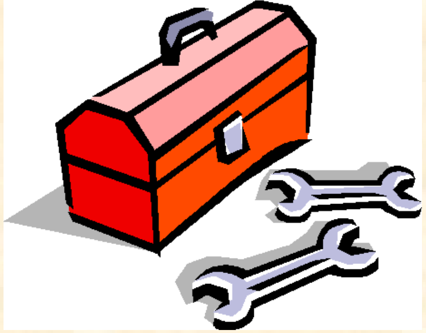
Lecture 6
The Standard Template Library

EECS 281: Data Structures & Algorithms

Q: Why C++? A: The STL

“Nothing has made life easier to programmers using C++ than the Standard Template Library. Though Java, C# and .NET have their own libraries which are as good as C++'s STL (may be even better when it comes to certain aspects) the STL is simply inevitable. If you master the usage of STL and learn to write your own ~~macros and~~ libraries you're all set to rule the competitive programming world, provided your algorithmic knowledge is strong.”

http://www.quora.com/TopCoder/Why-most-people-on-TopCoder-use-C++-as-their-default-language



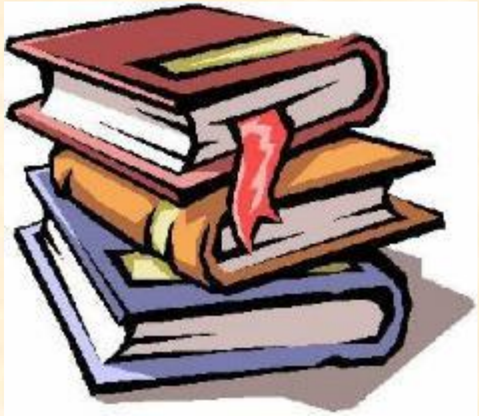
What is STL?

- STL = Standard Template Library
- Included in C++, expanded in C++11
  - Part of stdlibc++ (not stdlibc)
  - Well-documented
  - High-quality implementations of best algorithms and data structs at your fingertips
- All implementations are entirely in headers
  - No linking necessary
  - All code is available (take a look at it!)

Contents of STL

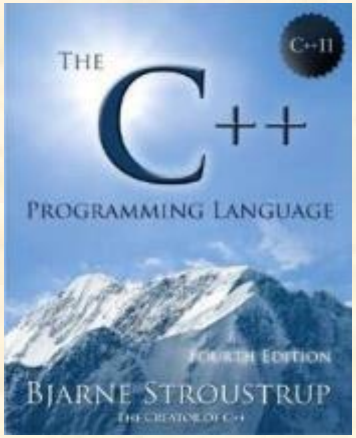
http://en.wikipedia.org/wiki/Standard_Template_Library

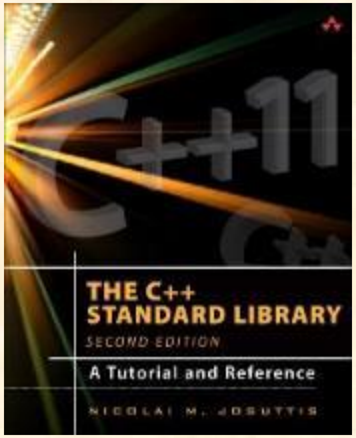
- Containers and iterators
- Memory allocators
- Utilities and function objects
- Algorithms



STL Resources

- The C++ Language, 4e or 5e by Bjarne Stroustrup
- The C++ Standard Library: *A Tutorial and Reference,* 2e by Nicolai Josuttis (covers C++11)
- See cppreference.com ("run this code" feature)
- See cplusplus.com ("edit & run" feature)





http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=standardTemplateLibrary

Using Libraries vs. Do-it-Yourself Pros

- Some algorithms and data structures are hard to implement
  - introsort, red-black trees
- Some are hard to implement well
  - hash-tables, mergesort (`stable_sort()`)
- Uniformity for simple algorithms
  - `max<>(), swap<>(), set_union<>()`
- Reduces debugging time for complicated programs
  - >50% development time is spent testing & debugging
  - Using high-quality libraries reduces debugging time

Using Libraries vs. Do-it-Yourself Cons

- Libraries only contain general-purpose implementations
- Specialized code may run faster
  - Your own code may be able to skip unnecessary checks on input

Using Libraries vs. Do-it-Yourself

Trade-offs

Need to understand a library well to fully utilize it

- Data structures
- Algorithms
- Complexities of operations

Need to know algorithmic details

- STL `sort()`
  - Implemented with *O(N* log *N)* worst-case time
  - In practice is typically faster than quicksort
- STL `nth_element()`
  - Implemented in average-case linear time
- In older STL, linked lists did not store their size!

Learning STL Algorithms

- Online tutorials and examples
- http://www.cplusplus.com/
  - Discusses possible implementations of STL functions, including some subtle mistakes one can make
  - You can copy/modify examples and run them online
- Practice with small tester programs
  - Try algorithms with different data structures/input
- Detailed coverage in books
  - Josuttis, Stroustrup, recent editions of algorithms books

Learning Software Libraries

- Nearly impossible to remember all of stdlibc and stdlibc++
- Not necessary to learn all functions by heart
- Ways to learn a library
  - Skim through documentation
  - Study what seems interesting
  - Learn how to use those pieces
  - Come back when you need something

familiarity and lookup skill versus memorization

The most valuable skill is knowing how to look things up!

C++ Features that STL Relies On

- Type `bool`
- const-correctness and const-casts
- Namespaces

```
using namespace std;
using std::vector;
```

- Templates
- Inline functions
- Exception handling
- Implicit initialization
- Operator overloading
- Extended syntax for `new()`
- Keywords `explicit` and `mutable`

C++ features that are used to *implement* the STL

Some Explanations

- The keyword `explicit` should be used with 1-parameter constructors to prevent accidental conversion from another type

  Given `explicit FeetInches(int feet);`

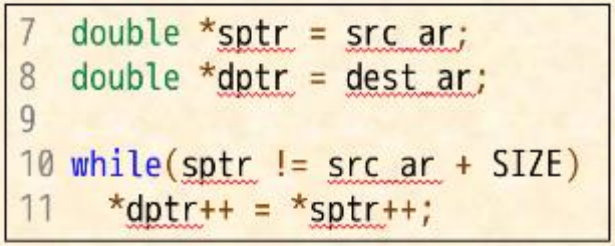
  ```
  FeetInches fi1(3); // OK: 3 feet, 0 inches
  FeetInches fi = 3; // Error
  ```

- A `mutable` member variable can be modified by a `const` member function
  - Used in Project 2, “UnorderedFastPQ” class

Pointers, Generic Programming

- STL helps minimize use of pointers and dynamic memory allocation
  - Debugging time is dramatically reduced
- Can reuse same algorithms with multiple data structures
  - This is much more difficult (and less type-safe) in pure C

Pointer++



```
double *sptr = src_ar;
double *dptr = dest_ar;

while(sptr != src_ar + SIZE)
  *dptr++ = *sptr++;
```

Why would you use pointers when the code seems simpler without them?

Performance and Big-O

- Most STL implementations have the best possible big-O complexities, given their interface
  - Example: `sort()` is $O(n \log n)$ worst case
- Some have surprising complexity
  - `nth_element()` has $O(n)$ average case time
- Some have poor performance even with a good implementation (linked list)

Main priority in STL is time performance; it's very difficult to beat the STL's speed!

STL Containers

All basic containers are available in STL

- vector<> and deque<>
  - stack<> and queue<> are “adaptors”
- bit_vector is same as vector<bool>
- set<> and multi_set<> (plus unordered)
- map<> and multi_map<> (plus unordered)
- list<>
- array<> //Not very useful, fixed size

STL Linked List Containers

Container	**Pointers**	**.size()**
`list<>`	Doubly-linked	*O*(1)
`slist<> **`	Singly-linked	Can be *O*(*n*)
`forward_list<>`	Singly-linked	Does not exist

** DO NOT USE! The autograder will deduct points because `slist<>` includes smart pointers

Copying and Sorting Arrays (C++11+)

```cpp
#include <vector>
#include <algorithm>
using namespace std;
const size_t N = 100;

int main() {
  vector<int> v(N, -1);
  int ar[N];

  for (size_t j = 0; j != N; ++j)
    v[j] = (j * j * j) % N;
  copy(v.begin(), v.end(), ar);  // copy over
  copy(ar, ar + N, v.begin());  // copy back
  sort(ar, ar + N);
  sort(v.begin(), v.end());
  vector<int> reversed(v.rbegin(), v.rend());
} // main()
```

Types of Iterators

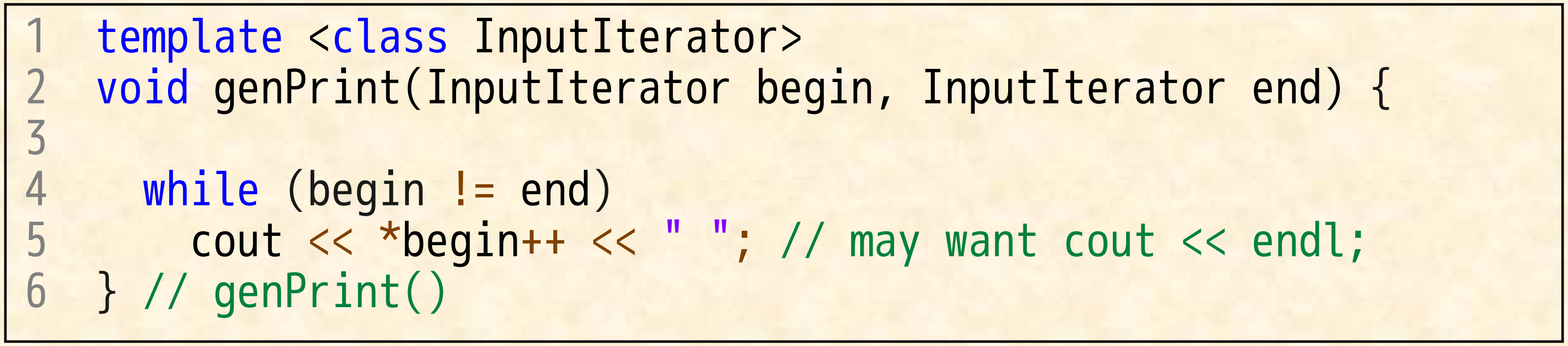
- Access members of a container class
- Similar to pointers; all can be copy-constructed

`input_iterator`	Read values with forward movement. No multiple passes. Can be incremented, compared, and dereferenced.
`output_iterator`	Write values with forward movement. No multiple passes. Can be incremented, and dereferenced.
`forward_iterator`	Read or write values with forward movement. Can be incremented, compared, dereferenced, and store the iterator’s value. Can access the same value more than once.
`bidirectional_iterator`	Same as `forward_iterator` but can also decrement.
`random_iterator`	Same as `bidirectional_iterator` but can also do pointer arithmetic and pointer comparisons.
`reverse_iterator`	An iterator adaptor (that inherits from either a `random_iterator` or a `bidirectional_iterator`) whose ++ operation moves in reverse.

http://www.cppreference.com/iterators.html

Using Iterators

- Iterators generalize pointers
- Allow for implementation of the same algorithm for multiple data structures
  - **Compare: vector iterators to linked-list iterators (!)**
- Support the concept of sequential containers
- Iterators help writing faster code for traversals
  - Compare: **ar[i++]** to ***(it++)**



```
template <class InputIterator>
void genPrint(InputIterator begin, InputIterator end) {

  while (begin != end)
    cout << *begin++ << " "; // may want cout << endl;
} // genPrint()
```

Iterator Ranges

- All STL containers that support iterators support
  - `.begin(), .end(), .cbegin(), .cend(), .rbegin(), .rend()`
  - "begin" is inclusive, "end" is exclusive **(one past last)**
- What about C arrays? - they are not classes!
  - C++14+ adds `std::begin(), std::end(), std::cbegin(),` …
    (illustrated on the next slide)
- STL operates on *iterators ranges*, not containers
  - A range can capture *any fraction* of a container
  - Iterator ranges (unlike indices) need no *random access*
  - *Faster traversal* than with indices

Copying and Sorting Arrays (C++14+)

```
#include <vector>
#include <algorithm>
using namespace std;
const size_t N = 100;

int main() {
  vector<int> v(N, -1);
  int ar[N];

  for (size_t j = 0; j != N; ++j)
    v[j] = (j * j * j) % N;
  copy(begin(v), end(v), begin(ar));  // copy over
  copy(begin(ar), end(ar), begin(v));  // ⋯ back
  sort(begin(ar), end(ar));
  sort(begin(v), end(v));
  vector<int> reversed(rbegin(v), rend(v));
} // main()
```

(Not) Using Iterators

- You might be tempted to write a template version without iterators
- DON’T DO THIS: leads to multiple compiler errors due to ambiguity

```
template <class Container>
ostream& operator<<(ostream& out,
   const Container& c) {

  auto it = c.begin();
  while (it != c.end())
    out << *it++ << " ";
  return out;
}
```

```
template <class Cont>
ostream& operator<<(ostream& out,
   const Cont& c) {

  for (auto &x: c)
    out << x << endl;
  return out;
}
```

http://en.cppreference.com/w/cpp/language/range-for

A Better Method

```
// Overload for each container type you need to output
template <class T>
ostream &operator<<(ostream &out, const vector<T> &c) {
  for (auto &x : c)
    out << x << " ";
  return out;
} // operator<<()
```

- This code compiles without ambiguities
- Just implement another version for `list<T>`, `deque<T>`, etc.

Memory Allocation & Initialization

- Initializing elements of a container
- Containers of pointers
- Behind-the-scenes memory allocation

new in C++11

Data structure	**Memory overhead**
`vector<>`	Compact
`list<>`	Not very compact
`unordered_map<>`	Memory hog

10 x 20 array

```
vector<vector<int>> twoDimArray(10);
for (size_t i = 0; i < 10; ++i)
  twoDimArray[i] = vector<int>(20, -1);
// or
for (size_t i = 0; i < 10; ++i)
  twoDimArray[i].resize(20, -1);
```

streamlined

```
vector<vector<int>> twoDimArray(10, vector<int>(20, -1));
```

Memory overhead of std::vector

- Three pointers (3 * 8 bytes) – *O*(1) space
    1. Begin allocated memory
    2. End allocated memory
    3. End used memory
    - `vector<SmallClass>` vs. `vector<LargeClass>`
    - Large overhead when using many small vectors
- `vector<vector<vector<T>>> ar3d(a, b, c);`
    - Overhead in terms of pointers: 3 + 3a + 3ab
- Reorder dimensions to reduce overhead: a < b < c
    - Or ensure *O*(1) space overhead by arithmetic indexing

Utilities and Function Objects

- `swap<>`, `max<>`
- See STL docs for more utilities
- Function objects (functors) remove the need for function pointers
  - Compare STL `sort()` with older `qsort()`
- New since C++11
  - "lambdas" (instead of functors)
  - Not covered in EECS 281

Using a Functor

- Suppose we have a class Employee that we want to sort
  - Don’t overload `operator<()`
    - We might want to sort Employee objects many different ways
  - Use helper class: a functional object or “functor”

```
struct SortByName {
  bool operator()(const Employee &left,
                  const Employee &right) const {
    return left.getName() < right.getName();
  } // operator()()
};
```

Index Sorting

```cpp
class SortByCoord {
  const vector<double> &_coords;

public:
  SortByCoord(const vector<double> &z) : _coords(z) {}

  bool operator()(unsigned int i, unsigned int j) const {
    return _coords[i] < _coords[j];
  } // operator()()
};

vector<unsigned int> idx(100);
vector<double> xCoord(100);
for (unsigned int k = 0; k != 100; ++k) {
    idx[k] = k;
    xCoord[k] = rand() % 1000 / 10.0;
} // for

SortByCoord sbx(xCoord); // sbx is a function object!
sort(begin(idx), end(idx), sbx);
```

Try this!

Filling a Container with Values

- Instead of using a loop, there is a simple function called `iota()`, standard as of C++11

```
// Fill a vector with values, starting at 0
// Must #include <numeric>
iota(begin(v), end(v), 0);
```

Generating Random Permutations

(great for testing a program)

```cpp
#include <iostream>
#include <vector>
#include <algorithm>  // Needed for shuffle()
#include <numeric>    // Needed for iota()
#include <random>     // Needed for random_device and mt19937
using namespace std;

int main() {
  random_device rd;  // Create a device to start random # generation
  mt19937 g(rd());   // Create a Mersenne Twister to generate random #s
  int size = 20;     // Could also read size from cin
  vector<int> values(size);

  iota(values.begin(), values.end(), 0);
  shuffle(values.begin(), values.end(), g);

  for (auto v : values)
    cout << v << " ";

  cout << endl;

  return 0;
} // main()
```

Debugging STL-heavy Code: *Compiler Errors*

- Compiler often *complains about STL headers*, not your code – **induced errors**
- You will need to sift through many lines of messages, to find line reference to your code
- Good understanding of type conversions in C++ is often required to fix problems
- Double-check *function signatures*

Debugging STL-heavy Code: *Runtime Debugging*

- Crashes can occur in STL code, started by an error in your code
- Debugging needed with ANY library
- In gdb, use "**where**" or "**bt**" commands to find code that calls STL
- 90% of STL-related crashes are due to user's dangling pointers or references going out of scope

Relative Performance of STL Containers (1)

Filling an empty container with different values

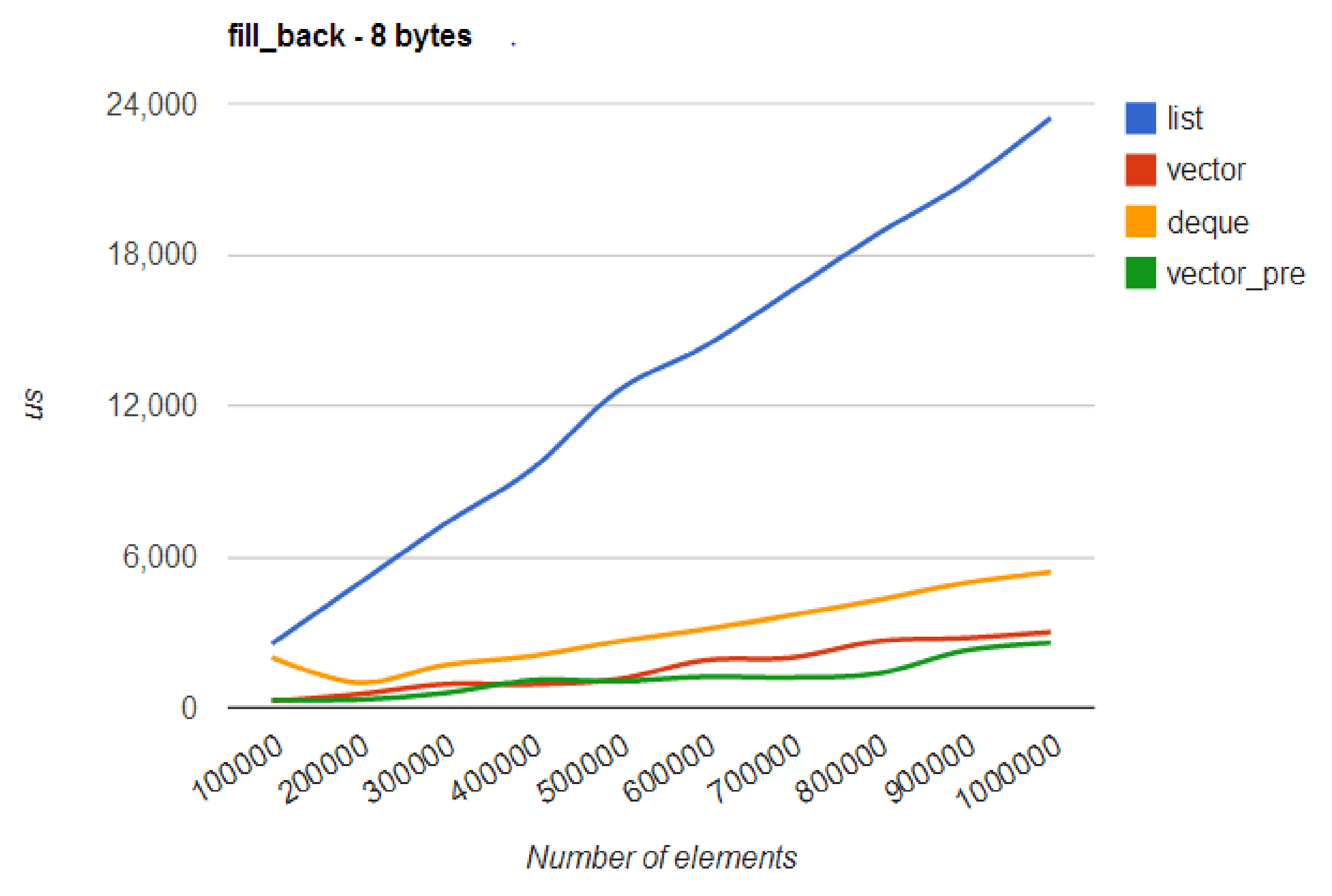
vector_pre used

`vector::resize()`

(a single allocation)

Intel Core i7 Q820 @1.73GHz

GCC 4.7.2 (64b)
-02 -std=c++11
-march=native

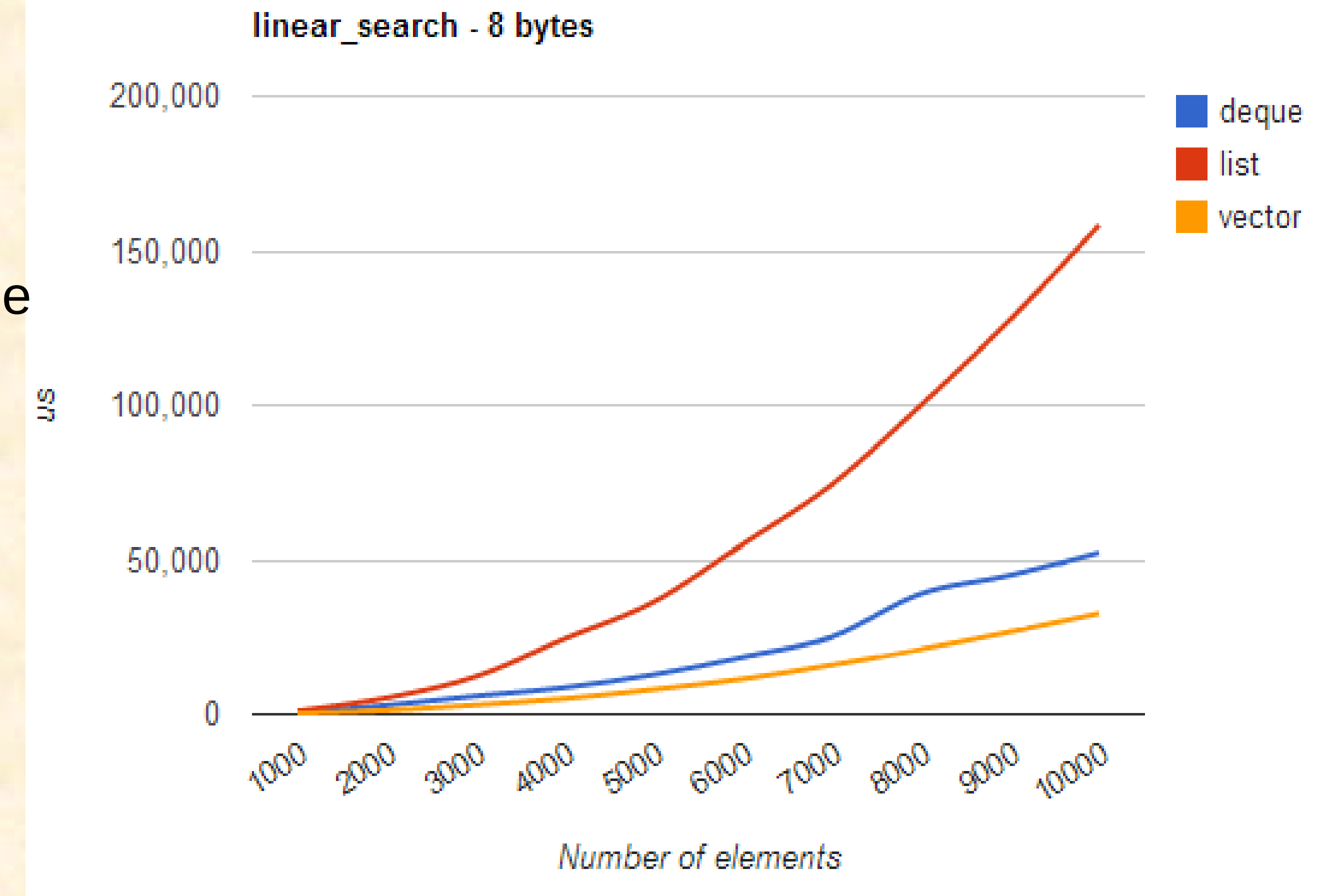


http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque/

Relative Performance of STL Containers (2)

Fill the container with numbers [0, *N*], shuffle at random;

search for each value using `std::find()`



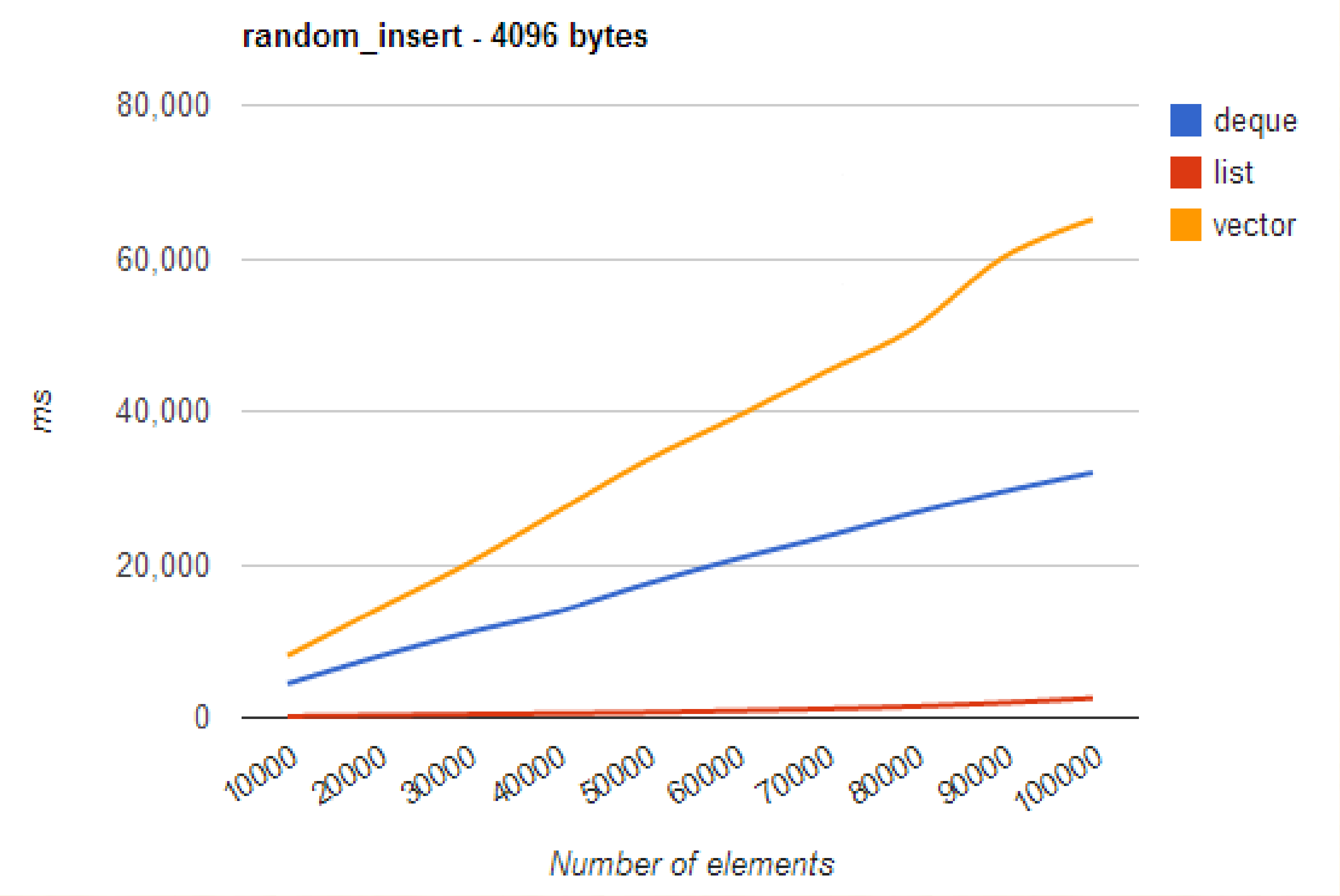
http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque/

Relative Performance of STL Containers (3)

Fill the container with numbers [0, *N*], shuffle at random;

Pick a random position by linear search

Insert 1000 values



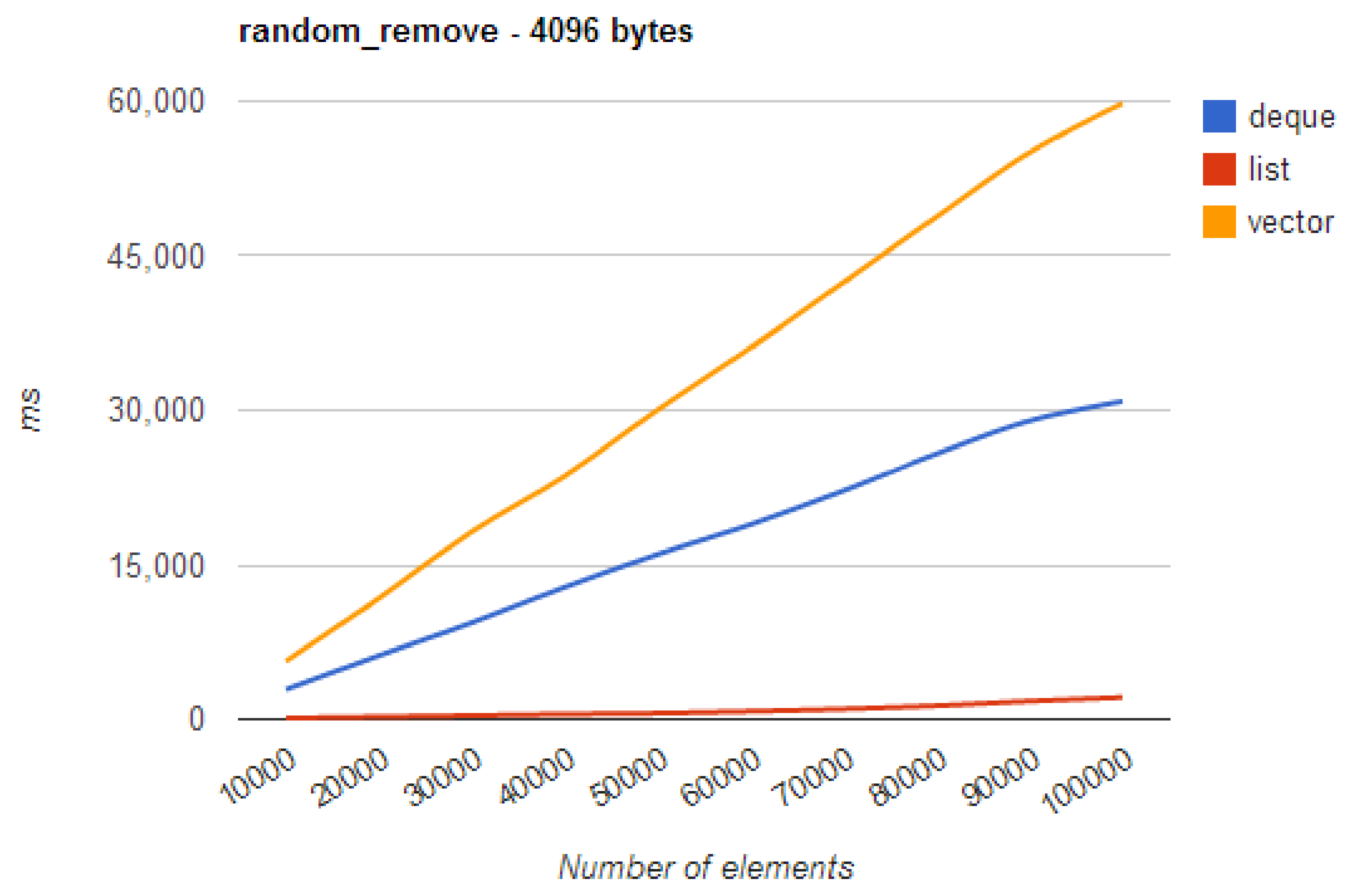
http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque/

Relative Performance of STL Containers (4)

Fill the container with numbers [0, *N*], shuffle at random;

Pick a random position by linear search

Remove 1000 elements

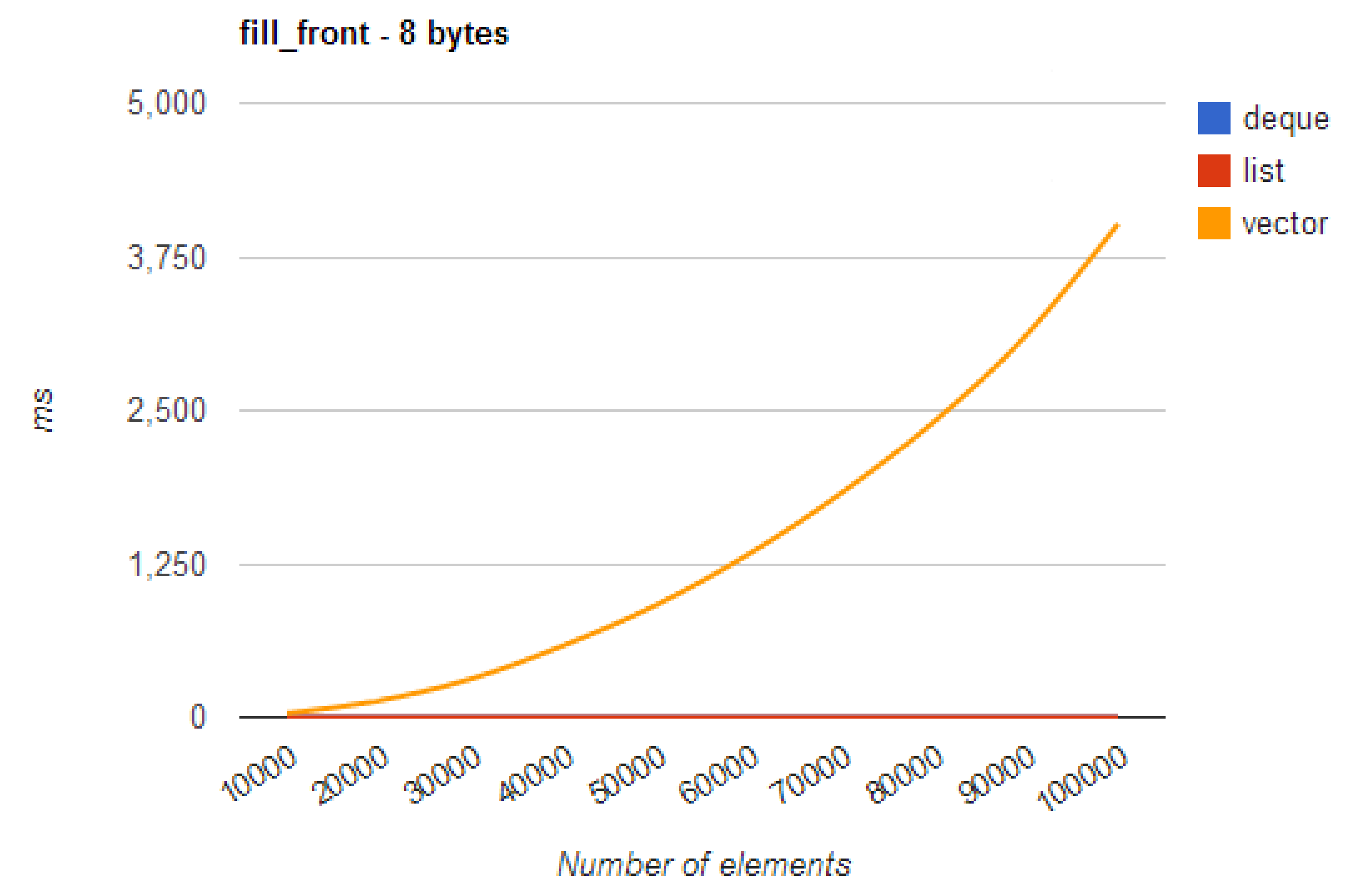


http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque/

Relative Performance of STL Containers (5)

Insert new values at the front

A vector needs to move all prior elts, but a list does not



http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque/

Learning STL

http://en.wikipedia.org/wiki/Standard_Template_Library

- Main reference: the Josuttis book
- Examples online, run your own examples
- Read documentation for more info
- Same methods with different containers
- Show your code to TAs, ask for comments on coding style
- Familiarize yourself with the library