# **Module Code:** CSE401

# Session 4:Arrays & Functions

**Session speaker :**

**Prema Monish**
premamonish.tlll@msruas.ac.in

# Session Objectives

- To learn about array declaration and manipulation
- To learn about matrix operation using two dimensional arrays
- To learn about string handling using arrays.
- To learn about functions and function declaration in C
- To about passing values to the functions

# Session Topics

- Accessing the array elements and array initialization
- Single and multidimensional array
- Functions in C and function definitions
- Passing arguments to a function
- Recursive Functions

# Arrays

- Some times, we need to handle multiple values through the program. Say for example, need to accept sales of a product for 12 months and find months of maximum and minimum sales.
- One way to accept sales of 12 months is declaration of 12 variables, accepting the data into individual variables and finding maximum and minimum using if-else selection statement. It would be a 100 line program.

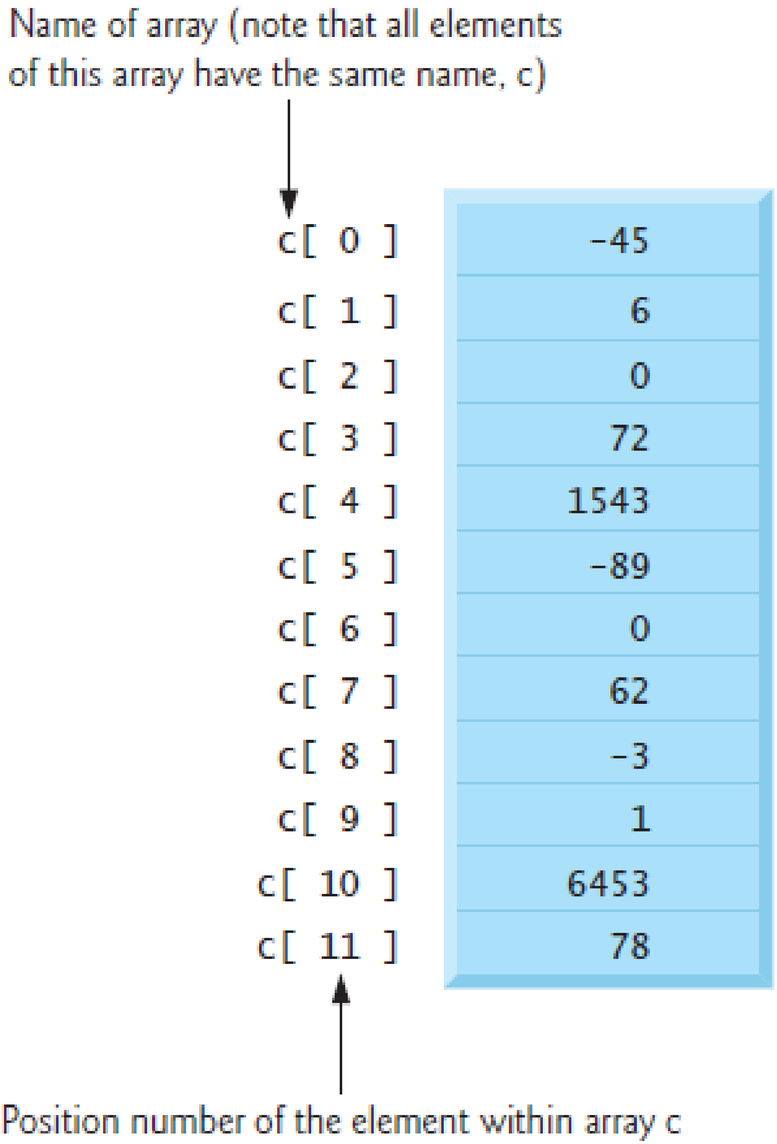
# Arrays

- Collection of elements of same data type
- All these elements are stored in consecutive memory locations
- Values can repeat – It is not a set

- The collection is represented by one name in the programming language
- Each individual data in the array is referenced by a subscript or index (positive integer constant or expression) enclosed in a pair of square brackets []

# Arrays

- Array contains 12 elements
- The first element in every array is the zeroth element
  - first element of array c is referred to as c[0]
  - second element of array c is referred to as c[1]
- In general, the i$^{th}$ element of array c is referred to as c[i – 1]



Name of array (note that all elements of this array have the same name, c)

| | |
|---|---|
| c[ 0 ] | -45 |
| c[ 1 ] | 6 |
| c[ 2 ] | 0 |
| c[ 3 ] | 72 |
| c[ 4 ] | 1543 |
| c[ 5 ] | -89 |
| c[ 6 ] | 0 |
| c[ 7 ] | 62 |
| c[ 8 ] | -3 |
| c[ 9 ] | 1 |
| c[ 10 ] | 6453 |
| c[ 11 ] | 78 |

Position number of the element within array c

# Defining Arrays

- Arrays occupy space in memory
- Programmer specify the type of each element and the number of elements required by each array
- Syntax

  <data type> <identifier> [<*constant* size>];
- Example

```
int numArray[20]; /*tell the computer to reserve 20 spaces.
          Elements are from numArray[0] to numArray[19]*/
```

or

```
 #define N 20
 int numArray[N];
```

# Array  Initialisation

- Individual elements of the array can be initialised
  - Initial values must be constants, never be variables or function calls

- Example

```
int numArray[4]= {10,20,30,40};
        /*4 is size. numArray[0]=10,… numArray[3]=40*/
or
 int numArray[4];
numArray[0]=10;
 numArray[1]=20;
```

# Array Initialisation contd.

- The array definition

```
int n[5] = { 32, 27, 64, 18, 95, 14 };
```

causes a syntax error because there are six initializers and only five array elements

- If the array size is omitted from a definition with an initializer list, the number of elements in the array will be the number of elements in the initializer list
- For example,

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

would create a five-element array

# Address of the Array Elements

- Array name is the same as the address of the array’s first element

```
int array[5];  /* define an array of size 5 */
printf( " array = %p \n &array[0] = %p \n &array = %p\n",
array, &array[ 0 ], &array );
```

```
Output : array = 0012FF78
         &array[0] = 0012FF78
         &array = 0012FF78
```

- %p conversion specifier
  - a special conversion specifier for printing addresses
  - Normally outputs addresses as hexadecimal numbers

# Algorithms

- Arrays

  *<identifier>: array* [<initial value> .. <final value>] *of* <Primitive data type>*;*

  *<identifier>[*<index value>*]*

- Examples

  *numArray: array* [0 .. n] *of* Integer;

  *numArray[10]* := 20*;*

# Algorithm - Reading an Array

```
Algorithm sigmaN (numArray: array [0 .. N] of Integer):Integer
var  i, temp: Integer; {temp is the return value}
begin
    for i in 0 to N, step 1 do
    begin
        writeln (‘Please enter the number at index ’, i, ‘:’);
        readln (numArray[i]);
    end
end
```

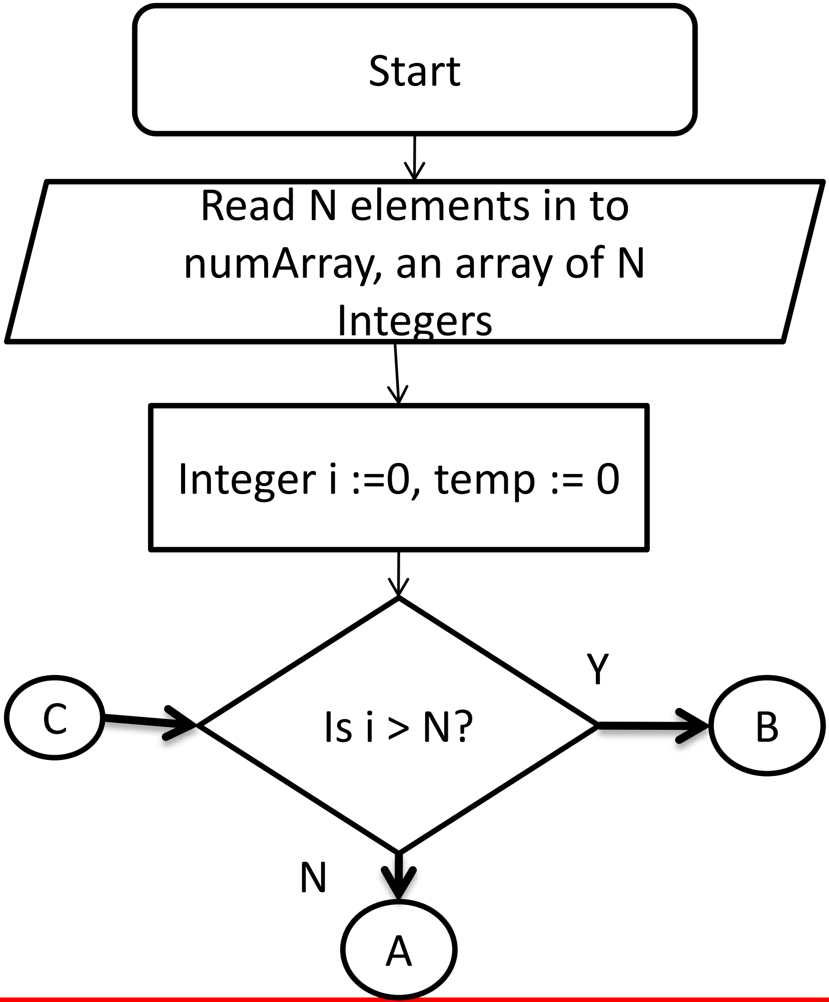
# Algorithm – Summation of N numbers

**Example:**

```
Algorithm sigmaN (numArray: Array [0 .. N] of Integer):Integer
var  i, temp: Integer; {temp is the return value}
Begin
    temp := 0;
    for i in 0 to n, step 1 do
    begin
        temp := temp + numArray[i];
    end
    writeln("Summation of numArray is:",temp);
stop
```

# Flow Charts

- Summation of N numbers

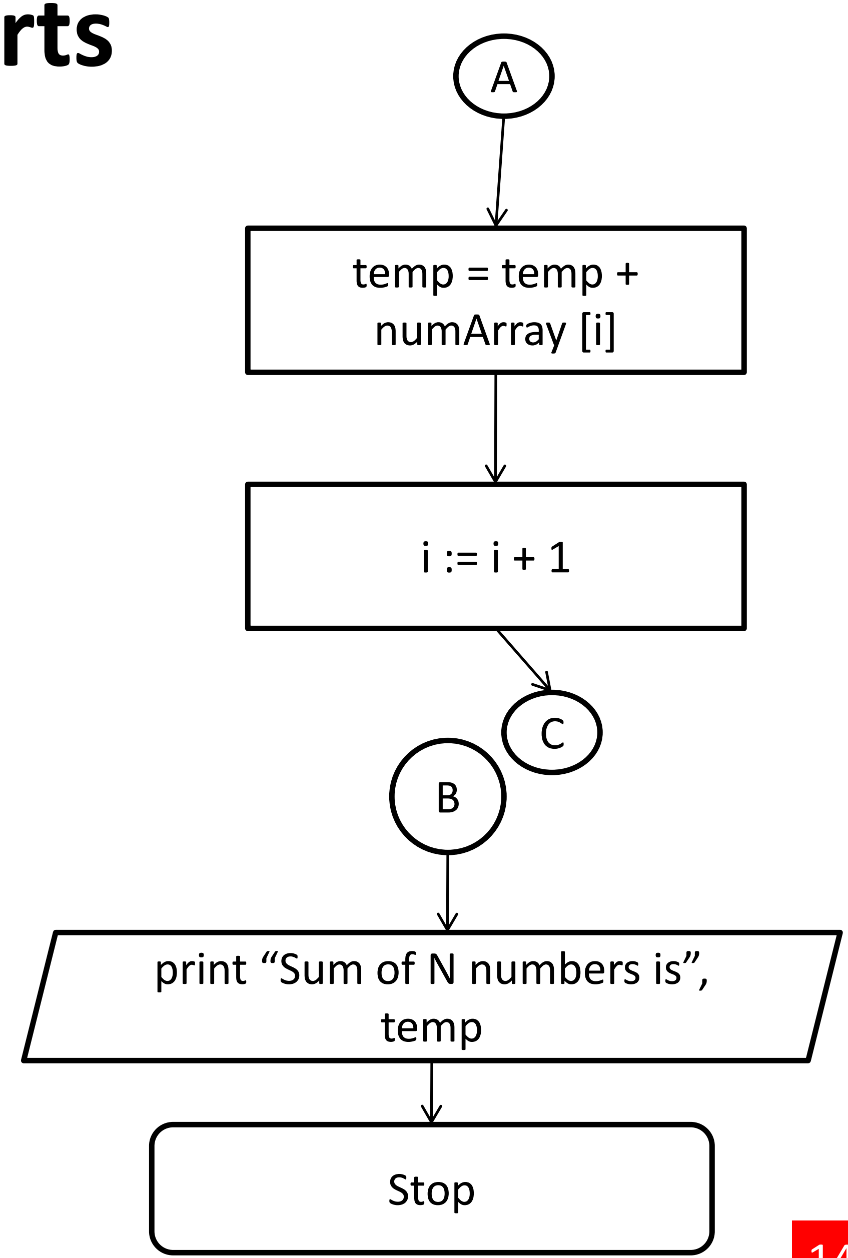


Start

Read N elements in to numArray, an array of N Integers

Integer i :=0, temp := 0

Is i > N?



temp = temp + numArray [i]

i := i + 1

print “Sum of N numbers is”, temp

Stop

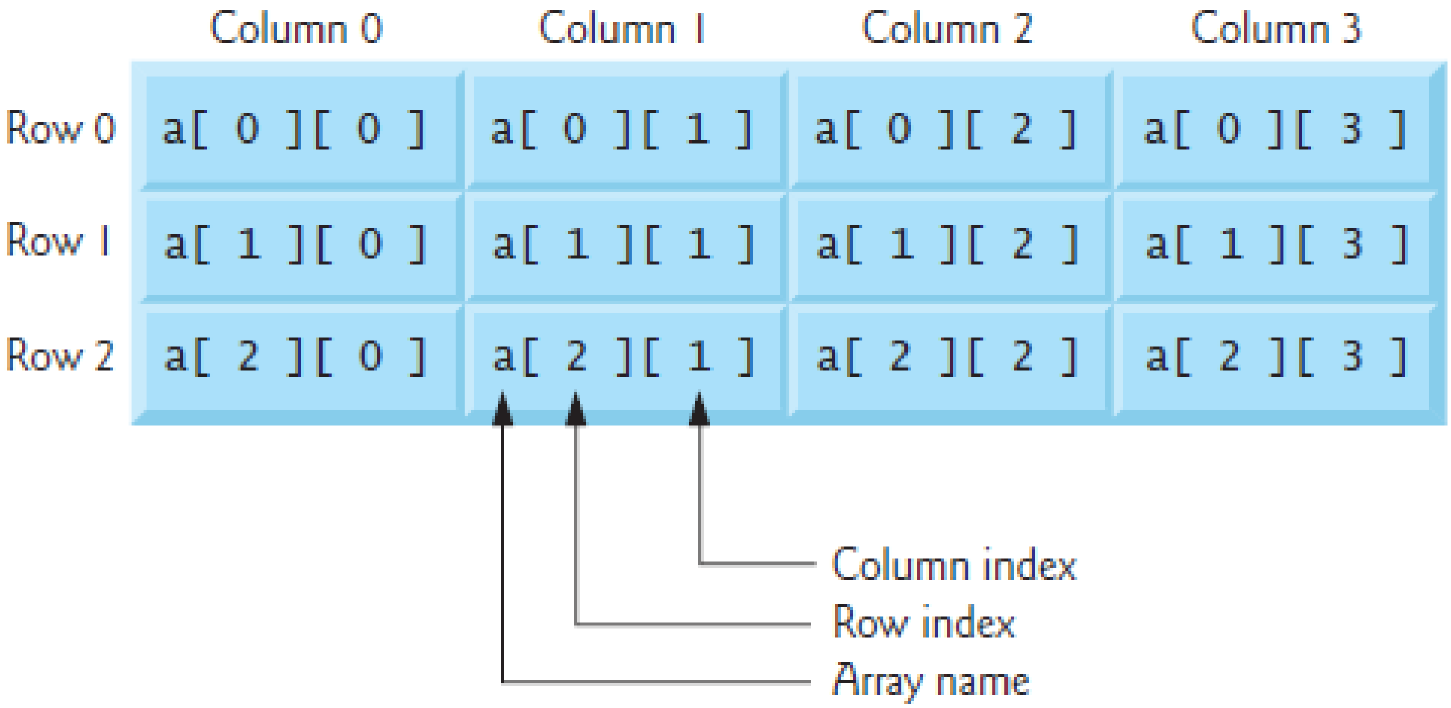
# Two-Dimensional Arrays

- A two-dimensional array can be think as a table which will have x number of rows and y number of columns

- In general, an array with *m rows and n* columns is called an *m-by-n array*

# Two-Dimensional Arrays

- The array contains three rows and four columns, so it’s said to be a 3-by-4 array



|  | Column 0 | Column 1 | Column 2 | Column 3 |
|---|---|---|---|---|
| Row 0 | a[ 0 ][ 0 ] | a[ 0 ][ 1 ] | a[ 0 ][ 2 ] | a[ 0 ][ 3 ] |
| Row 1 | a[ 1 ][ 0 ] | a[ 1 ][ 1 ] | a[ 1 ][ 2 ] | a[ 1 ][ 3 ] |
| Row 2 | a[ 2 ][ 0 ] | a[ 2 ][ 1 ] | a[ 2 ][ 2 ] | a[ 2 ][ 3 ] |

# Accessing Two-Dimensional Array Elements

- Array declaration

  <data type> <identifier> [<row size>] [<column size>];

  int matrix[2][3]; //array name is matrix with 2 rows and 3 columns

- An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array.
- Example:

  int val = a[2][3]; // take 4th element from the 3rd row of the array

# Initializing Two-Dimensional Arrays

- Multidimensional arrays may be initialized by specifying bracketed values for each row

- An array with 3 rows and each row has 4 columns

```
int a[3][4] = { {0, 1, 2, 3} , {4, 5, 6, 7} , {8, 9, 10, 11}};
```

is equivalent to

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

# Initializing Two-Dimensional Arrays contd.

- If there are not enough initializers for a given row, the remaining elements of that row are initialized to 0

```
int b[2][2] = { { 1 }, { 3, 4 } };
```

would initialize

b[0][0] to 1

b[0][1] to 0

b[1][0] to 3

b[1][1] to 4

# Multi-dimensional Arrays

- General form of a multidimensional array declaration:

  type name[size1][size2]...[sizeN];

- For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array:

  int threedim[5][10][4];

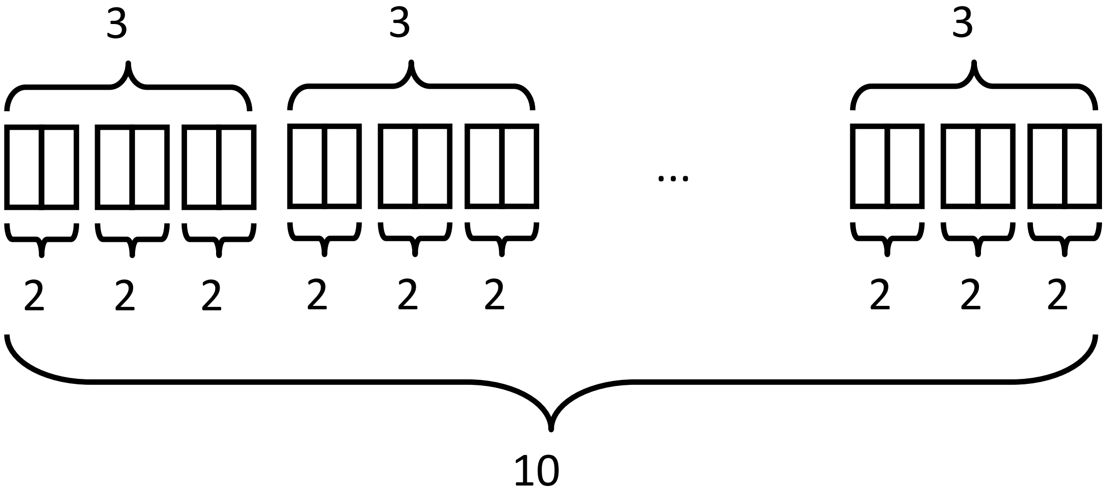
# Multidimensional Arrays

- Array declarations read right-to-left

  int a[10][3][2];

  “an array of ten arrays of three arrays of two ints”

- In memory



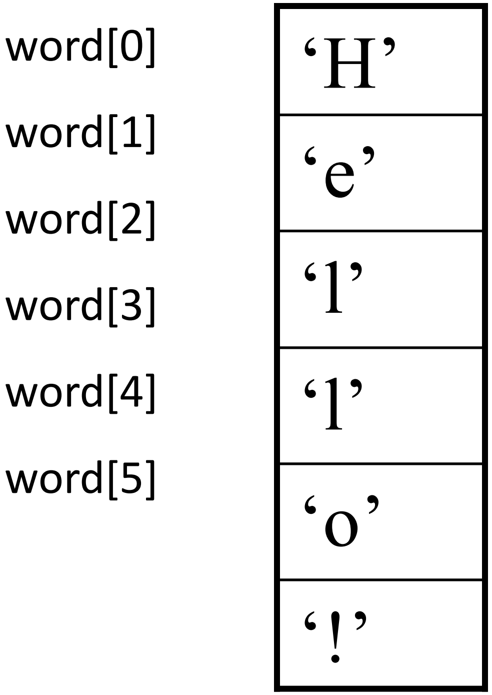
# Twodimensional Arrays

```c
#include<stdio.h>
int main()
{
int a[50][50],n,m,i,j;
printf("Enter the class of matrix:\n");
scanf("%d%d",&n,&m);
printf("Enter %dx%d matrix:\n",n,m);
for(i=0;i<n;i++)
  for(j=0;j<m;j++)
      scanf("%d",&a[i][j]);
printf("The given matrix:\n");
for(i=0;i<n;i++)
{
   for(j=0;j<m;j++)
       printf("%d",a[i][j]);
   printf("\n");
}
return 0;
}
```

```
Enter the class of matrix:
3   4
Enter 3×4 matrix:
12   45   67   88
32   56   44   23
78   56   33   89
The given matrix:
12   45   67   88
32   56   44   23
78   56   33   89
```

# Arrays Of Characters

- If you want to deal with variables that can hold more than a single character, then the array of characters comes into play.
- Eg: char word [ ]={ ‘H’,’e’,’l’,’l’,’o’,’ !’};



word[0] ‘H’

word[1] ‘e’

word[2] ‘l’

word[3] ‘l’

word[4] ‘o’

word[5] ‘!’

The array word in memory

# Array of Strings

- Strings in C are an array of characters terminated with a null character, '\0',.
- This means that the length of a string is the number of characters it contains plus one to store the null character.
- Examples:

```
char          string_1 = "Hello";
char string_2[ ]  = "Hello";
char string_3[6] = "Hello";
```

One can use the string format specifier, %s, to handle strings.

# Initializing and Displaying character strings

- Initialization of character arrays

```
char word []={“Hello!”};
char word[]=“Hello!”;
char word[]={“H’,’e’,’l’,’l’,’o’,’ !’,’\0’};
char word[7]={“Hello!”};
char word[6]={“Hello!”};
```

- Displaying character strings

```
printf(“Hello!”);
printf(“%s”,word);
```

# The Null String

- A character string that contains no characters other than the null character has a special time  in the C language, it is called the *null string.*
- The length of the null string is *zero.*
- In C , the null string is denoted by an adjacent pair of double quotation , so the statement

  char buffer[100]=“ “;

  defines a character array called buffer and sets its value to the null string.
- The character string “ “ is not the same as the character string “   “

# Function

- A **word with a pair of parenthesis** is called a function. Some of the statements like clrscr(), getch(), exit(), printf() and scanf() are the words with parenthesis, so called functions.
- Even the main() is a word with parenthesis, so it is also called as a function
- A statement or statements are defined with a pair of braces is called function body.
- **Types of functions**
  1. Predefined function – printf() ,scanf(), getch(), clrscr(), sqrt() and pow()
  2. User defined functions – main() etc

# Functional parts of a function

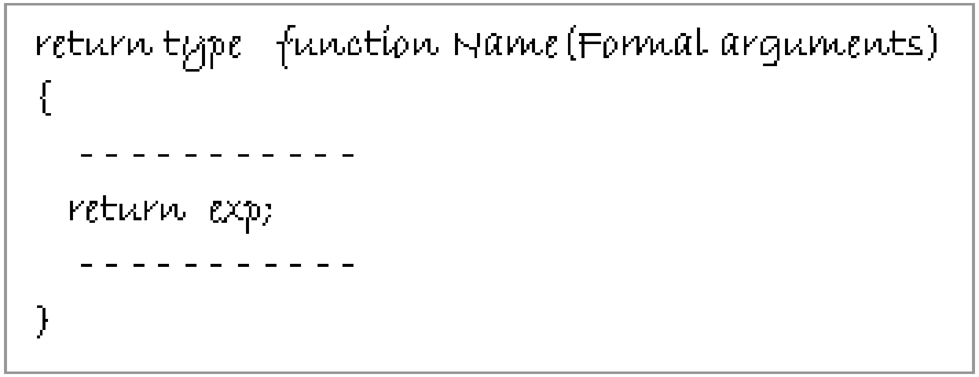
- Defining and using of any function required three things.
    1. Function declaration.
    2. Function calling statement.
    3. Function definition.

# Function definition

- Any function has a name, which must be a valid identifier.
- List of formal arguments are defined, the values of which are assigned by the actual arguments of calling statement.
- Body of the function is defined with in { }
- “return” is the statement used to return maximum a single value to the calling function.
- “return” is also used to terminate the execution of a function.
- Return type is specified if the function returns a non-integer, **void** must be specified if a function doesn’t return any value.

# Function definition

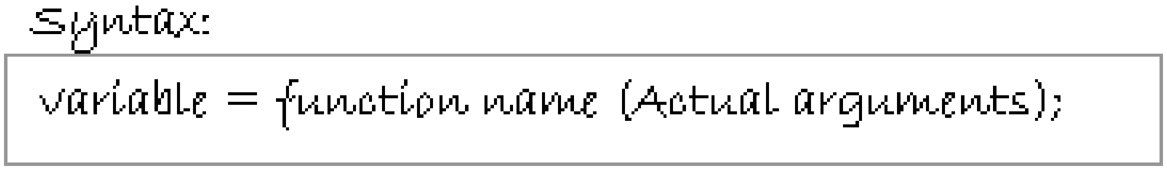
Syntax



```
return type   function Name(Formal arguments)
{
   - - - - - - - - - - - -
  return  exp;
   - - - - - - - - - - - -
}
```

# Function calling statement

- It is the statement initiates the execution of a function.
- List of actual arguments are specified to send arguments to the function definition.
- Here the type, number and sequence of actual arguments must be equal to the type, number and sequence of formal arguments.
- The calling statement is assigned to the (l-value) variable if the function returns a value.



Syntax:

```
variable = function name (Actual arguments);
```

# Function declaration/Prototype

- By default C compiler considers arguments and return type as integers.
- If we use other types as arguments and return value then the behavior of function must be informed to the compiler by writing prototype.
- Prototype is also called function declaration statement can be defined either before or within the calling statement.

# Functional parts of a function

- It makes the compiler job easy to identify and understand a function.

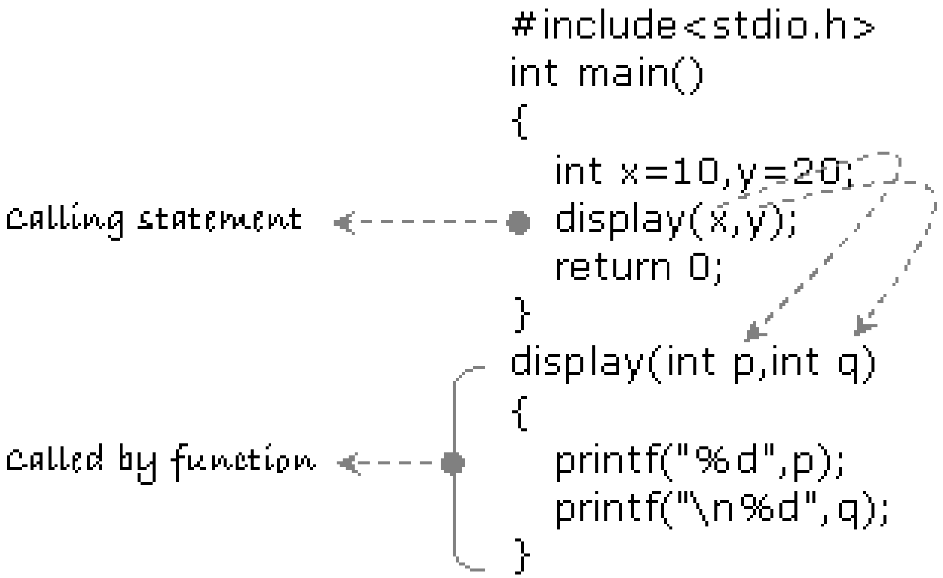
```
#include<stdio.h>
void display(int,int,int);
int main()
{
int x=10;
display(++x,++x,x++);
return 0;
}
void display(int p,int q,int r)
{
printf("%d\t%d\t%d",p,q,r);
}
```

Output:

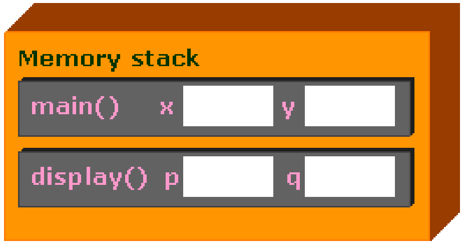
```
13   12   10
```

# Function

- A list of variables is specified with the function calling statement called **actual arguments**.
- Similar list of variables in terms of number and type are specified with the called by function called **formal arguments**.



```
#include<stdio.h>
int main()
{
   int x=10,y=20;
   display(x,y);
   return 0;
}
display(int p,int q)
{
   printf("%d",p);
   printf("\n%d",q);
}
```



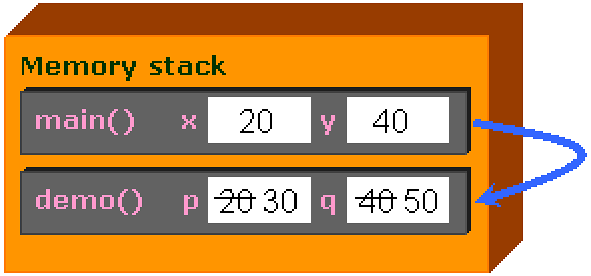
//X and y are holding actual values

//values of p, q are assigned by the actual arguments

Output:
10
20

# Pass by value

- Only the values of actual arguments are assigned to the formal arguments.
- The change in formal arguments doesn’t make any change in actual arguments because actual and formal arguments are different memory locations.



```
#include<stdio.h>
int main()
{
  int x=20,y=40;
  demo(x,y); /* sending arguments */
  printf("x=%d",x);  /* printing x, y after demo() execution */
  printf("\ny=%d",y);
  return 0;
}
demo(int p,int q)
{
  p=p+10;  /* modifying formal arguments */
  q=q+10;
}
```

Output:

```
x=20
y=40
```

# Returning a value

- **return** is the keyword used to interrupt the function execution and send the control back to the calling function.

```
#include<stdio.h>
int main()                  /* calling function */
{
  display();
}
 void display()
{
  printf("One");
  printf("\nTwo");
  return;                  /* termination point */
  printf("\nThree");
  printf("\nFour");
}
```

```
Output:
One
Two
```

# Returning a value

- The same return keyword is also used to return a value from the function
- E.g

```
return 5;                /* returns 5 */
return x;                /* returns the value of x */
return 2*(x+y);   /* returns the result of expression */
```

```
#include<stdio.h>
int main()
{
  int x;
  x=december();/* returned value assigns to x */
  printf("Last month %d",x);
  return 0;
}
december()
{
  return 12;  /* returns 12 */
}
```

Output:
```
Last month 12
```

# Category of Functions - Demo

- Functions can be categorized into
  - No arguments, no return value
  - Arguments , no return value
  - No arguments, return value
  - Arguments, return value

# Headers

- Each standard library has a corresponding header
- It contains
  - the function prototypes for all the functions in that library
  - definitions of various data types and constants needed by those functions
- You can create custom headers
  - Programmer-defined headers should also use the .h filename extension
  - A programmer-defined header can be included by using the #include preprocessor directive

# Math Library functions

- Allow you to perform certain common mathematical calculations
- Example, a programmer desiring to calculate and print the square root of 900.0 might write

  ```
  printf( "%.2f", sqrt( 900.0));
  ```

  - When this statement executes, the math library function sqrt is called
  - The number 900.0 is the argument of the sqrt function
  - The preceding statement would print 30.00
  - The sqrt function takes an argument of type double and returns a result of type double

# Math Library functions

- Include the math header by using the preprocessor directive #include <math.h> when using functions in the math library

- Function arguments may be constants, variables, or expressions
  - If c1 = 13.0, d = 3.0 and f = 4.0, then the statement

```
printf( "%.2f", sqrt( c1 + d * f ) );
```

calculates and prints the square root of 13.0 + 3.0 *4.0 = 25.0, namely 5.00

# Arrays and Functions

- Name of array is constant storing the address of first element

- Function prototype

```
 void myFunction(int [], int);
```

- Function definition

```
void myFunction(int myArray[], int myArraySize){
    …
}
```

# Arrays and Functions contd.

- C automatically passes arrays to functions by reference

- Passing arrays
  - Specify array name without brackets

```
int myArray[32];
myFunction(myArray,32);
```

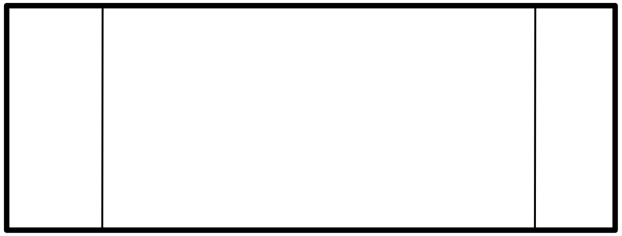
    - Array size usually passed to function, unlike char array no special terminator

- Passing array elements
  - Subscripted name in function call

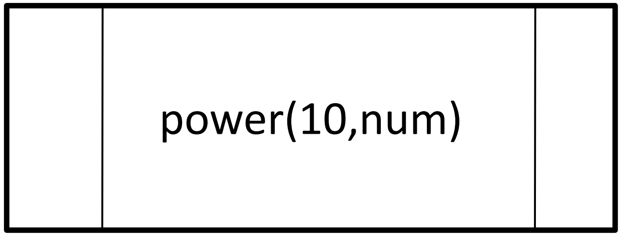
```
myFunction(myArray[10]);
```

# Flow Charts and Algorithms

- Predefined Process



- Examples



- Use function call
- For each function, define separate algorithm
- Examples
  - add(a,b);

```
Algorithm add
    (a,b:Integer):Integer
var c:Integer; {The result}
Begin
    c := a+b;
End
```

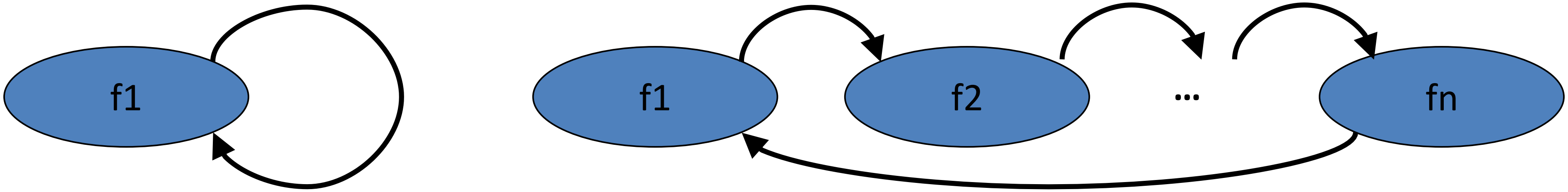
# Why functions?

- To develop a complex and big application, the total application has to divide into small and easily manageable parts called modules.
- A module is a part or unit of total application.
- In C language a module is defined using function.
- C language supports modularity using functions so is called Structure programming language.

# Recursion

- C functions can be used recursively
  - A function may call itself either directly or indirectly

- When a function calls itself recursively, each invocation gets a fresh set of all the automatic variables, independent of the previous step



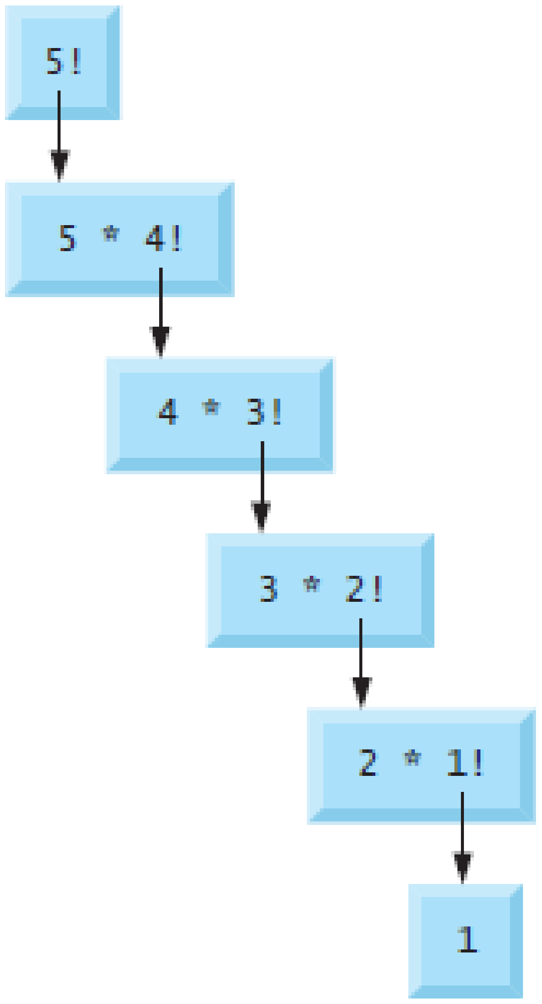
# Factorial: In Mathematics and C

$$N! = \begin{bmatrix} 1 & \text{if } N = 0 \\ N*(N-1)! & \text{if } N > 0 \end{bmatrix}$$

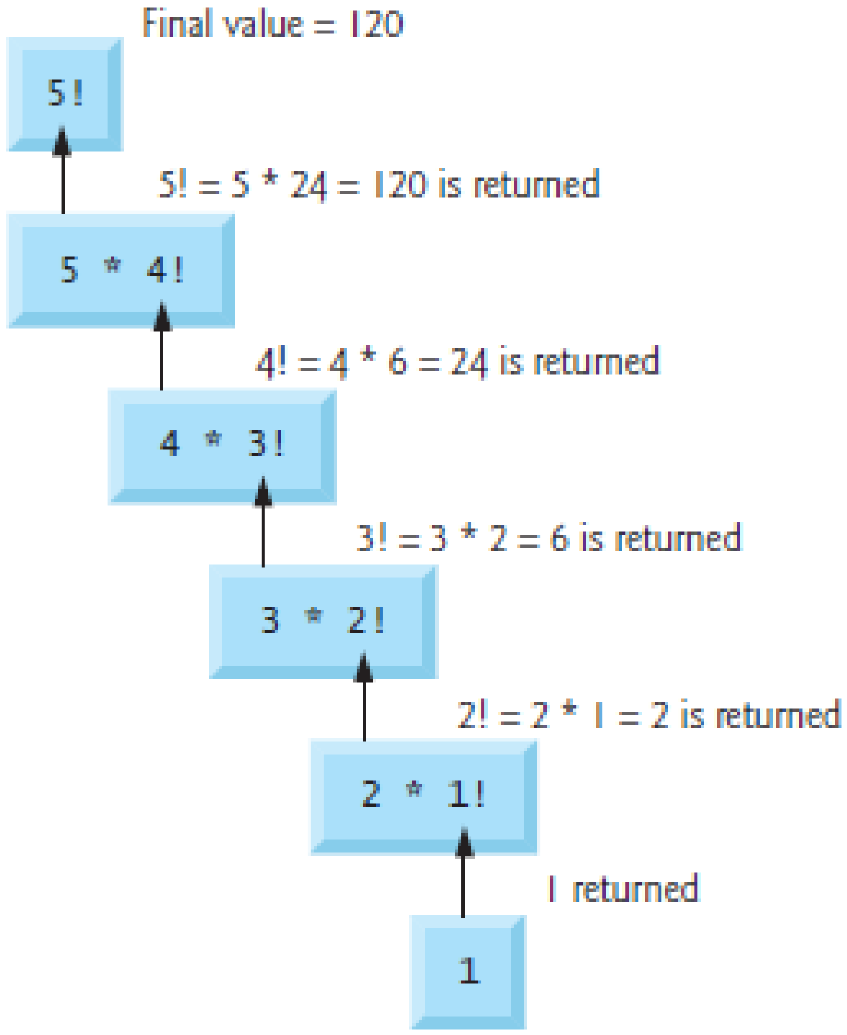
Example using Definition

```
4! = 4 * 3!
   = 4 * 3 * 2!
   = 4 * 3 * 2 * 1!
   = 4 * 3 * 2 * 1 * 0!
   = 4 * 3 * 2 * 1 * 1
```

# Recursion - Example



(a) Sequence of recursive calls.


Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 returned
1

(b) Values returned from each recursive call.

# Recursive Function

- Determine the base case(s)
  - the one for which you know the answer
  - There must be a terminating condition

- Determine the general case(s)
  - the one where the problem is expressed as a smaller version of itself

# Factorial Using Recursion

```
Algorithm fact(x :Integer):Integer
var xFactorial :Integer; {The factorial}
begin
        {assert x=>0}
        If (x <= 1) then
        begin
                xFactorial := 1;
        end
        else    {if it is greater than 1}
        begin
            xFactorial  := x * fact ( x-1 );
        end
end
```

# Equivalence with Iteration

- All recursive functions have iterative equivalents
- Example:

```
Algorithm fact(x :integer):Integer
Var iLoop, xFactorial :Integer; {The factorial}
begin
        {assert x=>0}
        xFactorial := 1;
        for iLoop in 1 to n, step 1 do
        begin
            xFactorial  := xFactorial * iLoop;
        end
end
```

# Iteration v/s Recursion

***Iteration***

- Uses repetition structures such as for,while loops.
- It is counter controlled and the body of the loop terminates when the termination condition is failed.
- They execute much faster and occupy less memory and can be designed easily.

***Recursion***

- Uses selection structures such as if-else,switch statements.
- It is terminated when a base condition.The terminal condition is gradually reached by invocation of the same function repeatedly.
- It is expensive in terms of processor time and memory usage.

# Merits and Demerits

- Merits
  - More compact
  - Easier to write and understand than the non-recursive equivalent

- Demerits
  - Take more time compared to iterative methods
  - A stack of the values being processed must be maintained
  - Consume additional memory

# Why Avoid Recursion?

- What happens when the value of n increases?
- How many variables will be there in memory?

# Recursion

```
#include<stdio.h>
int factorial(int);
int main()
{
 int n;
 printf("Enter an integer:");
 scanf("%d",&n);
 fact=factorial(n);
 printf("Factorial of the number %ld",fact);
 return 0;
}
int factorial(int n)
{
 int f;
 if(n==1)
  return 1;
 f=n*factorial(n-1);
 return f;
}
```

**Output:**

```
Enter an integer:3
Factorial of the number: 6
```

# Summary

- An array lets you declare and work with a collection of values of the same type.
- One of the feature about array indexing is that, you can use a loop to manipulate the index.
- Two dimensional  arrays
- Strings in C are an array of characters terminated with a  null character, '\0',.
- A function can be thought of as a mini-program, where you can pass in information, execute a group of statements and return an optional value.
- Recursion is the name given for expressing anything in terms of itself. A function which contains a call to itself or a call to another function.

# ANY QUERIES