COMSC-200 Lecture Topic 15
The Standard Template Library

Reference
Deitel Ch.21.1
anaturb.net
Nicolai Josuttis
informit.com

Why The STL In C++?
simplifies arrays, linked lists, maps
  map: table of key/value pairs
supports arrays of user-defined structs/classes
  auto-sizing and object-oriented
data type in class definition is *variable*

Advantages Of The STL
alternative to managing resizeable arrays
alternative to managing linked-list pointers
alternative to using inheritance and polymorphism
a high-performance solution
built-in functions for sorting and searching
the STL is *cleverly conceived*
  designed for performance and flexibility
more than one way to do something
  so choose the appropriate template

Components Of The STL
"containers": `vector`, `list`, and `map`
  `vector` is a resizable array
  `list` is a doubly-linked list
  `map` is a table of key/value pairs
"iterators": traversing pointers (const and non-const)
"algorithms": functions for sorting and searching
the header files:
```
<vector>, <list>, <deque>
<map>, <bitset>, <algorithm>
```

The STL For Java Programmers
STL: `vector`; Java: `ArrayList`
STL: `list`; Java: `LinkedList`
STL: `set`; Java: `TreeSet`
STL: `map`; Java: `TreeMap`
STL: `iterator`; Java: `Iterator`
STL: algorithms; Java: `Arrays` and `Collections`
BUT: Java stores refs to dynamically-allocated objects
  STL makes *object arrays*, possibly on the stack

Some Requirements For Using Objects
dynamic memory allocation in object:
  operator= and copy constructor
searching and sorting:
  operator== and operator<
const data members *and* assignment used in STL:
  operator=

The C++11 `array` Template Class
```
#include <array>
using std::array;
```
declaration: `array<int, 10> a;`

The `vector` Template Class
```
#include <vector>
using std::vector;
```
implementation is an array *of any data type*
  not just references, as in Java
constructors:
  as an array: `vector<int> a(10);` is like `int a[10];`
    requires default constructor for objects
  as linked list: `vector<int> b;` *initially empty list*
  using objects: `vector<Time> t(5, Time(12, 0, 0));`
member functions:
  `a.size();` #of accessible array elements
  `a.empty();` true if size is zero
  `a[i]` as getter and setter (don't use `a.at(i)`)
  `a.resize(...)` changes size
  `b.push_back(...)` expands size
size vs. capacity:
  `b.capacity()` *for internal purposes*
BEWARE: don't exceed array bounds with `a[i]`

Max/Min With Iterators
"super pointers"
needed by some algorithms, like...
finding the max/min value:
```
vector<int>::iterator i;
i = std::max_element(a.begin(), a.end());
if (i != a.end()) cout << *i;
```
  requires `operator<`
  or 3rd parameter: `bool(*)(T,T)`

Outputting The Contents
```
#include <iterator>
std::ostream_iterator<int> out(cout, " "); // tell it the delimiter
std::copy(a.begin(), a.end(), out);
```
works with arrays, too

Sorting
```
#include <algorithm> // for the sort function
std::sort(a.begin(), a.end());
```
  requires `operator<`
  or 3rd parameter: `bool(*)(T,T)`
works with arrays, too
```
int a[100];
std::sort(a, a+100);
```

Copying Elements Between `vector`s
`std::copy(b.begin(), b.end(), a.begin());` // copy `b` over `a`
  be sure not to copy too many elements!
works with arrays, too

Other Containers
vectors are good for adding to the *end*
vectors are not good for adding anywhere else
  but they are fast!
`deque<int>` is good for adding to either end
`list<int>` is good for adding anywhere
`set<int>` always in "order"

just like a static array
size must be know in advance
advantages:
array object "knows" its size
supports assignment (a = b;)

but they are not the fastest
functions: STL Quick Reference PDF
