C++ Made Easier: How Vectors Grow

By Andrew Koenig and Barbara E. Moo, April 01, 2001

Introduction

Suppose we want to read a bunch of values of type **double** from a file into a data structure that will allow us to get at the values efficiently. The usual way to do so is something like this:

```
vector<double> values;
double x;

while (cin >> x)
   values.push_back(x);
```

When this loop completes, **values** will hold all the values, and we will be able to access any of those values efficiently by writing **values[i]** with a suitable value for **i**.

Intuitively, the standard-library **vector** class is like a built-in array: we can think of it as holding its elements in a single block of contiguous memory. Indeed, although the C++ Standard does not explicitly require that the elements of a **vector** occupy contiguous memory, the standards committee decided in its October, 2000 meeting that this requirement was missing due to an oversight, and voted to include the requirement as part of its Technical Corrigendum. This delayed imposition was not a particular hardship, because every existing implementation already worked that way.

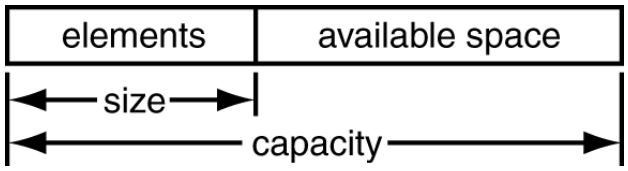
If the elements of a vector are in contiguous memory, it is easy to see

how the implementaiton can access an individual element efficiently — it simply uses the same mechanism as the built-in arrays use. What is not so easy to see is how an implementation can arrange for a **vector** to grow efficiently, when growth must inevitably involve copying elements from one area of memory to another. Although modern processors are often particularly good at copying contiguous blocks of data from one part of memory to another, such copies are not free. Therefore, it is useful to think about how a standard-library implementation might be able to arrange for a **vector** to grow incrementally without consuming inordinate amounts of time or space.

The rest of this article will discuss a simple, efficient strategy for managing such growth.

Size and Capacity

The first thing to realize about how the **vector** class works is that a **vector** is more than just a block of memory. Instead, every **vector** has not one but two sizes associated with it. One, called its size, is the number of elements that it contains. The other, called its capacity, is the total amount of memory that is available for storing elements. If **v** is a **vector**, then **v.size()** and **v.capacity()** return **v**'s size and capacity, respectively. You might imagine a vector looking like this:



The point of having this extra memory at the end of the **vector**, of course, is to let **push_back** append an element to the **vector** without having to allocate more memory. If the memory that happens to be adjacent to the end of the **vector** is presently unoccupied, it might be possible to make the **vector** grow simply by annexing that memory to the **vector**. However, such luck is rare: most of the time, it will be necessary

to allocate new memory, copy all of the **vector**'s present elements into that memory, destroy those elements, and deallocate the memory that the elements formerly occupied. The advantage of having extra memory in a **vector** is that such reallocation, which is likely to be expensive, doesn't happen every time one tries to append an element to a **vector**.

How expensive is reallocation? It involves four steps:

1. Allocate enough memory for the desired new capacity;
2. Copy the elements from the old memory to the new;
3. Destroy the elements in the old memory; and
4. Deallocate the old memory.

If there are *n* elements, we know that steps (2) and (3) each take *O(n)* time. Unless the price of allocating or deallocating a block of memory grows more quickly than *O(n),* these steps will dominate the overall execution time. Therefore, we can conclude that reallocating a **vector** with a size of *n* takes *O(n)* time — regardless of the capacity that we use for the reallocation.

This conclusion suggests a tradeoff. If when we reallocate, we ask for a lot of extra memory, we will not have to reallocate again for quite a while, so reallocations will consume relatively less time. The cost of this strategy will be a lot of wasted space. On the other hand, we can ask for only a little extra memory. Doing so will conserve space, but only by spending time in extra reallocations. In other words, we have a classic opportunity to trade time for space, and vice versa.

Reallocation Strategy

As an example of one extreme, let's assume that every time we fill the **vector**, we increase its capacity by one. This strategy wastes as little space as possible, but reallocates the entire **vector** each time we append an element. We said that reallocating an *n*-element **vector** takes *O(n)*

time, so if we start with an empty **vector** and grow it to *k* elements, the total time will be *O(1+2+...+k),* or *O(k2)*. That's terrible! Can we do better?

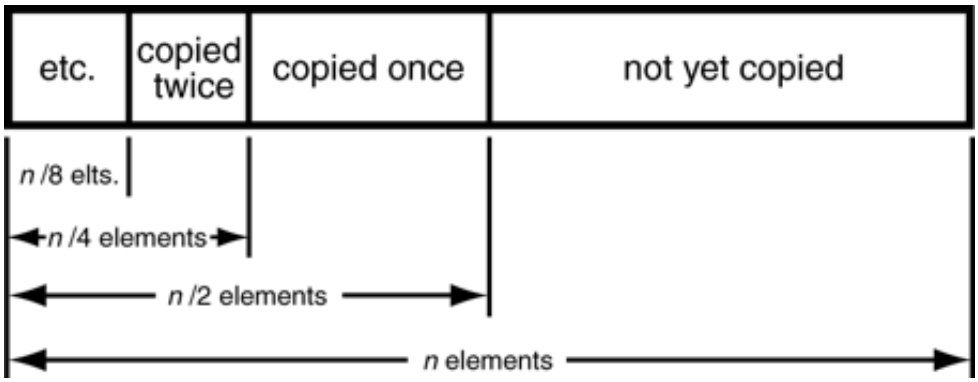
Suppose, for example, that instead of increasing the **vector**'s capacity by one, we increase it by a constant *C*. It should be clear that this strategy will reduce the number of reallocations by a factor of *C,* so it is certainly an improvement. But how much of an improvement?

One way to understand the improvement is to realize that with this new strategy, a reallocation happens for every chunk of *C* elements. Suppose, then, that we allocate *K* chunks, for a total of *K*x*C* elements. Then the first reallocation will copy *C* elements, the second will copy 2x*C* elements, and so on. Big-O notation ignores constant factors, so we can divide out all the factors of *C* to obtain a total time of *O(1+2+...+K),* or *O(K2)*. In other words, the time is still quadratic in the number of elements, but with a much smaller factor.

Despite the smaller factor, quadratic behavior is still bad, even with a fast processor. In fact, it is especially bad with a fast processor, because fast processors usually come with lots of memory. Programmers with access to fast processors with lots of memory usually try to use all that memory sooner or later, and a programmer who does so is apt to discover that the processor speed is to no avail if there is a quadratic algorithm around.

We have just shown that an implementation that wants to be able to allocate large **vector**s in less than quadratic time cannot use a strategy of increasing the **vector**'s capacity by a constant each time it fills up. Instead, the amount of additional memory allocated must grow each time the **vector** grows. This fact suggests a simple strategy: how about starting the **vector** out with a single element, and then doubling its capacity each time we reallocate? Remarkably, it turns out that this strategy allows us to build up an *n*-element **vector** in *O(n)* time.

To understand how such efficiency is possible, consider the state of a **vector** when we have completely filled it up and are about to reallocate it:



Half of the elements were appended to the **vector** since the last reallocation, so they have never been copied. Of the ones that have been copied, half of those have been copied exactly once, half of the remainder have been copied twice, and so on.

In other words, *n/2* elements have been copied one or more times, *n/4* elements have been copied two or more times, and so on. Therefore, the total number of copies that have been made up to this point is *n/2 + n/4 +...,* which is approximately *n*. (This approximation becomes more accurate as *n* increases.) In addition to the copies, *n* elements have been appended to the **vector**, but the total number of operations is still *O(n),* not *O(n2)*.

Discussion

The C++ Standard does not mandate that the **vector** class manage its memory in any particular way. What it does is to require that creating an *n*-element **vector** through repeated calls to **push_back** take no longer than *O(n)* time. The strategy that we have been discussing is probably the most straightforward that is capable of meeting that requirement.

Because **vector**s have good time performance for such operations, there is no reason to avoid loops such as

```
vector<double> values;
double x;
while (cin >> x)
   values.push_back(x);
```

Yes, the implementation will reallocate the **vector**'s elements as it grows, but this reallocation will take no more than a constant factor longer than it would have taken if we had been able to predict the ultimate size of the **vector** in advance.

Exercises

1. Suppose we tried to make our little loop faster by writing it this way:

```
while (cin >> x) {
   if (values.size() ==
       values.capacity())
       values.reserve(values.size() +
          1000);
   values.push_back(x); }
```

What would be the effect? The **reserve** member function does a reallocation, thereby changing the **vector**'s capacity to be greater than or equal to its argument.

2. Suppose that instead of doubling the size of the **vector** each time, we tripled it. What would be the effect on performance? In particular, would the execution time still be *O(n)* for creating an *n*-element **vector**?

3. Suppose you know how many elements your **vector** is ultimately going to have. In that case, you can call **reserve** to preallocate the right amount of memory before you begin filling it. Experiment with your local implementation to find out how much of a difference calling **reserve** makes in your programs' execution time.

Andrew Koenig *is a member of the Large-Scale Programming Research Department at AT&T's Shannon Laboratory, and the Project Editor of the C++ standards committee. A programmer for more than 30 years, 15 of them in C++, he has published more than 150 articles about C++ and speaks on the topic worldwide. He is the author of C Traps and Pitfalls and co-author of Ruminations on C++.*

Barbara E. Moo *is an independent consultant with 20 years' experience in the software field. During her nearly 15 years at AT&T, she worked on one of the first commercial projects ever written in C++, managed the company's first C++ compiler project, and directed the development of AT&T's award-winning WorldNet Internet service business. She is co-author of Ruminations on C++ and lectures worldwide.*