Standard Template Library (STL)

Standard Template Library

- The STL is part of the standard C++ library
- The STL contains many class and function templates that may be used to store, search, and perform algorithms on data structures
- You should implement your own data structures and algorithms only if the ones provided in the STL do not suffice
- The STL consists of:
  - Container classes (data structures)
  - Iterators
  - Algorithms

Containers

- Sequence Containers - store sequences of values
  - ordinary C++ arrays
  - vector
  - deque
  - list
- Associative Containers - use "keys" to access data rather than position (Account #, ID, SSN, ...)
  - set
  - multiset
  - map
  - multimap
- Container Adapters - specialized interfaces to general containers
  - stack
  - queue
  - priority_queue

Sequence Containers: C++ arrays

- Fixed-size
- Quick random access (by index number)
- Slow to insert or delete in the middle
- Size cannot be changed at runtime
- Accessed using operator []

Sequence Containers: vector

- Resizable array
  - `#include <vector>`
  - `vector<string> vec;`
- Quick random access (by index number)
  - `operator [], at, front, back`
- Slow to insert or delete in the middle
  - `insert, erase`
- Quick to insert or delete at the end
  - `push_back, pop_back`
- Other operations
  - `size, empty, clear, …`

Sequence Containers: deque

- Like vector, but with quick insert and delete at both ends
- Resizable array
  - `#include <deque>`
  - `deque<string> dq;`
- Quick random access (by index number)
  - `operator [], at, front, back`
- Slow to insert or delete in the middle
  - `insert, erase`
- Quick to insert or delete at both ends
  - `push_front, pop_front, push_back, pop_back`
- Other operations
  - `size, empty, clear, …`

Sequence Containers: list

- Doubly-linked list
  - `#include <list>`
  - `list<string> lst;`
- Quick to insert or delete at any location
  - `insert, erase, push_front, pop_front, push_back, pop_back`
- Quick access at both ends
  - `front, back`
- Slow random access
  - no operator [], traverse using iterator
- Other operations
  - `size, empty, clear`
  - `reverse, sort, unique, merge, splice, …`

Associative Containers: set

- Stores a set of values (i.e., "keys")
- Values are unique (stored only once)
- Implemented as a balanced binary search tree
  - `#include <set>`
  - `set<string> s;`
- Fast insert and delete
  - `insert, erase`
- Fast search
  - `find`
- Other operations
  - `size, empty, clear, …`

Associative Containers: multiset

- Stores a set of values (i.e., "keys")
- Like set, but values need not be unique
- Implemented as a balanced binary search tree
  - `#include <set>`
  - `multiset<string> ms;`
- Fast insert and delete
  - `insert, erase`
- Fast search
  - `find`
- Other operations
  - `size, empty, clear, …`

Associative Containers: map

- Stores a set of (key, value) pairs
- Each key has one value
- Implemented as a balanced binary search tree
  - `#include <map>`
  - `map<string, int> m;`
- Fast insert and delete
  - `m["fred"] = 99;`
  - `insert, erase`
- Fast search
  - `int x = m["fred"];`
  - `find`
- Other operations
  - `size, empty, clear, …`

Associative Containers: multimap

- Stores a set of (key, value) pairs
- Like map, but each key can have multiple values
- Implemented as a balanced binary search tree
  - `#include <map>`
  - `multimap<string, int> mm;`
- Fast insert and delete
  - `insert, erase`
- Fast search
  - `find`
- Other operations
  - `size, empty, clear`

Associative Containers: sorting

- STL associative containers are implemented internally using a balanced BST
  - Key classes stored in associative containers must implement

    ```
    bool operator <(T other)
    ```

  - If they don't, you can alternatively pass a comparator class to the template that it should use to order elements
  - A comparator class overrides

    ```
    bool operator()(T a, T b)
    ```

Associative Containers: comparator example

```
set<Employee *> employees;    // BST sorts based on pointer values
                              // (probably not what you want)


class EmployeeComparator {
public:
   bool operator() (const Employee * a, const Employee * b) {
      return (a->GetID() < b->GetID());
   }
};

Set<Employee *, EmployeeComparator> employees;
                              // BST sorts based on employee IDs
                              // (much better!)
```

Container Adapters: stack

- Provides stack interface to other containers
  - `#include <stack>`
  - `stack<string> st;`
- Stack operations
  - `push, pop, top`
  - `size, empty, …`
- Can be used with vector, deque, or list
  - `stack<string> st; //uses a deque by default`
  - `stack< string, vector<string> > st;`
  - `stack< string, list<string> > st;`
- Extra space needed to avoid >>

Container Adapters: queue

- Provides queue interface to other containers
  - `#include <queue>`
  - `queue<string> q;`
- Queue operations
  - `push, pop, top`
  - `size, empty, …`
- Can be used with deque or list
  - `queue<string> q; //uses a deque by default`
  - `queue< string, list<string> > q;`
- Extra space needed to avoid >>

Container Adapters: priority_queue

- Provides priority queue interface to other containers
  - `#include <queue>`
  - `priority_queue<int> pq;`
- Priority queue operations
  - `push, pop, top`
  - `size, empty, …`
- Can be used with deque or vector
  - `priority_queue<int> pq; //uses a vector by default`
  - `priority_queue< int, deque<int> > pq;`
- Extra space needed to avoid >>

Iterators

- We need a way to iterate over the values stored in a container
- Iteration with C++ arrays:

```
const int SIZE = 10;
string names[SIZE];

for (int x=0; x < SIZE; ++x) {
   cout << names[x] << endl;
}

OR

string * end = names + SIZE;
for (string * cur = names; cur < end; ++cur) {
   cout << *cur << endl;
}
```

Iterators

- How do you iterate over the values stored in an STL container?
- For vectors and deques, you can iterate like this:

```
vector<string> names;

names.push_back("fred");
names.push_back("wilma");
names.push_back("barney");
names.push_back("betty");

for (int x=0; x < names.size(); ++x) {
   cout << names[x] << endl;
}
```

- This style of iteration doesn't work for the other container types

Iterators

- STL's solution to the iteration problem is based on iterators
- Iterators are pointer-like objects that that can be used to access the values in a container
- All containers have a method named `begin` that returns an iterator object that points to the first value in the container
- Iterator objects overload most of the pointer operators
  - `++, --` move the next or previous container value
  - `*, ->` access the value pointed to by the iterator
  - `==, !=` compare iterators for equality

Iterators

- How do you know when you've reached the end of the container's values?
- All containers have a method named `end` that returns a special iterator value that represents the end of the container (similar to a null pointer)

```
set<string> names;

names.insert("fred");
names.insert("wilma");
names.insert("barney");
names.insert("betty");

set<string>::iterator it;
for (it = names.begin(); it != names.end(); ++it) {
   cout << *it << endl;
}
```

Iterators

- In what order are the container's values returned by iterators?
- For sequences there is a natural first to last order
- For sets and maps the values are returned by doing an in-order traversal of the underlying binary search tree (i.e., the values are returned in sorted order)

Iterators

- You can also traverse a container in reverse order using reverse iterators and the `rbegin` and `rend` container methods

```
set<string> names;

names.insert("fred");
names.insert("wilma");
names.insert("barney");
names.insert("betty");

set<string>::reverse_iterator rit;
for (rit = names.rbegin(); rit != names.rend(); ++rit) {
   cout << *rit << endl;
}
```

Algorithms

- The STL provides many functions that can operate on any STL container
- These functions are called algorithms
- Some STL algorithms only work on certain containers
- #include <algorithm>

```
vector<string> names;

names.push_back("fred");
names.push_back("wilma");
names.push_back("barney");
names.push_back("betty");

unique(names.begin(), names.end());
sort(names.begin(), names.end());

vector<string>::iterator it;
for (it = names.begin(); it != names.end(); ++it) {
   cout << *it << endl;
}
```

Algorithms

```
class PrintFunc {
public:
   void operator ()(const string & s) const {
      cout << s << endl;
   }
};

vector<string> names;

names.push_back("fred");
names.push_back("wilma");
names.push_back("barney");
names.push_back("betty");

unique(names.begin(), names.end());
sort(names.begin(), names.end());

PrintFunc print;
for_each(names.begin(), names.end(), print);
```

Writing Classes That Work with the STL

- Classes that will be stored in STL containers should explicitly define the following:
  - default (no-arg) constructor
  - copy constructor
  - destructor
  - operator =
  - operator ==
  - operator <
- Not all of these are always necessary, but it might be easier to define them than to figure out which ones you actually need
- Many STL programming errors can be traced to omitting or improperly defining these methods

STL in Web Crawler

- StopWords - set<string>
- PageHistory - map<string, Page *>
- PageQueue - queue<Page *>
- WordIndex - map<string, set<Page *> >
- HTML element attributes - map<string, string>

STL in Web Cloner

- PageQueue - queue<Page *>
- PageHistory - map<URL, Page *>
- HTML element attributes - map<string, string>

STL in Chess

- Board - vector< vector<Square> >
- MoveHistory - stack<Move>
- Piece::GetCandidateMoves - set<BoardPosition>
- Game::GetLegalMoves - set<BoardPosition>
- XML element attributes - map<string, string>