**Why array index starts from zero ?**

Consider int arr[100]. The answer lies in the fact how the compiler interprets arr[i] ( 0<=i<100).

arr[i] is interpreted as \*(arr + i). Now, arr is the address of the array or address of 0th index element of the array. So, address of next element in the array is arr + 1 (because elements in the array are stored in consecutive memory locations), further address of next location is arr + 2 and so on. Going with the above arguments, arr + i mean the address at i distance away from the starting element of the array. Therefore, going by this definition, i will be zero for the starting element of the array because the starting element is at 0 distance away from the starting element of the array. To fit this definition of arr[i], indexing of array starts from 0.

**Intro to C++ Compilers**

In order to get started with C++, you will need to learn a little about compilers and how C++ runs on your computer.

When all is said and done, computers only understand one language, machine language. Machine language is entirely made up of binary bits, or 0s and 1s.

While it would be possible to program in binary, it would be incredibly tedious and time consuming. So, we humans developed programming languages to make it easier to develop software.

Assembly language is a direct 1 to 1 with machine language. Languages like C, C++, and COBOL are a little higher and need to be compiled down. It goes even higher. Languages like JavaScript and Python have components that get translated into C++ or other low level languages before they get compiled, effectively making them “higher” languages than C or C++.

Because computer architecture is made up of electronic switches and cables that can only work with binary 1s and 0s, you need a compiler to translate your code from high level C++ to machine language that the CPU can understand.

**How compilers work**

Compilers are utility programs that take your code and transform it into executable machine code files.

When you run a compiler on your code, first, the preprocessor reads the source code (the C++ file you just wrote). The preprocessor searches for any preprocessor directives (lines of code starting with a #). Preprocessor directives cause the preprocessor to change your code in some way (by usually adding some library or another C++ file).

Next, the compiler works through the preprocessed code line by line translating each line into the appropriate machine language instruction. This will also uncover any syntax errors that are present in your source code and will throw an error to the command line.

Finally, if no errors are present, the compiler creates an object file with the machine language binary necessary to run on your machine. While the object file that the compiler just created is likely enough to do something on your computer, it still isn’t a working executable of your C++ program. There is a final important step to reach an executable program.

C++ contains a vast library to aid in performing difficult tasks like I/O and hardware manipulation. You can include these libraries with preprocessor directives, but the preprocessor doesn’t automatically add them to your code.

In order for you to have a final executable program, another utility known as the linker must combine your object files with the library functions necessary to run the code.

Think of it as having all the necessary blocks to build a house. The compiler made all the blocks but the linker is the one that sticks them all together to finally create a house. Once this is done, you now have a functioning executable file!

VISUALIZE THE PROCES OF COMPILATION TO EXECUTIONBY CPU : <https://youtu.be/ZTu0kf-7h08>

1)sourceFiles - > Preprocessor(.ii file, removes comments, header files, spaces) -> compilation(.sfiles, compiles and converts into assembly level language giving instructions for storing them, in which variety of resistors In CPU ARM Cortex micro-controller) -> assembly (.obj file, converts to binaly low level language) -> linker (links all obj files, headers, standard library & external library) -> LOADER -> loads the executable code in a memory RAM, and from there it is sent to CPU for output.

LINKERS AND CODES ADVANCED CONCEPT: <https://youtu.be/H4s55GgAg0I>

Let's start with some math facts:

* For a positive n, aⁿ = a⨯a⨯…⨯a n times
* For a negative n, aⁿ = ⅟a⁻ⁿ = ⅟(a⨯a⨯…⨯a). This means *a* cannot be zero.
* For n = 0, aⁿ = 1, even if *a* is zero or negative.

So let's start from the positive n case, and work from there.

Since we want our solution to be recursive, we have to find a way to define aⁿ based on a smaller n, and work from there. The usual way people think of recursion is to try to find a solution for n-1, and work from there.

And indeed, since it's mathematically true that aⁿ = a⨯(aⁿ⁻¹), the naive approach would be very similar to what you created:

public static int pow( int a, int n) {

if ( n == 0 ) {

return 1;

}

return ( a \* pow(a,n-1));

}

However, the complexity of this is O(n). Why? Because For n=0 it doesn't do any multiplications. For n=1, it does one multiplication. For n=2, it calls pow(a,1) which we know is one multiplication, and multiplies it once, so we have two multiplications. There is one multiplication in every recursion step, and there are n steps. So It's O(n).

In order to make this O(log n), we need every step to be applied to a **fraction** of n rather than just n-1. Here again, there is a math fact that can help us: an₁+n₂ = an₁⨯an₂.

This means that we can calculate aⁿ as an/2⨯an/2.

But what happens if n is odd? something like a⁹ will be a4.5⨯a4.5. But we are talking about integer powers here. Handling fractions is a whole different thing. Luckily, we can just formulate that as a⨯a⁴⨯a⁴.

So, for an even number use an/2⨯an/2, and for an odd number, use a⨯ an/2⨯an/2 (integer division, giving us 9/2 = 4).

public static int pow( int a, int n) {

if ( n == 0 ) {

return 1;

}

if ( n % 2 == 1 ) {

// Odd n

return a \* pow( a, n/2 ) \* pow(a, n/2 );

} else {

// Even n

return pow( a, n/2 ) \* pow( a, n/2 );

}

}

# What’s difference between “array” and “&array” for “int array[5]” ?

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 29 May, 2017

If someone has defined an array such as “int array[5]”, what’s the meaning of “array” or “&array”? Are they both same or are they different? You might be tempted to think that they both would point to the very first element of the array i.e. they both will have same address. Let us find out!

To check this, the very first thing that comes to mind is the following program.

|  |
| --- |
| #include "stdio.h"  int main()  {     int array[5];       /\* If %p is new to you, you can use %d as well \*/     printf("array=%p : &array=%p\n", array, &array);       return 0;  } |

So you got same address for both “array” and “&array”. Again, you are tempted to think that both are same. Well, they aren’t not! How come a variable and its & (i.e. address-of) be same. It doesn’t look logical but we saw that both “array” and “&array” are printing same address. May be it’s too soon to conclude. The crux of this post is that even though they both are resulting in same address but they are different types of addresses. And this is the difference between “array” and “&array”.

And just to show this difference, I would suggest to take a look at the following program.

|  |
| --- |
| #include "stdio.h"  int main()  {     int array[5];       /\* If %p is new to you, you can use %d as well \*/     printf("array=%p : &array=%p\n", array, &array);       printf("array+1 = %p : &array + 1 = %p", array+1, &array+1);       return 0;  } |

array=0x7ffeca8b59d0 : &array=0x7ffeca8b59d0

array+1 = 0x7ffeca8b59d4 : &array + 1 = 0x7ffeca8b59e4

With pointer arithmetic, we know what happens when we add an integer to a pointer. So can you guess the output of the above program without running it? Shouldn’t “array+1” and “&array+1” point to same address. Well you might be surprised

Basically, “array” is a “**pointer to the first element of array**” but “&array” is a “**pointer to whole array of 5 int**”. Since “array” is pointer to int, addition of 1 resulted in an address with increment of 4 (assuming int size in your machine is 4 bytes). Since “&array” is pointer to array of 5 ints, addition of 1 resulted in an address with increment of 4 x 5 = 20 = 0x14. Now you see why these two seemingly similar pointers are different at core level. This logic can be extended to multidimensional arrays as well. Suppose double twoDarray[5][4] is a 2D array. Here, “twoDarray” is a pointer to array of 4 int but “&twoDarray” is pointer to array of 5 rows arrays of 4 int”. If this sounds cryptic, you can always have a small program to print these after adding 1. We hope that we could clarify that any array name itself is a pointer to the first element but & (i.e. add

# **C++ Array of Pointers**

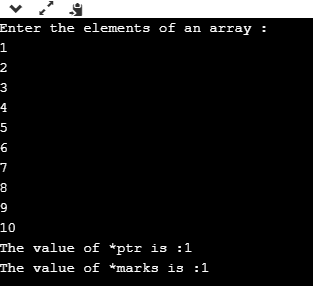
Array and pointers are closely related to each other. In C++, the name of an array is considered às a pointer, i.e., the name of an array contains the address of an element. C++ considers the array name as the address of the first element. For example, if we create an array, i.e., marks which hold the 20 values of integer type, then marks will contain the address of first element, i.e., marks[0]. Therefore, we can say that array name (marks) is a pointer which is holding the address of the first element of an array.

**Let's understand this scenario through an example.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** \*ptr;  // integer pointer declaration
6. **int** marks[10]; // marks array declaration
7. std::cout << "Enter the elements of an array :" << std::endl;
8. **for**(**int** i=0;i<10;i++)
9. {
10. cin>>marks[i];
11. }
12. ptr=marks; // both marks and ptr pointing to the same element..
13. std::cout << "The value of \*ptr is :" <<\*ptr<< std::endl;
14. std::cout << "The value of \*marks is :" <<\*marks<<std::endl;
15. }

In the above code, we declare an integer pointer and an array of integer type. We assign the address of marks to the ptr by using the statement ptr=marks; it means that both the variables 'marks' and 'ptr' point to the same element, i.e., marks[0]. When we try to print the values of \*ptr and \*marks, then it comes out to be same. Hence, it is proved that the array name stores the address of the first element of an array.

**Output**



### **Array of Pointers**

An array of pointers is an array that consists of variables of pointer type, which means that the variable is a pointer addressing to some other element. Suppose we create an array of pointer holding 5 integer pointers; then its declaration would look like:

1. **int** \*ptr[5];         // array of 5 integer pointer.

In the above declaration, we declare an array of pointer named as ptr, and it allocates 5 integer pointers in memory.

The element of an array of a pointer can also be initialized by assigning the address of some other element. Let's observe this case through an example.

1. **int** a; // variable declaration.
2. ptr[2] = &a;

In the above code, we are assigning the address of 'a' variable to the third element of an array 'ptr'.

We can also retrieve the value of 'a' be dereferencing the pointer.

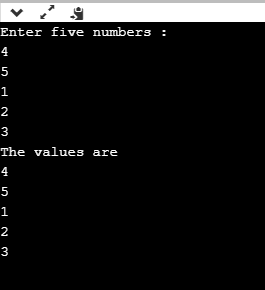
1. \*ptr[2];

**Let's understand through an example.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** ptr1[5]; // integer array declaration
6. **int** \*ptr2[5]; // integer array of pointer declaration
7. std::cout << "Enter five numbers :" << std::endl;
8. **for**(**int** i=0;i<5;i++)
9. {
10. std::cin >> ptr1[i];
11. }
12. **for**(**int** i=0;i<5;i++)
13. {
14. ptr2[i]=&ptr1[i];
15. }
16. // printing the values of ptr1 array
17. std::cout << "The values are" << std::endl;
18. **for**(**int** i=0;i<5;i++)
19. {
20. std::cout << \*ptr2[i] << std::endl;
21. }
22. }

In the above code, we declare an array of integer type and an array of integer pointers. We have defined the 'for' loop, which iterates through the elements of an array 'ptr1', and on each iteration, the address of element of ptr1 at index 'i' gets stored in the ptr2 at index 'i'.

**Output**



Till now, we have learnt the array of pointers to an integer. Now, we will see how to create the array of pointers to strings.

### **Array of Pointer to Strings**

An array of pointer to strings is an array of character pointers that holds the address of the first character of a string or we can say the base address of a string.

The following are the differences between an array of pointers to string and two-dimensional array of characters:

* An array of pointers to string is more efficient than the two-dimensional array of characters in case of memory consumption because an array of pointer to strings consumes less memory than the two-dimensional array of characters to store the strings.
* In an array of pointers, the manipulation of strings is comparatively easier than in the case of 2d array. We can also easily change the position of the strings by using the pointers.

Let's see how to declare the array of pointers to string.

First, we declare the array of pointer to string:

1. **char** \*names[5] = {"john",
2. "Peter",
3. "Marco",
4. "Devin",
5. "Ronan"};

In the above code, we declared an array of pointer names as 'names' of size 5. In the above case, we have done the initialization at the time of declaration, so we do not need to mention the size of the array of a pointer. The above code can be re-written as:

1. **char** \*names[ ] = {"john",
2. "Peter",
3. "Marco",
4. "Devin",
5. "Ronan"};

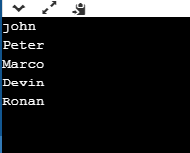
In the above case, each element of the 'names' array is a string literal, and each string literal would hold the base address of the first character of a string. For example, names[0] contains the base address of "john", names[1] contains the base address of "Peter", and so on. It is not guaranteed that all the string literals will be stored in the contiguous memory location, but the characters of a string literal are stored in a contiguous memory location.

**Let's create a simple example.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **char** \*names[5] = {"john",
6. "Peter",
7. "Marco",
8. "Devin",
9. "Ronan"};
10. **for**(**int** i=0;i<5;i++)
11. {
12. std::cout << names[i] << std::endl;
13. }
14. **return** 0;
15. }

In the above code, we have declared an array of char pointer holding 5 string literals, and the first character of each string is holding the base address of the string.

**Output**



**USING POINTER CAUSES SECURITY PROBLEMS C++:**

Actually you an do many things through pointer, which is actually not permitted or safe.

Like i take an example. What if i say that you can actually print the private variable through the pointer without any member function. Here is the sample code.

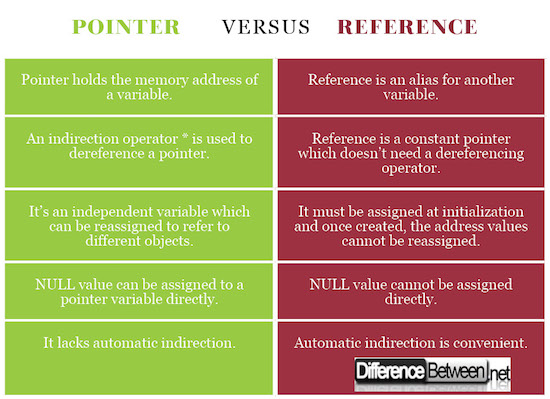
1. #include <iostream>
2. #include <string>
3. using namespace std;
5. class Example
6. {
7. private:
8. int a=10; //private variable
9. public:
10. int b=20; //public variable
11. };
13. int main()
14. {
15. Example exe;
16. int \*ptr;
17. ptr=&exe.b; //ptr that stores the address of b
18. --ptr; //decrease the value by 1
19. cout<<"The output:"<< \*(ptr) <<endl;
20. return 0;
21. }

OUTPUT:

1. The output:10

let me explain what actually the catch here, As we know that the memory allocation for the class is in sequential manner. So in line 17, we stores the address of public variable , then we decrement the address , which is address of a, i.e. a private variable of the class. So, at the line 19, we will actually able to print the value of private variable.

So as you can say that it is the violation of Data abstraction. As like this we can do many more things, which we can use pointer for many disallowed things, which is also a type of hacking.



**ENCAPSULATION:**

**Realtime Example 2:**  
When you log into your email accounts such as Gmail, Yahoo Mail, or Rediff mail, there is a lot of internal processes taking place in the backend and you have no control over it.

When you enter the password for logging, they are retrieved in an encrypted form and verified, and then you are given access to your account.

You do not have control over it that how the password has been verified. Thus, it keeps our account safe from being misused.

**Realtime Example 3:**  
Suppose you have an account in the bank. If your balance variable is declared as a public variable in the bank software, your account balance will be known as public, In this case, anyone can know your account balance. So, would you like it? Obviously No.

So, they declare balance variable as private for making your account safe, so that anyone cannot see your account balance.

The person who has to see his account balance, will have to access only private members through methods defined inside that class and this method will ask your account holder name or user Id, and password for authentication.

Thus, we can achieve security by utilizing the concept of data hiding. This is called Encapsulation in Java.

**LINKLIST DELETION:**

**5 4 2 1 7 9 => 5-4 is 1 hop, 4-2 is 1 hop + 2-1 is 1 hop = Total 3 hop, and we will reach the parent of the node to be deleted, here k=2 from the end, which is to be deleted., formula to reach the parent of the node to be deleted, is n-k-1, where n is length of link list, k is the node to be deleted.**

**so we first run a loop, upto last node, and calculate the parent node by n-k-1, then we wil use another loop to reach the parent of deleted node and then delete the next node.**

**int parentIndex;**

**while(head!=null){**

**//count the length of linklst**

**}**

**parentIndex = count - k -1;**

**for(int i=1;i<=parentIndex ;i++){**

**if(i==parentIndex)**

**{**

**parenIndex->next = parentIndex->next->next;**

**so when we reach parentNode, then the next of parent Node holds the address of node to be deleted, so in the next of parent, we store the next of next Node, i.e.,the next node's address hold by the node to be deleted, will be now stored in parent->next.**

**}**