The C++ Standard Template Library

Marshall Clow

Qualcomm

7 March 2017

Introduction

About the author

Marshall is a member of the C++ standards committee, and maintains the LLVM C++ standard library (`libc++`).

He is a long time member of the Boost open source project, and is the author of the `Boost.Algorithm` library. He also maintains `Boost.Array` and `Boost.StringAlgo` as well.

Marshall has worked at Qualcomm since 2002, and has worked in several groups including the Eudora email client, QPSI and QOSP.

Coding Conventions

In this course, I'll show you a lot of code snippets. In general, they will be incomplete, due to the limitations of fitting code on slides.
Instead of:

```
#include <iostream>

int main () {
    std::cout << "Hello World" << std::endl;
    return 0;
    }
```

I will show you:

```
    cout << "Hello World" << endl;
```

You should assume that all the necessary header files have been included, and that "`using namespace std;`" came before the code shown.

Target language

In general, I will be targeting C++11 in this course. I will (usually) call out where things are different between C++03 and C++11. When C++14 or C++1z has a facility for doing something easier, I will try to mention it.

I will not be covering things that were part of C++03, but have been deprecated and/or removed in later versions of the standard. Things like `auto_ptr`, `bind_1st`, `mem_fun` and so on.

What is the STL?

The Standard Template Library (henceforth the STL) is a set of classes and functions that were proposed by Alex Stepanov and Meng Lee as an extension to the C++ standard library about 1994 and included into the C++ standard library in the C++98 standard

The STL embodies the concept of "generic programming", the idea that you can write a single piece of code that works for a wide variety of different types of data. In C++, the mechanism for this is almost always templates.

Why is this important?

1. By using a set of standard components, you can concentrate on doing what you need to do, rather than spending your time building infrastructure.
2. You can program "at a higher abstraction level", with better vocabulary terms.

Example: What do these two bits of code do?

```
vector<int> v{0,1,3,5,7,9,2,4,6,8};
bool flag = true;
for (int i = 1; (i <= v.size()) && flag; i++) {
    flag = false;
    for (int j = 0; j < (v.size() -1); j++) {
        if (v[j+1] < v[j]) {
            swap(v[j], v[j+1]);
            flag = true;
        }
    }
}
for (int i:v) cout << i << ' ';
```

```
vector<int> v{0,1,3,5,7,9,2,4,6,8};
sort(v.begin(), v.end());
for (int i:v) cout << i << ' ';
```

Why use the STL?

1. **Efficiency** - The STL is specified with complexity guarantees, and contains very little inheritance or virtual functions.
2. **Type Safety** - C++ is a strongly-typed language, and the STL uses the facilities of the language to ensure type safety.
3. **Consistency** - Wherever possible, the STL is consistent. The concept of `iterators` abstracts the idea of traversing a collection of values, meaning that a single implementation of an algorithm can work with many different collections.
4. **Extensibility** - You can define your own containers, iterators, or algorithms to work with the STL.

The Path to Standardization

1. 1979: Alex Stepanov began working on generic programming at GE Research & Development with David Musser and Deepak Kapur.
2. 1984: Stepanov and Musser continue work on generic programming at Bell Labs and then HP Labs. This work was done mostly in Ada, but about this time they started using C++ as well.
3. 1992: Ming Lee joined Alex's team at HP Labs.
4. 1993: Andrew Koenig of Bell Labs became aware of the work and asked Stepanov to present the main ideas at a November 1993 meeting of the ANSI/ISO committee for C++ standardization
5. 1994: A reworked proposal was accepted by the ISO committee for standardization.
6. 1998: The C++ standard that included the STL was published (C++98)

C++98/03

First Release of the C++ Standard

1. Contained the STL.
2. Containers: `vector`, `deque`, `list`, `map` and `set`.
3. Algorithms `sort`, `find`, `copy`, `partition` and so on.
4. Utilities `pair`, binders and so on.

C++11

New language features

1. auto
2. range-based for loop
3. lambdas
4. theading
5. variadic templates
6. rvalue references
7. constexpr
8. initializer_lists

C++11

New library features

1. tuple
2. random
3. chrono
4. regex
5. unordered containers: `unordered_set`, `unordered_map`, etc.
6. new containers: `array`, `forward_list`
7. threading (`async`, `future`, etc)

C++14

1. Generic lambdas
2. return type deduction
3. More constexpr
4. compile-time integer sequences

C++1z

Will almost certainly be C++17

1. variant/optional/any/string_view
2. Parallel algorithms
3. constexpr if
4. structured bindings
5. Special math functions
6. constexpr lambdas

Farther Out

1. Networking
2. Concepts
3. Coroutines
4. Ranges

Compiler Support

1. C++98/03 - Support for C++03 took a long time to appear. gcc had most of it by gcc 4.2 (about 2007); maybe sooner.
2. C++11 came much faster.
    1. `clang/libc++` had full C++11 support around 2012.
    2. `gcc/libstdc++` had full support with the release of 4.9 (April 2014)
3. C++14 came faster still.
    1. `clang/libc++` had full C++14 support for the 3.5 release (September 2014).
    2. `gcc/libstdc++` had almost everything in the 5.1 release (April 2015) and full support in the 6.1 release (April 2016).
4. C++1z (which will almost certainly be C++17) is almost here.
    1. `clang/libc++` will have most of it late next year.
    2. `gcc/libstdc++` will have a lot (if not all) of it in early 2018.

Language Concepts

Object Lifetimes

Object lifetime is one of the most important concepts in C++.
An object's lifetime begins when its constructor completes successfully.
An object's lifetime ends when its destructor is entered.

1. A constructor turns raw memory into an object.
2. A destructor turns an object into raw memory.

Templates

A template is a way of defining a “pattern” for a class or a function.
Templates have parameters, and then when you provide the parameters, the template is “instantiated”.
A template may be instantiated many times - each time with different parameters.
You can have "specializations" of templates, which give different results for different types.

Template functions

```
template <typename T>
bool positive(T t) { return t > 0; }

void pos () {
    positive(2);            // calls positive<int>
    positive(2.5);          // calls positive<double>
    positive(string{"2.5"}); // fails to compile
   }
```

Template class

```
template <typename T>
struct wrapper {
    wrapper() : val_() {}
    template <typename T1>
        wrapper(const T1 &t1) : val_(t1) {}
    T & get() { return val_; }
    T const & get() const { return val_; }
private:
    T val_;
    };

wrapper<string> ws{"123"};
```

Variadic Templates (C++11)

You can define a template to have an unknown number of template parameters.
This is similar in principle to the "variadic arguments" to functions in C (like `printf`).

Variadic templates

```
template <typename... T>
struct wrapper2 {/*..*/};

template <typename... T>
bool func(T... args) { return true; }

void proc() {
    wrapper2<int, double> f; // a variable
 // calls func<int, string>
    func(1, string{"foo"});
}
```

Exceptions

Exceptions are part of lifetime management; it is the only way to signal that a constructor has failed.
The standard library throws exceptions to report errors.

Move Semantics and Rvalue references (C++11)

1. The idea behind move semantics is that it is wasteful to copy an object which is about to be destroyed.
2. Instead, you can "suck the guts out" of the source object into the new object, and leave behind the empty husk to be destroyed.
3. Move semantics is an optimization.
4. For some types, "moving" can be as expensive as copying (or more!). `int` for example.

Move Semantics

```
vector<int> make_v() {
    return vector<int>{0,1,2,3,4,5,6,7};
}

void foo() {
    vector<int> v1 = make_v();
//  do stuff with v1....
}
```

Rvalue special member functions

`Mve &&` is the type of "an rvalue of type `Mve`".

```
class Mve {
  Mve() {}                              // default ctor
  Mve(const Mve &);                     // copy ctor
  Mve(Mve &&);                          // move ctor
  Mve & operator=(const Mve &);         // copy assignment
  Mve & operator=(Mve &&);              // move assignment
};
```

Constexpr (C++11)

Calculations can be done at compile-time.

```
constexpr int func(int x) { return x + 2; }

int main ()
{
    int arr[func(4)];
    static_assert(sizeof(arr) == 6*sizeof(int), "");
}
```

Containers

What is a container?

1. A container is a data structure that holds a collection of objects, all of the same type.
2. In general, a container can hold elements of any type, as long as they meet certain functional requirements.
3. A container provides ways of accessing the contained objects (called the elements), modifying them, enumerating them (using iterators), and other operations.
4. A container "owns" the elements in the container, and manages their lifetimes. When a container is destroyed, all the elements in the container are destroyed.

Example of Container Usage

```
vector<int> v{0,1,3,5,7,9,2,4,6,8};
assert(v.size() == 10); // member function
assert(!v.empty());     // another member fn
v.push_back(12);        // append to the sequence
assert(v.size() == 11); // now bigger!
assert(v[5] == 9);      // read an element
v[8] = 23;              // assign to an element
for (int i:v)
    cout << i << ' ';
cout << endl;
//  prints: 0 1 3 5 7 9 2 4 23 8 12
```

Types of Containers in the STL

1. Sequential containers
2. Associative containers
3. Unordered associative containers (C++11)
4. Container adapters
5. Not quite containers

But first - Iterators

What is an iterator

1. A generalization of the concept of a pointer
2. Manages the traversal of a sequence and accessing the members of sequence
3. A single iterator is a position
4. Two iterators denote a sequence $[begin, end)$

Operations on iterators

1. You can dereference it (access the data that it "points to")
2. You can copy it (most iterators)
3. You can compare it to another iterator
4. You can advance it "to the next position"
5. You can advance it "to the previous position" (most iterators)

Traversing a list

```
template <typename T>
struct list_element {
    list_element() : value() {}
    list_element(const T& t) : value(t) {}

    T value;
    list_element *next;
};

// NULL marks end of list
template <typename T>
void for_each_list(list_element<T> *list)
{
    while (list != nullptr)
    /* do something with list->value */
        list = list->next;
}
```

Traversing an array

```
template <typename T>
void for_each_array(T* arr, size_t sz)
{
    for (size_t i = 0; i < sz; ++i)
    /* do something with *arr */
        ++arr;
}
```

Either one with iterators

```
template <typename Iterator>
void for_each(Iterator first, Iterator last)
{
    for (; first != last; ++first)
    /* do something with *first */ ;
}

    vector<int> v;
    for_each(v.begin(), v.end());
    list<int> l;
    for_each(l.begin(), l.end());
    int arr[1];
    for_each(arr, arr+1);
```

Two iterators denote a "half-open" range.

A half-open range means up to, but not including the value pointed to by the end iterator.
Two equal iterators denote an empty range.
The end iterator doesn't "point at" anything; don't dereference it.

Categories of iterators

There are various categories of iterators defined in the STL.
They differ in the operations that you can perform on an iterator.

1. Input iterators
2. Forward iterators
3. Bidirectional iterators
4. Random-access iterators
5. Output iterators

Input iterators

Input iterators are the least capable category of iterators.
A pair of input iterators defines a sequence that may be traversed only once, and only in a single direction.
Advancing to the next element in the sequence is done with the increment operator: `iter++`.
Input iterators may only be advanced a single element at a time.
Examples include:

1. Reading from a file.
2. Reading from a network socket.

Forward iterators

Forward iterators have all the abilities of input iterators, *plus* the ability to traverse a sequence more than once.
This means, for example, that you can make a copy of a forward iterator, and then use the copy to return to a spot in the sequence.
Calculating the "distance" between two forward iterators is an order-N operation.
One place you'll find Forward iterators is when traversing a singly-linked list.
The container `forward_list` provides forward iterators.

Bidirectional iterators

Bidirectional iterators have all the abilities of forward iterators, *plus* the ability to "back up" in a sequence.
Like moving forward, moving backwards is done one element at a time.
Advancing to the previous element in the sequence is done with the decrement operator: `iter-`.
You'll see bidirectional iterators in several places, but the most obvious is traversing a doubly-linked list.
The container `list` provides bidirectional iterators.

Random-Access iterators

Random Access iterators have all the abilities of bidirectional iterators, *plus* the ability to "skip around" in a sequence.
You can advance by N places in constant time, and calculate the distance between two iterators in constant time.
Indexing works as well: `iter[5]`
You'll see random-access iterators in several places, usually with contiguous storage.
For example, the containers `vector` and `array` provide random-access iterators.

Output iterators

In general, you can write through iterators that are not output iterators.
There are "constant iterators", which are read-only.
However, there are a class of iterators that all you can do with them is write to them. These are called “output iterators”.
An example of such an iterator is `std::insert_iterator`, which takes the values written to it, and inserts them into a container.

Containers and iterators

Each container has its own iterator classes.
The capabilities of the iterator depend on the container; `forward_list` provides forward iterators; `vector` provides random-access iterators.
A container usually provides several iterator classes:

1. regular iterators
2. constant iterators
3. reverse iterators
4. constant reverse iterators

Note: a `list<int>::iterator` is a different type than a `vector<int>::iterator`.

Traversing a container

```
vector<int> v{0,1,2,3,4,5,6,7};
typedef vector<int>::iterator It;
typedef vector<int>::const_iterator CIt;
typedef vector<int>::reverse_iterator RIt;
typedef vector<int>::const_reverse_iterator CRIt;

for (It   it = v.begin();   it != v.end();   ++it) {
for (CIt  it = v.cbegin();  it != v.cend();  ++it) {
for (RIt  it = v.rbegin();  it != v.rend();  ++it) {
for (CRIt it = v.crbegin(); it != v.crend(); ++it) {
```

Traversing a container (2)

```
vector<int> v{0,1,2,3,4,5,6,7};

for (auto it = v.begin();   it != v.end();   ++it) {
for (auto it = v.cbegin();  it != v.cend();  ++it) {
for (auto it = v.rbegin();  it != v.rend();  ++it) {
for (auto it = v.crbegin(); it != v.crend(); ++it) {
```

Container typedefs

Each container starts with a collection of typedefs roughly like this:

```
template <typename T, size_t sz>
struct array {
  typedef T                 value_type;
  typedef value_type&       reference;
  typedef const value_type& const_reference;
  typedef value_type*       iterator;
  typedef const value_type* const_iterator;
  typedef value_type*       pointer;
  typedef const value_type* const_pointer;
  typedef size_t            size_type;
  typedef ptrdiff_t         difference_type;
  typedef ...               reverse_iterator;
  typedef ...               const_reverse_iterator;
  // rest of class ...
  };
```

const vs. non-const members

Containers have overloaded `begin()` and `end()` methods.
Some containers have other overloaded methods as well.

```
template <typename T, size_t sz>
struct array {
  // rest of class ...
  iterator       begin();
  const_iterator begin() const;
  iterator       end();
  const_iterator end() const;

  reference       operator[](size_t idx);
  const_reference operator[](size_t idx) const;
  // rest of class ...
  };
```

std::iterator_traits

Sometimes you don't have a container, just an iterator (or a pair of iterators).
The "traits class" `std::iterator_traits` can tell you about the iterators.
`std::iterator_traits<Iter>` is a struct with nested typedefs:

1. difference_type
2. value_type
3. pointer
4. reference
5. iterator_category

The `iterator_category` is one of a set of types, indicating what kind of iterator it is (random-access, bidirectional, forward, input, output)

Containers and allocators

Every container (except `array`) takes a template parameter that is an allocator.
An allocator is an object that manages the allocation and deallocation of memory for the container.

```
template <typename T,
          typename Alloc = std::allocator<T>> struct vector;
```

Sequential Containers

Sequential Containers

1. array (C++11)
2. vector
3. deque
4. list
5. forward_list (C++11)

Sequential containers

1. Store elements in a particular order
2. The order is maintained by the caller
3. If you can insert, you can insert anywhere
4. Searching is linear

Sequential container non-member functions

1. Comparisons
2. `std::swap()`

std::array

std::array (1)

The simplest container

1. Introduced in C++11
2. A replacement for built-in arrays
3. same layout, same codegen as built-in array
4. size is fixed at compile time
5. iterator support (forward, backward, const, non-const)
6. built-in comparisons
7. works with the standard algorithms
8. No implicit conversion to a pointer
9. Member functions `operator[]`, and `at()` for indexed access
10. Member functions `front()`, `back()`, `data()`, `size()` and `swap()`

std::array (2)

The simplest container

```
array<int, 5> ar1{0,1,3,5,7};
assert(ar1.size() == 5);
for (int i:ar1)
    cout << i << ' ';
cout << endl;

int ar2[] = {0,2,4,6,8};
for (int i:ar2)
    cout << i << ' ';
cout << endl;
```

std::array (3)

Performance of std::array

1. Same performance as built-in arrays
2. Constant-time indexing (`operator[]`, and `at()`)
3. Constant time for `front()`, `back()`, `data()`, `size()`.
4. Random-access iterators
5. `swap()` can be expensive because it has to swap all the elements.

std::array (4)

Working with legacy APIs

```
void legacy(int *p, size_t count){/* ... */}

int main ()
{
array<int, 5> ar1{0,1,3,5,7};
int ar2[] = {0,2,4,6,8};

legacy(ar1.data(), ar1.size());
legacy(ar2, 5);
// array implicitly decays to pointer
// have to know '5' (usually #defined)
}
```

array member functions

1. `size()`, `max_size()` and `empty()`
2. `front()`, `back()`, `at()`, `operator[]` and `data()`
3. `fill()`

array non-member functions

1. `std::get<idx>()`

Use std::array instead of built-in arrays.

std::vector

std::vector (1)

Your go-to container

```
template <typename T, typename A = allocator<T>>
class vector;
```

1. like array, but size can be changed at run-time
2. same layout as std::array
3. cheap to append to; expensive to prepend to.
4. You can insert in the middle, too - but that can be expensive.

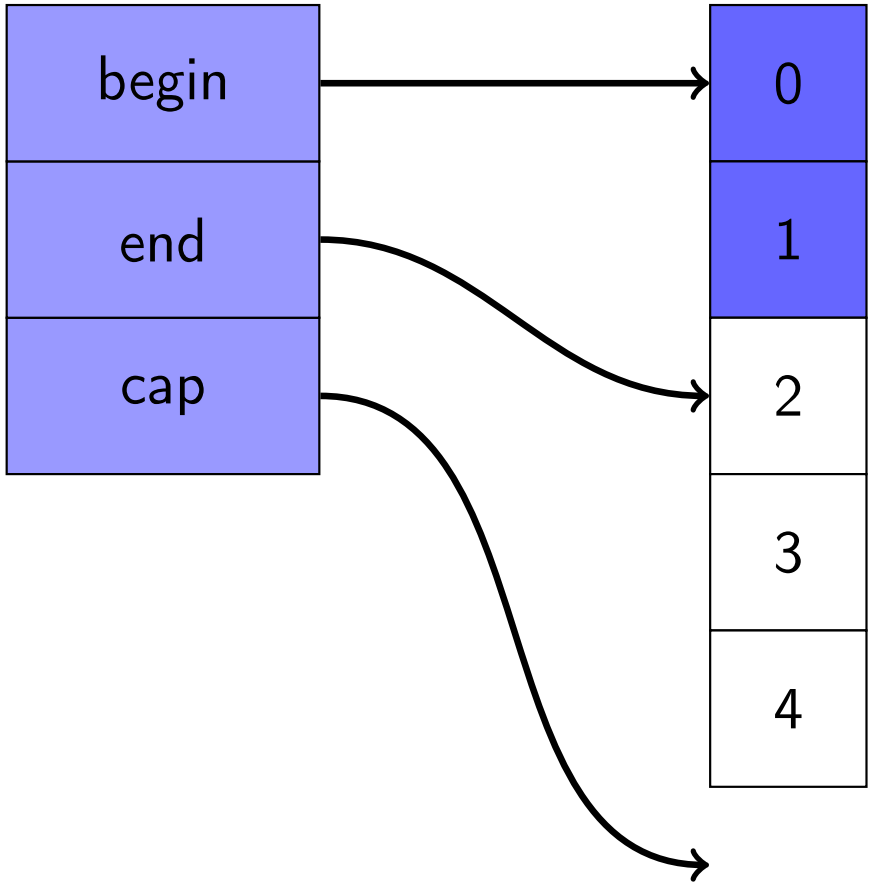
How does std::vector work?

1. The actual `vector` object is fairly small; it contains the allocator and (generally) three pointers.
2. The pointers point to a block of memory (allocated using the allocator) that holds the elements of the vector.
3. The block of memory may have unused storage at the end.
4. A vector has both a `size()` and a `capacity()` - many people mix these up.
    1. `size()` tells you how many elements are stored in the vector.
    2. `capacity()` tells you how many elements there is room for in the vector.

Non-empty vector

`size() == 2`

`capacity() == 5`



vector sizes (1)

```
vector<int> v{0,1,3,5,7,9,2,4,6,8};
assert(v.size() == 10);
assert(v.capacity() >= 10);
v.resize(12);  // add more elements
assert(v.size() == 12);
v.reserve(20); // ensure that space exists
assert(v.size() == 12);
assert(v.capacity() >= 20);
v.push_back(10);
assert(v.size() == 13);
```

vector sizes (2)

1. When you add an element to a vector, and there is no extra storage (`size() == capacity()`), then the vector has to reallocate.
2. A reallocation involves allocating more memory, copying (or moving, if possible) all the elements into the new block of memory, and then destroying the elements in the old block and deallocating it.
3. Vector `push_back()` is "amortized constant time", which means that reallocatons happen less frequently as the vector gets bigger.
4. When an element is added to or removed from a vector, then existing iterators may be "invalidated".

std::vector

Performance of std::vector

1. Constant-time indexing (`operator[]`, and `at()`)
2. Constant time for `front()`, `back()`, `data()`, `size()`.
3. Random-access iterators
4. `swap()` is very cheap because it just swaps pointers.
5. "amortized constant" time for `push_back()`
6. Inserting anywhere except the end can be expensive.
7. if an insertion causes a reallocation, this can be expensive.

vector member functions (1)

1. `empty()` and `size()`
2. `operator[]` and `at()` for indexed access
3. `front()`, `back()`, `data()` and `swap()`
4. `begin()`, `end()`, `rbegin()`, `rend()`
5. `cbegin()`, `cend()`, `crbegin()`, `crend()`
6. `push_back()` and `pop_back()`
7. `clear()` and `resize()`
8. `reserve()` and `shrink_to_fit()`

vector member functions (2)

1. `insert()`, `erase()` and `clear()`
2. `swap()`

variations on insert

```
iterator insert(const_iterator pos, const T& value);
iterator insert(const_iterator pos, T&& value);

iterator insert(const_iterator pos,
                size_type count, const T& value);

template <typename InIter>
  iterator insert(const_iterator pos,
                  InIter first, InIter last);

iterator insert(const_iterator pos,
                std::initializer_list<T> ilist);
```

variations on erase

```
iterator erase (const_iterator position);

iterator erase (const_iterator first,
                const_iterator last);
```

erasing from an array (by hand)

```
vector<int> v{0,1,2,3,4,5,6,7,8,9};
auto it = v.begin();
while (it != v.end())
{
    if (*it % 2 == 0)
        it = v.erase(it);
    else
        ++it;
}
for (int i:v) cout << i << ' ';
//  prints 1 3 5 7 9
```

std::deque

std::deque (1)

```
template <typename T, typename A = allocator<T>>
class deque;
```

1. Random-access, like `array` and `vector`.
2. cheap to append to and prepend to.
3. You can insert in the middle, too - but that can be expensive.

How does std::deque work?

1. A deque has a sequence of fixed-sized blocks, each containing a sequence of elements.
2. All of the blocks except the first and the last are full.
3. Prepending adds an element on the front of the first block (no items need to be moved)
4. Appending adds an element on the back of the last block (no items need to be moved)

deque member functions (1)

1. `size()` and `empty()`
2. `resize()` and `max_size()`
3. `front()`, `back()`, `at()` and `operator[]`

deque member functions (2)

1. `insert()`, `erase()` and `clear()`
2. `push_back()` and `pop_back()`
3. `push_front()` and `pop_front()`
4. `swap()`

std::list

std::list

```
template <typename T, typename A = allocator<T>>
class list;
```

1. List implements a doubly-linked list.
2. List's iterators are bidirectional.
3. cheap to insert and remove anywhere in the container
4. You move chunks of lists between containers with `splice()` or `merge()`

list member functions (1)

1. `size()` and `empty()`
2. `resize()` and `max_size()`
3. `front()` and `back()`

list member functions (2)

1. `insert()`, `erase()` and `clear()`
2. `push_back()` and `pop_back()`
3. `push_front()` and `pop_front()`
4. `swap()`

list operations (3)

1. `merge()` and `splice()`
2. `remove()` and `remove_if()`
3. `reverse()`, `unique()` and `sort()`

std::forward_list

std::forward_list

```
template <typename T, typename A = allocator<T>>
class forward_list;
```

1. Introduced in C++11.
2. forward_list implements a singly-linked list.
3. forward_list's iterators are forward iterators.
4. cheap to insert and remove anywhere in the container; but you can only do so after an element.

forward_list member functions (1)

1. `size()` and `empty()`
2. `resize()` and `max_size()`
3. `front()`

forward_list member functions (2)

1. `insert_after()`, `erase_after()`, `splice_after()` and `clear()`
2. `push_front()` and `pop_front()`
3. `swap()`
4. `merge()` merges two (sorted) lists, producing combined, sorted list.
5. `reverse()`, `unique()` and `sort()`

Associative Containers

Associative containers

1. Order of elements in the container is determined by the container.
2. Sets and Dictionaries (called "maps")
3. Efficient lookup by value (logarithmic)
4. Unique values (`set`) and duplicates (`multiset`)
5. Comes in ordered and unordered varieties (`set` and `unorderd_set`)

Associative Ordering Example

```
vector<int> v;
set<int> s;
unordered_set<int> u;
v.push_back(5); s.insert(5); u.insert(5);
v.push_back(9); s.insert(9); u.insert(9);
v.push_back(1); s.insert(1); u.insert(1);
for (int i:v) cout << i << ' ';
for (int i:s) cout << i << ' ';
for (int i:u) cout << i << ' ';
//   prints 5 9 1   1 5 9   1 9 5
```

Digression: Strict Weak Ordering

There are many places in the standard library that are parameterized on a comparison function, but you can use your own comparison functions if you wish. In general, these default to `std::less`, which implements "less than".
Any replacement function must meet the requirements of a strict weak ordering on the set of values passed to it.

1. A comparison function is irreflexive: `!comp(x, x)` for all `x`.
2. A comparison function is transitive: `comp(a, b)` and `comp(b, c)` implies `comp(a, c)`.

map and set

```
template <typename T,
          typename Compare = less<T>,
          typename Alloc = allocator<T>
> class set;
```

```
template <typename Key,
          typename Value,
          typename Compare = less<Key>,
          typename Alloc
               = allocator<pair<const Key,Value>>
> class map;
```

Another Digression: pair?

std::pair is a small struct designed to hold values of two types.
It has two fields, named first and second.
Think of it like this:

```
template <typename T1, typename T2>
struct pair
{
    T1 first;
    T2 second;
};
```

std::set

1. Unique values - no duplicates
2. Always sorted, using the comparison predicate
3. Efficient lookup by value (logarithmic)
4. Iterators are bidirectional

set member functions

1. `size()` and `empty()`.
2. `resize()` and `max_size()`.
3. `insert()`, `erase()` and `clear()`.

How to insert into a set

```
pair<iterator,bool> insert(const value_type& val);
pair<iterator,bool> insert(value_type&& val);

iterator insert(iterator position,
                const value_type& val);
iterator insert(const_iterator position,
                value_type&& val);

template <class InputIterator>
  void insert(InputIterator first,
              InputIterator last);

void insert(initializer_list<value_type> il);
```

Searching a set

```
const_iterator find(const value_type& val) const;
iterator       find(const value_type& val);

      iterator lower_bound(const value_type& val);
const_iterator lower_bound(const value_type& val) co

      iterator upper_bound(const value_type& val);
const_iterator upper_bound(const value_type& val) co

pair<const_iterator,const_iterator>
      equal_range(const value_type& val) const;

pair<iterator,iterator>
      equal_range(const value_type& val);
```

std::multiset

Same interface as set, but can have duplicate values

1. Always sorted (like set)
2. Efficient lookup by value (like set)
3. Iterators are bidirectional (like set)
4. Minor differences in `insert`

How to insert into a multiset

```
iterator insert(const value_type& val);
iterator insert(value_type&& val);

iterator insert(iterator position,
                const value_type& val);
iterator insert(const_iterator position,
                value_type&& val);

template <class InputIterator>
  void insert(InputIterator first,
              InputIterator last);

void insert(initializer_list<value_type> il);
```

std::map

1. stores key-value pairs
2. Efficient lookup by key (logarithmic)
3. Unique keys - no duplicates
4. can use `operator[]` to lookup by key.
5. Iterators are bidirectional
6. `value_type` is `pair<const Key, Value>`

Indexing a map

```
map<int, float> m{{5,5.0}, {1,1.0}, {9,9.0}};
assert(m.size() == 3);
assert(m[1] == 1.0);
m[12] = 34.0; // add new element
assert(m.size() == 4);
m[5] = 17.0; // change existing element
assert(m.size() == 4);
if (m[67] == 0.0) // adds a new element
    ;
assert(m.size() == 5);
assert(m.find(8) == m.end());
```

std::map

Don't use operator[] to search in a map

unordered_map and unordered_set (C++11)

```
template <typename T,
          typename Hash = hash<T>,
          typename Compare = equal_to<T>,
          typename Alloc = allocator<T>
> class unordered_set;
```

```
template <typename Key,
          typename Value,
          typename Hash = hash<Key>,
          typename Compare = equal_to<Key>,
          typename Alloc
               = allocator<pair<const Key,Value>>
> class unordered_map;
```

unordered_map and unordered_set (2)

These classes are implemented using hash tables.
A hash function takes an object and returns a `size_t`. Objects that compare equal MUST have the same hash value.
They're called 'unordered' because the elements are not sorted in the container.
They have member functions for managing the number of buckets and the load factor.

Container adapters

1. Take a container and add functionality
2. `stack`, `queue` and `priority_queue`

stack

```
template <typename T,
          typename Container = deque<T>>
class stack;
```

Member functions:

1. push
2. top
3. pop
4. empty and size

pop does not return a value - that's what top is for.

queue

```
template <typename T,
          typename Container = deque<T>>
class queue;
```

Member functions:

1. push
2. front
3. back
4. pop
5. empty and size

priority_queue

```
template <typename T,
          typename Container = vector<T>,
          typename Compare
            = less<typename Container::value_type>>
class priority_queue;
```

Member functions:

1. push
2. top
3. pop
4. empty and size

top gets the "largest" value from the queue.

std::string

std::basic_string

A container that holds 'character-like' elements

```
template <typename T,
          typename Traits = char_traits<T>,
          typename Alloc  = allocator<T>
> class basic_string;
```

1. A sequence container
2. Uses a 'traits class' for character operations
3. Has a null at the end
4. Has a small-space optimization
5. Takes an allocator - but may not use it

basic_string (2)

There are four pre-defined specializations of basic_string:

```
typedef basic_string<char>     string;
typedef basic_string<wchar_t>  wstring;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
```

string (3)

Conceptually, `string` is very similar to `vector<char>`:

1. It manages a single block of contiguous storage.
2. It will reallocate the storage block when more space is needed.
3. It is cheap to append, but expensive to insert in the middle or the beginning.

string (4)

`string` has a very rich set of modifiers. Most have options taking:

1. A `string`.
2. A pointer to a null-terminated character sequence (c-string).
3. A pointer and a length. (not necessarily null-terminated)
4. A `string_view`. (C++17)

Some of string::assign's signatures

```
basic_string& assign(const basic_string& str);
basic_string& assign(basic_string&& str);
basic_string& assign
              (basic_string_view<charT, traits> sv);
basic_string& assign(const value_type* s,
                     size_type n);
basic_string& assign(const value_type* s);
basic_string& assign(size_type n, value_type c);
template<class InputIterator>
        basic_string& assign(InputIterator first,
                             InputIterator last);
basic_string& assign(initializer_list<value_type>);
```

string member functions

1. `empty()` and `size()`
2. `operator[]` and `at()` for indexed access
3. `front()`, `back()`, `data()`, `c_str()` and `swap()`
4. `begin()`, `end()`, `rbegin()`, `rend()`
5. `cbegin()`, `cend()`, `crbegin()`, `crend()`
6. `push_back()` and `pop_back()`
7. `clear()` and `resize()`
8. `reserve()` and `shrink_to_fit()`

string search functions

string also has a large set of member functions for searching in the of the string.

1. `find()` and `rfind()`
2. `compare()`
3. `find_first_of()` and `find_last_of()`
4. `find_first_not_of()` and `find_last_not_of()`

Functors

A functor is something that you can call as if it was a function.

1. A function pointer
2. An object that has a `operator()` member function.
3. A `std::function` object.
4. A lambda function

Functor Example

```
int Func(int x) { return x; }
struct Struct {
    int operator() (int x) const { return x; }
};
auto Lambda = [](int x) { return x; };
function<int(int)> Std{Func};

int main ()
{
    Struct s;
    assert(Func(1)      == 1);
    assert(s(2)         == 2);
    assert(Lambda(3)    == 3);
    assert(Std(4)       == 4);
    assert(Struct()(5) == 5);
}
```

Functors in the standard library (1)

There are a large set of predefined functors that do common operations.

1. Comparisons (`less`, `equal_to`, `greater`, `not_equal_to`, etc.)
2. Arithmetic operations (`plus`, `minus`, `multiplies`, `divides`, etc.)
3. Logical operations (`logical_and`, `logical_or`, `logical_xor`, etc.)
4. Bitwise operations (`bit_and`, `bit_or`, `bit_xor`, etc.)

Functors in the standard library (2)

There are a lot of places that the standard library uses functors to customize templates.

1. Algorithms (`sort`, `find_if`, `equal`, `remove_if`, etc.)
2. Containers (`set`, `map`, `unordered_set`, `unordered_map`, etc.)

Object Lifetimes - reprise

An object's lifetime begins when its constructor completes successfully.
An object's lifetime ends when its destructor is entered.

1. A constructor turns raw memory into an object.
2. A destructor turns an object into raw memory.

Resource Acquisition is Initialization (RAII)

A very simple idea: Acquire a resource in an object's constructor, and release it in the object's destructor.

1. No leaks - the destructors run automatically.
2. They run even if you return early.
3. They run even if something throws an exception.

RAII Example (1)

```
struct FileHolder {
    FileHolder() : f_(nullptr) {}
    FileHolder(FILE *f) : f_(f) {}

    ~FileHolder()
    { if (f_) ::fclose(f_); }

    operator FILE *() const {return f_;}
    explicit operator bool () const
    {return f_ != nullptr;}

private:
    FILE * f_;
};
```

RAII Example (2)

```
bool foo(const char *name) {
    FileHolder f(fopen(name, "rb"));
    if (f) {
        array<char, 100> buffer;
        fread(buffer.data(),
              sizeof(char),
              buffer.size(),
              f);
        /* other stuff */
        return true;
        }
    return false;
}
```

Smart pointers

The most popular use of RAII.
Manage the lifetime of an object on the heap; delete the object when the smart pointer is destroyed.

1. Before C++11, we had `auto_ptr`
2. C++11 introduced `unique_ptr` and `shared_ptr`
3. C++11 also deprecated `auto_ptr`

What happened to auto_ptr?

Before C++11, we had `std::auto_ptr`.
It was deprecated in C++11, and has been removed outright in the upcoming C++17 standard.
`std::unique_ptr` is the replacement for `auto_ptr`

1. `auto_ptr` was an attempt to implement move semantics without language support. It worked for simple cases, and failed subtly in other cases.
2. You couldn't put `auto_ptrs` into containers.

std::unique_ptr

unique_ptr

1. Look and acts like a pointer
2. Low overhead lifetime management
3. No shared ownership
4. Not copyable, but can be moved.

```
template <class T, class D = default_delete<T>>
  class unique_ptr;

template <class T, class D>
  class unique_ptr<T[],D>;
```

Using unique_ptr

```
struct T { void doMe(){}}; // does something

unique_ptr<T> factory(/*params*/)
{
    return unique_ptr<T>(new T(/*params*/));
}

int main () {
   auto p = factory(/*params*/);
   p->doMe(); // calls member function
   return 0;
   }
```

So how does it do that?

A smart pointer overloads `operator *` and `operator ->` to provide a pointer-like interface.

How do I customize this?

`unique_ptr` has a second template parameter, called a "deleter".
This allows you to provide code to be called to clean up the resource managed by the `unique_ptr`.

The default deleter, `std::default_delete<T>` just calls `delete` on the pointer:

```
template <class T>
struct default_delete
{
  default_delete() noexcept = default;
  void operator() (T* ptr) const noexcept
  { delete ptr; }
};
```

std::shared_ptr

shared_ptr: sometimes you just want to share

1. Look and acts like a pointer
2. Shared ownership semantics
3. Copyable
4. No shared ownership
5. weak_ptr support (more on this later)

Using shared_ptr

std::shared_ptr can be used just like a unique_ptr

```
struct T { void doMe(){}}; // does something

shared_ptr<T> factory(/*params*/)
{
    return unique_ptr<T>(new T(/*params*/));
}

int main () {
    auto p = factory(/*params*/);
    p->doMe(); // calls member function
    return 0;
    }
```

How does shared_ptr work? (1)

A shared_ptr consists of three parts:

1. The actual shared_ptr object. This is just a pointer.
2. A "control block" on the heap that contains:
   1. A pointer to the referenced object.
   2. The deleter
   3. A count of the number of shared_ptrs that reference this object.
3. The actual object referenced by the shared_ptr.

How does shared_ptr work? (2)

When you copy a shared_ptr:

1. You make a new shared_ptr object (just a pointer)
2. The reference count in the control block is incremented.

When a shared_ptr is destroyed:

1. The reference count in the control block is decremented.
2. If the reference count is now zero, then:
    1. The object referenced by the shared_pointer is destroyed (using the deleter)
    2. The control block is destroyed.

Assignment is just a destruction then a copy

What about cycles?

You can make a list of objects using `shared_ptr`. Each object holds a `shared_ptr` to the next item in the list.
When you delete the first element in the list, it then destroys the `shared_ptr` to the second item, which deletes the second item, and so on.
However, if you make a circular list, then the data structure will live forever, because the reference counts in the `shared_ptr` will never go to zero.

weak_ptr

a weak_ptr solves the cycle problem.
If you have a cyclic data structure, you should use a weak_ptr; this will allow you to free the data structure when you're through with it.

You cannot use a weak_ptr (as a pointer) directly; instead, you can get a shared_ptr from it - if the underlying object has not been deleted.

weak_ptr - how does it work?

A weak_ptr contains a pointer to a control block, just like a shared_ptr. The control block contains two counts: the first one records the number of shared_ptrs, and the second records the number of weak_ptrs.
When there are no more shared_ptrs, the object is deleted. If there no weak_ptrs, then the control block is deleted as well.

make_shared

A shared_ptr normally requires two heap allocations; one for the control block, the other for the actual object. For some cases, this extra overhead (either in space or in time) is too large.

`make_shared` solves this problem: it allocates a single memory block that holds both the control block and the allocated object.

Algorithms

Introduction to Algorithms

Algorithms are generic actions (usually) on sequences
They use iterators to traverse the sequences.

min - a simple algorithm

```
template <typename T>
const T& min(const T& a, const T& b)
{
    return b < a ? b : a;
}

template <typename T, typename Compare>
const T&
min(const T& a, const T& b, Compare comp)
{
    return comp(b, a) ? b : a;
}
```

Types of algorithms

1. Non-mutating Algorithms
2. Mutating Algorithms
3. Numeric Algorithms
4. Utilities

Non-mutating Algorithms (1)

1. `for_each`
2. Searching for an element or a pattern
3. Find a minimum or maximum
4. Comparing sequences
5. Sequence categorization

for_each

for_each calls a functor for each element in the sequence.

```
template <typename InIter, typename Function>
Function
for_each(InIter first, InIter last, Function f);


// Sample implementation
template <typename InIter, typename Function>
Function
for_each(InIter first, InIter last, Function f)
{
    for (; first != last; ++first)
        f(*first);
    return f;
}
```

Searching a sequence (1)

`find` searches for a value or a condition

```
template <typename InIter, typename T>
InIter find(InIter first, InIter last,
            const T& value);

template <typename InIter, typename Predicate>
InIter find_if(InIter first, InIter last,
               Predicate pred);

template <typename InIter, typename Predicate>
InIter find_if_not(InIter first, InIter last,
                   Predicate pred);
```

Searching a sequence (2)

`find_first_of` searches for one of a list of values

```
template <typename FwdIter1, typename FwdIter2>
FwdIter1
find_first_of(FwdIter1 first1, FwdIter1 last1,
              FwdIter2 first2, FwdIter2 last2);

template <typename FwdIter1, typename FwdIter2,
          typename BinaryPredicate>
FwdIter1
find_first_of(FwdIter1 first1, FwdIter1 last1,
              FwdIter2 first2, FwdIter2 last2,
              BinaryPredicate pred);
```

Searching a sequence (3)

search searches for a pattern

```
template <typename FwdIter1, typename FwdIter2>
FwdIter1 search(FwdIter1 first1, FwdIter1 last1,
                FwdIter2 first2, FwdIter2 last2);

template <typename FwdIter1, typename FwdIter2,
          typename BinaryPredicate>
FwdIter1 search(FwdIter1 first1, FwdIter1 last1,
                FwdIter2 first2, FwdIter2 last2,
                BinaryPredicate pred);
```

Searching a sequence (4)

`find_end` is like `search`, but it finds the last occurrence.

```
template <typename FwdIter1, typename FwdIter2>
Iter1 find_end(FwdIter1 first1, FwdIter1 last1,
               FwdIter2 first2, FwdIter2 last2);

template <typename FwdIter1, typename FwdIter2,
          typename BinaryPredicate>
Iter1 find_end(FwdIter1 first1, FwdIter1 last1,
               FwdIter2 first2, FwdIter2 last2,
               BinaryPredicate pred);
```

Counting values in a sequence

```
// In <iterator>
template <typename InIter>
typename iterator_traits<InIter>::difference_type
distance(Iter first, Iter last);

// In <algorithm>
template <typename Iter, typename T>
typename iterator_traits<Iter>::difference_type
count(Iter first, Iter last, const T& value);

template <typename Iter, typename Predicate>
typename iterator_traits<Iter>::difference_type
count_if(Iter first, Iter last, Predicate pred);
```

The associative containers have member functions count that take a value.

Finding a minimum value

```
template <typename T>
const T&
min(const T& a, const T& b);

template <typename T, typename Compare>
const T&
min(const T& a, const T& b, Compare comp);

template<typename T>
T
min(initializer_list<T> t);

template<typename T, typename Compare>
T
min(initializer_list<T> t, Compare comp);
```

Finding a maximum value

```
template <typename T>
const T&
max(const T& a, const T& b);

template <typename T, typename Compare>
const T&
max(const T& a, const T& b, Compare comp);

template<typename T>
T
max(initializer_list<T> t);

template<typename T, typename Compare>
T
max(initializer_list<T> t, Compare comp);
```

"clamping" a value (C++17)

```
template<typename T>
const T& clamp(const T& v,
               const T& lo, const T& hi);

template<typename T, typename Compare>
const T& clamp(const T& v,
               const T& lo, const T& hi,
               Compare comp);
```

Finding the minimum element in a sequence

```
template <typename FwdIter>
FwdIter
min_element(FwdIter first, FwdIter last);

template <typename FwdIter, typename Compare>
FwdIter
min_element(FwdIter first, FwdIter last,
            Compare comp);
```

Finding the maximum element in a sequence

```
template <typename FwdIter>
FwdIter
max_element(FwdIter first, FwdIter last);

template <typename FwdIter, typename Compare>
FwdIter
max_element(FwdIter first, FwdIter last,
            Compare comp);
```

Finding the minimum and maximum element in a sequence

```
template<typename FwdIter>
pair<FwdIter, FwdIter>
minmax_element(FwdIter first, FwdIter last);

template<typename FwdIter, typename Compare>
pair<FwdIter, FwdIter>
minmax_element(FwdIter first, FwdIter last,
               Compare comp);
```

Comparing sequences (1)

```
template <typename InIter1, typename InIter2>
bool equal(InIter1 first1, InIter1 last1,
           InIter2 first2);

// Added in C++14
template <typename InIter1, typename InIter2>
bool equal(InIter1 first1, InIter1 last1,
           InIter2 first2, InIter2 last2);
```

There are two more algorithms here taking a comparison predicate.

Comparing sequences (2)

```
template <typename InIter1, typename InIter2>
pair<InIter1, InIter2>
mismatch(InIter1 first1, InIter1 last1, InIter2 firs

// Added in C++14
template <typename InIter1, typename InIter2>
pair<InIter1, InIter2>
mismatch(InIter1 first1, InIter1 last1,
         InIter2 first2, InIter2 last2);
```

Returns the position in each sequence where they diverge.
There are two more algorithms here taking a comparison predicate.

Comparing sequences (3)

```
template <typename FwdIter1, typename FwdIter2>
bool is_permutation(FwdIter1 first1,
                    FwdIter1 last1,
                    FwdIter2 first2);

// Added in C++14
template <typename FwdIter1, typename FwdIter2>
bool is_permutation(FwdIter1 first1,
                    FwdIter1 last1,
                    FwdIter2 first2,
                    FwdIter2 last2);
```

Tells you if one sequence is a permutation of the other - the same elements, but in a different order.
There are two more algorithms here taking a comparison predicate.

Operations on sorted sequences (1)

```
template <typename FwdIter, typename T>
FwdIter lower_bound(FwdIter first, FwdIter last,
                    const T& value);

template <typename FwdIter, typename T>
FwdIter upper_bound(FwdIter first, FwdIter last,
                    const T& value);

template <typename FwdIter, typename T>
pair<FwdIter, FwdIter>
equal_range(FwdIter first, FwdIter last,
            const T& value);
```

There are three more algorithms here taking a comparison predicate.

Operations on sorted sequences (2)

```
template <typename InIter1, typename InIter2>
bool includes(InIter1 first1, InIter1 last1,
              InIter2 first2, InIter2 last2);
```

There is another algorithm here taking a comparison predicate.
Returns true if if every element from $[first2, last2)$ is a member of $[first, last)$.

Mutating Algorithms (1)

1. Copying or filling a sequence
2. Rearranging a sequence
3. Replacing elements in a sequence
4. Removing duplicates
5. Remove/Transform/Swap

Copying a sequence (1)

```
template <typename InIter, typename OutIter>
OutIter copy(InIter first, InIter last,
             OutIter result);
```

```
template<typename InIter, typename OutIter,
         typename Predicate>
OutIter copy_if(InIter first, InIter last,
                OutIter result, Predicate pred);
```

Copying a sequence (2)

```
template<typename InIter, typename Size,
         typename OutIter>
OutIter copy_n(InIter first, Size n,
               OutIter result);
```

```
template <typename BiDiIter1, typename BiDiIter2>
BiDiIter2
copy_backward(BiDiIter1 first, BiDiIter1 last,
              BiDiIter2 result);
```

Filling a sequence

```
template <typename FwdIter, typename T>
void
fill(FwdIter first, FwdIter last, const T& value);

template <typename OutIter, typename Size,
          typename T>
OutIter
fill_n(OutIter first, Size n, const T& value);
```

Sorting a sequence (1)

```
template<typename RanIter>
void sort(RanIter first, RanIter last);

template<typename RanIter>
void stable_sort(RanIter first, RanIter last);
```

stable_sort leaves equivalent values in the same relative order as before.

```
template <typename FwdIter>
bool is_sorted(FwdIter first, FwdIter last);
```

There's a second version of each of these that takes a comparison predicate.

Sorting a sequence (2)

```
template<typename RanIter>
void partial_sort(RanIter first, RanIter middle,
                  RanIter last);
```

Places the first `middle-first` sorted elements from the range [*first*, *last*) into the range [*first*, *middle*). The rest of the elements in the range [middle, last) are placed in an unspecified order.

```
template <typename InIter, typename RanIter>
RanIter
partial_sort_copy(InIter first, InIter last,
                  RanIter result_first,
                  RanIter result_last);
```

Like `partial_sort`, but writes the results into the range starting at `result_first`.
Again, there's a second version of each of these that takes a comparison predicate.

Sorting a sequence (3)

```
template <typename RanIter>
void nth_element(RanIter first,
                 RanIter nth,
                 RanIter last);
```

`nth_element` is a partial sorting algorithm that rearranges elements in [*first*, *last*) such that:

1. The element pointed at by `nth` is changed to whatever element would occur in that position if [*first*, *last*) was sorted.
2. All of the elements before this new nth element are less than or equal to the elements after the new nth element.

Again, there's a second version that takes a comparison predicate.

Partitioning a sequence (1)

Given a predicate, a sequence is partitioned if all the elements in the sequence that satisfy the predicate occur before any elements that fail to satisfy the predicate.

The sequence `{0, 15, 10, 5, 20, 20, 25}` is partitioned with respect to the predicate:
`[](int item) {return item < 20;}`,
but not for the predicate:
`[](int item) {item % 2 == 0;}`.

Partitioning a sequence (2)

```
template <typename FwdIter, typename Predicate>
FwdIter
partition(FwdIter first, FwdIter last,
          Predicate pred);

template <typename FwdIter, typename Predicate>
FwdIter
stable_partition(FwdIter first, FwdIter last,
                 Predicate pred);

template <typename InIter, typename Predicate>
bool
is_partitioned(InIter first, InIter last,
               Predicate pred);
```

Partitioning a sequence (3)

```
template<typename FwdIter, typename Predicate>
FwdIter
partition_point(FwdIter first, FwdIter last,
                Predicate pred);
```

Returns the first element in the sequence that fails to satisfy the predicate.

```
template <typename InIter, typename OutIter1,
          typename OutIter2, typename Predicate>
pair<OutIter1, OutIter2>
partition_copy(InIter first, InIter last,
               OutIter1 out_t, OutIter2 out_f,
               Predicate pred);
```

Copies all the elements that satisfy the predicate to out_t and all the others to out_f.

Rotating a sequence

```
template <typename FwdIter>
FwdIter
rotate(FwdIter first, FwdIter middle, FwdIter last);

template <typename FwdIter, typename OutIter>
OutIter
rotate_copy(FwdIter first, FwdIter middle,
            FwdIter last, OutIter result);
```

After the call to `rotate`, the elements [*middle*, *last*) are at the start of the sequence, and followed by [*first*, *middle*).

Replacing values in a sequence (1)

```
template <typename FwdIter, typename T>
void replace(FwdIter first, FwdIter last,
             const T& old_value,
             const T& new_value);

template <typename FwdIter, typename Predicate,
          typename T>
void replace_if(FwdIter first, FwdIter last,
                Predicate pred,
                const T& new_value);
```

Replaces all of the appropriate elements in [*first*, *last*) with `new_value`.

Replacing values in a sequence (2)

```
template <typename InIter, typename OutIter,
          typename T>
OutIter replace_copy(InIter first, InIter last,
                     OutIter result,
                     const T& old_value,
                     const T& new_value);

template <typename InIter, typename OutIter,
          typename Predicate, typename T>
OutIter replace_copy_if(InIter first, InIter last,
                        OutIter result,
                        Predicate pred,
                        const T& new_value);
```

Removing elements from a sequence (1)

Q: How do you remove elements from a sequence, given a pair of iterators?
A: You don't.

In general, you can't affect the size of a sequence through an iterator. Usually, you have some kind of container that manages the storage.

Each container in the standard library (except `array`) has a member function named `erase` which removes one or more elements from the container.

Removing elements from a sequence (2)

```
template <typename FwdIter, typename T>
FwdIter remove(FwdIter first, FwdIter last,
               const T& value);

template <typename FwdIter, typename Predicate>
FwdIter remove_if(FwdIter first, FwdIter last,
                  Predicate pred);
```

`remove` and `remove_if` take all the elements in the sequence that are "to be kept" and moves them to the beginning of the sequence, and returns an iterator to the first element in the sequence which is to be removed.

Removing elements from a sequence (3)

So how do I actually get rid of something?

```
vector<int> v{0,1,3,5,7,9,2,4,6,8};
auto pred = [](int i){ return i % 3 == 0; };
assert(v.size() == 10);
auto it = remove_if(v.begin(), v.end(), pred);
assert(v.size() == 10);  // nothing removed!
v.erase(it, v.end());
assert(v.size() == 6);   // and they're gone!

// This is called the "erase-remove" idiom
v.erase(remove(v.begin(), v.end(), 4), v.end());
assert(v.size() == 5);
```

Removing elements from a sequence (4)

```
template <typename InIter, typename OutIter,
          typename T>
OutIter remove_copy(InIter first, InIter last,
                    OutIter result, const T& value);

template <typename InIter, typename OutIter,
          typename Predicate>
OutIter remove_copy_if(InIter first, InIter last,
                       OutIter result,
                       Predicate pred);
```

Copies elements from the range [*first*, *last*), to another range beginning at result, omitting the elements which satisfy specific criteria.

Unique Elements (1)

```
template <typename FwdIter>
FwdIter unique(FwdIter first, FwdIter last);

template <typename FwdIter,
          typename BinaryPredicate>
FwdIter unique(FwdIter first, FwdIter last,
               BinaryPredicate pred);
```

Works similarly to `remove` and `remove_if`; duplicates are not preserved in their original forms.
These only look for adjacent duplicates. If you want to remove all the duplicates, then you should feed it sorted input.

Unique Elements (2)

```
template <typename InIter, typename OutIter>
OutIter unique_copy(InIter first, InIter last,
                    OutIter result);

template <typename InIter, typename OutIter,
          typename BinaryPredicate>
OutIter unique_copy(InIter first, InIter last,
                    OutIter result,
                    BinaryPredicate pred);
```

Transforming the elements of a sequence

```
template <typename InIter, typename OutIter,
          typename UnaryOp>
OutIter transform(InIter first, InIter last,
                  OutIter result, UnaryOp op);

template <typename InIter1, typename InIter2,
          typename OutIter, typename BinaryOp>
OutIter transform(InIter1 first1, InIter1 last1,
                  InIter2 first2,
                  OutIter result,
                  BinaryOp op);
```

Applies the given operation to each element of the input range(s), and stores the results into the output range starting at `result`.

Transforming the elements of a sequence (2)

```
map<int, string> m {{1, "One"},
                    {2, "Two"}, {3, "Three"}};
vector<string> v;

int main () {
    transform(m.begin(), m.end(),
        back_inserter<vector<string>>(v),
        [](auto p) { return p.second; });
    for (const auto &s:v) cout << s << ' ';
    }
```

Swapping via iterators

```
template <typename FwdIter1, typename FwdIter2>
void iter_swap(FwdIter1 a, FwdIter2 b);

template <typename FwdIter1, typename FwdIter2>
FwdIter2 swap_ranges(FwdIter1 first1,
                     FwdIter1 last1,
                     FwdIter2 first2);
```

Numeric Algorithms (1)

1. Accumulate
2. Adjacent difference
3. Partial sums, etc.
4. Working with permutations