std::vector::reserve performance penalty

Asked 10 years, 11 months ago Active 10 years, 11 months ago Viewed 5k times

8

```
inline void add(const DataStruct& rhs) {
   using namespace boost::assign;
   vec.reserve(vec.size() + 3);
   vec += rhs.a, rhs.b, rhs.c;
}
```

2

The above function was executed for about 17000 times and it performed (as far as I can see. There was some transformation involved) about 2 magnitudes worse **with** the call to vector::reserve.

I always was under the impression that reserve can speed up push_back even for small values but this doesn't seem true and I can't find any obvious reasons why it shouldn't be this way. Does reserve prevent the inlining of the function? Is the call to size() too expensive? Does this depend on the platform? I'll try and code up some small benchmark to confirm this in a clean environment.

Compiler: `gcc (GCC) 4.4.2` with -g -O2

c++ performance stl stdvector

asked Nov 16 '09 at 15:23

pmr
52.7k 9 98 144

1 Have you tried reserving space for 17000*3 inputs? There may be some overhead with your extra function call, which could account for the difference. – int3 Nov 16 '09 at 15:27

@splicer: The number was due to the testdata. The actual number of calls is variable. – pmr Nov 16 '09 at 15:44

my point was that what you're doing is only efficient with larger numbers. James Schek's answer gives you a way to do it with a variable number of calls, so long as you know the total to start with. otherwise you're better off letting the default implementation handle it for you. – int3 Nov 16 '09 at 15:51

Are you calling add() on the same class instance? In that case you are growing vec by 3 for each add whereas pushback by itself would grow vec far less frequently. – Patrick Nov 16 '09 at 16:02

6 Answers

Active | Oldest | Votes

26

GCC implementation of `reserve()` will allocate exact number of elements, while `push_back()` will grow internal buffer exponentially by doubling it, so you are defeating the exponential growth and forcing reallocation/copy on each iteration. Run your test under `ltrace` or `valgrind` and see the number of `malloc()` calls.

answered Nov 16 '09 at 15:38

Nikolai Fetissov
75.9k 11 102 163

1 +1. I have just checked GCC sources and been going to write the same. – P Shved Nov 16 '09 at 15:41

Is the exponential growth of the buffer through push_back() guaranteed by the standard or implementation defined? – pmr Nov 16 '09 at 18:04

No, exponential growth is not guaranteed by the standard, it's an implementation detail, same as this particular behavior of reserve(). – Nikolai Fetissov Nov 16 '09 at 18:54

9 Exponential growth IS effectively guaranteed by the standard, as it's the only reasonable way to get amortized O(1) behavior for push_back (unreasonable solutions include allocating all memory ;)) – MSalters Nov 17 '09 at 9:41

1 The behavior of GCC can be explained too: by letting you set exactly the expected size, they allow you the opportunity of not wasting memory in cases where you know actually how much you'll need. – Matthieu M. Nov 17 '09 at 10:14

7

You only use `reserve()` if you know in advance the number of elements. In that case `reserve()` space for all elements at once.

Otherwise just use `push_back()` and rely on the default strategy - it will reallocate exponentially and greatly reduce the number of reallocations at a cost of slightly suboptimal memory consumption.

answered Nov 16 '09 at 15:30

sharptooth
157k 79 466 895

Use only reserve if you know in advance how much place it will use.

7

Reserve will need to copy your whole vector...

If you do a push_back and the vector is too small, then it will do a reserve (vec.size()*2).

If you don't know beforehand how big your vector is going to be and if you need random access, consider using std::deque.

edited Nov 16 '09 at 15:41

answered Nov 16 '09 at 15:26

Tristram Gräbener

9,041 3 29 46

It's not an heuristic. Is proved that on average the insertion of a new element is O(1). Read any book on amortized analysis (eg. Introduction to Algorithms). – Alexandru Nov 16 '09 at 15:37

1 well.. I think it would be fair to say that it's a heuristic that mathematically proven to have the best performance in the average case. – int3 Nov 16 '09 at 15:40

You can prove the same with multiplying by 3 instead of two. But I'll edit it anyways if it bothers you that much. – Tristram Gräbener Nov 16 '09 at 15:41

1 I would say it is neither a heurstic nor an algorithm. It is a strategy that works well for a lot of usage scenarios. – Dolphin Nov 16 '09 at 15:49

3 Multiplying the current allocated size by any constant (besides 0) will result in logarithmic growth. Changing the constant is just tuning, Big O performance is still the same. 1.5 favors smaller vectors, 2 favors larger. They chose 2 because it is a nice round number in computer land, and any tuning would be meaningless as my usage will differ from yours. If you need tune the behavior you can easily subvert the default behavior using size() and reserve(). – deft_code Nov 16 '09 at 16:19

5

When std::vector needs to reallocate it grows its allocation size by N*2, where n is its current size. This results in a logarithmic number of reallocs as the vector grows.

If instead the std::vector grew its allocated space by a constant amount, the number of reallocs would grow linearly as the vector grows.

What you've done is essentially cause the vector to grow by a constant amount 3, meaning linear growth. Linear is obviously worse that logarithmic, especially with big numbers.

Generally, the only growth better than logarithmic is constant. That is why the standards committee created the reserve method. If you can avoid all reallocs (constant) you will perform better than the default logarithmic behavior.

That said you may want to consider Herb Sutter's comments about preferring std::deque over vector www.gotw.ca/gotw/054.htm

answered Nov 16 '09 at 16:12

deft_code
49.6k 26 133 211

3

Move the reserve outside of the add.

Each time you invoke "add", you are reserving atleast 3 extra elements. Depending on the implementation of vector, this *could* be increasing the size of the backing array almost every time you call "add". That is would definately cause the performance difference that you describe.

The correct way to use reserve is something like:

```
vec.reserve(max*3);
for(int i=0; i<max; i++)
   add(i);
```

answered Nov 16 '09 at 15:32

James Schek
17.1k 7 46 64

3

If you profiled the code I bet you would see that the += IS very fast, the problem is the reserve is killing you. You should really only use reserve when you have some knowledge of how big the vector will grow to. If you can guess ahead of time then do ONE reserve, otherwise just go with the default push_back.

answered Nov 16 '09 at 15:38

Dolphin

4,472 1 27 25