c++11 std::array vs static array vs std::vector

First question, is it a good thing to start using c++11 if I will develop a code for the 3 following years?

Then if it is, what is the "best" way to implement a matrix if I want to use it with Lapack? I mean, doing `std::vector<std::vector< Type > > Matrix` is not easily compatible with Lapack.

Up to now, I stored my matrix with `Type* Matrix(new Type[N])` (the pointer form with `new` and `delete` were important because the size of the array is not given as a number like 5, but as a variable).

But with C++11 it is possible to use std::array. According to this site, this container seems to be the best solution... What do you think?

c++ arrays c++11 matrix lapack

edited Sep 18 '13 at 9:59

Bartek Banachewicz
17.4k 3 34 68

asked Sep 18 '13 at 9:42

PinkFloyd
446 1 6 15

2 One question at a time please. – Kerrek SB Sep 18 '13 at 9:43

@KerrekSB : well, I know but the first question is a yes/no question... – PinkFloyd Sep 18 '13 at 9:47

2 gist.github.com/rmartinho/3959961 – R. Martinho Fernandes Sep 18 '13 at 10:12

@PinkFloyd: The first question would have got closed immediately for being "primarily opinion-based", and the second technical question would have got more attention. Mixing the two leaves you with an offputting, mediocre question that people might just skip over. – Kerrek SB Sep 18 '13 at 10:15

@R.MartinhoFernandes I actually wrote more or less exactly the same code for my matrix with c++03. but i have trouble to sort it. this is why i considered using c++11 and maybe use c++11 std::array instead of a pointer – PinkFloyd Sep 18 '13 at 10:17

2 Answers

First things first, if you are going to learn C++, learn C++11. The previous C++ standard was released in 2003, meaning it's already *ten* years old. That's a lot in IT world. C++11 skills will also smoothly translate to upcoming C++1y (most probably C++14) standard.

The main difference between `std::vector` and `std::array` is the dynamic (in size and allocation) and static storage. So if you want to have a matrix class that's always, say, 4x4, `std::array<float, 4*4>` will do just fine.

Both of these classes provide `.data()` member, which should produce a compatible pointer. Note however, that `std::vector<std::vector<float>>` will NOT occuppy contiguous `16*sizeof(float)` memory (so `v[0].data()` **won't** work). If you need a dynamically sized matrix, use single `vector` and resize it to the `width*height` size.

Since the access to the elements will be a bit harder (`v[width * y +x]` or `v[height * x + y]`), you might want to provide a wrapper class that will allow you to access arbitrary field by row/column pair.

Since you've also mentioned C-style arrays; `std::array` provides nicer interface to deal with the same type of storage, and thus should be preferred; there's nothing to gain with static arrays over `std::array` .

edited Sep 18 '13 at 10:22

answered Sep 18 '13 at 9:55

Bartek Banachewicz
17.4k 3 34 68

Thx for your complete answer. But your last remark seems to disagree with "The struct combines the performance and accessibility of a C-style array with the benefits of a standard container, such as knowing its own size, supporting assignment, random access iterators, etc." (see the link i gave i my question) – PinkFloyd Sep 18 '13 at 10:12

How does it disagree, exactly? I think it spells pretty much the same thing. – Bartek Banachewicz Sep 18

'13 at 10:21

it seems to me that they are implying that other containers (namely `std::vector`) are not as efficient a the C-style array, but `std::array` is... but i don't know enough to compare the efficiency of C-style array, `std::vector` and `std::array`. – PinkFloyd Sep 18 '13 at 10:33

4 `std::vector` has (a very slight) performance impact due to its dynamic nature; because the size of the required memory is not known at compile time, the OS memory allocation call has to be made. C-style arrays and `std::array` are OTOH of a fixed size and thus they can be allocated on stack without the need to ask OS for more memory. The access to elements is equally fast in both approaches and boils down to simple pointer arithmetic. If you are *that* serious about performance, you should probably be using SIMD instructions and types to make use of HW-accelerated calculations. – Bartek Banachewicz Sep 18 '13 at 10:37

1 If a `std::array` is allocated statically (i.e. in the data section; by declaring it outside of any function or with the `static` keyword inside a function) then won't its address be known at compile time, allowing some optimisations (i.e. no need to dereference a pointer)? So it should be faster? – Evgeni Sergeev Dec 26 '13 at 4:09

This is a very late reply to the question, but if someone reads this, I just want to point out that one should almost never implement a matrix as a "vector of vectors". The reason is that each row of the matrix gets stored in some random location on the heap. This means that matrix operations will do a lot of random memory accesses leading to cache misses, which slows down the implementation considerably.

In other words, if you care at all about performance, just allocate an `array/std::array/std::vector` of size `rows * columns`, then use wrapper functions that transforms a pair of integers to the corresponding element in the array. Unless you need to support things like returning references to rows of the matrix, then all of these options should work just fine.

answered Jul 12 at 19:53

Edvard Fagerholm
349 1 9

thx for your advice but I've already done it this way :-) – PinkFloyd Jul 13 at 6:52

@DOUGLASO.MOEN This depends entirely on the degree of contention for the heap. If you just allocated a vector of vectors straightforwardly, without anything else attempting an allocation (or releasing one), the buffers may well end up one right after the other. – defube Jul 20 at 15:36