The Standard Template Library

- The Standard Template Library (STL) is a collection of commonly used containers and algorithms.
- They are implemented with templates so they can be used with a variety of data types.
- Two main categories: **containers** and **algorithms**
- **Containers:** template classes that deal with storing and accessing collection of objects in various ways. (e.g. `vector`)
- **Algorithms:** template functions which perform common tasks (e.g. sorting a collection of objects)
- We will talk about the most common containers and algorithms.

Containers

- Containers allow the programmer to store collection of objects.
- There are many different containers with different characteristics.
- We have seen `vector`: similar to an array, but can grow and shrink dynamically.
- Another class `deque` is almost the same as `vector`: the only difference is that there is `push_front()` and `pop_front()` as well.
- `deque` stands for "double-ended queue".
- `vector` and `deque` are "random-access" containers: you can access any element in the container just as easily (by `[]`).
- Other containers may be "forward containers" or "reversible containers". i.e. the elements are accessed from beginning to end (or from end to beginning).

Containers

There are two broad types of containers

Sequences: the elements are arranged in a "linear" order. You can insert and remove elements at specified positions. (e.g. `vector`, `deque`, `list`)

Associative Containers: allows access of elements indexed by **keys**. e.g. keys may be student ID, and the associated information may be grade. You can insert and remove elements, but not at a specific location. (e.g. `set`, `multiset`, `map`, `multimap`)

Container Adaptors

- **Container Adaptors** are special containers.
- They are special in that the elements can only be accessed in a certain way.
- `stack`: can only insert or remove elements at the “top”.
- `queue`: can only insert at one end and remove at the other end.
- `priority_queue`: can insert an element, and only remove the “largest” element.

Iterators

- **Iterators** provide a uniform way of accessing elements in different types of containers (except container adapters).
- They work like pointers: they point to elements in the container.
- You can dereference iterators (`*`, `->`), increment (`++`), decrement (`--`) and do iterator arithmetic (like pointer arithmetic)
- These operations are done with the same operators you used for pointers (through operator overloading).
- Many STL algorithms use iterators to specify the range in a container to operate on. You can use pointers in these too!

Iterators

- The basic way to declare an iterator for a container is:

```
container::iterator i;
```

- For example, `vector<int>::iterator i;`
- An iterator for one type of container is different from an iterator for another type of container (e.g. `int *` is different from `char *`).
- Every container (other than container adapters) has `begin()` and `end()` member functions.
- `begin()`: returns an iterator that points to the first element in the container. (e.g. in an array `A`, `A` is a pointer to the first element.)
- `end()`: returns an iterator that points to **one element past the end** of the container. (e.g. in an array `A` of size `n`, `A+n` points to one-past-the-end.)

Iterators

- A typical loop to run through the elements in a container:

```
list<double> L;
// code to put elements into L
list<double>::iterator it;
for (it = L.begin(); it != L.end(); ++it)
  cout << *it << endl;
```

- We use `++it` instead of `it++` for efficiency.
- We can also use the loop above with pointers if `begin()` and `end()` are defined appropriately.
- Order of elements for sequences: based on the order we used to build the sequence.
- Order of associative containers: sorted based on keys (from smallest to largest).

Range-based for loops

- New in C++11:

```
list<double> L;
// code to put elements into L
for (double d : L) {
  cout << d << endl;
}
```

- This is equivalent to the iterator version above.
- Use `auto` to automatically deduce type (useful in template functions).
- Use reference if you want to modify the elements.
- Use constant reference if you do not want copying (and do not want to change elements).

Iterators

There are a number of different types of iterators.

- Iterators to constants (`const_iterator`): analogous to pointers to constants—you cannot change what they point to.
- Reverse iterators (`reverse_iterator`): move in reverse order. Use `rbegin()` and `rend()` (why can't you use normal iterators and `--it`?)
- Reverse iterators to constants (`const_reverse_iterator`).

Iterators

Another way to look at iterators:

Forward: supports increments (all iterators we study)

Bidirectional: supports increments and decrements (most iterators we study)

Random Access: supports increments, decrements, and iterator arithmetic (i.e. just like pointers). Only supported by `vector` and `deque`.

Defining Intervals with Iterators

- Many member functions of containers operate on a section (interval) of the container.
- An iterval is usually specified by an iterator pointing to the beginning, and an iterator pointing to one-past-the-end. We usually denote the interval as `[begin, end)`.
- For arrays, we can use `A+i`, `A+j` to refer to the interval `A[i..j-1]`.
- To specify the whole container, use `begin()` and `end()` for the container.

Operations with Iterators

If `C` is a sequence and `p`, `i`, and `j` are iterators to the appropriate data types:

- `seq<type> C(i, j)`: constructs a sequence `C` and initialize it with the elements in `[i,j)`. Note that `i` and `j` are iterators to a different container (can be a different type, but element type is the same). They can even be pointers to array elements.

  e.g. `vector<int> v(A, A+5);` If `A` is an integer array, this initializes `v` to the first 5 elements of `A`.
- `C.assign(i, j)`: similar to above, except it is an assignment.
- `C.insert(p, e)`: inserts the value `e` into the position `p`. `p` must be an iterator for `C`.
- `C.insert(p, n, e)`: inserts `n` copies of `e` into the position `p`.

Operations with Iterators

- `C.insert(p, i, j)`: inserts the elements in `[i,j)` into the position `p`.
- `C.erase(p)`: erases the element at position `p`.
- `C.erase(i, j)`: erases the elements in the interval `[i,j)`. `i` and `j` must be iterators for `C`.

Algorithms

- There are a number of commonly used algorithms in STL.
- Need to `#include <algorithm>`.
- Many algorithms work on containers and use iterators to specify intervals.
- That means they work on arrays and pointers too.

Insert Iterators

- We often want to insert elements to the end of a container.
- But `end()` returns one-past-the-end, and does not point to a valid location.
- Use `back_inserter()`. e.g.

```
copy(C1.begin(), C1.end(), back_inserter(C2));
```

  This inserts all elements of `C1` to the end of `C2`.

  (i.e. uses `push_back()` on each element copied.)
- `front_inserter()` works in a similar way.
- To insert in the middle (at position pointed to by iterator `it`), use `inserter(C, it)` where `C` is the container.

Common Algorithms

- `copy(p, q, r)`: copies the range `[p, q)` into the location referred to by `r`.
- `transform(p, q, r, f)`: transforms the element `x` in the range `[p,q)` to `f(x)` and stores the result in to `r`. (`f` is a unary function, `r` can be the same as `p`).
- `fill(p, q, val)`: sets the elements in `[p,q)` to `val`. e.g. `fill(v.begin(), v.end(), 10);` sets all elements in container to 10.
- `find(p, q, val)`: returns an iterator to an element in the range `[p,q)` whose value is `val`. Returns `q` if not found. e.g.

```
if (find(v.begin(), v.end(), 10) != v.end()) {
  cout << "found" << endl;
}
```

Common Algorithms

- `sort(p, q)`: sorts the elements in `[p,q)` from smallest to largest (`operator<` defined for elements).
- `min_element(p, q)`: returns an iterator pointing to the smallest element in `[p,q)`. `max_element(p,q)` is similar.
- `binary_search(p, q, val)`: returns true if and only if the sorted sequence `[p,q)` contains `val`. If you actually want to find the locations, use `equal_range()`.

Algorithms

The STL has many more algorithms. See various web sites if you want to find out more.

Function Parameters

- Many STL algorithms take an optional parameter to fine-tune its behavior.
- e.g. `transform` uses a unary function to specify the desired transformation.
- e.g. `sort`: what if you want to sort from largest to smallest, or in some other order?
- There are two ways to pass in the function parameter into an algorithm: pointers to functions or function objects.

Pointers to Functions

```
int f(int x) { return x*x; }

int A[5] = {1, 2, 3, 4, 5};
transform(A, A+5, A, f);
```

`f` is treated as a pointer to the function `f`.

Pointers to Functions

```
bool less_than(const string &s1, const string &s2)
{
  if (s1.length() != s2.length())
    return s1.length() > s2.length();
  else
    return s1 < s2;
}

string A[5];
...
sort(A, A+5, less_than);
```

Sorts `A` from longest string to shortest string, break ties lexicographically.

Anonymous (Lambda) Functions

- In C++ you can define functions with no names. They can be used as parameters to pass into other functions.

```
transform(A, A+5, A, [](int x) { return x*x; });
```

- The start of the function is `[]`, followed by parameter list.
- Body of function is enclosed in braces.
- Usually no need to specify return types (deduced automatically).

Anonymous (Lambda) Functions

- return types can be specified explicitly:

```
transform(A, A+5, A, [](int x) -> int { return x*x; });
```

- You can assign an anonymous function to a variable if you wish:

```
auto square = [](int x) { return x*x; };
```

  You must use `auto` to get the type.

- Advanced: `[]` needs not be empty. It captures content of other variables to be used inside the function.

Sequences: vector and deque

- Can access any element easily.
- Inserting/deleting in the middle of sequence may be expensive.
- Difference: with vector it is easy to add to the back, with deque it is also easy to add to the front.

Sequence: list

- Can easily access first and last elements (`begin()` and `rbegin()`).
- All other elements: must use iterators and step through with `++` and `--`. i.e. no indexing with `[]`
- Inserting/removing element at any point: very fast.
- Some algorithms need random access iterators. e.g. `sort`.
- But `list` provides its own sort function.
- Example: a text editor stores the text as a list of characters.

Associative Containers

- Tables whose entries are identified by **keys** rather than positions. e.g. name, student ID.
- The data type of the keys must be **comparable**: `operator<` must be defined (default), or you can supply your own comparison function.
- The entries are sorted: you can iterate through the entries from smallest key value to largest key value (or vice versa).
- You cannot insert elements at a particular position.
- Provides bidirectional iterators, but not random access.
- Most standard algorithms can be applied through iterators.
- Accessing entries are relatively efficient. We will talk about how the data is stored later on.

Associative Container: map

- A `map` is a table of **key-value** pair. For example, a name-telephone number pair.
- There is at most one entry associated to each key.
- Entries are accessed by the key. e.g. we can access a phone number by name.
- To declare a map, you need to specify the data types for the key and the value:

```
#include <map>

map<string, int> marks;   // store student marks by name
```

- Items are stored as `pair<key_type,value_type>`.

Associative Container: map

- The easiest way to access entries is through the [] operator:

```
marks["John Doe"] = 75;
```

  This adds the entry with key = "John Doe" and value = 75. If an entry with the same key already exists, it is replaced.

- If you write m[k] where m is a map and there is no entry with key k, an entry is created whose value is the default value (default constructor for value type is called).

- You can use iterators and begin() and end() to iterate through a map. An iterator points to a pair<key_type,value_type>.

- If p is such a pair, p.first gives the key and p.second gives the value.

- If it is an iterator to a map element, it->first gives the key and it->second gives the value.

Associative Container: map

Some operations require parameters of `pair`. Use `make_pair(key, value)` to make a pair.

Common functions:

- `insert(p)`: inserts the pair `p` into the map. Returns a pair `<it,b>` such that `it` points to the inserted pair if `b` is true, or `b` is false if an entry with the same key already exists.
- `find(k)`: returns an iterator that points to the key-value pair in the map whose key is `k`. If such a pair does not exist, returns `end()`.
- `count(k)`: returns the number of pairs with the given key.
- `erase(k)`: erases all entries with the given key.
- `clear()`: empties the map.

Associative Containers: set and multiset

- These are similar to the mathematical notion of set and multiset.
- Similar to `map` and `multimap`, but entries are keys only (no value).
- Must specify key type:

```
#include <set>
set<string> names;
```

- The supported functions are similar to `map/multimap`, except that the parameters are keys instead of pairs. See p. 457–458.
- There are also `set_union`, `set_intersection`, `set_difference`, `set_symmetric_difference`, and `includes` (i.e. subset). They have the usual meanings from mathematics.

Examples

```
set_union(s1.begin(), s1.end(), s2.begin(), s2.end(),
          inserter(s3, s3.begin()));
```

inserts the union of `s1` and `s2` into `s3`.

```
if (s1.count("John Doe") > 0)
  cout << "member" << endl;
else
  cout << "not member" << endl;
```

Stacks

- A `stack` is a container in which you can `push` elements into the top, and `pop` elements from the top.
- "Last in first out"
- Include `<stack>`
- The operation `top()` returns the element at the top. Use `pop()` to remove it.
- It is an error to use `top()` or `pop()` if the stack is empty. Use `empty()` or `size()` to check first.
- Efficient.

Queues

- A `queue` is a container in which you can `push` elements into the back, and `pop` elements from the front.
- "First in first out"
- Include `<queue>`
- The operation `front()` and `back()` gives the element in the front and back of the queue.
- It is an error to pop from an empty queue.
- Efficient.

Priority Queues

- A `priority_queue` is a queue where elements are ordered based on "priority". A comparison function must be defined for the elements (`<` is default).
- Include `<queue>`
- You can `push()` and `pop()` elements.
- The element at the top is the largest element (defined by the comparison).
- If there are multiple largest element, the top may be any one.
- Relatively efficient.