C++
Introduction to STL:
The Standard Template Library

General Concepts

- STL defines a framework for defining composable library components
- it allows generic programming (containers and algorithms are generic)
- it defines some standard exceptions
- it defines containers, iterators, function objects, and algorithms
- programmers may extend it by obeying conventions
- http://www.stlport.org

Standard C++ Exceptions

- `exception` is the root
  - `bad_alloc` is thrown when global operator new fails
  - `bad_cast` is thrown when dynamic cast type doesn't match
  - `bad_typeid` is thrown when typeid is called on null pointer
  - `ios::failure` is thrown on I/O error
  - etc.
- http://www.cplusplus.com/reference/exception/exception/

STL exceptions

- STL extends these with its own exceptions
  - `logic_error` subclasses
    - domain error, e.g., violations of domain limits (e.g., positive)
    - invalid_argument, e.g., a bitset init requires string with 0s/1s
    - length_error, e.g., appending too many characters onto a string
    - out_of_range, e.g., indexing via operator [] out of bounds
  - `runtime_error`
    - `range_error`, e.g., function return value is erroneous
    - `overflow_error`, e.g., arithmetic overflow
    - `underflow_error`, e.g., arithmetic underflow

Containers

- containers hold collections of objects
- typically implemented as an array or a linked structure
- there are two general kinds of containers
  - sequence containers (ordered collections---position matters, e.g., vector index, list position)
    - `vector, deques, list`
  - associative containers (sorted collections---kept sorted by operations)
    - `set, multiset, map, multimap`

Vectors

- elements are kept in a dynamic array
- appending or removing at end is fast
- provides fast random access via `operator []`
- modifying in middle is more expensive

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  vector<int> v;
  for ( int i = 0; i < 6; ++i ) // fill v
    v.push_back(i); // appends to end of v
  for ( int i = 0; i < v.size(); ++i ) // print v
    cout << v[i] << endl;
}
```

- class `string` is similar to a `vector<char>`

Deques

- pronounced “deck”; short for *double ended queue*
- elements are kept in a dynamic array (possibly multiple scattered arrays, so no pointer arithmetic!)
- appending or removing at either end is fast
- provides fast random access via `operator []`
- modifying in middle is more expensive
- note: modifying the deque invalidates all iterators pointing at it

```
#include <iostream>
#include <deque>
using namespace std;
int main()
{
  deque<int> d;
  for ( int i = 0; i < 6; ++i ) // fill d
    d.push_front(i); // appends to front of d
  d.push_back(10); // appends 10 to back of d
  for ( int i = 0; i < d.size(); ++i ) // print d
    cout << d[i] << endl;
}
```

Lists

- implemented as a doubly linked list of elements
- *does not* provide random access via `operator []`
- efficient insert/removal of any element

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
  list<char> L;
  for ( char c = 'A'; c <= 'Z'; ++c ) // fill L
    L.push_back(c);
  for ( ; ! L.empty(); L.pop_front() )
    cout << L.front() << endl;
}
```

Iterators

- are used to step through the elements of a collection of objects
- collections may be containers or subsets of containers
- collections all return iterators
  - `begin()` - the start of the collection
  - `end()` - one past the end of the collection
- defines a small, common interface to any container
- similar to pointer arithmetic on arrays

Iterator Operations

- `operator *` returns the element of the current position
- `operator ->` allows member selection from current position
- `operator ++` moves to the next element
- `operator --` moves to the previous element
- `operator ==, operator !=` compare for equality
- `operator =` assigns an iterator

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
  list<char> L;
  for ( char c = 'A'; c <= 'Z'; ++c ) // fill L
    L.push_back(c);
  for ( list<char>::iterator p = L.begin(); p != L.end(); ++p )
    cout << *p << endl;
}
```

Container Iterator

- every container `C` defines two nested types
  - `C :: iterator`
    - iterate in read/write mode
  - `C :: const_iterator`
    - iterate in read-only mode

Iterator Categories

- STL provides iterators that provide good performance for their representation
- *bidirectional iterators*: `++`, `--`
  - e.g., `list`, `set`, `multiset`, `map`, `multimap`
- *random access iterators*, bidirectional plus `operator []`
  - e.g., `vector`, `deque`, `string`
- there are other categories, e.g., for file I/O

Iterator Adapters

- iterator adaptors allow modification to other iterators
- STL provides several predefined iterator adaptors

Insert Iterator

- *insert iterator*, AKA *inserters*
- allows an algorithm to insert rather than overwrite
- may insert at front, end, or at a given position

```
#include <...>
int main()
{
  list<int> L;
  for ( int i = 1; i <= 10; ++i )
    L.push_back( i );
  vector<int> V;
  copy( L.begin(), L.end(), // source
      back_inserter(V) ); // destination calls  push_back
  deque<int> D;
  copy(L.begin(), L.end() // source
     front_inserter(D) ) // destination calls  push_front
  set<int> S;
  // only  inserter works on associative containers (no order)
  copy(L.begin(), L.end(), // source calls  insert
     inserter(S, S.begin()) ); // destination, arg 2 is position
}
```

Stream Iterator

a *stream iterator* works on an I/O stream

```
#include <...>
int main()
{
  vector<string> V;
  // places all words from cin into V
  copy( istream_iterator<string>(cin), // start of source
      istream_iterator<string>(), // end of source, EOF
      back_inserter(V) ); // destination
  sort( V.begin(), V.end() ); // sort all the words
  // print all words (without duplicates) to cout one per line
  unique_copy( V.begin(), V.end(), // source
        ostream_iterator<string>(cout, "\n") ); // dest
}
```

Reverse Iterator

- *reverse iterator* switch increment to decrement and vice versa
- all containers can create reverse iterators via `rbegin()` and `rend()`.

```
#include <...>
int main()
{
  vector<int> V;
  for ( int i = 1; i <= 10; ++i )
    V.push_back( i );
  // print all elements to cout in reverse order separated
by space
  copy( V.rbegin(), V.rend(),
      ostream_iterator<int>(cout, " ") );
}
```

Associative Containers

- elements are kept in a sorted order
- must have `less<T>` defined, defaults to `operator <`
- two elements are `==` if neither is less than the other
- typically implemented as a binary search tree
- *O(log N)* for insert and lookup

Sets and Multisets

- elements are sorted by their own value
- sets: each element is unique (no duplicates)
- multisets: same as sets, but duplicates are allowed

```
#include <iostream>
#include <set>
using namespace std;
int main()
{
  set<char> S;
  for ( char c = 'A'; c <= 'Z'; ++c ) // fill S
    S.insert(c);
  for ( char c = 'A'; c <= 'Z'; ++c ) // try to fill S again
    S.insert(c); // no insertion performed
  for ( set<char>::iterator p = S.begin(); p != S.end(); ++p )
    cout << *p << endl; // only see each character once
}
// change set S to multiset and you'll see duplicates
```

Pairs

- used by some of the STL containers

```
template
  <typename T1, typename T2>
struct pair
{
  T1 first;
  T2 second;
  pair(const T1 & a, const T2 & b)
    : first(a), second(b)
  {
  }
  // operators ==, <, >, etc
};
template
  <class T1, class T2>
pair<T1,T2> make_pair(const T1 & a, const T2 & b)
{
  return pair<T1,T2>(a,b);
} // allows easy building of pairs (vs
  //constructor, as types need not be specified)
```

Maps and Multimaps

- elements are key/value pairs
- also known as *associative arrays*
- like an array, but indexed by any type (key type)
- elements are sorted by their keys
- maps: each element is unique (no duplicates)
- multimaps: same as maps, but each key may have multiple values
- multimaps are sometimes called a *dictionary*

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main()
{
  multimap<int,string> M;
  M.insert( make_pair(5,"tagged") );
  M.insert( make_pair(2,"a") );
  M.insert( make_pair(1,"this") );
  M.insert( make_pair(4,"of") );
  M.insert( make_pair(6,"strings") );
  M.insert( make_pair(1,"is") );
  M.insert( make_pair(3,"multimap") );
  for ( multimap<int,string>::iterator p = M.begin(); p != M.end(); ++p )
    cout << p->second << endl;
  // prints: this is a multimap of tagged strings
  // or: is this a multimap of tagged strings
}
```

- change `multimap M` to a `map`, and output will be
  `// is a multimap of tagged strings`

Maps as Associative Arrays

can use maps as arrays indexed by strings

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main()
{
  map<string,int> A;
  A["Bill"] = 53;
  A["George"]  = 49;
  A[”Arthur"] = 47;
  for ( map<string,int>::iterator p = A.begin(); p != A.end(); ++p )
    cout << p->first << " is " << p->second;
  // prints: Arthur is 47, Bill is 53, George is 49
}
```

Container Adapters

- not containers per se, rather they wrap around an existing container
- `stack`
  - elements are managed in Last-In-First-Out (LIFO) order
- `queue`
  - elements are managed in First-In-First-Out (FIFO) order
- `priority_queue`
  - elements are managed in highest-value-first-out
  - `operator <` is used by default

Algorithms

- used to process elements of collections
- they can search, sort, modify, or use elements
- they use iterators, so they work on all containers

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
  vector<int> V;
  V.push_back(2);
  V.push_back(5);
  V.push_back(4);
  V.push_back(1);
  V.push_back(6);
  V.push_back(3);
  vector<int>::iterator pos;
  pos = min_element( V.begin(), V.end() );
  cout << "Min = " << *pos << endl;
  pos = max_element( V.begin(), V.end() );
  cout << "Max = " << *pos << endl;
  sort( V.begin(), V.end() );
  pos = find( V.begin(), V.end(), 2 ); // finds value 2
  reverse( pos, V.end() );
}
```

Kinds of Algorithms

- main kinds are:
  - non-modifying, modifying, removing, mutating, sorting, union/ intersection, numeric
- key suffixes:
  - `*_if` takes a function object and applies if function returns true
    - `find` searches for an element based on value `val`

      ```
      InputIterator find (InputIterator first, InputIterator last, const T& val)
      ```

    - `find_if` searches for an element satisfying a supplied predicate `pred`

      ```
      InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred)
      ```

  - `*_copy` elements are copied into a destination
    - `reverse` reverses elements inside a range

      ```
      void reverse (BidirectionalIterator first, BidirectionalIterator last)
      ```

    - `reverse_copy` copies elements into another range in reverse
- numeric algorithms must `#include <numeric>`
- other algorithms must `#include <algorithm>`
- includes `min, max, swap`

Function Objects

- class objects that behave like functions
- they define `operator ()`
- predefined function objects include
  - `less<T>, greater<T>, negate<T>, multiplies<T>`, etc.
  - must `#include <functional>`

```
#include <...>
#include <functional>
int main()
{
  set<int> S;
  // ... fill S
  transform( S.begin(), S.end(), // first source
         S.begin(), // second source
         S.begin(), // destination
         multiplies<int>() ); // operation
}
```

- function objects may be composed

Functions (or Function Objects) as Algorithm Arguments

some algorithms take functions as arguments

for_each

```
#include <...>
void print(int i)
{
  cout << i << ' ';
}
int main()
{
  list<int> L;
  for ( int i = 1; i <= 5; ++i )
    L.push_back( i );
  // print all ints in the list
  for_each( L.begin(), L.end(), print );
}
```

transform

```
#include <...>
int square(int i)
{
  return i * i;
}
int main()
{
  list<int> L1, L2;
  for ( int i = 1; i <= 5; ++i )
    L1.push_back( i );
  // puts squares of L1 into L2
  transform( L1.begin(), L1.end(),
back_inserter(L2), square );
```

sort

```
#include <...>
struct Person {
  string first;
  string last;
}
bool lessThan( const Person & p1,
const Person & P2 )
{
  return p1.last < p2.last ||
p2.last == p2.last
&& p1.first < p2.first;
}

int main()
{
  list<Person> people;
  //...
  sort( people.begin(),
people.end(), lessThan );
  //...
}
```

Non-modifying Algorithms

- `for_each()`
  - performs an operation on each element
- `count()`
  - returns the number of elements
- `count_if()`
  - returns the number of elements satisfying a predicate
- `min_element()`
  - returns the smallest valued element
- `max_element()`
  - returns the largest valued element
- `find()`
  - returns the position of the given value
- `find_if()`
  - returns the first element that satisfies a predicate
- `equal()`
- and more... http://www.cplusplus.com/reference/algorithm/

Modifying Algorithms

- `copy()`
  - copies a range
- `transform()`
  - modifies and copies elements according to a specified function
- `merge()`
  - joins two ranges
- `swap_ranges()`
  - swaps elements from two ranges
- `fill()`
  - replaces each element with a specified value
- `generate()`
  - replaces each element with result of a function
- `replace()`
  - replaces each element with specified value with another specified value
- `replace_if()`
  - replaces elements which satisfy a predicate with a specified value
- and more…

Removing Algorithms

- `remove()` will remove specified elements from the collection (shifts elements forward) but does not affect the size!
- `erase()` will remove specified elements from the collection and shrink the container

```
#include <...>
int main()
{
  list<int> L;
  for ( int i = 1; i <= 5; ++i )
  {
    L.push_back( i );
    L.push_front( i );
  }
  // remove all 1s
  list<int>::iterator end = remove( L.begin(), L.end(), 1 ); // moves elements forward
  cout << "Number of elements removed: " << distance( end, L.end() );
  L.erase( end, L.end() ); // delete elements in “limbo” off the end
  // remove doesn't erase automatically
  // better to do the following:
  L.erase( 2 ); // delete all 2s from L
}
```

- `remove()`
  - removes all elements with given value
- `remove_if()`
  - removes all elements matching predicate
- `remove_copy()`
  - copies elements that do not match a given value
- `remove_copy_if()`
  - copies elements that do not match a predicate
- `unique()`
  - removes adjacent duplicates
- `unique_copy()`
  - copies elements while removing adjacent duplicates

Mutating Algorithms

- `reverse()`
  - reverses order of the elements
  - `reverse (BidirectionalIterator first, BidirectionalIterator last)`
- `reverse_copy()`
  - reverses order of the elements into another container
  - `reverse_copy (BidirectionalIterator first, BidirectionalIterator last, OutputIterator result)`
- `rotate()`
  - shifts them one to the right with wrap around to the front
- `rotate_copy()`
  - copies elements while rotating
- `next_permutation()`
  - permutates the order of the elements
- `prev_permutation()`
  - permutates the order of the elements
- `random_shuffle()`
  - moves the elements into a random order
- and more...

Sorting Algorithms

- `sort()`
  - sorts elements in range
- `stable_sort()`
  - preserves order of equal elements
- `partial_sort()`
  - sorts until the first N elements are in order
- `partial_sort_copy()`
  - copies elements in sorted order
- `nth_element()`
  - sorts around the Nth position
- `make_heap()`
- `push_heap()`
- `pop_heap()`
- `sort_heap()`
  - heap sort operations
- and more...

Algorithms on Sorted Ranges

- `binary_search()`
  - find element in range
- `includes()`
  - true if elements of one range are all in another range
- `lower_bound()`
  - finds the first element <= a specified value
- `upper_bound()`
  - finds the first element >= a specified value
- `equal_range()`
  - returns the range of elements equal to a given value
- `merge()`
  - merges two ranges together
- and more…

Numeric Algorithms

- `accumulate()`
  - combine all element values; by default, sum them up
- `inner_product()`
  - combines all elements of two ranges
- `adjacent_difference()`
  - combines each element with its immediate predecessor
- `partial_sum()`
  - combines each element with all of its predecessor