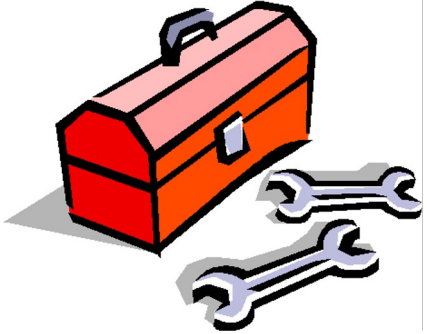
Lecture 7
The Standard Template Library

EECS 281: Data Structures & Algorithms

STL Basics

Data Structures & Algorithms

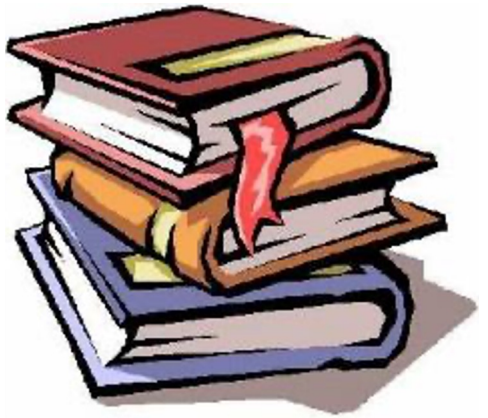


What is STL?

- 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

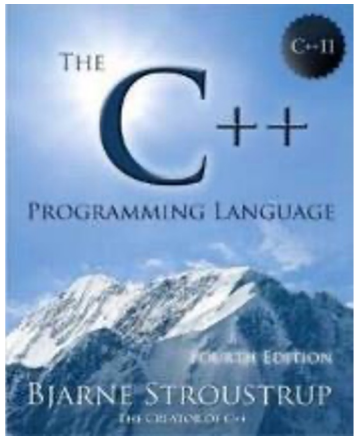
- Containers and iterators
- Memory allocators
- Utilities and function objects
- Algorithms

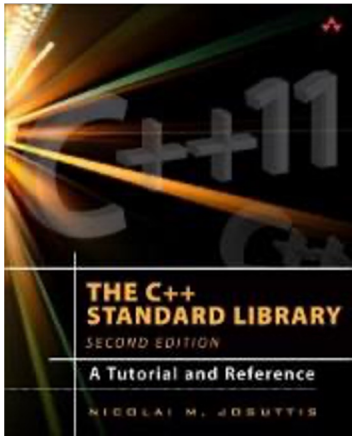


http://en.wikipedia.org/wiki/Standard_Template_Library

STL Resources

- The C++ Language, 4e 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, Merge sort (`std::stable_sort()`)
- Uniformity for simple algorithms
  - `std::max<>()`, `std::swap<>()`, `std::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 `std::sort()`
  - Implemented with ***O*(*n* log *n*)** worst-case time
  - In practice is typically faster than quicksort
- STL `std::nth_element()`
  - Implemented in average-case linear time
- In older STL, linked lists did not store their size!

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

```
explicit FeetInches(int feet);
FeetInches a(3);  // OK: 3 feet, 0 inches
FeetInches b = 3;  // Error
```

- A `mutable` member variable can be modified by a `const` member function

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++

```
7  double *sptr = src_ar;
8  double *dptr = dest_ar;
9
10 while(sptr != src_ar + SIZE)
11   *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: `std::sort()` is ***O*(*n* log *n*)** worst case
- Some have surprising complexity
  - `std::nth_element()` is ***O*(*n*)** average case
- 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 Basics

Data Structures & Algorithms

STL Containers and Iterators

Data Structures & Algorithms

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<>` (Limited use, size fixed at compile time)

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

† Includes smart pointers

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++)`

```
1  template <class InputIterator>
2  void genPrint(InputIterator begin, InputIterator end) {
3
4    while (begin != end)
5      cout << *begin++ << " ";  // may want cout << endl;
6  }  // genPrint()
```

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 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.

https://en.cppreference.com/w/cpp/iterator

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()`, ...
- 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 (C++11+)

```
1  #include <vector>
2  #include <algorithm>
3  using namespace std;
4  const size_t N = 100;
5
6  int main() {
7    vector<int> v(N, -1);
8    int a[N];
9
10   for (size_t j = 0; j != N; ++j)
11     v[j] = (j * j * j) % N;
12   copy(v.begin(), v.end(), a);  // copy over
13   copy(a, a + N, v.begin());  // copy back
14   sort(a, a + N);
15   sort(v.begin(), v.end());
16   vector<int> reversed(v.rbegin(), v.rend());
17 }  // main()
```

Copying and Sorting (C++14+)

```
1  #include <vector>
2  #include <algorithm>
3  using namespace std;
4  const size_t N = 100;
5
6  int main() {
7    vector<int> v(N, -1);
8    int a[N];
9
10   for (size_t j = 0; j != N; ++j)
11     v[j] = (j * j * j) % N;
12   copy(begin(v), end(v), begin(a));  // copy over
13   copy(begin(a), end(a), begin(v));  // copy back
14   sort(begin(a), end(a));
15   sort(begin(v), end(v));
16   vector<int> reversed(rbegin(v), rend(v));
17 }  // 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 = begin(c);
  while (it != end(c))
    out << *it++ << " ";
  return out;
}  // operator<<()
```

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

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

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

A Better Method

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

- This code compiles without ambiguities
- Needs multiple versions for `list<>`, `deque<>`, etc.

Memory Allocation & Initialization

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

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

new in C++11

10 x 20 array

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

streamlined

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

`std::vector<>` Memory Overhead

- 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

STL Containers and Iterators

Data Structures & Algorithms

Functors and Lambdas

Data Structures & Algorithms

Using a Functor

Suppose a class Employee needs to be sorted

- Don't overload `operator<()`
  - Employee objects may need sorting multiple ways
- Use helper class: a functional object or "functor"

```
1  struct SortByName {
2    bool operator()(const Employee &left,
3                    const Employee &right) const {
4      return left.getName() < right.getName();
5    }  // operator()()
6  };  // SortByName{}
```

Index Sorting

```
1   class SortByCoord {
2     const vector<double> &_coords;
3
4   public:
5     SortByCoord(const vector<double> &z) : _coords(z) {}
6
7     bool operator()(size_t i, size_t j) const {
8       return _coords[i] < _coords[j];
9     }  // operator()()
10  };  // SortByCoord{}
11
12  vector<size_t> idx(100);
13  vector<double> xCoord(100);
14  for (size_t k = 0; k != 100; ++k) {
15    idx[k] = k;
16    xCoord[k] = rand() % 1000 / 10.0;
17  }  // for
18
19  SortByCoord sbx(xCoord);  // sbx is a function object!
20  sort(begin(idx), end(idx), sbx);
```

Try this!
https://bit.ly/3dDLSMS

Filling a Container with Values

Instead of using a loop, there is a simple function **`std::iota()`** (since C++11)

```
// Fill a vector with values, starting at 0
#include <numeric>
vector<int> v(100);
iota(begin(v), end(v), 0);
// v contains {0, 1, 2, ···, 99}
```

Lambdas

- A **lambda** is an anonymous function object that can be defined on the fly, directly in the code where it is called
  - Improves code readability by keeping code localized (no need to name a function elsewhere that you will only ever use once)
  - Makes STL algorithms easier to use
  - Can be passed around and used as function arguments (e.g., callback functions)

Anatomy of a Lambda

- A lambda expression consists of the following:

```
[captures list] (parameter list) {function body}
```

  - The *captures list* specifies variables from the current scope that will be available inside the lambda
  - The *parameter list* specifies the argument names and types passed into the lambda
  - The *function body* defines the behavior of the lambda
- The following lambda takes in two integers and checks if their difference is greater than 5:

```
[] (int n1, int n2) { return abs(n1 - n2) > 5; }
```

Anatomy of a Lambda

- Compiler will attempt to deduce the return type of a lambda function body whenever possible
- The arrow operator can be used to explicitly specify a return type
  - When type returned is different from args
  - When type returned cannot be implicitly deduced

```
[captures list] (parameter list) -> return_type {function body}
```

```
[] (double x, double y) -> int { return x + y; }
```

Lambda Example

Find the first odd number in a vector

```
1   struct IsOddPred {  // Returns true if number is odd
2     bool operator()(int n) {
3       return n % 2 == 1;
4     }  // operator()()
5   };  // IsOddPred{}
6
7   int main() {
8     vector<int> vec = {24, 32, 54, 86, 53, 47, 92, 61};
9
10    // Using the functor defined above
11    auto it1 = find_if(begin(vec), end(vec), IsOddPred());
12
13    // Using a lambda defined in place
14    auto it2 = find_if(begin(vec), end(vec), [](int n) {
15      return n % 2 == 1;
16    });  // find_if()
17  }  // main()
```

Lambda Variable Capture

- A lambda can use parameters and other variables in its function body
- To use variables from its surrounding scope, captures are required
- Variables placed within [ ] before the parameter list are captured (by value or reference with &)

```
1   int main() {
2     vector<int> vec = {45, 63, 28, 21, 80, 91, 88, 72, 34, 57, 50, 69};
3     int factor;
4
5     cout << "Enter a number between 1 and 25: " << endl;
6     cin >> factor;
7     cout << "The first number that is a multiple of " << factor << " is: ";
8     cout << *find_if(begin(vec), end(vec), [factor](int n) {
9       return n % factor == 0;
10    });  // find_if()
11  }  // main()
```

Lambda Variable Capture

There are several ways to capture variables in a lambda:

- `[]` captures no variables from the surrounding scope
- `[=]` captures all variables in the surrounding scope by value (i.e., a copy of each variable is made)
- `[&]` captures all variables in the surrounding scope by reference
- `[foo]` captures only the variable foo, by value
- `[&foo]` captures only the variable foo, by reference
- `[foo, bar]` captures only the variables foo and bar, by value
- `[=, &foo]` captures all variables in the surrounding scope by value except for foo, which is captured by reference
- `[&, foo]` captures all variables in the surrounding scope by reference except for foo, which is captured by value
- `[this]` captures the current object – needed if a lambda defined in an object's member function needs access to its member variables

Lambda Exercise

- Given a vector of Message objects, where each Message stores a send time and a receive time:

```
struct Message {
  uint64_t sent;
  uint64_t received;
};
```

- Use lambdas and an STL algorithm to implement a function which takes in a threshold value and returns true if there exists a Message in the vector that took longer than threshold to send

```
bool msgPastThreshold(vector<Message> &msg, uint64_t threshold);
```

HINT: The `std::any_of()` function may be useful here.

```
bool any_of(InputIt first, InputIt last, UnaryPredicate p);
```

Lambda Exercise Solution

- Solution:

```
1   bool msgPastThreshold(vector<Message> &msg, uint64_t threshold) {
2     return any_of(begin(msg), end(msg),
3                   [threshold](const Message &m) {
4                     return m.received - m.sent > threshold;
5     });  // any_of()
6   }  // msgPastThreshold()
```

- Bonus: sort Message objects in ascending order of total send duration (no need to define a separate comparator)

```
7   void sortMsgBySendDuration(vector<Message> &msg) {
8     sort(begin(msg), end(msg),
9          [](const Message &lhs, const Message &rhs) {
10           return   (lhs.received - lhs.sent)
11                  < (rhs.received - rhs.sent);
12    });  // sort()
13  }  // sortMsgBySendDuration()
```

Functors and Lambdas

Data Structures & Algorithms

Using the STL

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

Learning STL

- 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 edition algorithms books
- Read the STL card posted on Canvas
  - Files / Resources / algolist.pdf

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!

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**

Randomization

- C++11 introduced the <random> library to support random number generation:
  - **Generators:** generate random numbers (e.g., `std::random_device`, `std::mt19937`, `std::mt19937_64`)
  - **Distributions:** convert generator output into statistical distributions (e.g, `std::normal_distribution`, `std::uniform_int_distribution`, `std::uniform_real_distribution`)
- Generally preferred over `rand()`
- https://en.cppreference.com/w/cpp/numeric/random
- Randomization is great for testing code

Generating Random Permutations

(great for testing a program)

```
1   #include <iostream>
2   #include <vector>
3   #include <algorithm>  // Needed for shuffle()
4   #include <numeric>    // Needed for iota()
5   #include <random>     // Needed for random_device and mt19937_64
6   using namespace std;
7
8   int main() {
9     random_device rd;  // Create a device to start random # generation
10    mt19937_64 mt(rd());  // Create a Mersenne Twister to generate random #s
11    int size = 20;  // Could also read size from cin
12    vector<int> values(size);
13
14    iota(begin(values), end(values), 0);
15    shuffle(begin(values), end(values), mt);
16
17    for (auto v : values)
18      cout << v << " ";
19
20    cout << endl;
21
22    return 0;
23  }  // main()
```

Generating Random Numbers

(great for testing a program)

```
1   #include <iostream>
2   #include <random>      // Needed for random_device and mt19937_64
3   using namespace std;
4
5   int main() {
6     random_device rd;  // Create a device to start random # generation
7     mt19937_64 mt(rd());  // Create a Mersenne Twister to generate random #s
8     int size = 10;  // Could also read size from cin
9
10    cout << size << '\n';
11    // Run a loop, generating random x/y coordinates; useful for Project 4!
12    for (int i = 0; i < size; ++i) {
13      // mt() generates a large random number, % limits the range, - makes
14      // about half the numbers negative; the range is -10 to 10 (inclusive)
15      int x = static_cast<int>(mt() % 21) - 10;
16      int y = static_cast<int>(mt() % 21) - 10;
17      cout << x << ' ' << y << '\n';
18    } // for
19    cout << endl;
20
21    return 0;
22  }  // main()
```

Using the STL

Data Structures & Algorithms

STL Container Performance

Data Structures & Algorithms

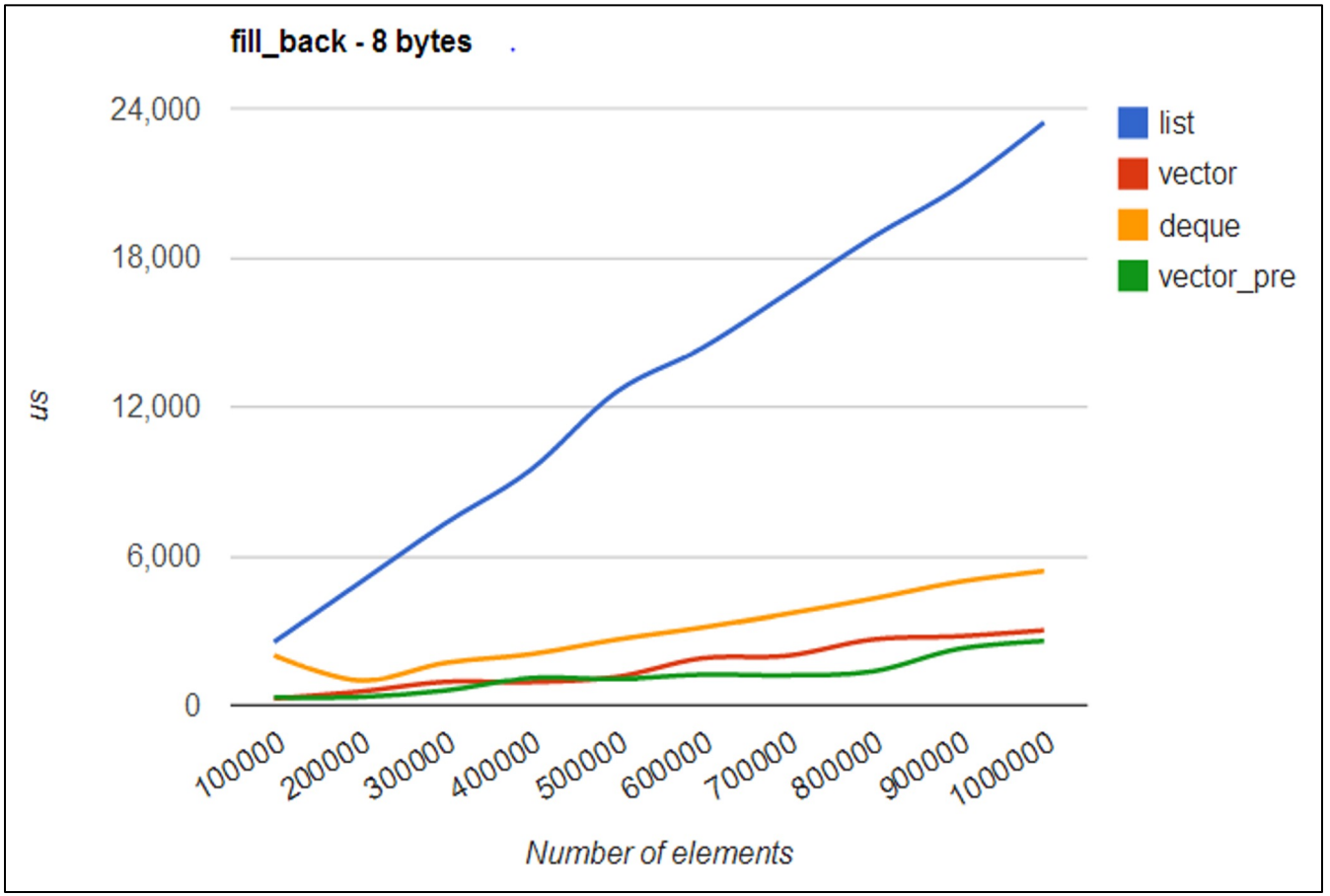
Relative Performance of STL Containers (1)

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) `-O2 -std=c++11 -march=native`

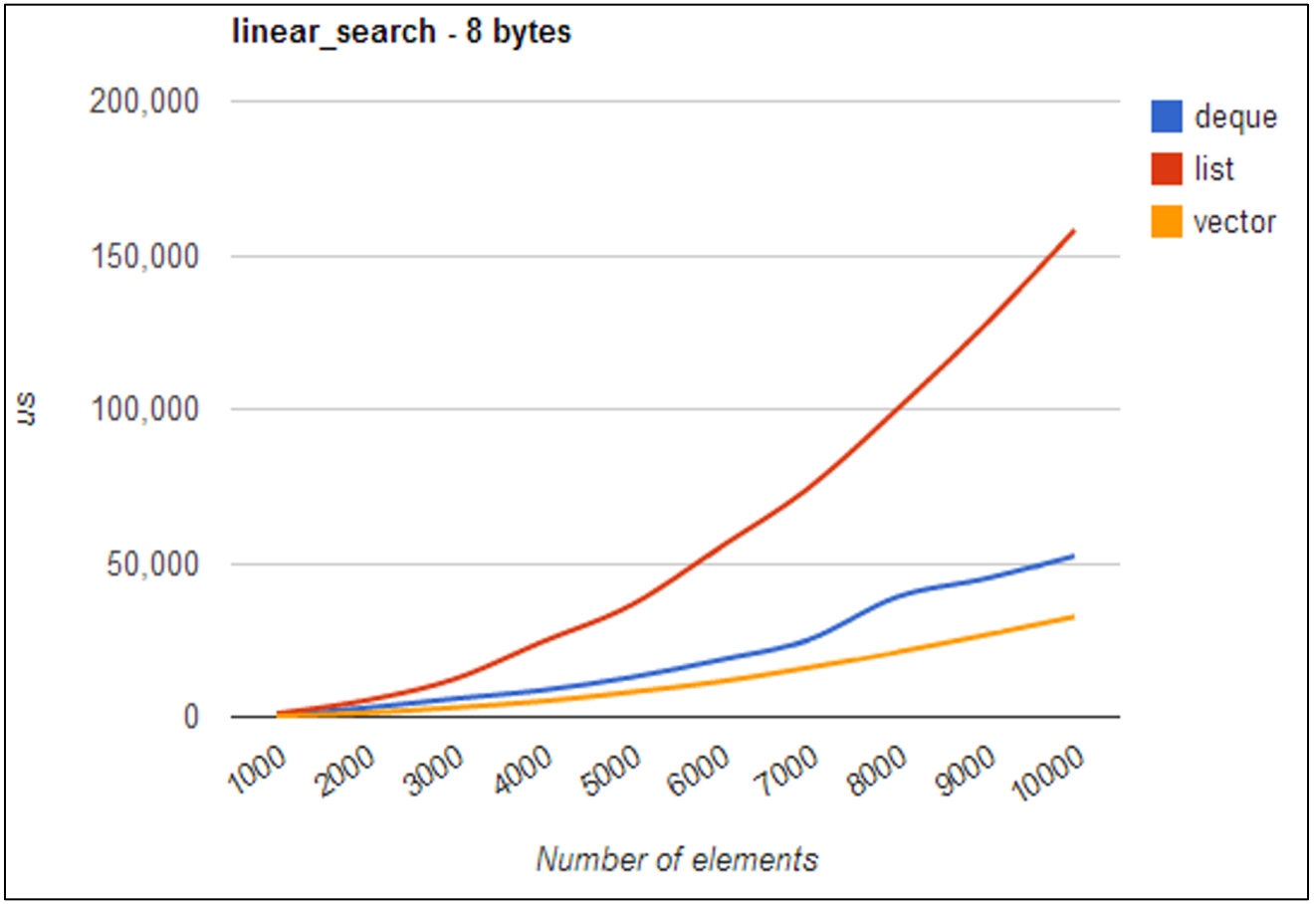


http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque

Relative Performance of STL Containers (2)

Fill the container with numbers `[0, N)`, shuffle at random;

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



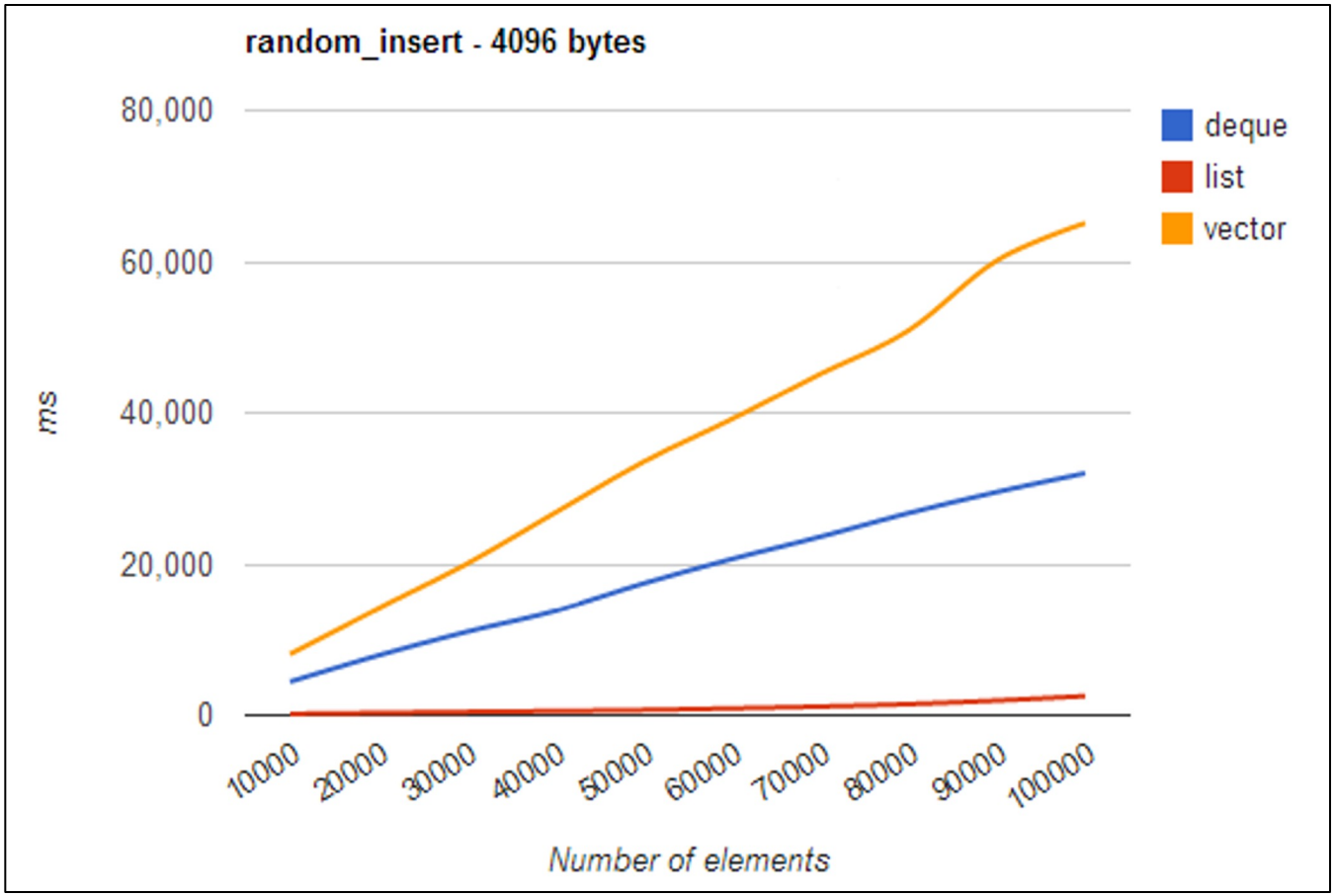
http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque

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



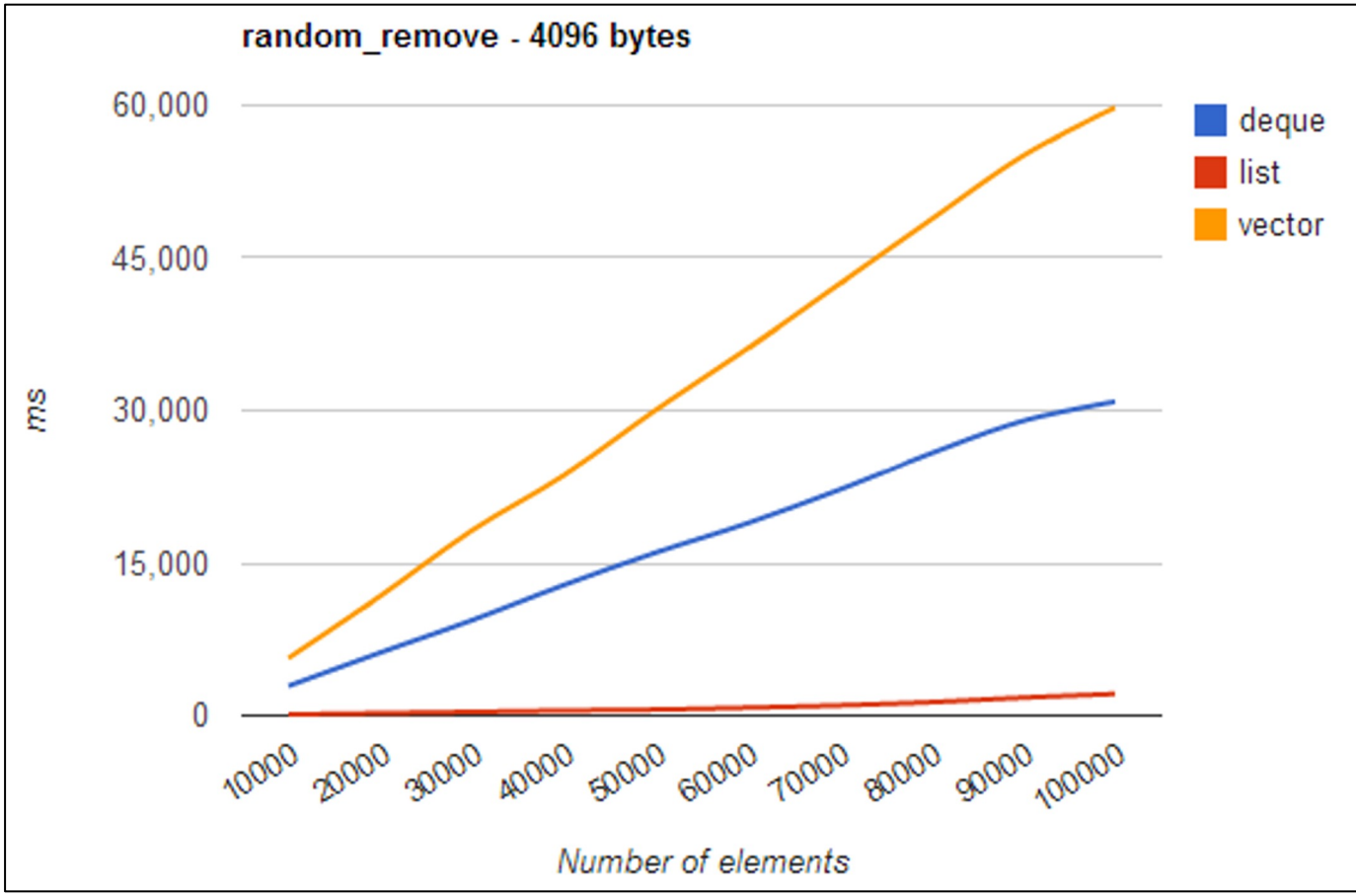
http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque

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

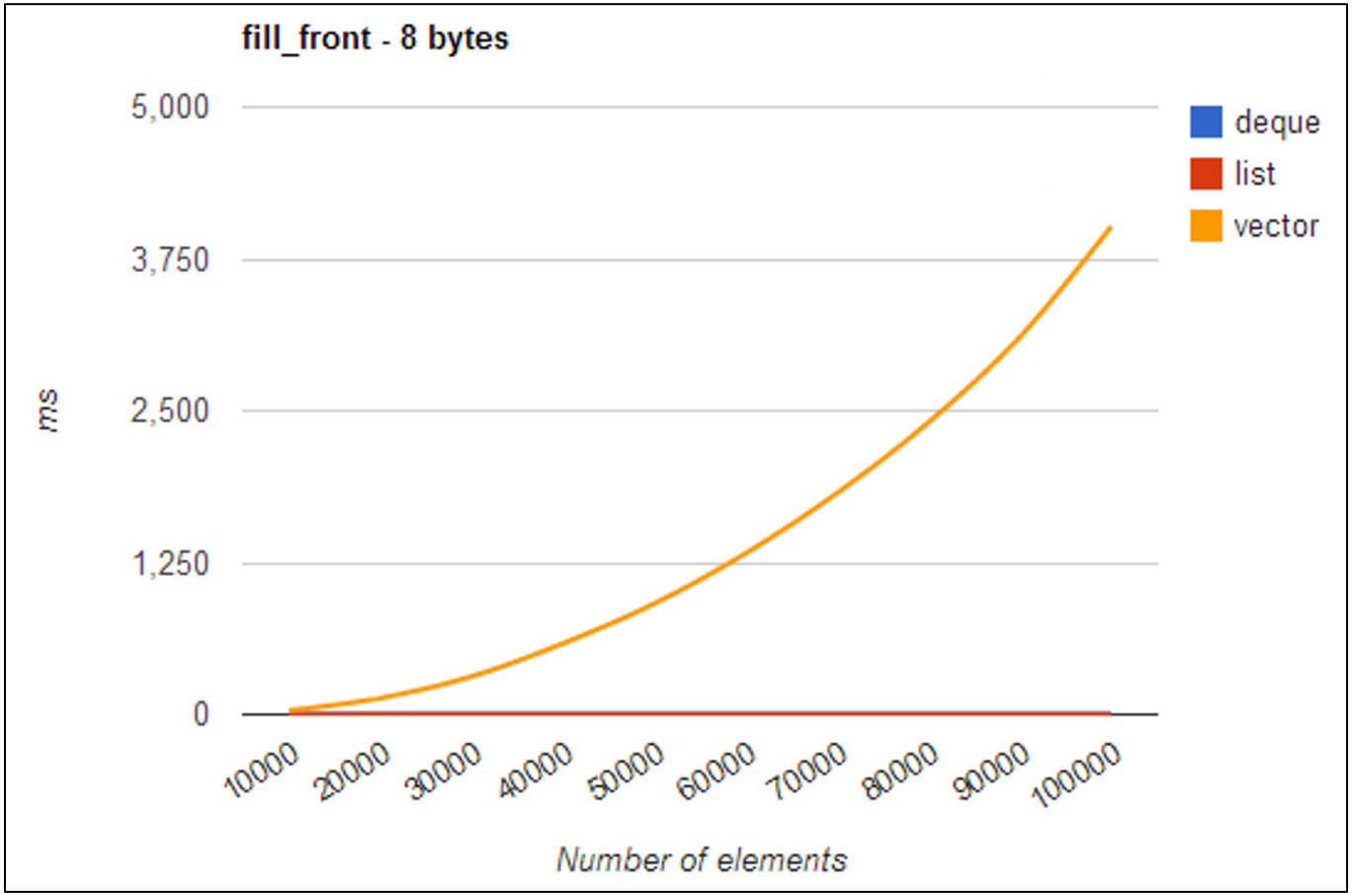


http://www.baptiste-wicht.com/2012/12/cpp-benchmark-vector-list-deque

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

STL Container Performance

Data Structures & Algorithms