Lecture Material

- Standard C++ library
  - STL (*Standard Template Library*)

STL – General View

- STL – library of reusable components
  - Meant to provide support for C++ development with containers, algorithms, iterators, etc.
- Easy to use and very powerful (and efficient)
- Not OOP, but generic programming
- http://en.cppreference.com/w/cpp

Containers	Classes that contain other objects
Iterators	"Pointers" into containers, used as index into containers
Adaptors	Classes that "adapt" other classes
Allocators	Objects for allocating memory

Some of the Containers in STL

vector<T>	Random access, varying length, constant time insert/delete at end
deque<T>	Random access, varying length, constant time insert/delete at either end
list<T>	Linear time access, varying length, constant time insert/delete anywhere in list
stack<T>	Usual stack implementation
set<Key>	Collection of unique Key values
map<Key,T>	Collection of T Values indexed by unique Key values

Common in Most Containers

- Some common member functions in most containers, for example
  - *size()* returns the number of elements in a container
  - *push_back()* adds objects at the "end" of a container
- Access to data in containers
  - direct access to data via *operator[ ]* or *at()* member function
- Iterators
  - way of accessing elements in the container, using a for loop with an "index"
  - several available, forward, backward, const, etc.

STL Vector Container

- The STL *vector* mimics the behavior of a dynamically allocated array and also supports automatic resizing at runtime (if you add data via the *insert* and *push_back*).

vector declarations:	`vector<int> iVector;`
	`vector<int> jVector(100);`
	`vector<int> kVector(Size); // Size is int var`

vector element access:	`jVector[23] = 71; // set member`
	`jVector[41]; // get member`
	`jVector.at(23); // get member`
	`jVector.front(); // get first member`
	`jVector.back(); // get last member`

vector reporters:	`jVector.size(); // num elements in container`
	`jVector.capacity(); // capacity of container`
	`jVector.max_capacity(); // max capacity of elements`
	`jVector.empty();`

vector Constructors

- The *vector* template provides several constructors:
  - `vector<T> V; //empty vector`
  - `vector<T> V(n,value);`
    `//vector with n copies of value`
  - `vector<T> V(n);`
    `//vector with n copies of default for T`
- The *vector* template also provides a suitable deep copy constructor and assignment overload.

vector Example

```
#include <iostream>
#include <vector> // for vector template definition
using namespace std;

int main() {
 int MaxCount = 100;
 vector<int> iVector(MaxCount);
 for (int Count = 0; Count < MaxCount; Count++) {
   iVector[Count] = Count;
 }
}
```

Initial vector size

Access like an array

- Warning: the capacity of this vector will NOT automatically increase as needed if access is performed using the [] operator. Using *insert()* and *push_back()* to add members in the array will grow the vector as needed.

STL *vector* Indexing

- In the simplest case, a vector object may be used as a simple dynamically allocated array:

```
int MaxCount = 100;
vector<int> iVector(MaxCount);
...
for (int Count = 0; Count < 2*MaxCount; Count++) {
 cout << iVector[Count];
```

Efficiency

  - No runtime checking of the vector index bounds
  - No dynamic growth. Errors produce an access violation (if we are lucky).

```
int MaxCount = 100;
vector<int> iVector(MaxCount);
...
for (int Count = 0; Count < 2*MaxCount; Count++) {
 cout << iVector.at(Count);
```

Safety

- Use of the *at()* member function causes an *out_of_range* exception in the same situation.

STL Iterators

- Iterator
  - An object that keeps track of a location within an associated STL container object, providing support for traversal (increment/decrement), dereferencing, and container bounds detection.
  - An iterator is declared with an association to a particular container type and its implementation is both dependent upon that type and of no particular importance to the user.
  - Iterators are fundamental to many of the STL algorithms and are a necessary tool for making good use of the STL container library.
  - Each STL container type includes member functions *begin()* and *end()* which effectively specify iterator values for the first element and for "one-past-end" element.

vector Iterator

- The STL *vector* iterator mimics the behavior of pointer access to a dynamically allocated array.

iterator declaration:	`vector<int>::iterator idx;` `vector<int> jVector;`

access iterator from vector:	`jVector.begin(); // gets iterator`
	`jVector.end(); // gets sentinel (iterator)`

vector element access via iterator:	`idx[i]; // access ith element`
	`*idx; // access to element pointed by idx`
	`++idx; // moves pointer to next element`
	`--idx; // moves pointer to previous element`

```
vector<T> v;
vector<T>::iterator idx;
for (idx = v.begin(); idx != v.end(); ++idx)
    do something with *idx
```

```
vector<T> v;
for (auto idx = v.begin(); idx != v.end(); ++idx)
    do something with *idx
```

```
vector<T> v;
for (auto& i : v)
   do something with i
```

Types of Iterators

- Different containers provide different types of iterators
  - Forward iterator - defines ++ only
  - Bidirectional - define ++ and -- on iterator
  - Random-access - define ++, -- and [x]
    - Addition, subtraction of integers: r+n, r-n
    - Jump by integer n: r+=n, r-=n
    - Iterator subtraction r - s yields integer
    - Has an indexing operator []
  - Constant and mutable iterators
    - Constant iterators - *p does not allow you to modify the element in the container
    - Mutable allows you to edit the container

```
for (p = v.begin(); p != v.end(); ++p)
     *p = new value
```

  - Reverse iterator, allows to traverse container from end to beginning

```
reverse_iterator rp;
for (rp = v.rbegin(); rp != v.rend(); ++rp)
     process *rp
```

Constant Iterators

- Constant iterator must be used when object is const – typically for parameters.
- Type is defined by container class: *vector<T>::const_iterator*

```
void ivecPrint(const vector<int>& V, ostream& Out) {
 vector<int>::const_iterator It; // MUST be const

 for (It = V.begin(); It != V.end(); ++It) {
   cout << *It;
 }
 cout << endl;
}
```

STL *vector* Iterator Example

- The example below makes a copy of the *BigInt* vector

```
string DigitString = "45658228458720501289";
vector<int> BigInt;

for (int i = 0; i < DigitString.length(); i++) {
 BigInt.push_back(DigitString.at(i) - '0');
}
vector<int> Copy;
vector<int>::iterator It;
for (It = BigInt.begin(); It != BigInt.end(); ++It) {
 Copy.push_back(*It);
}
```

Advance the iterator to the next element.

Iterator initialization

Sentinel value.

- The vector *Copy* is initially empty. *push_back()* will enlarge target vector to the appropriate size
- We use prefix, and not suffix, iterator incrementation operator

STL Iterator Operations

- Each STL iterator provides certain facilities via a standard interface:

```
string DigitString = "45658228458720501289";
vector<int> BigInt;

for (int i = 0; i < DigitString.length(); ++i) {
 BigInt.push_back(DigitString.at(i) - '0');
}

vector<int>::iterator It;

It = BigInt.begin();
int FirstElement = *It;

++It;

It = BigInt.end();

--It;
int LastElement = *It;
```

Create an iterator for *vector<int>* objects.

Target the first element of *BigInt* and copy it.

Step to the second element of *BigInt*.

Now *It* targets a non-element of *BigInt*.
Dereferencing *It* can yield an access violation.

Back *It* up to the last element of *BigInt*.

Insertion into *vector* Objects

- Insertion at the end of the vector (using *push_back()*) is most efficient.
  - Inserting elsewhere requires shifting data in memory.
- A *vector* object is potentially like array that can increase size.
- The capacity of a vector e.g. doubles in size if insertion is performed when vector is "full".
- Insertion invalidates any iterators that target elements following the insertion point.
- Reallocation (enlargement) invalidates any iterators that are associated with the vector object.
- You can set the minimum size of a vector object V with *V.reserve(n)*.

insert() Member Function

- An element may be inserted at an arbitrary position in a vector by using an iterator and the *insert()* member function:

```
vector<int> Y;
for (int m = 0; m < 100; ++m) {

 Y.insert(Y.begin(), m);

 cout << setw(3) << m
      << setw(5) << Y.capacity()
      << endl;
}
```

```
Index   Cap
0       1
1       2
2       4
3       4
4       8
    . . .
8       16
    . . .
15      16
16      32
    . . .
31      32
33      64
63      64
    . . .
64      128
```

- This is the worst case; insertion is always at the beginning of the sequence and that maximizes the amount of shifting.
- There are overloadings of *insert()* for inserting an arbitrary number of copies of a data value and for inserting a sequence from another vector object.

Deletion from *vector* Objects

- As with insertion, deletion requires shifting (except for the special case of the last element).
  - Member for deletion of last element: *V.pop_back()*
  - Member for deletion of specific element, given an iterator *It*: *V.erase(It)*
- Deletion invalidates iterators that target elements following the point of deletion, so

```
j = V.begin();
while (j != V.end())
 V.erase(j++);
```

  doesn't work
- Member for deletion of a range of values:

  V.erase(Iter1, Iter2)

Container Comparison

- Two containers of the same type are equal if:
  - they have same size
  - elements in corresponding positions are equal
- The element type in the container must have equality operator
- For other comparisons (lexicographical) element type must have appropriate operator (<, >, . . .)

STL *deque* Container

- *deque*
  - double-ended queue
- Provides efficient insert/delete from either end
- Also allows insert/delete at other locations via iterators
- Adds *push_front()* and *pop_front()* methods to those provided for vector
- Otherwise, most methods and constructors the same as for vector
- Requires header file <*deque*>

STL *list* Container

- Essentially a doubly linked list
- Not random access, but constant time insert and delete at current iterator position
- Some differences in methods from *vector* and *deque* (e.g., no *operator[]*)
- Insertions and deletions do not invalidate iterators

Associative Containers

- A standard array is indexed by values of a numeric type:
  - *A[0],...,A[Size-1]*
  - dense indexing
- An associative array would be indexed by any type:
  - *A["alfred"], A["judy"]*
  - sparse indexing
- Associative data structures support direct lookup ("indexing") via complex key values
- The STL provides templates for a number of associative structures

Ordered Associative Containers

- The values (objects) stored in the container are maintained in sorted order with respect to a key type (e.g., an ID field in an Employee object)

set<Key>	collection of unique *Key* values
multiset<Key>	possibly duplicate *Key*s
map<Key,T>	collection of *T* values indexed by unique *Key* values
multimap<Key,T>	possibly duplicate *Key*s

Unordered Associative Containers

- The values (objects) stored in the container do not require an ordering
- However, they require a hash function

unordered_set<Key, Hash>	collection of unique *Key* values
unordered_multiset<Key, Hash>	possibly duplicate *Key*s
unordered_map<Key,T, Hash>	collection of *T* values indexed by unique *Key* values
unordered_multimap<Key,T, Hash>	possibly duplicate *Key*s

Sets and Multisets

- Both set and multiset templates store key values, which must have a defined ordering.
  - set only allows distinct objects (by order) whereas multiset allows duplicate

```
set<int> iSet;             // fine, built-in type has < operator
set<Employee> Payroll;     // class Employee did not
                           // implement a < operator
```

  - the key type has to implement operator <

```
bool Employee::operator<(const Employee& Other) const {
   return (ID < Other.ID);
}
```

set Example

```
#include <functional>
#include <set>
using namespace std;
#include "employee.h"

void EmpsetPrint(const set<Employee> S, ostream& Out);


int main() {
  Employee Ben("Ben", "Keller", "000-00-0000");
  Employee Bill("Bill", "McQuain", "111-11-1111");
  Employee Dwight("Dwight", "Barnette", "888-88-8888");
  set<Employee> S;
  S.insert(Bill);
  S.insert(Dwight);
  S.insert(Ben);
  EmpsetPrint(S, cout);
}
void EmpsetPrint(const set<Employee> S, ostream& Out) {
  set<Employee>::const_iterator It;
  for (It = S.begin(); It != S.end(); ++It)
     Out<<*It<<endl;
}
```

```
000-00-0000 Ben Keller
111-11-1111 Bill McQuain
888-88-8888 Dwight Barnette
```

Choosing a Container

- A *vector* may used in place of a dynamically allocated array
- A *list* allows dynamically changing size for linear access
- A *set* may be used when there is a need to keep data sorted and random access is unimportant
- A *map* should be used when data needs to be indexed by a unique non-integral key
- Use *multiset* or *multimap* when a set or map would be appropriate except that key values are not unique

Imagine this short program...

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

int
main ()
{
  vector < int >v;
  vector < int >::iterator idx;
  int i, total;
  cout << "Enter numbers, end with ^D" << endl;
  cout << "% ";
  while (cin >> i)
    {
      v.push_back (i);
      cout << "% ";
    }
  cout << endl << endl;
  cout << "Numbers entered = " << v.size () << endl;
  for (idx = v.begin (); idx != v.end (); ++idx)
    cout << *idx << endl;
  total = 0;
  for (idx = v.begin (); idx != v.end (); ++idx)
    total = total + *idx;
  cout << "Sum = " << total << endl;
};
```

Common code repeated to process container

Improved...

```
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

void print (int i) {
  cout << i << endl;
};
int main ()
{
  vector < int >v;
  vector < int >::iterator idx;
  int i, total;
  cout << "Enter numbers, end with ^D" << endl;
  cout << "% ";
  while (cin >> i)
    {
      v.push_back (i);
      cout << "% ";
    }
  cout << endl << endl;
  cout << "Numbers entered = " << v.size () << endl;
  for_each (v.begin (), v.end (), print);
  total = accumulate (v.begin (), v.end (), 0);
  cout << "Sum = " << total << endl;
}
```

Using the STL

Generic Algorithms

- Common algorithms that work on the container classes
  - Implement sort, search and other basic operations
- Three types of algorithms that work on sequence containers discussed here:
  - Mutating-Sequence Algorithms
    - *fill(), fill_n(), partition(), shuffle(), remove_if(), sort() ...*
  - Non-Mutating-Sequence Algorithms
    - *count(), count_if(), find(), for_each(),*
  - Numerical algorithms (from <numeric>)
    - *accumulate(), reduce(), inner_product(), inclusive_scan(), ...*

Mutating Functions

- Functions that modify a container in different ways
- Access to the container is done through an iterator
  - Assume

    vector<char> charV;

`void fill(iterator, iterator, T)`	`charV.fill(charV.begin(), charV.end(), 'x')` puts 'x' in all positions of the vector
`iterator fill_n(iterator, int, T)`	`charV.fill_n(charV.begin(), 5, 'a')` puts 'a' in first 5 positions

`void generate(iterator, iterator, function)`	`char nextLetter() {`
	`  static char letter = 'A';`
	`  return letter++;`
	`}`
	`charV.generate(charV.begin(), charV.end(), nextLetter);`
	fills the array with the result of calling *nextLetter* for each element

Non-mutating (Mathematical Algorithms)

- Assume

  vector<int> v;

`T min_element(iterator, iterator)`	`min_element(v.begin(), v.end())` returns the minimum element from the container

`function for_each (iterator, iterator, function)`	`void put(int val)`
	`{ cout << val << endl; }`
	`for_each(v.begin(), v.end(), put);`
	executes the function *put()* for each element in the array; in this case prints all values

`int count(iterator, iterator, T)`	`v.count(v.begin(), v.end(), 5)` returns how many times 5 appears in the container

`int count_if(iterator, iterator, function)`	`bool GT10(int val)`
	`{ return val > 10; }`
	`v.count_if(v.begin(), v.end(), GT10);`
	returns a count of the elements that are greater than 10 in the container

Other Useful Ones

- Assume

  vector<int> v;

`iterator find(iterator, iterator, T)`	`iterator r =find(v.begin(), v.end(), 25);`
	`if (r == v.end())`
	`  cout << "Not found" << endl;`
	`else`
	`  cout << "Found at " << (r - v.begin());`

`iterator find(iterator, iterator, function)`	As the find above, but uses a function for testing
`bool binary_search (iterator, iterator, T)`	Binary search over the container to find value

`iterator copy(iterator, iterator, iterator)`	Copy from a container to another container. Useful when combined with *ostream_iterator*
	`ostream_iterator<int> output(cout, " ");`
	`copy(v.begin(), v.end(), output);`

Much More

- STL has many more operations, several other containers, and other functionality
- Style of programming using STL is called generic programming
  - Write functions that depend on some operations that are defined on the types you will process
  - For example, the *find()* operation relies on the *operator==* to be available on the data type
- For a particular function, we talk about the "set of types" that can be used with the function
  - e.g. in the *find()*, the set is all those types for which *operator==* is defined
- Note the relationship to OOP… not much. The set of types that define some operations such that they can be used in a particular generic function do not need to be related via inheritance and thus polymorphism is not used

Pointers in STL

- STL is very flexible, it can store any data type in any of its containers

```
vector< int > v;
vector< int >::iterator vi;
v.push_back( 45 );
for (vi = v.begin(); vi != v.end(); ++vi) {
  int av = *vi;
}

vector< Foo * > v;
vector< Foo * >::iterator vi;
v.push_back( new Foo( value) );
for (vi = v.begin(); vi != v.end(); ++vi) {
  Foo * av = *vi;
}
```

- The collection does not free the memory allocated for objects, to which it stores the pointers
- If you want that behaviour, make a vector of *unique_ptr*s or *shared_ptr*s

Function Objects in STL

- The function object is an object with function call operator *operator()* defined, so that in the example below

```
FunctionObjectType fo;
// ...
fo();
```

  the expression *fo()* is an invocation of *operator()* of object *fo*, and not a call of function *fo*

Instead of

```
void fo(void) {
 // statements
}
```

we write

```
class FunctionObjectType {
public:
 void operator() (void){
   // statements
 }
};
```

- The function objects can be used in STL in all places, where the pointer to a function is acceptable

Function Objects - Why to Use Them?

- The function objects have the following advantages compared to function pointers
  - The function object can have a state. We can have two instances of a function object of the same type in different states. It is not possible with functions
  - The function object is usually more efficient than the function pointer
    - The compiler can perform inlining
  - It can be used as a template argument, e.g. defining a hash function

The Function Object Example

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdlib.h>
#include <time.h>
using namespace std;

bool GTRM(long val)
{
  return val > (RAND_MAX >> 1);
}

int main ()
{
  srandom(time(NULL));
  vector < long > v(10);
  generate(v.begin(),v.end(),
    random);
  cout << count_if(v.begin(),
    v.end(),GTRM);
  cout <<endl;
};
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdlib.h>
#include <time.h>
using namespace std;

template <class T> class greater_than
{
  T reference;
public:
  greater_than (const T & v): reference (v)
  {}
  bool operator()  (const T & w) {
    return w > reference;
  }
};
int main ()
{
  srandom (time (NULL));
  vector < long >v (10);
  generate (v.begin (), v.end (), random);
  cout << count_if (v.begin (), v.end (),
       greater_than<long> (RAND_MAX >> 1));
  cout << endl;
};
```

The *unordered_set* Example

```
struct Employee {
  std::string FirstName, LastName, ID;
  Employee (const std::string & fn, const std::string & ln,
            const std::string & I):FirstName (fn), LastName (ln), ID (I) {};
  bool operator==(const Employee& o) const  {
  return (FirstName == o.FirstName) && (LastName == o.LastName)
          && (ID == o.ID); }
};

struct EmpHash {
  std::size_t operator()(const Employee & o) const {
    return std::hash<std::string>()(o.FirstName)
         ^ (std::hash<std::string>()(o.LastName) << 1)
         ^ (std::hash<std::string>()(o.ID) << 2);}
};

int main () {
  Employee Ben ("Ben", "Keller", "000-00-0000");
  Employee Bill ("Bill", "McQuain", "111-11-1111");
  unordered_set<Employee, EmpHash> S;
  S.insert (Bill);
  S.insert (Ben);
}
```

Anonymous functions (*lambda expressions*)

- When we are using function pointers or functions objects, their definition are far away from the point od application. It makes understanding what the code is doing more difficult.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdlib.h>
#include <time.h>
using namespace std;

int main ()
{
  srandom (time (NULL));
  vector < long >v (10);
  generate (v.begin (), v.end (), random);
  cout << count_if (v.begin (), v.end (),
                  [](long i) -> bool { return i > RAND_MAX >> 1;  } ) << endl;

};
```

Anonymous functions (*lambda expressions*)

- The return type specification can be omitted in this case, as the compiler can determine it automatically.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stdlib.h>
#include <time.h>
using namespace std;

int main ()
{
  srandom (time (NULL));
  vector < long >v (10);
  generate (v.begin (), v.end (), random);
  cout << count_if (v.begin (), v.end (),
                    [](long i) { return i > RAND_MAX >> 1;  } ) << endl;

};
```

Anonymous functions (*lambda expressions*)

- An anonymous function can be stored in a variable of type *std::function*. An anonymous function can be more complex and contain variable definitions:

```
int main ()
{
  function<int(int,int)> f =
                          [](int x, int y) -> int {int z = x + y; return z + x;};
  cout << f(3,4) << endl;
};
```

- If we do not want to write complex declarations, we can use the auto keyword. The return type specification can be also skipped in this case.

```
auto f = [](int x, int y) {int z = x + y; return z + x;};
```

Closure

- An object binding the function and its enviroment. The closure specification is required, when the function uses the variables defined in enclosing scope.

```
int main ()
{
  vector<int> numbers = {1,2,3,4};
  int sum = 0;
  for_each(numbers.begin(), numbers.end(), [&sum](int x) { sum += x; });
  cout << sum << endl;
};
```

- In the example above, the *sum* variable is captured by reference. As the last argument to *for_each* a function object, storing the reference to *sum*, is passed.

Closure

- Capturing sum by value will not work in this case.

```
for_each(numbers.begin(), numbers.end(), [sum](int x) { sum += x; });
```

- It can be used however to return an anonymous function from another function:

```
auto fun()
{
  int sum=12;
  return [sum](int x) { return sum + x;};
}

int main ()
{
  cout << fun()(4) << endl;
};
```

- Here, in turn, capturing by reference will not work.

Capture specification

[]	Capture nothing
[&]	Capture any referenced variable by reference
[=]	Capture any referenced variable by value
[=,&foo]	Capture any referenced variable by value, but capture variable foo by reference
[bar]	Capture bar by value; don't capture anything else
[this]	Capture the this pointer of the enclosing class

```
class C {
  int c;
public:
  C(int _c): c(_c) {};
  auto fun() {
    return [this](int x) { return c + x;};
  }
  void print(function<int(int)>f) {
      cout << fun()(3) << endl;
  }
};
```

```
int main () {
  C c1(1);
  C c2(2);
  auto f = c2.fun();
  c1.print(f);
};
```