

'C' PROGRAMMING

BCA 203

SELF LEARNING MATERIAL



DIRECTORATE OF DISTANCE EDUCATION

SWAMI VIVEKANAND SUBHARTI UNIVERSITY

MEERUT – 250 005,

UTTAR PRADESH (INDIA)

SLM Module Developed By :

Author:

Reviewed by :

Assessed by:

Study Material Assessment Committee, as per the SVSU ordinance No. VI (2)

Copyright © **Gayatri Sales**

DISCLAIMER

No part of this publication which is material protected by this copyright notice may be reproduced or transmitted or utilized or stored in any form or by any means now known or hereinafter invented, electronic, digital or mechanical, including photocopying, scanning, recording or by any information storage or retrieval system, without prior permission from the publisher.

Information contained in this book has been published by Directorate of Distance Education and has been obtained by its authors from sources be lived to be reliable and are correct to the best of their knowledge. However, the publisher and its author shall in no event be liable for any errors, omissions or damages arising out of use of this information and specially disclaim and implied warranties or merchantability or fitness for any particular use.

Published by: Gayatri Sales

Typeset at: Micron Computers

Printed at: Gayatri Sales, Meerut.

C PROGRAMMING

UNIT-I

Arrays

Definition, declaration and initialization of one dimensional array; Accessing array elements; Displaying array elements; Sorting arrays; Arrays and function; Two-Dimensional array: Declaration and Initialization, Accessing and Displaying, Memory representation of array [Row Major, Column Major]; Multidimensional array

UNIT-II

Pointers -Definition and declaration, Initialization; Indirection operator, address of operator; pointer arithmetic; dynamic memory allocation; arrays and pointers; function and pointers

UNIT-III

Strings

Definition, declaration and initialization of strings; standard library function: strlen(), strcpy(), strcat(), strcmp(); Implementation without using standard library functions

Structures

Definition and declaration; Variables initialization; Accessing fields and structure operations; Nested structures; Union: Definition and declaration; Differentiate between Union and structure

UNIT-IV

Introduction C Preprocessor

Definition of Preprocessor; Macro substitution directives; File inclusion directives; Conditional compilation

Bitwise Operators

Bitwise operators; Shift operators; Masks; Bit field

UNIT-V

File handling

Definition of Files, Opening modes of files; Standard function: fopen(), fclose(), feof(), fseek(), fwind(); Using text files: fgetc(), fputc(), fscanf() ,Command line arguments

UNIT-I

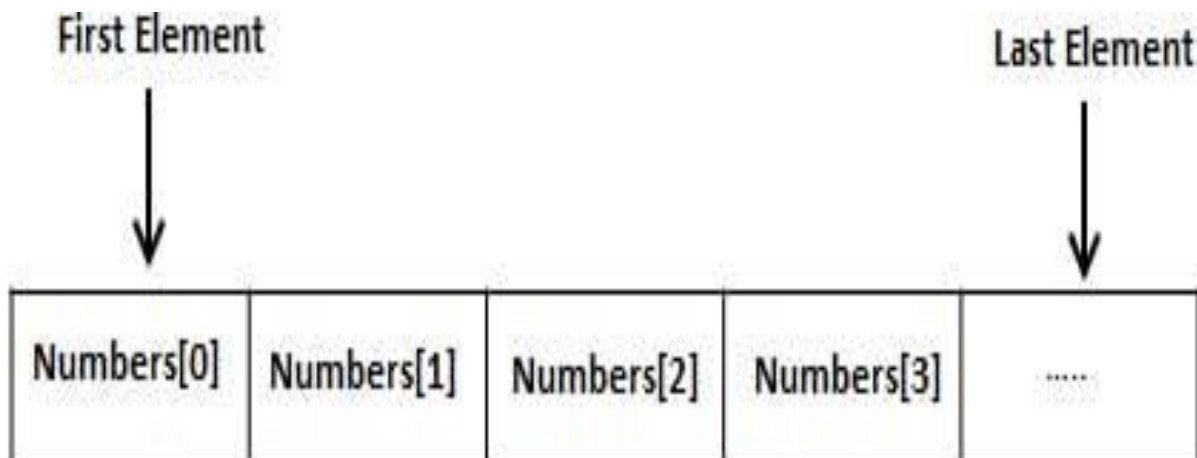
Arrays

Definition

Arrays are a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as `number0`, `number1`, ..., and `number99`, you declare one array variable such as `numbers` and use `numbers[0]`, `numbers[1]`, and ..., `numbers[99]` to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows –

```
type arrayName [ arraySize ];
```

This is called a single-dimensional array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type `double`, use this statement –

```
double balance[10];
```

Here balance is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows

–

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [].

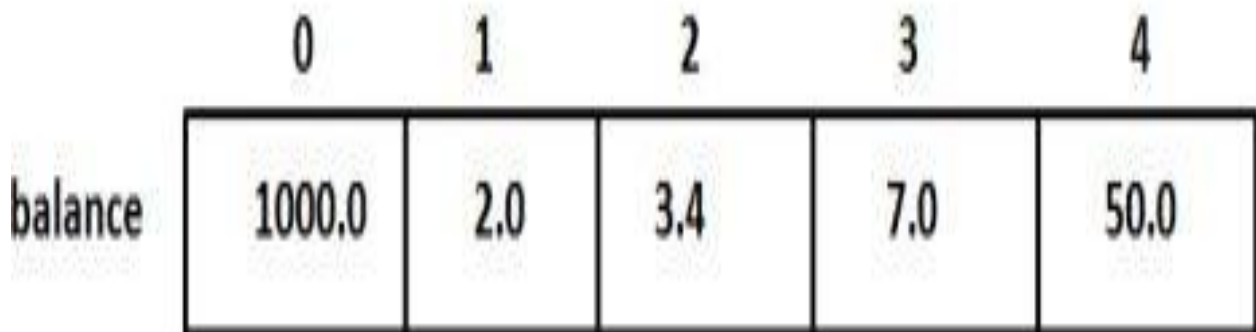
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write –

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array –

```
balance[4] = 50.0;
```

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above –



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example –

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays –

```
#include <stdio.h>

int main () {

    int n[ 10 ]; /* n is an array of 10 integers */
    int i,j;

    /* initialize elements of array n to 0 */
    for ( i = 0; i < 10; i++ ) {
        n[ i ] = i + 100; /* set element at location i to i + 100 */
    }

    /* output each array element's value */
    for ( j = 0; j < 10; j++ ) {
        printf("Element[%d] = %d\n", j, n[j] );
    }

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

Arrays in Detail

Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer –

Sr.No.	Concept & Description
1	<p>Multi-dimensional arrays</p> <p>C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.</p>
2	<p>Passing arrays to functions</p> <p>You can pass to the function a pointer to an array by specifying the array's name without an index.</p>
3	<p>Return array from a function</p> <p>C allows a function to return an array.</p>
4	<p>Pointer to an array</p> <p>You can generate a pointer to the first element of an array by simply specifying the array name, without any index.</p>

Declaration and initialization of one dimensional array

The variable allows us to store a single value at a time, what if we want to store roll no. of 100 students? For this task, we have to declare 100 variables, then assign values to each of them. What if there are 10000 students or more? As you can see declaring that many variables for a single entity (i.e student) is not a good idea. In a situation like these arrays provide a better way to store data.

What is an Array?

An array is a collection of one or more values of the same type. Each value is called an element of the array. The elements of the array share the same variable name but each element has its own unique index number (also known as a subscript). An array can be of any type, For example: int, float, char etc. If an array is of type int then it's elements must be of type int only.

To store roll no. of 100 students, we have to declare an array of size 100 i.e roll_no[100]. Here size of the array is 100 , so it is capable of

storing 100 values. In C, index or subscript starts from 0, so roll_no[0] is the first element, roll_no[1] is the second element and so on. Note that the last element of the array will be at roll_no[99] not at roll_no[100] because the index starts at 0.

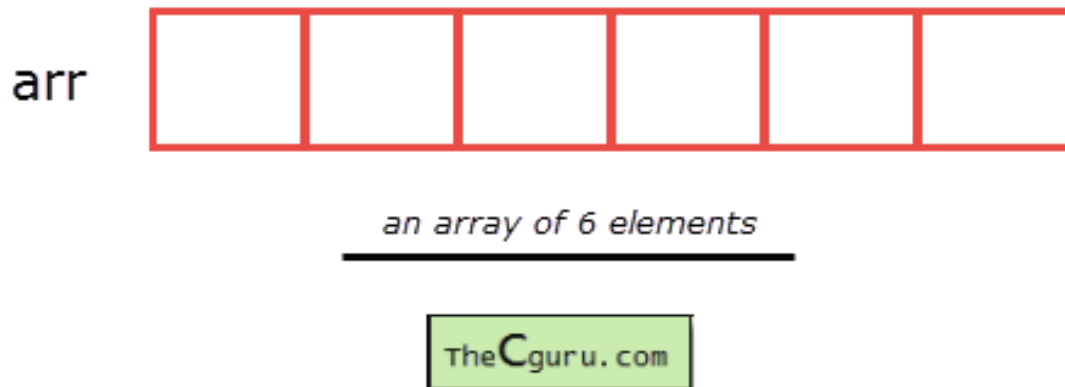


Arrays can be single or multidimensional. The number of subscript or index determines the dimensions of the array. An array of one dimension is known as a one-dimensional array or 1-D array, while an array of two dimensions is known as a two-dimensional array or 2-D array.

Let's start with a one-dimensional array.

One-dimensional array

Conceptually you can think of a one-dimensional array as a row, where elements are stored one after another.



Syntax: datatype array_name[size];

datatype: It denotes the type of the elements in the array.

array_name: Name of the array. It must be a valid identifier.

size: Number of elements an array can hold. here are some example of array declarations:

```
1 int num[100];
2 float temp[20];
3 char ch[50];
```

num is an array of type int, which can only store 100 elements of type int.

temp is an array of type float, which can only store 20 elements of type float.

ch is an array of type char, which can only store 50 elements of type char.

Note: When an array is declared it contains garbage values.

The individual elements in the array:

```
1 num[0], num[1], num[2], ....., num[99]
2 temp[0], temp[1], temp[2], ....., temp[19]
3 ch[0], ch[1], ch[2], ....., ch[49]
```

We can also use variables and symbolic constants to specify the size of the array.

```
1 #define SIZE 10
2
```

```

3 int main()
4 {
5     int size = 10;
6
7     int my_arr1[SIZE]; // ok
8     int my_arr2[size]; // not allowed until C99
9     // ...
10}

```

Note: Until C99 standard, we were not allowed to use variables to specify the size of the array. If you are using a compiler which supports C99 standard, the above code would compile successfully. However, If you're using an older version of C compiler like Turbo C++, then you will get an error.

The use of symbolic constants makes the program maintainable, because later if you want to change the size of the array you need to modify it at once place only i.e in the #define directive.

Accessing elements of an array

The elements of an array can be accessed by specifying array name followed by subscript or index inside square brackets (i.e []). Array subscript or index starts at 0. If the size of an array is 10 then the first element is at index 0, while the last element is at index 9. The first valid subscript (i.e 0) is known as the lower bound, while last valid subscript is known as the upper bound.

```
int my_arr[5];
```

then elements of this array are;

First element – my_arr[0]

Second element – my_arr[1]

Third element – my_arr[2]

Fourth element – my_arr[3]

Fifth element – my_arr[4]

Array subscript or index can be any expression that yields an integer value. For example:

```
1int i = 0, j = 2;
2my_arr[i]; // 1st element
3my_arr[i+1]; // 2nd element
4my_arr[i+j]; // 3rd element
```

In the array my_arr, the last element is at my_arr[4], What if you try to access elements beyond the last valid index of the array?

```
1printf("%d", my_arr[5]); // 6th element
2printf("%d", my_arr[10]); // 11th element
3printf("%d", my_arr[-1]); // element just before 0
```

Sure indexes 5, 10 and -1 are not valid but C compiler will not show any error message instead some garbage value will be printed. The C language doesn't check bounds of the array. It is the responsibility of the programmer to check array bounds whenever required.

Processing 1-D arrays

The following program uses for loop to take input and print elements of a 1-D array.

```
1 #include<stdio.h>
2
3 int main()
4 {
5     int arr[5], i;
6
7     for(i = 0; i < 5; i++)
8     {
9         printf("Enter a[%d]: ", i);
10        scanf("%d", &arr[i]);
11    }
12
13    printf("\nPrinting elements of the array: \n\n");
14
15    for(i = 0; i < 5; i++)
16    {
17        printf("%d ", arr[i]);
18    }
19
20    // signal to operating system program ran fine
```

```
21 | return 0;
22 | }
Try it now
```

Expected Output:

```
1 | Enter a[0]: 11
2 | Enter a[1]: 22
3 | Enter a[2]: 34
4 | Enter a[3]: 4
5 | Enter a[4]: 34
6 |
7 | Printing elements of the array:
8 |
9 | 11 22 34 4 34
```

How it works:

In Line 5, we have declared an array of 5 integers and variable `i` of type `int`. Then a `for` loop is used to enter five elements into an array. In `scanf()` we have used `&` operator (also known as the address of operator) on element `arr[i]` of an array, just like we had done with variables of type `int`, `float`, `char` etc. Line 13 prints "Printing elements of the array" to the console. The second `for` loop prints all the elements of an array one by one.

The following program prints the sum of elements of an array.

```
1 | #include<stdio.h>
2 |
3 | int main()
4 | {
5 |     int arr[5], i, s = 0;
6 |
7 |     for(i = 0; i < 5; i++)
8 |     {
9 |         printf("Enter a[%d]: ", i);
10 |         scanf("%d", &arr[i]);
11 |     }
```

```
12
13 for(i = 0; i < 5; i++)
14 {
15     s += arr[i];
16 }
17
18 printf("\nSum of elements = %d ", s);
19
20 // signal to operating system program ran fine
21 return 0;
22}
Try it now
```

Expected Output:

```
1Enter a[0]: 22
2Enter a[1]: 33
3Enter a[2]: 56
4Enter a[3]: 73
5Enter a[4]: 23
6
7Sum of elements = 207
```

How it works: The first for loop asks the user to enter five elements into the array. The second for loop reads all the elements of an array one by one and accumulate the sum of all the elements in the variable s. Note that it is necessary to initialize the variable s to 0, otherwise, we will get the wrong answer because of the garbage value of s.

Initializing Array

When an array is declared inside a function the elements of the array have garbage value. If an array is global or static, then its elements are automatically initialized to 0. We can explicitly initialize elements of an array at the time of declaration using the following syntax:

Syntax: datatype array_name[size] = { val1, val2, val3, valN };

datatype is the type of elements of an array.

array_name is the variable name, which must be any valid identifier.

size is the size of the array.

val1, val2 ... are the constants known as initializers. Each value is separated by a comma(,) and then there is a semi-colon (;) after the closing curly brace (}).

Accessing array elements

After we have understood what are arrays, how they are to be declared, and its applications, it is time to understand how to access the elements.

Although, we store similar data items into a group called array which is referenced to by a single name like marks, temp etc as seen previously, however in order to retrieve and access the elements we do not have a single operation.

We need to use loops for this purpose. We access the elements by incrementing the value of the index/subscript. These subscripts are integer values.

An array element can be accessed through an index number.

Example: Consider an integer array called rollno.

```
int rollno[10]; //array declaration
```

```
int i;
```

```
for (i=0; i<10; i++)
```

```
{
```

```
rollno[i] = i+1; //accessing elements of array and assigning value to them.
```

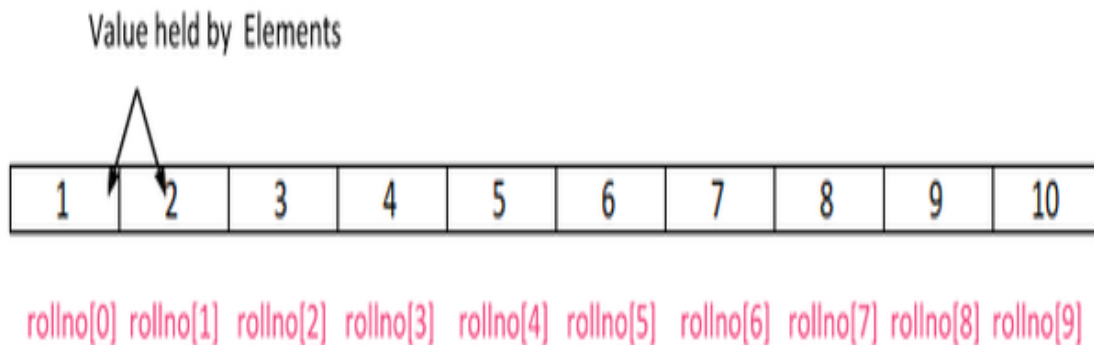
```
}
```

Copy

RUN

In the above example, we are accessing initially the first element of the array which is at the 0th index, hence the counter $i = 0$. Then we assign value to it using the assignment operator. Similarly, we access and assign value to each element until the last one which is located at 9th index, hence $i < 10$. Because if we put $i \leq 10$; then what happens is when $i = 10$; there is no `rollno[10]` as a part of the array; because the last index is 9 as array initialises from 0.

The output of the above code would be as follows:



This leads us to an important conclusion that , in order to access or do any operation on array elements we have to use loop and perform the same operation on each element by iterating the index value.

A very essential part of arrays is to understand how the compiler knows where individual elements of array are located in memory. So, let us understand how is it done.

Memory Address calculation of array elements:

You will be surprised to know that when we use the array name, we are referring to nothing but the first byte of the array. Array name corresponds to the address of the first byte of the array.

The subscripts or index are offset / you can say the distance from the starting location of the array.

Also, all the elements are located in consecutive memory locations. Thus using the name of array which is the base address and offset, C calculates address of all elements in run-time.

Formula to calculate address:

Address of element of $a[i] = \text{base address of array } a[] + \text{data_type_size} * (i - \text{starting index})$

Here, 'a' is the array, i is the index of element whose address we are calculating (offset), data_type_size refers to the memory blocks reserved for a particular data type; for e.g: if your array is of integer type; each element will occupy 2 bytes. Starting index is usually 0.

For instance, correlate this calculation to your calculation of age. You just require today's date and the date of birth in order to derive age, right? So consider your base address is your DOB and today's date is the offset or index. Voila! It's easy to relate to it now, isn't it!

Example:

You are given the following array :

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

Base Address = 2000, Calculate the address of rollno[3].

Answer:

The array will be represented in memory as follows:

1	2	3	4
rollno[0]	rollno[1]	rollno[2]	rollno[3]
2000	2002	2004	2006

As we know storing int requires 2 bytes.

Hence,

data_type_size = 2

starting index = 0

base address of array rollno[] = 2000

i =3

Hence rollno[3] = 2000 + 2*(3-0) =2000 + 6 = 2006.

In the next section, we shall see how to initialize, input, and assign values in an array.

Displaying array elements

This is a simple program to create an array and then to print it's all elements.

Now, just know about arrays.

Arrays are the special variables that store multiple values under the same name in the contiguous memory allocation. Elements of the array can be accessed through their indexes.

Here, 1, 2, 3, 4 and 5 represent the elements of the array. These elements can be accessed through their corresponding indexes, 1.e., 0, 1, 2, 3 and 4.

Algorithm

- **STEP 1:** START
- **STEP 2:** INITIALIZE arr[] = {1, 2, 3, 4, 5}.
- **STEP 3:** PRINT "Elements of given array:"
- **STEP 4:** REPEAT STEP 5 for(i=0; i<arr.length; i++)
- **STEP 5:** PRINT arr[i]
- **STEP 6:** END

Program:

1. **public class** PrintArray {
2. **public static void** main(String[] args) {
3. //Initialize array
4. **int** [] arr = **new int** [] {1, 2, 3, 4, 5};
5. System.out.println("Elements of given array: ");

```
6. //Loop through the array by incrementing value of i
7. for (int i = 0; i < arr.length; i++) {
8.     System.out.print(arr[i] + " ");
9. }
10. }
11. }
```

Output:

```
Elements of given array:
1    2  3  4  5
```

Sorting arrays

The sort() method sorts an array alphabetically:

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort(); // Sorts the elements of fruits
```

Reversing an Array

The reverse() method reverses the elements in an array.

You can use it to sort an array in descending order:

Example

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort(); // First sort the elements of fruits
fruits.reverse(); // Then reverse the order of the elements
```

Numeric Sort

By default, the sort() function sorts values as **strings**.

This works well for strings ("Apple" comes before "Banana").

However, if numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1".

Because of this, the `sort()` method will produce incorrect result when sorting numbers.

You can fix this by providing a **compare function**:

The Compare Function

The purpose of the compare function is to define an alternative sort order.

The compare function should return a negative, zero, or positive value, depending on the arguments:

```
function(a, b){return a - b}
```

When the `sort()` function compares two values, it sends the values to the compare function, and sorts the values according to the returned (negative, zero, positive) value.

If the result is negative a is sorted before b.

If the result is positive b is sorted before a.

If the result is 0 no changes are done with the sort order of the two values.

Example:

The compare function compares all the values in the array, two values at a time (a, b).

When comparing 40 and 100, the `sort()` method calls the `compare function(40, 100)`.

The function calculates $40 - 100$ (a - b), and since the result is negative (-60), the sort function will sort 40 as a value lower than 100.

You can use this code snippet to experiment with numerically and alphabetically sorting:

```
<button onclick="myFunction1()">Sort Alphabetically</button>
```

```
<button onclick="myFunction2()">Sort Numerically</button>
```

```
<p id="demo"></p>
```

```
<script>
```

```
var points = [40, 100, 1, 5, 25, 10];
```

```
document.getElementById("demo").innerHTML = points;
```

```
function myFunction1() {
  points.sort();
  document.getElementById("demo").innerHTML = points;
}

function myFunction2() {
  points.sort(function(a, b){return a - b});
  document.getElementById("demo").innerHTML = points;
}
</script>
```

Arrays and function

- Things to note about C-style arrays:
 - An array is not a type
 - An array is a primitive C-style construct that consists of many items stored consecutively and accessed through a single variable name (and indexing)
 - This is actually done by remembering the starting address of an array, and computing an offset
 - The name of an array acts as a special kind of variable -- a **pointer** -- which stores the starting address of the array
- An array can be passed into a function as a parameter
 - Because an array is not a *single item*, the array contents are not passed "by value" as we are used to with normal variables
 - The normal meaning of "pass by value" is that the actual argument value is *copied* into a local formal parameter variable
 - In the case of arrays, just the *pointer* is copied as a parameter. We'll see this in more detail when we get to pointers

- When an array is sent into a function, only its starting address is really sent
- This means the function will **always** have access to the actual array sent in
- Returning an array from a function works similarly, but we need pointers to use them well (not yet covered)
- Example function:
 - `void PrintArray (int arr[], int size)`
 - `{`
 - `for (int i = 0; i < size; i++)`
 - `cout << arr[i] << ' ';`
 - `}`

Note that:

- The variable `arr` acts as the local array name inside the function
- There is no number in the brackets. `int []` indicates that this is an array parameter, for an array of type `int`
- It's usually a good idea to pass in the array size as well, as another parameter. This helps make a function work for any size array
- Sample call to the above function:
 - `int list[5] = {2, 4, 6, 8, 10};`
 - `PrintArray(list, 5); // will print: 2 4 6 8 10`

Using `const` with array parameters

- Remember: When passing an array into a function, the function *will* have access to the contents of the original array!
- Some functions that *should* change the original array:
 - `Sort()`, `Reverse()`, `SwapElements()`

- What if there are functions that should *not* alter the array contents?
 - PrintArray(), Sum(), Average()
- Put const in front of the array parameter to guarantee that the array contents will not be changed by the function:
 - void PrintArray (const int arr[], const int size)
 - {
 - for (int i = 0; i < size; i++)
 - cout << arr[i] << ' ';
 - }

Notice that the const on the variable size is a good idea in this example, too. Why?

arrayfunc.cpp -- Here's an example program illustrating a couple of functions that take array parameters.

Practice exercises

Try writing the following functions (each takes in one or more arrays as a parameter) for practice:

- A function called Sum that takes in an array of type double and returns the sum of its elements
- A function called Average that takes in an integer array and returns the average of its elements (as a double)
- A function called Reverse that takes in an array (any type) and reverses its contents
- A function called Sort that takes in an array of integers and sorts its contents in ascending order
- A function called Minimum that takes in an array of type double and returns the smallest number in the array

Two- Dimensional array

An array of arrays is known as 2D array. The two dimensional (2D) array in C programming is also known as matrix. A matrix can be represented as a table of rows and columns. Before we discuss more about two Dimensional array lets have a look at the following C program.

Simple Two dimensional (2D) Array Example

For now don't worry how to initialize a two dimensional array, we will discuss that part later. This program demonstrates how to store the elements entered by user in a 2d array and how to display the elements of a two dimensional array.

```
#include<stdio.h>
int main(){
    /* 2D array declaration*/
    int disp[2][3];
    /*Counter variables for the loop*/
    int i, j;
    for(i=0; i<2; i++) {
        for(j=0;j<3;j++) {
            printf("Enter value for disp[%d][%d]:", i, j);
            scanf("%d", &disp[i][j]);
        }
    }
    //Displaying array elements
    printf("Two Dimensional array elements:\n");
    for(i=0; i<2; i++) {
        for(j=0;j<3;j++) {
            printf("%d ", disp[i][j]);
            if(j==2){
                printf("\n");
            }
        }
    }
    return 0;
}
```

Output:

```
Enter value for disp[0][0]:1
```

```
Enter value for disp[0][1]:2
Enter value for disp[0][2]:3
Enter value for disp[1][0]:4
Enter value for disp[1][1]:5
Enter value for disp[1][2]:6
Two Dimensional array elements:
1 2 3
4 5 6
```

Initialization of 2D Array

There are two ways to initialize a two Dimensional arrays during declaration.

```
int disp[2][4] = {
    {10, 11, 12, 13},
    {14, 15, 16, 17}
};
```

OR

```
int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};
```

Although both the above declarations are valid, I recommend you to use the first method as it is more readable, because you can visualize the rows and columns of 2d array in this method.

Things that you must consider while initializing a 2D array

We already know, when we initialize a normal array (or you can say one dimensional array) during declaration, we need not to specify the size of it. However that's not the case with 2D array, you must always specify the second dimension even if you are specifying elements during the declaration. Let's understand this with the help of few examples –

```
/* Valid declaration*/
int abc[2][2] = {1, 2, 3 ,4 }
/* Valid declaration*/
int abc[][2] = {1, 2, 3 ,4 }
/* Invalid declaration – you must specify second dimension*/
int abc[][] = {1, 2, 3 ,4 }
```



```
/* Invalid because of the same reason mentioned above*/  
int abc[2][] = {1, 2, 3, 4 }
```

How to store user input data into 2D array

We can calculate how many elements a two dimensional array can have by using this formula:

The array `arr[n1][n2]` can have $n1*n2$ elements. The array that we have in the example below is having the dimensions 5 and 4. These dimensions are known as subscripts. So this array has **first subscript** value as 5 and **second subscript** value as 4.

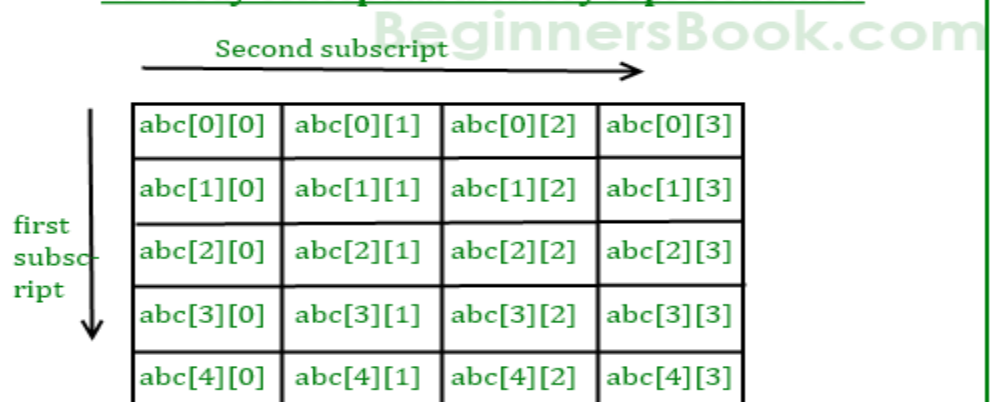
So the array `abc[5][4]` can have $5*4 = 20$ elements.

To store the elements entered by user we are using two for loops, one of them is a nested loop. The outer loop runs from 0 to the (first subscript -1) and the inner for loops runs from 0 to the (second subscript -1). This way the the order in which user enters the elements would be `abc[0][0]`, `abc[0][1]`, `abc[0][2]`...so on.

```
#include<stdio.h>  
int main(){  
    /* 2D array declaration*/  
    int abc[5][4];  
    /*Counter variables for the loop*/  
    int i, j;  
    for(i=0; i<5; i++) {  
        for(j=0;j<4;j++) {  
            printf("Enter value for abc[%d][%d]:", i, j);  
            scanf("%d", &abc[i][j]);  
        }  
    }  
    return 0;  
}
```

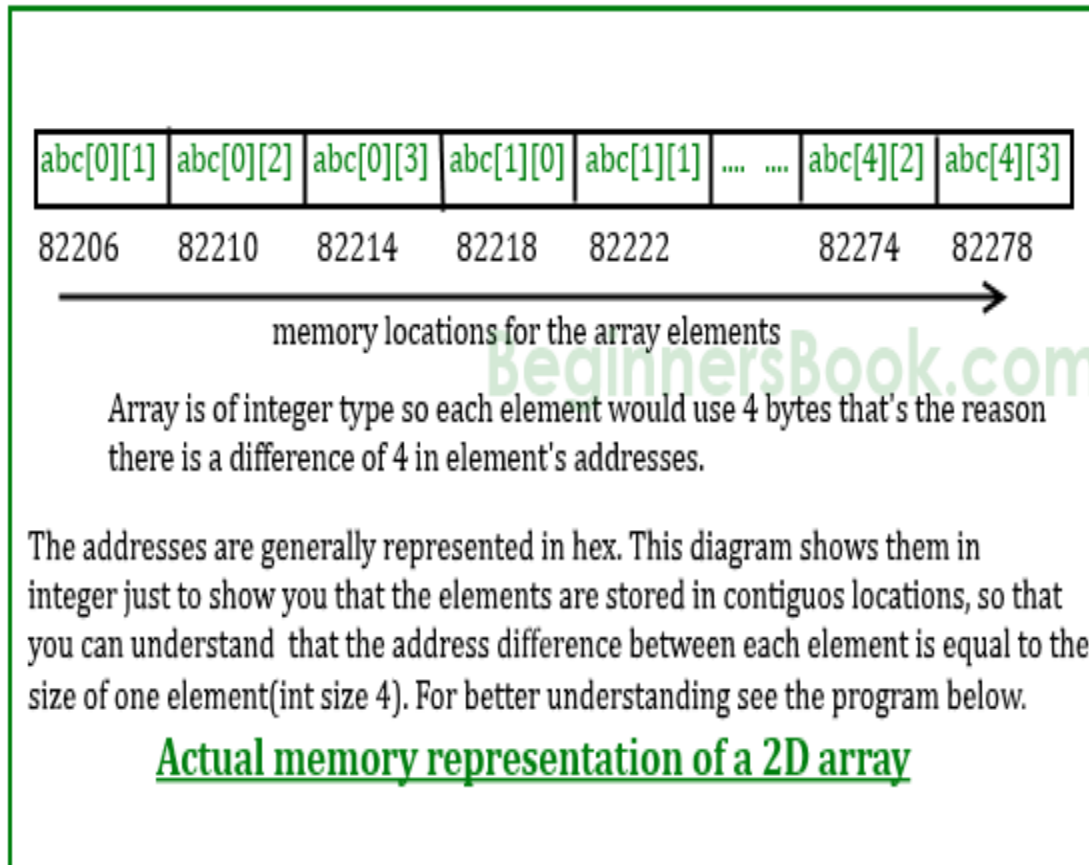
In above example, I have a 2D array `abc` of integer type. Conceptually you can visualize the above array like this:

2D array conceptual memory representation



Here my array is `abc[5][4]`, which can be conceptually viewed as a matrix of 5 rows and 4 columns. Point to note here is that subscript starts with zero, which means `abc[0][0]` would be the first element of the array.

However the actual representation of this array in memory would be something like this:



Pointers & 2D array

As we know that the one dimensional array name works as a pointer to the base element (first element) of the array. However in the case 2D arrays the logic is slightly different. You can consider a 2D array as collection of several one dimensional arrays.

So `abc[0]` would have the address of first element of the first row (if we consider the above diagram number 1).

similarly `abc[1]` would have the address of the first element of the second row. To understand it better, lets write a C program –

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

```

{
int abc[5][4] ={
    {0,1,2,3},
    {4,5,6,7},
    {8,9,10,11},
    {12,13,14,15},
    {16,17,18,19}
};
for (int i=0; i<=4; i++)
{
/* The correct way of displaying an address would be
* printf("%p ",abc[i]); but for the demonstration
* purpose I am displaying the address in int so that
* you can relate the output with the diagram above that
* shows how many bytes an int element uses and how they
* are stored in contiguous memory locations.
*
*/
    printf("%d ",abc[i]);
}
return 0;
}

```

Output:

```
1600101376 1600101392 1600101408 1600101424 1600101440
```

The actual address representation should be in hex for which we use %p instead of %d, as mentioned in the comments. This is just to show that the elements are stored in contiguous memory locations. You can relate the output with the diagram above to see that the difference between these addresses is actually number of bytes consumed by the elements of that row.

The addresses shown in the output belongs to the first element of each row `abc[0][0]`, `abc[1][0]`, `abc[2][0]`, `abc[3][0]` and `abc[4][0]`.

Declaration and Initialization

C variables are names used for storing a data value to locations in memory. The value stored in the c variables may be changed during program execution.

C is a strongly typed language. Every variable must be declared, indicating its data type before it can be used. The declaration can also involve explicit initialization, giving the variable a value; a variable that is declared but not explicitly initialized is of uncertain value (and should be regarded as dangerous until it is initialized). In K&R C, declarations must precede all other statements, but in modern versions of C, this rule is relaxed so that you don't have to declare a variable until just before you start using it:

The basic form of a variable declaration is shown here:

```
type identifier [= value][, identifier [= value] ...] ;
```

- The type is one of C's data types like int, char, double etc.
- The identifier is the name of the variable. You can initialize the variable by specifying an equal sign and a value. Keep in mind
- You can initialize the variable by specifying an equal sign and a value. Keep in mind that the initialization expression must result in a value of the same (or compatible) type as that specified for the variable.
- To declare more than one variable of the specified type, use a comma-separated list.

Declaration of Variable

Declaration of variable in c can be done using following syntax:

```
data_type variable_name;  
or  
data_type variable1, variable2,...,variablen;
```

where `data_type` is any valid c data type and `variable_name` is any valid identifier.

For example,

```
int a;  
float variable;  
float a, b;
```

Initialization of Variable

C variables declared can be initialized with the help of assignment operator '='.

Syntax

```
data_type variable_name=constant/literal/expression;  
or  
variable_name=constant/literal/expression;
```

Example:

```
int a=10;  
int a=b+c;  
a=10;  
a=b+c;
```

Multiple variables can be initialized in a single statement by single value, for example, a=b=c=d=e=10;

NOTE: C variables must be declared before they are used in the c program. Also, since c is a case-sensitive programming language, therefore the c variables, abc, Abc and ABC are all different.

int Variable Declaration and Variable Initialization in two steps:

Save Source File Name as : **IntExample.c Program**

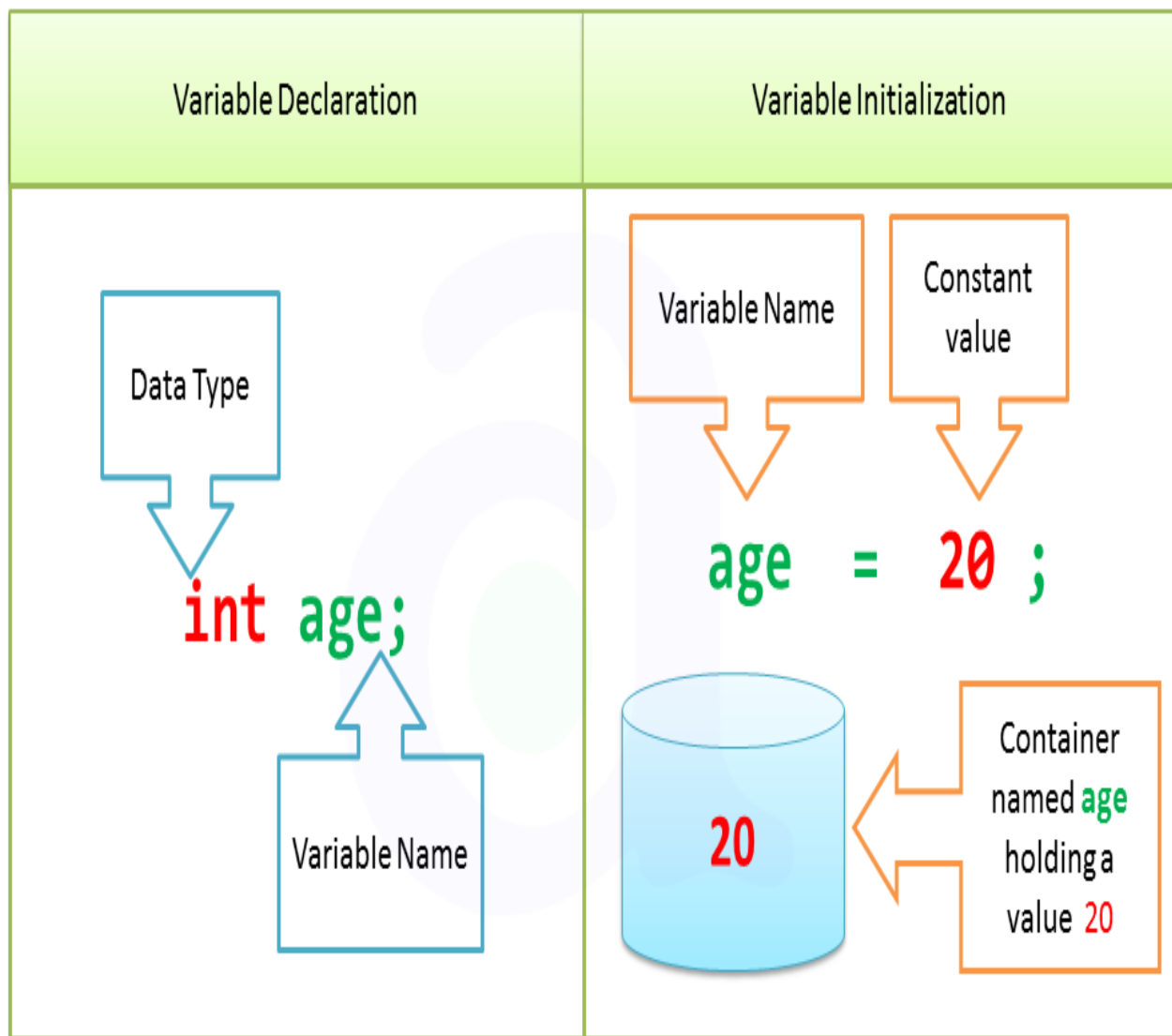
```
/*  
 C program to demonstrate Declaration and initialization  
 C program to demonstrate int data type  
*/  
  
#include  
void main()  
{  
    int age; // Declaration of variable age  
    age = 20; // initialization of the variable age  
    printf("%d \n",age);
```

```
}
```

Output

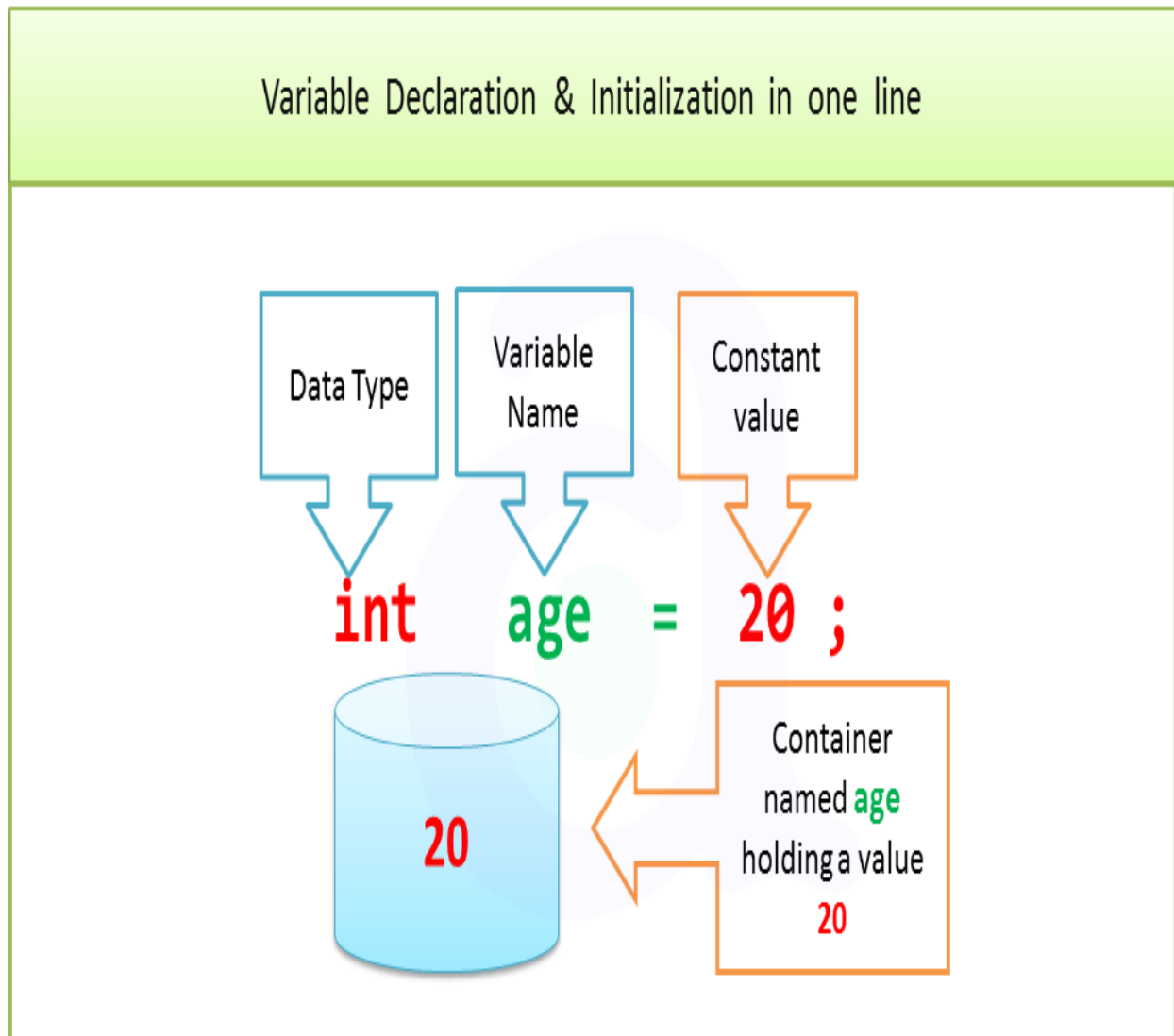
20

Press any key to continue . . .



`int` variable Declaration and Initialization

int Variable Declaration and Variable Initialization in one line:



`int` variable Declaration and Initialization

```
/*  
C program to demonstrate Declaration and initialization  
C program to demonstrate int data type  
*/  
  
#include
```



```
void main()
{
    int age=20; // Declaration and initialization of variable age
    printf("%d \n",age);
}
```

Output

20

Press any key to continue . . .

Multiple variables can be initialized in a single statement by single value

```
a=b=c=d=e=10;
```

```
/*
C program to demonstrate Declaration and initialization
C program to demonstrate int data type
*/

#include
void main()
{

    int a, b, c, d, e; // Declaration of variable
    a=b=c=d=e=10; // initialization of variable
    printf("%d \n",a);
    printf("%d \n",b);
    printf("%d \n",c);
    printf("%d \n",d);
    printf("%d \n",e);
}
```

Output

10

10

10

10

10

Press any key to continue . . .

Point To remember

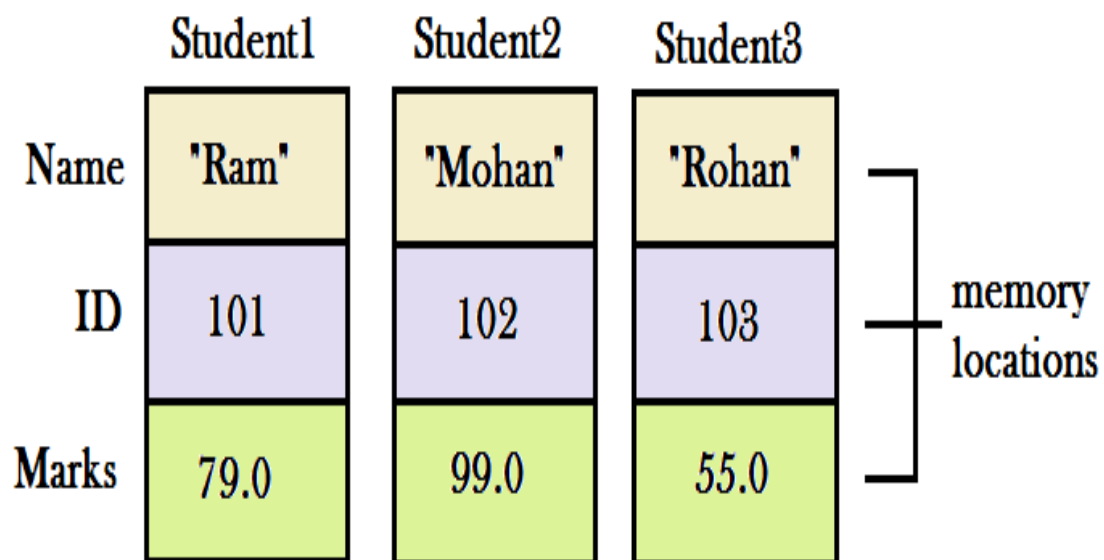
- Variables should be declared in the C program before to use.
- Memory space is not allocated for a variable while declaration. It happens only on the variable definition.
- Variable initialization means assigning a value to the variable.

Type	Syntax
Variable declaration	<code>data_type variable_name;</code> Example: <code>int x, y, z; char flat, ch;</code>
Variable initialization	<code>data_type variable_name = value;</code> Example: <code>int x = 50, y = 30; char flag = 'x', ch='l';</code>

Accessing and Displaying

The **array** allows us to store multiple data of the same data type under the same name. What if we need to store multiple data of different data type under the same name? No problem, C Programming language allows us to create **structures** which let us store multiple data belonging to different data types under the same name.

Structures are mostly used for maintaining records like mark sheets with student details like name, ID number, and marks stored under the same name. Another example is for maintaining library records with the name of the book, author, book ID, and subject stored under the same name. This is how data gets stored in a **structure**:



codinggeek.com

Table of Contents

- Advantages of Structures over Arrays
- Declaring a structure
 - Note:
- Accessing Structure elements
 - 1) Using Dot(.) operator
 - Program to store and display the student details using Structure and Dot operator

- 2) Using arrow(->) operator
- Program to store and display the student details using Structure and Arrow operator
- Recommended -

ADVANTAGES OF STRUCTURES OVER ARRAYS

To begin with, **structures** can handle multiple data types that are not possible in **arrays**. Arrays can only handle the same type of data. Consider the example stated above of the students' details. We can make use of the array in order to store and access the name, ID number, and marks but we lose a particular identity relating the data to only one student. In the case of **structures**, this is not a problem. We can store and access the data relating to one student without losing the particular identity even with an enormous amount of data.

DECLARING A STRUCTURE

We already know that **structure** falls under **user-defined data types**. So we define a new data type using the keyword "struct". The syntax of the declaration is given below:

1. **struct** structure_name // give any name to your structure
2. {
3. datatype1 element1;
4. datatype2 element2;
5. datatype3 element3;
6. };

Now we can define variables that shall contain the elements declared under "struct". This is shown below:

1. **struct** structure_name //Give any name to your structure
2. {
3. datatype1 element1;
4. datatype2 element2;

5. datatype3 element3;
6. };
7. **struct** structure_name variable_declarations;

Let us consider the student example:

1. **struct** student
2. {
3. **char** name[10];
4. **int** ID;
5. **float** marks;
6. };
7. **struct** student s1,s2,s3;

NOTE:

- The declaration of a **structure** only defines the form of the structure and not allocates memory for storage. The memory is allocated after defining the variables. **Eg: s1,s2,s3;**
- The declaration of a structure should be followed by a semicolon(**;**).
- If we initialize a structure variable to **{0}** then all its elements will be initialized with a value **0**. **Eg: struct student s1={0};**
- Structures are declared before any function or variable is defined. For complex codes, they are put in another file and called in the program as header files using **#include**.

ACCESSING STRUCTURE ELEMENTS

There are two ways to access the structure elements –

1) USING DOT(.) OPERATOR

In an **array**, we use the subscript to access the different members of the array.

In **structures**, we use the dot operator (**.**) also known as the member access operator.

This operator is placed between the structure variable name and the structure element that we need to access. For example: if we need to access the name of the first student then we call it in the following format: **s1.name**

Similarly, for id: **s1.id**

Printing these values:

PROGRAM TO STORE AND DISPLAY THE STUDENT DETAILS USING STRUCTURE AND DOT OPERATOR

```
1. #include <stdio.h>
2. int main()
3. {
4.     struct student//declaring the structure with name student
5.     {
6.         char name[10];//stores the name of the student
7.         int id;//stores the ID of the student
8.         float marks;//stores the marks of the student
9.     };
10.
11. /*initializing values to all the structure variables*/
12. struct student s1={"Ram",101,79.0};
13. struct student s2={"Mohan",102,99.0};
14. struct student s3={"Rohan",103,55.0};
```

```
15. struct student s4={0};//all values initialized as 0

16.

17. /*displaying student details*/

18. printf("Details of student1:\n");

19. printf(" Name= %s\n ID= %d\n marks= %f\n\n",s1.name,s1.id,s1.marks);

20. printf("Details of student2:\n");

21. printf(" Name= %s\n ID= %d\n marks= %f\n\n",s2.name,s2.id,s2.marks);

22. printf("Details of student3:\n");

23. printf(" Name= %s\n ID= %d\n marks= %f\n\n",s3.name,s3.id,s3.marks);

24. printf("Details of student4:\n");

25. printf(" Name= %s\n ID= %d\n marks= %f\n\n",s4.name,s4.id,s4.marks);

26. return 0;

27. }
```

Output:-

Details of student1:

Name= Ram

ID= 101

marks= 79.000000

Details of student2:

Name= Mohan

ID= 102

marks= 99.000000

Details of student3:

Name= Rohan

ID= 103

marks= 55.000000

Details of student4:

Name=

ID= 0

marks= 0.000000

2) USING ARROW(->) OPERATOR

Another way to access the structure is using the **arrow operator (-> operator)**. This is mostly used in case of structure pointers or when we pass the address of a structure to a function(call by reference). This operator is used in the same way as the dot operator, i.e., between the structure pointer name and structure element. Here is an example of how we use an arrow operator (This program only discuss about how to access a value of a structure using a pointer but if you have to create a structure based on user value then you have to use memory allocation using malloc or any other related method along with **scanf()** to read the values for the declared structure pointer):

PROGRAM TO STORE AND DISPLAY THE STUDENT DETAILS USING STRUCTURE AND ARROW OPERATOR

1. #include <stdio.h>
- 2.
3. /*declaring the structure*/


```
4. struct student{
5. char name[10];
6. int id;
7. double marks;
8. };
9.
10. void display(struct student *);//declaring the function to display the contents of the
    structure address passed to the function
11.
12. int main()
13. {
14. struct student s1={"Ram",101,79.0};
15. struct student s2={"Mohan",102,99.0};
16. struct student *x;
17. x = &s1;//storing the address of the structure variable s1
18.
19. /*displaying the details of student1 using pointers*/
20. printf("The student details are as follows:\n");
21. printf(" Name= %s\n ID= %d\n marks= %f\n\n",x->name,x->id,x->marks);
22. display(&s2);//sending the address of the structure variable s2 to function display()
23.
```

```
24. return 0;

25.}

26.

27./*displaying the contents of the address sent to the function*/

28. void display(struct student *s)

29.{

30. printf("The student details are as follows:\n");

31. printf(" Name= %s\n ID= %d\n marks= %f\n\n",s->name,s->id,s->marks);

32.}
```

Output:-

The student details are as follows:

Name= Ram

ID= 101

marks= 79.000000

The student details are as follows:

Name= Mohan

ID= 102

marks= 99.000000

Memory representation of array [Row Major, Column Major]

Computer hardware does not directly support the concept of multidimensional arrays. Computer memory is unidimensional, providing memory addresses that start at zero and increase serially to the highest available location. Multidimensional arrays are therefore a software concept: software (IDL in this case) maps the elements of a multidimensional array into a contiguous linear span of memory addresses. There are two ways that such an array can be represented in one-dimensional linear memory. These two options, which are explained below, are commonly called row major and column major. All programming languages that support multidimensional arrays must choose one of these two possibilities. This choice is a fundamental property of the language, and it affects how programs written in different languages share data with each other.

Before describing the meaning of these terms and IDL's relationship to them, it is necessary to understand the conventions used when referring to the dimensions of an array. For mnemonic reasons, people find it useful to associate higher level meanings with the dimensions of multi-dimensional data. For example, a 2-D variable containing measurements of ozone concentration on a uniform grid covering the earth might associate latitude with the first dimension, and longitude with the second dimension. Such associations help people understand and reason about their data, but they are not fundamental properties of the language itself. It is important to realize that no matter what meaning you attach to the dimensions of an array, IDL is only aware of the number of dimensions and their size, and does not work directly in terms of these higher order concepts. Another way of saying this is that `arr[d1, d2]` addresses the same element of variable `arr` no matter what meaning you associate with the two dimensions.

In the IDL world, there are two such conventions that are widely used:

• In image processing, the first dimension of an image array is the column, and the second dimension is the row.

Hence, the dominant convention in IDL documentation is to refer to the first dimension of an array as the column.

• In the standard mathematical notation used for linear algebra, the first dimension of an array (or matrix) is the row, and the second dimension is the column, which is opposite of the image processing convention.

In computer science, the way array elements are mapped to memory is always defined using the mathematical [row, column] notation. Much of the following discussion utilizes the $m \times n$ array shown in the following figure, with m rows and n columns:

An $m \times n$ array represented in mathematical notation.

Given such a 2-dimensional matrix, there are two ways that such an array can be represented in 1-dimensional linear memory – either row by row (row major), or column by column (column major):

• **Contiguous First Dimension (Column Major):** In this approach, all elements of

the first dimension (m in this case) are stored contiguously in memory. The 1-D linear address of element A_{d_1, d_2} is therefore given by the formula $(d_2 * m + d_1)$. As you move linearly through the memory of such an array, the first (leftmost) dimension changes the fastest, with the second dimension (n, in this case) incrementing every time you come to the end of the first dimension:

$$A_{0,0}, A_{1,0}, \dots, A_{m-1,0}, A_{0,1}, A_{1,1}, \dots, A_{m-1,1}, \dots$$

Computer languages that map multidimensional arrays in this manner are called column major, following the mathematical [row, column] notation. IDL and Fortran are both examples of column-major languages.

Contiguous Second Dimension (Row Major): In this approach, all elements of the second dimension (n, in this case) are stored contiguously in memory. The 1-D linear address of element A_{d_1, d_2} is therefore given by the formula $(d_1 * n + d_2)$. As you move linearly through the memory of such an array, the second dimension changes the fastest, with the first dimension (m in this case) incrementing every time you come to the end of the second dimension:

$$A_{0,0}, A_{0,1}, \dots, A_{0,n-1}, A_{1,0}, A_{1,1}, \dots, A_{1,n-1}, \dots$$

Computer languages that map multidimensional arrays in this manner are known as row major. Examples of row-major languages include C and C++.

The terms row major and column major are widely used to categorize programming languages. It is important to understand that when programming languages are discussed in this way, the mathematical convention in which the first dimension represents the row and the second dimension represents the column is used. If you use the image-processing convention in which the first dimension represents the column and the second dimension represents the row you should be careful to make note of the distinction.

Note: IDL users who are comfortable with the IDL image-processing-oriented array notation [column, row] frequently follow the reasoning outlined above and incorrectly conclude that IDL is a row-major language. The often-overlooked cause of this mistake is that the standard definition of the terms row major and column major assume the mathematical [row, column] notation. In such cases, it can be helpful to look beyond the row/column terminology and think in terms of which dimension is contiguous in memory.

Note that the $m \times n$ array shown above could be represented with equal accuracy as having m columns and n rows, as shown in the figure below. This corresponds to the image-processing [column, row] notation. It's important to note that while the representation shown is the transpose of the representation in the figure above, the

data stored in the computer memory are identical. Only the two-dimensional representation, which takes its form from the notational convention used, has changed.

An $m \times n$ array represented in image-processing notation.

IDL's choice of column-major array layout reflects its roots as an image processing language. The fact that the elements of the first dimension are contiguous means that the elements of each row of an image array (using [column, row] notation, as shown in the second figure, are contiguous. This is the order expected by most graphics hardware, providing an efficiency advantage for languages that naturally store data that way. Also, this ordering minimizes virtual memory overhead, since images are accessed linearly.

It should be clear that the higher-level meanings associated with array dimensions (row, column, latitude, longitude, etc.) are nothing more than a human notational device. In general, you can assign any meaning you wish to the dimensions of an array, and as long as your use of those dimensions is consistent, you will get the correct answer, regardless of the order in which IDL chooses to store the actual array elements in computer memory. Thus, it is usually possible to ignore these issues. There are times however, when understanding memory layout can be important:

Sharing Data With Other Languages

If binary data written by a row major language is to be input and used by IDL, transposition of the data is usually required first. Similarly, if IDL is writing binary data for use by a program written in a row major language, transposition of the data before writing (or on input by the other program) is often required.

Calling Code Written In Other Languages

When passing IDL data to code written in a row major language via dynamic linking (CALL_EXTERNAL, LINKIMAGE, DLMS), it is often necessary to transpose the data before passing it to the called code, and to transpose the results.

Matrix Multiplication

Understanding the difference between the IDL # and ## operators requires an understanding of array layout. For a discussion of how the ordering of such data relates to IDL mathematics routines, see Manipulating Arrays.

1-D Subscripting Of Multidimensional Array

IDL allows you to index multidimensional arrays using a single 1-D subscript. For example, given a two dimensional 5x7 array, ARRAY[2,3] and ARRAY[17] refer to the same array element. Knowing this requires an understanding of the actual array layout in memory ($d_2 \cdot m + d_1$, or $3 \cdot 5 + 2$, which yields 17).

Efficiency

Accessing memory in the wrong order can impose a severe performance penalty if your data is larger than the physical memory in your computer. Accessing elements of an array along the contiguous dimension minimizes the amount of memory paging required by the virtual memory subsystem of your computer hardware, and will therefore be the most efficient. Accessing memory across the non-contiguous dimension can cause each such access to occur on a different page of system memory. This forces the virtual memory subsystem into a cycle in which it must continually force current pages of memory to disk in order to make room for new pages, each of which is only momentarily accessed. This inefficient use of virtual memory is commonly known as thrashing.

Multidimensional array

In C/C++, we can define multidimensional arrays in simple words as array of arrays. Data in multidimensional arrays are stored in tabular form (in row major order).

General form of declaring N-dimensional arrays:

```
data_type array_name[size1][size2]....[sizeN];
```

data_type: Type of data to be stored in the array.

Here data_type is valid C/C++ data type

array_name: Name of the array

size1, size2,... ,sizeN: Sizes of the dimensions

Examples:

Two dimensional array:

```
int two_d[10][20];
```

Three dimensional array:

```
int three_d[10][20][30];
```

Size of multidimensional arrays

Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.

For example:

The array **int x[10][20]** can store total $(10*20) = 200$ elements.

Similarly array **int x[5][10][20]** can store total $(5*10*20) = 1000$ elements.

Two-Dimensional Array

Two – dimensional array is the simplest form of a multidimensional array. We can see a two – dimensional array as an array of one – dimensional array for easier understanding.

- The basic form of declaring a two-dimensional array of size x, y:

Syntax:

- **data_type array_name[x][y];**
- **data_type:** Type of data to be stored. Valid C/C++ data type.
- We can declare a two dimensional integer array say 'x' of size 10,20 as:
- `int x[10][20];`
- Elements in two-dimensional arrays are commonly referred by `x[i][j]` where i is the row number and 'j' is the column number.
- A two – dimensional array can be seen as a table with 'x' rows and 'y' columns where the row number ranges from 0 to (x-1) and column number ranges from 0 to (y-1). A two – dimensional array 'x' with 3 rows and 3 columns is shown below:

Initializing Two – Dimensional Arrays: There are two ways in which a Two-Dimensional array can be initialized.

First Method:

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

The above array have 3 rows and 4 columns. The elements in the braces from left to right are stored in the table also from left to right. The elements will be filled in the array in the order, first 4 elements from the left in first row, next 4 elements in second row and so on.

Better Method:

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

This type of initialization make use of nested braces. Each set of inner braces represents one row. In the above example there are total three rows so there are three sets of inner braces.

Accessing Elements of Two-Dimensional Arrays: Elements in Two-Dimensional arrays are accessed using the row indexes and column indexes.

Example:

```
int x[2][1];
```

The above example represents the element present in third row and second column.

Note: In arrays if size of array is N. Its index will be from 0 to N-1. Therefore, for row index 2 row number is $2+1 = 3$.

To output all the elements of a Two-Dimensional array we can use nested for loops. We will require two for loops. One to traverse the rows and another to traverse columns.

filter_none

edit

play_arrow

brightness_4

```
// C++ Program to print the elements of a
```

```
// Two-Dimensional array
```

```
#include<iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    // an array with 3 rows and 2 columns.
```

```
    int x[3][2] = {{0,1}, {2,3}, {4,5}};
```

```
    // output each array element's value
```

```
    for (int i = 0; i < 3; i++)
```

```
    {
```

```
        for (int j = 0; j < 2; j++)
```



```

    {
        cout << "Element at x[" << i
            << "]" << j << "]: ";
        cout << x[i][j]<<endl;
    }
}

return 0;
}

```

Output:

```

Element at x[0][0]: 0
Element at x[0][1]: 1
Element at x[1][0]: 2
Element at x[1][1]: 3
Element at x[2][0]: 4
Element at x[2][1]: 5

```

Three-Dimensional Array

Initializing Three-Dimensional Array: Initialization in Three-Dimensional array is same as that of Two-dimensional arrays. The difference is as the number of dimension increases so the number of nested braces will also increase.

Method 1:

```

int x[2][3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                11, 12, 13, 14, 15, 16, 17, 18, 19,
                20, 21, 22, 23};

```

Better Method:

```
int x[2][3][4] =
{
  { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },
  { {12,13,14,15}, {16,17,18,19}, {20,21,22,23} }
};
```

Accessing elements in Three-Dimensional Arrays:

Accessing elements in Three-Dimensional Arrays is also similar to that of Two-Dimensional Arrays. The difference is we have to use three loops instead of two loops for one additional dimension in Three-dimensional Arrays.

filter_none

edit

play_arrow

brightness_4

// C++ program to print elements of Three-Dimensional

// Array

```
#include<iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    // initializing the 3-dimensional array
```

```
    int x[2][3][2] =
```

```
    {
```

```
        { {0,1}, {2,3}, {4,5} },
```

```
        { {6,7}, {8,9}, {10,11} }
```

```

};

// output each element's value
for (int i = 0; i < 2; ++i)
{
    for (int j = 0; j < 3; ++j)
    {
        for (int k = 0; k < 2; ++k)
        {
            cout << "Element at x[" << i << "]" << j
                << "[" << k << "] = " << x[i][j][k]
                << endl;
        }
    }
}

return 0;
}

```

Output:

```

Element at x[0][0][0] = 0
Element at x[0][0][1] = 1
Element at x[0][1][0] = 2
Element at x[0][1][1] = 3
Element at x[0][2][0] = 4
Element at x[0][2][1] = 5
Element at x[1][0][0] = 6
Element at x[1][0][1] = 7

```

Element at $x[1][1][0] = 8$

Element at $x[1][1][1] = 9$

Element at $x[1][2][0] = 10$

Element at $x[1][2][1] = 11$

In similar ways, we can create arrays with any number of dimension. However the complexity also increases as the number of dimension increases.

The most used multidimensional array is the Two-Dimensional Array.

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Attention reader! Don't stop learning now. Get hold of all the important DSA concepts with the **DSA Self Paced Course** at a student-friendly price and become industry ready.

UNIT-II

Pointers

Definition and declaration

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined –

```
#include <stdio.h>

int main () {

    int var1;
    char var2[10];

    printf("Address of var1 variable: %x\n", &var1 );
    printf("Address of var2 variable: %x\n", &var2 );

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Address of var1 variable: bff5a400

Address of var2 variable: bff5a3f6

What are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations

–

```
int *ip; /* pointer to an integer */
double *dp; /* pointer to a double */
float *fp; /* pointer to a float */
char *ch /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations –

Live Demo

```
#include <stdio.h>

int main () {

    int var = 20; /* actual variable declaration */
    int *ip; /* pointer variable declaration */

    ip = &var; /* store address of var in pointer variable*/

    printf("Address of var variable: %x\n", &var );

    /* address stored in pointer variable */
    printf("Address stored in ip variable: %x\n", ip );

    /* access the value using the pointer */
    printf("Value of *ip variable: %d\n", *ip );
```

```
return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program –

Live Demo

```
#include <stdio.h>

int main () {

    int *ptr = NULL;

    printf("The value of ptr is : %x\n", ptr );

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
The value of ptr is 0
```

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows –

```
if(ptr) /* succeeds if p is not null */
if(!ptr) /* succeeds if p is null */
```

Pointers in Detail

Pointers have many but easy concepts and they are very important to C programming. The following important pointer concepts should be clear to any C programmer –

Sr.No.	Concept & Description
1	Pointer arithmetic There are four arithmetic operators that can be used in pointers: ++, --, +, -
2	Array of pointers You can define arrays to hold a number of pointers.
3	Pointer to pointer C allows you to have pointer on a pointer and so on.
4	Passing pointers to functions in C Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.
5	Return pointer from functions in C C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well.

In this tutorial, we will learn how to declare, initialize and use a pointer. We will also learn what NULL pointer are and where to use them. Let's start

Declaration of C Pointer variable

General syntax of pointer declaration is,

```
datatype *pointer_name;
```

Data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing. void type pointer works with all data types, but is not often used.

Here are a few examples:

```
int *ip    // pointer to integer variable
float *fp; // pointer to float variable
double *dp; // pointer to double variable
char *cp;  // pointer to char variable
```

Initialization of C Pointer variable

Pointer Initialization is the process of assigning address of a variable to a **pointer** variable. Pointer variable can only contain address of a variable of the same data type. In C language **address operator &** is used to determine the address of a variable. The **&** (immediately preceding a variable name) returns the address of the variable associated with it.

```
#include<stdio.h>

void main()
{
    int a = 10;
    int *ptr;    //pointer declaration
    ptr = &a;    //pointer initialization
}
```

Pointer variable always point to variables of same datatype. Let's have an example to showcase this:

```
#include<stdio.h>

void main()
{
```

```
float a;

int *ptr;

ptr = &a;    // ERROR, type mismatch
}
```

If you are not sure about which variable's address to assign to a pointer variable while declaration, it is recommended to assign a NULL value to your pointer variable. A pointer which is assigned a NULL value is called a **NULL pointer**.

```
#include <stdio.h>

int main()
{
    int *ptr = NULL;

    return 0;
}
```

Using the pointer or Dereferencing of Pointer

Once a pointer has been assigned the address of a variable, to access the value of the variable, pointer is **dereferenced**, using the **indirection operator** or **dereferencing operator** `*`.

```
#include <stdio.h>

int main()
{
    int a, *p; // declaring the variable and pointer
```

```

a = 10;

p = &a; // initializing the pointer

printf("%d", *p); //this will print the value of 'a'

printf("%d", *&a); //this will also print the value of 'a'

printf("%u", &a); //this will print the address of 'a'

printf("%u", p); //this will also print the address of 'a'

printf("%u", &p); //this will print the address of 'p'

return 0;
}

```

Points to remember while using pointers

1. While declaring/initializing the pointer variable, `*` indicates that the variable is a pointer.
2. The address of any variable is given by preceding the variable name with Ampersand `&`.
3. The pointer variable stores the address of a variable. The declaration `int *a` doesn't mean that `a` is going to contain an integer value. It means that `a` is going to contain the address of a variable storing integer value.
4. To access the value of a certain address stored by a pointer variable, `*` is used. Here, the `*` can be read as **'value at'**.

Time for an Example!

Let's take a simple code example,

```
#include <stdio.h>

int main()
{
    int i = 10;    // normal integer variable storing value 10
    int *a;       // since '*' is used, hence its a pointer variable

    /*
        '&' returns the address of the variable 'i'
        which is stored in the pointer variable 'a'
    */
    a = &i;

    /*
        below, address of variable 'i', which is stored
        by a pointer variable 'a' is displayed
    */
    printf("Address of variable i is %u\n", a);

    /*
        below, '*a' is read as 'value at a'
        which is 10
    */
}
```

```
printf("Value at the address, which is stored by pointer variable a is %d\n", *a);

return 0;
}
```

Address of variable i is 2686728 (The address may vary)

Value at an address, which is stored by pointer variable a is 10

Initialization

A declaration of an object may provide its initial value through the process known as initialization.

For each declarator, the initializer, if not omitted, may be one of the following:

= expression (1)

= { initializer-list } (2)

where initializer-list is a non-empty comma-separated list of initializers (with an optional trailing comma), where each initializer has one of three possible forms:

expression (1)

{ initializer-list } (2)

designator-list = initializer (3) (since C99)

where designator-list is a list of either array designators of the form [constant-expression] or struct/union member designators of the form . identifier; see array initialization and struct initialization. (since C99)

Note: besides initializers, brace-enclosed initializer-list may appear

in compound literals, which are expressions of the form:

```
( type ) { initializer-list }
```

Explanation

The initializer specifies the initial value stored in an object.

Explicit initialization

If an initializer is provided, see

- scalar initialization for the initialization of scalar types
- array initialization for the initialization of array types
- struct initialization for the initialization of struct and union types.

Implicit initialization

If an initializer is not provided:

- objects with automatic storage duration are initialized to indeterminate values (which may be trap representations)
- objects with static and thread-local storage duration are zero-initialized

Zero initialization

In some cases, an object is zero-initialized if it is not initialized explicitly, that is:

- pointers are initialized to null pointer values of their types
- objects of integral types are initialized to unsigned zero
- objects of floating types are initialized to positive zero
- all elements of arrays, all members of structs, and the first members of unions are zero-initialized, recursively, plus all padding bits are initialized to zero

(on platforms where null pointer values and floating zeroes have all-bit-zero representations, this form of initialization for statics is normally implemented by

allocating them in the .bss section of the program image)

Notes

When initializing an object of static or thread-local storage duration, every expression in the initializer must be a constant expression or string literal.

Initializers cannot be used in declarations of objects of incomplete type, VLAs, and block-scope objects with linkage.

The initial values of function parameters are established as if by assignment from the arguments of a function call, rather than by initialization.

If an indeterminate value is used as an argument to any standard library call, the behavior is undefined. Otherwise, the result of any expression involving indeterminate values is an indeterminate value (e.g. `int n;`, `n` may not compare equal to itself and it may appear to change its value on subsequent reads)

The term zero initialization is not used in C standard. It is adopted from C++ and used here for convenience of explanation.

There is no special construct in C corresponding value initialization in C++, however, `= {0}` (or `(T){0}` in compound literals) (since C99) can be used instead, as C standard does not allow empty structs, empty unions, or arrays of zero length.

Example

```
#include <stdlib.h>
int a[2]; // initializes a to {0, 0}
int main(void)
{
    int i;      // initializes i to an indeterminate value
    static int j; // initializes j to 0
    int k = 1;  // initializes k to 1

    // initializes int x[3] to 1,3,5
    // initializes int* p to &x[0]
    int x[] = { 1, 3, 5 }, *p = x;

    // initializes w (an array of two structs) to
    // { { {1,0,0}, 0}, { {2,0,0}, 0} }
    struct {int a[3], b;} w[] = {[0].a = {1}, [1].a[0] = 2};

    // function call expression can be used for a local variable
    char* ptr = malloc(10);
```

```
free(ptr);

// Error: objects with static storage duration require constant initializers
// static char* ptr = malloc(10);

// Error: VLA cannot be initialized
// int vla[n] = {0};
}
```

Indirection operator

An indirection operator, in the context of C#, is an operator used to obtain the value of a variable to which a pointer points. While a pointer pointing to a variable provides an indirect access to the value of the variable stored in its memory address, the indirection operator dereferences the pointer and returns the value of the variable at that memory location. The indirection operator is a unary operator represented by the symbol (*).

The indirection operator can be used in a pointer to a pointer to an integer, a single-dimensional array of pointers to integers, a pointer to a char, and a pointer to an unknown type.

The indirection operator is also known as the dereference operator.

The (*) symbol is used in declaring pointer types and in performing pointer indirection, while the 'address-of' operator (&) returns the address of a variable. Hence, the indirection operator and the address-of operator are inverses of each other.

C# allows using pointers only in an unsafe region, which implies that the safety of the code within that region is not verified by the common language runtime (CLR). In the unsafe region, the indirection operator is allowed to read and write to a pointer. The following C# statements illustrate the usage of the indirection operator:

- `int a = 1, b; // line 1`
- `int *pInt = &a; // line 2`
- `b = *pInt; // line 3`

In the first line above, a and b are integer variables and a is assigned a value of 1. In line 2, the address of a is stored in the integer pointer pInt (line 2). The dereference operator is used in line 3 to assign the value at the address pointed to by pInt to the integer variable b.

The indirection operator should be used to dereference a valid pointer with an address aligned to the type it points to, so as to avoid undefined behavior at runtime. It should not be applied to a void pointer or to an expression that is not of a pointer type, to avoid compiler errors. However, after casting a void pointer to the right pointer type, the indirection operator can be used.

When declaring multiple pointers in a single statement, the indirection operator should be written only once with the underlying type and not repeated for each pointer name. The indirection operator is distributive in C#, unlike C and C++. When the indirection operator is applied to a null pointer, it results in an implementation-defined behavior. Since this operator is used in an unsafe context, the keyword `unsafe` should be used before it along with the `/unsafe` option during compilation. This definition was written in the context of C#

address of operator

An address-of operator is a mechanism within C++ that returns the memory address of a variable. These addresses returned by the address-of operator are known as pointers, because they "point" to the variable in memory.

The address-of operator is a unary operator represented by an ampersand (&). It is also known as an address operator.

Address operators commonly serve two purposes:

1. To conduct parameter passing by reference, such as by name
2. To establish pointer values. Address-of operators point to the location in the memory because the value of the pointer is the memory address/location where the data item resides in memory.

For example, if the user is trying to locate age 26 within the data, the integer variable would be named `age` and it would look like this: `int age = 26`. Then the address operator is used to determine the location, or the address, of the data using `"&age"`.

From there, the Hex value of the address can be printed out using `"cout << &age"`. Integer values need to be output to a long data type. Here the address location would read `"cout << long (&age)"`.

The address-of operator can only be applied to variables with fundamental, structure, class, or union types that are declared at the file-scope level, or to subscripted array

references. In these expressions, a constant expression that does not include the address-of operator can be added to or subtracted from the address-of expression. This definition was written in the context of C++

pointer arithmetic

Pointers variables are also known as address data types because they are used to store the address of another variable. The address is the memory location that is assigned to the variable. It doesn't store any value.

Hence, there are only a few operations that are allowed to perform on Pointers in C language. The operations are slightly different from the ones that we generally use for mathematical calculations. The operations are:

1. Increment/Decrement of a Pointer
2. Addition of integer to a pointer
3. Subtraction of integer to a pointer
4. Subtracting two pointers of the same type

Increment/Decrement of a Pointer

Increment: It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores **address 1000** is incremented, then it will increment by 2(**size of an int**) and the new address it will points to **1002**. While if a float type pointer is incremented then it will increment by 4(**size of a float**) and the new address will be **1004**.

Decrement: It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.

For Example:

If an integer pointer that stores **address 1000** is decremented, then it will decrement by

2(**size of an int**) and the new address it will points to **998**. While if a float type pointer is decremented then it will decrement by 4(**size of a float**) and the new address will be **996**. Below is the program to illustrate pointer increment/decrement:

```
// C program to illustrate  
// pointer increment/decrement
```

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

```
// Driver Code
```

```
int main()
```

```
{
```

```
    // Integer variable
```

```
    int N = 4;
```

```
    // Pointer to an integer
```

```
    int *ptr1, *ptr2;
```

```
    // Pointer stores
```

```
    // the address of N
```

```
    ptr1 = &N;
```

```
    ptr2 = &N;
```

```
    printf("Pointer ptr1 "
```

```
        "before Increment: ");
```

```
    printf("%p \n", ptr1);
```

```
    // Incrementing pointer ptr1;
```

```
    ptr1++;
```

```
    printf("Pointer ptr1 after"
```

```
        " Increment: ");
```

```
    printf("%p \n\n", ptr1);
```

```
    printf("Pointer ptr1 before"
```

```
        " Decrement: ");
```

```
    printf("%p \n", ptr1);
```

```
    // Decrementing pointer ptr1;
```

```

ptr1--;

printf("Pointer ptr1 after"
      " Decrement: ");
printf("%p \n\n", ptr1);

return 0;
}

```

Output:

```

Pointer ptr1 before Increment: 0x7ffcb19385e4
Pointer ptr1 after Increment: 0x7ffcb19385e8

Pointer ptr1 before Decrement: 0x7ffcb19385e8
Pointer ptr1 after Decrement: 0x7ffcb19385e4

```

Addition

When a pointer is added with a value, the value is first multiplied by the size of data type and then added to the pointer.

```

filter_none
edit
play_arrow
brightness_4

```

```

// C program to illustrate pointer Addition
#include <stdio.h>

```

```

// Driver Code
int main()
{
    // Integer variable
    int N = 4;

    // Pointer to an integer
    int *ptr1, *ptr2;

```

```

// Pointer stores the address of N
ptr1 = &N;
ptr2 = &N;

printf("Pointer ptr2 before Addition: ");
printf("%p \n", ptr2);

// Addition of 3 to ptr2
ptr2 = ptr2 + 3;
printf("Pointer ptr2 after Addition: ");
printf("%p \n", ptr2);

return 0;
}

```

Output:

```

Pointer ptr2 before Addition: 0x7ffffdcd984
Pointer ptr2 after Addition: 0x7ffffdcd990

```

Subtraction

When a pointer is subtracted with a value, the value is first multiplied by the size of the data type and then subtracted from the pointer.

Below is the program to illustrate pointer Subtraction:

```

filter_none
edit
play_arrow
brightness_4

// C program to illustrate pointer Subtraction
#include <stdio.h>

// Driver Code
int main()
{
    // Integer variable
    int N = 4;

```

```

// Pointer to an integer
int *ptr1, *ptr2;

// Pointer stores the address of N
ptr1 = &N;
ptr2 = &N;

printf("Pointer ptr2 before Subtraction: ");
printf("%p \n", ptr2);

// Subtraction of 3 to ptr2
ptr2 = ptr2 - 3;
printf("Pointer ptr2 after Subtraction: ");
printf("%p \n", ptr2);

return 0;
}

```

Output:

```

Pointer ptr2 before Subtraction: 0x7ffcf1221b24
Pointer ptr2 after Subtraction: 0x7ffcf1221b18

```

Subtraction of Two Pointers

The subtraction of two pointers is possible only when they have the same data type. The result is generated by calculating the difference between the addresses of the two pointers and calculating how many bits of data it is according to the pointer data type. The subtraction of two pointers gives the increments between the two pointers.

For Example:

Two integer pointers say **ptr1(address:1000)** and **ptr2(address:1016)** are subtracted. The difference between address is 16 bytes. Since the size of int is 2 bytes, therefore the **increment between ptr1 and ptr2** is given by **(16/2) = 8**.

Below is the implementation to illustrate the Subtraction of Two Pointers:

```

filter_none
edit

```

```

play_arrow
brightness_4

// C program to illustrate Subtraction
// of two pointers
#include <stdio.h>

// Driver Code
int main()
{
    int x;

    // Integer variable
    int N = 4;

    // Pointer to an integer
    int *ptr1, *ptr2;

    // Pointer stores the address of N
    ptr1 = &N;
    ptr2 = &N;

    // Incrementing ptr2 by 3
    ptr2 = ptr2 + 3;

    // Subtraction of ptr2 and ptr1
    x = ptr2 - ptr1;

    // Print x to get the Increment
    // between ptr1 and ptr2
    printf("Subtraction of ptr1 "
           "& ptr2 is %d\n",
           x);

    return 0;
}

```

Output:

```
Subtraction of ptr1 & ptr2 is 3
```

Pointer Arithmetic on Arrays:

Pointers contain addresses. Adding two addresses makes no sense because there is no idea what it would point to. Subtracting two addresses lets you compute the offset between the two addresses. An array name acts like a pointer constant. The value of this pointer constant is the address of the first element. **For Example:** if an array named arr then arr and &arr[0] can be used to reference array as a pointer.

Below is the program to illustrate the Pointer Arithmetic on arrays:

Program 1:

```
filter_none
edit
play_arrow
brightness_4

// C program to illustrate the array
// traversal using pointers
#include <stdio.h>

// Driver Code
int main()
{

    int N = 5;

    // An array
    int arr[] = { 1, 2, 3, 4, 5 };

    // Declare pointer variable
    int* ptr;

    // Point the pointer to first
    // element in array arr[]
    ptr = arr;

    // Traverse array using ptr
    for (int i = 0; i < N; i++) {

        // Print element at which
        // ptr points
```



```
    printf("%d ", ptr[0]);
    ptr++;
}
}
```

Output:

```
1 2 3 4 5
```

Program 2:

```
filter_none
```

```
edit
```

```
play_arrow
```

```
brightness_4
```

```
// C program to illustrate the array
// traversal using pointers in 2D array
#include <stdio.h>
```

```
// Function to traverse 2D array
// using pointers
```

```
void traverseArr(int* arr,
                int N, int M)
```

```
{
```

```
    int i, j;
```

```
    // Traversing rows of 2D matrix
    for (i = 0; i < N; i++) {
```

```
        // Traversing columns of 2D matrix
        for (j = 0; j < M; j++) {
```

```
            // Print the element
            printf("%d ", *((arr + i * M) + j));
```

```
        }
        printf("\n");
```

```
    }
}
```

```
// Driver Code
```

```

int main()
{

    int N = 3, M = 2;

    // A 2D array
    int arr[][2] = { { 1, 2 },
                    { 3, 4 },
                    { 5, 6 } };

    // Function Call
    traverseArr((int*)arr, N, M);
    return 0;
}

```

Output:

```

1 2
3 4
5 6

```

dynamic memory allocation

Since C is a structured language, it has some fixed rules for programming. One of it includes changing the size of an array. An array is collection of items stored at continuous memory locations.

As it can be seen that the length (size) of the array above made is 9. But what if there is a requirement to change this length (size). For Example,

- If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
- Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

This procedure is referred to as **Dynamic Memory Allocation in C**.

Therefore, **C Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

Let's look at each of them in greater detail.

1. C malloc() method

“**malloc**” or “**memory allocation**” method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It initializes each block with default garbage value.

Syntax:

```
ptr = (cast-type*) malloc(byte-size)
```

For Example:

```
ptr = (int*) malloc(100 * sizeof(int));
```

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

If space is insufficient, allocation fails and returns a NULL pointer.

Example:

```
filter_none

edit
play_arrow
brightness_4
#include <stdio.h>

#include <stdlib.h>

int main()
{

// This pointer will hold the
// base address of the block created
int* ptr;
int n, i;

// Get the number of elements for the array
n = 5;
printf("Enter number of elements: %d\n", n);

// Dynamically allocate memory using malloc()
ptr = (int*)malloc(n * sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL) {
```

```

printf("Memory not allocated.\n");
exit(0);
}
else {

// Memory has been successfully allocated
printf("Memory successfully allocated using malloc.\n");

// Get the elements of the array
for (i = 0; i < n; ++i) {
ptr[i] = i + 1;
}

// Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
printf("%d, ", ptr[i]);
}
}

return 0;
}

```

Output:

```

Enter number of elements: 5
Memory successfully allocated using malloc.
The elements of the array are: 1, 2, 3, 4, 5,

```

2. C calloc() method

“**calloc**” or “**contiguous allocation**” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.

Syntax:

```
ptr = (cast-type*)calloc(n, element-size);
```

For Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for 25 elements each with the size of the float.

If space is insufficient, allocation fails and returns a NULL pointer.

Example:

```
filter_none
```

```
edit
```

```
play_arrow
```

```
brightness_4
```

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

```
#include <stdlib.h>
```

```
int main()
```

```
{
```

```
// This pointer will hold the
```

```
// base address of the block created
```

```
int* ptr;
```

```
int n, i;
```

```
// Get the number of elements for the array
n = 5;
printf("Enter number of elements: %d\n", n);

// Dynamically allocate memory using calloc()
ptr = (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully
// allocated by calloc or not
if (ptr == NULL) {
printf("Memory not allocated.\n");
exit(0);
}
else {

// Memory has been successfully allocated
printf("Memory successfully allocated using calloc.\n");

// Get the elements of the array
for (i = 0; i < n; ++i) {
ptr[i] = i + 1;
}

// Print the elements of the array
```

```

printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
printf("%d, ", ptr[i]);
}
}

return 0;
}

```

Output:

```

Enter number of elements: 5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5,

```

3. C free() method

“**free**” method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

```
free(ptr);
```

Example:

```

filter_none

edit
play_arrow
brightness_4
#include <stdio.h>

#include <stdlib.h>

```



```

int main()
{

// This pointer will hold the
// base address of the block created
int *ptr, *ptr1;
int n, i;

// Get the number of elements for the array
n = 5;
printf("Enter number of elements: %d\n", n);

// Dynamically allocate memory using malloc()
ptr = (int*)malloc(n * sizeof(int));

// Dynamically allocate memory using calloc()
ptr1 = (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL || ptr1 == NULL) {
printf("Memory not allocated.\n");
exit(0);
}
else {

```

```
// Memory has been successfully allocated
printf("Memory successfully allocated using malloc.\n");

// Free the memory
free(ptr);
printf("Malloc Memory successfully freed.\n");

// Memory has been successfully allocated
printf("\nMemory successfully allocated using calloc.\n");

// Free the memory
free(ptr1);
printf("Calloc Memory successfully freed.\n");
}

return 0;
}
```

Output:

```
Enter number of elements: 5
Memory successfully allocated using malloc.
Malloc Memory successfully freed.

Memory successfully allocated using calloc.
Calloc Memory successfully freed.
```

4. C realloc() method

“**realloc**” or “**re-allocation**” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value.

Syntax:

```
ptr = realloc(ptr, newSize);
```

where ptr is reallocated with new size 'newSize'.

If space is insufficient, allocation fails and returns a NULL pointer.

Example:

```
filter_none
```

```
edit
```

```
play_arrow
```

```
brightness_4
```

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

```
#include <stdlib.h>
```

```
int main()
```

```
{
```

```
// This pointer will hold the
```

```
// base address of the block created
```

```
int* ptr;
```

```
int n, i;
```

```
// Get the number of elements for the array
n = 5;
printf("Enter number of elements: %d\n", n);

// Dynamically allocate memory using calloc()
ptr = (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL) {
printf("Memory not allocated.\n");
exit(0);
}
else {

// Memory has been successfully allocated
printf("Memory successfully allocated using calloc.\n");

// Get the elements of the array
for (i = 0; i < n; ++i) {
ptr[i] = i + 1;
}

// Print the elements of the array
```

```

printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
printf("%d, ", ptr[i]);
}

// Get the new size for the array
n = 10;
printf("\n\nEnter the new size of the array: %d\n", n);

// Dynamically re-allocate memory using realloc()
ptr = realloc(ptr, n * sizeof(int));

// Memory has been successfully allocated
printf("Memory successfully re-allocated using realloc.\n");

// Get the new elements of the array
for (i = 5; i < n; ++i) {
ptr[i] = i + 1;
}

// Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
printf("%d, ", ptr[i]);
}

```

```
free(ptr);  
}
```

```
return 0;  
}
```

Output:

```
Enter number of elements: 5  
Memory successfully allocated using calloc.  
The elements of the array are: 1, 2, 3, 4, 5,  
  
Enter the new size of the array: 10  
Memory successfully re-allocated using realloc.  
The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

arrays and pointers

In this tutorial, you'll learn about the relationship between arrays and pointers in C programming. You will also learn to access array elements using pointers.

Before you learn about the relationship between arrays and pointers, be sure to check these two topics:

- C Arrays
- C Pointers

Relationship Between Arrays and Pointers

An array is a block of sequential data. Let's write a program to print addresses of array elements.

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

```

int main() {
    int x[4];
    int i;

    for(i = 0; i < 4; ++i) {
        printf("&x[%d] = %p\n", i, &x[i]);
    }

    printf("Address of array x: %p", x);

    return 0;
}

```

Output

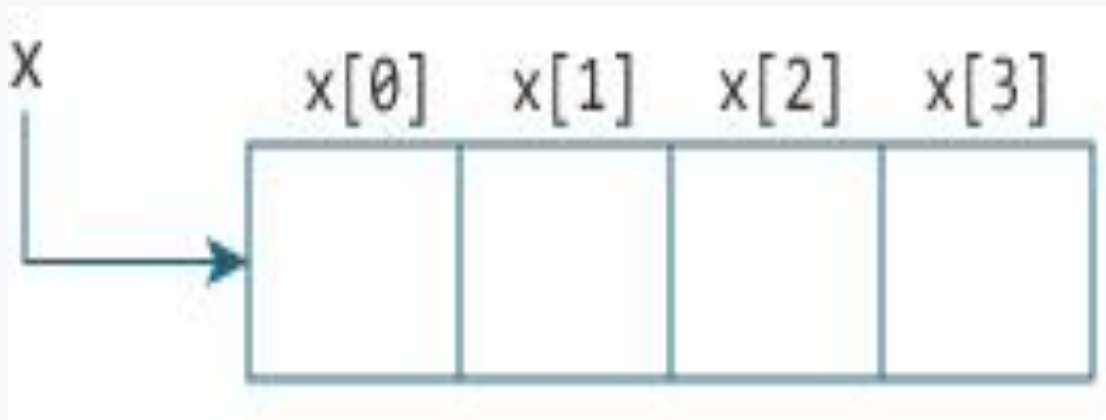
```

&x[0] = 1450734448
&x[1] = 1450734452
&x[2] = 1450734456
&x[3] = 1450734460
Address of array x: 1450734448

```

There is a difference of 4 bytes between two consecutive elements of array `x`. It is because the size of `int` is 4 bytes (on our compiler).

Notice that, the address of `&x[0]` and `x` is the same. It's because the variable name `x` points to the first element of the array.



From the above example, it is clear that `&x[0]` is equivalent to `x`. And, `x[0]` is equivalent to `*x`.

Similarly,

- `&x[1]` is equivalent to `x+1` and `x[1]` is equivalent to `*(x+1)`.
- `&x[2]` is equivalent to `x+2` and `x[2]` is equivalent to `*(x+2)`.
- ...
- Basically, `&x[i]` is equivalent to `x+i` and `x[i]` is equivalent to `*(x+i)`.

Example 1: Pointers and Arrays

```
#include <stdio.h>
int main() {
    int i, x[6], sum = 0;
    printf("Enter 6 numbers: ");
    for(i = 0; i < 6; ++i) {
        // Equivalent to scanf("%d", &x[i]);
        scanf("%d", x+i);

        // Equivalent to sum += x[i]
        sum += *(x+i);
    }
    printf("Sum = %d", sum);
    return 0;
}
```

When you run the program, the output will be:

```
Enter 6 numbers: 2
3
4
4
12
4
Sum = 29
```


Here, we have declared an array `x` of 6 elements. To access elements of the array, we have used pointers.

In most contexts, array names decay to pointers. In simple words, array names are converted to pointers. That's the reason why you can use pointers to access elements of arrays. However, you should remember that **pointers and arrays are not the same**. There are a few cases where array names don't decay to pointers. To learn more, visit: [When does array name doesn't decay into a pointer?](#)

Example 2: Arrays and Pointers

```
#include <stdio.h>
int main() {
    int x[5] = {1, 2, 3, 4, 5};
    int* ptr;

    // ptr is assigned the address of the third element
    ptr = &x[2];

    printf("*ptr = %d \n", *ptr); // 3
    printf("*ptr+1 = %d \n", *(ptr+1)); // 4
    printf("(ptr-1) = %d", *(ptr-1)); // 2

    return 0;
}
```

When you run the program, the output will be:

```
*ptr = 3
*(ptr+1) = 4
*(ptr-1) = 2
```

In this example, `&x[2]`, the address of the third element, is assigned to the `ptr` pointer. Hence, `3` was displayed when we printed `*ptr`.

And, printing `*(ptr+1)` gives us the fourth element. Similarly, printing `*(ptr-1)` gives us the second element.

function and pointers

Pointers give greatly possibilities to 'C' functions which we are limited to return one value. With pointer parameters, our functions now can process actual data rather than a copy of data.

In order to modify the actual values of variables, the calling statement passes addresses to pointer parameters in a function.

In this tutorial, you will learn-

- Functions Pointers Example
- Functions with Array Parameters
- Functions that Return an Array
- Function Pointers
- Array of Function Pointers
- Functions Using void Pointers
- Function Pointers as Arguments

Functions Pointers Example

For example, the next program swaps two values of two:

```
void swap (int *a, int *b);
int main() {
    int m = 25;
    int n = 100;
    printf("m is %d, n is %d\n", m, n);
    swap(&m, &n);
    printf("m is %d, n is %d\n", m, n);
    return 0;}
void swap (int *a, int *b) {
    int temp;
    temp = *a;
    *a = *b;
```

```
*b = temp;}  
}
```

Output:

```
m is 25, n is 100  
m is 100, n is 25
```

The program swaps the actual variables values because the function accesses them by address using pointers. Here we will discuss the program process:

1. We declare the function responsible for swapping the two variable values, which takes two integer pointers as parameters and returns any value when it is called.
2. In the main function, we declare and initialize two integer variables ('m' and 'n') then we print their values respectively.
3. We call the swap() function by passing the address of the two variables as arguments using the ampersand symbol. After that, we print the new swapped values of variables.
4. Here we define the swap() function content which takes two integer variable addresses as parameters and declare a temporary integer variable used as a third storage box to save one of the value variables which will be put to the second variable.
5. Save the content of the first variable pointed by 'a' in the temporary variable.
6. Store the second variable pointed by b in the first variable pointed by a.
7. Update the second variable (pointed by b) by the value of the first variable saved in the temporary variable.

Functions with Array Parameters

In C, we cannot pass an array by value to a function. Whereas, an array name is a pointer (address), so we just pass an array name to a function which means to pass a pointer to the array.

For example, we consider the following program:

```

int add_array (int *a, int num_elements);
int main() {
    int Tab[5] = {100, 220, 37, 16, 98};
    printf("Total summation is %d\n", add_array(Tab, 5));
    return 0;}
int add_array (int *p, int size) {
    int total = 0;
    int k;
    for (k = 0; k < size; k++) {
        total += p[k]; /* it is equivalent to total +=*p ;p++; */}
    return (total);}

```

Output:

```
Total summation is 471
```

Here, we will explain the program code with its details

1. We declare and define `add_array()` function which takes an array address (pointer) with its elements number as parameters and returns the total accumulated summation of these elements. The pointer is used to iterate the array elements (using the `p[k]` notation), and we accumulate the summation in a local variable which will be returned after iterating the entire element array.
2. We declare and initialize an integer array with five integer elements. We print the total summation by passing the array name (which acts as address) and array size to the **`add_array()`** called function as arguments.

Functions that Return an Array

In C, we can return a pointer to an array, as in the following program:

```

#include <stdio.h>
int * build_array();
int main() {
    int *a;
    a = build_array(); /* get first 5 even numbers */
    for (k = 0; k < 5; k++)
        printf("%d\n", a[k]);
}

```

```
return 0;}
int * build_array() {
    static int Tab[5]={1,2,3,4,5};
    return (Tab);}
```

Output:

```
1
2
3
4
5
```

And here, we will discuss the program details

1. We define and declare a function which returns an array address containing an integer value and didn't take any arguments.
2. We declare an integer pointer which receives the complete array built after the function is called and we print its contents by iterating the entire five element array.

Notice that a pointer, not an array, is defined to store the array address returned by the function. Also notice that when a local variable is being returned from a function, we have to declare it as static in the function.

Function Pointers

As we know by definition that pointers point to an address in any memory location, they can also point to at the beginning of executable code as functions in memory.

A pointer to function is declared with the * ,the general statement of its declaration is:

```
return_type (*function_name)(arguments)
```

You have to remember that the parentheses around (*function_name) are important because without them, the compiler will think the function_name is returning a pointer of return_type.

After defining the function pointer, we have to assign it to a function. For example, the next program declares an ordinary function, defines a function pointer, assigns the

function pointer to the ordinary function and after that calls the function through the pointer:

```
#include <stdio.h>
void Hi_function (int times); /* function */
int main() {
    void (*function_ptr)(int); /* function pointer Declaration */
    function_ptr = Hi_function; /* pointer assignment */
    function_ptr (3); /* function call */
    return 0;}
void Hi_function (int times) {
    int k;
    for (k = 0; k < times; k++) printf("Hi\n");}
```

Output:

```
Hi
Hi
Hi
```

1. We define and declare a standard function which prints a Hi text k times indicated by the parameter times when the function is called
2. We define a pointer function (with its special declaration) which takes an integer parameter and doesn't return anything.
3. We initialize our pointer function with the Hi_function which means that the pointer points to the Hi_function().
4. Rather than the standard function calling by taping the function name with arguments, we call only the pointer function by passing the number 3 as arguments, and that's it!

Keep in mind that the function name points to the beginning address of the executable code like an array name which points to its first element. Therefore, instructions like `function_ptr = &Hi_function` and `(*funptr)(3)` are correct.

NOTE: It is not important to insert the address operator `&` and the indirection operator `*` during the function assignment and function call.

Array of Function Pointers

An array of function pointers can play a switch or an if statement role for making a decision, as in the next program:

```
#include <stdio.h>
int sum(int num1, int num2);
int sub(int num1, int num2);
int mult(int num1, int num2);
int div(int num1, int num2);

int main()
{ int x, y, choice, result;
  int (*ope[4])(int, int);
  ope[0] = sum;
  ope[1] = sub;
  ope[2] = mult;
  ope[3] = div;
  printf("Enter two integer numbers: ");
  scanf("%d%d", &x, &y);
  printf("Enter 0 to sum, 1 to subtract, 2 to multiply, or 3 to divide: ");
  scanf("%d", &choice);
  result = ope[choice](x, y);
  printf("%d", result);
return 0;}

int sum(int x, int y) {return(x + y);}
int sub(int x, int y) {return(x - y);}
int mult(int x, int y) {return(x * y);}
int div(int x, int y) {if (y != 0) return (x / y); else return 0;}
Enter two integer numbers: 13 48
Enter 0 to sum, 1 to subtract, 2 to multiply, or 3 to divide: 2
624
```

Here, we discuss the program details:

1. We declare and define four functions which take two integer arguments and return an integer value. These functions add, subtract, multiply and divide the two

arguments regarding which function is being called by the user.

2. We declare 4 integers to handle operands, operation type, and result respectively. Also, we declare an array of four function pointer. Each function pointer of array element takes two integers parameters and returns an integer value.
3. We assign and initialize each array element with the function already declared. For example, the third element which is the third function pointer will point to multiplication operation function.
4. We seek operands and type of operation from the user typed with the keyboard.
5. We called the appropriate array element (Function pointer) with arguments, and we store the result generated by the appropriate function.

The instruction `int (*ope[4])(int, int);` defines the array of function pointers. Each array element must have the same parameters and return type.

The statement `result = ope[choice](x, y);` runs the appropriate function according to the choice made by the user. The two entered integers are the arguments passed to the function.

Functions Using void Pointers

Void pointers are used during function declarations. We use a `void *` return type permits to return any type. If we assume that our parameters do not change when passing to a function, we declare it as `const`.

For example:

```
void * cube (const void *);
```

Consider the following program:

```
#include <stdio.h>
void* cube (const void* num);
int main() {
    int x, cube_int;
    x = 4;
    cube_int = cube (&x);
```



```
printf("%d cubed is %d\n", x, cube_int);  
return 0;}
```

```
void* cube (const void *num) {  
    int result;  
    result = (*(int *)num) * (*(int *)num) * (*(int *)num);  
    return result;}
```

Result:

```
4 cubed is 64
```

Here, we will discuss the program details:

1. We define and declare a function that returns an integer value and takes an address of unchangeable variable without a specific data type. We calculate the cube value of the content variable (x) pointed by the num pointer, and as it is a void pointer, we have to type cast it to an integer data type using a specific notation (* datatype) pointer, and we return the cube value.
2. We declare the operand and the result variable. Also, we initialize our operand with value "4."
3. We call the cube function by passing the operand address, and we handle the returning value in the result variable

Function Pointers as Arguments

Another way to exploit a function pointer by passing it as an argument to another function sometimes called "callback function" because the receiving function "calls it back."

In the stdlib.h header file, the Quicksort "qsort()" function uses this technique which is an algorithm dedicated to sort an array.

```
void qsort(void *base, size_t num, size_t width, int (*compare)(const void *, const void *)  
)
```

- void *base : void pointer to the array.
- size_t num : The array element number.
- size_t width The element size.
- int (*compare (const void *, const void *) : function pointer composed of two arguments and returns 0 when the arguments have the same value, <0 when arg1 comes before arg2, and >0 when arg1 comes after arg2.

The following program sorts an integers array from small to big number using qsort() function:

```
#include <stdio.h>
#include <stdlib.h>
int compare (const void *, const void *);
int main() {
    int arr[5] = {52, 14, 50, 48, 13};
    int num, width, i;
    num = sizeof(arr)/sizeof(arr[0]);
    width = sizeof(arr[0]);
    qsort((void *)arr, num, width, compare);
    for (i = 0; i < 5; i++)
        printf("%d ", arr[ i ]);
    return 0;}

int compare (const void *elem1, const void *elem2) {
    if ((*int *)elem1 == *(int *)elem2) return 0;
    else if ((*int *)elem1 < *(int *)elem2) return -1;
    else return 1;}
```

Result:

```
13 14 48 50 52
```

Here, we will discuss the program details:

1. We define compare function composed of two arguments and returns 0 when the arguments have the same value, <0 when arg1 comes before arg2, and >0 when arg1 comes after arg2. The parameters are a void pointers type casted to the

appropriate array data type (integer)

2. We define and initialize an integer array The array size is stored in the **num** variable and the size of each array element is stored in width variable using sizeof() predefined C operator.
3. We call the **qsort** function and pass the array name, size, width, and comparison function defined previously by the user in order to sort our array in ascending order. The comparison will be performed by taking in each iteration two array elements until the entire array will be sorted.
4. We print the array elements to be sure that our array is well sorted by iterating the entire array using for loop.

UNIT-III

Strings

Definition

Strings are actually one-dimensional array of characters terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows –

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

Following is the memory presentation of the above defined string in C/C++ –

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string –

```

#include <stdio.h>

int main () {

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
printf("Greeting message: %s\n", greeting );
return 0;
}

```

When the above code is compiled and executed, it produces the following result –

Greeting message: Hello

C supports a wide range of functions that manipulate null-terminated strings –

Sr.No.	Function & Purpose
1	strcpy(s1, s2); Copies string s2 into string s1.
2	strcat(s1, s2); Concatenates string s2 onto the end of string s1.
3	strlen(s1); Returns the length of string s1.
4	strcmp(s1, s2); Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1.

6

strstr(s1, s2);

Returns a pointer to the first occurrence of string s2 in string s1.

The following example uses some of the above-mentioned functions –

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

int main () {

char str1[12] = "Hello";
char str2[12] = "World";
char str3[12];
int len ;

/* copy str1 into str3 */
strcpy(str3, str1);
printf("strcpy( str3, str1) : %s\n", str3 );

/* concatenates str1 and str2 */
strcat( str1, str2);
printf("strcat( str1, str2): %s\n", str1 );

/* total length of str1 after concatenation */
len = strlen(str1);
printf("strlen(str1) : %d\n", len );

return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

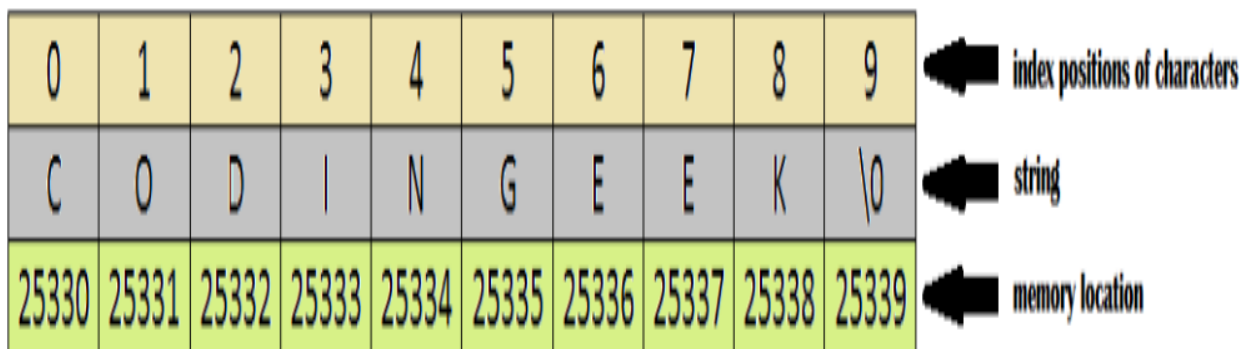
declaration and initialization of strings

A **string** is basically a character array which terminates with a `'\0'`. It can have alphabets in both cases, digits and special characters. It is like having a group of characters that forms words or sentences.

The `'\0'` is also known as the **null** character. It is placed at the end of any character array or string. For example:

```
char name={'C','O','D','I','N','G','E','E','K','\0'};
```

For some declarations, even if the programmer doesn't put the `'\0'` at the end of the character array the compiler will automatically do that. `'\0'` is only one character, although it might look like two. The ASCII code of `'\0'` is **0** which is not the same as `'0'` which has an ASCII code of **48**. `'\0'` is what distinguishes a string from a normal character array. The string related functions identify `'\0'` and recognize where a string ends. The figure shown below shows how a string is stored in a memory:



Note that the memory locations are random and it might differ for different compilers and systems.

DECLARATION AND INITIALIZATION OF STRING

Before starting our operations on **strings** we need to declare a string. There are two ways declaring a string. As a character array and as a pointer:

1. **char** name[10]; // as a character array
2. **char** *name; // as a pointer

A string can be initialized in many different ways. Here are those ways:

1. **char** name[]={ 'C','O','D','I','N','G','E','E','K','\0'}; //as an unsized array. This method requires the user to put a '\0' at the end
2. **char** name[10]={ 'C','O','D','I','N','G','E','E','K','\0'}; //as a sized array.
3. **char** name[]="CODINGEEK"; //unsized array. Puts '\0' automatically
4. **char** name[10]="CODINGEEK"; //sized array.

TAKING STRING INPUT FROM A USER

Scanf() is used to take inputs from the user. Strings can be taken as input using **scanf()** function. The format is given below:

1. **char** name[10];
2. **scanf**("%s", name);

As we already know that string might contain blank spaces, one limitation with **scanf()** is that it terminates as soon as it encounters a blank space. Example:

We enter: String input

Output: String

To avoid this complication we enter the elements of a string like any normal character array. Do not forget to add '\0' at the end of the string. The syntax is given below:

1. **char** name[20],ch;//declaration
2. **int** i=0;
3. **while**(ch!="\n")
4. {
5. ch=getchar();
6. name[i]=ch;
7. i++;
8. }

SIMPLE INPUT PROGRAM ON STRING

Here is a simple program on **string** which takes input from the user and displays the string:

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

2. int main()

3. {

4. char name[20],ch;//declaration of a character array

5. int i=0;

6. printf("Enter a string:\n");

7. //taking string input from the user

8. while(ch!='\n')

9. {

10.ch=getchar();

11.name[i]=ch;

12.i++;

13.}

14.name[i]='\0';//ending the string

15.i=0;

16.//display the string

17.printf("The string is:\n");

18.while(name[i]!='\0')
```

```
19. {  
20. printf("%c",name[i]);  
21. i++;  
22. }  
23. return 0;  
24. }
```

Output:-

Enter a string:

The bell rang

The string is:

The bell rang

standard library function

Have you ever wondered why Library Functions in C possess great importance in programming? Clear your confusion, because we are going to justify the importance of Library Functions in C through this tutorial and will cover all the important aspects related to it. These concepts will help you in a tremendous way to enhance your programming skills.

In this tutorial, we will discuss:

Library Functions in

**Standard
Library
Functions**

**Significance of
Standard
Library
Functions**

**Common
Header
Files**

In order to acknowledge the potency of standard library functions, let us consider a situation where you want to simply display a statement without the use of standard output statements, like **printf()** or **puts()** or any other inbuilt display functions. That would probably be a tedious task and an extensive knowledge of programming at the engineering level would be required to simply display a statement as your program output.

This is where the standard library functions in C come into play!

Before we move on it is required to be well acquainted with the skills of Functions in C

1. Standard Library Functions in C

Standard Library Functions are basically the inbuilt functions in the C compiler that makes things easy for the programmer.

As we have already discussed, every C program has at least one function, that is, the **main() function**. The main() function is also a standard library function in C since it is inbuilt and conveys a specific meaning to the C compiler.

2. Significance of Standard Library Functions in C

2.1 Usability

Standard library functions allow the programmer to use the pre-existing codes available in the C compiler without the need for the user to define his own code by deriving the logic to perform certain basic functions.

2.2 Flexibility

A wide variety of programs can be made by the programmer by making slight modifications while using the standard library functions in C.

2.3 User-friendly syntax

We have already discussed in Function in C tutorial that how easy it is to grasp and use the syntax of functions.

2.4 Optimization and Reliability

All the standard library functions in C have been tested multiple times in order to generate the optimal output with maximum efficiency making it reliable to use.

2.5 Time-saving

Instead of writing numerous lines of codes, these functions help the programmer to save time by simply using the pre-existing functions.

2.6 Portability

Standard library functions are available in the C compiler irrespective of the device you are working on. These functions connote the same meaning and hence serve the same purpose regardless of the operating system or programming environment.

3. Header Files in C

In order to access the standard library functions in C, certain header files need to be included before writing the body of the program.

Don't move further, if you are not familiar with the Header Files in C.

Here is a tabular representation of a list of header files associated with some of the standard library functions in C:

HEADER FILE	MEANING	ELUCIDATION
------------------------	----------------	--------------------

<stdio.h>	Standard input-output header	Used to perform input and output operations like scanf() and printf().
<string.h>	String header	Used to perform string manipulation operations like strlen and strcpy.
<conio.h>	Console input-output header	Used to perform console input and console output operations like clrscr() to clear the screen and getch() to get the character from the keyboard.
<stdlib.h>	Standard library header	Used to perform standard utility functions like dynamic memory allocation using functions such as malloc() and calloc().
<math.h>	Math header	Used to perform mathematical operations like sqrt() and pow() to obtain the square root and the power of a number respectively.
<ctype.h>	Character type header	Used to perform character type functions like isalpha() and isdigit() to find whether the given character is an alphabet or a digit. respectively.
<time.h>	Time header	Used to perform functions related to date and time like setdate() and getdate() to modify the system date and get the CPU time respectively.
<assert.h>	Assertion header	Used in program assertion functions like assert() to get an integer data type as a parameter which prints stderr only if the parameter passed is 0.
<locale.h>	Localization header	Used to perform localization functions like setlocale() and localeconv() to set locale and get locale conventions respectively.
<signal.h>	Signal header	Used to perform signal handling functions like signal() and raise() to install signal handler and to raise the signal in the program respectively.

<setjmp.h>	Jump header	Used to perform jump functions.
<stdarg.h>	Standard argument header	Used to perform standard argument functions like <code>va_start</code> and <code>va_arg()</code> to indicate the start of the variable-length argument list and to fetch the arguments from the variable-length argument list in the program respectively.
<errno.h>	Error handling header	Used to perform error handling operations like <code>errno()</code> to indicate errors in the program by initially assigning the value of this function to 0 and then later changing it to indicate errors.

Get a complete guide to learn Data types in C

Let us discuss some of the commonly used Standard library functions in C in detail:

3.1 <stdio.h>

This is the basic header file used in almost every program written in the C language. It stands for standard input and standard output used to perform input-output functions, some of which are:

- **printf()**– Used to display output on the screen.
- **scanf()**– To take input from the user.
- **getchar()**– To return characters on the screen.
- **putchar()**– To display output as a single character on the screen.
- **fgets()**– To take a line as an input.
- **puts()**– To display a line as an output.
- **fopen()**– To open a file.
- **fclose()**– To close a file.

Here is a simple program in C that illustrates the use of <stdio.h> to use `scanf()` and `printf()` functions:

```
#include<stdio.h> // Use of stdio.h header

int main()

{

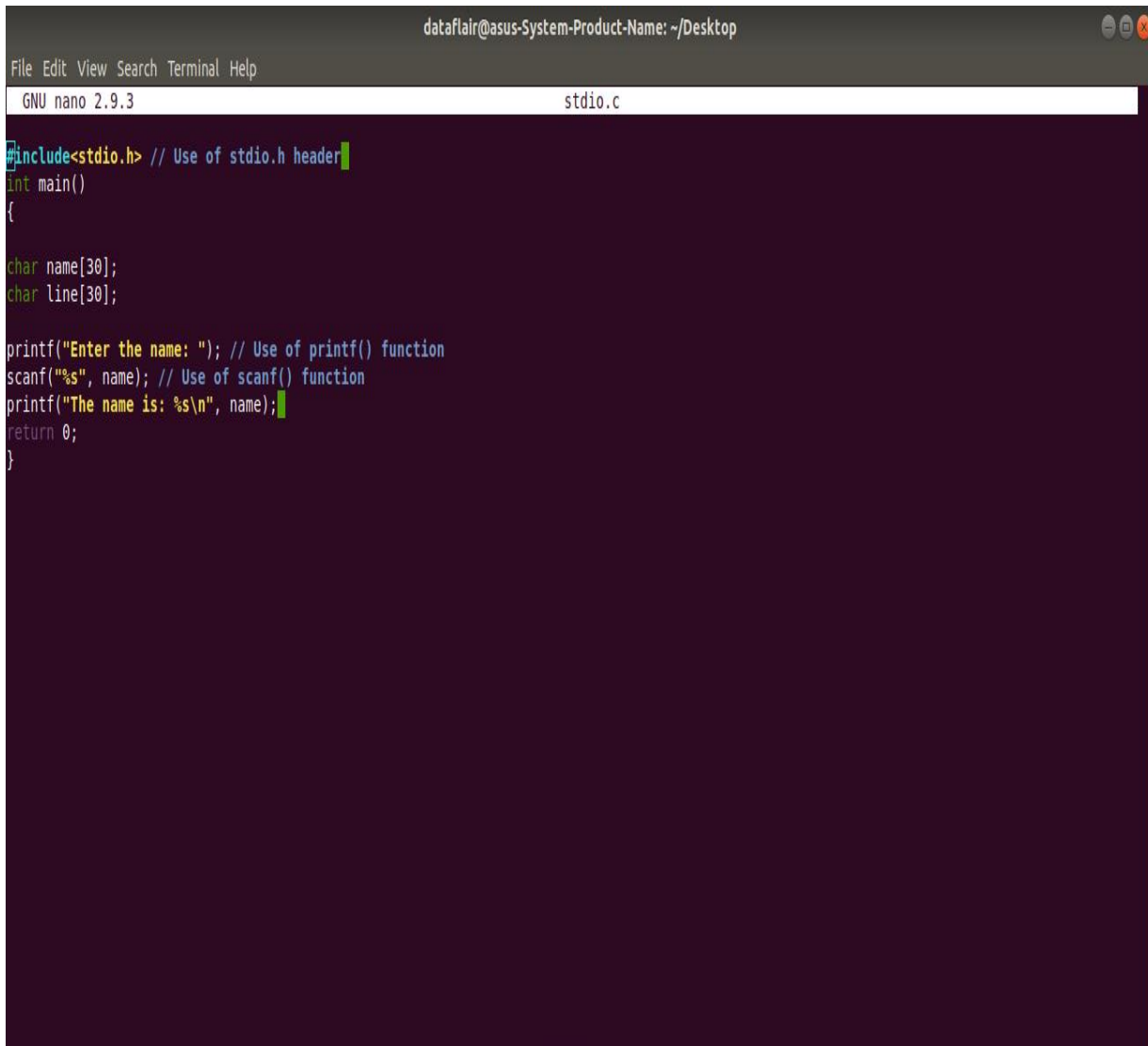
char name[30];

char line[30];

printf("Enter the name: "); // Use of printf() function
```

```
scanf("%s", name); // Use of scanf() function
printf("The name is: %s\n", name);
return 0;
}
```

Code on Screen-

A screenshot of a terminal window titled 'dataflair@asus-System-Product-Name: ~/Desktop'. The window shows the GNU nano 2.9.3 editor editing a file named 'stdio.c'. The code in the editor is as follows:

```
#include<stdio.h> // Use of stdio.h header
int main()
{
char name[30];
char line[30];

printf("Enter the name: "); // Use of printf() function
scanf("%s", name); // Use of scanf() function
printf("The name is: %s\n", name);
return 0;
}
```

Output-

```
dataflair@admin4-H110M-H: ~/Desktop
File Edit View Search Terminal Help
dataflair@admin4-H110M-H:~/Desktop$ gcc stdio.c -o stdio
dataflair@admin4-H110M-H:~/Desktop$ ./stdio
Enter the name: DataFlair
The name is: DataFlair
dataflair@admin4-H110M-H:~/Desktop$
```

3.2 <string.h>

We have already discussed the various **string manipulation functions in C** in detail.

3.3 <stdlib.h>

Functions such as malloc(), calloc(), realloc() and free() can be used while dealing with dynamic memory allocation of variables.

Let us learn these 4 basic functions before using them in our program.

It should be clear that these functions are used for dynamic memory allocation of variables, that is in contrast to arrays that allocate memory in a static (fixed) manner.

3.3.1 malloc()

malloc() stands for memory allocation. This function is responsible for reserving a specific block of memory and returns a null pointer during the execution of the program.

Syntax-

```
pointer_name = (cast_type * ) malloc (no_of_bytes * size_in_bytes_of_cast_type)
```

For instance,

```
pointer = ( float* ) malloc ( 100 * sizeof ( float ) );
```

Here is a code in C which illustrates the use of malloc() function to find the sum of numbers entered by the user-

```
#include <stdio.h>

#include <stdlib.h> // Use of stdlib header

int main()
{
printf("Welcome to DataFlair tutorials!\n\n");
int no_of_elements, iteration, *pointer, sum = 0;
printf("Enter number of elements: ");
scanf("%d", &no_of_elements);

pointer = (int*) malloc(no_of_elements * sizeof(int)); // Use of malloc() function
if(pointer == NULL)
{
printf("Sorry! Memory is not allocated.");
exit(0);
}

printf("Enter the elements: ");

/* Implementation of dynamic memory allocation */
for(iteration = 0; iteration < no_of_elements; iteration++)
```

```

{
scanf("%d", pointer + iteration);

sum += *(pointer + iteration);

}

printf("The sum of the elements is = %d\n", sum);

return 0;

}

```

Code on Screen-

```

dataflair@asus-System-Product-Name: ~/Desktop
File Edit View Search Terminal Help
GNU nano 2.9.3 malloc.c Modified

#include <stdio.h>
#include <stdlib.h> // Use of stdlib header
int main()
{

printf("Welcome to DataFlair tutorials!\n\n");
int no_of_elements, iteration, *pointer, sum = 0;

printf("Enter number of elements: ");
scanf("%d", &no_of_elements);

pointer = (int*) malloc(no_of_elements * sizeof(int)); // Use of malloc() function

if(pointer == NULL)
{
printf("Sorry! Memory is not allocated.");
exit(0);
}
printf("Enter the elements: ");

/* Implementation of dynamic memory allocation */

for(iteration = 0; iteration < no_of_elements; iteration++)
{
scanf("%d", pointer + iteration);
sum += *(pointer + iteration);
}

printf("The sum of the elements is = %d\n", sum);
return 0;
}

```

Output-

```
dataflair@admin4-H110M-H: ~/Desktop
File Edit View Search Terminal Help
dataflair@admin4-H110M-H:~/Desktop$ gcc malloc.c -o malloc
dataflair@admin4-H110M-H:~/Desktop$ ./malloc
Welcome to DataFlair tutorials!

Enter number of elements: 5
Enter the elements: 1 2 3 4 5
The sum of the elements is = 15
dataflair@admin4-H110M-H:~/Desktop$
```

Grab this Samurai Technique to Learn Arrays in C 3.3.2 calloc()

calloc stands for contiguous allocation. It is similar to malloc in all respects except the fact that it initializes the memory to 0 and has the ability to allocate numerous blocks of memory before the execution of the program.

Syntax-

```
pointer_name = (cast_type*) calloc (no_of_bytes, size_of_cast_type);
```

For instance,

```
pointer = ( int *) calloc (50, sizeof ( int ) );
```

Here is a code in C similar to malloc() that illustrates the use of calloc() function to find the sum of numbers entered by the user:

```
#include <stdio.h>

#include <stdlib.h> // Use of stdlib header

int main()
{
printf("Welcome to DataFlair tutorials!\n\n");
int no_of_elements, iteration, *pointer, sum = 0;
printf("Enter number of elements: ");
scanf("%d", &no_of_elements);
pointer = (int*) calloc(no_of_elements, sizeof(int));
if(pointer == NULL)
{
printf("Sorry! Memory is not allocated.");
exit(0);
}
printf("Enter the elements: ");
/* Implementation of dynamic mememory allocation */
for(iteration = 0; iteration < no_of_elements; iteration++)
{
scanf("%d", pointer + iteration);
sum += *(pointer + iteration);
}
printf("The sum of the elements is = %d\n", sum);
return 0;
}
```

Code on Screen-

```
dataflair@asus-System-Product-Name: ~/Desktop
File Edit View Search Terminal Help
GNU nano 2.9.3          calloc.c          Modified

#include <stdio.h>
#include <stdlib.h> // Use of stdlib header
int main()
{

printf("Welcome to DataFlair tutorials!\n\n");
int no_of_elements, iteration, *pointer, sum = 0;

printf("Enter number of elements: ");
scanf("%d", &no_of_elements);

pointer = (int*) calloc(no_of_elements, sizeof(int));

if(pointer == NULL)
{
printf("Sorry! Memory is not allocated.");
exit(0);
}
printf("Enter the elements: ");

/* Implementation of dynamic memory allocation */

for(iteration = 0; iteration < no_of_elements; iteration++)
{
scanf("%d", pointer + iteration);
sum += *(pointer + iteration);
}

printf("The sum of the elements is = %d\n", sum);
return 0;
}
```

Output-

```
dataflair@admin4-H110M-H: ~/Desktop
File Edit View Search Terminal Help
dataflair@admin4-H110M-H:~$ cd Desktop
dataflair@admin4-H110M-H:~/Desktop$ touch calloc.c
dataflair@admin4-H110M-H:~/Desktop$ gcc calloc.c -o calloc
dataflair@admin4-H110M-H:~/Desktop$ ./calloc
Welcome to DataFlair tutorials!

Enter number of elements: 10
Enter the elements: 1
2
3
4
5
6
7
8
9
10
The sum of the elements is = 55
dataflair@admin4-H110M-H:~/Desktop$
```

3.3.3 realloc()

realloc stands for reallocation. It is used to change the size of the previously allocated memory in case the previously allocated memory is insufficient to meet the required needs of the **variable in C**.

Syntax-

```
pointer_name = realloc(pointer_name, new_size);
```

For instance,

If initially,

```
pointer = ( int* ) malloc( 20 * sizeof( int ) );
```

Then, using realloc

```
pointer = realloc(pointer, 24 * sizeof( int ) );
```

Here is a code in C that illustrates the use of realloc() function:

```
#include <stdio.h>

#include <stdlib.h> // Use of stdlib header

int main()
{
printf("Welcome to DataFlair tutorials!\n\n");

int *pointer,*new_pointer, iteration;

pointer = (int *)malloc(2*sizeof(int));

*pointer = 5;

*(pointer + 1) = 10;

new_pointer = (int *)realloc(pointer, 3*sizeof(int));

*(new_pointer + 2) = 15;

printf("The elements are: ");

for(iteration = 0; iteration < 3; iteration++)

printf("%d\n ", *(new_pointer + iteration));

return 0;

}
```

Code on Screen-

```
dataflair@asus-System-Product-Name: ~/Desktop
File Edit View Search Terminal Help
GNU nano 2.9.3 realloc.c Modified

#include <stdio.h>
#include <stdlib.h> // Use of stdlib header
int main()
{
printf("Welcome to DataFlair tutorials!\n\n");

int *pointer,*new_pointer, iteration;

pointer = (int *)malloc(2*sizeof(int));

*pointer = 5;
*(pointer + 1) = 10;

new_pointer = (int *)realloc(pointer, 3*sizeof(int));
*(new_pointer + 2) = 15;
printf("The elements are: ");
for(iteration = 0; iteration < 3; iteration++)
printf("%d\n ", *(new_pointer + iteration));
return 0;
}
```

Output-


```
dataflair@admin4-H110M-H: ~/Desktop
File Edit View Search Terminal Help
dataflair@admin4-H110M-H:~/Desktop$ gcc realloc.c -o realloc
dataflair@admin4-H110M-H:~/Desktop$ ./realloc
Welcome to DataFlair tutorials!

The elements are: 5
10
15
dataflair@admin4-H110M-H:~/Desktop$
```

3.3.4 free()

free is responsible to free the dynamically allocated memory done by malloc(), calloc() or realloc() to the system.

Syntax-

```
free( pointer_name );
```

For instance,

```
free( pointer );
```

Here is a code in C similar to malloc() that illustrates the use of free() function to find the sum of numbers entered by the user:

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

```
#include <stdlib.h> // Use of stdlib header
```

```
int main()
```

```
{
```

```
printf("Welcome to DataFlair tutorials!\n\n");
```

```
int no_of_elements, iteration, *pointer, sum = 0;
```

```
printf("Enter number of elements: ");
```

```
scanf("%d", &no_of_elements);
```

```
pointer = (int*) malloc(no_of_elements * sizeof(int)); // Use of malloc() function
```

```
if(pointer == NULL)
```

```
{
```

```
printf("Sorry! Memory is not allocated.");
```

```
exit(0);
```

```
}
```

```
printf("Enter the elements: ");
```

```
/* Implementation of dynamic memmory allocation */
```

```
for(iteration = 0; iteration < no_of_elements; iteration++)
```

```
{
```

```
scanf("%d", pointer + iteration);
```

```
sum += *(pointer + iteration);
```

```

}

printf("The sum of the elements is = %d\n", sum);

free(pointer); // Use of free() function

return 0;

}

```

Code on Screen-

```

dataflair@asus-System-Product-Name: ~/Desktop
File Edit View Search Terminal Help
GNU nano 2.9.3 free.c Modified

#include <stdio.h>
#include <stdlib.h> // Use of stdlib header
int main()
{

printf("Welcome to DataFlair tutorials!\n\n");
int no_of_elements, iteration, *pointer, sum = 0;

printf("Enter number of elements: ");
scanf("%d", &no_of_elements);

pointer = (int*) malloc(no_of_elements * sizeof(int)); // Use of malloc() function

if(pointer == NULL)
{
printf("Sorry! Memory is not allocated.");
exit(0);
}
printf("Enter the elements: ");

/* Implementation of dynamic memory allocation */

for(iteration = 0; iteration < no_of_elements; iteration++)
{
scanf("%d", pointer + iteration);
sum += *(pointer + iteration);
}

printf("The sum of the elements is = %d\n", sum);
free(pointer); // Use of free() function
return 0;
}

```

Output-

```
dataflair@admin4-H110M-H: ~/Desktop
File Edit View Search Terminal Help
dataflair@admin4-H110M-H:~$ cd Desktop
dataflair@admin4-H110M-H:~/Desktop$ touch free.c
dataflair@admin4-H110M-H:~/Desktop$ gcc free.c -o free
dataflair@admin4-H110M-H:~/Desktop$ ./free
Welcome to DataFlair tutorials!

Enter number of elements: 4
Enter the elements: 1 2 3 4
The sum of the elements is = 10
dataflair@admin4-H110M-H:~/Desktop$
```

3.4 <math.h>

The math header is of great significance as it is used to perform various mathematical operations such as:

- **sqrt()** – This function is used to find the square root of a number

- **pow()** – This function is used to find the power raised to that number.
- **fabs()** – This function is used to find the absolute value of a number.
- **log()** – This function is used to find the logarithm of a number.
- **sin()** – This function is used to find the sine value of a number.
- **cos()** – This function is used to find the cosine value of a number.
- **tan()** – This function is used to find the tangent value of a number.

Here is a code in C that illustrates the use of some of the basic <math.h> functions:

```
#include <stdio.h>

#include <math.h>

int main()
{
printf("Welcome to DataFlair tutorials!\n\n");

double number=5, square_root;

int base = 6, power = 3, power_result;

int integer = -7, integer_result;

square_root = sqrt(number);

printf("The square root of %lf is: %lf\n", number, square_root);

power_result = pow(base,power);

printf("%d raised to the power %d is: %d\n", base, power, power_result);

integer_result = fabs(integer);

printf("The absolute value of %d is: %d\n", integer, integer_result);

return 0;

}
```

Code on Screen-

```
dataflair@asus-System-Product-Name: ~/Desktop
File Edit View Search Terminal Help
GNU nano 2.9.3 math.c Modified

#include <stdio.h>
#include <math.h>
int main()
{
printf("Welcome to DataFlair tutorials!\n\n");

double number=5, square_root;
int base = 6, power = 3, power_result;
int integer = -7, integer_result;

square_root = sqrt(number);
printf("The square root of %lf is: %lf\n", number, square_root);

power_result = pow(base,power);
printf("%d raised to the power %d is: %d\n", base, power, power_result);

integer_result = fabs(integer);
printf("The absolute value of %d is: %d\n", integer, integer_result);
return 0;
}
```

Output-

```
dataflair@admin4-H110M-H: ~/Desktop
File Edit View Search Terminal Help
dataflair@admin4-H110M-H:~/Desktop$ gcc math.c -o math -lm
dataflair@admin4-H110M-H:~/Desktop$ ./math
Welcome to DataFlair tutorials!

The square root of 5.000000 is: 2.236068
6 raised to the power 3 is: 216
The absolute value of -7 is: 7
dataflair@admin4-H110M-H:~/Desktop$
```

3.5 <ctype.h>

This function is popularly used when it comes to character handling.

Some of the functions associated with <ctype.h> are:

- **isalpha()** – Used to check if the character is an alphabet or not.

- **isdigit()** – Used to check if the character is a digit or not.
- **isalnum()** – Used to check if the character is alphanumeric or not.
- **isupper()** – Used to check if the character is in uppercase or not
- **islower()** – Used to check if the character is in lowercase or not.
- **toupper()** – Used to convert the character into uppercase.
- **tolower()** – Used to convert the character into lowercase.
- **isctrl()** – Used to check if the character is a control character or not.
- **isgraph()** – Used to check if the character is a graphic character or not.
- **isprint()** – Used to check if the character is a printable character or not
- **ispunct()** – Used to check if the character is a punctuation mark or not.
- **isspace()** – Used to check if the character is a white-space character or not.
- **isxdigit()** – Used to check if the character is hexadecimal or not.
 - <ctype.h> is not supported in Linux but it works fairly well in Microsoft Windows.
- 3.6 <conio.h>
 - It is used to perform console input and console output operations like **clrscr()** to clear the screen and **getch()** to get the character from the keyboard.
 - **Note:** The header file <conio.h> is not supported in Linux. It works fairly well on Microsoft Windows.
- 4. Summary

In this tutorial, we discussed the meaning of Standard library functions in C. We covered each and every aspect associated with it such that the reader understands it in the best way. Then, we discussed the significance of standard library functions in detail. Thereafter, we discussed the various header files needed to access the standard library functions with the help of illustrative C programs. It is safe to proclaim that after completing this tutorial, we have successfully conquered the core concepts of C programming by getting a firm grip on Standard library functions.

strlen()

The C library function **size_t strlen(const char *str)** computes the length of the string **str** up to, but not including the terminating null character.

Declaration

Following is the declaration for strlen() function.

```
size_t strlen(const char *str)
```


Parameters

- **str** – This is the string whose length is to be found.

Return Value

This function returns the length of string.

Example

The following example shows the usage of `strlen()` function.

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

int main () {
    char str[50];
    int len;

    strcpy(str, "This is tutorialspoint.com");

    len = strlen(str);
    printf("Length of |%s| is |%d|\n", str, len);

    return(0);
}
```

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

Length of |This is tutorialspoint.com| is |26|

strcpy()

The C library function **`char *strcpy(char *dest, const char *src)`** copies the string pointed to, by **`src`** to **`dest`**.

Declaration

Following is the declaration for `strcpy()` function.

```
char *strcpy(char *dest, const char *src)
```

Parameters

- **dest** – This is the pointer to the destination array where the content is to be copied.
- **src** – This is the string to be copied.

Return Value

This returns a pointer to the destination string **dest**.

Example

The following example shows the usage of `strcpy()` function.

Live Demo

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

int main () {
    char src[40];
    char dest[100];

    memset(dest, '\0', sizeof(dest));
    strcpy(src, "This is tutorialspoint.com");
    strcpy(dest, src);

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

    return(0);
}
```

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

Final copied string : This is tutorialspoint.com

strcat()

The C library function **char *strcat(char *dest, const char *src)** appends the string pointed to by **src** to the end of the string pointed to by **dest**.

Declaration

Following is the declaration for strcat() function.

```
char *strcat(char *dest, const char *src)
```

Parameters

- **dest** – This is pointer to the destination array, which should contain a C string, and should be large enough to contain the concatenated resulting string.
- **src** – This is the string to be appended. This should not overlap the destination.

Return Value

This function returns a pointer to the resulting string dest.

Example

The following example shows the usage of strcat() function.

Live Demo

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

int main () {
    char src[50], dest[50];

    strcpy(src, "This is source");
    strcpy(dest, "This is destination");

    strcat(dest, src);

    printf("Final destination string : |%s|", dest);

    return(0);
}
```

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

Final destination string : |This is destinationThis is source|

strcmp()

The C library function **int strcmp(const char *str1, const char *str2)** compares the string pointed to, by **str1** to the string pointed to by **str2**.

Declaration

Following is the declaration for strcmp() function.

```
int strcmp(const char *str1, const char *str2)
```

Parameters

- **str1** – This is the first string to be compared.
- **str2** – This is the second string to be compared.

Return Value

This function return 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

The following example shows the usage of strcmp() function.

Live Demo

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

int main () {
    char str1[15];
    char str2[15];
    int ret;

    strcpy(str1, "abcdef");
    strcpy(str2, "ABCDEF");

    ret = strcmp(str1, str2);
```

```

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);
}

```

Let us compile and run the above program that will produce the following result –
str2 is less than str1

Implementation without using standard library functions

In the following C program we are counting the number of characters in a given String to find out and display its length on console. Upon execution of this program, the user would be asked to enter a string, then the program would count the chars and would display the length of input string as output.

C Program – finding length of a String without using standard library function strlen

```

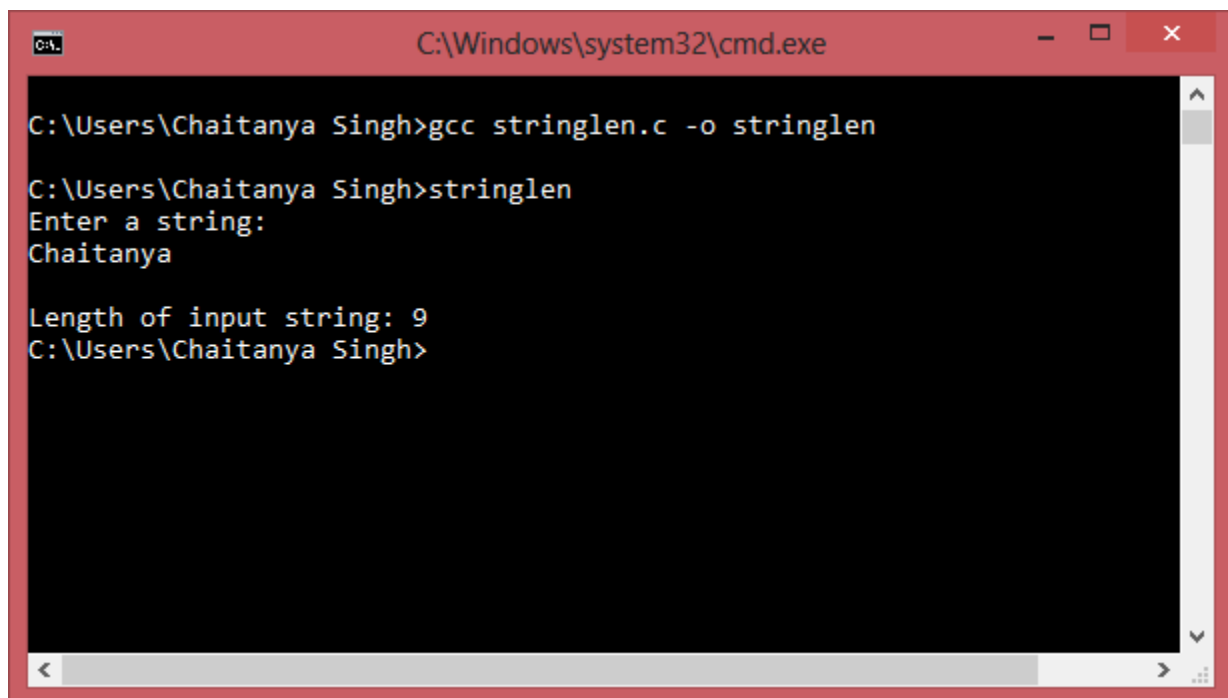
/* C Program to find the length of a String without
 * using any standard library function
 */
#include <stdio.h>
int main()
{
    /* Here we are taking a char array of size
     * 100 which means this array can hold a string
     * of 100 chars. You can change this as per requirement
     */
    char str[100],i;
    printf("Enter a string: \n");
    scanf("%s",str);

    // '\0' represents end of String

```

```
for(i=0; str[i]!='\0'; ++i);  
    printf("\nLength of input string: %d",i);  
  
return 0;  
}
```

Output:



```
C:\Windows\system32\cmd.exe  
  
C:\Users\Chaitanya Singh>gcc stringlen.c -o stringlen  
  
C:\Users\Chaitanya Singh>stringlen  
Enter a string:  
Chaitanya  
  
Length of input string: 9  
C:\Users\Chaitanya Singh>
```

Structures

Definition and declaration

This article will cover a brief on user defined data types in C. C supports various data types, out of which there are few where programmer gets some benefit. User defined data types are used to create a variable which can contain few many other data types inside them. One variable containing many other variables, that's what we mean by user defined data types.

A user will have complete privilege to keep certain combinations of data types as per the need and call it a new variable itself.

What is structure in C?

We can start with structures in this article and following articles will continue on the other types. Structure uses a keyword `struct`. A structure can have a declaration like below

Declaration/Definition of a structure

```
struct tagname{
    char    x;
    int     y;
    float   z;
};
```

Above is the definition for the structure, it says the compiler that what variables it will contain. Now we need to declare an object of that structure. Also in the above we can think of the `tagname` as data types names `int` , `char`, etc.

Declaration of a structure variable

Now use the `tagname` for getting an object, below is the syntax

```
struct tagname structvariable;
```

When we write the above piece of code, compiler would allocate contiguous memory which can accommodate everything this struct has.

Accessing structure members through structure variable

Since we have created many variables inside the structure we may want to access them and do some operations, below are the few examples:

```
structvariable.x='A';
/* this will write 'A' for the element x of structure structvariable*/
```

Similarly we can also access all other variables

```
structvariable.y=20;
structvariable.z=10.20f;
```

Structure pointer - Declaration, Accessing of Structure members

The objects to a structure can also be a pointer same like we can create a pointer of int. To achieve this which we will need following declaration:

```
struct tagname *structPtrVariable;
```

To access the elements inside the structure we should be using the following syntax

```
structPtrVariable->x = 'A' // here '.' is replace by '->'
structPtrVariable->y = 20;
structPtrVariable->z = 10.20f;
```

Above we have created two objects for the `struct tagname`. Those two objects have independent memory allocated for each of them. Both of them can be compared with the normal declaration of the variables for example:

```
int a;
int *a;
```

Structures play very important role in big systems where we need to combine several data's into a set and need to capture multiples of that set, perform operations and many more.

It even helps on the smaller systems.

A small piece of code will help understand the use of structures better.

```
#include <stdio.h>

struct tagname {
    char Char;
    int Int;
    float Dec;
};

int main()
{
    struct tagname StructObj;
```



```
struct tagname *ptrStructObj=&StructObj;

StructObj.Char='H';
ptrStructObj->Int=927;
ptrStructObj->Dec=911.0f;

printf("%C\n",StructObj.Char);
printf("%d\n",ptrStructObj->Int);
printf("%f",ptrStructObj->Dec);
printf("\n");

return 0;
}
```

Output

```
H
927
911.000000
```

Variables initialization

C variables are names used for storing a data value to locations in memory. The value stored in the c variables may be changed during program execution.

Declaration of Variable

Declaration of variable in c can be done using following syntax:

```
data_type variable_name;
or
data_type variable1, variable2,...,variablen;
```

where data_type is any valid c data type and variable_name is any valid identifier.

For example,

- 1 int a;
- 2 float variable;
- 3 float a, b;

Initialization of Variable

C variables declared can be initialized with the help of assignment operator '='.

Syntax

```
data_type variable_name=constant/literal/expression;  
or  
variable_name=constant/literal/expression;
```

Example

- 1 int a=10;
- 2 int a=b+c;
- 3 a=10;
- 4 a=b+c;

Multiple variables can be initialized in a single statement by single value, for example, a=b=c=d=e=10;

NOTE: C variables must be declared before they are used in the c program. Also, since c is a case sensitive programming language, therefore the c variables, abc, Abc and ABC are all different.

Constant and Volatile Variables

Constant Variables

C variables having same or unchanged value during the execution of a program are called constant variables. A variable can be declared as constant using keyword **const**. For example,

- 1 const int a=100;

Now, if we try to change its value, then it is invalid.

Volatile Variables

Those variables that can be changed at any time by some external sources from outside or same program are called volatile variables.

Any variable in c can be declared as volatile using keyword **volatile**.

Syntax

```
volatile data_type variable_name;
```

NOTE: If the value of a variable in the current program is to be maintained constant and desired not to be changed by any other external operation, then the variable declaration will be `volatile const d=10;`

Accessing fields and structure operations

Structure is a group of variables of different data types represented by a single name. Lets take an example to understand the need of a structure in C programming.

Lets say we need to store the data of students like student name, age, address, id etc. One way of doing this would be creating a different variable for each attribute, however when you need to store the data of multiple students then in that case, you would need to create these several variables again for each student. This is such a big headache to store data in this way.

We can solve this problem easily by using structure. We can create a structure that has members for name, id, address and age and then we can create the variables of this structure for each student. This may sound confusing, do not worry we will understand this with the help of example.

How to create a structure in C Programming

We use **struct** keyword to create a **structure in C**. The struct keyword is a short form of **structured data type**.

```
struct struct_name {  
    DataType member1_name;  
    DataType member2_name;  
    DataType member3_name;  
    ...  
}
```

```
};
```

Here struct_name can be anything of your choice. Members data type can be same or different. Once we have declared the structure we can use the struct name as a data type like int, float etc.

First we will see the syntax of creating struct variable, accessing struct members etc and then we will see a complete example.

How to declare variable of a structure?

```
struct struct_name var_name;
```

or

```
struct struct_name {  
    DataType member1_name;  
    DataType member2_name;  
    DataType member3_name;  
    ...  
} var_name;
```

How to access data members of a structure using a struct variable?

```
var_name.member1_name;  
var_name.member2_name;  
...
```

How to assign values to structure members?

There are three ways to do this.

1) Using Dot(.) operator

```
var_name.memeber_name = value;
```

2) All members assigned in one statement

```
struct struct_name var_name =  
{value for memeber1, value for memeber2 ...so on for all the members}
```

3) **Designated initializers** – We will discuss this later at the end of this post.

Example of Structure in C

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

```

/* Created a structure here. The name of the structure is
 * StudentData.
 */
struct StudentData{
    char *stu_name;
    int stu_id;
    int stu_age;
};
int main()
{
    /* student is the variable of structure StudentData*/
    struct StudentData student;

    /*Assigning the values of each struct member here*/
    student.stu_name = "Steve";
    student.stu_id = 1234;
    student.stu_age = 30;

    /* Displaying the values of struct members */
    printf("Student Name is: %s", student.stu_name);
    printf("\nStudent Id is: %d", student.stu_id);
    printf("\nStudent Age is: %d", student.stu_age);
    return 0;
}

```

Output:

```

Student Name is: Steve
Student Id is: 1234
Student Age is: 30

```

Nested Structure in C: Struct inside another struct

You can use a structure inside another structure, which is fairly possible. As I explained above that once you declared a structure, the **struct struct_name** acts as a new data type so you can include it in another struct just like the data type of other data members. Sounds confusing? Don't worry. The following example will clear your doubt.

Example of Nested Structure in C Programming

Lets say we have two structure like this:

Structure 1: stu_address

```
struct stu_address
{
    int street;
    char *state;
    char *city;
    char *country;
}
```

Structure 2: stu_data

```
struct stu_data
{
    int stu_id;
    int stu_age;
    char *stu_name;
    struct stu_address stuAddress;
}
```

As you can see here that I have nested a structure inside another structure.

Assignment for struct inside struct (Nested struct)

Lets take the example of the two structure that we seen above to understand the logic

```
struct stu_data mydata;
mydata.stu_id = 1001;
mydata.stu_age = 30;
mydata.stuAddress.state = "UP"; //Nested struct assignment
..
```

How to access nested structure members?

Using chain of “.” operator.

Suppose you want to display the city alone from nested struct –

```
printf("%s", mydata.stuAddress.city);
```

Use of typedef in Structure

typedef makes the code short and improves readability. In the above discussion we have seen that while using structs every time we have to use the lengthy syntax, which makes the code confusing, lengthy, complex and less readable. The simple solution to this issue is use of typedef. It is like an alias of struct.

Code without typedef

```
struct home_address {  
    int local_street;  
    char *town;  
    char *my_city;  
    char *my_country;  
};  
...  
struct home_address var;  
var.town = "Agra";
```

Code using typedef

```
typedef struct home_address{  
    int local_street;  
    char *town;  
    char *my_city;  
    char *my_country;  
}addr;  
..  
..  
addr var1;  
var.town = "Agra";
```

Instead of using the struct home_address every time you need to declare struct variable, you can simply use addr, the typedef that we have defined.

Designated initializers to set values of Structure members

We have already learned two ways to set the values of a struct member, there is another way to do the same using designated initializers. This is useful when we are doing assignment of only few members of the structure. In the following example the structure variable s2 has only one member assignment.

```
#include <stdio.h>
struct numbers
{
    int num1, num2;
};
int main()
{
    // Assignment using using designated initialization
    struct numbers s1 = {.num2 = 22, .num1 = 11};
    struct numbers s2 = {.num2 = 30};

    printf ("num1: %d, num2: %d\n", s1.num1, s1.num2);
    printf ("num1: %d", s2.num2);
    return 0;
}
```

Output:

```
num1: 11, num2: 22
num1: 30
```

Nested structures

C provides us the feature of nesting one structure within another structure by using which, complex data types are created. For example, we may need to store the address of an entity employee in a structure. The attribute address may also have the subparts as street number, city, state, and pin code. Hence, to store the address of the employee, we need to store the address of the employee into a separate structure and nest the structure address into the structure employee. Consider the following program.

1. #include<stdio.h>
2. **struct** address


```
3. {
4.   char city[20];
5.   int pin;
6.   char phone[14];
7. };
8. struct employee
9. {
10.  char name[20];
11.  struct address add;
12.};
13.void main ()
14.{
15.  struct employee emp;
16.  printf("Enter employee information?\n");
17.  scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
18.  printf("Printing the employee information....\n");
19.  printf("name: %s\nCity: %s\nPincode: %d\nPhone: %s",emp.name,emp.add.city,emp.
    add.pin,emp.add.phone);
20.}
```

Output

```
Enter employee information?
```

Arun

Delhi

110001

1234567890

Printing the employee information....

name: Arun

City: Delhi

Pincode: 110001

Phone: 1234567890

The structure can be nested in the following ways.

1. By separate structure
2. By Embedded structure

1) Separate structure

Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

1. **struct** Date
2. {
3. **int** dd;
4. **int** mm;
5. **int** yyyy;
6. };

```
7. struct Employee
8. {
9.   int id;
10.  char name[20];
11.  struct Date doj;
12.}emp1;
```

As you can see, doj (date of joining) is the variable of type Date. Here doj is used as a member in Employee structure. In this way, we can use Date structure in many structures.

2) Embedded structure

The embedded structure enables us to declare the structure inside the structure. Hence, it requires less line of codes but it can not be used in multiple data structures. Consider the following example.

```
1. struct Employee
2. {
3.   int id;
4.   char name[20];
5.   struct Date
6.   {
7.     int dd;
8.     int mm;
9.     int yyyy;
```

10. }doj;

11.}emp1;

Accessing Nested Structure

We can access the member of the nested structure by Outer_Structure.Nested_Structure.member as given below:

1. e1.doj.dd
2. e1.doj.mm
3. e1.doj.yyyy

C Nested Structure example

Let's see a simple example of the nested structure in C language.

1. #include <stdio.h>
2. #include <string.h>
3. **struct** Employee
4. {
5. **int** id;
6. **char** name[20];
7. **struct** Date
8. {
9. **int** dd;
10. **int** mm;
11. **int** yyyy;

```
12. }doj;
13.}e1;
14. int main( )
15.{
16. //storing employee information
17. e1.id=101;
18. strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
19. e1.doj.dd=10;
20. e1.doj.mm=11;
21. e1.doj.yyyy=2014;
22.
23. //printing first employee information
24. printf( "employee id : %d\n", e1.id);
25. printf( "employee name : %s\n", e1.name);
26. printf( "employee date of joining (dd/mm/yyyy) : %d/%d/%d\n", e1.doj.dd,e1.doj.mm,e
1.doj.yyyy);
27. return 0;
28.}
```

Output:

```
employee id : 101
employee name : Sonoo Jaiswal
employee date of joining (dd/mm/yyyy) : 10/11/2014
```

Passing structure to function

Just like other variables, a structure can also be passed to a function. We may pass the structure members into the function or pass the structure variable at once. Consider the following example to pass the structure variable employee to a function display() which is used to display the details of an employee.

```
1. #include<stdio.h>
2. struct address
3. {
4.     char city[20];
5.     int pin;
6.     char phone[14];
7. };
8. struct employee
9. {
10.    char name[20];
11.    struct address add;
12.};
13.void display(struct employee);
14.void main ()
15.{
16.    struct employee emp;
17.    printf("Enter employee information?\n")
    ;
```

```
18. scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
19. display(emp);
20.}
21. void display(struct employee emp)
22.{
23. printf("Printing the details...\n");
24. printf("%s %s %d %s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);
25.}
```

Union:

Definition and declaration

Unions can be assumed same as structures but with little difference. **Unions** can be used to create a data type containing several others data types inside it and we can use an object of that union to access the members inside it.

Below is the declaration for a Union in C:

Union declaration

```
union tagname
{
    int a;
    char b;
};
```

Here, `union` is the keyword to declare a union, `tagname` is the union name, `a` and `b` are the members of the union `tagname`.

Union variable/object declaration

Now we should create an object for the union in order to access the elements inside it. Below is how we can do that:

```
union tagname object;
```

Here, object is the union variable name, that will be used to access the union elements.

Accessing Union elements

Union elements can be accessed using dot (.) operator, use `union_variable_name.element_name` to access particular element.

```
object.a= 10;  
object.b= 'H';
```

Till now the union must have looked same as the structures in C. But there is a great difference between structure and the unions. When we create a structure, the memory allocated for it is based on the elements inside the structure. So if a structure has two elements one int and one char than the size of that structure would be at least 5 bytes (if int takes 4 bytes and 1 byte is for char).

In case of the union the size of memory allocated is equal to the size of the element which takes largest size.

So for the union above the size would be only 4 bytes not 5 bytes.

Take the below example:

```
union tagname  
{  
    int a;  
    char s;  
    char t;  
};
```

In this case if we create the object

```
union tagname object;
```


Size of this object should be 4 bytes only. This feature of unions gives some benefits but care should be taken while operating with unions. Since the memory allocated is equal to the largest element in the union, values will be overwritten.

Take the below example for better understanding:

```
#include<stdio.h>
union tagname
{
    int a;
    char s;
    char t;
};

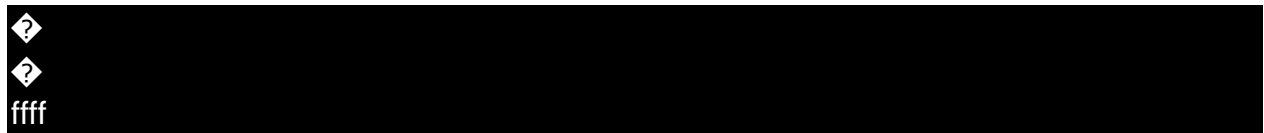
union tagname object;

int main()
{
    object.s='H';
    object.t='E';
    object.a=0xFFFF;

    printf("%c\n",object.s);
    printf("%c\n",object.t);
    printf("%x\n",object.a);

    return 0;
}
```

Output



```
?
?
ffff
```

So if the last written element is higher than the size of the previously written elements in the union, than one might not be able to retrieve the values of the previously written values.

Below is the benefit of the union:

```

#include<stdio.h>

union tagname
{
    int a;
    unsigned char s[4];
};

union tagname object;

int main()
{
    object.a=0xAABBCCDD;

    printf("%d\n",sizeof(object));
    printf("%X\n",object.a);

    char i;
    for(i=3;i>=0;i--)
        printf("%X\n",object.s[i]);
    return 0;
}

```

Output

```

4
AABBCCDD
AA
BB
CC
DD

```

Differentiate between Union and structure

In C we have container for both i.e. for same type data and multiple type data. For storage of data of same type C provides concept of Array which stores data variables of same type while for storing data of different type C has concept of structure and union that can store data variable of different type as well.

Since both Structure and Union can hold different type of data in them but now on the basis of internal implementation we can find several differences in both of these containers.

Following are the important differences between Structure and Union.

Sr. No.	Key	Structure	Union
1	Definition	Structure is the container defined in C to store data variables of different type and also supports for the user defined variables storage.	On other hand Union is also similar kind of container in C which can also holds the different type of variables along with the user defined variables.
2	Internal implementation	Structure in C is internally implemented as that there is separate memory location is allotted to each input member	While in case Union memory is allocated only to one member having largest size among all other input variables and the same location is being get shared among all of these.
3	Syntax	Syntax of declare a Structure in C is as follow :	On other syntax of declare a Union in C is as follow:
		<pre>struct struct_name{ type element1; type element2; . . } variable1, variable2, ...;</pre>	<pre>union u_name{ type element1; type element2; . . } variable1, variable2, ...;</pre>

Sr. No.	Key	Structure	Union
4	Size	As mentioned in definition Structure do not have shared location for its members so size of Structure is equal or greater than the sum of size of all the data members.	On other hand Union does not have separate location for each of its member so its size or equal to the size of largest member among all data members.
5	Value storage	As mentioned above in case of Structure there is specific memory location for each input data member and hence it can store multiple values of the different members.	While in case of Union there is only one shared memory allocation for all input data members so it stores a single value at a time for all members.
6	Initialization	In Structure multiple members can be can be initializing at same time.	On other hand in case of Union only the first member can get initialize at a time.

UNIT-IV

Introduction C Preprocessor

Definition of Preprocessor

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives –

Sr.No.	Directive & Description
1	#define Substitutes a preprocessor macro.
2	#include Inserts a particular header from another file.
3	#undef Undefines a preprocessor macro.
4	#ifdef Returns true if this macro is defined.
5	#ifndef Returns true if this macro is not defined.

6	#if Tests if a compile time condition is true.
7	#else The alternative for #if.
8	#elif #else and #if in one statement.
9	#endif Ends preprocessor conditional.
10	#error Prints error message on stderr.
11	#pragma Issues special commands to the compiler, using a standardized method.

Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use #define for constants to increase readability.

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

These directives tell the CPP to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

```
#undef FILE_SIZE
```

```
#define FILE_SIZE 42
```

It tells the CPP to undefine existing FILE_SIZE and define it as 42.

```
#ifndef MESSAGE  
#define MESSAGE "You wish!"  
#endif
```

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG  
/* Your debugging statements here */  
#endif
```

It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the -DDEBUG flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

Predefined Macros

ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

Sr.No.	Macro & Description
1	__DATE__ The current date as a character literal in "MMM DD YYYY" format.
2	__TIME__ The current time as a character literal in "HH:MM:SS" format.
3	__FILE__ This contains the current filename as a string literal.
4	__LINE__

	This contains the current line number as a decimal constant.
5	__STDC__ Defined as 1 when the compiler complies with the ANSI standard.

Let's try the following example –

[Live Demo](#)

```
#include <stdio.h>

int main() {

    printf("File :%s\n", __FILE__ );
    printf("Date :%s\n", __DATE__ );
    printf("Time :%s\n", __TIME__ );
    printf("Line :%d\n", __LINE__ );
    printf("ANSI :%d\n", __STDC__ );

}
```

When the above code in a file **test.c** is compiled and executed, it produces the following result –

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

Preprocessor Operators

The C preprocessor offers the following operators to help create macros –

The Macro Continuation (\) Operator

A macro is normally confined to a single line. The macro continuation operator (\) is used to continue a macro that is too long for a single line. For example –

```
#define message_for(a, b) \
    printf("#a " and " #b ": We love you!\n")
```


The Stringize (#) Operator

The stringize or number-sign operator ('# '), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list. For example –

[Live Demo](#)

```
#include <stdio.h>

#define message_for(a, b) \
    printf("#a " and " #b ": We love you!\n")

int main(void) {
    message_for(Carole, Debra);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Carole and Debra: We love you!

The Token Pasting (##) Operator

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example –

[Live Demo](#)

```
#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d", token##n)

int main(void) {
    int token34 = 40;
    tokenpaster(34);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

token34 = 40

It happened so because this example results in the following actual output from the preprocessor –

```
printf ("token34 = %d", token34);
```

This example shows the concatenation of token##n into token34 and here we have used both **stringize** and **token-pasting**.

The Defined() Operator

The preprocessor **defined** operator is used in constant expressions to determine if an identifier is defined using **#define**. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero). The defined operator is specified as follows –

Live Demo

```
#include <stdio.h>

#if !defined (MESSAGE)
    #define MESSAGE "You wish!"
#endif

int main(void) {
    printf("Here is the message: %s\n", MESSAGE);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Here is the message: You wish!

Parameterized Macros

One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number as follows –

```
int square(int x) {
    return x * x;
}
```

We can rewrite above the code using a macro as follows –

```
#define square(x) ((x) * (x))
```

Macros with arguments must be defined using the **#define** directive before they can be used. The argument list is enclosed in parentheses and must immediately follow the

macro name. Spaces are not allowed between the macro name and open parenthesis. For example –

[Live Demo](#)

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void) {
    printf("Max between 20 and 10 is %d\n", MAX(10, 20));
    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Max between 20 and 10 is 20

Macro substitution directives

C provides a language feature called preprocessor. It is a special step in compiler chain. C statements start with # symbol are processed by the preprocessor before compilation. The result of preprocessor is expanded source, which will be input to the compiler.

In previous session, we have discoursed how Expanded source can be generated in different platforms

Types of preprocessor directives:

C language supports different preprocessor statements like

- Macro substitution
- File inclusions
- Conditional inclusion/Compilation
- operators

Rules in writing preprocessor directives

We must follow certain rules while writing preprocessor statement, Some of these rules are

- All preprocessor directives must be started with # symbol
- Every preprocessor statement may be started from the first column (Optional)
- There shouldn't be any space between # and directive
- A preprocessor statement must not be terminated with a semicolon
- Multiple preprocessor statements must not be written in a single line
- Preprocessor statements can be written any where within the block, function or outside any function

Macro substitution

Macro substitution has a name and replacement text, defined with #define directive. The preprocessor simply replaces the name of macro with replacement text from the place where the macro is defined in the source code.

Syntax:

```
#define name replacement text
```

Now we will see the fact through an example.

```
/* Prog.c */  
  
#include<stdio.h>  
  
#define NUM int  
  
#define OUT printf
```

```

NUM main()

{

NUM a,b,c;

a=45;

b=25;

c=a+b;

OUT("Sum %d",c);

return 0;

}

```

In the above example NUM, OUT are two macros and having their replacement text. The preprocessor replaces the macro names with their replacement text as

```

/* Prog.i */

prog.c 3: int main()

prog.c 4: {

prog.c 5: int a,b,c;

prog.c 6: a=45;

prog.c 7: b=25;

prog.c 8: c=a+b;

```

```
prog.c 9: printf("Sum %d",c);
```

```
prog.c 10: return 0;
```

```
prog.c 11: }
```

The above code is intermediate or expanded source generated by C preprocessor. Here we can find the replacement of macros with their replacement text.

1. While defining a macro, name of macro is written in capitals to identify that it is a macro and the replacement text may be a constant, a word, a statement or a part of program. While defining a macro in multiple lines \ is placed at the end of each line
2. Macro is not a variable, takes no space; its value can't be changes by assigning a new value
3. The scope of macro is from its point of declaration to the end of source file being compiled
4. A macro definition may use previous macro definition
5. Macro substitution is only done for tokens but not for strings
6. A macro can be undefined using #undef

Point 1:

Constant as macro

Example:

```
#define PI 3.14
```

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

```
int main()
{
    int rad;

    float area,cir;

    printf("Enter the radios:");

    scanf("%d",&rad);

    printf("Area %f",PI*rad*rad);

    printf("\nCircumference %f",2*PI*rad);

    return 0;
}
```

Execution:

Enter the radios: 15

Area 706.500000

Circumference 94.200005

Example explained:

Show Expanded Source

While executing the program PI is replaced with 3.14 by preprocessor in the source code.

If we want to change the value of PI as 3.142857 in place of 3.14 then we no need to change the total program rather it is enough to change the macro definition as

```
#define PI 3.142857
```

Specification: Accept the current month energy meter reading and previous month meter reading and print the power bill.

```
#define C 6.80

#define I 12.50

#define A 0.50

#include<stdio.h>

int main()

{

    int cmr,pmr,nu;

    char type;

    float bill;

    printf("The type of connection [domestic/commercial/industrial/agriculture] d/c/i/a:");

    scanf("%c",&type);

    printf("Enter the current month meter reading:");

    scanf("%d",&cmr);

    printf("Enter the previous month meter reading:");

    scanf("%d",&pmr);

    nu=cmr-pmr;
```



```

if(type=='d')

bill=nu*D;

else if(type=='c')

bill=nu*C;

else if(type=='i')

bill=nu*I;

else

bill=nu*A;

printf("Number of units consumed %d",nu);

printf("\nTotal bill %f",bill);

return 0;

}

```

Execution:

The type of connection [domestic/commercial/industrial/agriculture] d/c/i/a: c

Enter the current month meter reading: 500

Enter the previous month meter reading: 200

Number of units consumed: 300

Total bill 2040

Example explained:

Show Expanded Source

If the tariff changes in future then replacement text of macros may be changed without changing the code in the program.

Defining a word as macro

A word can be defined as macro. It is mostly done to improve the readability and understandability of program.

Example:

```
#define out printf

#define in scanf

#include<stdio.h>

int main()

{

int x,y,z;

out("Enter two numbers:\n");

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

z=x+y;

out("Sum=%d",z);

return 0;

}
```

Execution:

Enter two numbers:

6

9

Sum=15

Example explained:

Show Expanded Source

The preprocessor replaces “printf” in place of “out” and “scanf” in place of “in” before compilation of program.

Defining a part of program as macro

A statement or number of statements can be defined with a macro.

Example:

```
#define B int main()
```

```
#define C }
```

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

```
B
```

```
{
```

```
printf("Hello world");
```

```
return 0;
```

```
C
```

Output:

Hello world

Example explained:

Show Expanded Source

Here B is replaced with `int main()` and C is replaced with `}` by the preprocessor before compilation.

Example:

```
#define B int main()\n\n    {\n\n#define C return 0;\n\n    }\n\n#include<stdio.h>\n\nB\n\nprintf("Hello world");\n\nC
```

Output:

Hello world

Example explained

Show Expanded Source

Here multiple statements are defining as macro. `\` is used to extend a line.

PREVIOUS POST: 12.3 – Generating assembly code

NEXT POST: 12.5 – Working with macros – Part 2

File inclusion directives

In this article, I am going to discuss the File Inclusion Directives in C with Examples. Please read our previous article, where we discussed Macro Substitution Directives in C. At the end of this article, you will understand what File Inclusion Directives in C are and when and how to use File Inclusion Directives in C Program with examples. File Inclusion Pre-Processor (`#include`) in C:

By using this pre-processor, we can include a file in another file. Generally, by using this pre-processor, we are including the Header file. A header file is a source file that

contains forward declaration of predefined functions, global variables, constants value, predefined datatypes, predefined structures, predefined macros, inline functions. .h files don't provide any implementation part of predefined functions; it provides only forward declaration (prototype). A C program is a combination of predefined and user-defined functions. .C file contains the implementation part of user-defined functions and calling statements of predefined functions. If the functions are user-defined or predefined, the logic part must be required. Project-related .obj files provide the implementation of user-defined functions, .lib files provides implementation part of pre-defined functions which is loaded at the time of linking.

As per the function approach, when we are calling a function which is defined later for avoiding the compilation error, we are required to go for forwarding declaration i.e. prototype is required. If the function is user-defined, we can provide forward declaration explicitly but if it is the pre-defined function, we required to use header-file. In C programming language, .h files provide prototypes of pre-defined function. As a programmer, it is possible to provide the forward declaration of pre-defined function explicitly but when we are providing forward declaration then compiler thinks it is a user-defined function so not recommended. .h files don't pass for compilation process but .h file code is compiled. When we are including any header files at the time of pre-processing, that header file code will be substituted into current source code and with current source code header file code also compile.

Syntax: `#include<filename.h>` Or `#include "filename.h"`

`#include<filename.h>`:

By using this syntax, when we are including header file then it will be loaded from default directory i.e. C:\TC\INCLUDE. Generally, by using this syntax we are including pre-defined header files. When we are including pre-defined header files. When we are including user-defined header files by using this syntax then we need to place a user-defined header file in predefined header directory i.e. C:\TC\INCLUDE.

`#include "filename.h"`:

By using this syntax, when we are including header, then it is loaded from the current working directory. Generally, by using this syntax we are including user-defined header files. By using this syntax, when we are including pre-defined header files then first it will search in the current project directory if it is not available then loaded from default directory so it is a time-taking process.

In the next article, I am going to discuss Conditional Compilation Directives in C language. Here, in this article, I try to explain File Inclusion Directives in C. I hope you enjoy this File Inclusion directive in C article. I would like to have your feedback. Please post your feedback, question, or comments about this article.

Conditional compilation

This C Tutorial explains Conditional Compilation in C programming.

Well! We all know that while debugging the source code of a program, we generally include `printf()` statements at several places of doubts to know until where has been execution going correct, until where have been if the values of required variables evaluated correctly? These statements we rather would not physically remove from the source code as the same might be required again while maintenance modifications of the program. In such situations, conditional compilation is the perfect! Let's consider a simple ex.,

```
#include <stdio.h>
#define DEBUG printf("value of x = %d and y = %d.\n", x, y)

void increase(int, int);
int main(void)
{
    int x = 5, y = 6;

    x++;
    y++;
    DEBUG;
    increase(++x, y++);
    DEBUG;

    ++x;
    ++y;
    DEBUG;
}

void increase(int x, int y)
{
    DEBUG;
    x++;
    y++;
    DEBUG;
}
```

Notice that we inserted `DEBUG` statements at several places to know the modified values of variables `x` and `y` to ascertain the way how are they being evaluated in the program? But, of course, we wouldn't like them to appear in the output once the evaluation trend is clear. we rather hide them by enclosing in conditional directives as follows,

```
#if 0
    DEBUG;
#endif
```

Notice the `#if` construct which had its matching `#endif`. This is the simplest conditional compilation construct. Constant exp. zero '0' following `#if` is considered false and therefore preprocessor deleted the entire `#if` construct from its output while they are present in the source code.

`#if` construct also has optional `#elif` and `#else` constructs. There can be used any no. of `#elif` constructs. Let's consider their syntax first,

```
#if constan_exp
    /* statements */
#elif constant-exp
    /* other statements */
#else
    /* othet statements */
```

Notice that constant-exp must be a `#defined` symbol or literal constant! Variables that don't attain their values until run time are not legal candidates because their values can't be determined at the compile time. Let's see another ex.,

```
#define MOUSE 10
#define CAT 0

#if MOUSE
    #include "mouse.h"
#elif CAT
    #include "cat.h"
#else
    #include "horse.h"
#endif

int main(void)
{
    int x = 10;

    printf("value of x is %d\n", x);
    return 0;
}
```

Observe in above program, how conditional compilation directives cause compiler what fragment of code to compile and what to skip. Also, note that any constant exp. is evaluated only if all previous ones are false. If none of the constant exp. is true, and else clause is present, it's then executed. Preprocessor simply deletes those clauses, constant exp. for which is false.

Bitwise Operators

The following table lists the Bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then –

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e., 1100 0011

<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Example

Try the following example to understand all the bitwise operators available in C –

[Live Demo](#)

```
#include <stdio.h>

main() {

    unsigned int a = 60;    /* 60 = 0011 1100 */
    unsigned int b = 13;   /* 13 = 0000 1101 */
    int c = 0;

    c = a & b;    /* 12 = 0000 1100 */
    printf("Line 1 - Value of c is %d\n", c);

    c = a | b;    /* 61 = 0011 1101 */
    printf("Line 2 - Value of c is %d\n", c);

    c = a ^ b;    /* 49 = 0011 0001 */
    printf("Line 3 - Value of c is %d\n", c);

    c = ~a;       /* -61 = 1100 0011 */
    printf("Line 4 - Value of c is %d\n", c);

    c = a << 2;   /* 240 = 1111 0000 */
    printf("Line 5 - Value of c is %d\n", c);

    c = a >> 2;   /* 15 = 0000 1111 */
    printf("Line 6 - Value of c is %d\n", c);
}
```

When you compile and execute the above program, it produces the following result –

Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15

Shift operators

The bitwise shift operators are the right-shift operator (\gg), which moves the bits of shift-expression to the right, and the left-shift operator (\ll), which moves the bits of shift-expression to the left.¹

Syntax

```
shift-expression << additive-expression  
shift-expression >> additive-expression
```

Remarks

Important

The following descriptions and examples are valid on Windows for x86 and x64 architectures. The implementation of left-shift and right-shift operators is significantly different on Windows for ARM devices. For more information, see the "Shift Operators" section of the **Hello ARM** blog post.

Left Shifts

The left-shift operator causes the bits in shift-expression to be shifted to the left by the number of positions specified by additive-expression. The bit positions that have been vacated by the shift operation are zero-filled. A left shift is a logical shift (the bits that are shifted off the end are discarded, including the sign bit). For more information about the kinds of bitwise shifts, see Bitwise shifts.

The following example shows left-shift operations using unsigned numbers. The example shows what is happening to the bits by representing the value as a bitset. For more information, see bitset Class.

C++Copy

```
#include <iostream>  
#include <bitset>
```

```

using namespace std;

int main() {
    unsigned short short1 = 4;
    bitset<16> bitset1{short1}; // the bitset representation of 4
    cout << bitset1 << endl; // 0b00000000'00000100

    unsigned short short2 = short1 << 1; // 4 left-shifted by 1 = 8
    bitset<16> bitset2{short2};
    cout << bitset2 << endl; // 0b00000000'00001000

    unsigned short short3 = short1 << 2; // 4 left-shifted by 2 = 16
    bitset<16> bitset3{short3};
    cout << bitset3 << endl; // 0b00000000'00010000
}

```

If you left-shift a signed number so that the sign bit is affected, the result is undefined. The following example shows what happens when a 1 bit is left-shifted into the sign bit position.

C++Copy

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short short1 = 16384;
    bitset<16> bitset1(short1);
    cout << bitset1 << endl; // 0b01000000'00000000

    short short3 = short1 << 1;
    bitset<16> bitset3(short3); // 16384 left-shifted by 1 = -32768
    cout << bitset3 << endl; // 0b10000000'00000000

    short short4 = short1 << 14;
    bitset<16> bitset4(short4); // 4 left-shifted by 14 = 0
    cout << bitset4 << endl; // 0b00000000'00000000
}

```

Right Shifts

The right-shift operator causes the bit pattern in shift-expression to be shifted to the right by the number of positions specified by additive-expression. For unsigned

numbers, the bit positions that have been vacated by the shift operation are zero-filled. For signed numbers, the sign bit is used to fill the vacated bit positions. In other words, if the number is positive, 0 is used, and if the number is negative, 1 is used.

Important

The result of a right-shift of a signed negative number is implementation-dependent. Although the Microsoft C++ compiler uses the sign bit to fill vacated bit positions, there is no guarantee that other implementations also do so.

This example shows right-shift operations using unsigned numbers:

C++Copy

```
#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned short short11 = 1024;
    bitset<16> bitset11{short11};
    cout << bitset11 << endl;    // 0b00000100'00000000

    unsigned short short12 = short11 >> 1; // 512
    bitset<16> bitset12{short12};
    cout << bitset12 << endl;    // 0b00000010'00000000

    unsigned short short13 = short11 >> 10; // 1
    bitset<16> bitset13{short13};
    cout << bitset13 << endl;    // 0b00000000'00000001

    unsigned short short14 = short11 >> 11; // 0
    bitset<16> bitset14{short14};
    cout << bitset14 << endl;    // 0b00000000'00000000
}
```

The next example shows right-shift operations with positive signed numbers.

C++Copy

```
#include <iostream>
#include <bitset>

using namespace std;
```

```

int main() {
    short short1 = 1024;
    bitset<16> bitset1(short1);
    cout << bitset1 << endl;    // 0b000000100'00000000

    short short2 = short1 >> 1; // 512
    bitset<16> bitset2(short2);
    cout << bitset2 << endl;    // 0b00000010'00000000

    short short3 = short1 >> 11; // 0
    bitset<16> bitset3(short3);
    cout << bitset3 << endl;    // 0b00000000'00000000
}

```

The next example shows right-shift operations with negative signed integers.

C++Copy

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short neg1 = -16;
    bitset<16> bn1(neg1);
    cout << bn1 << endl; // 0b11111111'11110000

    short neg2 = neg1 >> 1; // -8
    bitset<16> bn2(neg2);
    cout << bn2 << endl; // 0b11111111'11111000

    short neg3 = neg1 >> 2; // -4
    bitset<16> bn3(neg3);
    cout << bn3 << endl; // 0b11111111'11111100

    short neg4 = neg1 >> 4; // -1
    bitset<16> bn4(neg4);
    cout << bn4 << endl; // 0b11111111'11111111

    short neg5 = neg1 >> 5; // -1
    bitset<16> bn5(neg5);
    cout << bn5 << endl; // 0b11111111'11111111
}

```

Shifts and Promotions

The expressions on both sides of a shift operator must be integral types. Integral promotions are performed according to the rules described in Standard Conversions. The type of the result is the same as the type of the promoted shift-expression.

In the following example, a variable of type **char** is promoted to an **int**.

C++Copy

```
#include <iostream>
#include <typeinfo>

using namespace std;

int main() {
    char char1 = 'a';

    auto promoted1 = char1 << 1; // 194
    cout << typeid(promoted1).name() << endl; // int

    auto promoted2 = char1 << 10; // 99328
    cout << typeid(promoted2).name() << endl; // int
}
```

Additional Details

The result of a shift operation is undefined if additive-expression is negative or if additive-expression is greater than or equal to the number of bits in the (promoted) shift-expression. No shift operation is performed if additive-expression is 0.

C++Copy

```
#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned int int1 = 4;
    bitset<32> b1{int1};
    cout << b1 << endl; // 0b00000000'00000000'00000000'00000100
}
```

```

    unsigned int int2 = int1 << -3; // C4293: '<<': shift count negative or too big,
undefined behavior
    unsigned int int3 = int1 >> -3; // C4293: '>>': shift count negative or too big,
undefined behavior
    unsigned int int4 = int1 << 32; // C4293: '<<': shift count negative or too big,
undefined behavior
    unsigned int int5 = int1 >> 32; // C4293: '>>': shift count negative or too big,
undefined behavior
    unsigned int int6 = int1 << 0;
    bitset<32> b6{int6};
    cout << b6 << endl; // 0b00000000'00000000'00000000'00000100 (no change)
}

```

Footnotes

¹ The following is the description of the shift operators in the C++11 ISO specification (INCITS/ISO/IEC 14882-2011[2012]), sections 5.8.2 and 5.8.3.

The value of $E1 \ll E2$ is $E1$ left-shifted $E2$ bit positions; vacated bits are zero-filled. If $E1$ has an unsigned type, the value of the result is $E1 \times 2^{E2}$, reduced modulo one more than the maximum value representable in the result type. Otherwise, if $E1$ has a signed type and non-negative value, and $E1 \times 2^{E2}$ is representable in the corresponding unsigned type of the result type, then that value, converted to the result type, is the resulting value; otherwise, the behavior is undefined.

The value of $E1 \gg E2$ is $E1$ right-shifted $E2$ bit positions. If $E1$ has an unsigned type or if $E1$ has a signed type and a non-negative value, the value of the result is the integral part of the quotient of $E1/2^{E2}$. If $E1$ has a signed type and a negative value, the resulting value is implementation-defined.

Masks

Bit strings and bitwise operators are often used to make masks. A mask is a bit string that "fits over" another bit string and produces a desired result, such as singling out particular bits from the second bit string, when the two bit strings are operated upon. This is particularly useful for handling flags; programmers often wish to know whether one particular flag is set in a bit string, but may not care about the others. For example, you might create a mask that only allows the flag of interest to have a non-zero value, then AND that mask with the bit string containing the flag.

Consider the following mask, and two bit strings from which we want to extract the final bit:

```

mask = 00000001
value1 = 10011011
value2 = 10011100

```

```
mask & value1 == 00000001
mask & value2 == 00000000
```

The zeros in the mask mask off the first seven bits and only let the last bit show through. (In the case of the first value, the last bit is 1; in the case of the second value, the last bit is 0.)

Alternatively, masks can be built up by operating on several flags, usually with inclusive OR:

```
flag1 = 00000001
flag2 = 00000010
flag3 = 00000100
```

```
mask = flag1 | flag2 | flag3
```

```
mask == 00000111
```

See *Opening files at a low level*, for a code example that actually uses bitwise OR to join together several flags.

It should be emphasized that the flag and mask examples are written in pseudo-code, that is, a means of expressing information that resembles source code, but cannot be compiled. It is not possible to use binary numbers directly in C.

The following code example shows how bit masks and bit-shifts can be combined. It accepts a decimal number from the user between 0 and 128, and prints out a binary number in response.

```
#include <stdio.h>
#define NUM_OF_BITS 8

/* To shorten example, not using argp */
int main ()
{
    char *my_string;
    int input_int, args_assigned;
    int nbytes = 100;
    short my_short, bit;
    int idx;

    /* This hex number is the same as binary 10000000 */
    short MASK = 0x80;

    args_assigned = 0;
```



```

input_int = -1;

while ((args_assigned != 1) ||
      (input_int < 0) || (input_int > 128))
{
    puts ("Please enter an integer from 0 to 128.");
    my_string = (char *) malloc (nbytes + 1);
    getline (&my_string, &nbytes, stdin);
    args_assigned = sscanf (my_string, "%d", &input_int);
    if ((args_assigned != 1) ||
        (input_int < 0) || (input_int > 128))
        puts ("\nInput invalid!");
}

my_short = (short) input_int;

printf ("Binary value = ");

/*
   Convert decimal numbers into binary
   Keep shifting my_short by one to the left
   and test the highest bit. This does
   NOT preserve the value of my_short!
*/

for (idx = 0; idx < NUM_OF_BITS; idx++)
{
    bit = my_short & MASK;
    printf ("%d", bit/MASK);
    my_short <<= 1;
}

printf ("\n");
return 0;
}

```

Bit field

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follows –

```

struct {
    unsigned int widthValidated;
    unsigned int heightValidated;
} status;

```

This structure requires 8 bytes of memory space but in actual, we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes. For example, the above structure can be re-written as follows –

```
struct {
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
} status;
```

The above structure requires 4 bytes of memory space for status variable, but only 2 bits will be used to store the values.

If you will use up to 32 variables each one with a width of 1 bit, then also the status structure will use 4 bytes. However as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes. Let us check the following example to understand the concept –

[Live Demo](#)

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

/* define simple structure */
struct {
    unsigned int widthValidated;
    unsigned int heightValidated;
} status1;

/* define a structure with bit fields */
struct {
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
} status2;

int main( ) {
    printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
    printf( "Memory size occupied by status2 : %d\n", sizeof(status2));
    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

Bit Field Declaration

The declaration of a bit-field has the following form inside a structure –

```
struct {  
    type [member_name] : width ;  
};
```

The following table describes the variable elements of a bit field –

Sr.No.	Element & Description
1	type An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.
2	member_name The name of the bit-field.
3	width The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

The variables defined with a predefined width are called **bit fields**. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows –

```
struct {  
    unsigned int age : 3;  
} Age;
```

The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so. Let us try the following example –

[Live Demo](#)

```
#include <stdio.h>  
#include <string.h>  
  
struct {  
    unsigned int age : 3;
```

```
} Age;

int main() {

    Age.age = 4;
    printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
    printf( "Age.age : %d\n", Age.age );

    Age.age = 7;
    printf( "Age.age : %d\n", Age.age );

    Age.age = 8;
    printf( "Age.age : %d\n", Age.age );

    return 0;
}
```

When the above code is compiled it will compile with a warning and when executed, it produces the following result –

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

UNIT-V

File handling

Definition of Files

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file
- Writing to the file
- Deleting the file

Functions for file handling

There are many functions in the C library to open, read, write, search and close the file. A list of file functions are given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position

8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

Opening File: fopen()

We must open a file before it can be read, write, or update. The fopen() function is used to open a file. The syntax of the fopen() is given below.

1. **FILE *fopen(const char * filename, const char * mode);**

The fopen() function accepts two parameters:

- The file name (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like "**c://some_folder/some_file.ext**".
- The mode in which the file is to be opened. It is a string.

We can use one of the following modes in the fopen() function.

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode

a+	opens a text file in read and write mode
rb	opens a binary file in read mode
wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and write mode

The fopen function works in the following way.

- Firstly, It searches the file to be opened.
- Then, it loads the file from the disk and place it into the buffer. The buffer is used to provide efficiency for the read operations.
- It sets up a character pointer which points to the first character of the file.

Consider the following example which opens a file in write mode.

```
1. #include<stdio.h>
2. void main( )
3. {
4. FILE *fp ;
5. char ch ;
6. fp = fopen("file_handle.c","r") ;
7. while ( 1 )
8. {
9. ch = fgetc ( fp ) ;
10. if ( ch == EOF )
11. break ;
12. printf("%c",ch) ;
```

- 13.}
14. fclose (fp) ;
- 15.}

Output

The content of the file will be printed.

```
#include;
void main( )
{
FILE *fp; // file pointer
char ch;
fp = fopen("file_handle.c","r");
while ( 1 )
{
ch = fgetc ( fp ); //Each character of the file is read and stored in the character file.
if ( ch == EOF )
break;
printf("%c",ch);
}
fclose (fp );
}
```

Closing File: fclose()

The fclose() function is used to close a file. The file must be closed after performing all the operations on it. The syntax of fclose() function is given below:

1. **int** fclose(**FILE** *fp);

Opening modes of files

A File can be used to store a large volume of persistent data. Like many other languages 'C' provides following file management functions,

1. Creation of a file
2. Opening a file
3. Reading a file
4. Writing to a file
5. Closing a file

Following are the most important file management functions available in 'C,'

function	purpose
fopen ()	Creating a file or opening an existing file
fclose ()	Closing a file
fprintf ()	Writing a block of data to a file
fscanf ()	Reading a block data from a file
getc ()	Reads a single character from a file
putc ()	Writes a single character to a file
getw ()	Reads an integer from a file
putw ()	Writing an integer to a file
fseek ()	Sets the position of a file pointer to a specified location
ftell ()	Returns the current position of a file pointer
rewind ()	Sets the file pointer at the beginning of a file

- How to Create a File
- How to Close a file:

- Writing to a File
 - fputc() Function:
 - fputs () Function:
 - fprintf()Function:
- Reading data from a File
- Interactive File Read and Write with getc and putc

How to Create a File

Whenever you want to work with a file, the first step is to create a file. A file is nothing but space in a memory where data is stored.

To create a file in a 'C' program following syntax is used,

```
FILE *fp;
fp = fopen ("file_name", "mode");
```

In the above syntax, the file is a data structure which is defined in the standard library.

fopen is a standard function which is used to open a file.

- If the file is not present on the system, then it is created and then opened.
- If a file is already present on the system, then it is directly opened using this function.

fp is a file pointer which points to the type file.

Whenever you open or create a file, you have to specify what you are going to do with the file. A file in 'C' programming can be created or opened for reading/writing purposes. A mode is used to specify whether you want to open a file for any of the below-given purposes. Following are the different types of modes in 'C' programming which can be used while working with a file.

File Mode	Description
r	Open a file for reading. If a file is in reading mode, then no data is deleted if a file is already present on a system.

w	Open a file for writing. If a file is in writing mode, then a new file is created if a file doesn't exist at all. If a file is already present on a system, then all the data inside the file is truncated, and it is opened for writing purposes.
a	Open a file in append mode. If a file is in append mode, then the file is opened. The content within the file doesn't change.
r+	open for reading and writing from beginning
w+	open for reading and writing, overwriting a file
a+	open for reading and writing, appending to file

In the given syntax, the filename and the mode are specified as strings hence they must always be enclosed within double quotes.

Example:

```
#include <stdio.h>
int main() {
FILE *fp;
fp = fopen ("data.txt", "w");
}
```

Output:

File is created in the same folder where you have saved your code.

You can specify the path where you want to create your file

```
#include <stdio.h>
int main() {
FILE *fp;
fp = fopen ("D://data.txt", "w");
}
```

How to Close a file

One should always close a file whenever the operations on file are over. It means the contents and links to the file are terminated. This prevents accidental damage to the file.

'C' provides the fclose function to perform file closing operation. The syntax of fclose is as follows,

```
fclose (file_pointer);
```

Example:

```
FILE *fp;
fp = fopen ("data.txt", "r");
fclose (fp);
```

The fclose function takes a file pointer as an argument. The file associated with the file pointer is then closed with the help of fclose function. It returns 0 if close was successful and EOF (end of file) if there is an error has occurred while file closing.

After closing the file, the same file pointer can also be used with other files.

In 'C' programming, files are automatically close when the program is terminated. Closing a file manually by writing fclose function is a good programming practice.

Writing to a File

In C, when you write to a file, newline characters '\n' must be explicitly added.

The stdio library offers the necessary functions to write to a file:

- **fputc(char, file_pointer):** It writes a character to the file pointed to by file_pointer.
- **fputs(str, file_pointer):** It writes a string to the file pointed to by file_pointer.
- **fprintf(file_pointer, str, variable_lists):** It prints a string to the file pointed to by file_pointer. The string can optionally include format specifiers and a list of variables variable_lists.

The program below shows how to perform writing to a file:

fputc() Function:

```
#include <stdio.h>
int main() {
    int i;
    FILE * fptr;
    char fn[50];
    char str[] = "Guru99 Rocks\n";
    fptr = fopen("fputc_test.txt", "w"); // "w" defines "writing mode"
    for (i = 0; str[i] != '\n'; i++) {
        /* write to file using fputc() function */
        fputc(str[i], fptr);
    }
    fclose(fptr);
    return 0;
}
```

Output:

The above program writes a single character into the **fputc_test.txt** file until it reaches the next line symbol "\n" which indicates that the sentence was successfully written. The process is to take each character of the array and write it into the file.

1. In the above program, we have created and opened a file called fputc_test.txt in a write mode and declare our string which will be written into the file.
2. We do a character by character write operation using for loop and put each character in our file until the "\n" character is encountered then the file is closed using the fclose function.

fputs () Function:

```
#include <stdio.h>
int main() {
    FILE * fp;
    fp = fopen("fputs_test.txt", "w+");
    fputs("This is Guru99 Tutorial on fputs", fp);
    fputs("We don't need to use for loop\n", fp);
    fputs("Easier than fputc function\n", fp);
    fclose(fp);
    return (0);
}
```

OUTPUT:

1. In the above program, we have created and opened a file called fputs_test.txt in a write mode.
2. After we do a write operation using fputs() function by writing three different strings
3. Then the file is closed using the fclose function.

fprintf()Function:

```
#include <stdio.h>
int main() {
    FILE *fptr;
    fptr = fopen("fprintf_test.txt", "w"); // "w" defines "writing mode"
    /* write to file */
    fprintf(fptr, "Learning C with Guru99\n");
    fclose(fptr);
    return 0;
}
```

OUTPUT:

1. In the above program we have created and opened a file called fprintf_test.txt in a write mode.
2. After a write operation is performed using fprintf() function by writing a string, then the file is closed using the fclose function.

Reading data from a File

There are three different functions dedicated to reading data from a file

- **fgetc(file_pointer):** It returns the next character from the file pointed to by the file pointer. When the end of the file has been reached, the EOF is sent back.
- **fgets(buffer, n, file_pointer):** It reads n-1 characters from the file and stores the string in a buffer in which the NULL character '\0' is appended as the last character.
- **fscanf(file_pointer, conversion_specifiers, variable_addresses):** It is used to parse and analyze data. It reads characters from the file and assigns the input to a list of variable pointers variable_addresses using conversion specifiers. Keep in

mind that as with scanf, fscanf stops reading a string when space or newline is encountered.

The following program demonstrates reading from fputs_test.txt file using fgets(), fscanf() and fgetc () functions respectively :

```
#include <stdio.h>
int main() {
    FILE * file_pointer;
    char buffer[30], c;

    file_pointer = fopen("fprintf_test.txt", "r");
    printf("----read a line----\n");
    fgets(buffer, 50, file_pointer);
    printf("%s\n", buffer);

    printf("----read and parse data----\n");
    file_pointer = fopen("fprintf_test.txt", "r"); //reset the pointer
    char str1[10], str2[2], str3[20], str4[2];
    fscanf(file_pointer, "%s %s %s %s", str1, str2, str3, str4);
    printf("Read String1 |%s|\n", str1);
    printf("Read String2 |%s|\n", str2);
    printf("Read String3 |%s|\n", str3);
    printf("Read String4 |%s|\n", str4);

    printf("----read the entire file----\n");

    file_pointer = fopen("fprintf_test.txt", "r"); //reset the pointer
    while ((c = getc(file_pointer)) != EOF) printf("%c", c);

    fclose(file_pointer);
    return 0;
}
```

Result:

```
----read a line----
Learning C with Guru99

----read and parse data----
Read String1 |Learning|
Read String2 |C|
Read String3 |with|
```

Read String4 |Guru99|

----read the entire file----

Learning C with Guru99

1. In the above program, we have opened the file called "fprintf_test.txt" which was previously written using fprintf() function, and it contains "Learning C with Guru99" string. We read it using the fgets() function which reads line by line where the buffer size must be enough to handle the entire line.
2. We reopen the file to reset the pointer file to point at the beginning of the file. Create various strings variables to handle each word separately. Print the variables to see their contents. The fscanf() is mainly used to extract and parse data from a file.
3. Reopen the file to reset the pointer file to point at the beginning of the file. Read data and print it from the file character by character using getc() function until the EOF statement is encountered
4. After performing a reading operation file using different variants, we again closed the file using the fclose function.

Interactive File Read and Write with getc and putc

These are the simplest file operations. Getc stands for get character, and putc stands for put character. These two functions are used to handle only a single character at a time.

Following program demonstrates the file handling functions in 'C' programming:

```
#include <stdio.h>
int main() {
    FILE * fp;
    char c;
    printf("File Handling\n");
    //open a file
    fp = fopen("demo.txt", "w");
    //writing operation
    while ((c = getchar()) != EOF) {
        putc(c, fp);
    }
    //close file
    fclose(fp);
    printf("Data Entered:\n");
    //reading
```



```
fp = fopen("demo.txt", "r");
while ((c = getc(fp)) != EOF) {
    printf("%c", c);
}
fclose(fp);
return 0;
}
```

Output:

1. In the above program we have created and opened a file called demo in a write mode.
2. After a write operation is performed, then the file is closed using the fclose function.
3. We have again opened a file which now contains data in a reading mode. A while loop will execute until the eof is found. Once the end of file is found the operation will be terminated and data will be displayed using printf function.
4. After performing a reading operation file is again closed using the fclose function.

Standard function

In this tutorial, you'll learn about the standard library functions in C. More specifically, what are they, different library functions in C and how to use them in your program.

C Standard library functions or simply C Library functions are inbuilt functions in C programming.

The prototype and data definitions of these functions are present in their respective header files. To use these functions we need to include the header file in our program. For example,

If you want to use the printf() function, the header file <stdio.h> should be included.

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

```
int main()
```

```
{
```

```
    printf("Catch me if you can.");
```

```
}
```

If you try to use `printf()` without including the `stdio.h` header file, you will get an error.

Advantages of Using C library functions

1. They work

One of the most important reasons you should use library functions is simply because they work. These functions have gone through multiple rigorous testing and are easy to use.

2. The functions are optimized for performance

Since, the functions are "standard library" functions, a dedicated group of developers constantly make them better. In the process, they are able to create the most efficient code optimized for maximum performance.

3. It saves considerable development time

Since the general functions like printing to a screen, calculating the square root, and many more are already written. You shouldn't worry about creating them once again.

4. The functions are portable

With ever-changing real-world needs, your application is expected to work every time, everywhere. And, these library functions help you in that they do the same thing on every computer.

Example: Square root using `sqrt()` function

Suppose, you want to find the square root of a number.

To can compute the square root of a number, you can use the `sqrt()` library function. The function is defined in the `math.h` header file.

```
#include <stdio.h>

#include <math.h>

int main()
{
    float num, root;

    printf("Enter a number: ");

    scanf("%f", &num);
```

```
// Computes the square root of num and stores in root.  
root = sqrt(num);  
printf("Square root of %.2f = %.2f", num, root);  
return 0;  
}
```

When you run the program, the output will be:

Enter a number: 12

Square root of 12.00 = 3.46

Library Functions in Different Header Files

C Header Files

<code><assert.h></code>	Program assertion functions
<code><ctype.h></code>	Character type functions
<code><locale.h></code>	Localization functions
<code><math.h></code>	Mathematics functions
<code><setjmp.h></code>	Jump functions
<code><signal.h></code>	Signal handling functions
<code><stdarg.h></code>	Variable arguments handling functions

C Header Files

<stdio.h> Standard Input/Output functions

<stdlib.h> Standard Utility functions

[<string.h>](#) String handling functions

<time.h> Date time functions

fopen()

The C library function **FILE *fopen(const char *filename, const char *mode)** opens the **filename** pointed to, by filename using the given **mode**.

Declaration

Following is the declaration for fopen() function.

```
FILE *fopen(const char *filename, const char *mode)
```

Parameters

- **filename** – This is the C string containing the name of the file to be opened.
- **mode** – This is the C string containing a file access mode. It includes –

Sr.No.	Mode & Description
1	"r" Opens a file for reading. The file must exist.
2	"w" Creates an empty file for writing. If a file with the same name already exists, its content is erased and the file is considered as a new empty file.

3	<p>"a"</p> <p>Appends to a file. Writing operations, append data at the end of the file. The file is created if it does not exist.</p>
4	<p>"r+"</p> <p>Opens a file to update both reading and writing. The file must exist.</p>
5	<p>"w+"</p> <p>Creates an empty file for both reading and writing.</p>
6	<p>"a+"</p> <p>Opens a file for reading and appending.</p>

Return Value

This function returns a FILE pointer. Otherwise, NULL is returned and the global variable errno is set to indicate the error.

Example

The following example shows the usage of fopen() function.

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

int main () {
    FILE * fp;

    fp = fopen ("file.txt", "w+");
    fprintf(fp, "%s %s %s %d", "We", "are", "in", 2012);

    fclose(fp);

    return(0);
}
```

Let us compile and run the above program that will create a file **file.txt** with the following content –

We are in 2012

Now let us see the content of the above file using the following program –

```
#include <stdio.h>

int main () {
    FILE *fp;
    int c;

    fp = fopen("file.txt", "r");
    while(1) {
        c = fgetc(fp);
        if( feof(fp) ) {
            break ;
        }
        printf("%c", c);
    }
    fclose(fp);

    return(0);
}
```

Let us compile and run the above program to produce the following result –

We are in 2012

fclose()

The C library function **int fclose(FILE *stream)** closes the stream. All buffers are flushed.

Declaration

Following is the declaration for fclose() function.

```
int fclose(FILE *stream)
```

Parameters

- **stream** – This is the pointer to a FILE object that specifies the stream to be closed.

Return Value

This method returns zero if the stream is successfully closed. On failure, EOF is returned.

Example

The following example shows the usage of `fclose()` function.

```
#include <stdio.h>

int main () {
    FILE *fp;

    fp = fopen("file.txt", "w");

    fprintf(fp, "%s", "This is tutorialspoint.com");
    fclose(fp);

    return(0);
}
```

Let us compile and run the above program that will create a file **file.txt**, and then it will write following text line and finally it will close the file using **fclose()** function.

feof()

The C library function **int feof(FILE *stream)** tests the end-of-file indicator for the given stream.

Declaration

Following is the declaration for `feof()` function.

```
int feof(FILE *stream)
```

Parameters

- **stream** – This is the pointer to a FILE object that identifies the stream.

Return Value

This function returns a non-zero value when End-of-File indicator associated with the stream is set, else zero is returned.

Example

The following example shows the usage of `feof()` function.

```
#include <stdio.h>

int main () {
    FILE *fp;
```

```

int c;

fp = fopen("file.txt", "r");
if(fp == NULL) {
    perror("Error in opening file");
    return(-1);
}

while(1) {
    c = fgetc(fp);
    if( feof(fp) ) {
        break ;
    }
    printf("%c", c);
}
fclose(fp);

return(0);
}

```

fseek()

The C library function `int fseek(FILE *stream, long int offset, int whence)` sets the file position of the stream to the given offset.

Declaration

Following is the declaration for `fseek()` function.

```
int fseek(FILE *stream, long int offset, int whence)
```

Parameters

- `stream` – This is the pointer to a FILE object that identifies the stream.
- `offset` – This is the number of bytes to offset from whence.
- `whence` – This is the position from where offset is added. It is specified by one of the following constants –

Sr.No.	Constant & Description
1	SEEK_SET

	Beginning of file
2	SEEK_CUR Current position of the file pointer
3	SEEK_END End of file

Return Value

This function returns zero if successful, or else it returns a non-zero value.

Example

The following example shows the usage of fseek() function.

```
#include <stdio.h>

int main () {
    FILE *fp;

    fp = fopen("file.txt", "w+");
    fputs("This is tutorialspoint.com", fp);

    fseek( fp, 7, SEEK_SET );
    fputs(" C Programming Language", fp);
    fclose(fp);

    return(0);
}
```

Let us compile and run the above program that will create a file file.txt with the following content. Initially program creates the file and writes This is tutorialspoint.com but later we had reset the write pointer at 7th position from the beginning and used puts() statement which over-write the file with the following content -

```
This is C Programming Language
```

Now let's see the content of the above file using the following program -

```
#include <stdio.h>

int main () {
```

```
FILE *fp;
int c;

fp = fopen("file.txt","r");
while(1) {
    c = fgetc(fp);
    if( feof(fp) ) {
        break;
    }
    printf("%c", c);
}
fclose(fp);
return(0);
}
```

Let us compile and run the above program to produce the following result –

This is C Programming Language

rewind()

The C library function void `rewind(FILE *stream)` sets the file position to the beginning of the file of the given stream.

Declaration

Following is the declaration for `rewind()` function.

```
void rewind(FILE *stream)
```

Parameters

- `stream` – This is the pointer to a FILE object that identifies the stream.

Return Value

This function does not return any value.

Example

The following example shows the usage of `rewind()` function.

[Live Demo](#)

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

```

int main () {
    char str[] = "This is tutorialspoint.com";
    FILE *fp;
    int ch;

    /* First let's write some content in the file */
    fp = fopen( "file.txt" , "w" );
    fwrite(str , 1 , sizeof(str) , fp );
    fclose(fp);

    fp = fopen( "file.txt" , "r" );
    while(1) {
        ch = fgetc(fp);
        if( feof(fp) ) {
            break ;
        }
        printf("%c", ch);
    }
    rewind(fp);
    printf("\n");
    while(1) {
        ch = fgetc(fp);
        if( feof(fp) ) {
            break ;
        }
        printf("%c", ch);
    }
    fclose(fp);

    return(0);
}

```

Let us assume we have a text file file.txt that have the following content –

```
This is tutorialspoint.com
```

Now let us compile and run the above program to produce the following result –

```
This is tutorialspoint.com
```

Using text files:

fgetc()

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

Declaration

Following is the declaration for fgetc() function.

```
int fgetc(FILE *stream)
```

Parameters

- **stream** – This is the pointer to a FILE object that identifies the stream on which the operation is to be performed.

Return Value

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

Example

The following example shows the usage of fgetc() function.

```
#include <stdio.h>

int main () {
    FILE *fp;
    int c;
    int n = 0;

    fp = fopen("file.txt", "r");
    if(fp == NULL) {
        perror("Error in opening file");
        return(-1);
    } do {
        c = fgetc(fp);
        if( feof(fp) ) {
            break ;
        }
        printf("%c", c);
    } while(1);

    fclose(fp);
    return(0);
}
```

Let us assume, we have a text file **file.txt**, which has the following content. This file will be used as an input for our example program –

```
We are in 2012
```

Now, let us compile and run the above program that will produce the following result –

```
We are in 2012
```

fputc()

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

Declaration

Following is the declaration for fputc() function.

```
int fputc(int char, FILE *stream)
```

Parameters

- **char** – This is the character to be written. This is passed as its int promotion.
- **stream** – This is the pointer to a FILE object that identifies the stream where the character is to be written.

Return Value

If there are no errors, the same character that has been written is returned. If an error occurs, EOF is returned and the error indicator is set.

Example

The following example shows the usage of fputc() function.

```
#include <stdio.h>

int main () {
    FILE *fp;
    int ch;

    fp = fopen("file.txt", "w+");
    for( ch = 33 ; ch <= 100; ch++ ) {
        fputc(ch, fp);
    }
}
```

```
fclose(fp);  
  
return(0);  
}
```

Let us compile and run the above program that will create a file **file.txt** in the current directory, which will have following content –

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcd
```

Now let's see the content of the above file using the following program –

```
#include <stdio.h>  
  
int main () {  
    FILE *fp;  
    int c;  
  
    fp = fopen("file.txt", "r");  
    while(1) {  
        c = fgetc(fp);  
        if( feof(fp) ) {  
            break ;  
        }  
        printf("%c", c);  
    }  
    fclose(fp);  
    return(0);  
}
```

Let us compile and run above program to produce the following result –

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcd
```

fscanf()

The C library function **int fscanf(FILE *stream, const char *format, ...)** reads formatted input from a stream.

Declaration

Following is the declaration for fscanf() function.

```
int fscanf(FILE *stream, const char *format, ...)
```

Parameters

- **stream** – This is the pointer to a FILE object that identifies the stream.

- **format** – This is the C string that contains one or more of the following items
 - Whitespace character, Non-whitespace character and Format specifiers. A format specifier will be as [=%[*][width][modifiers]type=], which is explained below –

Sr.No.	Argument & Description
1	<p>*</p> <p>This is an optional starting asterisk indicates that the data is to be read from the stream but ignored, i.e. it is not stored in the corresponding argument.</p>
2	<p>width</p> <p>This specifies the maximum number of characters to be read in the current reading operation.</p>
3	<p>modifiers</p> <p>Specifies a size different from int (in the case of d, i and n), unsigned int (in the case of o, u and x) or float (in the case of e, f and g) for the data pointed by the corresponding additional argument: h : short int (for d, i and n), or unsigned short int (for o, u and x) l : long int (for d, i and n), or unsigned long int (for o, u and x), or double (for e, f and g) L : long double (for e, f and g)</p>
4	<p>type</p> <p>A character specifying the type of data to be read and how it is expected to be read. See next table.</p>

fscanf type specifiers

type	Qualifying Input	Type of argument
c	<p>Single character: Reads the next character. If a width different from 1 is specified, the function reads width characters and stores them in the successive locations of the array passed as argument. No null character is appended at the end.</p>	char *

d	Decimal integer: Number optionally preceded with a + or - sign	int *
e, E, f, g, G	Floating point: Decimal number containing a decimal point, optionally preceded by a + or - sign and optionally followed by the e or E character and a decimal number. Two examples of valid entries are -732.103 and 7.12e4	float *
o	Octal Integer:	int *
s	String of characters. This will read subsequent characters until a whitespace is found (whitespace characters are considered to be blank, newline and tab).	char *
u	Unsigned decimal integer.	unsigned int *
x, X	Hexadecimal Integer	int *

- **additional arguments** – Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter (if any). There should be the same number of these arguments as the number of %-tags that expect a value.

Return Value

This function returns the number of input items successfully matched and assigned, which can be fewer than provided for, or even zero in the event of an early matching failure.

Example

The following example shows the usage of fscanf() function.

[Live Demo](#)

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



```

int main () {
    char str1[10], str2[10], str3[10];
    int year;
    FILE * fp;

    fp = fopen ("file.txt", "w+");
    fputs("We are in 2012", fp);

    rewind(fp);
    fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);

    printf("Read String1 |%s|\n", str1 );
    printf("Read String2 |%s|\n", str2 );
    printf("Read String3 |%s|\n", str3 );
    printf("Read Integer |%d|\n", year );

    fclose(fp);

    return(0);
}

```

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

```

Read String1 |We|
Read String2 |are|
Read String3 |in|
Read Integer |2012|

```

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.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ". Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes -

```

#include <stdio.h>

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

    printf("Program name %s\n", argv[0]);

    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 a single argument separated by space but inside double quotes, it produces the following result.

```
$/a.out "testing1 testing2"
```

Program name ./a.out

The argument supplied is testing1 testing2