Home

PUBLIC

Questions

Tags

Users

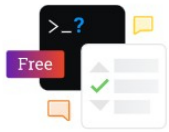
Companies

COLLECTIVES

Explore Collectives

TEAMS

Stack Overflow for Teams – Start collaborating and sharing organizational knowledge.



Create a free Team

Why Teams?

Why Teams?

Why is arr and &arr the same?

Ask Question

Asked 10 years, 5 months ago Modified 3 months ago Viewed 15k times

24

15

I have been programming c/c++ for many years, but todays accidental discovery made me somewhat curious... Why does both outputs produce the same result in the code below? (`arr` is of course the address of `arr[0]`, i.e. a pointer to `arr[0]`. I would have expected `&arr` to be the adress of that pointer, but it has the same value as `arr`)

```
int arr[3];
cout << arr << endl;
cout << &arr << endl;
```

Remark: This question was closed, but now it is opened again. (Thanks ?)

I know that `&arr[0]` and `arr` evaluates to the same number, but that is *not* my question! The question is why `&arr` and `arr` evaluates to the same number. If `arr` is a literal (not stored anyware), then the compiler should complain and say that `arr` is not an lvalue. If the address of the `arr` is stored somewhere then `&arr` should give me the address of that location. (but this is not the case)

if I write

> const int* arr2 = arr;

then `arr2[i]==arr[i]` for any integer `i`, but `&arr2 != arr`.

c++ c pointers

Share Follow

edited Feb 4, 2012 at 7:50
R. Martinho Fernandes
219k ● 71 ● 423 ● 503

asked Jan 18, 2012 at 20:04
ragnarius
5,340 ● 10 ● 44 ● 65

What does "the address of that pointer" mean? The pointer isn't stored in a variable. – David Schwartz Jan 18, 2012 at 20:07

1 @David Schwartz, If I had written int* arr2 = new int[3], then &arr2 would have been stored in a variable. Maybe arr is a literal constant? But if one take the address of a constant (e.g. &123) then the compiler complains (123 is not an lvalue). The compiler did not complain when I took the address of arr. – ragnarius Jan 18, 2012 at 20:13

6 I do not think it is a duplicate! – ragnarius Jan 18, 2012 at 20:27

Right, so your expectation makes no sense. If it was the address of the pointer, your code wouldn't compile. (Unless you expected it not to compiler. But in that case, how would you pass the address of an array to to other code like the address of any other variable?) – David Schwartz Jan 18, 2012 at 20:54

Normal pointers are stored somewhere and I can take the address of a normal pointer to find out where it is

The Overflow Blog

What Apple's WWDC 2022 means for developers

An Engineer's Field Guide to Great Technical Writing (Ep. 455)

Featured on Meta

Announcing the arrival of Valued Associate #1214: Dalmarus

Testing new traffic management tool

Ask Wizard Test Results and Next Steps

Linked

2 trying to understand what does the name of an array mean

1 Address of an address in C. Valid or not?

50 Different Pointer Arithmetic Results when Taking Address of Array

1 What does array name represent?

Related

2853 What is the difference between #include <filename> and #include "filename"?

2969 How do you set, clear, and toggle a single bit?

3710 What are the differences between a pointer variable and a reference variable in C++?

2888 When should static_cast, dynamic_cast, const_cast and reinterpret_cast be used?

1755 With arrays, why is it the case that a[5] == 5[a]?

4234 The Definitive C++ Book Guide and List

9765 What is the "-->" operator in C/C++?

3217 Improve INSERT-per-second performance of SQLite

stored. I think it makes sense to believe that I can take the address of arr too. Unless it is a literal, but then the compiler would have complained. – ragnarius Jan 18, 2012 at 21:26

Show **1** more comment

7 Answers

Sorted by: Highest score (default)

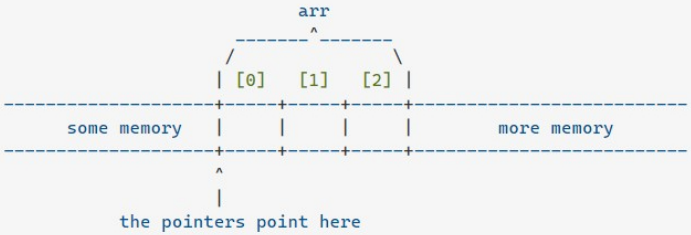
17

```
#include <cassert>

struct foo {
    int x;
    int y;
};

int main() {
    foo f;
    void* a = &f.x;
    void* b = &f;
    assert(a == b);
}
```

For the same reason the two addresses `a` and `b` above are the same. The address of an object is the same as the address of its first member (Their types however, are different).



As you can see in this diagram, the first element of the array is at the same address as the array itself.

Share Follow

edited Jan 18, 2012 at 20:16

answered Jan 18, 2012 at 20:08

R. Martinho Fernandes
219k ● 71 ● 423 ● 503

I tried it, but &f.x != &f.y . So the address of an object is only the same as the address of its FIRST member... – ragnarius Jan 18, 2012 at 20:20

1 @ragnarius: The address of the array is the same as the address of the first element of the array. Has nothing to do with the members of the elements. – Mooing Duck Jan 18, 2012 at 20:27

@ragnarius: that's exactly what I said. – R. Martinho Fernandes Jan 18, 2012 at 20:37

To bad my question was closed. I know that and array, when type casted to an int, has the same value as the address of the first element. But this was not my question.. – ragnarius Jan 18, 2012 at 20:49

Add a comment

They're not the same. They just are at the same memory location. For example, you can write

26447 Why is processing a sorted array faster than processing an unsorted array?

1808 Why should I use a pointer rather than the object itself?

Hot Network Questions

- Is this asbestos wiring insulation?
- How to iterate through list infinitely with +1 offset each loop
- Why MERGE doesn't insert more than 277 records into a table which is configured with temporal table and a non-clustered index on history table
- Can DDNS provider perform a MITM attack?
- Could a hand launched missile work?
- Are antennas specified for a certain frequency?
- Render display math properly in Standalone environment
- How do I set up an engine never to resign in Arena GUI?
- Convert List<char[]> into an Array char[] without using System.arraycopy()
- I might have found a fundamental problem with Novikov's Self-Consistency Solution principle?
- Can 您 to be used in plural form?
- My 1 Week Python Journey — A Quiz Game that I made in Python
- What is the theological justification behind chanting the holy names in Gaudiya Vaishnavism?
- Why is { w | |w| mod 3 = #_a(w) mod 3 } a Regular Language?
- Why does the kobold inventor in Tomb of Annihilation reference Appendix A?
- Is there an open-source translation movement?
- Finite coverings by closed subschemes
- Moving the equation to the left in "align"
- What good is my paper if the code is not open source?
- Where can I put my rear light that is designed for the seatpost if the saddle bag is in the way?
- Ding vs. Rapport, Candidates 2022: Why did Ding play 23.Qxe2 instead of 23.Qxd8?
- Is there any reason why The Chosen One should bother to do anything if their victory is prophesized?
- Why does the luminal test need hydrogen peroxide?
- How can I define a distribution and get the expectation in Mathematica?

Question feed

12

`arr+2` to get the address of `arr[2]`, but not `(&arr)+2` to do the same.

Also, `sizeof arr` and `sizeof &arr` are different.

Share Follow

edited Jun 5, 2017 at 14:02

nalzok
13.4k • 18 • 64 • 120

answered Jan 18, 2012 at 20:08

Mr Lister
44.1k • 15 • 107 • 146

4 It's good to note that `(&arr)+2` will compile, and maybe even run, but it's undefined behavior, and will *not* give you `arr[2]`. – Mooing Duck Jan 18, 2012 at 20:14

It won't compile here! `cannot convert 'int (*)[3]' to 'int*' in initialization` – Mr Lister Jan 18, 2012 at 20:16

1 `int(*)[3]` is convertable to an `int**` A pointer to an array is convertable to a pointer to a pointer. – Mooing Duck Jan 18, 2012 at 20:23

I see. My apologies, I am an idiot. I was trying to assign that to something incompatible in the same statement; just writing `(&arr)+2;` by itself compiles fine. – Mr Lister Jan 18, 2012 at 20:32

5 @Mooing: That is wrong, `int (*)[3]` is *not* convertible to `int**`. The pointer-to-array needs the exact size of the inner dimension to know how far to step: ideone.com/iDkB7. – Xeo Jan 18, 2012 at 20:53

Show **1** more comment

9

The two have the same value but different types.

When it's used by itself (not the operand of `&` or `sizeof`), `arr` evaluates to a pointer to `int` holding the address of the first `int` in the array. `&arr` evaluates to a pointer to array of three `int`s, holding the address of the array. Since the first `int` in the array has to be at the very beginning of the array, those *addresses* must be equal.

The difference between the two becomes apparent if you do some math on the results:

`arr+1` will be equal to `arr + sizeof(int)`.

`((&arr) + 1)` will be equal to `arr + sizeof(arr)` == `arr + sizeof(int) * 3`

Edit: As to how/why this happens, the answer is fairly simple: because the standard says so. In particular, it says (§6.3.2.1/3):

> Except when it is the operand of the sizeof operator or the unary & operator, 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 points to the initial element of the array object and is not an lvalue.

[note: this particular quote is from the C99 standard, but I believe there's equivalent language in all versions of both the C and C++ standards].

In the first case (`arr` by itself), `arr` is not being used as the operand of sizeof, unary &, etc., so it is converted (not promoted) to the type "pointer to type" (in this case, "pointer to int").

In the second case (`&arr`), the name obviously *is* being used as the operand of the unary `&` operator... so that conversion does not take place.

operator -- so that conversion does *not* take place.

Share Follow

edited Jan 18, 2012 at 20:36

answered Jan 18, 2012 at 20:09

Jerry Coffin
456k ● 76 ● 600 ● 1068

@MooingDuck: Where do you (think you) see anything related to the address of a pointer? (hint: there's no such thing here). There's no promotion involved here either. – Jerry Coffin Jan 18, 2012 at 20:16

you state that `arr` "yields" a pointer to `int`, which is ambiguous, and possibly misleading. – Mooing Duck Jan 18, 2012 at 20:25

I am not sure I understand your quote. We are discussing the unary & operator and your quote says that "Except it is the [...] unary & operator...". – ragnarius Jan 20, 2012 at 18:02

This answer is more clearer than the accepted answer. – InQusitive Aug 7, 2021 at 15:08

Add a comment

5

The address is the same but both expressions are different. They just start at the same memory location. The types of both expressions are different.

The value of `arr` is of type `int *` and the value of `&arr` is of type `int (*)[3]`.

`&` is the address operator and the address of an object is a pointer to that object. The pointer to an object of type `int [3]` is of type `int (*)[3]`

Share Follow

edited Jan 18, 2012 at 20:13

answered Jan 18, 2012 at 20:08

ouah
139k ● 15 ● 262 ● 325

As it happens, the address of the array itself and the address of its first element are numerically the same. But they have different types and behave differently in other respects. – David Schwartz Jan 18, 2012 at 20:11

The value of `arr` is of type `int (&)[3]` which gets "promoted" to `int*` since `cout` doesn't have an overload for arrays. – Mooing Duck Jan 18, 2012 at 20:12

Add a comment

4

They are not the same.

A bit more strict explanation:

`arr` is an **lvalue** of type `int [3]`. An attempt to use `arr` in some expressions like `cout << arr` will result in lvalue-to-rvalue conversion which, as there are no rvalues of array type, will convert it to an rvalue of type `int *` and with the value equal to `&arr[0]`. This is what you can display.

`&arr` is an **rvalue** of type `int (*)[3]`, pointing to the array object itself. No magic here :-) This pointer points to the same address as `&arr[0]` because the array object and its first member start in the exact same place in the memory. That's why you have the same result when printing them.

An easy way to confirm that they are different is comparing `*(arr)` and `*(&arr)`: the first is an lvalue of type `int` and the second is an lvalue of type `int[3]`.

Share Follow

answered Jan 18, 2012 at 22:29

Kos
68k • 23 • 165 • 229

Add a comment

2

Pointers and arrays can often be treated identically, but there are differences. A pointer does have a memory location, so you can take the address of a pointer. But an array has nothing pointing to it, at runtime. So taking the address of an array is, to the compiler, syntactically defined to be the same as the address of the first element. Which makes sense, reading that sentence aloud.

Share Follow

answered Jan 18, 2012 at 20:11

Graham Perks
22.4k • 8 • 57 • 81

Add a comment

0

I found Graham Perks' answer to be very insightful, I even went ahead and tested this in an online compiler:

```
int main()
{

int arr[3] = {1,2,3};
int *arrPointer = arr; // this is equivalent to: int *arrPointer = &arr;

printf("address of arr: %p\n", &arr);
printf("address of arrPointer: %p\n", &arrPointer);

printf("arr: %p\n", arr);
printf("arrPointer: %p\n", arrPointer);

printf("*arr: %d\n", *arr);
printf("*arrPointer: %d\n", *arrPointer);

return 0;
}
```

Outputs:

```
address of arr:        0x7ffed83efbac
address of arrPointer: 0x7ffed83efba0
arr:                   0x7ffed83efbac
arrPointer:            0x7ffed83efbac
*arr:                  1
*arrPointer:           1
```

It seems the confusion was that arr and arrPointer are equivalent. However, as Graham Parks detailed in his answer, they are not.

Visually, the memory looks something like this:

```
[Memory View]
[memory address: value stored]
```

```
arrPointer:
0x7ffed83efba0: 0x7ffed83efbac

arr:
0x7ffed83efbac: 1
0x7ffed83efbb0: 2
0x7ffed83efbb4: 3
```

As you can see, `arrPointer` is a label for memory address `0x7ffed83efba0` which has 4 bytes of allocated memory which hold the memory address of arr[0].

On the other hand, `arr` is a label for memory address `0x7ffed83efbac`, and as per Jerry Coffin's answer, since the type of variable `arr` is "array of type", it gets converted to a "pointer of type" (which points to the array's starting address), and thus printing `arr` yields `0x7ffed83efbac`.

The key difference is `arrPointer` **is** an actual pointer and has its own memory slot allocated to hold the value of the memory it's pointing to, so `&arrPointer != arrPointer`. Since `arr` is **not** technically a pointer but an array, the memory address we see when printing `arr` is not stored elsewhere, but rather determined by the conversion mentioned above. So, the values (not types) of `&arrPointer` and `arrPointer` are equal.

Share Follow

answered Mar 22 at 8:27

deniskov
1

Add a comment

Your Answer

Sign up or log in

Sign up using Google

Sign up using Facebook

Sign up using Email and Password

Post as a guest

Name

Email

Required, but never shown

Post Your Answer By clicking "Post Your Answer", you agree to our terms of service, privacy policy and cookie policy

Not the answer you're looking for? Browse other questions tagged c++ c pointers or ask your own question.