Increment and Decrement Operators in C

Increment Operator

Increment Operators are the unary operators used to increment or add 1 to the operand value. The Increment operand is denoted by the double plus symbol (++). It has two types, Pre Increment and Post Increment Operators.

Pre-increment Operator

The pre-increment operator is used to increase the original value of the operand by 1 before assigning it to the expression.

Syntax


```
1. X = ++A;
```

In the above syntax, the value of operand 'A' is increased by 1, and then a new value is assigned to the variable 'B'.

Example 1: Program to use the pre-increment operator in C

```
1. #include <stdio.h>
2. #include <conio.h>
3. int main ()
4. {
5.  // declare integer variables
6. int x, y, z;
7. printf (" Input the value of X: ");
8. scanf (" %d", &x);
9. printf ("  Input the value of Y: ");
10. scanf (" %d", &y);
11. printf (" Input the value of Z: ");
12. scanf (" %d", &z);
13. // use pre increment operator to update the value by 1
14. ++x;
15. ++y;
16. ++z;
17. printf (" \n The updated value of the X: %d ", x);
18. printf (" \n The updated value of the Y: %d ", y);
19. printf (" \n The updated value of the Z: %d ", z);
20. return 0;
21. }
```

Output

```
Input the value of X: 10
Input the value of Y: 15
Input the value of Z: 20

The updated value of the X: 11
The updated value of the Y: 16
The updated value of the Z: 21
```

Post increment Operator

The post-increment operator is used to increment the original value of the operand by 1 after assigning it to the expression.

Syntax

1. X = A++;

In the above syntax, the value of operand 'A' is assigned to the variable 'X'. After that, the value of variable 'A' is incremented by 1.

Example 2: Program to use the post-increment operator in C

```
1. #include <stdio.h>
2. #include <conio.h>
3. int main ()
4. {
5.  // declare integer variables
6. int x, y, z, a, b, c;
7. printf (" Input the value of X: ");
8. scanf (" %d", &x);
9. printf (" Input the value of Y: ");
10. scanf (" %d", &y);
11. printf (" Input the value of Z: ");
12. scanf (" %d", &z);
13. // use post-increment operator to update the value by 1
14. a = x++;
15. b = y++;
16. c = z++;
17. printf (" \n The original value of a: %d", a);
18. printf (" \n The original value of b: %d", b);
19. printf (" \n The original value of c: %d", c);
20. printf (" \n\n The updated value of the X: %d ", x);
21. printf (" \n The updated value of the Y: %d ", y);
22. printf (" \n The updated value of the Z: %d ", z);
23. return 0;
24. }
```

Output

```
Input the value of X: 10
Input the value of Y: 15
Input the value of Z: 20

The original value of a: 10
The original value of b: 15
The original value of c: 20

The updated value of the X: 11
The updated value of the Y: 16
The updated value of the Z: 21
```

Decrement Operator

Decrement Operator is the unary operator, which is used to decrease the original value of the operand by 1. The decrement operator is represented as the double minus symbol (--). It has two types, Pre Decrement and Post Decrement operators.

Pre Decrement Operator

The Pre Decrement Operator decreases the operand value by 1 before assigning it to the mathematical expression. In other words, the original value of the operand is first decreases, and then a new value is assigned to the other variable.

Syntax

1. B = --A;

In the above syntax, the value of operand 'A' is decreased by 1, and then a new value is assigned to the variable 'B'.

Output

```
Input the value of X: 5
Input the value of Y: 6
Input the value of Z: 7

The updated value of the X: 6
The updated value of the Y: 7
The updated value of the Z: 8
```

Post decrement Operator:

Post decrement operator is used to decrease the original value of the operand by 1 after assigning to the expression.

Syntax

1. B = A--;

In the above syntax, the value of operand 'A' is assigned to the variable 'B', and then the value of A is decreased by 1.

Difference between the Increment and Decrement Operator in C

Increment Operator	Decrement Operator
It is used to increment the value of a variable by 1.	It is used to decrease the operand values by 1.
The increment operator is represented as the double plus (++) symbol.	The decrement operator is represented as the double minus (--) symbol.
It has two types: pre-increment operator and post-increment operator.	Similarly, it has two types: the pre-decrement operator and the post-decrement operator.

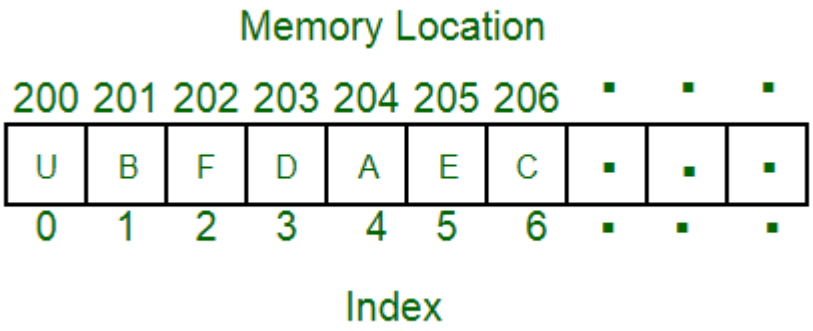
Pre increment operator means the value of the operator is incremented first and then used in the expression. The post-increment operator means the operand is first used in the expression and then performs the increment operation to the original value by 1.	Pre decrement means the value of the operator is decremented first and then assigned in the expression. Whereas the post decrement operator means the operand is first used in the expression and then performs the decrement operation to the operand's original value by 1.
Syntax for the pre increment operator: X = ++a; Syntax for the post increment operator: X = a++;	Syntax for the pre decrement operator: X = --a; Syntax for the post decrement operator: X = a--;
Both operators' works only to the single operand, not values.	Both operators' works only to the single operand, not values.

Introduction to Arrays

geeksforgeeks.org/introduction-to-arrays

October 27, 2017

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array). The base value is index 0 and the difference between the two indexes is the offset.
For simplicity, we can think of an array as a fleet of stairs where on each step is placed a value (let's say one of your friends). Here, you can identify the location of any of your friends by simply knowing the count of the step they are on.
Remember: "Location of next index depends on the data type we use".

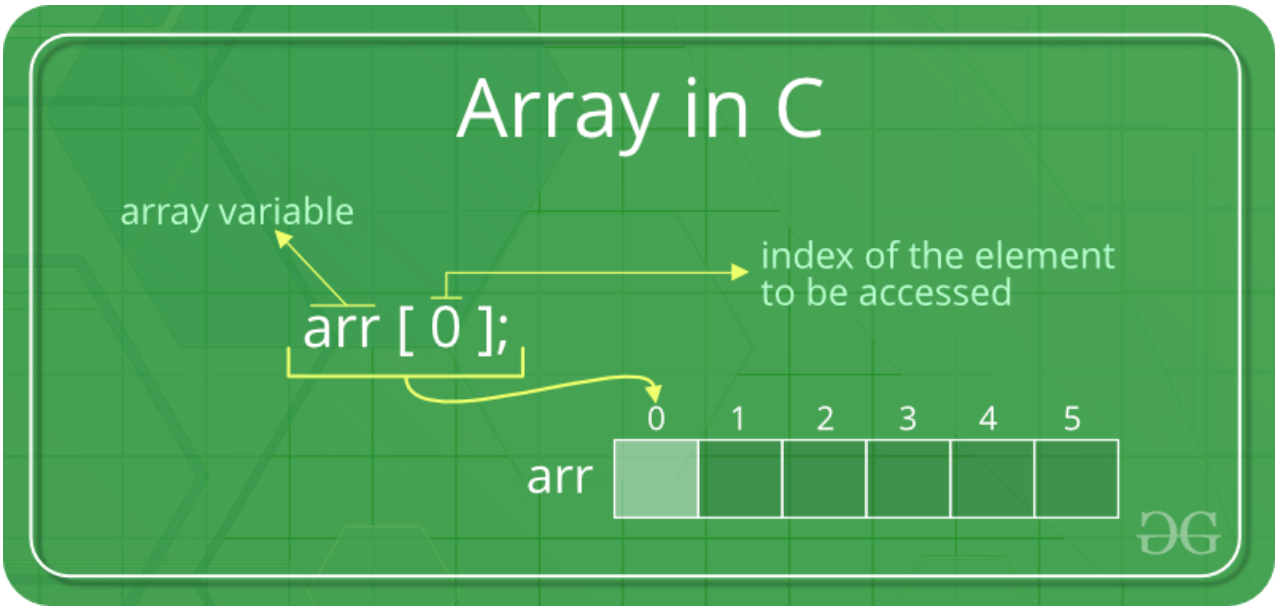


The above image can be looked at as a top-level view of a staircase where you are at the base of the staircase. Each element can be uniquely identified by its index in the array (in a similar way as you could identify your friends by the step on which they were on in the above example).

Array's size

In C language, the array has a fixed size meaning once the size is given to it, it cannot be changed i.e. you can't shrink it nor can you expand it. The reason was that for expanding if we change the size we can't be sure (it's not possible every time) that we get the next memory location to us for free. The shrinking will not work because the array, when declared, gets memory statically allocated, and thus compiler is the only one that can destroy it.
Types of indexing in an array:

- 0 (zero-based indexing): The first element of the array is indexed by a subscript of 0.
- 1 (one-based indexing): The first element of the array is indexed by the subscript of 1.
- n (N-based indexing): The base index of an array can be freely chosen. Usually, programming languages allowing n-based indexing also allow negative index values, and other scalar data types like enumerations, or characters may be used as an array index.



- C++
- C
- Java

```
#include <iostream>
using namespace std;
int main()
{
// Creating an integer array
// named arr of size 10.
int arr[10];
// accessing element at 0 index
// and setting its value to 5.
arr[0] = 5;
// access and print value at 0
// index we get the output as 5.
cout << arr[0];
return 0;
}
```

Output

5

Here the value 5 is printed because the first element has index zero and at the zeroth index, we already assigned the value 5.

Types of arrays :

1. One dimensional array (1-D arrays)
2. Multidimensional array

Advantages of using arrays:

- Arrays allow random access to elements. This makes accessing elements by position faster.
- Arrays have better cache locality which makes a pretty big difference in performance.
- Arrays represent multiple data items of the same type using a single name.

Disadvantages of using arrays:
You can't change the size i.e. once you have declared the array you can't change its size because of static memory allocation. Here Insertion(s) and deletion(s) are difficult as the elements are stored in consecutive memory locations and the shifting operation is costly too.
Now if take an example of the implementation of data structure Stack using array there are some obvious flaws.
Let's take the **POP** operation of the stack. The algorithm would go something like this.

1. Check for the stack underflow
2. Decrement the top by 1

What we are doing here is, that the pointer to the topmost element is decremented, which means we are just bounding our view, and actually that element stays there taking up the memory space. If you have an array (as a Stack) of any primitive data type then it might be ok. But in the case of an array of Objects, it would take a lot of memory.

Examples –

> // A character array in C/C++/Java
> char arr1[] = {'g', 'e', 'e', 'k', 's'};
>
> // An Integer array in C/C++/Java
> int arr2[] = {10, 20, 30, 40, 50};
>
> // Item at i'th index in array is typically accessed as "arr[i]".
> For example:
> arr1[0] gives us 'g'
> arr2[3] gives us 40

Usually, an array of characters is called a 'string', whereas an array of ints or floats is simply called an array.

Applications on Array

- Array stores data elements of the same data type.
- Arrays are used when the size of the data set is known.
- Used in solving matrix problems.
- Applied as a lookup table in computer.

- Databases records are also implemented by the array.
- Helps in implementing sorting algorithm.
- The different variables of the same type can be saved under one name.
- Arrays can be used for CPU scheduling.
- Used to Implement other data structures like Stacks, Queues, Heaps, Hash tables, etc.

Difference Between One-Dimensional and Two-Dimensional Array

One-Dimensional Vs. Two-Dimensional Array: Find the Difference Between One-Dimensional and Two-Dimensional Array

The term array refers to a collection of common name variables that have a similar type of data each. In this article, we will discuss the difference between one-dimensional and two-dimensional arrays. But let us first know their individual functionalities.

The one-dimensional array basically consists of a list of variables that have the very same data type. On the other hand, a two-dimensional array consists of a list of arrays- that have similar data types.

One can access any specified element in an array with the help of the index of that particular element in the array.

The arrays work very differently in Java as compared to that in C++. In the case of C++, it does not have any bound checking on the arrays. Java, on the other hand, has a strict bound checking on the arrays. There is more than one difference between a one-dimensional and two-dimensional array. They both vary in theways in which one can initialize, access, insert, traverse, delete, implement them. Let us get into them in the form of a comparison chart.

Parameters	One-Dimensional Array	Two-Dimensional Array
Basics	A one-dimensional array stores a single list of various elements having a similar data type.	A two-dimensional array stores an *array of various arrays,* or a *list of various lists*, or an *array of various one-dimensional arrays*.
Representation	It represents multiple data items in the form of a list.	It represents multiple data items in the form of a table that contains columns and rows.
Dimensions	It has only one dimension.	It has a total of two dimensions.
Parameters of Receiving	One can easily receive it in a pointer, an unsized array, or a sized array.	The parameters that receive it must define an array's rightmost dimension.

Total Size (in terms of Bytes)	Total number of Bytes = The size of array x the size of array variable or datatype.	Total number of Bytes = The size of array visible or datatype x the size of second index x the size of the first index.

Pointer to an Array | Array Pointer

geeksforgeeks.org/pointer-array-array-pointer

Pointer to Array

Consider the following program:

- C++
- C

```
#include <iostream>
using namespace std;
int main()
{
int arr[5] = { 1, 2, 3, 4, 5 };
int *ptr = arr;
cout << "\n" << ptr;
return 0;
}
// thus code is contributed by shivanisinghss2110
```

In this program, we have a pointer *ptr* that points to the 0^{th} element of the array. Similarly, we can also declare a pointer that can point to whole array instead of only one element of the array. This pointer is useful when talking about multidimensional arrays.
Syntax:

```
data_type (*var_name)[size_of_array];
```

Example:

```
int (*ptr)[10];
```

Here *ptr* is pointer that can point to an array of 10 integers. Since subscript have higher precedence than indirection, it is necessary to enclose the indirection operator and pointer name inside parentheses. Here the type of ptr is 'pointer to an array of 10 integers'.
Note : The pointer that points to the 0^{th} element of array and the pointer that points to the whole array are totally different. The following program shows this:

- C++
- C

```
// C++ program to understand difference between
// pointer to an integer and pointer to an
// array of integers.
#include <iostream>
using namespace std;
int main()
{
// Pointer to an integer
int *p;

// Pointer to an array of 5 integers
int (*ptr)[5];
int arr[5];

// Points to 0th element of the arr.
p = arr;

// Points to the whole array arr.
ptr = &arr;

cout << "p =" << p << ", ptr = " << ptr<< endl;
p++;
ptr++;
cout << "p =" << p << ", ptr = " << ptr<< endl;

return 0;
}
// This code is contributed by SHUBHAMSINGH10
```

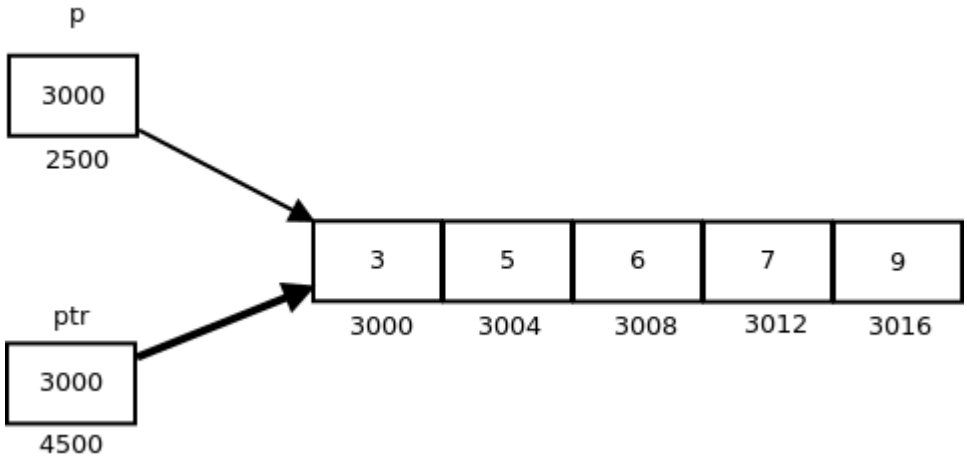
Output:

```
p = 0x7fff4f32fd50, ptr = 0x7fff4f32fd50
p = 0x7fff4f32fd54, ptr = 0x7fff4f32fd64
```

p: is pointer to 0^{th} element of the array *arr*, while ***ptr*** is a pointer that points to the whole array *arr*.

- The base type of *p* is int while base type of *ptr* is 'an array of 5 integers'.
- We know that the pointer arithmetic is performed relative to the base size, so if we write ptr++, then the pointer *ptr* will be shifted forward by 20 bytes.

The following figure shows the pointer p and ptr. Darker arrow denotes pointer to an array.



On dereferencing a pointer expression we get a value pointed to by that pointer expression. Pointer to an array points to an array, so on dereferencing it, we should get the array, and the name of array denotes the base address. So whenever a pointer to an array is dereferenced, we get the base address of the array to which it points.

- C++
- C

```
// C++ program to illustrate sizes of
// pointer of array
#include <bits/stdc++.h>
using namespace std;
int main()
{
int arr[] = { 3, 5, 6, 7, 9 };
int *p = arr;
int (*ptr)[5] = &arr;

cout << "p = " << p << ", ptr = " << ptr << endl;
cout << "*p = " << *p << ", *ptr = " << *ptr << endl;

cout << "sizeof(p) = " << sizeof (p) <<
", sizeof(*p) = " << sizeof (*p) << endl;
cout << "sizeof(ptr) = " << sizeof (ptr) <<
", sizeof(*ptr) = " << sizeof (*ptr) << endl;
return 0;
}
// This code is contributed by shubhamsingh10
```

Output:

```
p = 0x7ffde1ee5010, ptr = 0x7ffde1ee5010
*p = 3, *ptr = 0x7ffde1ee5010
sizeof(p) = 8, sizeof(*p) = 4
sizeof(ptr) = 8, sizeof(*ptr) = 20
```

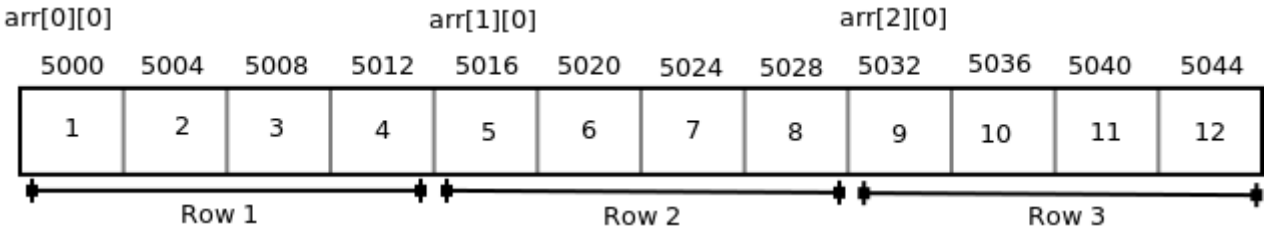
Pointer to Multidimensional Arrays:

> **Pointers and two dimensional Arrays:** In a two dimensional array, we can access each element by using two subscripts, where first subscript represents the row number and second subscript represents the column number. The elements of 2-D array can be accessed with the help of pointer notation also. Suppose arr is a 2-D array, we can access any element *arr[i][j]* of the array using the pointer expression ***(*(arr + i) + j)**. Now we'll see how this expression can be derived.
> Let us take a two dimensional array *arr[3][4]*:

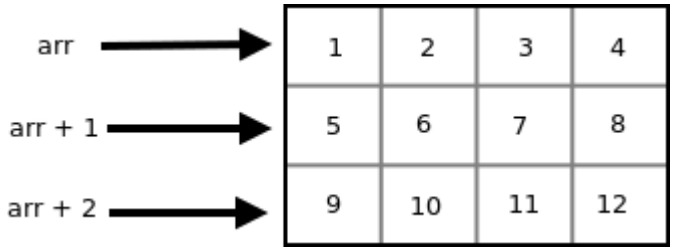
```
int arr[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };
```

	Col 1	Col 2	Col 3	Col 4
Row 1	1	2	3	4
Row 2	5	6	7	8
Row 3	9	10	11	12

Since memory in a computer is organized linearly it is not possible to store the 2-D array in rows and columns. The concept of rows and columns is only theoretical, actually, a 2-D array is stored in row-major order i.e rows are placed next to each other. The following figure shows how the above 2-D array will be stored in memory.



Each row can be considered as a 1-D array, so a two-dimensional array can be considered as a collection of one-dimensional arrays that are placed one after another. In other words, we can say that 2-D dimensional arrays that are placed one after another. So here *arr* is an array of 3 elements where each element is a 1-D array of 4 integers.
We know that the name of an array is a constant pointer that points to 0^{th} 1-D array and contains address 5000. Since *arr* is a 'pointer to an array of 4 integers', according to pointer arithmetic the expression arr + 1 will represent the address 5016 and expression arr + 2 will represent address 5032.
So we can say that *arr* points to the 0^{th} 1-D array, *arr + 1* points to the 1^{st} 1-D array and *arr + 2* points to the 2^{nd} 1-D array.



arr - Points to 0^{th} element of arr - Points to 0^{th} 1-D array - 5000
arr + 1 - Points to 1^{th} element of arr - Points to 1^{nd} 1-D array - 5016
arr + 2 - Points to 2^{th} element of arr - Points to 2^{nd} 1-D array - 5032

In general we can write:

arr + i Points to i^{th} element of arr -> Points to ith 1-D array

- Since arr + i points to i^{th} element of *arr*, on dereferencing it will get i^{th} element of *arr* which is of course a 1-D array. Thus the expression **(arr + i)* gives us the base address of i^{th} 1-D array.
- We know, the pointer expression **(arr + i)* is equivalent to the subscript expression *arr[i]*. So **(arr + i)* which is same as *arr[i]* gives us the base address of i^{th} 1-D array.
- To access an individual element of our 2-D array, we should be able to access any j^{th} element of i^{th} 1-D array.

- Since the base type of *(arr + i) is *int* and it contains the address of 0^{th} element of i^{th} 1-D array, we can get the addresses of subsequent elements in the i^{th} 1-D array by adding integer values to *(arr + i).
- For example *(arr + i) + 1 will represent the address of 1^{st} element of 1^{st}element of i^{th} 1-D array and *(arr+i)+2 will represent the address of 2^{nd} element of i^{th} 1-D array.
- Similarly *(arr + i) + j will represent the address of j^{th} element of i^{th} 1-D array. On dereferencing this expression we can get the j^{th} element of the i^{th} 1-D array.
- **Pointers and Three Dimensional Arrays**
  In a three dimensional array we can access each element by using three subscripts. Let us take a 3-D array-

```
int arr[2][3][2] = { {{5, 10}, {6, 11}, {7, 12}}, {{20, 30}, {21, 31}, {22, 32}} };
```

We can consider a three dimensional array to be an array of 2-D array i.e each element of a 3-D array is considered to be a 2-D array. The 3-D array *arr* can be considered as an array consisting of two elements where each element is a 2-D array. The name of the array *arr* is a pointer to the 0^{th} 2-D array.

arr	**Points to 0^{th} 2-D array.**
arr + i	**Points to i^{th} 2-D array.**
***(arr + i)**	**Gives base address of i^{th} 2-D array, so points to 0^{th} element of i^{th} 2-D array, each element of 2-D array is a 1-D array, so it points to 0^{th} 1-D array of i^{th} 2-D array.**
***(arr + i) + j**	**Points to j^{th} 1-D array of i^{th} 2-D array.**
***(*(arr + i) + j)**	**Gives base address of j^{th} 1-D array of i^{th} 2-D array so it points to 0^{th} element of j^{th} 1-D array of i^{th} 2-D array.**
***(*(arr + i) + j) + k**	**Reprents the value of j^{th} element of i^{th} 1-D array.**
***(*(arr + i) + j) + k)**	**Gives the value of k^{th} element of j^{th} 1-D array of i^{th} 2-D array.**

Thus the pointer expression ***(*(*(arr + i) + j) + k) is equivalent to the subscript expression arr[i][j][k].**
We know the expression *(arr + i) is equivalent to arr[i] and the expression *(*(arr + i) + j) is equivalent arr[i][j]. So we can say that arr[i] represents the base address of i^{th} 2-D array and arr[i][j] represents the base address of the j^{th} 1-D array.

- C++
- C

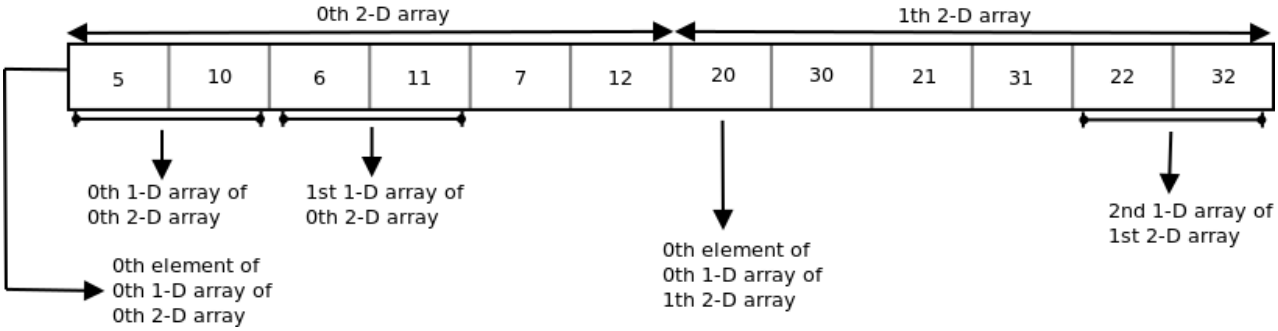
```
// C++ program to print the elements of 3-D
// array using pointer notation
#include <iostream>
using namespace std;
int main()
{
int arr[2][3][2] = {
{
```

```
{5, 10},
{6, 11},
{7, 12},
},
{
{20, 30},
{21, 31},
{22, 32},
}
};
int i, j, k;
for (i = 0; i < 2; i++)
{
for (j = 0; j < 3; j++)
{
for (k = 0; k < 2; k++)
cout << *(*(*(arr + i) + j) +k) << "\t" ;
cout << "\n" ;
}
}
return 0;
}
// this code is contributed by shivanisinghss2110
```

Output:

```
5    10
6    11
7    12
20    30
21    31
22    32
```

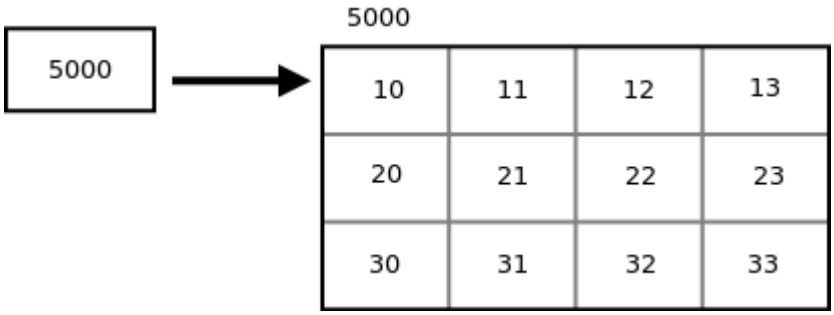
The following figure shows how the 3-D array used in the above program is stored in memory.



Subscripting Pointer to an Array

Suppose *arr* is a 2-D array with 3 rows and 4 columns and *ptr* is a pointer to an array of 4 integers, and *ptr* contains the base address of array *arr*.

```
int arr[3][4] = {{10, 11, 12, 13}, {20, 21, 22, 23}, {30, 31, 32, 33}};
int (*ptr)[4];
ptr = arr;
```



Since *ptr* is a pointer to an array of 4 integers, *ptr + i* will point to i^{th} row. On dereferencing *ptr + i*, we get base address of i^{th} row. To access the address of j^{th} element of i^{th} row we can add j to the pointer expression **(ptr + i)*. So the pointer expression **(ptr + i) + j* gives the address of j^{th} element of i^{th} row and the pointer expression **(*(ptr + i)+j)* gives the value of the j^{th} element of i^{th} row.
We know that the pointer expression *(*(ptr + i) + j) is equivalent to subscript expression ptr[i][j]. So if we have a pointer variable containing the base address of 2-D array, then we can access the elements of array by double subscripting that pointer variable.

- C++
- C

```
// C++ program to print elements of a 2-D array
// by scripting a pointer to an array
#include <iostream>
using namespace std;
int main()
{
int arr[3][4] = {
{10, 11, 12, 13},
{20, 21, 22, 23},
{30, 31, 32, 33}
};
int (*ptr)[4];
ptr = arr;
cout << ptr<< " " << ptr + 1<< " " << ptr + 2 << endl;
cout << *ptr<< " " << *(ptr + 1)<< " " << *(ptr + 2)<< endl;
cout << **ptr<< " " << *(*(ptr + 1) + 2)<< " " << *(*(ptr + 2) + 3)
<< endl;
cout << ptr[0][0]<< " " << ptr[1][2]<< " " << ptr[2][3]<< endl;
return 0;
}
// This code is contributed by shivanisinghss2110
```

Structure Pointer in C

In this section, we will discuss the Structure pointer in the C programming language. Before going to the concepts, let's understand the Structure. The **structure** is the collection of different data types grouped under the same name using the **struct** keyword. It is also known as the user-defined data type that enables the programmer to store different data type records in the Structure. Furthermore, the collection of data elements inside the Structure is termed as the **member**.

For example, suppose we want to create the records of a person containing name, age, id, city, etc., and these records cannot be grouped in the single dimension array. Therefore, we use the Structure to store multiple collections of data items.

Syntax to define Structure

```
1. struct structure_name
2. {
3. char name[40];
4. int roll_no;
5. float percentage;
6. char sub[30];
7. };
8. struct structure_name  str;  // str is a variable of the structure_name
```

Here **structure_name** is the name of the Structure that is defined using the **struct** keyword. Inside the structure_name, it collects different data types (int, char, float) elements known as the **member**. And the last, **str,** is a variable of the Structure.

Program to demonstrate the structure and access their member

In this program creates a student structure and access its member using structure variable and dot (.) operator.

struct.c

```
1. #include <stdio.h>
2. #include <string.h>
3. // create the Structure of student to store multiples items
4. struct student
5. {
6.     char name[ 30];
7.     int roll_no;
```

```
    char state[100];
    int age;
};
struct student s1, s2; // declare s1 and s2 variables of student structure
int main()
{
    // records of the student s1
    strcpy (s1.name, "John");
    s1.roll_no = 1101;
    strcpy (s1.state, "Los Angeles");
    s1.age = 20;
    // records of the student s2
    strcpy (s2.name, " Mark Douglas");
    s2.roll_no = 111;
    strcpy (s2.state, "California");
    s2.age = 18;
    // print the details of the student s1;
    printf (" Name of the student s1 is: %s\t ", s1.name);
    printf (" \n Roll No. of the student s1 is: %d\t ", s1.roll_no);
    printf (" \n The state of the student s1 is: %s\t ", s1.state);
    printf (" \n Age of the student s1 is: %d\t ", s1.age);
        // print the details of the student s2;
    printf ("\n Name of the student s1 is: %s\t ", s2.name);
    printf (" \n Roll No. of the student s1 is: %d\t ", s2.roll_no);
    printf (" \n The state of the student s1 is: %s\t ", s2.state);
    printf (" \n Age of the student s1 is: %d\t ", s2.age);
    return 0;
}
```

Output:

```
Name of the student s1 is: John
Roll No. of the student s1 is: 1101
 state of the student s1 is: Los Angeles
Age of the student s1 is: 20
Name of the student s1 is:  Mark Douglas
Roll No. of the student s1 is: 111
 The state of the student s1 is: California
Age of the student s1 is: 18
```

Explanation of the program: As we can see in the above program, we have created a structure with name **student,** and the student structure contains different members such as name (char), roll_no (int), state (char), age (int). The student structure also defines two variables like **s1** and **s2**, that access the structure members using dot operator inside the main() function.

Structure Pointer

The **structure pointer** points to the address of a memory block where the Structure is being stored. Like a pointer that tells the address of another variable of any data type (int, char, float) in memory. And here, we use a structure pointer which tells the address of a structure in memory by pointing pointer variable **ptr** to the structure variable.

Declare a Structure Pointer

The declaration of a structure pointer is similar to the declaration of the structure variable. So, we can declare the structure pointer and variable inside and outside of the main() function. To declare a pointer variable in C, we use the asterisk (*) symbol

before the variable's name.

```
1. struct structure_name *ptr;
```

After defining the structure pointer, we need to initialize it, as the code is shown:

Initialization of the Structure Pointer

```
1. ptr = &structure_variable;
```

We can also initialize a Structure Pointer directly during the declaration of a pointer.

```
1. struct structure_name *ptr = &structure_variable;
```

As we can see, a pointer **ptr** is pointing to the address structure_variable of the Structure.

Access Structure member using pointer:

There are two ways to access the member of the structure using Structure pointer:

1. Using (*) asterisk or indirection operator and dot (.) operator.
2. Using arrow (->) operator or membership operator.

Program to access the structure member using structure pointer and the dot operator

Let's consider an example to create a Subject structure and access its members using a structure pointer that points to the address of the Subject variable in C.

Program to access the structure member using structure pointer and arrow (->) operator

Let's consider a program to access the structure members using the pointer and arrow (->) operator in C.

Pointer2.c

```
1. #include <stdio.h>
2. // create Employee structure
3. struct Employee
4. {
5.     // define the member of the structure
6.     char name[30];
7.     int id;
8.     int age;
9.     char gender[30];
10.    char city[40];
11. };
12. // define the variables of the Structure with pointers
13. struct Employee emp1, emp2, *ptr1, *ptr2;
14. int main()
15. {
16.    // store the address of the emp1 and emp2 structure variable
17.    ptr1 = &emp1;
18.    ptr2 = &emp2;
19.    printf (" Enter the name of the Employee (emp1): ");
20.    scanf (" %s", &ptr1->name);
```

```
21.    printf (" Enter the id of the Employee (emp1): ");
22.    scanf (" %d", &ptr1->id);
23.    printf (" Enter the age of the Employee (emp1): ");
24.    scanf (" %d", &ptr1->age);
25.    printf (" Enter the gender of the Employee (emp1): ");
26.    scanf (" %s", &ptr1->gender);
27.    printf (" Enter the city of the Employee (emp1): ");
28.    scanf (" %s", &ptr1->city);
29.    printf (" \n Second Employee: \n");
30.          printf (" Enter the name of the Employee (emp2): ");
31.    scanf (" %s", &ptr2->name);
32.    printf (" Enter the id of the Employee (emp2): ");
33.    scanf (" %d", &ptr2->id);
34.    printf (" Enter the age of the Employee (emp2): ");
35.    scanf (" %d", &ptr2->age);
36.    printf (" Enter the gender of the Employee (emp2): ");
37.    scanf (" %s", &ptr2->gender);
38.    printf (" Enter the city of the Employee (emp2): ");
39.    scanf (" %s", &ptr2->city);
40.    printf ("\n Display the Details of the Employee using Structure Pointer");
41.    printf ("\n Details of the Employee (emp1) \n");
42.    printf(" Name: %s\n", ptr1->name);
43.    printf(" Id: %d\n", ptr1->id);
44.    printf(" Age: %d\n", ptr1->age);
45.    printf(" Gender: %s\n", ptr1->gender);
46.    printf(" City: %s\n", ptr1->city);
47.    printf ("\n Details of the Employee (emp2) \n");
48.    printf(" Name: %s\n", ptr2->name);
49.    printf(" Id: %d\n", ptr2->id);
50.    printf(" Age: %d\n", ptr2->age);
51.    printf(" Gender: %s\n", ptr2->gender);
52.    printf(" City: %s\n", ptr2->city);
53.    return 0;
54. }
```

Output:

```
Enter the name of the Employee (emp1): John
 Enter the id of the Employee (emp1): 1099
 Enter the age of the Employee (emp1): 28
 Enter the gender of the Employee (emp1): Male
 Enter the city of the Employee (emp1): California

 Second Employee:
 Enter the name of the Employee (emp2): Maria
 Enter the id of the Employee (emp2): 1109
 Enter the age of the Employee (emp2): 23
 Enter the gender of the Employee (emp2): Female
 Enter the city of the Employee (emp2): Los Angeles

 Display the Details of the Employee using Structure Pointer
 Details of the Employee (emp1)
 Name: John
 Id: 1099
 Age: 28
 Gender: Male
 City: California

 Details of the Employee (emp2) Name: Maria
 Id: 1109
 Age: 23
 Gender: Female
 City: Los Angeles
```
