

UNIT 2

Arrays

User-Defined Functions

String

CHAPTER-1 ARRAYS

TOPICS: Arrays (1-D, 2-D) Initialization and Declaration.

ARRAYS

- An **Array** is a *group of related data elements* that share a *common name*.
- *Data element* may be *int, float, char* or *double*. All these elements are stored in consecutive memory locations (in RAM).
- An **Array** is defined by a *single name* (identifier).
- Each individual data item is identified by a subscript or index, each *set of subscript* is enclosed in *square bracket*.
- Subscript specifies position of data item in array, and subscript must be an unsigned positive integer.
Hence arrays are also called *subscripted variables*.

a[0]	a[1]	a[2]	a[3]	a[4]
------	------	------	------	------

Where,

0,1, 2, 3, and 4 are **subscripts**.

a[0] indicates the *First* element in an array *named a*

a[1] indicates the *Second* element in an array *named a*

In general, **a[i]** indicates the *(i - 1)th* element in an array *named a*

- **Array** must be declared before it appears in the program, using the *data type*. At the same time the *size* of an array must be specified. This makes the compiler to know how much memory must be reserved for the array.

Classification of Arrays:

➤ **One dimensional** array.

➤ **Multi dimensional** arrays

- **Two dimensional** array
- **Three dimensional** array ...
- **“n” dimensional** array

- The **dimensionality** is determined by the number of subscripts in an array. If there is *one set of subscript*, then it is called **one dimensional**. If there are *two sets of subscripts* then it is called **two dimensional** and so on.

One Dimensional Array:

- It is a linear list of same data type with a single name.
- All the data items can be accessed by using the same name using a single set of subscript.

Declaration:

Data_type Array_name[Size] ;

Data_type : may be int, float, char or double.

Array_name : is an identifier which specifies the name of array.

Size : **maximum number of elements in array. It is an integer constant.**

Example:

float	height[10];
int	num[50];
char	name[20];
double	a[80];

Rules for subscripts:

- Each subscript must be a positive integer constant, or an integer expression.
- Subscript of subscript is not allowed.
- ‘C’ does not perform bounds checking, therefore, the maximum value of subscript should not cross the value declared.
- Subscript value ranges from 0 to one less than the maximum size.

int num[5];

num[0], num[1], num[2], num[3], num[4]

i = 0 i = 1 i = 2 i = 3 i = 4

- The *size* in **character** string represents the maximum number of characters that a *string* can hold.

char a[10];

Suppose **a[10]** has a value “WELL DONE”, then each character of the string is treated as an element of the array **a** and is stored in the memory as follows

‘W’	‘E’	‘L’	‘L’	‘ ’	‘D’	‘O’	‘N’	‘E’	‘\0’
-----	-----	-----	-----	-----	-----	-----	-----	-----	------

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] When

the compiler sees a *character* string, it *terminates with an additional null character*. Thus **a[9]** holds the null character **\0** at the end. Therefore when declaring character arrays, we must always allow one extra element space for the *null terminator*.

- The **total memory** that can be allocated to *one dimensional array* is computed as,

Total size = size * (sizeof (data_type)) ;

size : number of elements in one dimensional array

sizeof() : is an *unary operator* to find the size in bytes.

data_type : basic data types like int, float, char or double.

int=2 bytes, float =4, char=1, double=8.

Example: the total size of one dimensional array of 10 integers is 20 bytes.

Initialization of Arrays:

- Initialization is nothing but assigning some value to the variable that undergoes processing. Like initialization to ordinary variable, the *individual* elements of an array can also be initialized.
- All the initial values must be constant. But never be variables or function calls.
- Array elements can be initialized at the time of declaration.

Syntax of one dimensional array initialization statement:

Data_type array_name[size] = {ele1, ele2, ...}

Data_type may be int, float, char . . .

array_name is the name of the array.

Different ways of initializing arrays:

Size is the maximum number of elements.

➤ Initializing all specified memory locations.

ele1, ele2,... are the initial values enclosed within '{' and '}' separated by commas and must be

➤ Partial array initialization.

written in order in which they will be assigned.

➤ Initialization without size.

➤ String Initialization.

Initializing all specified memory locations:

Arrays can be initialized at the time of declaration when their initial values are known in advance.

Array elements can be initialized with data items of type int, float, char etc

Example: **int num[5] = {10, 15, 20, 25, 30};**

- During compilation, 5 memory locations are reserved by the compiler for the variable num and all these locations are initialized.

num[0]	num[1]	num[2]	num[3]	num[4]
10	15	20	25	30

- If the size of integer is 2 bytes, 10 bytes will be allocated for the variable num.
- If the number of initial values are *more* than the *size of array*, then compiler will give **error**.

Partial array initialization:

If the number of values to be initialized is less than the size of the array, then the elements are initialized in order from 0th location. The remaining locations will be initialized to zero automatically.

Example: **float total[5] = {0.0, 15.75, -10};**

Even though compiler allocates 5 memory locations, using this declaration statement, the compiler initializes first three locations with 0.0, 15.75 and -10.0. The next set of memory locations are automatically initialized to 0's by the compiler.

total[0]	total[1]	total[2]	total[3]	total[4]
0.0	15.75	-10.0	0.0	0.0

Initialization without size:

Example: **int counter[] = {1,1,1,1};**

Will declare the **counter** array to contain total number of initial values specified i.e four elements with initial values 1,1,1,1.

counter[0]	counter[1]	counter[2]	counter[3]
1	1	1	1

String Initialization:

Example: **char a[] = "COMPUTER";**

The array **a** is initialized as

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
C	O	M	P	U	T	E	R	\0

Even though the string "COMPUTER" contains 8 characters, because it is a string, it always ends with null character. So array size is 9 bytes.

Reading / Writing single dimensional arrays:

Consider the declaration `int a[5];` Here, FIVE memory locations are reserved and each item in the memory location can be accessed by specifying the index as `a[0]` through `a[4]`. In general, using `a[0]` through `a[n-1]` we can access *n data items*.

Data items can be stored by reading the *n data items* from the keyboard using `scanf()` function as

```
scanf ("%d", &a[0]);
scanf ("%d", &a[1]);
scanf ("%d", &a[2]);
. . . . .
. . . . .
. . . . .
scanf ("%d", &a[n-1]);
```

```
for(i=0; i<=n-1; i++)
{
    scanf ("%d", &a[i]);
}
```

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

Similarly, *n data items* stored in the array can be displayed as

```
printf ("%d", a[0]);
printf ("%d", a[1]);
printf ("%d", a[2]);
. . . . .
. . . . .
. . . . .
printf ("%d", a[n-1]);
```

```
for(i=0; i<=n-1; i++)
{
    printf ("%d", a[i]);
}
```

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

Multi Dimensional Array:

If the number of subscripts is more than one, then such arrays are called *Multi Dimensional Arrays*. The dimensionality is understood from number of pairs of square brackets ahead of array name.

Example :	Array1[]	One – dimensional Array
	Array2[][]	Two – dimensional Array
	Array3[][][]	Three – dimensional Array

Two Dimensional Array:

- Arrays with two sets of square brackets `[][]` are called *two – dimensional arrays*.
- A two – dimensional array is used when data items are arranged in row–wise and column–wise in a tabular or *matrix* form.
- To identify a particular item we have to specify two indexes (also called subscripts): the *first index* specifies the **row number** and *second index* specifies the **column number**.

Declaration:

- A two – dimensional array must be declared before it is use along with size of the array.
- The size used during declaration of the array is useful to reserve the specified memory locations.

Syntax of Two dimensional array declaration statement:

```
Data_type Array_name[row_size][column_size];
```

Data_type : may be int, float, char or double.

Array_name : is an identifier which specifies the name of array.

Size : **maximum number of elements in array. It is an integer constant.**

Example: **int a[3][4];**

The array **a** is a *two dimensional array* of integer type with **3 rows** and **4 columns**. This declaration informs the compiler to reserve 12 locations ($3*4 = 12$) continuously one after the other.

	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]

Initialization of Arrays:

- Initialization is nothing but assigning some value to the variable that undergoes processing. Like initialization to one dimensional array, the *individual* elements of two dimensional array can also be initialized.
- All the initial values must be constant. But never be variables or function calls.
- Array elements can be initialized at the time of declaration.

Syntax of two dimensional array initialization statement:

```
Data_type array_name[size1] [size2] = {ele1, ele2, ...}
```

Data_type may be int, float, char . . .

array_name is the name of the array.

Size1, Size2 are the maximum number of rows and columns respectively.

ele1, ele2,... are the initial values enclosed within ‘{‘ and ‘}’ separated by commas and must be written in order in which they will be assigned.

Different ways of initializing arrays:

- Initializing all specified memory locations.
- Partial array initialization.

Initializing all specified memory locations:

Arrays can be initialized at the time of declaration when their initial values are known in advance.

Array elements can be initialized with data items of type int, float, char etc

Example:

```
int a[4][3] = {
```

{1, 2, 3},
{4, 5, 6},
{7, 8, 9},
{10, 11, 12}

```
};
```

or

int a[4][3] = {{1,2,3},{4,5,6},{7,8,9},{10,11,12}};

or

int a[4][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};

		Columns		
		0	1	2
Rows	0	1	2	3
	1	4	5	6
	2	7	8	9
	3	10	11	12

Here, **a[0][0] = 1**, **a[2][1] = 8**, **a[3][2] = 12** and so on.

Partial array initialization: If the number of values to be initialized is less than the size of the array, then the elements are initialized from left to right one after the other. The remaining locations will be initialized to zero automatically.

Example: **float xy[2][2] = {0.0, 1.5, 2.0};**

Here, **xy** is a two dimensional array of 2x2 elements of type float. But, only first three elements are initialized. The fourth element becomes zero.

xy[0][0] = 1.0, **xy[0][1] = 1.5,**

xy[1][0] = 2.0 **xy[1][1] = 0.0**

Example: **int num[4][3] = {{1,2},{3,4},{5,6},{7,8}};**

This declaration indicates that the array **num** has **4 rows** and **3 columns** in which only first 2 columns of each row are initialized. The 3rd column of each row will be initialized with zero automatically.

Columns

		0	1	2
Rows	0	1	2	0
	1	3	4	0
	2	5	6	0
	3	7	8	0

Here, **num[0][2]**, **num[1][2]**, **num[2][2]**, **num[3][2]** are initialized with zero.

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

In this array declared each row should have three columns. But, here we are trying to initialize 1st row with 4 elements. This results in *syntax error* and compiler flashes an **error**.

Example: `a[][] = { };` This results in *syntax error* and compiler flashes an **error**.

Points to be remembered while initializing the two – dimensional array:

- The number of sets of initial values must be equal to the number of rows in array.
- One to One mapping is preserved. i.e the first set of initial value is assigned to first row elements and second set of initial value is assigned to the second element and so on.
- If the number of initial values in each initializing set is less than the number of the corresponding row elements, then all the elements of that row are automatically assigned to zeros.
- If the number of initial values in each initializing set exceeds the number of the corresponding row elements then there will be compilation error.

Memory map of two – dimensional array:

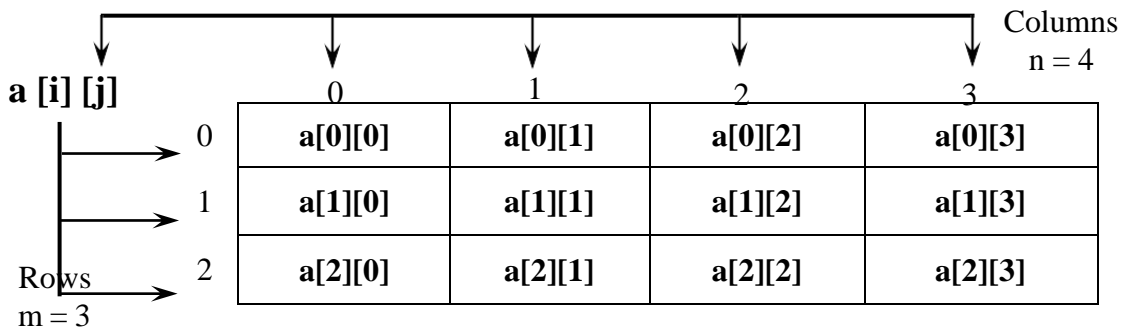
Example: `int a[3][2] = { {1, 2},`
 `{3, 4},`
 `{5, 6},`
 `};`

Even though we represent a matrix pictorially as a 2–dimensional array, in memory they are stored continuously one after the other. Assume that address of first byte of **a** [usually called base address] is 2000 and size of integer is 2 bytes. The memory map of 2–dimensional array **a** is

$a[0][0]$	$a[0][1]$	$a[1][0]$	$a[1][1]$	$a[2][0]$	$a[2][1]$
1	2	3	4	5	6
2000	2002	2004	2006	2008	2010

Reading / Writing two – dimensional array:

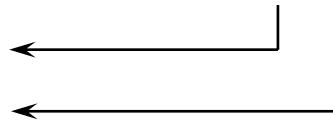
Any item can be accessed by specifying the row index and column index. Normally we access the items row by row as shown below.



Note that any element can be accessed by specifying $a[i][j]$

Index variable $i = 0, 1, 2$ that is $(m - 1)$

Index variable $j = 0, 1, 2, 3$ that is $(n - 1)$



To read / write each item *row wise* in 2–dimensional array, the *row index i* should be in the *outer loop* and *column index j* should be in the *inner loop*.

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

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

CHAPTER-2 User-defined Functions

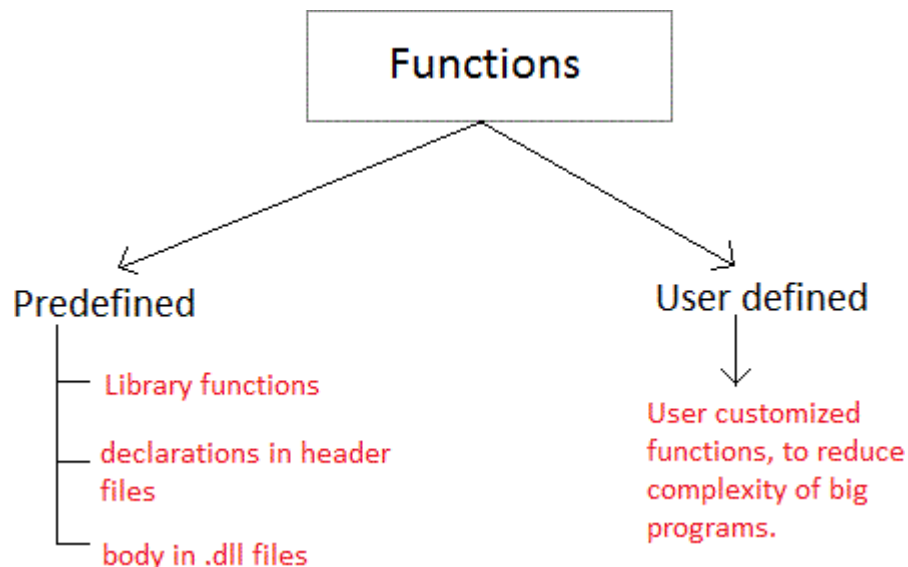
A function is a self-contained block of code that performs a particular task.

There are many situations where we might need to write same line of code for more than once in a program. This may lead to unnecessary repetition of code, bugs and even becomes boring for the programmer. So, C language provides an approach in which you can declare and define a group of statements once in the form of a function and it can be called and used whenever required.

These functions defined by the user are also known as User-defined Functions

C functions can be classified into two categories,

1. Library functions
2. User-defined functions



Library functions are those functions which are already defined in C library, example `printf()`, `scanf()`, `strcat()` etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

A User-defined functions on the other hand, are those functions which are defined by the user at the time of writing program. These functions are made for code reusability and for saving time and space.

BENEFITS OR NEED OF USING FUNCTIONS

1. It provides modularity to your program's structure.
2. It makes your code reusable. You just have to call the function by its name to use it, wherever required.
3. In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
4. It makes the program more readable and easy to understand.
5. There are times when some types of operation or calculation is repeated at many points throughout a program. So functions save both time and space.

A function may be used by many other programs, this means that a C program can build on what others have already done, instead of starting over, from scratch.

ELEMENTS OF USER-DEFINED FUNCTIONS

There are three elements related to functions

- Function definition
- Function call
- Function declaration

Function Declaration:

General syntax for function declaration is,

returntype functionName(type1 parameter1, type2 parameter2,...);

Like any variable or an array, a function must also be declared before its used. Function declaration informs the compiler about the function name, parameters is accept, and its return type. The actual body of the function can be defined separately. It's also called as **Function Prototyping**.

Function declaration consists of 4 parts.

- returntype
- function name
- parameter list
- terminating semicolon

returntype

When a function is declared to perform some sort of calculation or any operation and is expected to provide with some result at the end, in such cases, a return statement is added at the end of function body. Return type specifies the type of value(int, float, char, double) that function is expected to return to the program which called the function.

Note: In case your function doesn't return any value, the return type would be void.

functionName

Function name is an identifier and it specifies the name of the function. The function name is any valid C identifier and therefore must follow the same naming rules like other variables in C language.

parameter list

The parameter list declares the type and number of arguments that the function expects when it is called. Also, the parameters in the parameter list receives the argument values when the function is called. They are often referred as **formal parameters**.

Function definition:

The function definition is an independent program module that is specially written to implement the requirements of the function.

The general syntax of function definition is,

```
returntype functionName(type1 parameter1, type2 parameter2,...)
{
    // function body goes here
}
```

The first line *returntype* **functionName(type1 parameter1, type2 parameter2,...)** is known as **function header** and the statement(s) within curly braces is called **function body**.

Note: While defining a function, there is no semicolon(;) after the parenthesis in the function header, unlike while declaring the function or calling the function.

functionbody

The function body contains the declarations and the statements(algorithm) necessary for performing the required task. The body is enclosed within curly braces { ... } and consists of three parts.

- **local** variable declaration(if required).
- **function statements** to perform the task inside the function.
- a **return** statement to return the result evaluated by the function(if return type is void, then no return statement is required).

Calling a function:

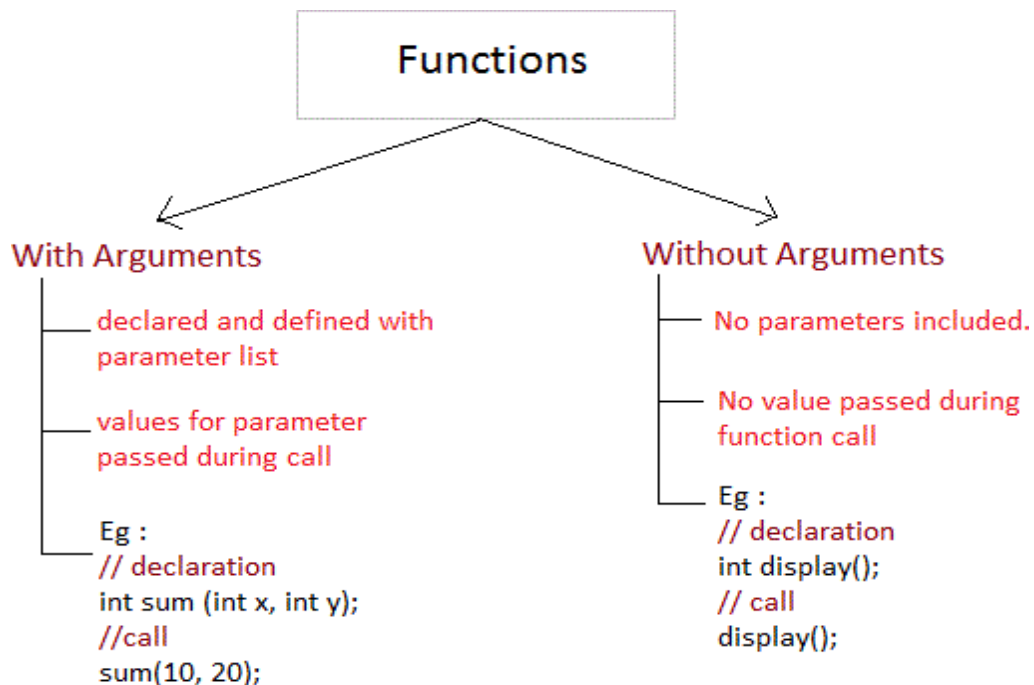
When a function is called, control of the program gets transferred to the function.

functionName(argument1, argument2,...);

In the example program, the statement multiply(i, j); inside the main() function is function call.

PASSING ARGUMENTS TO A FUNCTION

Arguments are the values specified during the function call, for which the formal parameters are declared while defining the function.




It is possible to have a function with parameters but no return type. It is not necessary, that if a function accepts parameter(s), it must return a result too.

```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ..
    result = multiply(i, j);
    ... ..
}

int multiply(int a, int b)
{
    ... ..
}
```



providing arguments while calling function

While declaring the function, we have declared two parameters a and b of type int. Therefore, while calling that function, we need to pass two arguments, else we will get compilation error. And the two arguments passed should be received in the function definition, which means that the function header in the function definition should have the two parameters to hold the argument values. These received arguments are also known as **formal parameters**. The name of the variables while declaring, calling and defining a function can be different.

Returning a value from function:

A function may or may not return a result. But if it does, we must use the return statement to output the result. return statement also ends the function execution, hence it must be the last statement of any function. If you write any statement after the return statement, it won't be executed.

```


#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ..
    result = multiply(i, j);
    ... ..
}

int multiply(int a, int b)
{
    ... ..
    return a*b;
}

```



The value returned by the function must be stored in a variable.

The datatype of the value returned using the return statement should be same as the return type mentioned at function declaration and definition. If any of it mismatches, you will get compilation error.

RETURN VALUES AND THEIR TYPES

A function may or may not send back any value to the calling function or The called function can only return one value per call.

SYNTAX:

return;

or

return (expression);

return;

–Plain return does not return any value

- Acts as the closing brace of the function
- The control is immediately passed back to the calling function

EXAMPLE:

```
if(error)
return;
```

return (expression);

- Return the value of the expression

EXAMPLE:

```
mul(int x,int y)
{
int p;
p = x*y;
return(p);
}
```

A function may have more than one return statements.

EXAMPLE:

```
fun()
{
if(x <= 0)
return(0);
else
return(1);
}
```

There will be times when we may find it necessary to receive the character, float or double type of data. We must explicitly specify the return type on both the function definition and the prototype declaration. The type-specifier tells the compiler, the type of data the function is to return. The called function must be declared at the start of the body in the calling function, like any other variable. This is to tell the calling function the type of data that the function is actually returning.

TYPES OF FUNCTION CALLS IN C

Functions are called by their names, Well if the function does not have any arguments, then to call a function you can directly use its name. But for functions with arguments, we can call a function in two different ways, based on how we specify the arguments, and these two ways are:

1. Call by Value
2. Call by Reference

Call by Value:

Calling a function by value means, we pass the values of the arguments which are stored or copied into the formal parameters of the function. Hence, the original values are unchanged only the parameters inside the function changes.

```
#include<stdio.h>
void calc(int x);

int main()
{
    int x = 10;
    calc(x);
    // this will print the value of 'x'
    printf("\nvalue of x in main is %d", x);
    return 0;
}

void calc(int x)
{
    // changing the value of 'x'
    x = x + 10 ;
    printf("value of x in calc function is %d ", x);
}
```

In this case, the actual variable x is not changed. This is because we are passing the argument by value, hence a copy of x is passed to the function, which is updated during function execution, and that copied value in the function is destroyed when the function ends(goes out of scope). So the variable x inside the main() function is never changed and hence, still holds a value of 10.

But we can change this program to let the function modify the original x variable, by making the function calc() return a value, and storing that value in x.

```
#include<stdio.h>
int calc(int x);

int main()
{
    int x = 10;
    x = calc(x);
    printf("value of x is %d", x);
    return 0;
}

int calc(int x)
{
    x = x + 10 ;
    return x;
}
value of x is 20
```

Call by Reference:

In call by reference we pass the address(reference) of a variable as argument to any function.

When we pass the address of any variable as argument, then the function will have access to our variable, as it now knows where it is stored and hence can easily update its value.

In this case the formal parameter can be taken as a **reference** or a **pointer**(don't worry about pointers, we will soon learn about them), in both the cases they will change the values of the original variable.

```

#include<stdio.h>
void calc(int *p);    // functin taking pointer as argument

int main()
{
    int x = 10;
    calc(&x);    // passing address of 'x' as argument
    printf("value of x is %d", x);
    return(0);
}

void calc(int *p)    //receiving the address in a reference pointer variable
{
    /*
        changing the value directly that is
        stored at the address passed
    */
    *p = *p + 10;
}

```

value of x is 20

CATEGORIES OF USER-DEFINED FUNCTIONS IN C

There can be 4 different types of user-defined functions, they are:

1. Function with no arguments and no return value
2. Function with no arguments and a return value
3. Function with arguments and no return value
4. Function with arguments and a return value

Function with no arguments and no return value:

Such functions can either be used to display information or they are completely dependent on user inputs.

Below is an example of a function, which takes 2 numbers as input from user, and display which is the greater number.

```
#include<stdio.h>
void greatNum();    // function declaration

int main()
{
    greatNum();    // function call
    return 0;
}

void greatNum()    // function definition
{
    int i, j;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    if(i > j) {
        printf("The greater number is: %d", i);
    }
    else {
        printf("The greater number is: %d", j);
    }
}
```


Function with no arguments and a return value:

We have modified the above example to make the function `greatNum()` return the number which is greater amongst the 2 input numbers.

```
#include<stdio.h>
int greatNum();    // function declaration

int main()
{
    int result;
    result = greatNum();    // function call
    printf("The greater number is: %d", result);
    return 0;
}

int greatNum()    // function definition
{
    int i, j, greaterNum;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    if(i > j) {
        greaterNum = i;
    }
    else {
        greaterNum = j;
    }
    // returning the result
    return greaterNum;
}
```

Function with arguments and no return value:

We are using the same function as example again and again, to demonstrate that to solve a problem there can be many different ways.

This time, we have modified the above example to make the function `greatNum()` take two int values as arguments, but it will not be returning anything.

```

#include<stdio.h>
void greatNum(int a, int b);    // function declaration

int main()
{
    int i, j;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    greatNum(i, j);    // function call
    return 0;
}

void greatNum(int x, int y)    // function definition
{
    if(x > y) {
        printf("The greater number is: %d", x);
    }
    else {
        printf("The greater number is: %d", y);
    }
}

```

Function with arguments and a return value:

This is the best type, as this makes the function completely independent of inputs and outputs, and only the logic is defined inside the function body.

```

#include<stdio.h>
int greatNum(int a, int b);    // function declaration

int main()
{
    int i, j, result;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    result = greatNum(i, j); // function call
    printf("The greater number is: %d", result);
}

```

```

    return 0;
}

int greatNum(int x, int y)    // function definition
{
    if(x > y) {
        return x;
    }
    else {
        return y;
    }
}

```

Nesting of Functions:

C language also allows nesting of functions i.e to use/call one function inside another function's body. We must be careful while using nested functions, because it may lead to infinite nesting.

```

function1()
{
    // function1 body here

    function2();

    // function1 body here
}

```

If function2() also has a call for function1() inside it, then in that case, it will lead to an infinite nesting. They will keep calling each other and the program will never terminate.

Not able to understand? Lets consider that inside the main() function, function1() is called and its execution starts, then inside function1(), we have a call for function2(), so the control of program will go to the function2(). But as function2() also has a call to function1() in its body, it will call function1(), which will again call function2(), and this will go on for infinite times, until you forcefully exit from program execution.

Command Line Arguments

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly

```
#include <stdio.h>

int main( int argc, char *argv[] ) {

    if( argc == 2 ) {
        printf("The argument supplied is %s\n", argv[1]);
    }
    else if( argc > 2 ) {
        printf("Too many arguments supplied.\n");
    }
    else {
        printf("One argument expected.\n");
    }
}
```

When the above code is compiled and executed with single argument, it produces the following result.

```
$/a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$/a.out testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$/a.out
One argument expected
```

It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and ***argv[n]** is the last argument. If no arguments are supplied, **argc** will be one, and if you pass one argument then **argc** is set at 2

Searching technique

- Searching technique refers to finding a key element among the list of elements.
- If the given element is present in the list, then the searching process is said to be successful.
- If the given element is not present in the list, then the searching process is said to be unsuccessful.
- C language provides two types of searching techniques. They are as follows –
 - Linear search
 - Binary search

1. Linear Search

- Searching for the key element is done in a linear fashion.
- It is the simplest searching technique.
- It does not expect the list to be sorted.
- Limitation – It consumes more time and reduce the power of system.
- Input (i/p) is unsorted list of elements, key.
- Output (o/p)
 - Success – If key is found.
 - Unsuccessful – Otherwise

```
#include <stdio.h>
void linsearch(int b[100],int n, int key);
int main()
{
    int i,n,a[100],key;
    printf("Enter the value of n\n");
    scanf("%d",&n);
    printf("Enter %d elements\n",n);
    for(i=0; i<n;i++)
    {
        if(b[i]==key)
        {
            printf("The key element is found at position %d",i+1);
            exit(0);
        }
    }
}
```

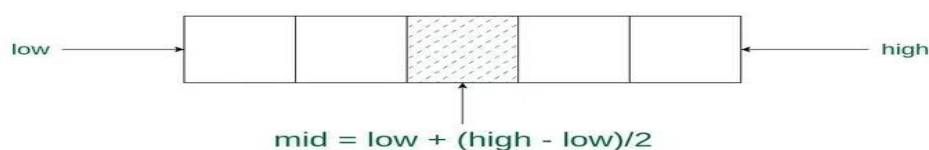
```
    printf("The Key element is not found in the list\n");
}
```

2. Binary search

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly **dividing the search interval in half**. The idea of binary search is to use the information that the array is sorted and reduce the time.

Binary Search Algorithm:

- In this algorithm it divides the search space into two halves by finding the middle index “mid”.



- Compare the middle element of the search space with the key.
- If the key is found at middle element, the process is terminated.
- If the key is not found at middle element, choose which half will be used as the next search space.
- If the key is smaller than the middle element, then the left side is used for next search.
- If the key is larger than the middle element, then the right side is used for next search.
- This process is continued until the key is found or the total search space is exhausted.
 - Input (i/p) is **sorted list** of elements, key.
 - Output (o/p)
 - Success – If key is found.
 - Unsuccessful – Otherwise

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
    int a[20],n,mid,l,h,i,key;
    printf("Enter the value for n\n");
    scanf("%d",&n);
    printf("enter %d elements in Ascending order \n",n);
```

```

for(i=0;i<=h)
{
    mid=(l+h)/2;
    if(a[mid]==key)
    {
        printf("The key if found at the position %d",mid+1);
        exit(0);
    }
    else if(key>a[mid])
    l=mid+1;
    else h=mid-1;
}
printf("The key is not found\n");
return 0;
}

```

Sorting Techniques

C language provides five sorting techniques, which are as follows –

- Bubble sort (or) Exchange Sort.
- Selection sort.
- Insertion sort (or) Linear sort.
- Quick sort (or) Partition exchange sort.
- Merge Sort (or) External sort.

1. Bubble sort

It is the simplest sorting technique which is also called as an exchange sort.

Procedure

- Compare the first element with the remaining elements in the list and exchange (swap) them if they are not in order.
- Repeat the same for other elements in the list until all the elements gets sorted.

```

#include<stdio.h>
#include<stdlib.h>
void ascending(int b[50],int n);

```

```

void descending(int b[50],int n);
int main()
{
    int n,i,j,a[100],ch;
    printf("Enter the value of N\n");
    scanf("%d",&n);
    printf("enter the %d elements\n",n);
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("Enter Your Choice\n");
    printf("1. Ascending order sorting\n 2. Descending order sorting\n");
    scanf("%d",&ch);
    if(ch==1)
        ascending(a,n);
    else

        descending(a,n);
    printf("The sorted array is\n");
    for(i=0;i<n;i++)
        printf("%d\t",a[i]);
    printf("Thank you\n");
    return 0;
}
void ascending (int b[30] , int n)
{
    int i,j,temp;
    for(i=0;i<n-1;i++)
        for(j=0;j<n-i-1;j++)
            if(b[j]>b[j+1])
            {
                temp=b[j];
                b[j]=b[j+1];
                b[j+1]=temp;
            }
}

```



```

}
void descending(int b[30],int n)
{
    int i,j,temp;
    for(i=0;i<n-1;i++)
    for(j=0;j<n-i-1;j++)
        if(b[j]<b[j+1])
        {
            temp=b[j];
            b[j]=b[j+1];
            b[j+1]=temp;
        }
}

```

2. Selection Sort

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    int a[50],n,temp,i,j,pos;
    printf("Enter the value of n\n");
    scanf("%d",&n);
    printf("Enter the numbers\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("Entered values are\n");
    for(i=0;i<n;i++)
        printf("%d\t", a[i]);
    for(i=0;i<n-1;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(a[j]<a[pos])
                pos=j;
        }
        temp=a[i];
        a[i]=a[pos];
    }
}

```

```

        a[pos]=temp;
    }
    printf("\nThe sorted values are\n");
    for(i=0;i<n;i++)
        printf("%d\t",a[i]);
    return 0;
}

```

Trace ad Transpose of Matrix

- The transpose of a matrix is a new matrix formed by interchanging its rows with columns. In simple words, the transpose of $A[i][j]$ is obtained by changing $A[i][j]$ to $A[j][i]$.
- The transpose of an $m \times n$ matrix will result in an $n \times m$ matrix.
- The trace of a square matrix is the sum of its diagonal entries.

Write a C program to transpose a matrix of order M x N and find the trace of the resultant matrix.

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i,j,m,n,a[10][10],b[10][10],trace=0;
    printf("Enter the size of the matrix\n");
    scanf("%d%d",&m,&n);
    printf("Enter %d x %d matrix\n",m,n);
    for(i=0;i<m;i++)
        for(j=0;j<n;j++)
            scanf("%d",&a[i][j]);
    printf("The entered matrix is\n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%d\t",a[i][j]);
        }
        printf("\n");
    }
    for(i=0;i<m;i++)
        for(j=0;j<n;j++)
            b[j][i]=a[i][j];
    printf("The transpose of the matrix is\n");
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
        {
            printf("%d\t",b[i][j]);
        }
    }
}

```

```

        }
        printf("\n");
    }
    if(m==n)
    {
        for(i=0;i<n;i++)
        {
            for(j=0;j<m;j++)
            {
                if(i==j)
                    trace=trace+b[i][j];
            }
        }
        printf("The trace is %d\n",trace);
    }
    else
        printf(" can't perform trace because it's not a square matrix\n");
    return 0;
}

```

Matrix Multiplication

Write a C program using functions to read two matrices A (M x N) and B (P x Q) and to compute the product of A and B if the matrices are compatible for multiplication.

```

#include <stdio.h>
#include <stdlib.h>
void read( int x[10][10], int r, int c);
void display(int x[10][10], int row, int col);
void multiply(int a[10][10],int b[10][10],int c[10][10],int r,int col,int p);
int main()
{
    int a[10][10],b[10][10],c[10][10],m,n,p,q,i,j,k;
    printf("Enter the order of matrix A\n");
    scanf("%d%d",&m,&n);
    printf("Enter the order of matrix B\n");
    scanf("%d%d",&p,&q);
    if(n!=p)
    {
        printf("not possible\n");
        exit(0);
    }
    printf("Enter the matrix A\n");
    read(a,m,n);
    printf("Enter the matrix B\n");
    read(b,p,q);

    printf("Entered matrix A is\n");
    display(a,m,n);
    printf("entered matrix B is\n");

```

```

display(b,p,q);
multiply(a,b,c,m,n,q);
printf("The resultant matrix is\n");
display(c,m,q);
return 0;
}
void read(int x[10][10], int r, int c)
{
    int i,j;
    for(i=0;i<r;i++)
    for(j=0;j<c;j++)
    scanf("%d",&x[i][j]);
}

void display(int x[10][10], int row, int col)
{
    int i,j;
    for(i=0;i<row;i++)
    {
        for(j=0;j<col;j++)
        {
            printf("%d\t",x[i][j]);
        }
        printf("\n");
    }
}

void multiply(int a[10][10],int b[10][10],int c[10][10],int m, int n, int q)
{
    int i, j, k;
    for(i=0;i<m;i++)
    {
        for(j=0;j<q;j++)
        {
            c[i][j]=0;
            for(k=0;k<n;k++)
            {
                c[i][j]=c[i][j]+a[i][k]*b[k][j];
            }
        }
    }
}

```

CHAPTER 3 Strings

Strings are sequence of characters. In C, there is no built-in data type for strings. String can be represented as a one-dimensional array of characters. A character string is stored in an array of character type, one ASCII character per location. String should always have a NULL character ('\0') at the end, to represent the end of string.

String constants can be assigned to character array variables. String constants are always enclosed within double quotes and character constants are enclosed within single quotes.

READING STRINGS (COMPILE TIME)

Examples:

(1) `char c[4]={ 's','u','m','\0' };`

(2) `char str[16]="qwerty";`

Creates a string. The value at `str[5]` is the character 'y'. The value at `str[6]` is the null character. The values from `str[7]` to `str[15]` are undefined.

(3) `char name[5];`

```
int main( )
{
    name[0] = 'G';
    name[1] = 'O';
    name[2] = 'O';
    name[3] = 'D';
    name[4] = '\0';
    return 0;
}
```

(4) `char name[5] = "INDIA"`

Strings are terminated by the null character, it is preferred to allocate one extra space to store null terminator.

String can be read either character-by-character or as an entire string (using %s format specifier). Array name itself specifies the base address and %s is a format specifier which will read a string until a white space character is encountered.

[Note: No need to use & operator while reading string using %s]

RUN TIME INITIALIZATION AND PRINTING

Example:

```
char name[20];
```

```

int i=0;
while((name[i] = getchar ()) != '\n' )
i++;
scanf( "%s" , name);
printf("%s" , name);

```

STRING MANIPULATION FUNCTIONS

C does not provide any operator, which manipulates the entire string at once. Strings are manipulated either using pointers or special routines (built-in) available from the standard string library string.h.

The following is the list of string functions available in string.h:

strcpy(string1, string2)	Copy string2 into string1
strcat(string1, string2)	Concatenate string2 onto the end of string1
strcmp(string1, string2)	Lexically compares the two input strings (ASCII comparison)
	returns 0 if string1 is equal to string2
	< 0 if string1 is less than string2
	> 0 if string1 is greater than string2
strlen (string)	Gives the length of a string
strrev (string)	Reverse the string and result is stored in same string.
strncat(string1, string2, n)	Append n characters from string2 to string1
strncmp(string1, string2, n)	Compare first n characters of two strings.
strncpy(string1, string2, n)	Copy first n characters of string2 to string1
strupr (string)	Converts string to uppercase
strlwr (string)	Converts a string to lowercase
strchr (string, c)	Find first occurrence of character c in string.
strrchr (string, c)	Find last occurrence of character c in string.

Examples

1. strlen

strlen(s1) calculates the length of string s1.

```

#include <stdio.h>
#include <string.h>
int main()
{
char name[ ]= "Hello";
int len1, len2;
len1 = strlen(name);
len2 = strlen("Hello World");
printf("length of %s = %d\n", name, len1);

```

```
printf("length of %s = %d\n", "Hello World", len2);
return 0;
}
```

Output

length of Hello = 5

length of Hello World = 11

strlen doesn't count '\0' while calculating the length of a string.

2. strcat

strcat(s1, s2) concatenates(joins) the second string s2 to the first string s1.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s2[ ]= "World";
    char s1[20]= "Hello";
    Dept. of CSE, NMAMIT Page 64
    strcat(s1, s2);
    printf("Source string = %s\n", s2);
    printf("Target string = %s\n", s1);
    return 0;
}
```

Output

Source string = World

Target string = HelloWorld

3. strncat

strncat(s1, s2, n) concatenates(joins) the first 'n' characters of the second string s2 to the first string s1.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s2[ ]= "World";
    char s1[20]= "Hello";
    strncat(s1, s2, 2);
    printf("Source string = %s\n", s2);
    printf("Target string = %s\n", s1);
    return 0;
}
```

OUTPUT

Source string = World

Target string = HelloWo

4. strcpy

strcpy(s1, s2) copies the second string s2 to the first string s1.

```
#include <string.h>
#include <stdio.h>
int main()
{
    Dept. of CSE, NMAMIT Page 65
    char s2[ ]= "Hello";
    char s1[10];
    strcpy(s1, s2);
    printf("Source string = %s\n", s2);
    printf("Target string = %s\n", s1);
    return 0;
}
```

Output

Source string = Hello

Target string = Hello

5. strncpy

strncpy(s1, s2, n) copies the first 'n' characters of the second string s2 to the first string s1.

```
#include <string.h>
#include <stdio.h>
int main()
{
    char s2[ ]= "Hello";
    char s1[10];
    strncpy(s1, s2, 2);
    s1[2] = '\0'; /* null character manually added */
    printf("Source string = %s\n", s2);
    printf("Target string = %s\n", s1);
    return 0;
}
```


Output

Source string = Hello

Target string = He

Please note that we have added the null character ('\0') manually.

6. strcmp

strcmp(s1, s2) compares two strings and finds out whether they are same or different. It compares the two strings character by character till there is a mismatch. If the two strings are identical, it returns a 0. If not, then it returns the difference between the ASCII values of the first non-matching pair of characters.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[ ]= "Hello";
    char s2[ ]= "World";
    int i, j;
    i = strcmp(s1, "Hello");
    j = strcmp(s1, s2);
    printf("%d \n %d\n", i, j);
    return 0;
}
0
-15
```

The difference between the ASCII values of H(72) and W(87) is -15.

7. strncmp

strncmp(s1, s2, n) compares the first 'n' characters of s1 and s2.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[ ]= "Hello";
    char s2[ ]= "World";
    int i, j;
    i = strncmp(s1, "He BlogsDope", 2);
    printf("%d\n", i);
    return 0;
}
```

Output

0

8. strchr

strchr(s1, c) returns a pointer to the first occurrence of the character **c** in the string **s1** and returns **NULL** if the character **c** is not found in the string **s1**.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[ ]= "Hello Blogsdope";
    char c = 'B';
    char *p;
    p = strchr(s1, c);
    printf("%s\n", p);
    return 0;
}
```

Output

Blogsdope

9. strrchr

strrchr(s1, c) returns a pointer to the last occurrence of the character **c** in the string **s1** and returns **NULL** if the character **c** is not found in the string **s1**.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s1[ ]= "Hello Blogsdope";
    char c = 'o';
    char *p;
    p = strrchr(s1, c);
    printf("%s\n", p);
    return 0;
}
```

Output

ope

10. strlwr()

strlwr() function converts a given string into lowercase.

```
#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

11. strupr()

strupr() function converts a given string into uppercase.

```
#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

12. strrev()

strrev() function reverses a given string in C language.

```
#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

