Programming in C and Data Structures (15PCD13/23)
FUNCTIONS, ARRAYS AND STRINGS

Topics:

Arrays and Strings: Using an array, using arrays with Functions, Multi-Dimensional arrays. String: Declaring, Initializing, Printing and reading strings, string manipulation functions, String input and output functions, array of strings, Programming examples and Exercises.

Functions: Functions in C, Argument Passing – call by value, call by reference, Functions and program structure, location of functions, void and parameter less Functions, Recursion, Programming examples and exercises

Course Outcome:

Illustrate the concepts of Arrays and Functions.

ARRAYS

Definition

An array is a series of variables, all being same type and size.

- Each variable in an array is called an *array element*.
- All the elements are of same data type, but may contain different values.
- The entire array is **contiguously stored** in memory.
- The position of each array element is known as array index or subscript.

Need for Array

In order to store values of same data type.

- Ease of declaration and data handling.

Types of array

One dimensional array

- Multidimensional array

Declaring Arrays

Syntax :

data_type array_name[array size/dimension];

Where

array_name is any valid identifier name

- Array size is an integer constant greater than 0

Examples

int a[10];

Here a is an array which can store a maximum of 10 integer values under a single reference name.

Hence its size in bytes = 10 * 2 bytes = 20 bytes.

float marks[6];

Here, marks is an array which can store a maximum of 6 floating point numbers.

Size in bytes = 6 * 4 bytes = 24 bytes.

char name[21];

- Here name is a character array which can store a maximum of 20 characters and 1 byte is always reserved for null character („\0").
- An array of characters is called as a string which is always terminated by „\0" character.
- Hence, the delimiter of the string is „\0"

Processing of array elements

- To process array elements, array index is considered.
- Index is also known as subscript.
- In C, the array index always starts from 0 and the last index is size -1.

Example

int A[5];
1st element A[0] 2nd element A[1] 3rd element A[2] 4th element A[3] 5th element A[4]

```
void main()
{
        int a[5], i;
        a[0]=10;
        a[3]=55;
        a[4]=40;
        for(i=0;i<5;i++)
        printf("%d\n", a[i]);
}
```

Output

```
10
Garbage value
Garbage value
55
40
```

Memory representation of 1-d arrays

Array elements are stored in contiguous (sequential) memory locations, ie, the array elements are continuously arranged in the primary memory.

Example

int a[6]; it occupies 6 * 2 bytes = 12 bytes of memory

Memory address	array index
0X100	a[0]
0X101	
0X102	a[1]
0X103	
0X104	a[2]
0X105	
0X106	a[3]
0X107	
0X108	a[4]
0X109	

0X10A	a[5]
0X10B	
0X10C	
0X10D	

Initialization of arrays

- Basic initialization
- Initialization without size
- Partial initialization

Basic initialization

int A[5] = {4, 8, 10, 3, 83};

4	8	10	3	83
A[0]	A[1]	A[2]	A[3]	A[4]

Initialization without size

int A[] = {4, 8, 56};Here compiler automatically calculates the size of array as 3.

4	8	56
A[0]	A[1]	A[2]

Partial initialization

intA[5] = {4, 8};

Here, A [0] = 4, A [1]=8, A [2]=garbage value, A [3]=garbage value, A [4]=garbage value

4	8	10	3	83
A[0]	A[1]	A[2]	A[3]	A[4]

Examples of strings

A string is a character array terminated by null character („\0“).

Example :

char name[8] = “STRING”;

„S“	„T“	„R“	„I“	„N“	„G“	„\0“	
0	1	2	3	4	5	6	7

char name[] = “PROGRAM”; Here, compiler automatically takes the size of string as 8.

„P“	„R“	„O“	„G“	„R“	„A“	„M“	„\0“
0	1	2	3	4	5	6	7

char name[5] = “Spring”; Since the array size is limited to 5, the output is “Spri”

„S“	„p“	„r“	„i“	„\0“
0	1	2	3	4

char name[8] = {„A“, „B“, „C“}; **Compiler will not add '\0' at the end of the string!!! So its not a string!**

„A“	„B“	„C“	Junk	Junk	Junk	Junk	Junk
0	1	2	3	4	5	6	7

Programs

1. **Write a program to input 10 integers and print them in different lines.**

```
void main()
{
        int a[10], i;
        printf("Enter 10 integers\n");
        for(i=0;i<10;i++)
        scanf("%d", &a[i]);
        printf("The integers are\n");
        for(i=0;i<10;i++)
        printf("%d\n", a[i]);
}
```

2. **Write a program to input 5 marks , find sum and average.**

```
void main()
{       float a[5], sum=0, avg;
        int i;
        printf("Enter 5 marks\n");
        for(i=0;i<5;i++)
        {
                scanf("%f", &a[i]);
                sum+=a[i];
        }
        avg=sum/5;
        printf("Sum=%f\nAverage=%f", sum, avg);
}
```

3. **Input n number of integers and find the smallest among them.**

```
void main()
{       int a[20], n, i, small;
        printf("Enter no. of elements\n");
        scanf("%d", &n);
        printf("Enter numbers\n");
        for(i=0;i<n;i++)
        scanf("%d", &a[i]);
        small=a[0];
        for(i=1;i<n;i++)
        {
                if(a[i]<small)
                small=a[i];
        }
```

```
        printf("The smallest number is %d", small);
}
```

4. Input n number of integers and print them in reverse order.

```
void main()
{
        int a[20], n, i;
        printf("Enter no. of elements\n");
        scanf("%d", &n);
        printf("Enter numbers\n");
        for(i=0;i<n;i++)
        scanf("%d", &a[i]);
        printf("The elements in reverse order are\n);
        for(i=n-1;i>=0;i--)
        printf("%d\n", a[i]);
}
```

5. Write a program to generate Fibonacci series for n terms.

```
void main()
{
        int a[20], n, i;
        printf("Enter no. of terms: ");
        scanf("%d", &n);
        a[0]=0;
        a[1]=1;
        for(i=2;i<=n;i++)
        a[i]=a[i-1] + a[i-2];
        printf("The Fibonacci series is\n");
        for(i=0;i<n;i++)
        printf("%d    ", a[i]);
}
```

6. Write a program to input n no. of integers, count the number of 0s, +ve numbers and –ve numbers.

```
void main()
{
        int a[20], n, i, zero=0, positive=0, negative=0;
        printf("Enter no. of terms: ");
        scanf("%d", &n);
        printf("Enter elements\n");
        for(i=0;i<n;i++)
        {
                scanf("%d", &a[i]);
                if(a[i]==0)
                zero++;
                else if(a[i]>0)
```

```
                positive++;
                else
                negative++;
        }
        printf("No. of zeros=%d\n+ve numbers=%d\n-ve numbers=%d",zero, positive, negative);
}
```

Bubble sort method

Sorting in ascending order

```
void main()
{
        int a[10], n, i, j, temp;
        printf("Enter no. of elements \n");
        scanf("%d", &n);
        printf("Enter elements\n");
        for(i=0;i<n;i++)
        scanf("%d", &a[i]);
        for(i=0;i<n-1;i++)              //      no. of passes
        {
                for(j=0;j<n-1-i ; j++)  //      no. of comparisons in each pass
                {
                        if(a[j]> a[j+1])
                        {
                                temp=a[j];
                                a[j]=a[j+1];
                                a[j+1]=temp;
                        }
                }
        }
        printf("The sorted numbers in ascending order is\n");
        for(i=0;i<n;i++)
        printf("%d\n", a[i]);
}
```

Sorting in descending order

```
void main()
{
        int a[10], n, i, j, temp;
        printf("Enter no. of elements \n");
        scanf("%d", &n);
        printf("Enter elements\n");
        for(i=0;i<n;i++)
        scanf("%d", &a[i]);
        for(i=0;i<n-1;i++)              //      no. of passes
        {
                for(j=0;j<n-1-i ; j++)  //      no. of comparisons in each pass
```

```
        {
            if(a[j] < a[j+1])
            {
                temp=a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
            }
        }
    }
    printf("The sorted numbers in descending order is\n");
    for(i=0;i<n;i++)
    printf("%d\n", a[i]);
}
```

Binary search

Pre conditions for binary search

-
- The array must be sorted either in ascending or descending order
- The lower bound and upper bound must be known.

```
void main()
{
    int a[10], n, i, key, high, low, mid;
    printf("Enter no. of elements \n");
    scanf("%d", &n);
    printf("Enter elements\n");
    for(i=0;i<n;i++)
    scanf("%d", &a[i]);
    // sort in ascending order by bubble sort method
    for(i=0;i<n-1;i++)
    {
        for(j=0;j<n-1-i ; j++)
        {
            if(a[j] > a[j+1])
            {
                temp=a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
            }
        }
    }
    printf("The sorted numbers in ascending order is\n");
    for(i=0;i<n;i++)
    printf("%d\n", a[i]);
    printf("Enter key element to be searched\n");
    scanf("%d", &key);
    low = 0;
```

```
        high=n-1;

        // binary search starts
        while(low<=high)
        {
                mid = (low + high)/2;
                if(a[mid]==key)
                {
                        printf("Search successful, key element found at %d position", mid +1);
                        break;
                }
                else if(a[mid]>high)
                low = mid + 1;
                else
                high = mid – 1;
        }
        if(low>=high)
        printf("Search unsuccessful, key element not found");
}
```

Two dimensional arrays

Declaration

Syntax :

data_type array_name [row size][column size];

Examples

int a[3][4];

The array „**a**" has a maximum of 3 rows and 4 columns.

Total no. of elements = 3 * 4 = 12

Total space occupied in the memory = 12 elements * 2 bytes = 24 bytes.

float x[10][5];

The array x has a maximum of 10 rows and 5 columns.

Total no. of elements = 10 * 5 = 50

Total space occupied in the memory = 50 elements * 4 bytes = 200 bytes.

char name[10][21];

The array name has a maximum of 10 names each having a maximum of 20 characters. I each string, 1 character is reserved for \0.

Total no. of characters = 10 * 21 = 210

Total space occupied in the memory = 210 elements * 1 byte = 210 bytes.

Memory representation of a 2-d array

Examples

int A[3][4];
The array A has a maximum of 3 rows and 4 columns.
Total no. of elements = 3 * 4 = 12
Total space occupied in the memory = 12 elements * 2 bytes = 24 bytes.

	0	1	2	3
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]

Initialization

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

	0	1
0	1	2
1	3	4
	5	6

int a[3][3] = { 4, 5, 7, 8, 2, 33, 1} ;

	0	1	2
0	4	5	7
1	8	2	33
2	1	junk	junk

char a[3][10] = { “Maths”, “Physics”, “Chemistry”} ;

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	**M**	**a**	**t**	**h**	**S**	**\0**				
1	**P**	**h**	**y**	**s**	**i**	**c**	**s**	**\0**		
2	**C**	**h**	**e**	**m**	**i**	**s**	**t**	**r**	**y**	**\0**

Examples:

Input 3 * 4 elements of a 2-D array and display in the matrix form

```
void main()
{
        int a[3][4], i, j;
        printf("Enter 3*4 elements\n");
        for(i=0;i<3;i++)
        {
                for(j=0;j<4;j++)
                scanf("%d", &a[i][j]);
        }
        printf("The matrix is\n");
        for(i=0;i<3;i++)
        {
                for(j=0;j<4;j++)
                printf("%d\t", a[i][j]);
                printf("\n");
        }
}
```

Input 2 matrices A(m * n) and B (p * q) , perform addition and subtraction on them.

```
void main()
{
        int a[6][6], b[6][6], c[6][6], d[6][6], m, n, p, q, i, j;
        printf("Enter size of A\n");
        scanf("%d%d", &m,&n);
        printf("Enter size of B\n");
        scanf("%d%d", &p,&q);
        if(m==p & n==q)
        printf("Addition & subtraction possible");
        else
        {
                printf("Addition & subtraction not possible");
                exit(0);
        }
        printf("Enter elements of A\n");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                scanf("%d", &a[i][j]);
        }
        printf("Enter elements of B\n");
        for(i=0;i<p;i++)
        {
                for(j=0;j<q;j++)
                scanf("%d", &b[i][j]);
```

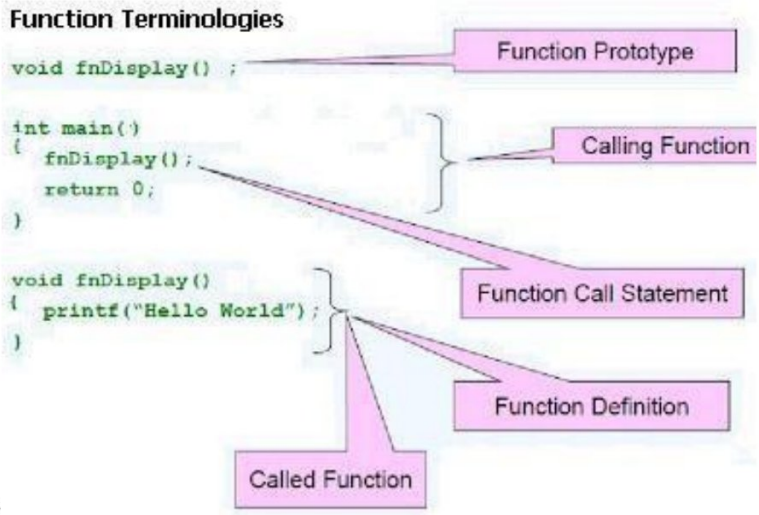
```
}
for(i=0;i<m;i++)
{
        for(j=0;j<n;j++)
        {
                c[i][j]=a[i][j] + b[i][j];
                d[i][j]=a[i][j] – b[i[j];
        }
}
printf(“Matrix C (Addition)\n”);
for(i=0;i<m;i++)
{
        for(j=0;j<n;j++)
        printf(“%d\t”, c[i][j]);
        printf(“\n”);
}
printf(“Matrix D (Subtraction)\n”);
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                printf(“%d\t”, d[i][j]);
                printf(“\n”);
        }
}
```

Programming in C and Data Structures (18CPS13/23)
FUNCTIONS

A complex problem is often easier to solve by dividing it into several smaller parts (pieces), each of which can be solved by itself. Each sub-problem is more manageable than the original problem. This is called ***structured*** programming.

These sub-problems which are easy to manage are made into ***Functions/Procedures/Modules*** in C. ***main()*** uses functions to solve the original problem. Thus A function is a block of code to perform a specific task.

Every C program has at least one function main( ). Without main() function, there is technically no C program.



Advantages of Functions

The following are the advantages of using user defined functions:

- Functions separate the concept (**what is done?**) from the implementation (**how it is done?**).
- Functions make programs easier to understand thus ***improves the readability of program.***
- Functions can be called several times in the same program, allowing the code to be ***reused.***

Functions in C

There are 2 types of functions in C programming:

- Library function
- User defined function

C allows the use of both User-Defined Functions and Library Functions.

Library Functions:

The functions that are written by designers of C compilers are called library function. The implementation details of these functionsare not known to the programmers

e.g printf(), scanf(), getchar(), putchar(), abs(), ceil(), rand(), sqrt(), pow()..

User Defined Function:

The functions written by the programmer to do specific tasks are called as user defined functions. The main() function is the best example for user defined function.

Elements of User Defined Function

The elements of user defined functions are:

1. Function declaration/ function prototype
2. Function call
3. Function definition.

Basic structure of C program with function is shown below:

```
#include <stdio.h>
void function_name(); //function declaration
int main()
{
.....
.....
function_name(); // function call
.....
.....
}
void function_name() //function definition
{
.....
.....
}
```

Function Declaration / Prototype

Every function in C program should be declared before they are used. Function declaration gives compiler information about function name, type of arguments to be passed and return type.

The syntax is shown below:

return_type function_name(type(1) argument(1),....,type(n) argument(n));

Here:

- Return-Type: Data type of the result returned
- (use void if nothing returned)
- Function-Name: Any valid Identifier
- Parameter list: Comma separated list of arguments
- Data type needed for each argument, If no arguments, use void or leave blank

Example:

```
void Display(void);
int print_data();
xyz();                  // default return type is int
double abc(int, int, char, float*);
int absolute(int);
void print_array(float*, int, int);
```

Function Call

Control of the program cannot be transferred to user-defined function unless it is called invoked.

The syntax is shown below:

function_name(argument(1),... argument(n));

Function Definition

Function definition contains programming codes to perform specific task.

The syntax is shown below:

return_type function_name(type(1) argument(1),..,type(n) argument(n))
{
//body of function
}

Example: Program to print a sentence using function.

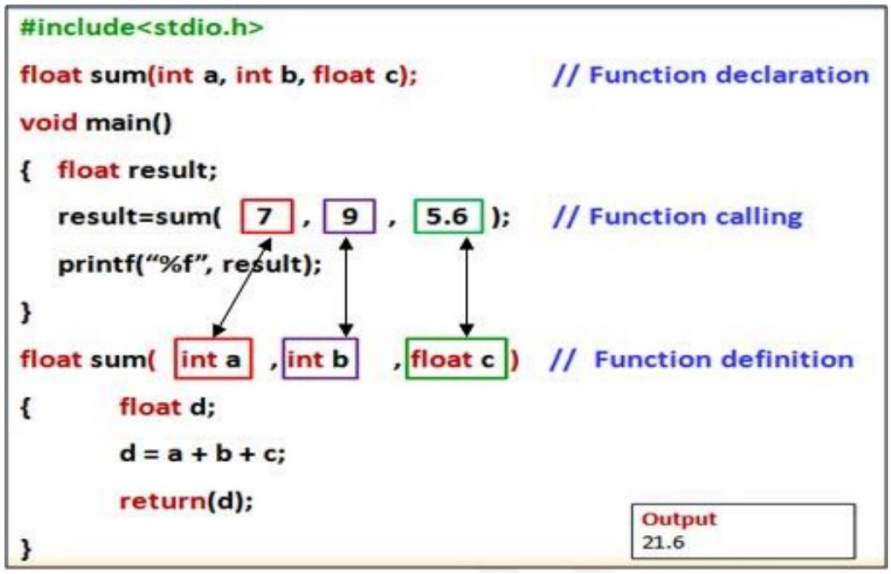
```
#include<stdio.h>
void display();          //function declaration
void main()
{

        display();       //function call
}
void display()           //function definition
{
        printf("C Programming");
        return;
}
```

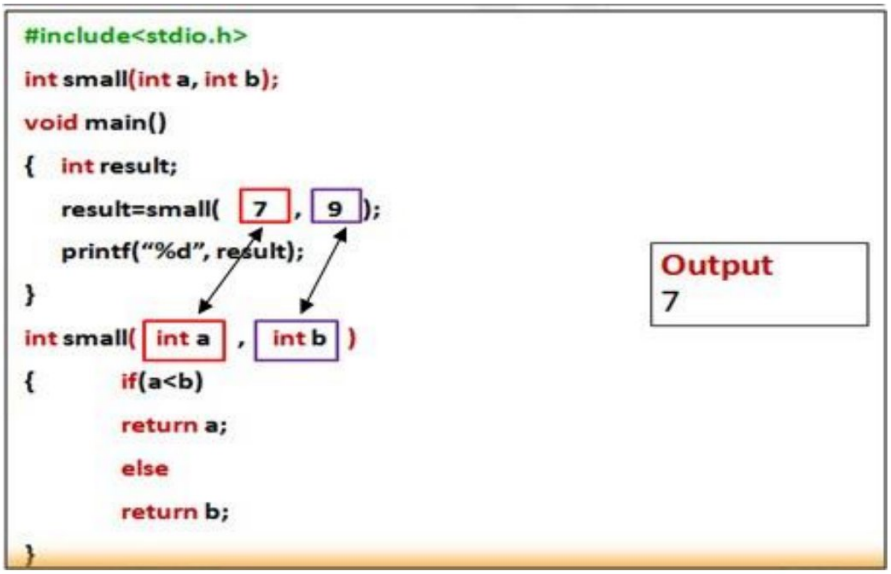
Output:
C Programming

Working of A Function

- As soon as the program is compiled, the function declaration arguments are matched with function definition arguments.
- If there is a mismatch, an error message is displayed.
- The compiler matches the
  - ✓ **Numer** of arguments
  - ✓ **Datatypes**
  - ✓ **Order** of arguments

  of function calling with function definition.
- Their names, however, do not need to match.





About Functions

- A program can have any number of functions in it.
- A function always **returns only one value.**
- The **default return type** of the function is **int.**
- The **data type** for each argument must be **specified** during function declaration and definition.
- Example :

  sum(int a, int b); // valid

  sum(int a, b); // invalid

A function **cannot be defined in another function definition**

Actual and Formal Arguments

Argument (or parameter) refers to data that is passed to function (function definition) while calling function. Arguments listed in function calling statements are referred to as actual arguments.

These actual values are passed to a function to compute a value or to perform a task. The arguments used in the function declaration are referred as formal arguments. They are simply formal variables that accept or receive the values supplied by the calling function. The number of actual and formal arguments and their data types should be same.

Example: Program to add two integers. Make a function add integers and display sum in main() function.

```
#include  <stdio.h>
int add(int a, int b);
int main()
{
        int a,b,sum;
        printf("Enters two number to add \n");
        scanf("%d %d", &a,&b);       //actual arguments
        sum=add(a,b);
        printf("\n sum=%d", sum);
        return 0;

}
```

Output:

Enters two number to add
2 3
Sum = 5

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

Function Call Methods

C Supports two types of calling a Function:

- Call by Value
- Call by Address

Call by Value method

- The formal argument is created and the value of actual argument is copied into it.
- The formal argument is created with different memory location. It is created when the function is called and erased from the memory when the function exits.
- Since the memory locations of formal and actual locations are different, the changes made to formal arguments are not reflected in the actual arguments.

Example 1

```
void demo(int a)
{
        a=20;
        printf("%d\n",a);
}
void main()
{
        int x=10;
        printf("%d\n",x);
        demo(x);
        printf("%d\n",x);
}
```

Working

main()	**demo()**
x = 10	**step 1**
	a = value of x = 10
	step 2
	a = 20

Output

10
20
10

Example 2

```
modify(int a)
{
        a*=3;
        printf("%d\n",a);
        return (a);
```

```
}
void main()
{
        int a=5, x;
        printf("%d\n",a);
        x = modify(a);
        printf("%d "d\n",x, a);
        a=modify(a);
        printf("%d\n",a);
}
```

Working

main()

a = 5

goto**modify() – 1st time**

after calling modify() – 1st time,

x = 15

goto**modify() – 2nd time**

after calling modify() – 2nd time,

a = 15

modify() – 1st time

a = value of actual argument a = 5

a*=3 ----> a = 5*3 = 15

return 15 to main()

modify() – 2nd time

a = value of actual argument a = 5

a*=3 ----> a = 5*3 = 15

return 15 to main()

Output

5

15

15 5

15

15

Example 3

```
void exchange(int a, int b)
{
        int t=a;
        a=b;
        b=t;
        printf("In exchange()\n");
        printf("%d %d\n", a, b);
}
void main()
{
        int a=12, b=43;
        printf("Before exchange()\n");
```

Output

Before exchange()

12 43

In exchange()

43 12

After exchange()

12 43

```
        printf("%d %d\n", a, b);
        exchange(a, b);
        printf("After exchange()\n");
        printf("%d %d\n", a, b);
}
```

Call by Reference method

- The formal argument is a reference of the actual argument.
- The formal argument shares the same memory location of the actual argument.
- Since the memory locations of formal and actual locations are same, the changes made to formal arguments are reflected in the actual arguments.
- The actual argument is call by reference method should be a variable and not a constant or an expression.

Example 1

```
void modify(int&a)
{
        a+=10;
        printf("%d\n",a);
}
void main()
{
        int x=5;
        printf("%d\n", x);
        modify(x);
        printf("%d ", x);
}
```

main()

(a) x = 5 + 10 = 15

Output

5

15

15

Example 2

```
modify(int &a)
{
        a*=3;
        printf("%d\n",a);
        return (a);
}
void main()
{       int x=5, y=1;
        printf("%d %d\n", x, y);
```

main()

x = 5 * 3 = 15

y = 15 * 3 = 75

Output

5 1

15

15 15

75

15 75

```
        y = modify(x);
        printf("%d    %d\n", x, y);
        modify(y);
        printf("%d    %d\n", x, y);
}
```

Example 3

```
void demo(int x, int &y)
{
        x*=3;
        y*=4;
        printf("%d    %d\n", x, y);
}
void main()
{
        int a=10, b=5;
        printf("%d    %d\n", a, b);
        demo(a, b);
        printf("%d    %d\n", a, b);
}
```

main()
a = 10
b = 5 * 4 = 20

Output

10 5
30 20
10 20

Example 4

```
void exchange(int &a, int &b)
{
        int t=a;
        a=b;
        b=t;
        printf("In exchange()\n");
        printf("%d    %d\n", a, b);
}
void main()
{
        int a=12, b=43;
        printf("Before exchange()\n");
```

Output

Before exchange()
12 43
In exchange()
43 12
After exchange()
43 12

```
        printf("%d   %d\n", a, b);
        exchange(a, b);
        printf("After exchange()\n");
        printf("%d   %d\n", a, b);
}
```

Differences between call by value and call by reference methods

Call by Value	Call by Reference
Formal argument is a copy of the Actual argument.	Formal argument is a reference of the Actual argument.
The memory locations of formal and actual arguments are different	The memory location of formal and actual arguments is same
Changes made to the formal arguments are not reflected in the actual arguments.	Changes made to the formal arguments are reflected in the actual arguments.
Slow	Faster
Example with output	Example with output

Basic Function Designs

1. void functions without parameters
2. void functions with parameters
3. non – void functions without parameters
4. non – void functions with parameters

void function without parameters

Example :

```
#include<stdio.h>
void add()
{
        int a, b, c;
        printf("Input 2 nos.:");
        scanf("%d%d", &a, &b);
        c=a+b;
        printf("sum=%d", c);
}
void main()
{
        add();
}
```

void function with parameters

Example :

```
#include<stdio.h>
void add(int a, int b)
{
        int c;
        c=a+b;
        printf("sum=%d", c);
}
void main()
{
        int x, y;
        printf("Input 2 nos.:");
        scanf("%d%d", &x, &y);
        add(x, y);
}
```

non-void function without parameters

Example :

```
#include<stdio.h>
int add()
{
        int a, b, c;
        printf("Input 2 nos:");
        scanf("%d%d", &a, &b);
        c=a+b;
        return c;
}
void main()
{
        int x;
        x = add();
        printf("sum=%d", x);
}
```

non-void function with parameters

Example :

```
#include<stdio.h>
int add(int a, int b)
{
        int c;
        c=a+b;
        return (c);
}
void main()
{
        int x, y, z;
        printf("Input 2 nos.:");
        scanf("%d%d", &x, &y);
        z=add(x, y);
        printf("sum=%d", z);
}
```

Scope

- Region of program in which a defined object(variable) is visible.
- It is the part of the program in which we can use the object(variable) name.

Types

- **Local scope**
  Variables defined within a block or a function have local scope.
- **Global scope**
  Variables defined outside all the blocks and functions have global scope.

Local variables

- Local variable is declared **inside a function.**
- It is **created** when the **function is called** and destroyed when the function exits.
- Since it is private to the function in which it is declared, it should be **used only within that function.**
- It is **alive and active** only **within the function** in which it is declared.

Example

```
sum(int a, int b)
{
    int c = a+b ;
    return c;
}
void main()
{
    int x=12, y=14, z;
    z=sum(x,y);
```

The variables **x , y and z** are local to **main()** and hence have to be used only in main().
The variables **a, b and c** are local to **sum()** and hence have to be used only in sum().

```
    printf("%d", z);
}
```

Global variables

- Global variable is declared **outside all the functions.**
- It is **created** when **program is executed** and destroyed when the program exits.
- Since it is a public variable, it can be **used anywhere** within the program.
- It is **alive and active throughout the program.**

Global variables

```
Example
int a=10;        // global variable
void demo()
{   a+=3;
    printf("%d\n",a);
}
void main()
{   printf("%d\n",a);
    a+=20;
    printf("%d\n",a);
    demo();
    printf("%d\n",a);
}
```

::a = 10 +20 = 30 +3 = 33

```
Output :
10
30
33
33
```

Recursion

- When a function calls itself it is called as **Recursion.**
- Many mathematical, searching and sorting algorithms, can be simply expressed in terms of a recursive definition.
- A recursive definition has two parts:
  - **Base condition** : When a function will terminate
  - **Recursive condition** :The invocation of a recursive call to the function
- When the problem is solved through recursion, the source code looks elegant.

/* program to find factorial of a number using recursion */

```
#include<stdio.h>
#include<process.h>
long int fact(int x)
{
        long int f;
        if(x == 1||x==0)
        return (1);
        else
        f = x * fact (x - 1);
        return (f);
}
void main( )
{
        int a;
```

```
        long int factor;
        printf ("Enter number :");
        scanf ("%d", &a);
        if(a<0)
        exit(0);
        factor = fact(a);
        printf ("Factorial value = %d", factor);
}
```

Write a recursive function to get nth term of Fibonacci series.

```
#include<stdio.h>
int fibo(int m)
{
        if(m==1 || m==2)
        return 1;
        else
        return (fibo(m-1) + fibo(m-2));
}
void main()
{
        Int num, result;
        printf("Enter nth number\n");
        scanf("%d", &num);
        result=fibo(num);
        printf("The nth fibonacci number is %d", result);
}
```

**********END**********

Example programs for functions:

WAP to input marks of type int. Write a function to take marks as argument and assign grade according to the marks obtained. Return the grade to the main() function.

Marks	Grade
75-100	A
60-74	B
50-59	C
40-49	D
<40	E

```
#include<stdio.h>
char assign(int);
void main()
{
        int marks;
        char grade;
        printf("Enter marks:");
        scanf("%d", &marks);
        grade=assign(marks);
```

```
        printf("The grade is %c\n", grade);
}
char assign(int m)
{
        char g;
        if(m>=75 && m<=100)
        g='A';
        else if(m>=60)
        g='B';
        else if(m>=50)
        g='C';
        else if(m>=40)
        g='D';
        else g='E';

        return(g);
}
```

Write a function to print LSB of an integer.

```
#include<stdio.h>
int LSB(int n)
{
        int m=n%10;
        return (m);
}
void main()
{
        intnum, r;
        printf("Enter an integer\n");
        scanf("%d", &num);
        r=LSB(num);
        printf("LSB=%d",r);
}
```

Write a function to add 2 LS digits of an integer

```
#include<stdio.h>
int first(int n)
{
        int m=n%10;
        return (m);
}
int second(int n)
{
        int  a,  b;
        a=n/10;
        b=a%10;
        return (b);
}
void main()
{
```

```
        intnum, r;
        printf("Enter an integer\n");
        scanf("%d", &num);
        r=first(num) + second(num);
        printf("Result=%d",r);
}
```

Write a functionthat takes x and n as arguments and return the sum.
Sum = $1 + x + x^2 + x^3 +$---------- $+x^n$
Use it in the main() function to print the result.

```
#include<stdio.h>
float sum(float x, int n)
{
        float s =0;
        int i;
        for(i=0;i<=n;i++)
        sum+=pow(x,i);
        return sum;
}
void main()
{
        float x, s;
        int n;
        printf("Enter x and n\n");
        scanf("%f %d", &x, &n);
        s=sum(x,n);
        printf("sum=%f", s);
}
```

Write a functionto find factorial of a number. Use it in the main() function to print the result.

```
#include<stdio.h>
#include<process.h>
Long int fact(int n)
{
        int i;
        long int f;
        for(i=1,f=1;i<=n;i++)
        f*=i;
        return f;
}
void main()
{
        Int num;
        long int result;
        printf("Enter an integer\n");
        scanf("%d", &num);
        if(num<0)
        {
                printf("Invalid number");
                exit(0);
```

```
        }
        result=fact(num);
        printf("The factorial is %ld", result);
}
```

Programming in C and Data Structures (18CPS13/23)
STRINGS

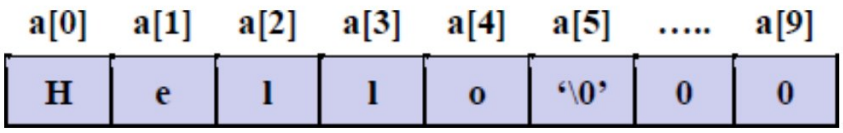
Strings:

String is an Array of Characters terminated by a NULL character '/0'.

Example:

char a[10] = "Hello";

The above string can be pictorially represented as shown below:



String Variable

There is no separate data type for handling strings in C. They are treated as just an array of characters. So, a variable which is used to store an array of characters is called a string variable.

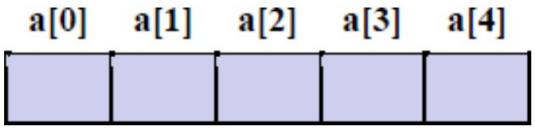
Declaration of String

Strings are declared in C in similar manner as arrays. Only difference is that, strings are of char type.

Example:

char s[5]; //string s can hold maximum of 5 characters including NULL character

The above code can be pictorially represented as shown below:

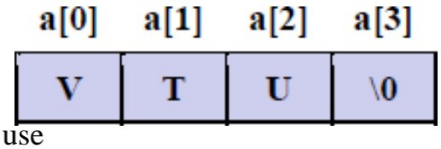


Initialization of String

Example:

char s[4]={'V', 'T', 'U' };

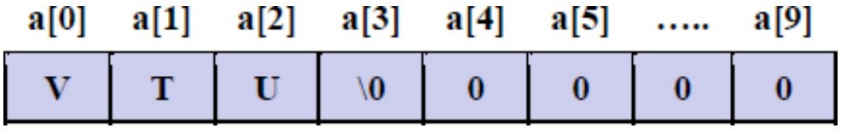
The above code can be pictorially represented as shown below:



we can also use

char s[10]="VTU";

Here 10 bytes of memory is allocated and only three characters are stored and remaining will be initialized to zero/ null character.



Example: Program to illustrate the initialization of string.

#include<stdio.h>

```
void main()
{
char s[4]={'V','T','U'}; //or char s[4]="VTU";
printf("Value in array s[0] : %c \n", s[0]);
printf("Value in array s[1] : %c \n", s[1]);
printf("Value in array s[2] : %c \n", s[2]);
}
```

Output:

```
Value in array s[0] : V
Value in array s[1] : T
Value in array s[2] : U
```

Reading & Printing Strings

The strings can be read from the keyboard and can be displayed onto the monitor using following formatted functions:

- Formatted input function: scanf()
- Formatted output function: printf()

Example: Program to illustrate the use of scanf() and printf().

```
#include<stdio.h>
void main()
{
        char name[10];
        printf("enter your name: \n");
        scanf("%s", name);
        printf("welcome: ");
        printf("%s", name);

}
```

Output:

```
Enter your name
Mark
Welcome Mark
```

STRING INPUT/OUTPUT FUNCTIONS: gets, puts

The strings can be read from the keyboard and can be displayed onto the monitor using following unformatted functions:

- Unformatted input function: gets()
- Unformatted output function: puts()

Example: Program to illustrate the use of gets() and puts().

```
#include<stdio.h>
void main()
{
        char name[10];
        printf("enter your name: \n");
        gets(name); //same as scanf("%s", name);
        printf("welcome: ");
        puts(name); //same as printf("%s", name);

}
```

Output:

```
enter your name:
rama
welcome: rama
```

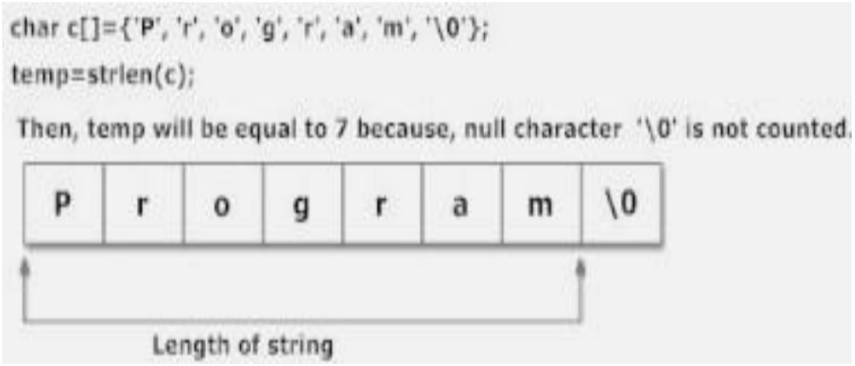
String Handling Functions

- Strings are often needed to be manipulated by programmer according to the need of a problem.
- All string manipulation can be done manually by the programmer but, this makes programming complex and large.
- To solve this, the C supports a large number of string handling functions.
- There are numerous functions defined in **<string.h>** header file.

1. strlen( )

- This function calculates the length of string. It takes only one argument, i.e., string-name.
- The syntax is shown below:

char strlen(char str[ ]);



Example: Program to illustrate the use of strlen().

```
#include<string.h>
#include<stdio.h>
void main()
{
        char c[20];
        int len;
        printf("Enter string whose length is to be found:");
        gets(c);
        len=strlen(c);
        printf("\n Length of the string is %d ", len);

}
```

Output:

```
Enter string whose length is to be found:
program
Length of the string is 7
```

2. strcpy( )

- This function copies the content of one string to the content of another string. It takes 2 arguments.
- The syntax is shown below:

char strcpy(destination,source);

where source and destination are both the name of the string.

Example: Program to illustrate the use of strcpy().

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

Output:

```
Enter string: notes
Copied string: notes
```

```
        char src[20],dest[20];
        printf("Enter string: ");
        gets(src);
        strcpy(dest, src);      //Content of string src is copied to string dest
        printf("Copied string: ");
        puts(dest);
}
```

3. strncpy( )

The C library function **strncpy** copies up to **n** characters from the string pointed to, by **src** to **dest**. In a case where the length of src is less than that of n, the remainder of dest will be padded with null bytes.

Syntax:

char strncpy(char dest[ ], const char src[ ], int n);

Example:

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

int main()
{

  char src[20] “this is tutorials”;
  char dest[12] ;

  strncpy(dest, src, 10);

  printf("Final copied string : %s\n", dest);

  return(0);
}
```

Output:

This is tu

4. strcat( )

- This function joins 2 strings. It takes two arguments, i.e., 2 strings and resultant string is stored in
- the first string specified in the argument.

The syntax is shown below:

Char strcat(char first_string[ ],char second_string[ ]);

Example: Program to illustrate the use of strcat().

```
#include <stdio.h>
#include <string.h>
void main()
{
        char str1[10], str2[10};
        printf("Enter First String:");
        gets(str1);
        printf("\n Enter Second String:");
        gets(str2);
```

Output:

Enter First String: rama
Enter Second String: krishna
Concatenated String is ramakrishna

strcat(str1,str2); //concatenates str1 and str2 and

```
        printf("\n Concatenated String is ");
        puts(str1); //resultant string is stored in str1
}
```

5. strncat( )

strncat( ) function in C language concatenates (appends) portion of one string at the end of another string. The syntax is shown below:

char strncat (char destination[ ], char source[ ], int num);

In strncat( ) operation, null character of destination string is overwritten by source string's first character and null character is added at the end of new destination string which is created after strncat( ) operation.

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

int main( )
{
  char source[ ] = " fresh2refresh" ;
  char target[ ]= "C tutorial" ;

  printf ( "\nSource string = %s", source ) ;
  printf ( "\nTarget string = %s", target ) ;

  strncat ( target, source, 5 ) ;


  printf ( "\nTarget string after strncat( ) = %s", target ) ;
}
```

Output:

Source string = fresh2refresh
Target string = C tutorial
Target string after strcat( ) = C tutorial fres

6. strcmp( )

- This function compares 2 string and returns value 0, if the 2 strings are equal. It takes 2 arguments, i.e., name of two string to compare.
- The syntax is shown below:

  char strcmp(char string1[ ], char string2[ ]);

Return Value

This function returns values that are as follows:

- if Return value < 0 then it indicates str1 is less than str2.
- if Return value > 0 then it indicates str2 is less than str1.
- if Return value = 0 then it indicates str1 is equal to str2.

Example: Program to illustrate the use of strcmp().

```
#include <string.h>
#include<stdio.h>
void main()
{
        char str1[30],str2[30];
        printf("Enter first string: ");
        gets(str1);
```

Output:

Enter first string: rama
Enter second string: rama
Both strings are equal

```
        printf("Enter second string: ");
        gets(str2);
        if(strcmp(str1,str2)==0)
        printf("Both strings are equal");
        else
        printf("Strings are unequal");
}
```

7. strncmp( )

The C library function **strncmp** compares at most the first **n** bytes of **str1** and **str2**.
Following is the declaration for strncmp() function.

```
int strncmp(char str1[ ], char str2[ ], int n);
```

Return Value

This function returns values that are as follows:

- if Return value < 0 then it indicates str1 is less than str2.
- if Return value > 0 then it indicates str2 is less than str1.
- if Return value = 0 then it indicates str1 is equal to str2.

Example:

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

int main ()
{

  char str1[15];
  char str2[15];
  int ret;
  strcpy(str1, "abcdef");
  strcpy(str2, "ABCDEF");
  ret = strncmp(str1, str2, 4);
  if(ret < 0)
  {
    printf("str1 is less than str2");
  }
  else if(ret > 0)
  {
    printf("str2 is less than str1");
  }
  else
  {
    printf("str1 is equal to str2");
  }
    return(0);
}
```

Output:

```
str2 is less than str1
```

8. strrev( )

strrev( ) function reverses a given string in C language.
Syntax for strrev( ) function is given below.

char strrev(char string[ ]);

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

int main()
{

  char name[30] = "Hello";
   printf("String before strrev( ) : %s\n",name);
   printf("String after strrev( ) : %s",strrev(name));
   return 0;
}
```

Output:

String before strrev( ) : Hello
String after strrev( ) : olleH

9. strupr( )

strupr( ) function converts a given string into uppercase.
Syntax for strupr( ) function is given below.

char strupr(char string[ ]);

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

int main()
{

   char str[ ] = "Modify This String To Upper";
   printf("%s\n",strupr(str));
   return  0;
}
```

Output:

MODIFY THIS STRING TO UPPER

10. strlwr( )

strlwr( ) function converts a given string into lowercase.
Syntax for strlwr( ) function is given below.

char *strlwr(char *string);

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

   char str[ ] = "MODIFY This String To LOwer";
   printf("%s\n",strlwr (str));
   return  0;
}
```

Output:

modify this string to lower