

MODULE-4

Arrays: Declaration, initialization & access of one dimensional & two-dimensional arrays. Programs using one- and two-dimensional arrays- sorting and searching arrays. Handling of Strings: Declaring & initializing string variables, reading strings from terminal, writing strings to screen, Arithmetic operations on characters, String Handling functions, table of strings. User defined functions: Need for user defined functions, Declaring, defining and calling C functions return values & their types, Categories of functions: With/without arguments, with/without return values. Nesting of functions.

Arrays

- Array is the collection of similar data types or collection of similar entity stored in contiguous memory location. Array of character is a string.
- Each data item of an array is called an element. And each element is unique and located in separated memory location.
- Each of elements of an array share a variable but each element having different index number.
known as subscript.
- An array can be a single dimensional or multi-dimensional and number of subscripts determines its dimension.
- And number of subscripts is always starts with zero. One dimensional array is known as vector and two-dimensional arrays are known as matrix.

Declaration of an array:

We know that all the variables are declared before they are used in the program.

Similarly, an array must be declared before it is used.

During declaration, the size of the array has to be specified. The size used during declaration of the array informs the compiler to allocate and reserve the specified memory locations.

Syntax: -data_type array_name[n];

where, n is the number of data items (or) index(or) dimension.

0 to (n-1) is the range of array.

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

An array must be declared before being used. Declaring an array means specifying three things:

1. Data type: what kind of values it can store, for example int, char, float, double
2. Name: to identify the array
3. Size: the maximum number of values that the array can hold.

Arrays are declared using the following syntax: type name[size];

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

For example int marks[10];

[Memory representation of an array of 10 elements]

1 st Element	2 nd element	3 rd element	4 th element	5 th element	6 th element	7 th element	8 th element	9 th element	10 th element
----------------------------	----------------------------	----------------------------	----------------------------	----------------------------	----------------------------	----------------------------	----------------------------	----------------------------	-----------------------------

Marks[0] marks[1] marks[2] marks[3] marks[4] marks[5] marks[6] marks[7] marks[8] marks[9]

The above statement declares marks to be an array containing 10 elements. In C, the array index (also known as subscript) starts from zero. This means that the array marks will contain 10 elements in all. The first element will be stored in marks[0], the second element in marks[1], and so on. Therefore, the last element, i.e., the 10th element will be stored in marks[9]. Note that 0, 1, 2, 3 written within square brackets are subscripts/index.

Ex: inta[5]; float x[10];

The declaration of an array tells the compiler that, the data type, name of the array, size of the array and for each element it occupies memory space. Like for int data type, it occupies 2 bytes for each element and for float it occupies 4 byte for each element etc.

INITIALIZATION OF AN ARRAY:

Array can be made initialized at the time of declaration itself. The general form of array initialization is as below

type name[n] = [element1, element2, element n];

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

The elements 1, element2... element n are the values of the elements in the array referenced by the same.

Example1:- `int codes[5] = [12,13,14,15,16];`

Example2:- `float a[3] = [1.2, 1.3, 1.4];`

Example3:- `char name [5] = ['S', 'U', 'N', 'I', 'L'];`

In above examples, let us consider one, it a character array with 5 elements and all the five elements area initialized to 5 different consecutive memory locations.

`name[0] = 'S'`

`name[1] = 'U'`

`name[2] = 'N'`

`name[3] = 'I'`

`name[4] = 'L'`

Types of arrays:

1. one dimensional array
2. two-dimensional array
3. multi-dimensional array

1.One dimensional array:

An array with only one subscript is termed one-dimensional array or 1-D array. It is used to store a list of values, all of which share a common name (1-D array name) and are distinguishable by subscript value.

Declaration of one-dimensional array:

`data-type variable-name[size];`

example: `int a[5];`

`/*Write a program to input values into an array and display them*/`

`#include<stdio.h>`

`int main()`

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

```
int arr[5],i;

for(i=0;i<5;i++)

{

printf("enter a values of array \n",i);

scanf("%d\n",&arr[i]);

}

printf("the array elements are: \n");

for (i=0;i<5;i++)

{

printf("%d\t",arr[i]);

}

return 0;

}
```

OUTPUT:

12
45
59
98
21

The array elements are 12 45 59 98 21

while initializing a single dimensional array, it is optional to specify the size of array.

If the size is omitted during initialization then the compiler assumes the size of array equal to the number of initializers.

For example: -

```
int marks []={99,78,50,45,67,89};
```

If during the initialization of the number the initializers is less then size of array, then all the remaining elements of array are assigned value zero.

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

For example: `-int Marks[5]={99,78};`

Here the size of the array is 5 while there are only two initializers so After this initialization, the value of the rest elements are automatically occupied by zeros such as

`Marks[0]=99 , Marks[1]=78 , Marks[2]=0, Marks[3]=0, Marks[4]=0`

Again if we initialize an array like

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

Then the all the element of the array will be initialized to zero. If the number of initializers is more than the size given in brackets then the compiler will show an error.

Access the Elements of an Array

To access an array element, refer to its **index number**.

Array indexes start with **0**: [0] is the first element. [1] is the second element, etc.

This statement accesses the value of the **first element [0]** in myNumbers:

Example:

```
int myNumbers[]={25, 50, 75, 100};
printf("%d",myNumbers[0]);
```

// Output: 25

C Program to illustrate element access using array

```
#include <stdio.h>
int main()
{
    // array declaration and initialization
    int arr[5] = { 15, 25, 35, 45, 55 };
    // accessing element at index 2 i.e 3rd element
    printf("Element at arr[2]: %d\n", arr[2]);
    // accessing element at index 4 i.e last element
    printf("Element at arr[4]: %d\n", arr[4]);
    // accessing element at index 0 i.e first element
    printf("Element at arr[0]: %d", arr[0]);
    return 0;
}
```

Output

```
Element at arr[2]: 35
Element at arr[4]: 55
Element at arr[0]: 15
```

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

Two dimensional arrays

Two-dimensional array is known as matrix. The array declaration in both the array i.e. in single dimensional array single subscript is used and in two-dimensional array two subscripts are used.

Its syntax is: Data-type array name[row][column];

Or we can say 2-d array is a collection of 1-D array placed one below the other.

Total no. of elements in 2-D array is calculated as **row*column**

Example: -int a[2][3];

Total no of elements=row*column is $2*3=6$

It means the matrix consist of 2 rows and 3 columns

For example: -

20 2 7

8 3 15

Positions of 2-D array elements in an array are as below

00 01 02

10 11 12

Position

Elements

Memory allocations

a [0][0]	a [0][0]	a [0][0]	a [0][0]	a [0][0]	a [0][0]
20	2	7	8	3	15
2000	2002	2004	2006	2008	2010

Accessing 2-d array /processing 2-d arrays

For processing 2-d array, we use two nested for loops. The outer for loop corresponds to the row and the inner for loop corresponds to the column.

For example:

int a [4][5];

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

for reading value:-

```
for(i=0;i<4;i++)  
{  
for(j=0;j<5;j++)  
{  
scanf("%d",&a[i][j]);  
}  
}
```

For displaying value: -

```
for(i=0;i<4;i++)  
{  
  
for(j=0;j<5;j++)  
{  
printf("%d",a[i][j]);  
}  
}
```

Initialization of 2-d array:

2-D array can be initialized in a way similar to that of 1-D array. for example: -

```
int mat[4][3]={ 11,12,13,14,15,16,17,18,19,20,21,22};
```

These values are assigned to the elements row wise, so the values of elements after this initialization are

Mat[0][0]=11, Mat[1][0]=14, Mat[2][0]=17 Mat[3][0]=20

Mat[0][1]=12, Mat[1][1]=15, Mat[2][1]=18 Mat[3][1]=21

Mat[0][2]=13, Mat[1][2]=16, Mat[2][2]=19 Mat[3][2]=22

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

While initializing we can group the elements row wise using inner braces.

for example:-

```
int mat[4][3]={ { 11,12,13},{ 14,15,16},{ 17,18,19},{ 20,21,22 } };
```

And while initializing, it is necessary to mention the 2nd dimension where 1st dimension is optional.

```
int mat[][3];
```

```
int mat[2][3];
```

```
int mat[][];
```

```
int mat[2][];
```

invalid

We can also give the size of the 2-D array by using symbolic constant. Such as

```
#define ROW 2;
```

```
#define COLUMN 3;
```

```
int mat[ROW][COLUMN];
```

Example program of 2D array

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

```
int main(){
```

```
int i=0,j=0;
```

```
int arr[4][3]={ { 1,2,3},{ 2,3,4},{ 3,4,5},{ 4,5,6 } };
```

```
//traversing 2D array
```

```
for(i=0;i<4;i++){
```

```
    for(j=0;j<3;j++){
```

```
        printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
```

```
    }//end of j
```

```
}//end of i
```

```
return 0;
```

```
}
```

Output:

```
arr[0][0] = 1
```

```
arr[0][1] = 2
```


PROBLEM SOLVING USING C PROGRAMMING (BCA103)

arr[0][2] = 3

arr[1][0] = 2

arr[1][1] = 3

arr[1][2] = 4

arr[2][0] = 3

arr[2][1] = 4

arr[2][2] = 5

arr[3][0] = 4

arr[3][1] = 5

arr[3][2] = 6

Searching:

This is the process by which one searches the group of elements for the desired element. There are different methods of searching but let us deal two popular methods of searching and they are linear search and binary search.

Linear Search:

This is one of the simplest techniques for searching an unordered table for a particular element. In this each and every entry in the table is checked in a sequential manner until the desired element is found.

Program :

1. C program to search given number using linear search technique

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[10], i, item,n;
    clrscr();
    printf("enter number of elements in
    array\n");   scanf("%d",&n);
    printf("\nEnter elements of an array:\n");
    for (i=0; i<n; i++)
```

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

```
scanf("%d", &a[i]);
printf("\nEnter item to search: ");
scanf("%d", &item);
for (i=0; i<n; i++)
if (item == a[i])
{

printf("\nItem found at location %d", i+1);
break;
}
if(i==n)
printf("\nItem does not exist.");
getch();
}
```

Binary Search:

In binary search the basic requirement is the elements of the array should have been sorted alphabetically or numerically in the ascending order.

- In this technique the approximate middle entry of the array is located, and its key value is examined.
- If its value is too high, then the key value of the middle entry of the first half of the table is examined and the procedure is repeated on the first half until the required element is found or the search interval becomes empty.
- If the value is too low then the key of the middle entry of the second half of the array is tried and the procedure is repeated on the second half.
- The procedure continues until the desired key is found or the search interval becomes empty.

If searching for 23 in the 10-element array:

	2	5	8	12	16	23	38	56	72	91
23 > 16, take 2 nd half	L								H	
	2	5	8	12	16	23	38	56	72	91
23 < 56, take 1 st half						L				H
	2	5	8	12	16	23	38	56	72	91
Found 23, Return 5						L	H			
	2	5	8	12	16	23	38	56	72	91

Sorting:

When you rearrange data and put it into a certain order, you are sorting the data. You can sort data alphabetically, numerically, and in other ways.

Often you need to sort data before you use searching algorithms to find a particular piece of data.

There are many different algorithms that can be used to sort data.

A few popular ones are listed below:

1. Bubble sort
2. Selection sort
3. Insertion sort

1. Bubble sort:

The idea of bubble sort is to compare two adjacent elements. If they are not in the right order, switch them.

Do this comparing and switching (if necessary) until the end of the array is reached. Repeat this process from the beginning of the array n times.

Bubble Sort Example :

Here we want to sort an array containing [8, 5, 1].

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

8, 5, 1 Switch 8 and 5
5, 8, 1 Switch 8 and 1
5, 1, 8 Reached end start again.
5, 1, 8 Switch 5 and 1
1, 5, 8 No Switch for 5 and 8
1, 5, 8 Reached end

C program for sorting given set of numbers using bubble sort technique.

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

void main()
{
int array[10];
inti, j, num, temp;
clrscr();
printf("Enter the value of num \n");
scanf("%d", &num);
printf("Enter the elements one by one \n");
for (i = 0; i<num; i++)
{
scanf("%d", &array[i]);
}
for (i = 0; i<num; i++)
{
for (j = 0; j < (num - i - 1); j++)
{
if (array[j] > array[j + 1])
{
temp = array[j];
array[j] = array[j + 1];
array[j + 1] = temp;
}
}
}
```

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

```

}
printf("Sorted array is...\n");
for (i = 0; i < num; i++)
{
printf("%d\n", array[i]);
}
getch();
}

```

2.Selection sort

Selection sort is to repetitively pick up the smallest element and put it into the right position:

- Find the smallest element, and put it to the first position.
- Find the next smallest element, and put it to the second position.
- Repeat until all elements are in the right positions.

A loop through the array finds the smallest element easily.

After the smallest element is put in the first position, it is fixed and then we can deal with the rest of the array.

Selection Sort Example Here we are sorting an array containing the following numbers:

8, 27, 33, 2, 20, 12, 19, 5

8, 27, 33, **2**, 20, 12, 19, 5

2, 27, 33, 8, 20, 12, 19, **5**

2, 5, 33, **8**, 20, 12, 19, 27

2, 5, 8, 33, 20, **12**, 19, 27

2, 5, 8, 12, 20, 33, **19**, 27

2, 5, 8, 12, 19, 33, **20**, 27

2, 5, 8, 12, 19, 20, 33, **27**

2, 5, 8, 12, 19, 20, 27, 33

2, 5, 8, 12, 19, 20, 27, 33 Resulting sorted array

3.Insertion sort

Insertion sort maintains a sorted sub-array, and repetitively inserts new elements into it. The process is as following:

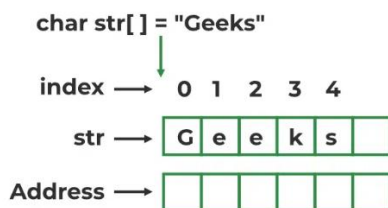
PROBLEM SOLVING USING C PROGRAMMING (BCA103)

- Take the first element as a sorted sub-array.
- Insert the second element into the sorted sub-array (shift elements if needed).
- Insert the third element into the sorted sub-array.
- Repeat until all elements are inserted.

STRINGS

A String in C programming is a sequence of characters terminated with a null character '\0'. The C String is stored as an array of characters. The difference between a character array and a C string is that the string in C is terminated with a unique character '\0'.

String in C



C String Declaration Syntax

Declaring a string in C is as simple as declaring a one-dimensional array. Below is the basic syntax for declaring a string.

```
char string_name[size];
```

In the above syntax **string_name** is any name given to the string variable and size is used to define the length of the string, i.e the number of characters strings will store.

There is an extra terminating character which is the Null character ('\0') used to indicate the termination of a string that differs strings from normal character arrays.

We can initialize the string as: `char name[]={ 'j', 'o', 'h', 'n', '\0' };`

Here each character occupies 1 byte of memory and last character is always NULL character.

Where '\0' and 0 (zero) are not same, where **ASCII** value of '\0' is 0 and ASCII value of 0 is 48. Array elements of character array are also stored in contiguous memory allocation. From the above we can represent as;

A String in C programming is a sequence of characters terminated with a null character '\0'. The C String is stored as an array of characters. The difference between a character array and a C string is that the string in C is terminated with a unique character '\0'.

j	o	h	n	\0
---	---	---	---	----

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

The terminating NULL is important because it is only the way that the function that work with string can know, where string end. String can also be **initialized** as;

```
char name[]="John";
```

Here the NULL character is not necessary and the compiler will assume it automatically.

String library function

There are several string library functions used to manipulate string and the prototypes for these functions are in header file "string.h".

strlen()

This function return the length of the string. i.e. the number of characters in the string excluding the terminating NULL character.

It accepts a single argument which is pointer to the first character of the string.

For example:

```
strlen("suresh");
```

It return the value 6.

Example:-

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

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

```
void main()
```

```
{
```

```
charstr[50];
```

```
print("Enter a string:");
```

```
gets(str);
```

```
printf("Length of the string is%d\n",strlen(str));
```

```
}
```

Output:

Enter a string: C in Depth

Length of the string is 8

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

strcmp()

This function is used to compare two strings.

If the two string match, strcmp() return a value 0 otherwise it return a non-zero value.

It compare the strings character by character and the comparison stops when the end of the string is reached or the corresponding characters in the two string are not same.

strcmp(s1,s2)

return a value: <0

when s1<s2

=0 when s1=s2

>0 when s1>s2

The exact value returned in case of dissimilar strings is not defined. We only know that if s1<s2 then a negative value will be returned and if s1>s2 then a positive value will be returned.

Example:

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

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

```
int main() {
```

```
    char s1[] = "Apple";
```

```
    char s2[] = "Applet";
```

```
        // Compare two strings and print result
```

```
    int res = strcmp(s1, s2);
```

```
    if (res == 0) printf("s1 and s2 are same");
```

```
    else if (res < 0)
```

```
        printf("s1 is lexicographically smaller than s2");
```

```
    else
```

```
        printf("s1 is lexicographically greater than s2");
```

```
    return 0;
```

```
}
```

Output

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

s1 is lexicographically greater than s2

strcpy()

- This function is used to copying one string to another string.
- The function strcpy(str1,str2) copies str2 to str1 including the NULL character.
- Here str2 is the source string and str1 is the destination string.
- The old content of the destination string str1 are lost. The function returns a pointer to destination string str1.

Example:-

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

void main()
{
    char str1[10],str2[10];
    printf("Enter a string:");
    scanf("%s",str2); strcpy(str1,str2);
    printf("First string:%s\t\tSecond string:%s\n",str1,str2);
    strcpy(str,"Delhi");
    strcpy(str2,"Bangalore");
    printf("First string :%s\t\tSecond string:%s",str1,str2);
}
```

strcat()

- This function is used to append a copy of a string at the end of the other string.
- If the first string is ""Purva" and second string is "Belmont" then after using this function the string becomes "PusvaBelmont".
- The NULL character from str1 is moved and str2 is added at the end of str1. The 2nd string str2 remains unaffected

Example:

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

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

```
int main() {  
    char s1[30] = "Hello, ";  
    char s2[] = "world!";  
  
    // Appends "world!" to "Hello, "  
    strcat(s1, s2);  
    printf("%s", s1);  
    return 0;  
}
```

Output**Hello, world****Reading Strings****1. Strings can be read using scanf() by writing**

```
scanf("%s", str);
```

Although the syntax of scanf() function is well known and easy to use, the main pitfall with this function is that the function terminates as soon as it finds a blank space. For example, if the user enters Hello World, then str will contain only Hello. This is because the moment a blank space is encountered, the string is terminated by the scanf() function. We may also specify a field width to indicate the maximum number of characters that can be read in. Remember that extra characters are left unconsumed in the input buffer.

2. The next method of reading a string is by using gets() function.

The string can be read by writing gets(str); gets() is a simple function that overcomes the drawbacks of the scanf() function. The gets() function takes the starting address of the string which will hold the input. The string inputted using gets() is automatically terminated with a null character.

3. strings can also be read by calling the getchar() function repeatedly to read a sequence of single characters (unless a terminating character is entered) and simultaneously storing it in a character array as shown below.

```
i=0;  
ch = getchar(); // Get a character  
while(ch != '\n')  
{  
    str[i] = ch; // Store the read character in str  
    i++;
```

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

```
ch = getchar(); // Get another character
}
str[i] = '\0'; // terminate str with null character
```

Writing Strings

Strings can be displayed on screen using three ways:

1. using printf() function
2. using puts() function
3. using putchar() function repeatedly

1.A string can be displayed using printf() by writing

printf("%s", str); We use the conversion character's to output a string. We may also use width and precision specifications along with %s. The width specifies the minimum output field width. If the string is short, extra space is either left padded or right padded. A negative width left pads short string rather than the default right justification. The precision specifies the maximum number of characters to be displayed. If the string is long, the extra characters are truncated.

example,

```
printf("%5.3s", str);
```

The above statement would print only the first three characters in a total field of five characters. Also these three characters are right justified in the allocated width. To make the string left justified, we must use a minus sign.

For example,

```
printf("%-5.3s", str);
```

2.The next method of writing a string is by using puts() function.

The string can be displayed by writing puts(str); puts() is a simple function that overcomes the drawbacks of the printf() function. The puts() function writes a line of output on the screen. It terminates the line with a newline character ('\n'). It returns an EOF (-1) if an error occurs and returns a positive number on success.

3.strings can also be written by calling the putchar() function repeatedly to print a sequence of single characters.

```
i=0;
while(str[i] != '\0')
{
    putchar(str[i]); // Print the character on the screen
    i++;
}
```

Example program of reading writing string

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

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

```
int main()
{
char str[] = "Introduction to C"; // Correct usage of printf with format specifiers
printf("\n%s", str);              // Prints the full string
printf("\n%20s", str);            // Right aligns the string in a field of width 20
printf("\n%-20s", str);          // Left aligns the string in a field of width 20
printf("\n%.4s", str);           // Prints the first 4 characters of the string
printf("\n%20.4s", str);         // Right aligns the first 4 characters in a field of width 20
printf("\n%-20.4s", str);        // Left aligns the first 4 characters in a field of width 20
return 0;
}
```

Output:

```
|Introduction to C|
|  Introduction to C|
|Introduction to C  |
|Intr|
|              Intr|
|Intr |
```

Arithmetic operations on characters

C allows us to manipulate characters the same way we do with numbers. Whenever a character constant or character variable is used in an expression, it is automatically converted into an integer value by the system. The integer value depends on the local character set of the system.

Character arithmetic is used to implement arithmetic operations like addition, subtraction, multiplication, and division on characters in C language.

In character arithmetic character converts into an integer value to perform the task. For this ASCII value is used.

It is used to perform actions on the strings.

C program to demonstrate character arithmetic.

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

```
int main()
```

```
{
```

```
    char ch1 = 125, ch2 = 10;
```

```
    ch1 = ch1 + ch2;
```

```
    printf("%d\n", ch1);
```

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

```
printf("%c\n", ch1 - ch2 - 4);
```

```
return 0;
```

```
}
```

Output

-121

Y

- $ch1 = 125 + 10;$
- $ch1$ now becomes 135. However, since char can typically hold values only within the range -128 to 127, **integer overflow** occurs
- $135 - 256 = -121$ (wrapping around using modular arithmetic).
- Internally, this value is treated as -121, but when using %d for printing, it interprets it as 121 (as it maps to the unsigned equivalent)

User defined functions:

The user defined functions defined by the user according to its requirement. Built-in function can't be modified, it can only read and can be used.

Syntax: -

Return type name of function (type 1 arg 1, type2 arg2, type3 arg3)

The **return_type** can be any valid C data type, and the **function_name** can be any valid identifier. The parameter list specifies the arguments that the function takes as input, and the body of the function contains the statements that are executed when the function is called.

Example program:

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

```
int add(int a, int b)
```

```
{
```

```
    int sum = a + b;
```

```
    return sum;
```

```
}
```

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

```
int main()
{
    int num1 = 5, num2 = 10, result;

    result = add(num1, num2);

    printf("The sum of %d and %d is %d\n", num1, num2, result);

    return 0;
}
```

Output:

The sum of 5 and 10 is 15

- In this example, we have defined a function called **add** that takes two integers as input and returns their sum. In the main function, we have declared two variables **num1** and **num2** and assigned them values **5** and **10** respectively. After that, we have called the **add function** with **num1** and **num2** as arguments and stored the result in a variable called **result**. Finally, we have printed the result using the **printf** function.

Needs for user defined functions

Code Reusability

- Functions allow you to write code once and reuse it multiple times without rewriting the logic.

Improved Code Organization

- Dividing a program into smaller, modular functions improves readability and makes it easier to maintain.
- Functions logically group related operations, such as input processing, calculations, and output.

Simplifies Debugging and Testing

- Functions isolate parts of the program, making it easier to test and debug specific functionality.

Reduces Redundancy

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

- By reusing functions, you eliminate the need for redundant code, saving development time and reducing errors.
- **Enhances Maintainability**
- Updating a single function updates its behavior across the program, reducing the effort needed for changes or improvements.
- **Encourages Modularity**
- Functions make programs modular, meaning different team members can work on individual functions independently.

The user-defined function in C can be divided into three parts:

1. Function Prototype
2. Function Definition
3. Function Call

1) Function Prototype

A function prototype is also known as a function declaration which specifies the **function's name, function parameters, and return type**. The function prototype does not contain the body of the function. It is basically used to inform the compiler about the existence of the user-defined function which can be used in the later part of the program.

Syntax:

```
return_type function_name (type1 arg1, type2 arg2, ... typeN argN);
```

2) Function Definition

- Function definition consists of the whole description and code of the function.
- It tells about what function is doing what are its inputs and what are its output.

It consists of two parts:

- function header (function name, function type and list of parameters)
- function body. (local variable declarations, functions statements and return statement)

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

Syntax: - return type function(type 1 arg1, type2 arg2, type3 arg3) /*function header*/

{

Local variable declaration;

Statement 1;

Statement 2;

Return value

}

The first line function_type function_name(parameter list) is known as the function header and the statements within the opening and closing braces constitute the function body.

3) Function Call

When the function gets called by the calling function then that is called, function call.

The compiler executes these functions when the semicolon is followed by the function name.

Example: -

function (arg1, arg2, arg3);

The argument that are used inside the function call are called actual argument

Ex: - int S=sum (a, b); //actual arguments

The function call statement invokes the function. When a function is invoked the compiler jumps to the called function to execute the statements that are part of that function. Once the called function is executed, the program control passes back to the calling function. Function call statement has the following syntax: function_name(variable1, variable2, ...);

/ C Program to illustrate the use of user-defined function

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

```
// Function prototype
```

```
int sum(int, int);
```

```
// Function definition
```

```
int sum(int x, int y)
```

```
{
```

```
    int sum;
```


PROBLEM SOLVING USING C PROGRAMMING (BCA103)

```
    sum = x + y;

    return x + y;

}

int main()

{

    int x = 10, y = 11;

    // Function call

    int result = sum(x, y);

    printf("Sum of %d and %d = %d ", x, y, result);

    return 0;

}
```

Output

Sum of 10 and 11 = 21

Components of Function Definition

There are three components of the function definition:

1. Function Parameters
2. Function Body
3. Return Value

1. Function Parameters

Function parameters (also known as arguments) are the values that are passed to the called function by the caller. We can pass none or any number of function parameters to the function.

We have to define the function name and its type in the function definition and we can only pass the same number and type of parameters in the function call.

Example

```
int foo (int a, int b);
```

Here, **a** and **b** are function parameters.

2. Function Body

The function body is the set of statements that are enclosed within { } braces. They are the statements that are executed when the function is called.

Example

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

Here, the statements between { and } is function body.

3. Return Value

The return value is the value returned by the function to its caller. A function can only return a single value and it is optional. If no value is to be returned, the return type is defined as void.

The **return keyword** is used to return the value from a function.

Syntax

```
return (expression);
```

Example

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

Passing Parameters to User-Defined Functions

We can pass parameters to a function in C using two methods:

1. Call by Value
2. Call by Reference

1. Call by value

In call by value, a copy of the value is passed to the function and changes that are made to the function are not reflected to the values. Actual and formal arguments are created in different memory locations.

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

Example

C program to show use of

// call by value

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

```
void swap(int a, int b)
```

```
{
```

```
    int temp = a;
```

```
    a = b;
```

```
    b = temp;
```

```
}
```

```
int main()
```

```
{
```

```
    int x = 10, y = 20;
```

```
    printf("Values of x and y before swap are: %d, %d\n", x,y);
```

```
    swap(x, y);
```

```
    printf("Values of x and y after swap are: %d, %d", x,y);
```

```
    return 0;
```

```
}
```

Output

Values of x and y before swap are: 10, 20

Values of x and y after swap are: 10, 20

Call by Reference

In a call by Reference, the address of the argument is passed to the function, and changes that are made to the function are reflected back to the values. We use the [pointers](#) of the required type to receive the address in the function.

Example

// C program to implement

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

// Call by Reference

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

```
void swap(int* a, int* b)
```

```
{
```

```
    int temp = *a;
```

```
    *a = *b;
```

```
    *b = temp;
```

```
}
```

```
int main()
```

```
{
```

```
    int x = 10, y = 20;
```

```
    printf("Values of x and y before swap are: %d, %d\n", x, y);
```

```
    swap(&x, &y);
```

```
    printf("Values of x and y after swap are: %d, %d", x,y);
```

```
    return 0;
```

```
}
```

Output

Values of x and y before swap are: 10, 20

Values of x and y after swap are: 20, 10

Category of Function based on argument and return type

- i) Function with no argument & no return value
- ii) Function with no argument but return value
- iii) function with argument but no return value
- iv) function with argument and return value

i) Function with no argument & no return value

Function that have no argument and no return value is written as:-

```
// void return type with no arguments
```

```
void function_name()
```

```
{ // no return value
```

```
return;
```

```
}
```

Calling function	Analysis	Called function
<pre>main () { --- --- fun (); }</pre>	<p>No arguments are passed</p> <p>No values are sent back</p>	<pre>fun () { --- --- }</pre>

Example:

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

```
void sum()
```

```
{
```

```
    int x, y;
```

```
    printf("Enter x and y\n");
```

```
    scanf("%d %d", &x, &y);
```

```
    printf("Sum of %d and %d is: %d", x, y, x + y);
```

```
}
```

```
int main()
```

```
{
```

```
    // function call
```

```
    sum();
```

PROBLEM SOLVING USING C PROGRAMMING (BCA103)

```
return 0;
```

Output

```
Enter x and y 4 5
```

```
Sum of x and y is 9
```

2) Function with no argument but return value

These functions take arguments (input) but do not return any value.

```
return_type function_name()
```

```
{ // program
```

```
return value;
```

```
}
```

Calling function	Analysis	Called function
<pre>main (.) { int c; --- c= fun (.); ---- ---- }</pre>	<p>No arguments are passed</p> <p>values are sent back</p>	<pre>fun (.) { ---- ---- return c; }</pre>

Example:

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

```
int sum()
```

```
{
```

```
    int x, y, s = 0;
```

```
    printf("Enter x and y\n");
```

```
    scanf("%d %d", &x, &y);
```

```
    s = x + y;
```

```

return s;

}

int main()
{
    // function call

    printf("Sum of x and y is %d", sum());

    return 0;
}

```

3) Function with argument but no return value

Functions that have arguments but no return values. Such functions are used to display or perform some operations on given arguments.

Syntax:

```
void function_name(type1 argument1, type2 argument2,...typeN argumentN)
```

```
{ // Program
```

```
return;
```

```
}
```

Calling function	Analysis	Called function
<pre>main () { --- --- fun (a, b); --- --- }</pre>	<p>Arguments are passed</p> <p>No values are sent back</p>	<pre>fun (int a, int b) { --- --- }</pre>

Example:

```
include <stdio.h>

void sum(int x, int y)
{
    printf("Sum of %d and %d is: %d", x, y, x + y);
}

int main()
{
    int x, y;

    printf("Enter x and y\n");

    scanf("%d %d", &x, &y);

    // function call

    sum(x, y);

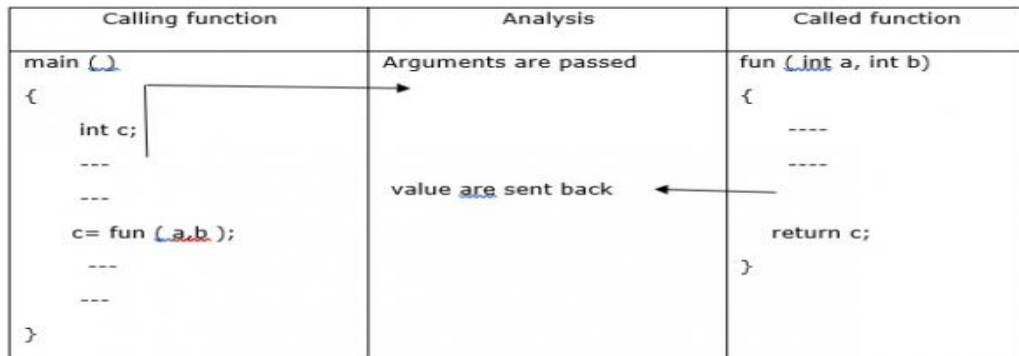
    return 0;
}
```

4) Function with argument and return value.

Functions that have arguments and some return value. These functions are used to perform specific operations on the given arguments and return their values to the user.

Syntax

```
return_type function_name(type1 argument1, type2 argument2,...typeN argumentN)
{ // program
return value;
}
```

Example:

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

```
int sum(int x, int y)
```

```
{
```

```
return x + y;
```

```
}
```

```
int main()
```

```
{
```

```
int x, y;
```

```
printf("Enter x and y\n");
```

```
scanf("%d %d", &x, &y);
```

```
// function call
```

```
printf("Sum of %d and %d is: %d", x, y, sum(x, y));
```

```
return 0;
```

```
}
```

Nesting of function

Some programmer thinks that defining a function inside another function is known as “nested function”. But the reality is that it is not a nested function, it is treated as lexical scoping. Lexical scoping is not valid in C because the compiler can’t reach/find the correct memory location of the inner function.

Nested function **is not supported** by C because we cannot define a function within another function in C. We can declare a function inside a function, but it’s not a nested function. Because nested functions definitions cannot [access local variables](#) of the surrounding blocks, they can access only global variables of the containing module. This is done so that lookup of global variables doesn’t have to go through the directory. As in C, there are two nested scopes: local and global (and beyond this, built-ins). Therefore, nested functions have only a limited use. If we try to approach nested function in C, then we will get compile time error.

// C program to illustrate the

// concept of Nested function.

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

```
int main(void)
```

```
{
```

```
    printf("Main");
```

```
    int fun()
```

```
    {
```

```
        printf("fun");
```

```
        // defining view() function inside fun() function.
```

```
        int view()
```

```
        {
```

```
            printf("view");
```

```
        }
```

```
    return 1;
```

```
}  
view();  
}
```

Output:

Compile time error: undefined reference to `view'