std::vector versus std::array in C++

What are the difference between a `std::vector` and an `std::array` in C++? When should one be preferred over another? What are the pros and cons of each? All my textbook does is list how they are the same.

c++ arrays vector

edited Aug 21 at 9:48
Martin
2,713 4 15 33

asked Dec 12 '10 at 22:55
Zud
710 1 9 23

2 Are you talking about the usage of the terms vector and array, or are you asking for a comparison of `std::vector` vs `std::array` (which is new to C++0x), or of `std::vector` vs C-style arrays? – Matteo Italia Dec 12 '10 at 23:01

I'm looking for a comparison of `std::vector` vs. `std::array` and how the terms are different. – Zud Dec 12 '10 at 23:02

Zud, `std::array` is not the same as a C++ array. `std::array` is a very thin wrapper around C++ arrays, with the primary purpose of hiding the pointer from the user of the class. I will update my answer. – ClosureCowboy Dec 12 '10 at 23:19

I updated the question title and text to reflect your clarification. – Matteo Italia Dec 12 '10 at 23:32

6 Answers

`std::vector` is a template class that encapsulate a dynamic array[1], stored in the heap, that grows and shrinks automatically if elements are added or removed. It provides all the hooks (`begin()` , `end()` , iterators, etc) that make it work fine with the rest of the STL. It also has several useful methods that let you perform operations that on a normal array would be cumbersome, like e.g. inserting elements in the middle of a vector (it handles all the work of moving the following elements behind the scenes).

Since it stores the elements in memory allocated on the heap, it has some overhead in respect to static arrays.

`std::array` is a template class that encapsulate a statically-sized array, stored inside the object itself, which means that, if you instantiate the class on the stack, the array itself will be on the stack. Its size has to be known at compile time (it's passed as a template parameter), and it cannot grow or shrink.

It's more limited than `std::vector` , but it's often more efficient, especially for small sizes, because in practice it's mostly a lightweight wrapper around a C-style array. However, it's more secure, since the implicit conversion to pointer is disabled, and it provides much of the STL-related functionality of `std::vector` and of the other containers, so you can use it easily with STL algorithms & co. Anyhow, for the very limitation of fixed size it's much less flexible than `std::vector` .

For an introduction to `std::array` , have a look at this article; for a quick introduction to `std::vector` and to the the operations that are possible on it, you may want to look at its documentation.

1. ~~Actually, I think that in the standard they are described in terms of maximum complexity of the different operations (e.g. random access in constant time, iteration over all the elements in linear time, add and removal of elements at the end in constant amortized time, etc), but AFAIK there's no other method of fulfilling such requirements other than using a dynamic array.~~ As stated by @Lucretiel, the standard actually requires that the elements are stored contiguously, so *it is* a dynamic array, stored where the associated allocator puts it.

edited Oct 21 '14 at 11:56

answered Dec 12 '10 at 23:13
Matteo Italia
60.3k 7 69 148

3 Regarding your footnote: While true, the standard also guarentees that pointer arithmetic on the internal elements works, which means that it does have to be an array: &vec[9] - &vec[3] == 6 is true. – Lucretiel Mar 12 '12 at 6:17

1 I'm pretty sure, that vector doesn't shrink automatically, but since C++11 you can call shrink_to_fit. – Dino Sep 9 '14 at 9:24

I'm totally confused by the term *static array* and I'm not sure what the right terminology is. You mean a static size array and not an static variable array (one using static storage). stackoverflow.com/questions/2672085/…. What is the correct terminology? Is static array a sloppy term for an array with a fixed size? – Z boson Oct 21 '14 at 11:30

@Zboson: I mean a statically-*sized* array; sorry, I agree that the terminology is confusing, I'll fix it immediately. – Matteo Italia Oct 21 '14 at 11:55

1 @Zboson: it's definitely not just you, *static* is quite an abused term; the very `static` keyword in C++ has three different unrelated meanings, and the term is also used often to talk about stuff that is fixed at compile time. I hope that "statically-sized" is a bit more clear. – Matteo Italia Oct 21 '14 at 12:55

Using the `std::vector<T>` class:

- ...is *just as fast* as using built-in arrays, assuming you are doing only the things built-in arrays allow you to do (read and write to existing elements).
- ...automatically resizes when new elements are inserted.
- ...allows you to insert new elements *at the beginning* or *in the middle* of the vector, automatically "shifting" the rest of the elements "up"(does that make sense?). It allows you to remove elements anywhere in the `std::vector`, too, automatically shifting the rest of the elements down.
- ...allows you to perform a range-checked read with the `at()` method (you can always use the indexers `[]` if you don't want this check to be performed).

There are ~~two~~ three main caveats to using `std::vector<T>`:

1. You don't have reliable access to the underlying pointer, which *may* be an issue if you are dealing with third-party functions that demand the address of an array.
2. The `std::vector<bool>` class is silly. It's implemented as a condensed bitfield, not as an array. Avoid it if you want an array of `bool` s!
3. During usage, `std::vector<T>` s are going to be a bit larger than a C++ array with the same number of elements. This is because they need to keep track of a small amount of other information, such as their current size, and because whenever `std::vector<T>` s resize, they reserve more space then they need. This is to prevent them from having to resize every time a new element is inserted. This behavior can be changed by providing a custom `allocator`, but I never felt the need to do that!

Edit: After reading Zud's reply to the question, I felt I should add this:

The `std::array<T>` class is not the same as a C++ array. `std::array<T>` is a very thin wrapper around C++ arrays, with the primary purpose of hiding the pointer from the user of the class (in C++, arrays are implicitly cast as pointers, often to dismaying effect). The `std::array<T>` class also stores its size (length), which can be very useful.

edited Dec 12 '10 at 23:32 answered Dec 12 '10 at 23:14
ClosureCowboy
5,973 5 35 63

2 It's 'just as fast" as using a dynamically-allocated built-in array. On the other hand, using an automatic array might have considerably different performance (and not only during allocation, because of locality effects). – Ben Voigt Aug 27 '13 at 18:13

For non-bool vectors in C++11 and later, you can call `data()` on a `std::vector<T>` to get the underlying pointer. You can also just take the address of element 0 (guaranteed to work with C++11, will probably work with earlier versions). – Matt Jun 25 at 18:01

What a vector is actually a wrapper around an array. Arrays are not objects but rather a representation of allocated memory. A vector will allocate a fixed amount of memory (an array), allowing you to add and remove elements easily. Once you have added enough elements that the array within the vector is full, it will create a new array of twice the size - moving every element from the old array into the new one. This is all done behind the scenes within the Vector class.

Use arrays whenever you know that you have a fixed length and do not intend to change that length. Otherwise, use a vector. A vector has the advantage of having convenient methods such as adding and removing elements with ease. However, when your vector hits its maximum size, it will have to expand its size (by 2) and move all of the elements over. This is an O(n) operation, where n is the number of elements in your vector (kind of slow).

In addition, your vector will always have unused space. While usually this doesn't matter, it is an advantage of an array with a fixed length.

edited Aug 21 at 9:43
Martin
2,713 4 15 33

answered Dec 12 '10 at 22:59
Ian Bishop
3,638 1 17 33

A vector is a container class while an array is an allocated memory.

answered Dec 12 '10 at 22:58
Saif al Harthi
2,264 1 10 20

1 Your answer seems to address `std::vector<T>` versus `T[]`, but the question is about `std::vector<T>` versus `std::array<T>`. – Keith Pinson Jan 23 '13 at 16:36

To emphasize a point made by @MatteoItalia, the efficiency difference is where the data is stored. Heap memory (required with `vector`) requires a call to the system to allocate memory and this can be expensive if you are counting cycles. Stack memory (possible for `array`) is virtually "zero-overhead" in terms of time, because the memory is allocated by just adjusting the stack pointer and it is done just once on entry to a function. The stack also avoids memory fragmentation. To be sure, `std::array` won't always be on the stack; it depends on where you allocate it, but it will still involve one less memory allocation from the heap compared to vector. If you have a

- small "array" (under 100 elements say) - (a typical stack is about 8MB, so don't allocate more than a few KB on the stack or less if your code is recursive)
- the size will be fixed
- the lifetime is in the function scope (or is a member value with the same lifetime as the parent class)
- you are counting cycles,

definitely use a `std::array` over a vector. If any of those requirements is not true, then use a `std::vector`.

edited Jun 29 at 18:21

answered Jun 29 at 18:16
Mark Lakata
7,371 2 37 59

One of the advantages that vectors have over arrays is that it is possible to find the ***current size*** of a vector using **vector_name.size()**.

As you can imagine, this can be quite useful in a variety of situations, where you can fetch number of elements in the array_list easily.

edited Aug 21 at 9:46
Martin
2,713 4 15 33

answered Jul 3 '14 at 7:50
tech_boy
13 7

3 You can also do that with std::array – Gerard Oct 1 '14 at 16:20