Why doesn't C++ support functions returning arrays?

Asked 11 years, 2 months ago Modified 1 year, 11 months ago Viewed 28k times

52

Some languages enable you to just declare a function returning an array like a normal function, like Java:

```
public String[] funcarray() {
   String[] test = new String[]{"hi", "hello"};
   return test;
}
```

Why doesn't C++ support something like `int[] funcarray(){}` ? You can return an array, but it's a real hassle to make such a function. And also, I heard somewhere that strings are just arrays of char. So if you can return a string in C++, why not an array?

c++ arrays function

Share Edit Follow Flag

asked Mar 1, 2011 at 16:38
Lockhead
2,303 • 7 • 31 • 48

1 Why not create the array using a pointer then return the pointer? – RageD Mar 1, 2011 at 16:57

@RageD Sure, you could do that. But couldn't the creators of C++ make array returning functions that do that automatically without bothering the programmer? – Lockhead Mar 1, 2011 at 17:03

1 @MisterSir: I would say it's more of a feature - it allows consistency. If you create an array using a pointer, you've dynamically allocated memory on the heap - that said, you can return by reference and remove any copy overhead (so size of an array does not effect efficiency). However, you do need to remember to free the memory you've allocated. – RageD Mar 1, 2011 at 18:25

7 @MisterSir - also, it's not *bothering the programmer*. C and C++ are not application programming languages. They are **systems** programming languages. As such, there are design decisions in these languages that reflect the intended type of work. Don't think high-level. Think low level. Go low, down to the metal. Review back the stuff we learned in assembly, computer org and operating systems. Then things will start to make much more sense when it comes to C and C++. – luis.espinal Mar 1, 2011 at 21:06

2 @luis.espinal: "C and C++ are not application programming languages. They are systems programming languages. [...] Don't think high level." - they're extremely heavily used for and well suited to both (C showing its age of course). Your point about history and use in systems programming aiding understanding is valid, but not the suggestion that either language isn't or can't be suitable for high-level / application programming. – Tony Delroy Apr 12, 2013 at 2:34

10 Answers

Sorted by: Highest score (default)

71

I'd wager a guess that to be concise, it was simply a design decision. More specifically, if you really want to know why, you need to work from the ground up.

Let's think about C first. In the C language, there is a clear distinction between "pass by reference" and "pass by value". To treat it lightly, the name of an array in C is really just a pointer. For all intents and purposes, the difference (generally) comes down to allocation. The code

静态数组, A a[]　，在栈上分配空间;
动态数组, 使用new, malloc,在堆上分配空间
char ch[]="hello";//在栈上
char* ch= new char[6];//在堆上

```
int array[n];
```

would create 4*n bytes of memory (on a 32 bit system) on the stack correlating to the scope of whichever code block makes the declaration. In turn,

```
int* array = (int*) malloc(sizeof(int)*n);
```

would create the same amount memory, but on the heap. In this case, what is in that memory isn't tied to the scope, only the reference TO the memory is limited by the scope. Here's where pass by value and pass by reference come in. Passing by value, as you probably know, means that when something is passed in to or returned from a function, the "thing" that gets passed is the result of evaluating the variable. In other words,

```
int n = 4;
printf("%d", n);
```

will print the number 4 because the construct `n` evaluates to 4 (sorry if this is elementary, I just want to cover all the bases). This 4 has absolutely no bearing or relationship to the memory space of your program, it's just a literal, and so once you leave the scope in which that 4 has context, you lose it. What about pass by reference? Passing by reference is no different in the context of a function; you simply evaluate the construct that gets passed. The only difference is that after evaluating the passed "thing", you use the result of the evaluation as a memory address. I once had a particular cynical CS instructor who loved to state that there is no such thing as passing by reference, just a way to pass clever values. Really, he's right. So now we think about scope in terms of a function. Pretend that you can have an array return type:

```
int[] foo(args){
    result[n];
    // Some code
    return result;
}
```

The problem here is that result evaluates to the address of the 0th element of the array. But when you attempt to access this memory from outside of this function (via the return value), you have a problem because you are attempting to access memory that is not in the scope with which you are working (the function call's stack). So the way we get around this is with the standard "pass by reference" jiggery-pokery:

```
int* foo(args){
    int* result = (int*) malloc(sizeof(int)*n));
    // Some code
    return result;
}
```

We still get a memory address pointing to the 0th element of the Array, but now we have access to that memory.

What's my point? In Java, it is common to assert that "everything is pass by value". This is true. The same cynical instructor from above also had this to say about Java and OOP in general: Everything is just a pointer. And he's also right. While everything in Java is in fact pass by value, almost all of those values are actually memory addresses. So in Java, the language does let you return an array or a String, but it does so by turning it in to the version with pointers for you. It also manages your memory for you. And automatic memory management, while helpful, is not efficient.

This brings us to C++. The whole reason C++ was invented was because Bjarne Stroustrup had been experimenting with Simula (basically the original OOPL) during his PhD work, and thought it was fantastic conceptually, but he noticed that it performed rather terribly. And so he began working on what was called C with Classes, which got renamed to C++. In doing so, his goal was to make a programming language that took SOME of the best features from Simula but remained powerful and fast. He chose to extend C due to its already legendary performance, and one tradeoff was that he chose to not implement automatic memory management or garbage collecting on such a large scale like other OOPL's. Returning an array from one of the template classes works because, well, you're using a class. But if you want to return a C array, you have to do it the C way. In other words, C++ does support returning an array EXACTLY the same way that Java does; it just doesn't do all of the work for you. Because a Danish dude thought it'd be too slow.

Share Edit Follow Flag

edited Feb 19, 2013 at 22:38
Lenar Hoyt
5,702 • 6 • 45 • 55

answered Mar 1, 2011 at 19:04
Doug Stephen
6,971 • 1 • 36 • 45

1 Also, in response to the "Stings are arrays of characters" comment; This is mostly true. In C, there is no such thing as a String type; you handle it by yourself. They are stored in arrays of characters, terminated by a null, and while there exists a String library to do things like find the length and all of that, it is done by parsing the string. A String in C++ or Java can be thought of as a class that contains an array of characters but also contains other member fields that maintain information about the array like length so it is easier to manipulate. So back to pass by reference. – Doug Stephen Mar 1, 2011 at 19:07

1 This is EXACTLY the answer I was looking for! Greatly improved my understanding of memory as well. Thank you! – Lockhead Mar 1, 2011 at 20:04

4 Not again... **arrays and pointers are different beasts** this type of answers, even with the *to treat it lightly* qualifier only add to the confusion. – David Rodríguez - dribeas Mar 1, 2011 at 20:25

2 I also never said an array was a pointer. I said that the NAME of an array was a pointer. Which, while very semantically false, was just a short and non-technical way of saying that except for in very special circumstances, the NAME of an array of type T will decay in to a pointer of type T pointing at the first element, though it goes without saying that the name of an array is an unmodifiable lvalue. But sorry nonetheless. I understand your concern. – Doug Stephen Mar 1, 2011 at 20:36

2 This should be nominated for some kind of awesome answer award. I just learned a whole bunch of stuff because it rearranged things I had known and taken for granted all along. – Mad Physicist Oct 15, 2014 at 22:41

32

C++ does support it - well sort of:

```
vector< string> func()
{
   vector<string> res;
   res.push_back( "hello" );
   res.push_back( "world" );
   return res;
}
```

Even C sort-of supports it:

```
struct somearray
{
  struct somestruct d[50];
};

struct somearray func()
{
   struct somearray res;
   for( int i = 0; i < 50; ++i )
   {
      res.d[i] = whatever;
   }
   // fill them all in
   return res;
}
```

A `std::string` is a class but when you say a string you probably mean a literal. You can return a literal safely from a function but actually you could statically create any array and return it from a function. This would be thread-safe if it was a const (read-only) array which is the case with string literals.

The array you return would degrade to a pointer though, so you would not be able to work out its size just from its return.

Returning an array, if it were possible, would have to be fixed length in the first place, given that the compiler needs to create the call stack, and then has the issue that arrays are not l-values so receiving it in the calling function would have to use a new variable with initialisation, which is impractical. Returning one may be impractical too for the same reason, although they might have used a special notation for return values.

Remember in the early days of C all the variables had to be declared at the top of the function and you couldn't just declare at first use. Thus it was infeasible at the time.

They gave the workaround of putting the array into a struct and that is just how it now has to remain in C++ because it uses the same calling convention.

Note: In languages like Java, an array is a class. You create one with new. You can reassign them (they are l-values).

Share Edit Follow Flag

edited Mar 2, 2011 at 11:22

answered Mar 1, 2011 at 16:42
CashCow
29.9k • 4 • 56 • 89

3 If the size of the array is fixed at compile time, you can use the time std::array<X,N> (or std::tr1::array<X,N> or boost::array<X,N>). – ysdx Mar 1, 2011 at 18:29

1 A std::vector is not an array, and neither a struct containing one. Those are simply mechanisms to work-around the limitation on returning arrays (the actual native type, not a struc or object wrapper for it). I understand where you are going with it, and these are workable examples. However, these are neither examples of a feature (returning **native type** arrays) being supported by C++ (or C), nor explain why the limitation exists in C++. – luis.espinal Mar 1, 2011 at 20:39

1 @luis C++ uses the same calling convention as C. Arrays are not l-values in C or C++ which is the main issue. – CashCow Mar 2, 2011 at 11:23

Your example is still returning an invalid pointer to local memory -- without a copy constructor to do a deep copy, the 'd' member of the return value will be identical to the 'd' member of local variable 'res', which points to memory on the stack which no longer exists. – c-urchin Oct 8, 2012 at 19:10

1 @v.oddou But an array isn't implicitly constructable from a pointer. An "array" function *parameter* is not an array, it is a pointer. It is allowed to look like an array to confuse people (someone probably thought it was a good idea sometime in the late 60s.) – juanchopanza Apr 23, 2015 at 5:34

26

Arrays in C (and in C++ for backwards compatibility) have special semantics that differ from the rest of the types. In particular, while for the rest of the types, C only has pass-by-value semantics, in the case of arrays the effect of the pass-by-value syntax simulates pass-by-reference in a strange way:

In a function signature, an argument of type *array of N elements of type T* gets converted to *pointer to T*. In a function call passing an array as argument to a function will *decay* the array to a *pointer to the first element*, and that pointer is copied into the function.

Because of this particular treatment for arrays --they cannot be passed by value--, they cannot be returned by value either. In C you can return a pointer, and in C++ you can also return a reference, but the array itself cannot be allocated in the stack.

If you think of it, this is not different from the language that you are using in the question, as the array is dynamically allocated and you are only returning a pointer/reference to it.

The C++ language, on the other hand, enables different solutions to that particular problem, like using `std::vector` in the current standard (contents are dynamically allocated) or `std::array` in the upcoming standard (contents can be allocated in the stack, but it might have a greater cost, as each element will have to be copied in those cases where the copy cannot be elided by the compiler). In fact, you can use the same type of approach with the current standard by using off-the-shelf libraries like `boost::array`.

Share Edit Follow Flag

answered Mar 1, 2011 at 16:53
David Rodríguez - dribeas
199k • 21 • 284 • 478

Regarding "In a function signature, [arrays -> pointers]" "[therefore] they cannot be returned by value". 8.3.5.5 does require 'any parameter of type "array of T"' be adjusted to use a pointer, but there's no statement saying that treatment applies to return types as they're not allowed. Your explanation makes it sound like the treatment for parameters is applied to returned types and yields a then-invalid signature. That's not so - plain and simple, array return types just aren't allowed: 8.3.5.8 "Functions shall not have a return type of type array or function". – Tony Delroy Apr 12, 2013 at 2:58

@TonyD : I think his explanation is good, and better than the accepted answer. the std::vector /array stuff at the end is a digression though. (because this is not the same semantic to use RVO/copy ellision and return value-semantics things, than return what you'd expect would be a pointer to a C-array, because of a well assimilated "decay to pointer" concept by every beginner in C. since its one of the first things learnt) – v.oddou Apr 23, 2015 at 5:32

8

> "You can't return array from the function because that array would be declared inside the function, and its location would then be the stack frame. However, stack frame is erased when function exits. Functions must copy return value from stack frame to return location, and that's not possible with arrays."

From a discussion here:

http://forum.codecall.net/c-c/32457-function-return-array-c.html

Share Edit Follow Flag

edited Mar 1, 2011 at 19:53
Evan Teran
84.1k • 29 • 174 • 237

answered Mar 1, 2011 at 16:40
Brandon Frohbieter
16.8k • 3 • 36 • 61

1 Downvote for copying verbatim from the link you're referencing. In addition, this answer is misleading. In particular "Functions must copy return value [sic]" is technically false, since functions can return references and pointers. – phooji Mar 1, 2011 at 17:03

7 I don't see a problem with the quote, linked the reference. – Brandon Frohbieter Mar 1, 2011 at 17:05

1 @phooji: references and pointers are both pointers, which are themselves both values. There's nothing misleading if you understand what a pointer is. – Inverse Mar 1, 2011 at 17:24

@Orbit: If you want your answer to look like you're quoting, then use quotation marks around "You can't return [...] with arrays" (see what I did there? :) Just adding the link is not enough because someone might still claim that you 'stole' their text; with the quotes it is clear that you're using someone else's text. – phooji Mar 1, 2011 at 17:31

9 I cannot agree with this answer. For most other types you can return by value and there is no problem with the fact that the returned object is inside the function: a copy is made (or elided if the compiler manages to do so). That is a common behavior and the fact that the same cannot be done with arrays is more of a design decision in the C language --inherited in C++. As a matter of fact, if you enclose the array in a struct, that is exactly what would happen: the struct (including the internal array) will be copied in the return statement. – David Rodríguez - dribeas Mar 1, 2011 at 17:39

7

Other have said that in C++, one use vector<> instead of the arrays inherited from C.

So why C++ doesn't allows to returns C arrays? Because C doesn't.

Why C doesn't? Because C evolved from B, a untyped language in which returning an array doesn't make sense at all. When adding types to B, it would have been meaningful to make it possible to return an array but that wasn't done in order to keep some B idioms valid and ease the conversion of programs from B to C. And since then, the possibility of making C arrays more usable as always been refused (and even more, not even considered) as it would break too much existing code.

Share Edit Follow Flag

edited Mar 1, 2011 at 17:58

answered Mar 1, 2011 at 16:55
AProgrammer
49.5k • 8 • 87 • 140

"making C arrays more usable... would break too much existing code" - not true. Existing progarms won't have compiled if they included functions returning arrays, so such features would only be relevant to new code choosing to use those functions and in no way invalidate existing code. Put another way, you're not postulating a change of existing behaviour, rather - it'd be new independent behaviour. – Tony Delroy Apr 12, 2013 at 3:03

@TonyD, you'd need either to remove the automatic decay of an array to a pointer, and that will break lot of code, or make so many special cases that you haven't make C arrays more usable at all, or change so few things that it won't worth the pain. – AProgrammer Apr 12, 2013 at 6:42

interesting assertion. Please help me understand your specific concerns. For context, consider `int[4] f() { int x[4]; ...populate x...; return x; }` and to make that useful in an intuitive way, let's add a requirement for new support of assignment to arrays both in the return and ala `int x[4] = f();` . I don't see how any of this would require pointer decay, nor need to change other code to prevent pointer decay. What kind of code do you see conflicting with this? – Tony Delroy Apr 12, 2013 at 7:55

@tonyd, if you don't change the current rules the result of f() would decay into a pointer (just like with `int (*p)[4]` . `*p` decays into a pointer). – AProgrammer Apr 12, 2013 at 8:13

But when would it decay? - it only decays if the assignment isn't possible with the original type. Much like `long x = get_char();` - the conversion to `long` is only attempted because the rhs operand to the assignment isn't already a `long` . So, what we're talking about is not some suppression of the pointer decay, but having something new work before it's ever considered. "(just like with int (*p)[4], *p decays into a pointer)" - not so, `*p` is still `int[4]` - confirmed by passing to `template <int N> void f(int (&a)[N]) { std::cout << N << '\n'; }` . The decay is last resort. – Tony Delroy Apr 12, 2013 at 9:52

3

You can return a pointer to the array. Just be careful about releasing the memory later.

```
public std::string* funcarray() {
    std::string* test = new std::string[2];
    test[0] = "hi";
    test[1] = "hello";
    return test;
}

// somewhere else:
std::string* arr = funcarray();
std::cout << arr[0] << " MisterSir" << std::endl;
delete[] arr;
```

Or you can just use one of the containers in the std namespace, like std::vector.

Share Edit Follow Flag

answered Mar 1, 2011 at 16:45
Jordi
5,738 • 10 • 39 • 40

Shouldn't I delete std::string* test too? – Lockhead Mar 1, 2011 at 16:58

1 @MisterSir - No, there is no need. `test` is a variable residing on stack and goes out of scope on function return. However, the location `test` was pointing resides on heap/free store and is returned to `arr` . So, if you delete `arr` , that's sufficient. – Mahesh Mar 1, 2011 at 17:42

2

"Why doesn't C++ support something like": Because it would not make any sense. In reference-based languages like JAVA or PHP, memory management is based on garbage collection. The portions of memory which have no references (no variable in your program points to it any more) is automatically freed. In this context you can allocate memory, and pass the reference around carefreely.

C++ code will be translated to machine code, and there is no GC defined in it. So in C and C++ there is a strong sense of **ownership** of memory blocks. You have to know if the pointer you go is yours to free at any time (in fact you *shoud* free it after use), or you have a pointer to a shared portion of memory, which is an absolute no-no to free.

In this environment you would win nothing with cretaing endless copies of an array every time it passes to and from a function. It is amuch more complex task to manage your arrays of data in c-like languages. There is no one-size-fits-all solution, and you need to know when to free memory.

Would an array returned by a function always be a copy (yours to free) or you have to make copies of them? Whet would you win by getting an array insted of a pointer to an array?

Share Edit Follow Flag

answered Mar 1, 2011 at 17:01
vbence
19.7k • 8 • 64 • 114

Why wouldn't it make sense to return arrays? C++ had to invent `std::array` in part to get over this arcane limitation. This has nothing to do with GC or references. C++ allows you to return objects by value (as does C, in fact.) Just not plain arrays. What doesn't make any sense is your answer. – juanchopanza Apr 23, 2015 at 5:38

I think the fundamental issue is that if a method is going to return something by value, space for the thing in question has to be reserved *before the method is called*. Since arrays of fixed size can be encapsulated in structures for that purposes, and since such structures behave more consistently and usefully than arrays types, there's little benefit to returning fixed-size array types. Arrays of variable size might be nice in some cases, but there's no reasonable mechanism for the caller to provide space for them. – supercat Apr 30, 2015 at 20:06

1

Return a `std::vector<>` instead of an array. In general, arrays do not work well with C++, and should generally be avoided.

Also, the `string` datatype is not just an array of characters, although a "quoted string" is. The `string` manages an array of characters, and you can get access to it with `.c_str()`, but there's more to a `string` than that.

Share Edit Follow Flag

answered Mar 1, 2011 at 16:43
David Thornley
55.4k • 8 • 89 • 155

1

Check out here. Really helpful.

- How do I return an array from a function?
- C++ Returning multidimension array from function
- Return 2d array from function in C++

Share Edit Follow Flag

edited May 23, 2017 at 12:00
Community Bot
1 • 1

answered Mar 1, 2011 at 16:45
Saurabh Gokhale
51.8k • 34 • 134 • 163

0

These answers are all missing the point. C++ just doesn't support it. It didn't even support a way to return a statically-sized array before `std::array<T, N>`. C++ *could* support returning even dynamically-sized arrays, but they don't. I'm sure there are defensible reasons why, but they could.

All you need to do is allocate the dynamic array on the stack, return the address and size of it, and make sure the caller bumps the stack pointer up to the end of the returned array. Possibly some stack frame fixing to do, but by no means impossible.

Share Edit Follow Flag

answered Jun 4, 2020 at 1:42
Nick Strupat
4,737 • 4 • 41 • 56