**C++**

**Introduction to STL:**

**The Standard Template Library**

**Raymond Klefstad, Ph.D.**

**STL General Concepts**

* STL defines a framework for defining composable library components
* it allows generic programming (containers and algorithms are type-generic)
* it defines some standard exceptions (at the bottom)
* it defines containers, iterators, function objects, and algorithms
* programmers may extend STL by obeying conventions
* Largely the creation of [Alexander Stepanov](http://en.wikipedia.org/wiki/Alexander_Stepanov)

**Containers**

* containers hold collections of objects
* typically implemented as an array or a linked structure (e.g., list or tree)
* there are two general kinds of containers
  + **sequence containers** (ordered collections)
    - [vector](http://www.cplusplus.com/reference/vector/vector/), [list](http://www.cplusplus.com/reference/list/list/), deque, [array](http://www.cplusplus.com/reference/array/array/)
  + **associative containers** (sorted collections)
    - set, multiset
    - map, multimap
    - hash\_set, hash\_map, hash\_multiset, hash\_multimap

**Vectors**

* elements are kept in a dynamic array [Link](http://en.wikipedia.org/wiki/Sequence_container_%28C%2B%2B%29#Vector)
* appending or removing at back 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;  
  }

**Lists**

* implemented as a doubly-linked list of elements [Link](http://en.wikipedia.org/wiki/Sequence_container_%28C%2B%2B%29#List)
* 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;  
  }

**Deques**

* short for  *double ended queue*
* elements are kept in a dynamic array [Link](http://en.wikipedia.org/wiki/Sequence_container_%28C%2B%2B%29#Deque)
* 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;  
  }

**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 common interface to any container
* syntax is 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 != - compares 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 = **begin(L)**; p != **end(L)**; **++p** )  
   cout << **\*p** << endl;

for ( **auto E : L** ) // from C++11, E is a copy of each element  
 cout << **E** << endl;  
 for ( **auto&& E : L** ) // E is a reference to each element  
 cout << **E** << endl;  
 }

Live Quiz P1

**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 (**[**link**](http://en.cppreference.com/w/cpp/iterator)**)**

* STL provides iterators that provide good performance for their representation

1. ***input iterators*** - can only be used to read a sequence of values
2. ***output iterators*** - can only be used to write a sequence of values
3. ***forward iterators*** - can be read, written to, and move forward ++
4. ***bidirectional iterators*** - can do both ++, --
   1. EG list, set, multiset, map, multimap
5. ***random access iterators*** - bidirectional plus operator []
   1. EG vector, deque, string
6. ***contiguous Iterators*** (C++ 17) - logical adjacent elements are also phyiscally adjacent
   1. . EG array, string, vector

**Iterator Adapters**

* iterator adaptors allow modification to other iterators
* STL provides several predefined iterator adaptors
* insert iterator, stream iterator, reverse iterator

**Insert Iterator Adapter**

* *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( begin(L), end(L), // source  
   **back\_inserter(V)** ); // destination calls  *V.push\_back*  
   deque<int> D;  
   copy( begin(L), end(L) // source  
   **front\_inserter(D)** ) // destination calls  *D.push\_front*  
   set<int> S;  
   // *inserter* only works on associative containers  
   copy( begin(L), end(L), // source calls  *S.insert*  
   **inserter( S, begin(S) )** ); // destination, arg 2 is position  
  }

Live Quiz P2

**Stream Iterator Adapter**

* 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( begin(V), end(V) ); // sort all the words  
   // print all words (without duplicates) to cout one per line  
   unique\_copy( begin(V), end(V), // source  
   **ostream\_iterator<string>(cout, "\n")** ); // dest

}

**Stream Iterator Adapter with Files**

* works with ifstream and ofstream   
  #include <iostream>  
  #include <fstream>  
  #include <iterator>  
  int main()  
  {  
   ifstream in(“input.data”);  
   ofstream out(“output.data”);  
   vector<string> V;  
   // places all words from cin into V  
   copy( **istream\_iterator<string>(in)**, // start of source  
   **istream\_iterator<string>()**, // end of source, EOF  
   back\_inserter(V) ); // destination  
   sort( begin(V), end(V) ); // sort all the words  
   // print all words (without duplicates) to cout one per line  
   unique\_copy( begin(V), end(V), // source  
   **ostream\_iterator<string>(out, "\n")** ); // dest  
  }

Live Quiz P3

**Reverse Iterator Adapter**

* *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( **rbegin(V)**, **rend(V)**,  
   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 balanced binary search tree (red black tree)
* gives O(log N) for insert and lookup

**Sets and Multisets**

* elements are sorted by their own value
* **set**s: each element is unique (no duplicates) [Link](http://www.cplusplus.com/reference/set/set/)
* **multiset**s: 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 = 'Z'; c >= 'A'; --c ) // fill S again  
   **S.insert(c);**  
   for ( auto E : S )  
   cout << E << endl; // only see each character once  
   if ( S.find(‘A’) != S.end() )  
   cout << “A is in there” << endl;  
  }
* change S to a multiset, and we'll see duplicates

XXX**Pairs**

* used by some of the STL containers e.g., **map**s  
  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 // allows easy building of pairs  
   <class T1, class T2>  
  pair<T1,T2> **make\_pair**(const T1 & a, const T2 & b)  
  {  
   return pair<T1,T2>(a,b);  
  }

**Maps and Multimaps**

* elements are key/value pairs
* like an array, but indexed by any type (often strings)
* elements are sorted by their keys (not values)
* **map**s: each element is unique (no duplicates)
* **multimap**s: same as maps, but each key may have multiple values
* **multimap**s 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 ( auto E : M )  
   cout << E.second << endl;  
   // prints: this is a multimap of tagged strings  
   // or: is this a multimap of tagged strings  
  }
* change 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 (often very useful)  
   #include <iostream>  
   #include <map>  
   #include <string>  
  using namespace std;  
  int main()  
  {  
   map<string,int> A;  
   A["Bill"] = 33;  
   A["George"] = 29;  
   A["Al"] = 27;  
   for ( auto E : A )  
   cout << E.first << " is " << E.second;  
   // prints: Al is 27, Bill is 33, George is 29  
  }

**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{2,5,4,1,6,3};  
   vector<int>::iterator pos;  
   pos = **min\_element**( begin(V), end(V) );  
   cout << "Min = " << \*pos << endl;  
   pos = **max\_element**( begin(V), end(V) );  
   cout << "Max = " << \*pos << endl;  
   **sort**( begin(V), end(V) );   
   pos = **find**( begin(V), end(V), 2 ); // finds value 2  
   **reverse**( pos, end(V) );  
  }

**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( begin(S), end(S), // first source  
     begin(S), // second source  
     begin(S), // destination  
     multiplies<int>() ); // operation  
    }
* function objects may be composed, include
* **arithmetic operations** (plus, minus, multiplies, divides, modulus, and negate)
* **comparisons** (equal\_to, not\_equal\_to, greater, less, greater\_equal, and less\_equal)
* **logical operations** (logical\_and, logical\_or, and logical\_not)

**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( begin(L), end(L), 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( begin(L1), end(L1), back\_inserter(L2), square );  
  }

Live Quiz P4

* 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( begin(people), end(people), lessThan );  
   //...  
  }

**Lambdas (Anonymous Functions)**

* You can define local function objects as a lambda
* E.g.,

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

[]( const Person & p1, const Person & p2 ){return p1.last < p2.last || p2.last == p2.last && p1.first < p2.first;} );  
 //...  
}

int main()  
{  
 ifstream in("input.data");  
 ofstream out("output.data");  
 map<string, int> M;  
 // places all words from cin into M  
 for\_each( **istream\_iterator<string>(in)**, // start of source  
 **istream\_iterator<string>()**, // end of source, EOF  
 [&](string & s){ ++ M[s]; } );  
 ...

}

* options for []
  + [] means capture nothing
  + [&] means capture referenced variables by reference
  + [=] means capture referenced variables by copy
  + [=,&M] means capture M by ref others by copy
  + [M] means capture M by copy, don’t capture anything else

Live Quiz P5

**Non-modifying Algorithms**

* for\_each()
  + performs an operation on each element
* count(), count\_if()
  + returns the number of elements satisfying a predicate (function returns non-zero)
* min\_element()
  + returns the smallest valued element
* max\_element()
  + returns the largest valued element
* find(), find\_if()
  + returns the first element that satisfies a predicate
* equal()
* and more...

**Modifying Algorithms**

* copy(), copy\_if()
  + 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(), replace\_if()
  + replaces each element with specified value with another specified value
* and more...

**Removing Algorithms**

* remove() will remove specified elements from the collection
* however, container methods, like 'erase', should be favored   
   #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 endx =  
   remove( begin(L), end(L), 1 ); // moves elements forward  
   cout << "Number elements removed: " << distance( end, end(L) );  
   L.erase( endx, end(L) ); // delete elements off the end  
   // remove doesn't erase automatically  
   // better to do the following:  
   L.erase( 2 ); // delete all 2s from L  
  }
* remove(), remove\_if()
  + removes all elements matching predicate
* remove\_copy(), 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\_copy()
  + reverses order of the elements into another container
* 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()
  + partitions around the Nth position (like qsort around pivot)
* make\_heap()
* push\_heap()
* pop\_heap()
* sort\_heap()
  + heap sort operations
* and more...

**Algorithms on Sorted Ranges**

* binary\_search()
  + find element in range [Link](http://en.cppreference.com/w/cpp/algorithm/binary_search) lower\_bound() does the work [Link](http://en.cppreference.com/w/cpp/algorithm/lower_bound)
* includes()
  + true if elements of one range are all in another range
* lower\_bound()
* 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 [Link](http://en.cppreference.com/w/cpp/algorithm/merge)
* and more...

**Numeric Algorithms**

* accumulate() [Link](http://en.cppreference.com/w/cpp/algorithm/accumulate)
  + combine all element values (processes sum, product, etc)
* inner\_product()
  + sum of products of corresponding elements of two ranges
* adjacent\_difference()
  + result[0] = x[0]; result[i] = x[i] - x[i-1]
* partial\_sum()
  + combines each element with all of it's predecessor

**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
  + bad\_exception is thrown if thrown exception isn't in throw spec
  + ios::failure is thrown on I/O error

**STL exceptions**

* STL extends these with its own exceptions
  + logic\_error
    - domain error, EG violations of domain limits (e.g., positive)
    - invalid\_argument, EG a bitset init requires string with 0s/1s
    - length\_error, EG appending too many characters onto a string
    - out\_of\_range, EG indexing via operator [] out of bounds
  + runtime\_error
    - range\_error, EG function return value is erroneous
    - overflow\_error, EG airthmetic overflow
    - underflow\_error, EG arithmetic underflow

**Useful Links**

* <http://www.josuttis.com/libbook/toc.html>
* <http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html>
* http://www.josuttis.com/libbook/cont/carray.hpp.html