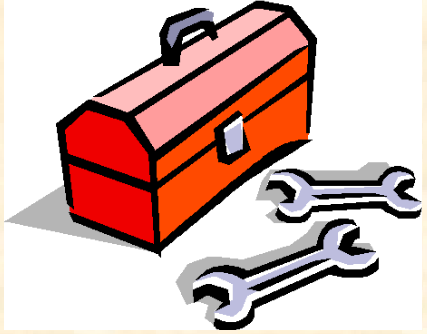
Lecture 6
The Standard Template Library

EEC S 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



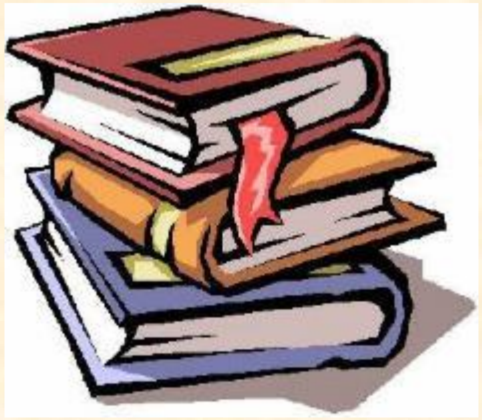
What is STL?

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

Contents of STL

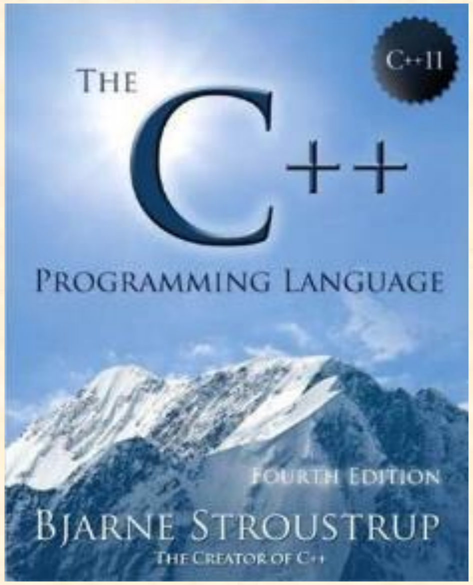
http://en.wikipedia.org/wiki/Standard_Template_Library

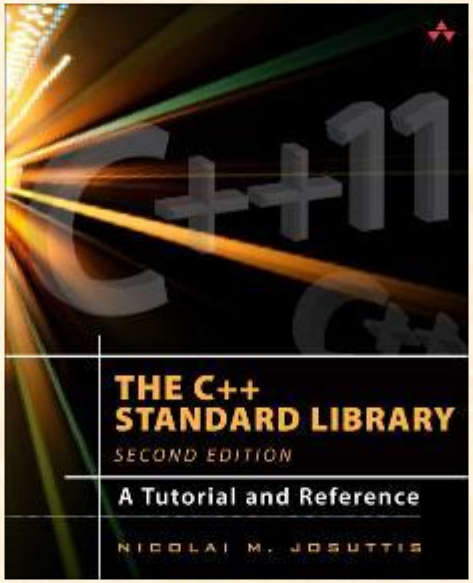
- Containers and iterators
- Memory allocators
- Utilities and function objects
- Algorithms
  - See http://www.sgi.com/tech/stl/ (not entirely up to date, but nicely presented)
  - See cppreference.com
  - See www.cplusplus.com/reference/stl/



STL Resources

- The C++ Language, 4e by Bjarne Stroustrup
- The C++ Standard Library: *A Tutorial and Reference,* 2e by Nicolai Josuttis (covers C++11)





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), `power<>()`
- 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 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
- 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
  - Users can copy/adapt examples
  - http://www.sgi.com/tech/stl
  - http://www.cplusplus.com/
- 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`

Features are used to *implement* STL

Fortunately, we will only *use* STL, not implement it

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
  - Often impossible in pure C

Performance and Big-O

- Most STL implementations have the best possible big-O complexities, given their interface
  - Example: `sort()`
- Notable Exceptions: `nth_element()` and linked lists

Main priority in STL is actual performance

It is very difficult to beat the speed of STL!

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<>`
- `map<>` and `multi_map<>`
- `list<>`
- `array<>`

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

Copying and Sorting Arrays

```
#include <vector>
#include <algorithm>
using namespace std;

const int N = 100;
int main() {
  vector<int> v(N, -1);
  int ar[N];

  for (unsigned 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());
}
```

(Not) Using Iterators

- Iterators generalize pointers
- Allow for implementation of same algorithm for multiple data structures
- Support concept of sequential containers

Don’t do this: leads to multiple errors due to ambiguity

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

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

```cpp
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;
}
```

- This code doesn't produce ambiguities when compiling
- 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

	Data structure	Memory overhead
	`vector<>`	Compact
	`list<>`	Not very compact
new in C++11	`unordered_map<>`	Memory hog

10 x 20 array

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

streamlined

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

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 `qsort` from stdlibc
- New in 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<()
  - Use helper class: a functional object

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

Index Sorting

```cpp
vector<int> idx(100);
vector<double> xCoord(100);
for (unsigned k = 0; k != 100; k++) {
    idx[k] = k;
    xCoord[k] = rand() % 1000 / 10.0;
}
// sbx is a function object!
SortByCoord sbx(xCoord)  // see below
sort(idx.begin(), idx.end(), sbx);
```

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

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

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

Try this!

Generating Random Permutations
(great for testing your program)

Container Example

```
srand(time(NULL));
vector<int> perm(N);
for (unsigned k = 0; k != N; k++)
  perm[k] = k;
random_shuffle(perm.begin(), perm.end());
```

Array Example

```
int perm[N];
for (unsigned k = 0; k != N; k++)
    perm[k] = k;
random_shuffle(perm, perm + N);
```

Filling a Container with Values

- Instead of using a loop, there is a utility function called `iota()`
  - Became standard with C++11

```
// Fill vector perm with values,
// starting at 0
iota(v.begin(), v.end(), 0);

// Fill array starting at 0
iota(perm, perm + N, 0);
```

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

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