used. Array declaration and definition tells the compiler

- Name of the array
- The size or the number of elements in the array
- The type of each element of the array
- Total amount of memory to be allocated to the array

The size of the array is a constant and must have a value at compilation time.

The declaration format or syntax of declaration is given below;

Type array_name [ROWSIZE][COLSIZE] ;

Total number of elements in a two-dimensional array is calculated by multiplying ROWSIZE and COLSIZE.

i.e., No of elements = ROWSIZE * COLSIZE

Total Memory Allocated = ROWSIZE * COLSIZE * sizeof(Type)

Where,

Type - refers to the type of each element of the arrays which can be int, float, char, double, short, long, etc.

array_name - refers to the name of the array to be specified with the help of rules of identifier

ROWSIZE - refers to the size or number of rows of the array.

COLSIZE - refers to the size or number of columns of the array.

These **ROWSIZE and COLSIZE** can be a symbolic constants, integer constants or integer arithmetic expressions.

Examples:

a. declare and define an array by the name **matrix** which contains **3 rows and 4 columns of integer type.** int matrix[3][4];

b. declare and define an array called **string** which contains 5 strings each of 10 characters in size.

char string[5][10];

Accessing Elements in Two-dimensional Arrays

C uses an **index** to access individual elements in an array. The index must be an integral value or an expression that evaluates to an integral value. For example, consider the following array

int matrix[3][3];

Pictorial representation of two-dimensional matrix is given below

Column Index / Row Index	**0**	**1**	**2**
0	**matrix[0][0]**	**matrix[0][1]**	**matrix[0][2]**
1	**matrix[1][0]**	**matrix[1][1]**	**matrix[1][2]**
2	**matrix[2][0]**	**matrix[2][1]**	**matrix[2][2]**

To access **first element** of **first row** of the array **matrix**, we can use **matrix[0][0],** since the index of first row is ZERO (0) and first column is ZERO (0).

Similarly, **second element of first row** of the array can be accessed by matrix[0][1], **third element of second row** can be accessed by matrix[1][2], etc.

Storing Values in two-dimensional Arrays

Declaration and definition only reserve space for the elements in the array. No values are stored.

If we want to store values in the array, we can use the following different methods of initialization

- initialize the elements **(Initialization)**
- read values from the keyboard **(Inputting)**
- assign values to each individual element **(Assignment)**

Initialization of two-dimensional array

Providing a value for each element of the array is called as initialization. The list of values must be enclosed in curly braces.

NOTE: It is a compile time error if we provide more values than the size of the array.

Example 1: Initializing all elements. All elements are initialized sequentially.

int numbers[3][2] = { 3, 23, 17, 8, 19,7};

In this example, two-dimensional arrays **numbers** is initialized as given below

Column Index / Row Index	0	1
0	3	23
1	17	8
2	19	7

Example 2: Initializing all elements but each row is initialized independently.

```
int Array[3][3] = { { 2, 3, 4},
                    {12, 8, 6},
                    {6, 3, 16}
                  };
```

In this example, all elements are initialized row by row as given below.

Column Index / Row Index	0	1	2
0	2	3	4
1	12	8	6
2	6	3	16

Example 3: Partial initialization.

```
int a[3][3] = { { 1 }, { 5 , 2 }, { 6 }  };
```

In this example, only specified elements are initialized row by row and uninitialized elements are set to ZERO by default as given below.

Column Index / Row Index	0	1	2
0	1	0	0
1	5	2	0
2	6	0	0

Inputting Values

Another way to fill the array is to read the values from the keyboard or file by using scanf statement or fscanf statement. For Example,

```
int numbers[3][2];
int i, j ;                  for(i=0;i<3;i++)
for(j=0;j<2;j++)                        scanf("%d" ,
&numbers[i][j]);
```

In this example, scanf function is repetitively called for 3*2 = 6 times and reads the values for all elements of the array. Since the starting index of row and column is ZERO, so index **i** and **j** are initialized to 0 in for loop and the condition i<3 iterates 3 times i.e., the number of rows and the condition j<2 iterates for 2 times i.e., the number of columns.

Assigning values

We can assign values to individual elements using the assignment operator. Any value that reduces to the proper type of array can be assigned to an individual array element. For Example,

```
int numbers[3][3];
numbers[0][0] = 10;   //assigns first element of first row to 10
numbers[1][2] = 4;    //assigns third element of second row to 4
```

VIII. Using Arrays with Functions

To process arrays in a large program, we have to be able to pass them to functions. We can pass arrays in two ways:

a. Passing individual elements
b. Passing the entire array

a. Passing individual elements

As we know that there are two parameter passing mechanisms, we can pass individual elements by either data values or by passing their addresses. **The default method is pass-byvalue.**

Passing Data Values

We pass data values of the individual elements of the array. The actual parameters in function call are individual elements of the array whose values are copied to the formal parameters.
For example:

```
int   arr[10];            // creates an array of integer numbers
fun (arr[3]);     // invokes a function fun with 4th element of array as parameter

void fun(int x)                 //the value of 4th element of array is copied to
{                               // formal parameter x

        Process x;
}
```

Passing Addresses

We pass addresses of the individual elements of the array. The actual parameters in function call are individual elements of the array whose addresses are copied to the formal parameters.
For example:

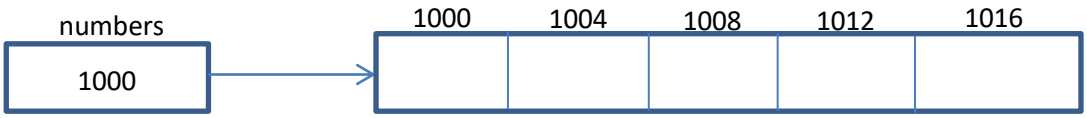
```
int   arr[10];            // creates an array of integer numbers
 fun (&arr[3]); // invokes a function fun with address of 4th element of array as
                // parameter

void fun(int *x)                 //the address of 4th element of array is copied to
{                               // formal parameter *x

        Process x;
}
```

b. Passing the entire array

Passing individual elements of the array is not a feasible method since it requires a lot of memory and time to process large array. For example, if an array consists of 20000 elements then if we pass individual elements to the function then another 20000 memory blocks are to be allocated in function to store the values of each element of the array. So instead of passing individual elements, starting address of whole array can be passed. Hence, for passing the entire array, **the default method of parameter passing mechanism is pass-by-address or reference.**

NOTE: In C, the name of the array is a primary expression whose value is the address of the first element in the array.

A pictorial representation of above NOTE is shown below. int numbers[5];



From the picture, we can understand that the name of array numbers is having the starting address i.e., 1000 in it. With the help of this address we can go through each and every element of the array. **Hence, to pass the entire array to the function, we can pass only name of the array** which contains starting address.

To pass the whole array, we simply use the array name as actual parameter in a function call. In function declaration or definition, we can declare a formal parameter in two different ways. First, it can use an array declaration with an empty set of brackets in the parameter list. Secondly, it can specify that the array parameter is a pointer.

Any modification in the function definition by using formal parameter will modify actual parameter since formal parameter is a reference to the actual parameter.

For example:
void fun (int num[ ],…); void fun (int *ptr, …); **Example Program: C program to generate N Fibonacci numbers**

Fibonacci numbers or series starts with either 0 and 1 or 1 and 1 then the next number in the series is sum of two previous terms.

Therefore,

Fibonacci number can be either 0, 1, 1, 2, 3, 5, 8, 13, 21, … or it can be 1, 1, 2, 3, 5, 8, 13, 21, 34, …

We will write a program for first series i.e., 0, 1, 1, 2, 3, 5, 8, 13, 21, …

```
#include<stdio.h> void
fibonacci(int *f, int N); int
main()
{
        int i, N, fib[100]; printf("Enter the number of
elements to be generated\n"); scanf("%d", &N);
        fibonacci(fib, N); // in this function call, starting address of fib array is copied to
                           // formal parameter *f which is pointer in declaration
        printf("The Fibonacci Numbers are\n");
        for(i=0;i<N;i++)
```

```
                printf("%d \t", fib[i]);          // prints the fibonacci numbers generated in
                                                  // function definition
        return 0;
}
void   fibonacci(int *f, int N)
{
        int   i;   f[0] = 0;
f[1]          =          1;
for(i=2;i<N;i++)
                f[i] = f[i-1] + f[i-2] ;
}
```

IX. Declaring, Initializing, Printing, and Reading Strings

Definition of String:

A C string is an array of characters terminated by null (\0) character. The \0 (null) character indicates the end of string. String in C is always variable in length. Following example illustrates how a string is stored in memory.

H	e	l	l	o	\0

What is important here is that the string "Hello" is stored in an array of characters that ends with null (\0) character.

NOTE 1: Empty string is represented by "" which requires one byte of memory.

NOTE 2: null (\0) character is known as delimiter.

Why null (\0) character?

String is not a data type but it is a data structure and its implementation is logical not physical. Since string length is variable in C, to identify the end of string this null character is used.

Definition of String Literal or constant

A string literal is also known as string constant, is a sequence of characters enclosed in a pair of double quotes. For example:

"Welcome to C programming"

"Hello" "abcd"

When string literals are used in a program, C automatically creates an array of characters, initializes it to a null-terminated string, and stores it and remembering its address.

Declaring a String:

Like other variables, the string has to be declared before it is used in the program. Since string is a sequence of characters, it is to be declared as array of characters.

For example:

If we want a string declaration for an eight-character string, including its delimiter then it must be declared as shown below char string[9];

Using this declaration, the compiler allocates 9 bytes of memory for the variable string.

Initializing a String:

We can initialize a string the same way that we initialize any other variable when it is defined.

Initialization of string can be done in different ways

1. **Initialization using string constant** char str[9] = “Good Day”; // in this case, the value is a string constant

2. **Initialization without size using string constant**

   We do not need to specify the size of the array if we initialize it when it is defined.

   char str[ ] = “January”;

3. **Initialization using array of characters**

   char str[9] = {„G, „o“, „o“, „d“, „ “, „D“, „a“, „y“, „\0“ };

NOTE 1: in this example, we must ensure that the null character is at the end of the string.

NOTE 2: Strings and Assignment Operator: One string cannot be assigned to another string using assignment operator. It is a compiler time error if we try to assign.

For example:

```
char str1[20] = “Hello”;
char str2[20];
str2 = str1 ;  // it“s a compiler time error: one string cannot be copied to another string using assignment operator.
```

Printing and Reading of Strings:

There are several functions to read the string from the keyboard or file and to display or print the string onto the monitor, printer or a file. They are discussed in the section **String Input and Output Functions.**

X. String Input and Output Functions

C programming treats all the devices as files. So devices such as the display or monitor are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	stdin	keyboard
Standard output	stdout	screen
Standard error	stderr	screen

The file pointers are the means to access the file for reading and writing purpose. This section explains how to read values from the keyboard and how to print the result on the screen.

C provides two basic ways to read and write strings.

1. **Formatted Input and Output Functions**: the functions which helps to read and write the data in the required format are known as formatted input and output functions

a. Formatted Input Functions: scanf() and fscanf()

b. Formatted Output Functions: printf() and fprintf()

Formatted Input Functions: scanf()

We read strings from the keyboard using the read-formatted function i.e. scanf().

The **conversion code** for a string is **„s‟**. Therefore, **format string or control string** to read a string is **“%s”**.

Scanf() do the following work when we enter string from the keyboard

A. First, it removes the whitespaces which appear before the string

B. Once it finds a character, then its starts reading characters until it finds a whitespace and stores in array in order.

C. Then it ends the string with a null character.

For example:

```
char     str[20];
scanf(“%s”, str);        // & operator must not be used since str contains starting
                         // address of string
```

If we enter the following string as input from keyboard, then it reads HELLO as string, terminates it with null character and stores in **str**. Therefore, str[20] will be “HELLO\0”.

Space	Space	Space	H	E	L	L	O	Space	W	O	R	L	D	Press Enter

In this example, the scanf() has skipped 3 leading white spaces and started reading from the character „H‟ until a white space. Remaining characters will not be read and left in buffer.

NOTE: The string conversion code skips whitespace.

Drawback of scanf():

The main drawback of this scanf() to read a string is that as soon as it encounters a white space it stops reading the string. Hence, it is not possible to read a string with multiple words separated by white space(sentence for example).

Formatted Output Functions: printf()

We print strings on monitor using the print-formatted function i.e. printf().

The **conversion code** for a string is **„s‟**. Therefore, **format string or control string** to print a string is **“%s”**. For example:

```
char     str[20] = “HELLO”;
printf(“%s”, str);        // prints the content of string str on scree i.e., HELLO
```

Example program to read and display a string using scanf and printf functions

```
#include<stdio.h> int
main()
{
 char  str[20];  printf(“Enter a
String\n”);
        scanf(“%s”, str);
```

```
    printf("String entered by You is : %s \n", str);
    return 0;
}
```

The program reads the string entered from the keyboard using scanf function and the same string will be displayed on screen using printf function.

2. **Unformatted Input and Output Functions**: the functions which reads and writes the data without any format are known as unformatted input and output functions
   a. Unformatted Input Functions: gets(), getchar(), getc(), and fgetc()
   b. Unformatted Output Functions: puts(), putchar(), putc(), and fputc()

Unformatted Input and output Functions: getchar() and putchar()

The **int getchar(void)** function reads the next available character from the keyboard and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen.

The **int putchar(int c)** function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the screen.

```c
#include <stdio.h>
int main( )
{
   int c;
   printf( "Enter a value :");
   c = getchar( );

   printf( "\nYou entered: ");
   putchar( c );

   return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows

./a.out

Enter a value : I am reading a character using getchar

You entered : I

From the output, we can understand that the getchar function reads only single character at a time i.e I and stores it in variable c. So only, putchar function prints value of variable c i.e. I on the screen.

Unformatted input and output functions: getc() and putc() getc()

The C library function **int getc(FILE *stream)** gets the next character (an unsigned char) from the specified stream and advances the position indicator for the stream.

Declaration

Following is the declaration for getc() function.

```
int getc(FILE *stream)
```

Parameters

- **stream** -- This is the pointer to a FILE object that identifies the stream on which the operation is to be performed. It should be **stdin** if we want to read data from keyboard.

Return Value

This function returns the character read as an unsigned char cast to an int or EOF on end of file or error.

putc()

The C library function **int putc(int ch, FILE *stream)** writes a character (an unsigned char) specified by the argument **ch** to the specified stream and advances the position indicator for the stream.

Declaration

Following is the declaration for putc() function.

```
int putc(int ch, FILE *stream)
```

Parameters

- **char** -- This is the character to be written. The character is passed as its int promotion.
- **stream** -- This is the pointer to a FILE object that identifies the stream where the character is to be written. It should be **stdout** to write on screen.

Return Value

This function returns the character written as an unsigned char cast to an int or EOF on error.

Example

The following example shows the usage of getc() and putc() function.

```
#include<stdio.h>
int main()
{    char
c;
   printf("Enter character: ");
c = getc(stdin);
   printf("Character entered: ");
   putc(c, stdout);
return(0);
}
```

Let us compile and run the above program that will produce the following result:

```
Enter character: A
Character entered: A
```

Unformatted Input Functions: gets() and puts()

The **char *gets(char *s)** function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (End of File). It makes null-terminated string after reading. Hence it is also known as **line-to-string** input function.

The **int puts(const char *s)** function writes the null-terminated string 's' from memory and a trailing newline to **stdout**. This function is also known as **string-to-line** output function.

```
#include <stdio.h> int main( )
{
  char str[100];    printf( "Enter a
value :");
  gets( str );

  printf( "\nYou entered: ");
puts( str );
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays it as follows

./a.out

Enter a value : I am reading a line

You entered : I am reading a line

From the output, we can understand that the gets function reads a line at a time i.e **I am reading a line** and stores it in variable str. So only, puts function prints value of variable str i.e. **I am reading a line** on the screen.

Program: C program to read a sentence and count the frequency of VOWELs and total count of consonants.

```
#include<stdio.h> int
main()
{
        char sentence[100];
        int vowelA=0, vowelE=0, vowelI=0, vowelO=0, vowelU=0, cons=0;
printf("Enter a sentence\n");  gets(sentence);          i=0;
        while((ch=sentence[i++])! = „\0“)
        {
                switch(ch)
                {
                        case „a“:
                        case „A“:
vowelA++;
break;                  case „e“:
                        case „E“: vowelE++;
                           break;
        case „i“:
                        case „I“: vowelI++;
                           break;
        case „o“:
                        case „O“:
vowelO++;
break;                  case „u“:
                        case „U“: vowelU++;
                                  break;
        default: cons++;
                }
        }
        printf("Frequency of Vowel A = %d\n" , vowelA);
printf("Frequency of Vowel E = %d\n" , vowelE);
printf("Frequency of Vowel I = %d\n" , vowelI);
printf("Frequency of Vowel O = %d\n" , vowelO);
printf("Frequency of Vowel U = %d\n" , vowelU);
printf("Total count of Consonants = %d\n" , cons);  return 0;
}
```

XI. String Manipulation Functions or String Handling Functions

Manipulation includes performing various operations on strings according to the need of a problem. Hence, in **C** programming language, there are a lot of functions that can be used to manipulate or to handle strings. The strings manipulations involve combining or concatenating strings, copying the content of one string to another string, comparing two strings, calculating the length of a string, finding a character or a substring in another string, and converting a string to long or double value that can be used for computations.

List of Built-In Functions available in C

a. strlen() - calculates the length of string
b. strcpy() - copies one string to another string
c. strncpy() - copies first N characters of one string to another string
d. strcat() - concatenates or joins two strings
e. strncat() - concatenates or joins first N characters of one string with another string
f. strcmp() - compares two strings
g. strncmp() - compares first N characters of two strings
h. strchr() - find first occurrence of a given character in string
i. strrchr() - find last occurrence of a given character in string
j. strstr() - find first occurrence of one string in another string
k. strtok() - parse or split the string into tokens using specified delimiters

NOTE: to use any of these functions in C program, one must include header file string.h since it contains all of these string handling functions.

a. strlen() - calculates the length of string

Definition:

The strlen function calculates the length, in bytes, of a given string. This calculation does not include the null character. The function returns length of string.

Syntax:

```
Variable = strlen(string);
```

Where string can be C string or it can be String literal or constant.

Example 1:

```
char str[ ] = "Hello";

int length;
length = strlen(str);
```

Output:

length=5 since, strlen() returns length of string i.e. 5 since the string constant "Hello" is having 5 characters.

Example 2:

```
int len; len = strlen("Hello
World");
```

Output:

len=11, since there are 11 characters in string constant “Hello World”

b. strcpy() - copies one string to another string Definition:

The strcpy function copies characters from source string to destination string up to and including the terminating null character. The strcpy function returns destination string.

Syntax:

strcpy(destination, source);

where,

destination - must be array of character type
source - can be either C string or string literal

Example 1:

```
char  src[20] = “Hello”;
char  dest[20];
strcpy(dest, src); // copies content of src to dest  printf(“dest = %s\n” , dest);
```

Output:

dest = Hello since strcpy copies content of src i.e “Hello” to dest. Therefore, dest will be “Hello”.

Example 2:

```
char  str[20];
strcpy(str, “Hello World”);  // copies string literal to str
printf(“str = %s\n” , str);
```

Output:

str = Hello World since strcpy copies string constant i.e “Hello World” to str. Therefore, str will be “Hello World”.

c. strncpy() - copies first N characters of one string to another string

Definition:

The strncpy function copies N characters from source string to destination string up to and including the terminating null character if length of source string is less than N. The function returns destination string.

Syntax: strcpy(destination, source, N);

where, destination - must be array of character type
source - can be either C string or string literal
N - must be an integer value

Example 1:

```
char  src[20] = “Hello”;
char  dest[20];
strcpy(dest, src, 2);  // copies first 2 characters of src to dest printf(“dest = %s\n” , dest);
```

Output:

dest = He since strncpy copies first 2 characters of src to dest. Therefore, dest

will be “He”.

Example 2:

```
char    str[20];
strcpy(str, “Hello World”, 7); // copies first 7 characters of string literal to str
printf(“str = %s\n” , str);
```

Output:

str = Hello W since strcpy copies first 7 characters of string constant to str. Therefore, str will be “Hello W”.

Example 3:

```
char    src[20] = “Hello”;
char    dest[20];
strcpy(dest, src, 20); // copies first 20 characters of src to dest printf(“dest =
%s\n” , dest);
```

Output:

dest = Hello since the length of string src is 5 which is less than 20 so it copies entire string in src to dest. Therefore, dest will be “Hello”.

d. strcat() - concatenates or joins two strings

Definition:

The strcat function concatenates or appends or joins source string with destination string. All characters from source string are copied including the terminating null character.

Syntax: strcat(destination, source);

where,

destination - must be an array
source - can be either C string or string literal

Example:

```
char    string1[20] = “Hello”;
char    string2[20] = “World”;
strcat(string1, string2);
printf(“String 1 =%s\n”, string1);
```

Output:

String 1 = HelloWorld since strcat concatenates content of string2 with string1. Therefore, string1 = “HelloWorld”.

e. strncat() - concatenates or joins first N characters of one string with another string

Definition:

The strncat function concatenates or appends or joins first N characters from source string to destination string. All characters from source string are copied including the terminating null character if length of source string is less than or equal to N.

Syntax: strcat(destination, source, N);

where,

destination - must be an array

source - can be either C string or string literal
N - must be an integer value

Example 1:

```
char string1[20] = "Hello";
char string2[20] = "World";
strcat(string1, string2, 3);
printf("String 1 =%s\n", string1);
```

Output:

String 1 = HelloWor since strcat concatenates first 3 characters of string2 with string1. Therefore, string1 = "HelloWor".

Example 2:

```
char string1[20] = "Hello";
char string2[20] = "World";
strcat(string1, string2, 20);
printf("String 1 =%s\n", string1);
```

Output:

String 1 = HelloWorld since length of string2 is less than 20 so entire source string is concatenated with string1. Therefore, string1 = "HelloWorld".

f. strcmp() - compares two strings

Definition:

The strcmp function compares the contents of string1 and string2 and returns an integer value indicating their relationship.

Syntax: strcmp(string1, string2)

where, string1 and string2 can be either C string or String constant

Return value:

- If string1 and string2 are equal then it returns ZERO
- If string1 is less than string2 then it returns negative value(<0)
- If string1 is greater than string2 then it returns positive value(>0)

Example 1:

```
char string1[20] = "Hello";
char string2[20] = "Hello";
n = strcmp(string1, string2);
```

Output:

n = 0 since string1 and string2 are equal

Example 2:

```
char string1[20] = "Hello";
char string2[20] = "World";
n = strcmp(string1, string2);
```

Output:

n = less than 0 since string1 is less than string2

Example 3:

char string1[20] = "Welcome"; char string2[20] = "Hi"; n = strcmp(string1, string2);

Output:

n = greater than 0 since string1 is greater than string2

g. strncmp() - compares first N characters of two strings Definition:

The strncmp function compares first N characters of string1 and string2 and returns an integer value indicating their relationship.

Syntax: strncmp(string1, string2, N)

where,

string1 and string2 can be either C string or String constant and N must be an integer value

Return value:

- If first N characters of string1 and string2 are equal then it returns ZERO
- If first N characters of string1 is less than string2 then it returns negative value(<0)
- If first N characters of string1 is greater than string2 then it returns positive value(>0)

Example 1:

char string1[20] = "Hello"; char string2[20] = "Hello"; X = strncmp(string1, string2, 3);

Output:

X = 0 since first 3 characters of string1 and string2 are equal **Example 2:**

char string1[20] = "Hello"; char string2[20] = "Horld"; X = strncmp(string1, string2, 2);

Output:

X = less than 0 since first 2 characters of string1 is less than string2

Example 3:

char string1[20] = "Welcome"; char string2[20] = "Hi"; X = strncmp(string1, string2,1);

Output:

n = greater than 0 since first 1 character of string1 is greater than string2

h. strchr() - find first occurrence of a given character in string Definition:

The strchr function searches string for the first occurrence of a specified character. The null character is also included in the search. The function returns a pointer to the first occurrence of given character in string or null pointer if no matching character is found.

Syntax: strchr(string, ch)

where,

string - can be C string or string constant

ch - character to be searched

Example:

```
char *s;
char buf [] = "This is a test";
s = strchr (buf, 't');
if (s != NULL)
    printf ("found a 't' at %s\n", s);
```

Output:

It will produce following result

found a „t‟ at test

i. strrchr() - find last occurrence of a given character in string Definition:

The strrchr function searches string for the last occurrence of a specified character. The null character is also included in the search. The function returns a pointer to the last occurrence of given character in string or null pointer if no matching character is found.

Syntax: strrchr(string, ch)

where,

string - can be C string or string constant ch - character to be searched **Example:**

```
char *s;
char buf [] = "This is testing";
s = strchr (buf, 't');
if (s != NULL)
   printf ("found a 't' at %s\n", s);
```

Output:

It will produce following result

found a „t‟ at ting

j. strstr() - find first occurrence of one string in another string Defintion:

The strstr function locates the first occurrence of the string2 in the string1 and returns a pointer to the beginning of the first occurrence in string1 if matching string2 is found otherwise null pointer.

Syntax: strstr(string1, string2);

where, string1 and string2 can be C string or string constant

Example:

```
char s1[ ] = "My House is small";
char *s2;
```

```
s2 = strstr(s1, "House");        printf
("Returned String : %s\n", s2);
```

Output:

It will produce the following result

Returned String : House is small

k. **strtok()** **- parse or split the string into tokens using specified delimiters**

Definition:

The strtok function is used to locate substrings known as tokens in a string. Its most common use is to split or parse the given string into tokens using one or more delimiters.

Syntax: strtok(string, delimiters)

where, string - can be either C string or string literal which is to be parsed

delimiters - one or more characters to be used to split or parse the given string

Example:

```
char| string[ ] = "ONE,TWO-THREE";
strtok(string, ",-");
```

In this example, the two delimiters specified are comma(,) and hypen(-) hence the given string "ONE,TWO-THREE" is parsed or split into three substrings namely "ONE" , "TWO" and "THREE".

Output:

"ONE"
"TWO"
"THREE""

XII. Array of Strings

As we know the definition of string is that it is an array of characters, then array of strings can be defined as array of array of characters. Hence, array of strings is basically an application of two-dimensional arrays. Each string is independent and at the same time they are grouped together through the array.

For example, if we need to store seven days of the week in the array then we have to arrange these as array of strings since we have to store seven days of the week and each day of the week is again an array of characters so it is two-dimensional array.

Declaration of array of strings

```
#define days 7 #define size 15
        char week[days][size];
```

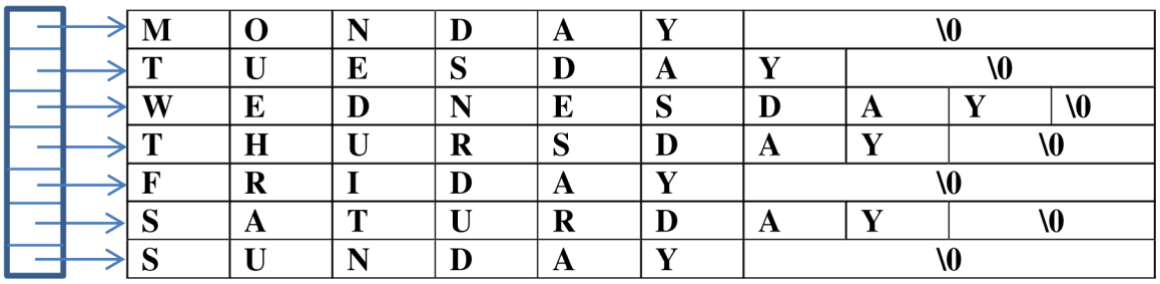
In this example, two-dimensional array called week is been created with seven days and each day can have maximum of 15 characters.

Initialization of array of strings a.

Initialization with SIZE

```
        #define days 7
#define size 15 char week[days][size] = {"Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday", "Sunday" };
```

Pictorial representation of array of string is shown below



M	O	N	D	A	Y	\0			
T	U	E	S	D	A	Y	\0		
W	E	D	N	E	S	D	A	Y	\0
T	H	U	R	S	D	A	Y	\0	
F	R	I	D	A	Y	\0			
S	A	T	U	R	D	A	Y	\0	
S	U	N	D	A	Y	\0			

b. Initialization without SIZE #define days 7

char week[days][ ] = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };

c. Inputting the values through keyboard

```
#define days 7
#define size 15
char week[days][size];
int i;
for(i=0;i<days;i++)
scanf("%s",week[i]);
```

Questions appeared in previous question papers

1. Write a C program to read N integers into an array A and find

i) the sum of odd numbers ii) the sum of even numbers iii) the average of all numbers

Output the results computed with appropriate headings.

Solution:

```
#include<stdio.h>          int
main( )
{
        int     i, a[100], osum=0, esum=0, N;
        float   average;
        printf("\n Enter the value of N:");
        scanf(%d",&N);
        printf("\n Enter %d integers \n",N);
        for(i=0;i<N;i++)
        scanf("%d", &a[i]);
        for(i=0;i<N;i++)
        {
                if(a[i]%2= =0)
                esum = esum +a[i];
                else
                        osum = osum +a[i];
        }
        average = (float)(esum+osum)/N;
```

```
        printf("\n Sum of even numbers = %d\n",esum);
        printf("\n Sum  of odd numbers = %d\n", osum);
        printf("\n The average of all numbers = %f\n",average);
        return 0;
    }
```

2. **Write a C Program to concatenate two strings without using built in function strcat().**

```
#include<stdio.h>  #include
<string.h>
void strconcat(char dest[],char src[])
{
    int len=strlen(dest),i=0;
    // start copying from the index len of dest  from src
    while (src[i]!= „\0“)
    {
        dest[len]=src[i];
        len++;
    i++;
    }
    dest[len]='\0';
}
int main(void)
{
    char dest[20]="ABCD",src[20]="EFG";
    strconcat(dest,src);
    printf("After concatinating the strings, destination string is %s\n",dest);
    return 0;
}
```

3. **Write a C Program to find length of a string without using built in function strlen().**

```
#include<stdio.h>
#include <string.h>
int strlength(char src[])
{
    int i=0;
    // start finding the length from first char to last char
    while (src[i]!= „\0“)
        i++;
    return i;
}
int main(void)
{
```

```
    char src[20];    printf(“Enter a string\n”);
    scanf(“%s”, src);        printf("Length of %s
string=%d\n", strlength(src));        return 0;
}
```

4. Write a C program to find cube of a number using function.

```
#include<stdio.h>   int cube(int n)
{
    return n*n*n;
}
int main(void)
{
    int n;
    printf("Enter the number whose cube has to be found\n");
    scanf("%d",&n);
    printf("The cube of the number %d is %d\n",n,cube(n));   return
    0;
}
```

5. Write a C program using recursion to print prime numbers between 1 and 100.

```
#include<stdio.h>   int isprime(int n, int p)
{
    if (p==1)
        return 1;
    else if (n%p==0)
        return 0;
     else
        return isprime(n,p-1);
}
int main(void)
{
    int i;
    printf("The prime numbers are :");
    for (i=2;i<100;i++)
        if (isprime(i, i/2)==1)
    printf("%d ",i);
    return 0;
}
```

6. Write a C Program to find the greatest number from a given one dimensional array.

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

```
    int   arr[20], i, large, n;
    printf("Enter number of
elements\n");
    scanf("%d", &n);
    printf("Enter %d elements\n", n);
    for(i=0; i<n; i++)
        scanf("%d",&arr[i]);
    printf("The largest element is: ");
    large = a[0];
    for(i=1; i<n; i++)
    {
        if(a[i] > large)
            large = a[i];
    }
    printf("%d", large);
    return 0;
}
```

7. Write a C function isprime(num) that accepts an integer argument and returns 1 if the argument is prime, a 0 otherwise. Write a C program that invokes this function to generate prime numbers between the given ranges. OR
Write a C Program to list prime numbers in the given range of numbers.
Solution:

```
#include<stdio.h>   int
isprime(int n, int p)
{
    if (p==1)
        return 1;
    else if (n%p==0)
        return 0;
     else
        return isprime(n,p-1);
}
int main(void)
{
    int i, num1, num2;
    printf("Enter range of numbers\n");
    scanf("%d%d", &num1, &num2);
    printf("The prime numbers are :");
    for (i=num1;i<=num2;i++)
        if (isprime(i,
i/2)==1)         printf("%d ",
i);         return 0;
```

```
}
```

8. Write a C program to evaluate polynomial f(x)= $a_4x^4+a_3x^3+a_2x^2+a_1x^1+a_0$.

Solution:

```
#include<stdio.h>  int
main()
{  int n, i, sum=0, a[10], x;  printf("\n
        Enter the value of n:");
        scanf("%d",&n);
        printf("\n Enter the %d co-efficient:\n",n+1);
        for(i=0;i<=n;i++)  scanf("%d",&a[i]);
        printf("\n Enter value of x :");
        scanf("%d",&x);  for(i=n;
        i>0; i--)
        sum=(sum+a[i])*x;
        sum = sum + a[0];
        printf("\n value of sum is :%d \n",sum);  return
        0;
}
```

9. Write a C program to read a matrix of size MxN and find the following I.

The sum of elements of each row

II. **The sum of elements of each column**

III. **Total sum of all elements of matrix**

```
#include<stdio.h>          int
main( )
{
        int     a[10][10], i, j , rsum, csum, tsum=0, m, n;
        printf("\n Enter the order of matrix:");
        scanf(%d%d",&m, &n);
        printf("\n Enter elements of matrix \n");
        for(i=0; i<m; i++)
        for(j=0; j<n; j++)
                scanf("%d", &a[i][j]);
        for(i=0; i<m; i++)
        {
                rsum=0;
        csum=0;
                for(j=0; j<n; j++)
                {
                        rsum = rsum +a[i][j];
                        csum = csum + a[j][i];
```

```
            }
            tsum = tsum + rsum
            printf("Sum of %d row = %d\n", i+1, rsum);
        printf("Sum of %d columns = %d\n", i+1, csum);
        }
        printf("Total sum of all elements = %d\n", tsum);
        return 0;
    }
```

10. Write a C program to find the largest element in a two dimensional array.

```
#include<stdio.h>          int
main( )
{
        int     i, j, a[10][10], large, m, n;
        printf("\n Enter the order of matrix:");
        scanf(%d%d",&m, &n);
        printf("\n Enter elements of matrix \n");
        for(i=0; i<m; i++)
        for(j=0; j<n; j++)
                        scanf("%d",         &a[i][j]);
        large = a[0][0];
        for(i=0; i<m; i++)
        {
                for(j=0; j<n; j++)
                {
                        if( a[i][j]>=large)
                                large = a[i][j];
                }
        }
        printf("Largest element in a matrix = %d\n", large);
        return 0;
    }
```

11. Write a C program to find the addition of two matrices a and b. Print the resultant matrix.

```
#include<stdio.h>          int
main( )
{
        int     i, j, a[10][10], b[10][10], c[10][10], m, n;
        printf("\n Enter the order of matrix a:");
        scanf(%d%d",&m, &n);
        printf("\n Enter elements of matrix A\n");
```

```
        for(i=0; i<m; i++)
        for(j=0; j<n; j++)
                scanf("%d", &a[i][j]);
        printf("\n Enter elements of matrix B\n");
        for(i=0; i<m; i++)          for(j=0;
j<n; j++)
  scanf("%d", &b[i][j]);  for(i=0; i<m; i++)
                for(j=0; j<n; j++)
                        c[i][j] = a[i][j] + b[i][j];
        printf("Resultant matrix C is:");
        for(i=0; i<m; i++)
        {               for(j=0; j<n; j++)
                        printf("%5d", c[i][j]);
                printf("\n");
        }
        return 0;
}
```

12. Write a C program that reads N integer numbers and arrange them in ascending order using Bubble Sort technique.

```
#include<stdio.h>  int
main()
{
        int n, i, j, a[10], temp;
 printf("Enter the no of elements:\n");
scanf("%d", &n);        printf("\n Enter %d
elements\n", n);
        for(i=0; i<n; i++)
scanf("%d",&a[i]);              printf("The
original elements are:\n");
        for(i=0; i<n; i++)
                printf("%d\t", a[i]);
        for(i=0; i<n-1; i++)
        {
                for(j=0; j<n-1-i; j++)
                {
                        if(a[i] >a[j])
                        {
                                temp = a[i];
                                a[i] = a[j];
                                a[j] = temp;
                        }
                }
        }
```

```
        printf("\n The Sorted array is
:\n");  for(i=0;i<n;i++)
printf("%d\n", a[i]);             return 0;
}
```

13. **Write a C program to search a Name in a list of names using Binary searching Technique.**

```
#include<stdio.h>  int
main()
{
        char    arr[25][20],    key[25];
int i, n, low, high, mid, cond;
 printf("Enter the number of strings you want to enter\n");     scanf("%d", &n);
        printf("Enter %d strings in alphabetical order\n", n);
        for (i=0; i<n; i++)
                scanf("%s", arr[i]);
        printf("Enter string to be searched\n");
        scanf("%s", key);
low = 0;        high = n-1;
while(low <= high)
{
                mid = (low + high) / 2;
                if((cond = strcmp(key, arr[mid])) == 0)
                {
                        printf("Key found at %d position\n", mid+1);
                        return 0;
                }
                else if(cond< 0)
                        high = mid - 1;
else
                        low = mid + 1;
        }
        printf("String  %s  is  not  found\n",  key);
return 0;
}
```

14. **Write a C program to search a number in a list of numbers using Binary searching Technique.**

```
#include<stdio.h>  int
main()
{
        int arr[25], key;       int   i,
n, low, high, mid, cond;
```

```
	printf("Enter the number of elements you want to enter\n");
scanf("%d", &n);	printf("Enter %d numbers\n", n);	for
(i=0; i<n; i++)
		scanf("%d", &arr[i]);
	printf("Enter the number to be searched\n");
	scanf("%d", &key);
	//first sort all the elements in ascending order using bubble sort
for(i=0; i<n-1; i++)
	{
		for(j=0; j<n-i-1; j++)
		{
			if(a[j]>a[j+1])
			{
				int temp = a[j];
			a[j] = a[j+1];
				a[j+1] = temp;
			}
		}
	}
	// search for a number using binary search
	low = 0;	high
= n-1;		while(low
<= high)
	{
		mid = (low + high) / 2;
		if(key	==	arr[mid])
	{
			printf("Key found at %d position\n", mid+1);
			return 0;
		}
		else if(key < arr[mid])
		high = mid - 1;
else
			low = mid + 1;
	}
	printf("key	%d	is	not	found\n",	key);
return 0;
}
```