An In-Depth Study of the STL Deque Container

Nitron

11 Nov 2003

This article presents an in-depth analysis of std::deque and offers guidance as to when to prefer using it as opposed to std::vector, by taking into consideration memory allocation and container performance.

Introduction

This article presents an in-depth look at the STL deque container. This article will discuss the benefits of the deque and under what circumstances you would use it instead of the vector. After reading this article, the reader should be able to explain the fundamental differences between vector and deque with respect to container growth, performance and memory allocation. Since deque is so similar in usage and syntax to vector, it is recommended that the reader refers to this article [^] on vector for information on implementing and using this STL container.

Deque Overview

The deque, like the vector, is also part of the Standard Template Library, the STL. The deque, or "double-ended queue", is very similar to the vector on the surface, and can act as a direct replacement in many implementations. Since it is assumed that the reader already knows how to effectively use the STL vector container, I have provided the table of deque member functions and operators below, solely for comparison and reference.

Deque Member Functions[1]

Function	Description
assign	Erases elements from a deque and copies a new set of elements to the target deque.
at	Returns a reference to the element at a specified location in the deque.
back	Returns a reference to the last element of the deque.

begin	Returns an iterator addressing the first element in the deque.
clear	Erases all the elements of a deque.
deque	Constructs a deque of a specific size or with elements of a specific value or with a specific allocator or as a copy of all or part of some other deque.
empty	Tests if a deque is empty.
end	Returns an iterator that addresses the location succeeding the last element in a deque.
erase	Removes an element or a range of elements in a deque from specified positions.
front	Returns a reference to the first element in a deque.
get_allocator	Returns a copy of the allocator object used to construct the deque.
insert	Inserts an element or a number of elements or a range of elements into the deque at a specified position.
max_size	Returns the maximum length of the deque.
pop_back	Deletes the element at the end of the deque.
pop_front	Deletes the element at the beginning of the deque.
push_back	Adds an element to the end of the deque.
push_front	Adds an element to the beginning of the deque.
rbegin	Returns an iterator to the first element in a reversed deque.
rend	Returns an iterator that points just beyond the last element in a reversed deque.
resize	Specifies a new size for a deque.
size	Returns the number of elements in the deque.
swap	Exchanges the elements of two deques.

Deque Operators[1]

Function	**Description**
operator[]	Returns a reference to the deque element at a specified position.

This striking similarity to vector then gives rise to the following question:

Q: If deque and vector offer such similar functionality, when is it preferable to use one over the other?

A: If you have to ask, use vector.

Um, okay... How about a little explanation please?

Why certainly, I'm glad you asked! I didn't pull that answer out of thin air, in fact, the answer comes from the C++ Standard[2] itself. Section 23.1.1 states the following:

> vector *is the type of sequence that should be used by default. ...* deque *is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.*

Interestingly enough, this article is practically devoted to fully understanding that statement.

What's New

After perusing the table above and comparing it with vector, you will notice two new member functions.

1. push_front() - Adds elements to the front of a deque.
2. pop_front() - Removes elements from the front of a deque.

These are called with the same syntax as push_back() and pop_back(). Therein lies the first feature that could possibly warrant the use of deque, namely, the need to add elements to both the front and back of the sequence.

What's Missing

You will also notice there are two member functions implemented in vector but not in deque, and, as you will see, deque doesn't need them.

1. capacity() - Returns the current capacity of a vector.
2. reserve() - Allocates room for a specified number of elements in a vector.

Herein lies the true beginning of our study. As it turns out, there is a stark difference between vector and deque in how they manage their internal storage under the hood. The deque allocates memory in chunks as it grows, with room for a fixed number of elements in each one. However, vector allocates its memory in contiguous blocks (which isn't necessarily a bad thing). But the interesting thing about vector is that the size of the internal buffer grows increasingly larger with each allocation after the vector realizes the current one isn't big enough. The following experiment sets out to prove why deque doesn't need capacity() or reserve() for that very reason.

Experiment 1 - Container Growth

Objective

The objective of this experiment is to observe the differences in container growth between the vector and the deque. The results of this experiment will illustrate these differences in terms of physical memory allocation and application performance.

Description

The test application for this experiment is designed to read text from a file and use each line as the element to push_back() onto the vector and the deque. In order to generate large numbers of insertions, the file may be read more than once. The class to handle the test is shown below:

Hide Shrink Copy Code

```
#include <deque>
#include <fstream>
#include <string>
#include <vector>

static enum modes
{
   FM_INVALID = 0,
   FM_VECTOR,
   FM_DEQUE
};

class CVectorDequeTest
{
  public:
     CVectorDequeTest();

     void ReadTestFile(const char* szFile, int iMode)
     {
        char buff[0xFFFF] = {0};
        std::ifstream    inFile;
```

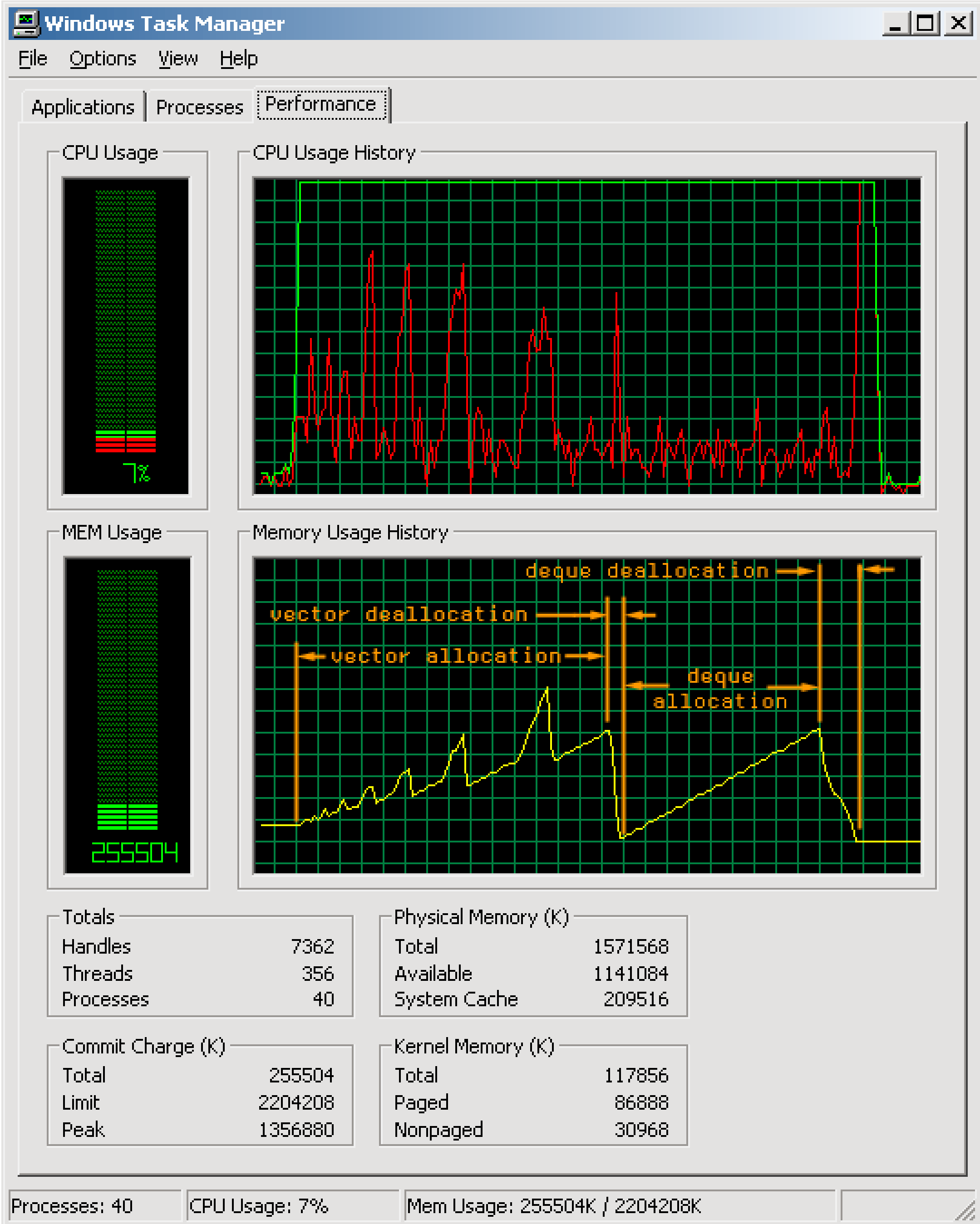
```
        inFile.open(szFile);

        while(!inFile.eof())
        {
           inFile.getline(buff, sizeof(buff));

           if(iMode == FM_VECTOR)
                 m_vData.push_back(buff);
           else if(iMode == FM_DEQUE)
                 m_dData.push_back(buff);
        }

        inFile.close();

      }

      virtual ~CVectorDequeTest();

  protected:
     std::vector<std::string> m_vData;
     std::deque<std::string> m_dData;
};
```

Results

The test was performed under the following conditions:

Processor	1.8 GHz Pentium 4
Memory	1.50 GB
OS	W2K-SP4
No. of Lines in File	9874
Avg. Chars per Line	1755.85
No. of Times File Read	45
Total Elements Inserted	444330

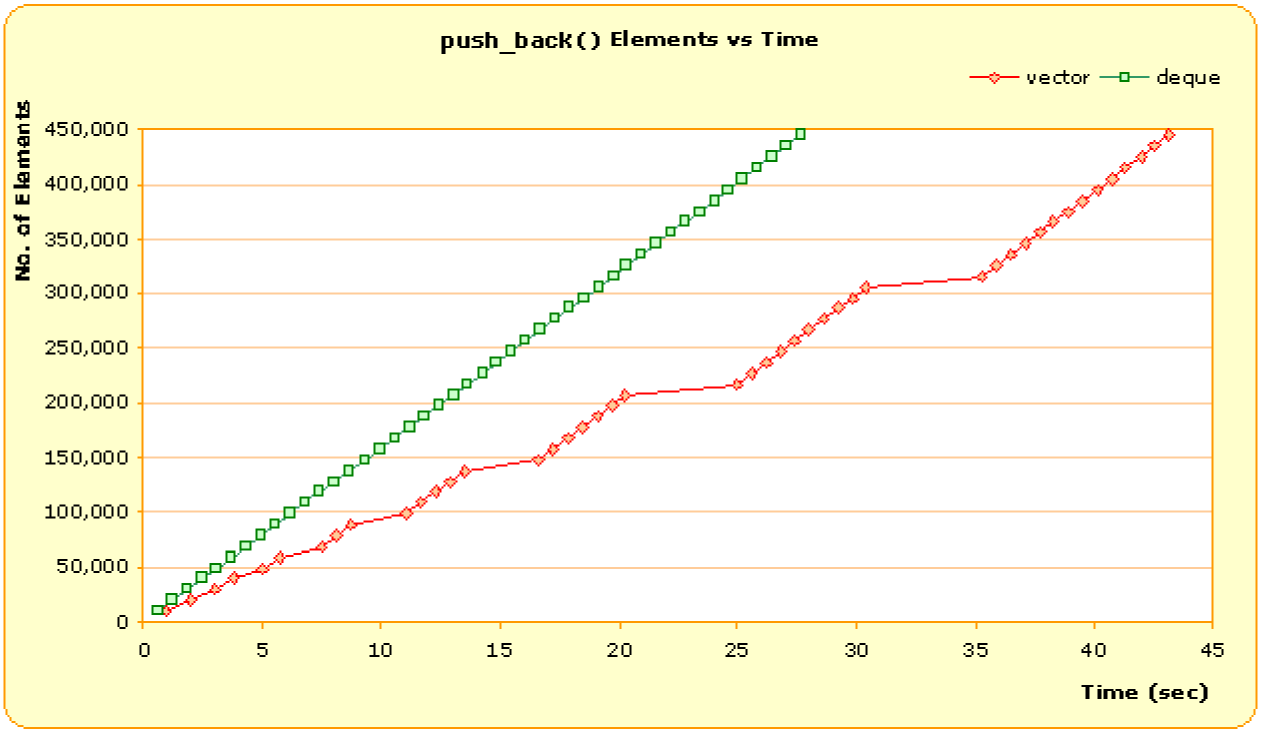
The system performance was logged via Windows Task Manager, and the program was timed using Laurent Guinnard's CDuration class. The system performance graph is illustrated below:



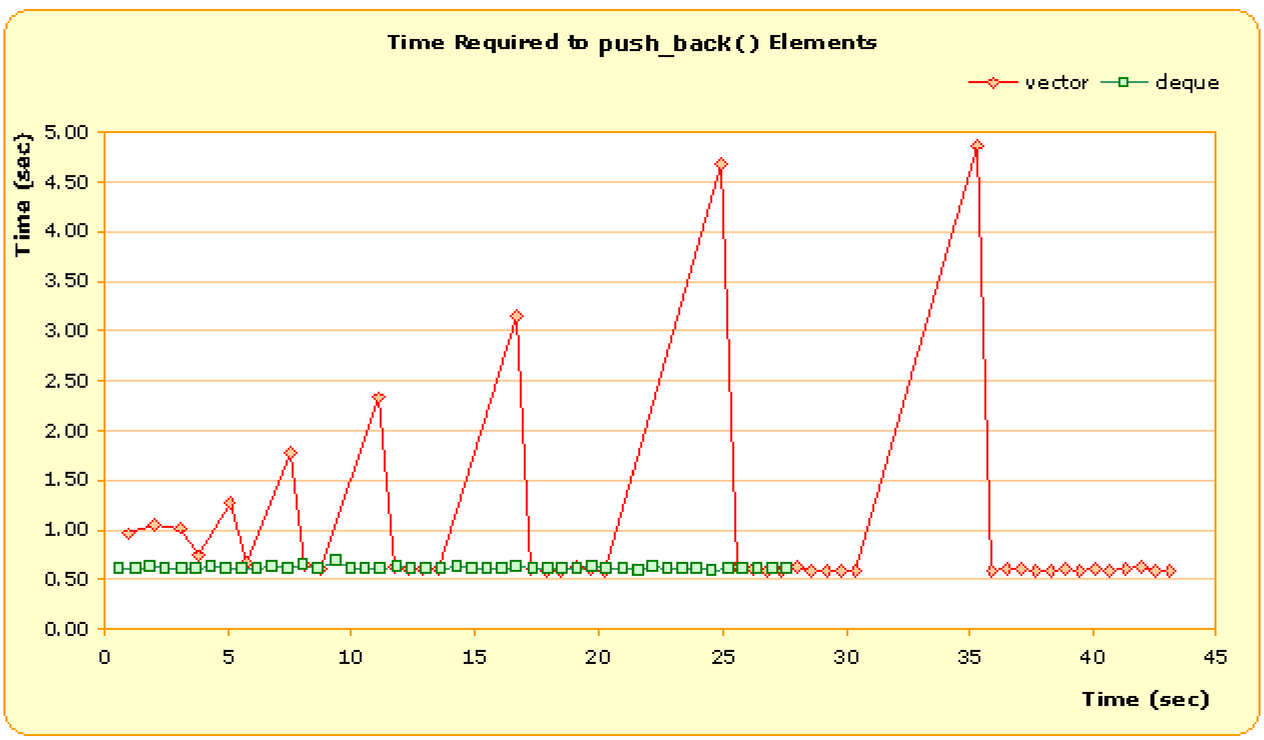
Note the peaks in memory usage during vector allocation, and how the peaks grow larger as vector allocates increasing internal buffer storage. Note also that deque does not exhibit this behavior, and the buffer continues to grow linearly with element insertion. The jump in kernel time during deque deallocation as well as the shape of the curve as memory is reclaimed was an unexpected result at first. I would have expected the deallocation to look similar to vector. After looking into things further and conducting some more tests, I was able to come up with a hypothesis: since deque memory is not contiguous, it must be more difficult to hunt down and reclaim. We will put this hypothesis to the test later, but first let's analyze the performance aspects of this experiment.

Just how long do those memory allocations take?

Notice in the figure below that no elements were being added during the time vector was out finding more memory.



It is also of interest to notice how long each set of push_back() takes. This is illustrated in the figure below. Remember, each sample is 9874 strings added, with an average length of 1755.85.



Experiment 2 - The Effects of vector::reserve()

Objective

The objective of this experiment is to observe the benefits of calling reserve() on a vector before a large number of elements will be added and compare these results with deque, in terms of memory allocation and performance.

Description

The test description for this experiment is the same as that of Experiment 1, except that the following code was added to the test class constructor:

Hide Copy Code

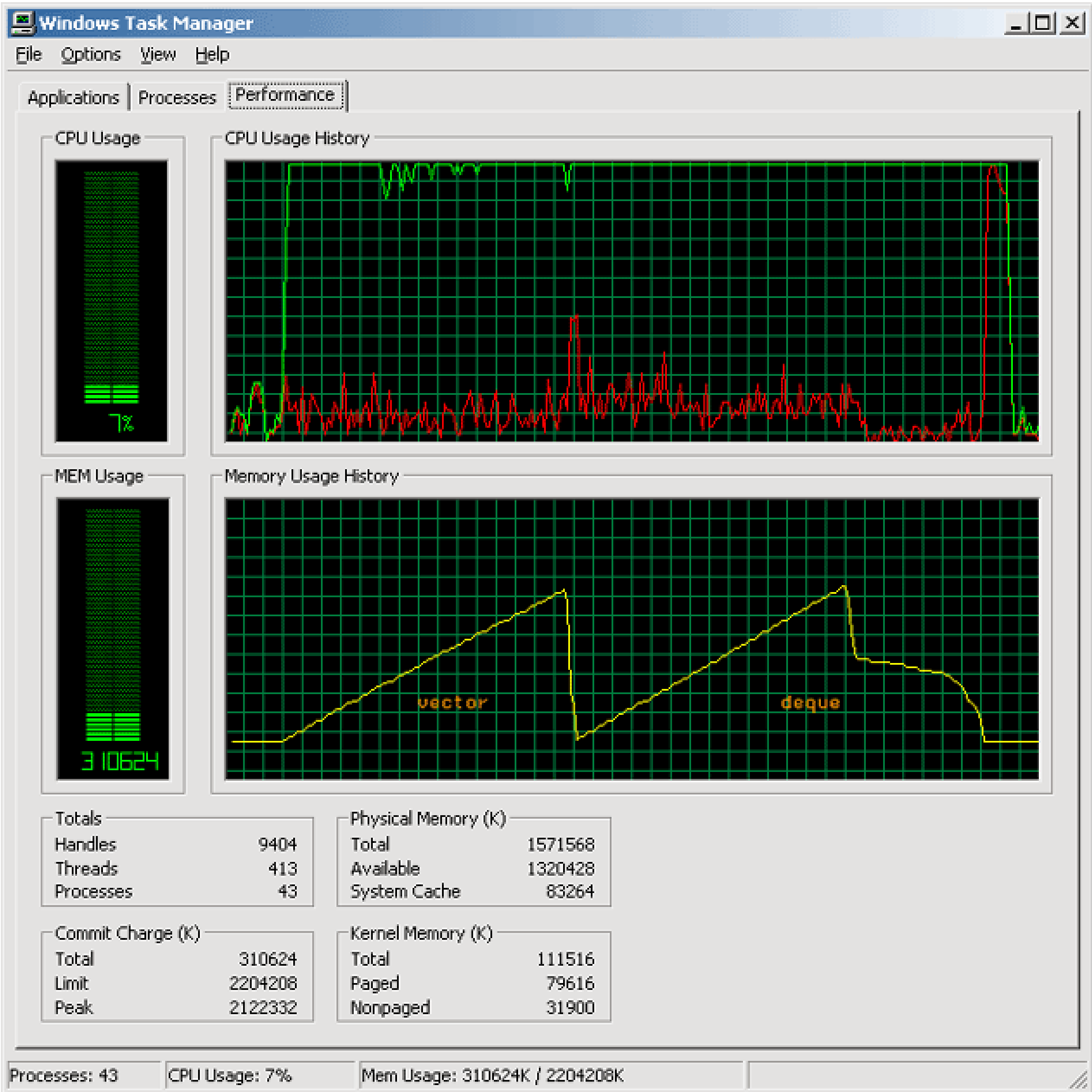
```
m_vData.reserve(1000000);
```

Results

The test was performed under the following conditions:

Processor	1.8 GHz Pentium 4
Memory	1.50 GB
OS	W2K-SP4
No. of Lines in File	9874
Avg. Chars per Line	1755.85
No. of Times File Read	70
Total Elements Inserted	691180

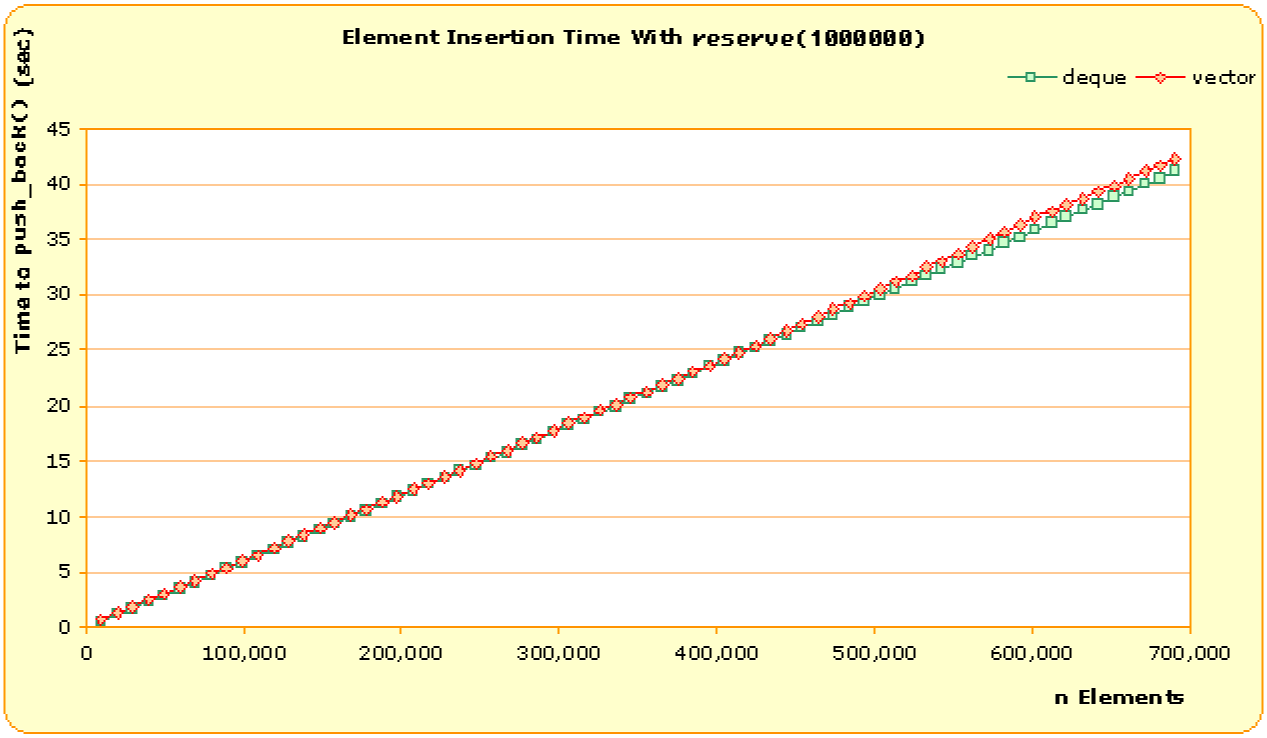
The system performance was logged via Windows Task Manager, and the program was timed using Laurent Guinnard's CDuration class. The system performance graph is illustrated below:



It is of interest to notice that vector no longer needs to allocate more internal buffer storage. The call to reserve() takes a single step to reserve more than enough space for our test platform of 691180 elements. As for the deque deallocation hypothesis, observe the drastic growth in memory deallocation time between this test and the previous one. We will quantify this in our next experiment.

How has this improved memory allocation performance?

The following figure illustrates the number of elements added to the containers over time:



As you can see, vector is now very close to deque in performance, when adding elements to the container. However, vector tends to be slightly more sporadic in how long it takes to insert a given set of elements. This is illustrated in the figure below:

A statistical analysis of the variability in vector vs. deque, with respect to the time it takes to insert 9874 elements of 1755.85 average length, is summarized in the following tables:

Vector

Mean	0.603724814 sec
Maximum	0.738313000 sec
Minimum	0.559959000 sec

Deque

Mean	0.588021114 sec
Maximum	0.615617000 sec
Minimum	0.567503000 sec

Std. Dev	0.037795736 sec	Std. Dev	0.009907800 sec
6-Sigma	0.226774416 sec	6-Sigma	0.059446800 sec

Experiment 3 - Reclaiming Memory

Objective

The objective of this experiment is to analyze and attempt to quantify the hypothesis that deque memory is more difficult to reclaim due to its non-contiguous nature.

Description

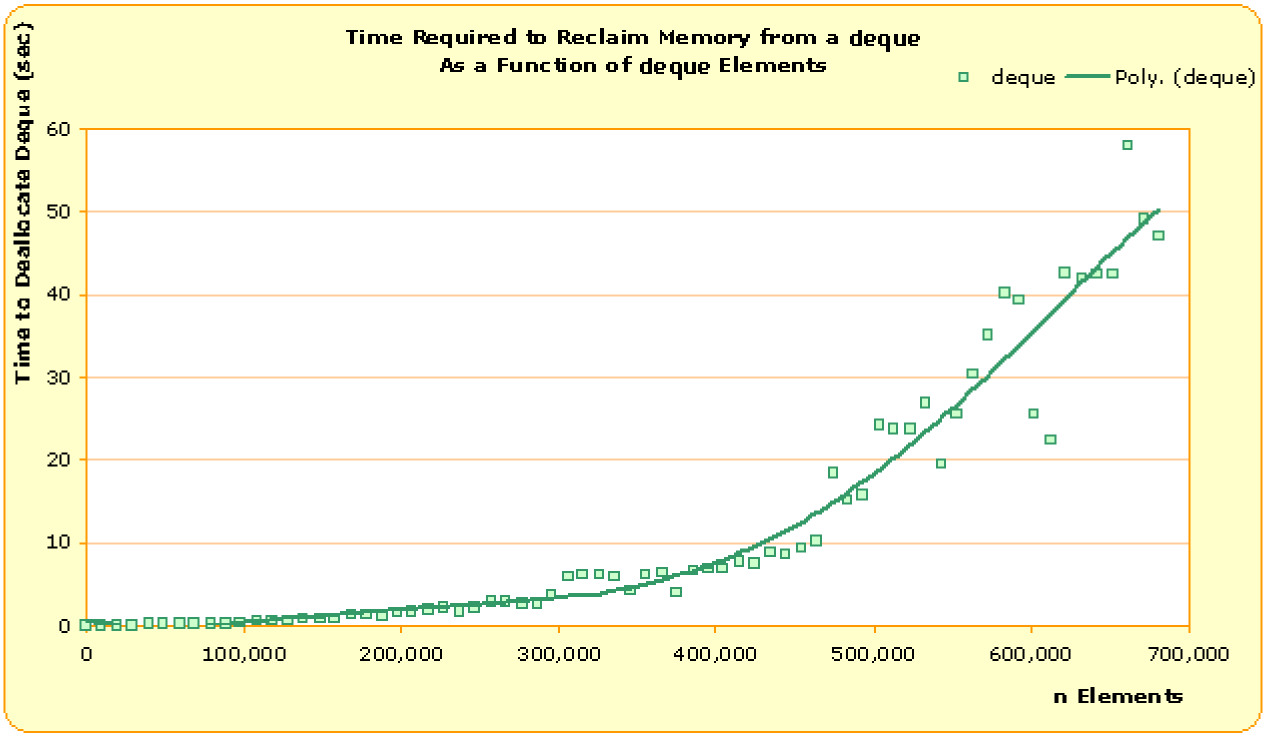
The test class from Experiment 1 will be utilized again in this experiment. The calling function is designed to allocate test classes of increasing size and log their performance accordingly. This implementation is as follows:

Hide Shrink Copy Code

```
for(xRun=0; xRun<NUMBER_OF_XRUNS; xRun++)
{
   df = new CVectorDequeTest;

   elapsed_time = 0;
   for(i=0; i<NUMBER_OF_RUNS*xRun; i++)
   {
      cout << "Deque - Run " << i << " of " << NUMBER_OF_RUNS*xRun << "... ";
      df->ReadTestFile("F:\\huge.csv",DF_DEQUE);

      deque_data.push_back(datapoint());

      deque_data.back().time_to_read = df->GetProcessTime();
      elapsed_time += deque_data.back().time_to_read;

      deque_data.back().elapsed_time = elapsed_time;

      cout << deque_data.back().time_to_read << " seconds\n";
   }

   vnElements.push_back(df->GetDequeSize());

   cout << "\n\nDeleting... ";

   del_deque.Start();
   delete df;
   del_deque.Stop();

   cout << del_deque.GetDuration()/1000000.0 << " seconds.\n\n";

   vTimeToDelete.push_back(del_deque.GetDuration()/1000000.0);
}
```

Results

This experiment was performed on the same platform as the previous two experiments, except that the number of allocations was varied from 9874 to 691180 across 70 increments. The following figure illustrates the time required to reclaim deque memory as a function of the number of elements in the deque. The deque was filled with strings with an average length of 1755.85 chars.



Although the actual time varies significantly from the trendline in several instances, the trendline holds accurate with an R^2=95.15%. The actual deviation of any given data point from the trendline is summarized in the following table:

	deque **Results**
Mean	0.007089269 sec
Maximum	11.02838496 sec
Minimum	-15.25901667 sec
Std. Dev	3.3803636 sec
6-Sigma	20.2821816 sec

This is fairly significant when compared to the results of vector in the same scenario. The following figure shows deallocation times for vector under the same loading as deque above:

The data in this test holds an R^2=81.12%. This could likely be improved with more iterations of each data point and averaging the runs. Nonetheless, the data is suitable to mark the point in question, and the deviation of any given data point from the trendline is summarized in the following statistical parameters:

	vector **Results**
Mean	-0.007122715 sec
Maximum	0.283452127 sec
Minimum	-0.26724459 sec
Std. Dev	0.144572356 sec
6-Sigma	0.867434136 sec

Experiment 4 - Performance Characteristics of vector::insert() vs. deque::insert()

Objective

The "claim to fame" as it were for deque is the promise of constant-time insert(). Just how does this stack up against vector::insert()? The objective of this experiment is (not surprisingly) to observe the performance characteristics of vector::insert() vs. deque::insert().

Description

There may be times when adding things to the back of a container doesn't quite suit your needs. In this case, you may want to employ insert(). This experiment also has the same form as Experiment 1, however instead of doing push_back(), the test does insert().

Results

As you can see in the following figures, the benefit of constant-time insertion offered by deque is staggering when compared against vector.

Note the difference in time-scales, as 61810 elements were added to these containers.

Experiment 5 - Container Retrieval Performance

Objective

This experiment will test the performance of vector::at(), vector::operator[], deque::at() and deque::operator[]. It has been suggested that operator[] is faster than at() because there is no bounds checking, also it has been requested to compare vector vs. deque in this same regard.

Description

This test will insert 1000000 elements of type std::string with a length of 1024 characters into each container and measure how long it takes to access them all via at() and operator[]. The test will be performed 50 times for each scenario and the results presented as a statistical summary.

Results

Well, perhaps surprisingly, there is very little difference in performance between vector and deque in terms of accessing the elements contained in them. There is also negligible difference between operator[] and at() as well. These results are summarized below:

vector::at()		deque::at()	
Mean	1.177088125 sec	Mean	1.182364375 sec
Maximum	1.189580000 sec	Maximum	1.226860000 sec
Minimum	1.168340000 sec	Minimum	1.161270000 sec
Std. Dev	0.006495193 sec	Std. Dev	0.016362148 sec
6-Sigma	0.038971158 sec	6-Sigma	0.098172888 sec

vector::operator[]		deque::operator[]	
Mean	1.164221042 sec	Mean	1.181507292 sec
Maximum	1.192550000 sec	Maximum	1.218540000 sec
Minimum	1.155690000 sec	Minimum	1.162710000 sec
Std. Dev	0.007698520 sec	Std. Dev	0.010275712 sec
6-Sigma	0.046191120 sec	6-Sigma	0.061654272 sec

Conclusions

In this article, we have covered several different situations where one could possibly have a need to choose between vector and deque. Let's summarize our results and see if our conclusions are in line with the standard.

When performing a large number of push_back() calls, remember to call vector::reserve().

In Experiment 1, we studied the behavior of container growth between vector and deque. In this scenario, we saw that since deque allocates its internal storage in blocks of pre-defined size, deque can grow at a constant rate. The performance of vector in this experiment then led us to think about calling vector::reserve(). This was then the premise for Experiment 2, where we basically performed the same experiment except that we had called reserve() on our vector. This then is grounds for holding on to vector as our default choice.

If you are performing many deallocations, remember that deque takes longer to reclaim memory than vector.

In Experiment 3, we explored the differences between reclaiming the contiguous and non-contiguous memory blocks of vector and deque, respectively. The results proved that vector reclaims memory in linear proportion to the number of elements whereas deque is exponential. Also, vector is several orders of magnitude than deque in reclaiming memory. As a side note, if you are performing your calls to push_back() within a tight loop or sequence, there is a significant possibility that most of the memory deque obtains, will be contiguous. I have tested this situation for fun and have found the deallocation time to be close to vector in these cases.

If you are planning to use insert(), or have a need for pop_front(), use deque.

Well, ok, vector doesn't have pop_front(), but based on the results of Experiment 4, it might as well not have insert() either. The results of Experiment 4 speak volumes about the need for deque and why it is part of the STL as a separate container class.

For element access, vector::at() wins by a nose.

After summing up the statistics of Experiment 5, I would have to say that although all the methods were close, vector::at() is the winner. This is because of the best balance between the raw mean of the access times as well as the lowest 6-sigma value.

What's all this 6-Sigma stuff?

Although a popular buzzword in industry today, 6-Sigma actually has its roots in statistics. If you generate a Gaussian distribution (or Bell-curve) for your sampled data, you can show that at one standard deviation (the symbol for std. deviation is the Greek letter, sigma, BTW) from the mean, you will have 68.27% of the area under the curve covered. At 2 Standard Deviations, 2-Sigma, you have 95.45% of the area under the curve, at 3 Standard Deviations, you will have 99.73% of the area and so forth until you get to 6 standard deviations, when you have 99.99985% of the area (1.5 Defects per million, 6-Sigma).

Final Words

I hope you have gained some insight into deque and have found this article both interesting and enlightening. Any questions or comments are certainly welcome and any discussion on vector or deque is encouraged.

References

1. Plauger, P.J. Standard C++ Library Reference. February, 2003. MSDN.
2. ISO/IEC 14882:1998(E). Programming Languages - C++. ISO and ANSI C++ Standard.
3. Schildt, Herbert. C++ from the Ground Up, Second Edition. Berkeley: 1998.
4. Sutter, Herb. More Exceptional C++. Indianapolis: 2002.

License

Share

About the Author

Nitron

Software Developer (Senior) Lockheed Martin
United States

Walter Storm is currently doing quantitative research and data science. Originally from Tunkhannock, PA., he has a B.S. in Aerospace Engineering from Embry-Riddle Aeronautical University[^], and an M.S. in Systems Engineering from SMU[^]. He has been professionally developing software in some form or another since January of 2001.

View Walter Storm's profile on LinkedIn.[^]

Comments and Discussions

You must **Sign In** to use this message board.

Search Comments

Spacing Relaxed Layout Normal Per page 25 Update

First Prev Next

article	**Member 13195584**	**11-May-17 21:46**
Experiment 3 - Exponential time to deallocate memory for a deque	**milo234**	**18-May-16 19:04**
Re: Experiment 3 - Exponential time to deallocate memory for a deque	Nitron	25-Sep-17 4:49
erase performance	**GHop**	**7-Feb-14 10:24**

Experiment 5 strange results (Container Retrieval Performance)	**Jean-Marc Dressler**	**9-Dec-10 0:49**
I do not agree with your analysis on experiment 3	**Shinhy Ku**	**25-Oct-08 6:15**
Re: I do not agree with your analysis on experiment 3	Nitron	31-May-10 17:57
Does it mean that we should prefer deque over vector	**yccheok**	**14-Sep-06 1:22**
Re: Does it mean that we should prefer deque over vector	Nitron	27-Sep-06 18:23
Very nice and...	**Muzero2**	**7-Feb-06 0:48**
6 Sigma explanation Error	**Steve Chernyavskiy**	**8-Jun-05 8:08**
Re: 6 Sigma explanation Error	Nitron	13-Jun-05 13:07
Very Usefull	**Sudhir Mangla**	**9-Mar-05 18:40**
Re: Very Usefull	Nitron	13-Jun-05 13:07
Well done	**nohoper**	**26-Dec-04 20:53**
Great article	**Simon Hughes**	**5-Oct-04 12:01**
Re: Great article	Nitron	13-Jun-05 13:09
Thank you	**mikeboedi**	**28-Mar-04 21:54**
But deque has linear insertion time?	**Eloff**	**29-Jan-04 7:13**
Re: But deque has linear insertion time?	Nitron	2-Feb-04 11:20
Re: But deque has linear insertion time?	Eloff	3-Feb-04 9:31
Re: But deque has linear insertion time?	EnderJSC	15-Mar-04 12:14
Re: But deque has linear insertion time?	Eloff	19-Mar-04 9:12
Re: But deque has linear insertion time?	Anonymous	26-May-04 7:35
Re: But deque has linear insertion time?	Albrecht Fritzsche	3-Dec-04 1:34

Last Visit: 3-Nov-20 19:08 Last Update: 3-Nov-20 19:08 Refresh 1 2 3 Next ⊳

General

News

Suggestion

Question

Bug

Answer

Joke

Praise

Rant

Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.