6.8 — Pointers and arrays

BY ALEX ON JULY 11TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

Pointers and arrays are intrinsically related in C++.

Array decay

In lesson **6.1 -- Arrays (part i)**, you learned how to define a fixed array:

```
int array[5] = { 9, 7, 5, 3, 1 }; // declare a fixed array of 5 integers
```

To us, the above is an array of 5 integers, but to the compiler, array is a variable of type int[5]. We know what the values of array[0], array[1], array[2], array[3], and array[4] are (9, 7, 5, 3, and 1 respectively).

In all but two cases (which we'll cover below), when a fixed array is used in an expression, the fixed array will **decay** (be implicitly converted) into a pointer that points to the first element of the array. You can see this in the following program:

```
#include <iostream>

int main()
{
    int array[5] = { 9, 7, 5, 3, 1 };

    // print address of the array's first element
    std::cout << "Element 0 has address: " << &array[0] << '\n';

    // print the value of the pointer the array decays to
    std::cout << "The array decays to a pointer holding address: " << array << '\n';


    return 0;
}
```

On the author's machine, this printed:

```
Element 0 has address: 0042FD5C
The array decays to a pointer holding address: 0042FD5C
```

It's a common fallacy in C++ to believe an array and a pointer to the array are identical. They're not. In the above case, array is of type "int[5]", and it's "value" is the array elements themselves. A pointer to the array would be of type "int *", and its value would be the address of the first element of the array.

We'll see where this makes a difference shortly.

All elements of the array can still be accessed through the pointer (we'll see how this works in the next lesson), but information derived from the array's type (such as how long the array is) can not be accessed from the pointer.

However, this also effectively allows us to treat fixed arrays and pointers identically in most cases.

For example, we can dereference the array to get the value of the first element:

```
int array[5] = { 9, 7, 5, 3, 1 };

// dereferencing an array returns the first element (element 0)
cout << *array; // will print 9!

```

```
char name[] = "Jason"; // C-style string (also an array)
cout << *name; // will print 'J'
```

Note that we're not *actually* dereferencing the array itself. The array (of type int[5]) gets implicitly converted into a pointer (of type int *), and we dereference the pointer to get the value at the memory address the pointer is holding (the value of the first element of the array).

We can also assign a pointer to point at the array:

```
#include <iostream>

int main()
{
    int array[5] = { 9, 7, 5, 3, 1 };
        std::cout << *array; // will print 9

        int *ptr = array;
        std::cout << *ptr; // will print 9

    return 0;
}
```

This works because the array decays into a pointer of type int *, and our pointer (also of type int *) has the same type.

Differences between pointers and fixed arrays

There are a few cases where the difference in typing between fixed arrays and pointers makes a difference. These help illustrate that a fixed array and a pointer are not the same.

The primary difference occurs when using the sizeof() operator. When used on a fixed array, sizeof returns the size of the entire array (array length * element size). When used on a pointer, sizeof returns the size of a memory address (in bytes). The following program illustrates this:

```
#include <iostream>

int main()
{
    int array[5] = { 9, 7, 5, 3, 1 };

    std::cout << sizeof(array) << '\n'; // will print sizeof(int) * array length

        int *ptr = array;
        std::cout << sizeof(ptr) << '\n'; // will print the size of a pointer

    return 0;
}
```

This program prints:

```
20
4
```

A fixed array knows how long the array it is pointing to is. A pointer to the array does not.

The second difference occurs when using the address-of operator (&). Taking the address of a pointer yields the memory address of the pointer variable. Taking the address of the array returns a pointer to the entire array. This pointer also points to the first element of the array, but the type information is different (in the above example, int(*)[5]). It's unlikely you'll ever need to use this.

Revisiting passing fixed arrays to functions

Back in lesson **6.2 -- Arrays (part ii)**, we mentioned that because copying large arrays can be very expensive, C++ does not copy an array when an array is passed into a function. When passing an array as an argument to a function, a fixed array decays into a pointer, and the pointer is passed to the function:

```
#include <iostream>

void printSize(int *array)
{
    // array is treated as a pointer here
    std::cout << sizeof(array) << '\n'; // prints the size of a pointer, not the size of the a
}

int main()
{
    int array[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
    std::cout << sizeof(array) << '\n'; // will print sizeof(int) * array length

    printSize(array); // the array argument decays into a pointer here

     return 0;
}
```

This prints:

```
32
4
```

Note that this happens even if the parameter is declared as a fixed array:

```
#include <iostream>

// C++ will implicitly convert parameter array[] to *array
void printSize(int array[])
{
    // array is treated as a pointer here, not a fixed array
    std::cout << sizeof(array) << '\n'; // prints the size of a pointer, not the size of the a
}

int main()
{
    int array[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
    std::cout << sizeof(array) << '\n'; // will print sizeof(int) * array length

    printSize(array); // the array argument decays into a pointer here

     return 0;
}
```

This prints:

```
32
4
```

In the above example, C++ implicitly converts parameters using the array syntax ([]) to the pointer syntax (*). That means the following two function declarations are identical:

```
void printSize(int array[]);
void printSize(int *array);
```

Some programmers prefer using the [] syntax because it makes it clear that the function is expecting an array, not just a pointer to a value. However, in most cases, because the pointer doesn't know how large the array is, you'll need to pass in the array size as a separate parameter anyway (strings being an exception because they're null terminated).

We lightly recommend using the pointer syntax, because it makes it clear that the parameter is being treated as a pointer, not a fixed array, and that certain operations, such as sizeof(), will operate as if the parameter is a pointer.

Recommendation: Favor the pointer syntax (*) over the array syntax ([]) for array function parameters.

An intro to pass by address

The fact that arrays decay into pointers when passed to a function explains the underlying reason why changing an array in a function changes the actual array argument passed in. Consider the following example:

```
#include <iostream>

// parameter ptr contains a copy of the array's address
void changeArray(int *ptr)
{
    *ptr = 5; // so changing an array element changes the _actual_ array
}

int main()
{
    int array[] = { 1, 1, 2, 3, 5, 8, 13, 21 };
    std::cout << "Element 0 has value: " << array[0] << '\n';

    changeArray(array);

    std::cout << "Element 0 has value: " << array[0] << '\n';

     return 0;
}
```

```
Element 0 has value: 1
Element 0 has value: 5
```

When changeArray() is called, array decays into a pointer, and the value of that pointer (the memory address of the first element of the array) is copied into the ptr parameter of function changeArray(). Although the value in ptr is a copy of the address of the array, ptr still points at the actual array (not a copy!). Consequently, when ptr is dereferenced, the actual array is dereferenced!

Astute readers will note this phenomena works with pointers to non-array values as well. We'll cover this topic (called passing by address) in more detail in the next chapter.

Arrays in structs and classes don't decay

Finally, it is worth noting that arrays that are part of structs or classes do not decay when the whole struct or class is passed to a function. This yields a useful way to prevent decay if desired, and will be valuable later when we write classes that utilize arrays.

In the next lesson, we'll take a look at pointer arithmetic, and talk about how array indexing actually works.

6.8a -- Pointer arithmetic and array indexing

Index

6.7a -- Null pointers

C++ TUTORIAL | PRINT THIS POST

197 comments to 6.8 — Pointers and arrays

« Older Comments 1 2 3

emijee
February 1, 2020 at 8:41 am · Reply

Write declarations for the following entities and initialize each of them:
a pointer to char an array with 10 int
a reference to an array with 10 int
a pointer to an array with 10 elements of type string
a pointer to a pointer to char
a constant int a pointer to a constant int
a constant pointer to an int

chai
December 27, 2019 at 12:51 pm · Reply

[code]
int array[5]{ 9, 7, 5, 3, 1 };
char vowels[]{ 'a', 'e', 'i', 'o', 'u','\0' };
std::cout << "\nElements in array: " << array; //address 00AFFD7C printed
std::cout << "\nElements in vowels: " << vowels; //aeiou printed

    std::cout << "\nArray element 0 has address: " << &array[0];//address 00AFFD7C printed
    std::cout << "\nvowels[0] has address: " << &vowels[0];   //aeiou printed
[code]
I don't understand why pointer to char array is different to pointer to int array. When dealing with pointers are we suppose to treat char very differently to int?

nascardriver
December 28, 2019 at 2:17 am · Reply

They're no different from each other. Only `std::cout` treats them differently.

Closing code tags use a forward slash
[/code]

Teru
November 7, 2019 at 6:55 am · Reply

Hi there!
Thank you for writing these lessons and answering our questions!
I have a lot of trouble with pointers and arrays in C++. I understand that an array decays into a pointer when passed into a function to avoid copying length arrays. I also understand that a pointer is a variable that carries a memory address. I understand what arrays are algorithmically (my first language was Java). However, I'm very confused about the rest of array decay and pointers.
When this post says "In all but two cases (which we'll cover below), when a fixed array is used in an expression, the fixed array will decay
(be implicitly converted) into a pointer that
points to the first element of the array." What does it mean for a fixed array to be "used in an expression"? Does that mean an array always decays? If so, what's the point of an array other than to give length information?
I also found a Stack Overflow answer <https://stackoverflow.com/questions/1461432/what-is-array-decaying> that says "Except when
1. it is the operand of the sizeof operator or
2. the unary (address-of) & operator,
3. or is a string literal used to initialize an array,

an expression that has type "array of type" is converted to an expression with type "pointer to type" that
1. points to the initial element of the array object and

is not an lvalue. (not a function or object).
I'm having a lot of trouble understanding why it decays, when it decays, and what's the point of an array if it's almost always going to decay?
If the array is different from the pointer to the array, how is the distinction made inside C++ if array is not a class? I don't understand what the difference between type "int[5]" and type "int *" means. Is the array a short hand for the pointer that is enforced by the compiler or is it in memory something fundamentally different?
Can an array ever be dereferenced?
Why do arrays in structs or classes not decay when passed into the function?
If you made it all the way to the bottom of this flurry of confused questions, I am deeply deeply grateful for your time and attention. It means a lot to me that you provide this wonderful service for free.
Best regards,
Teru

nascardriver
November 7, 2019 at 7:30 am · Reply

Hello Teru!

> why it decays
You can pass arrays around without making them decay, but then you're limited to arrays of a fixed size. Most of the time we want our functions to work with arbitrarily sized arrays. Because arrays of different sizes are distinct types, a function that accepts an array of size N won't work with an array of size M. When we let arrays decay, they have the same type, so we can re-use functions.

> when it decays
When you copy it into a pointer variable or use it like a pointer, eg. when calling a function with a pointer parameter. As I said before, the function _could_ make it so that the array doesn't decay, but then it only works with one size.

> what's the point of an array if it's almost always going to decay?
A decayed array is still an array, you just can't extract its size, so you have to keep track of the size yourself.

> array is not a class
There is an array class (And several other containers), you'll learn about it later.

> difference between type "int[5]" and type "int *"
One has size information, the other doesn't.

> Is the array a short hand for the pointer that is enforced by the compiler or is it in memory something fundamentally different?
I'm not sure I understand you. An array type has size information, but only at compile-time (Because it's a type, and types only exist at compile-time). The size isn't stored in memory and not otherwise present at run-time (At least not accessible to the programmer).

> Can an array ever be dereferenced?
It will decay first, then you dereference the pointer. You don't have to do anything to make the array decay, it happens automatically.

```
int arr[5]{};
*arr; // Decay to pointer and dereference of the pointer
// @arr is still an array type here, it only had to decay in the above expression.
```

> Why do arrays in structs or classes not decay when passed into the function?
Because the type of the struct/class contains the type of every member. As with passing an array by array type, the function that accepts the struct/class works only with arrays of one size (The size that's specified in the struct/class).

I hope I could clear some of your confusion. If you have any more questions or don't understand something I said, feel free to ask again :)

Teru
November 8, 2019 at 9:32 am · Reply

You cleared up so much confusion! Thank you so much!
Would you say it's correct for me to say:
1. An array is a pointer to a variable with information about the size of the array
2. Arrays of different sizes are different types from each other (int[5] is a different type from int[2])
?
> "An array type has size information, but only at compile-time (Because it's a type, and types only exist at compile-time). The size isn't stored in memory and not otherwise present at run-time (At least not accessible to the programmer)."
What does it mean that "types only exist at compile-time"? Isn't size of the array accessible through array.length?
Thanks again!

nascardriver
November 9, 2019 at 1:17 am · Reply

> An array is a pointer to a variable with information about the size of the array
Yes

> Arrays of different sizes are different types from each other (int[5] is a different type from int[2])
Yes

> types only exist at compile-time
Your processor doesn't know what types are. There are no types in a compiled program. Types only help you to organize data and prevent mistakes by using data in a wrong way.

> Isn't size of the array accessible through array.length
No, not in C++. You can use `std::size(array)` if `array` is an array-type (Not decayed). `std::size` runs at compile-time. In your program, there is no call to `std::size`, just the size of your array (There might be a call when optimization is disabled, but the return value is known at compile-time even then).

hellmet
October 20, 2019 at 9:51 am · Reply

I have a small doubt. While messing around with C (due to a course on Coursera), I learnt that 'sizeof' is actually an 'operator' in the C and C++ language, and hence the value it returns is computed at compile time and stored in the binary itself! So there is no 'runtime' cost to find the size of the array in this case. Just to make sure (I'm also following a book on OS organization), I checked with 'Compiler Explorer' and sure enough, it explicitly moves 20 into the value. I think std::size/std::ssize work this way too. Also, I don't see any other usage where 'array' doesn't degrade to pointer. All this makes me think, is it really meaningful to say that 'array' knows its size?

nascardriver
October 20, 2019 at 10:16 am · Reply

> hence the value it returns is computed at compile time
What you said is correct, but it's not the reason why `sizeof` is computed at compile-time. Operators are functions, they can be evaluated at run-time.

> I checked with 'Compiler Explorer'
Compiler explorer is a great tool, but don't use it to prove anything. You found out that the compiler you selected computes `sizeof` at compile-time, that doesn't mean that the language requires this to happen.

> I think std::size/std::ssize work this way too
`std::size` and `std::ssize` don't use `sizeof`, but they can be computed at compile-time when passed an array.

> is it really meaningful to say that 'array' knows its size?
The type of the array has a size. When an array decays it changes its type to a simple pointer, which doesn't have any information about the size. The size is a part of the type, not of the value. Once you learn about templates you'll understand how types can store information. Arrays don't use templates, it should help nonetheless.

hellmet
October 20, 2019 at 10:35 am · Reply

> The size is a part of the type, not of the value.
Ahhh! That makes sense now!

> Once you learn about templates you'll understand how types can store information.
I looked around and found that C doesn't have templates. Does this mean in C/C++, the mechanism by which types store the size are similar?

I somehow can't seem to wrap my head around the fact that a type's array can also store it's size, since everything is a number in the end? In an array, there doesn't seem to be any extra block allocated around it to store its size? How does this happen!

Thank you very much for your time and patience!

nascardriver
October 21, 2019 at 5:01 am · Reply

> Does this mean in C/C++, the mechanism by which types store the size are similar?
You don't need templates for built-in arrays. Built-in types have their own properties and rules which can't be reproduced by using the language. The compiler knows how these types work, but they're not defined in any .cpp file or similar.
I mentioned templates because they can be used to create custom types with attached information and to extract the size out of an array. Built-in types should function the same in C and C++.

> there doesn't seem to be any extra block allocated around it to store its size?
Types in C++ are only a help for the programmer and compiler. There are no types at run-time (There's an exception which doesn't matter now). Since the array's size is a part of the array's type, the size doesn't exist at run-time (Unless you use a very weird compiler that keeps the size for whatever reason).

hellmet
October 21, 2019 at 5:13 am · Reply

Ohh alright! That makes more sense now! These are just programing aids! Gotcha, Thanks!

Samira Ferdi
September 5, 2019 at 5:16 pm · Reply

Hi, Alex and Nascardriver!

Is it true that how we indexing array is actually indexing through pointer arithmetic?

nascardriver
September 6, 2019 at 12:35 am · Reply

Yes, that's why this weird syntax works

```
int arr[3];

int i{ 2[arr] }; // Same as arr[2]
```

Vitra

August 26, 2019 at 12:34 am · Reply

Thanks much but I'm still not clear about this case:

int main()
{
int a[5] = {1, 2, 3, 4, 5};

std::cout << (a + 1) << '\n';

// Does array a decay to pointer in the line below?
// Why a = a + 1 got an error?

std::cout << a++ << '\n';

return 0;
}

nascardriver
August 26, 2019 at 8:00 am · Reply

> Does array a decay to pointer in the line below?
That's an error.

> Why a = a + 1 got an error?
You can't assign to arrays. This is also the reason why `a++` doesn't work.

Samira Ferdi
August 12, 2019 at 7:02 pm · Reply

Hi, Alex and Nascardriver!

What do you think about my code? Are there anythings should I change or simplify?

```
#include <iostream>

void printArrayElements(int *array, const int length)
{
    std::cout << "All elements of the array (unordered):\n";
    for(int index{ 0 }; index < length; ++index)
        std::cout << array[index] << ' ';

    std::cout << "\n\n";
}

void findMaxElement(int *array, const int length)
{
    //Assuming that the maximum element exist at index 0
    int maxIndex{ 0 };

    for(int currentIndex{ 0 }; currentIndex < length; ++currentIndex)
        //Find the other possibility the existance of the greater element
        //than the maximum element that I assummed
        if(array[maxIndex] < array[currentIndex])
            maxIndex = currentIndex;

    std::cout << "Maximum element in the array " << array[maxIndex]
              << " and has maximum index " << maxIndex << "\n\n";
}

void findMinElement(int *array, const int length)
```

```
28  {
29      //Assuming that the minimum element exist at index 0
30      int minIndex{ 0 };
31      for(int currentIndex{ 0 }; currentIndex < length; ++currentIndex)
32          //Find the other possibility the existance of the lesser element
33          //than the minimum element that I assummed
34          if(array[minIndex] > array[currentIndex])
35              minIndex = currentIndex;
36
37      std::cout << "Minimum element in the array is " << array[minIndex]
38                << " and has minimum index " << minIndex << "\n\n";
39  }
40
41  int main()
42  {
43      int array[]{ 78, 23, 597, 123, 44, 2, 98, 23, 90 };
44      const int length{ sizeof(array) / sizeof(array[0]) };
45
46      printArrayElements(array, length);
47
48      findMaxElement(array, length);
49
50      findMinElement(array, length);
51
52      return 0;
53  }
```

But, I have a question. The "array" argument in the my three functions is hard to tell that is it an array or just a normal variable named "array" or pointer variable named "array". The name of those argument can tell us that those are arrays. But, it is just based on the name of argument, but I'm still not sure enough. So, I think that for the function argument we should write array syntax (array[]) instead of just "array". But, what do you think about this?

```
    printArrayElements(&array[0], length);

    findMaxElement(&array[0], length);

    findMinElement(&array[0], length);
```

nascardriver
August 12, 2019 at 11:55 pm · Reply

- Wrap line 18+, 32+ in curly brackets, because they exceed 1 line.
- Line 43, 44: Should be `constexpr`. Line 44 could use `std::ssize` (Or `std::size` if your compiler doesn't support `std::ssize` yet).

> But, what do you think about this?
That doesn't help. The caller knows the type of the variables they're passing.
You can change your functions to use array syntax

```
void finMaxElement(int array[], const int length)
//                 ^^^^^^^^^^^
```

It has the same meaning, but indicates that the function wants an array.

Samira Ferdi
August 12, 2019 at 5:47 pm · Reply

Hi, Alex and Nascardriver!

Because fixed-array decay into a pointer when passing it to the function, so, I can do this. But, is this way considerably a good way?

```
1  #include <iostream>
2
3  void printElements(int array[], int length)
4  {
5      for(int i{ 0 }; i < length; ++i)
6          std::cout << array + i << " has value: " << *(array + i) << '\n';
7  }
8
9  int main()
10 {
11     int arr[]{ 99, 20, 14, 80 };
12
13     printElements(arr, 4);
14
15     return 0;
16 }
```

nascardriver
August 13, 2019 at 12:23 am · Reply

Please be more specific about what you mean.
Line 3 is fine.
Line 6 should use `array[i]`.
Line 13 should use `std::ssize` or `std::size`.

Alex
August 13, 2019 at 11:50 am · Reply

Passing an array by pointer with a separate length parameter is how things used to be done.

In modern C++, you're better off using std::array and templates to avoid any mismatch between the array and length parameters.

Samira Ferdi
August 13, 2019 at 5:02 pm · Reply

Thanks Alex and Nascardriver! I really really appreciate your answers!
I forget to make my length parameter to be const and I think it should be const. But, once again, thank you!

Anastasia
July 31, 2019 at 8:51 am · Reply

Hi!
In the conclusion to this chapter Alex wrote: "Pointers to const values are primarily used in function parameters (for example, when passing an array to a function) to help ensure the function doesn’t inadvertently change the passed in argument."
But it seems that passing an array as const (a pointer pointing to a const value) doesn't fully ensure that the values won't be overwritten, since the values are treated as const only while accessed by the pointer the array is converted into. But nothing prevents another pointer from accessing and modifying them.

```
1  #include <iostream>
2
3  void changeFirstValue(const int array[]) {
4      // this is not allowed
5      *array = 6; // error: assignment of read-only location ‘* array’
6
7      // the value stored at the address the pointer is pointing to can be eventually overwri
8      int *another_pointer { const_cast<int*>(array) };
9      *another_pointer = 100;
10 }
11
12 int main() {
13
14     const int array[] {1, 2, 3, 4, 5};
15     changeFirstValue(array);
16     std::cout << array[0] << '\n'; // prints '100'
17
18     return 0;
19 }
```

There probably will be a more detailed explanation of how to handle this in the future chapters, looking forward to that.

nascardriver
July 31, 2019 at 9:26 am · Reply

Line 9 produces undefined behavior, you're not allowed to modify an entity after casting away its constness.
A `const_cast` should only be used if, for whatever reason, a `const` member function isn't marked as such.

Anastasia
July 31, 2019 at 9:58 am · Reply

> Line 9 produces undefined behavior

Hm, I don't see any warnings while compiling that snippet with all the flags turned on by '-Wall' (gcc 7.4.0).

And that wasn't the point anyway. Even if my example doesn't make a lot of sense, what bothers me is that it seems I can't be really sure the const value(s) I'm passing by address to a function won't get changed. This is confusing and disturbing, especially considering the fact that this wasn't an issue before (when most of the examples we were dealing with previously were passed to functions by value).

Anastasia
July 31, 2019 at 10:19 pm · Reply

*Sorry for a bit of misinformation. The conclusion I've mentioned above belongs to the chapter 6.10 'Pointers and const', not this one. I went ahead and read that chapter, because it wasn't clear to me how to deal with const and pointers(in particular with values passed to functions by address), but it is still confusing.

nascardriver
August 1, 2019 at 1:25 am · Reply

> I don't see any warnings
There aren't warnings for everything.

> I can't be really sure
CPP is a low enough language to modify the entire program at run-time, you can't be sure about anything if you go by that.
If you assume that the code is well-defined (Yours isn't), then `const` variables cannot be changed.
Line 9 most likely changes the value of `array[0]` to `100`, but it might as well shut off your computer or start playing a song.

Anastasia
August 1, 2019 at 1:48 am · Reply

I wondered why my snippet compiled at all, it seemed dangerous to do something like that even to a newbie like me. Well, I won't call that reassuring, but thank you for clarifying this a bit.

Harshit
April 16, 2019 at 11:17 pm · Reply

Hi Alex,

Here in my code, I can see that an array in char does not decay into pointers whereas an array of int type does. Why does this happen?

```
int main()
{
    int array[]={9,7,5,3,1};

    cout<<*array<<'\n'; // prints first element of the array
    cout<<&array[0]<<'\n'; // print address of the first element
    cout<<array<<'\n'; // print address of the first element again, i.e array decays into a

    char name[]="Harshit";
    cout<<*name<<'\n'; // prints first element
    cout<<&name[0]<<'\n'; // prints the complete name
    cout<<name<<'\n'; // prints the complete name
    return 0;
}
```

nascardriver
April 17, 2019 at 3:06 am · Reply

@std::cout::operator<< treats char* as strings.

Harshit
April 17, 2019 at 4:26 am · Reply

Okay, thanks.

Jeff
April 9, 2019 at 6:50 pm · Reply

Hi Alex.

You mentioned that Arrays in structs and classes don't decay... But I have some code here that shows me the array is decaying. What am I not understanding here?

```
class Array
{
private:
    int m_some_array[23]{ 22,33,44, 55 };
public:
    void checkArrayDecay()
    {
        // Decay seems to work
        *m_some_array = 15;

        for (int i = 0; i < 4; ++i)
            std::cout << *(m_some_array + i) << std::endl;
    }
};

int main()
{
    Array ar;
    ar.checkArrayDecay();

    std::cin.get();
    return 0;
}
```

nascardriver

April 10, 2019 at 2:35 am · Reply

```
#include <iostream>

struct S
{
  int arr[5]{};
};

void fn1(int arr[])
{
  // @arr decayed to a pointer.
  std::cout << sizeof(arr) << '\n'; // Output: 8
}

void fn2(S s)
{
  // @s.arr didn't decay, because it's a member of a struct.
  std::cout << sizeof(s.arr) << '\n'; // Output: 20
}

int main(void)
{
  int arr[5]{};

  fn1(arr);

  S s{};

  fn2(s);
```

```
    return 0;
}
```

Behzad
April 4, 2019 at 2:51 pm · Reply

I would suggest adding this code to clarify the effect of address-of operator (&) on an array:

```
#include <iostream>
using namespace std;

int main()
{
    int array[5] {11, 12};
    int *ptr {nullptr};
    ptr = array;

    cout << "int array[5] {11, 12};"   << endl;
    cout << "int *ptr {nullptr};;" << endl;
    cout << "ptr = array;"         << endl << endl;

    cout << " ptr         = " <<  ptr << " : The address stored in ptr = address of the fir
    cout << "*ptr         = " << *ptr << "             : Value of the first entry of array"
    cout << "&ptr         = " << &ptr << " : Address of ptr" << endl;
    cout << " array       = " << array<< " : Address of the first entry of array" << endl;
    cout << "*array       = " <<*array<< "             : Value of the first entry of array"
    cout << "&array       = " <<&array<< " : A pointer to an array. The value of this point
    cout << "*(&array)    = " << *(&array)  << " : = array" << endl;
    cout << "**(&array)   = " << **(&array) << "             : = *array" << endl;
    cout << "*(&array)[0] = " << *(&array)[0] << "             : = *((&array) + 0)  Note: (
    cout << "*(&array)[1] = " << *(&array)[1] << "             : = *((&array) + 1)  Note: (
}
```

Lakshya Malhotra
March 28, 2019 at 2:13 pm · Reply

Hi,
I can't explain what's going on with my code. So I am trying to print the length of a C-string and then output the string itself. My code is outputting the length fine but it is not printing the string.

```
#include <iostream>

int mystrlen(char *ptr)
{
    std::cout << "ptr = " << ptr << " " << '\n';
    unsigned int len = 0;
    while (*ptr != '\0')
    {
        len++;
        ptr++;
    }
    std::cout << "len = " << len << '\n';
    for (int i=0; i<len; ++i)
    {
        std::cout << ptr[i] << " ";
    }
    return len;
}
```

```
19
20  int main()
21  {
22      char ptr1[] = "hello";
23      char *ptr = ptr1;
24
25      mystrlen(ptr);
26
27      return 0;
28  }
```

nascardriver
March 29, 2019 at 1:31 am · Reply

* Line 6, 13, 22, 23: Initialize your variables with brace initializers. You used copy initialization.

You're modifying @ptr in line 10. In line 15, @ptr points to the 0-terminator.

Lakshya Malhotra
March 29, 2019 at 1:53 am · Reply

Thanks for your reply. I appreciate it.

Jagadeesh Takkalaki
February 18, 2019 at 10:03 am · Reply

```
#include <iostream>
using namespace std;

unsigned int mystrlen(char *ptr)
{
    cout<<"ptr = "<<ptr<<" ";
    unsigned int len=0;
    while(*ptr!=NULL)
    {
        len++;
        ptr++;
    }
    cout<<"len= "<<len<<endl;
    return len;
}
void changestr(char *ptr)
{
    unsigned int i=0;
    int len=mystrlen(ptr);
    for(int i=0;i<len;i++)
    {
        if(ptr[i]>='a'||ptr[i]<='z')
        {
         ptr[i]-='a'-'A';
        }
    }

     cout<<ptr<<endl; //expected output is RAM SHAM BAM but getting RAM
}
```

```
30  int main() {
31      // your code goes here
32      char ptr1[]="ram sham bam";
33      char *ptr=ptr1;
34      changestr(ptr);
35      cout<<"ptr ="<<ptr;
36      return 0;
37  }
```

What is the problem with above code? I am not getting entire string as output.It is printing only characters of string until space, not printing character after space.
Expected value of ptr1:RAM SHAM BAM
Present Output:RAM

nascardriver
February 19, 2019 at 5:50 am · Reply

* Line 7, 18, 19, 20, 32, 33: Initialize your variables with brace initializers. You used copy initialization.
* Line 2: Initialize your variables with brace initializers.
* Use ++prefix unless you need postfix++.
* Don't use "using namespace".
* Use the auto-formatting feature of your editor.
* Line 22: Should be &&, not ||.
* Enable compiler warnings, read them, fix them.

Jeroen P. Broks
February 18, 2019 at 2:35 am · Reply

Now since <type>* and <type>[] will as parameters just pass the pointer, I did put things on the test with <type>[<length>].

```
#include <iostream>

using namespace std; // I know it's bad practise, but I was short on time and lazy, and thi


void myF(int p[5]){
    cout << "In function\n";
    cout << &p << '\n';
    cout << p[0] << '\n';
}


int main(){
    int p[5];
    p[0]=20;
    cout << "In main\n";
    cout << &p << '\n';
    cout << p[0] << '\n';
    myF(p);
    return 0;
}
```

And the result was this:

```
In main
0x7fff50288800
20
```

```
In function
0x7fff502887c8
20
```

I suppose that this method DOES copy the array as a whole into the function, or am I wrong?

Alex
February 19, 2019 at 7:44 pm · Reply

Built-in arrays work strangely in C/C++. In such a case, the array parameter is treated as a pointer to an array, and the size information is discarded. So the array isn't copied, just the pointer to the array is copied.

If you want to retain the type information, pass the array by reference, or better, use std::array.

Bad_at_Coding
January 25, 2019 at 11:59 am · Reply

Not that this one is going to be important, but we can actually change the following

```
int length { sizeof( my_arr ) / sizeof( my_arr [ 0 ] ) };
```

to

```
int length { sizeof( my_arr ) / sizeof( *my_arr ) };
```

Doug
January 23, 2019 at 4:38 pm · Reply

"The variable array contains the address of the first element of the array, as if it were a pointer! You can see this in the following program:"

I'm not sure this is correct. That program demonstrates decay to a pointer, nothing more.

An array variable no more contains an address than a structure variable contains the address of the structure or an integer variable contains the address of the integer. Instead, an array variable contains the contents of the array, just like a structure variable contains the contents of the structure.

Alex
January 24, 2019 at 8:14 pm · Reply

I agree. I've updated the lesson accordingly. Thanks for the correction.

Minh
November 15, 2018 at 6:15 am · Reply

hey, i want to ask a pretty simple thing
let us have a data structure like

```
struct node
{
    node *left = nullptr;
    node *right = nullptr;
    node *parent = nullptr;
}
typedef struct node* pnode;
```

I have the pnode A's left is B and right is C. A's parent is X.
I want to swap B to be the parent of A. So B->(A)->C
When I do this

```
void rR(pnode X)
{
    pnode target = X->left; // this is B
    pnode tg = target->right; // this is B's right
    target->right = X;
    target->parent = X->parent;
    X->parent = target;
    X->left = tg;
    tg->parent = X;
}
```

What I want to do is target's parent will point to X's parent but X remain X, but the node X change into X's parent too, so how to point target's parent to X's parent. :(
Thank you in advance.

Jack
November 13, 2018 at 12:39 am · Reply

Hi guys,
Can you just tell me what does this mean in more detail?
(strings being an exception because they're null terminated).
and how could I pass the length of an array in a seperate parameter?

Alex
November 13, 2018 at 11:00 pm · Reply

Because strings always end in a '\0' character, we don't necessarily need to know the length of a string to know where it ends. Instead, we can keep going until we run into the '\0'.

This doesn't work for other arrays because those arrays don't typically have an element that signals termination.

> how could I pass the length of an array in a seperate parameter?

Add an "int length" parameter to your function. If you know the array's length already, you can pass it as an argument. If you don't know it but your array is a fixed array, you can use the sizeof(array)/sizeof(array[0]) trick to get the length. If you have a dynamic array and you don't know the length, you're out of luck.

Jack
November 13, 2018 at 11:07 pm · Reply

Thank you sir, that was superb.

« Older Comments 1 2 3