neueda

STL

02 – Adapters and Algorithms

Contents

Adapters

- `std::stack`
- `std::queue`
- `std::priority_queue`

Algorithms

- Managing data
- Searching
- Sorting

neueda

Adapters

02 – Adapters and Algorithms

STL Container Adapters

The STL has 3 class templates that have adapted existing Containers into the containers **`stack<>, queue<>`** and **`priority_queue<>`**

Each of these container adapters have a default inner containers where items are stored.

`stack<>` defaults to **`std::deque<>`**

`queue<>` defaults to **`std::deque<>`**

`priority_queue<>` defaults to **`std::vector<>`**

Programmers can changed these if and when they see fit

std::stack

```
#include <stack>
```

Also knows as **LIFO**

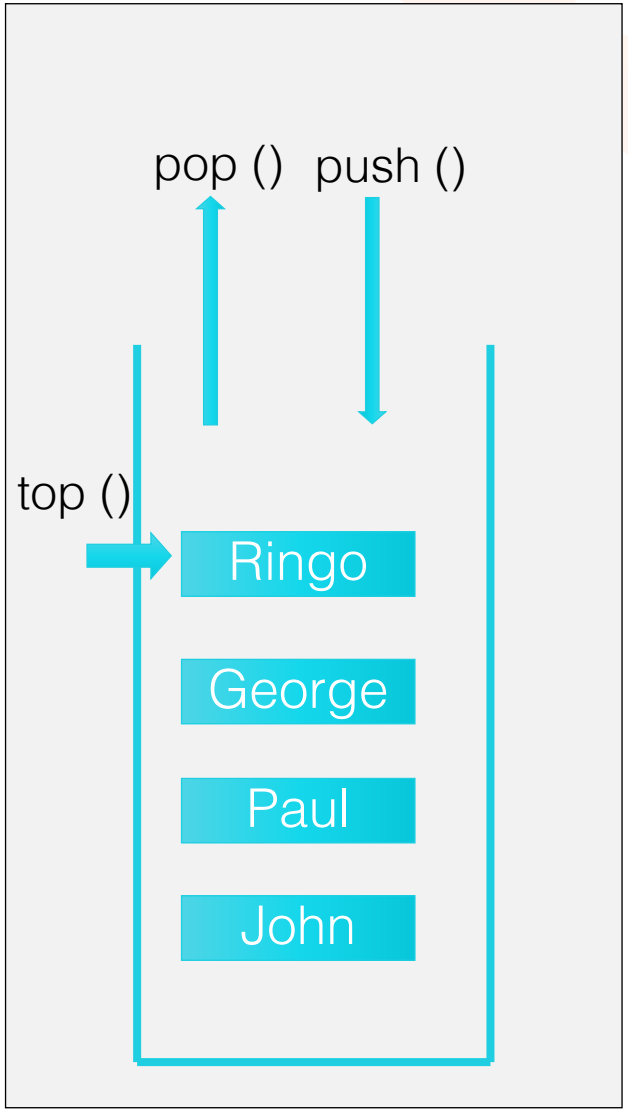
With **push()** you can insert any number of elements into the stack

With **pop()** you can remove the elements in the opposite order in which there were inserted (Last-In, First-Out)

With **top()** you can get a reference to the top element of the stack

NOTE:

pop() removes the top element but does not return it
top() returns the next element without removing it



neueda

The Passenger Class

To demonstrate uses of the STL container classes a simple Passenger class will be used.

The idea is to encapsulate the essential details of passengers on the Titanic so that, in the highly unlikely event of being hit by an ice-berg, passengers get onto available lifeboats in a calm and orderly manner.

Here, Passengers have

- a first and last name,
- a type (First class, second class or third class)
- a gender

This will allow women before men and first_class before second_class before third_class

The Passenger Class

```
// STL Headers
#include <string>
#include <iosfwd>

// Forward Declarations
class Passenger;

// comparison operators
bool operator<  (const Passenger& lhs, const Passenger& rhs);
bool operator== (const Passenger& lhs, const Passenger& rhs);

enum class PassengerType { FIRST_CLASS, SECOND_CLASS, THIRD_CLASS};
enum class GenderType    { FEMALE, MALE};

class Passenger
{
public:
     friend bool operator<  (const Passenger& lhs, const Passenger& rhs);
     Friend bool operator== (const Passenger& lhs, const Passenger& rhs);

      /// .....
}; // class Passenger
```

The Passenger Class

```
bool operator== (const Passenger& lhs, const Passenger& rhs) {
            return lhs.IsEqual(rhs);
}
bool operator<  (const Passenger& lhs, const Passenger& rhs) {
            return lhs.IsLessThan(rhs);
}
bool Passenger::IsEqual(const Passenger& src) const {
            return _firstName == src._firstName &&
                   _lastName == src._lastName &&
                   _passengerType == src._passengerType &&
                   _gender == src._gender;

}

bool Passenger::IsLessThan(const Passenger& src) const {
            return ( (_gender > src._gender) ||
                     ((_gender == src._gender) && (_passengerType > src._passengerType)).
                   );
}
```

Building a Passenger List

```
Passengers CreatePassengers() {
    Passengers passengers;

    passengers.emplace_back("John", "Lennon", PassengerType::FIRST_CLASS, GenderType::MALE);
    passengers.emplace_back("Yoko", "Ono", PassengerType::FIRST_CLASS, GenderType::FEMALE);
    passengers.emplace_back("Julian", "Lennon", PassengerType::FIRST_CLASS, GenderType::MALE);

    passengers.emplace_back("Paul", "McCartney", PassengerType::SECOND_CLASS, GenderType::MALE);
    passengers.emplace_back("Lynda", "McCartney", PassengerType::SECOND_CLASS, GenderType::FEMALE);
    passengers.emplace_back("Stella", "McCartney", PassengerType::SECOND_CLASS, GenderType::FEMALE);

    passengers.emplace_back("George", "Harrison", PassengerType::THIRD_CLASS, GenderType::MALE);
    passengers.emplace_back("Patti", "Boyd", PassengerType::THIRD_CLASS, GenderType::FEMALE);
    passengers.emplace_back("Dhani", "Harrison", PassengerType::THIRD_CLASS, GenderType::MALE);
    passengers.emplace_back("Olivia", "Harrison", PassengerType::THIRD_CLASS, GenderType::FEMALE);

    passengers.emplace_back("Ringo", "Starr", PassengerType::SECOND_CLASS, GenderType::MALE);

    return passengers;
}
```

In header file somewhere
using **`Passengers = std::vector<Passenger>;`**

Using the std::stack<>

```
using BealtesStack = std::stack<Passenger>;
BealtesStack beatles;

Passengers passengers = CreatePassengers();

for (const auto& p : passengers)
     beatles.push(p);

std::cout << "TOP  OF STACK =====> " << beatles.top() << std::endl;
std::cout << "SIZE  OF QUEUE =====> " << beatles.size() << std::endl;


while (! beatles.empty() )
{
     std::cout << beatles.top() << std::endl;

     beatles.pop();
}
```

A Stack of Beatles

Push each beatle onto the stack

Some details of the stack

top – returns a reference to the top element
Does not remove it

pop – removes top element from stack, does not return anything

std::queue

```
#include <queue>
```

Also knows as FIFO

push() inserts an element into the queue

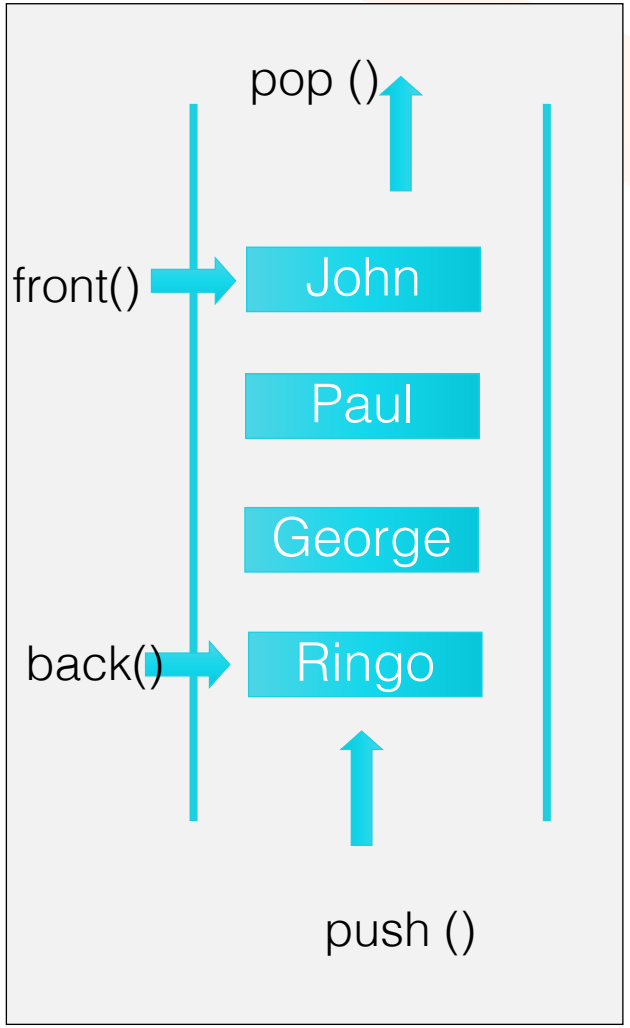
front(), returns the next available element in the queue (the element that was returned last)

back() returns the last available element in the queue (the element that was inserted last)

pop() removes an element from the queue.

NOTE:

pop() removes the top element but does not return it.
front() and **back()** returns the element without removing it



neueda

Using the `std::queue<>`

```
using BealtesQueue = std::queue<Passenger>;
BealtesQueue beatles;

Passengers passengers = CreatePassengers();

for (const auto& p : passengers)
     beatles.push(p);

std::cout << "FRONT OF QUEUE =====> " << beatles.front() << std::endl;
std::cout << "BACK  OF QUEUE =====> " << beatles.back() << std::endl;
std::cout << "SIZE  OF QUEUE =====> " << beatles.size() << std::endl;

while (! beatles.empty() )
{
    std::cout << beatles.front() << std::en

    beatles.pop();
}
```

A Queue of Beatles

push each beatle into the queue

Some details of the queue

front & **back** – return references to the top element
They do not remove it

pop – removes top element from stack, does not return anything

std::priority_queue

```
#include <queue>
```

Elements inserted into queue according to their priority.
First in is not necessarily first out.

push() inserts an element into the queue

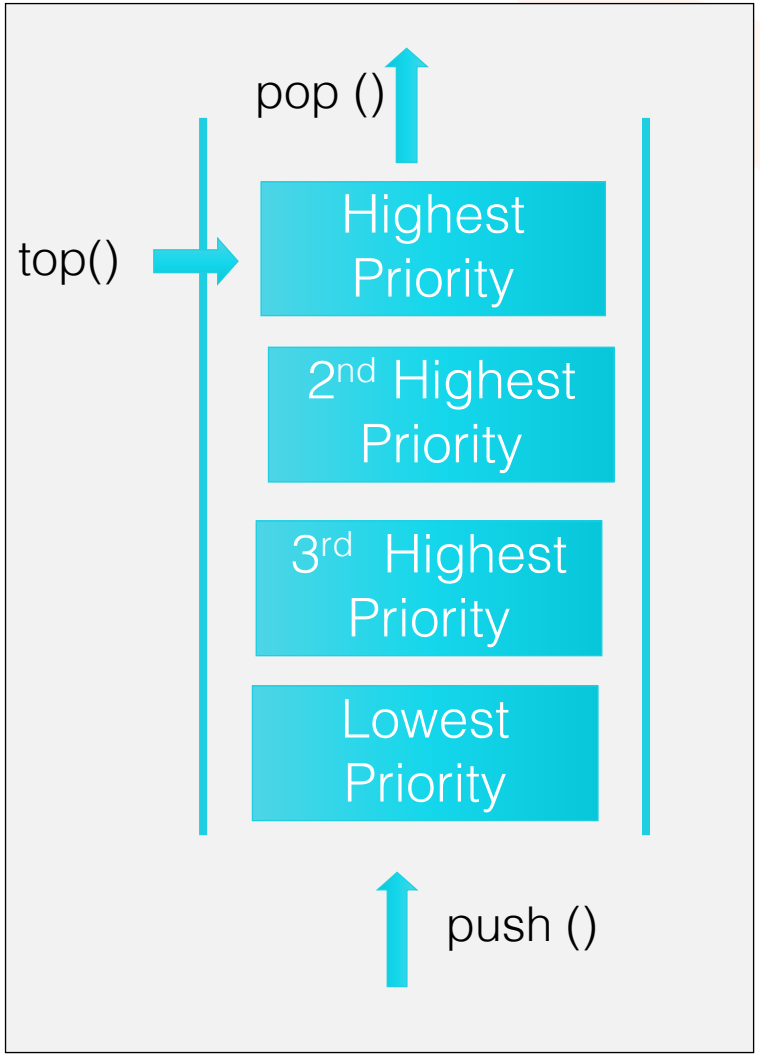
top() returns the next available element in the queue (the element that was returned last)

pop() removes an element from the queue.

NOTE:

As with other containers

pop() removes the top element but does not return it
top() returns the next element without removing it



Using the std::priority_queue<>

```
using BealtesPriorityQueue = std::priority_queue<Passenger>;
BealtesPriorityQueue beatles;

Passengers passengers = CreatePassengers();

for (const auto& p : passengers)
     beatles.push(p);

std::cout << "TOP OF PRIORITY QUEUE =====> " << beatles.top() << std::endl;
std::cout << "SIZE  OF PRIORITY QUEUE =====> " << beatles.size() << std::endl;

while (! beatles.empty() )
{
     std::cout << beatles.top() << std::endl;

     beatles.pop();
}
```

A Priority Queue of Beatles

push each beatle into the container

Some details of the container

top – returns a reference to the top element
Does not remove it

pop – Same as pop in other adapters

neueda

Algorithms

02 – Adapters and Algorithms

Contents

Introduction to Algorithms and Iterators

Overview

Iterator categories

Input iterators

Output iterators

Forward iterators

Bidirectional iterators

Random access iterators

STL Algorithms

The STL provides various template functions known as ***"algorithms"***

- Most are defined in the **`<algorithm>`** header
- Also some numeric algorithms defined in the **`<numeric>`** header

Algorithms use **iterators**

- You typically provide a start iterator and an end iterator
- The **start** iterator is **inclusive**
- The **end** iterator is **exclusive**

Algorithms are extremely generic!

- **Iterators** => you can use any container type
- **Templates** => you can use any data types
- **Function objects** => you can apply any functionality

Example Algorithm: for_each (1 of 3)

Here's a example of the **for_each** algorithm

- Takes iterators that indicate the start and end of a collection
- Takes a unary function that will be applied on each item

```
#include <vector>
#include <algorithm>
using namespace std;
…

vector<int> vec;
vec.push_back(3);
vec.push_back(15);
vec.push_back(29);

for_each(vec.begin(),
         vec.end(),
         printer<int>());
```

```
template <typename T>
struct printer
{
  void operator() (const T& n) const;
};

template <typename T>
void printer<T>::operator() (const T& n) const
{
  cout << n << " ";
}
```

Example Algorithm: for_each (2 of 3)

Here's the formal definition of the **for_each** algorithm in the **<algorithm>** header file

```
template <typename InputIterator, typename UnaryFunction>
UnaryFunction
for_each(InputIterator first, InputIterator last, UnaryFunction f);
```

for_each has two template type parameters:

- An "input iterator"
  - More details in a moment …
- A unary function
  - i.e. either a pointer to a function that takes a single argument
  - or an object that has an operator() method that takes a single argument

Example Algorithm: for_each (3 of 3)

```
std::vector<int> collection{ 1,2,3,4,5,6,7,8,9 };

std::cout << "Initial Series" << std::endl;
PrintCollection(collection);

std::for_each(collection.begin(), collection.end(), [](int& n) { n = n*n; });
std::cout << "After Lambda" << std::endl;
PrintCollection(collection);

Total t = std::for_each(collection.begin(), collection.end(), Total());
std::cout << "Collection Total " << t.total << std::endl;
```

```
void PrintCollection(const std::vector<int>& collection)
{
        for (const auto& c : collection) std::cout << ' ' << c;
        std::cout << std::endl;
}
```

OUTPUT

```
Initial Series
 1 2 3 4 5 6 7 8 9
After Lambda
 1 4 9 16 25 36 49 64 81
Collection Total 285
```

Iterator Categories

Input iterators
- Single-pass input

Output iterators
- Single-pass output

Forward iterators
- General single-pass

Bidirectional iterators
- Allow iteration in both directions

Random access iterators
- Support constant-time indexing

Input Iterators

Designed for sequential input operations

Each element pointed by the iterator is read once, and then the iterator is incremented

```
template <typename T>
class InputIterator
{
public:
  InputIterator(const InputIterator &);
  ~InputIterator();

  InputIterator & operator=(const InputIterator &);

  InputIterator & operator++();
  InputIterator   operator++(int);

  const T & operator*();
  const T * operator->();

  bool operator==(const InputIterator &) const;
  bool operator!=(const InputIterator &) const;
```

Output Iterators

Designed for sequential output operations

Each element pointed by the iterator is assigned a value once, and then the iterator is incremented

```
template <typename T>
class OutputIterator
{
public:
  OutputIterator(const OutputIterator &);
  ~OutputIterator();

  OutputIterator & operator=(const OutputIterator &);

  OutputIterator & operator++();
  OutputIterator   operator++(int);

  T & operator*();
  …
```

Forward Iterators

Designed for general unidirectional sequential access

Have all the capabilities of input iterators and output iterators

```
template <typename T>
class ForwardIterator
{
public:
  ForwardIterator();
  ForwardIterator(const ForwardIterator &);
  ~ForwardIterator();

  ForwardIterator & operator=(const ForwardIterator &);

  ForwardIterator & operator++();
  ForwardIterator   operator++(int);

  T & operator*();
  T * operator->() const;

  bool operator==(const ForwardIterator &) const;
  bool operator!=(const ForwardIterator &) const;
```

Bidirectional Iterators

Designed for general bidirectional sequential access

- Have all the capabilities of forward iterators, plus ability to go back
- All the STL containers support at least bidirectional iterators

```
template <typename T>
class BidirectionalIterator
{
public:

  // All the ForwardIterator capabilities, plus…

  BidirectionalIterator & operator--();
  BidirectionalIterator   operator--(int);
  …
```

Random Access Iterators

Designed to give full pointer syntax and semantics

- In some cases (e.g. arrays), pointers can actually be used!
- Supported by **`vector`**, **`deque`**, and **`string`** (and arrays!)

```
template <typename T, typename D>
class RandomAccessIterator
{
public:

  // All the BidirectionalIterator capabilities, plus…

  RandomAccessIterator & operator+=(D);
  RandomAccessIterator & operator-=(D);
  RandomAccessIterator   operator+(D) const;
  RandomAccessIterator   operator-(D) const;

  D operator-(RandomAccessIterator) const;
  T &operator[](D) const;

  bool operator< (const RandomAccessIterator &) const;
  bool operator<=(const RandomAccessIterator &) const;
  bool operator> (const RandomAccessIterator &) const;
  bool operator>=(const RandomAccessIterator &) const;
  …
```

Managing Data

Populating items

Copying items

Replacing items

Removing items

Removing adjacent duplicates

Reversing items

Populating Items (1 of 3)

`fill()` and **`fill_n()`**

Fill all (or the first n) items with the specified value

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

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

`generate()` and **`generate_n()`**

Generate a value for all (or the first n) items, via a function

```
template <typename ForwardIterator, typename GeneratorFunction>
void generate(ForwardIterator first, ForwardIterator last, GeneratorFunction gen);
```

```
template <typename OutputIterator, typename Size, typename GeneratorFunction>
void generate_n(OutputIterator first, Size n, GeneratorFunction gen);
```

Populating Items (2 of 3)

```
std::vector<int> collection{ 1,2,3,4,5,6,7,8,9 };

std::cout << "Initial Series" << std::endl;

PrintCollection(collection);

std::fill(collection.begin(), collection.begin() + 2, 5);

std::cout << "After fill" << std::endl;

PrintCollection(collection);

std::fill_n(collection.begin(), collection.size(), 42);

std::cout << "After fill_n" << std::endl;

PrintCollection(collection);
```

OUTPUT

```
Initial Series
 1 2 3 4 5 6 7 8 9
After fill
 5 5 3 4 5 6 7 8 9
After fill_n
 42 42 42 42 42 42 42 42 42
```

Populating Items (3 of 3)

```
std::vector<int> collection{ 1,2,3,4,5,6,7,8,9 };

std::cout << "Initial Series" << std::endl;
PrintCollection(collection);

std::generate(collection.begin(), collection.begin() + 5, Generator());
std::cout << "After generate" << std::endl;
PrintCollection(collection);

std::generate_n(collection.begin() + 5, 4,  Generator());
std::cout << "After generate_n“   << std::endl;
PrintCollection(collection);
```

```
struct Generator {
   int value;

   Generator() { value = 1000; }
   int operator() () {
         return ++value;
   }
}; // Generator
```

OUTPUT

```
Initial Series
 1 2 3 4 5 6 7 8 9
After generate
 1001 1002 1003 1004 1005 6 7 8 9
After generate_n
 1001 1002 1003 1004 1005 1001 1002 1003 1004
```

Copying Items (1 of 2)

copy()

Copies items in the range [first, last) to the result destination

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator first,
                    InputIterator last,
                    OutputIterator result      );
```

copy_backward()

Similar to **copy()**, but begins the copy operation at the end

Useful if the ranges overlap

```
template <typename BidirectionalIterator1, typename
BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
                                     BidirectionalIterator1 last,
                                     BidirectionalIterator2 result);
```

Copying Items (2 of 2)

```
std::vector<int> collection{ 1,2,3,4,5,6,7,8,9 };
std::list<int> destination(collection.size());
std::cout << "Initial Series" << std::endl;

PrintCollection(collection);

std::copy(collection.begin(), collection.end(), destination.begin());

std::cout << "Destination Series" << std::endl;
PrintCollection(destination);

std::fill(collection.begin() + 5, collection.end(), 42);

std::copy_backward(collection.begin(), collection.begin() + 5, destination.end());

std::cout << "After copy backwards - Destination Series" << std::endl;
PrintCollection(destination);
```

OUTPUT
Initial Series
1 2 3 4 5 6 7 8 9
Destination Series
1 2 3 4 5 6 7 8 9
After copy backwards - Destination Series
1 2 3 4 1 2 3 4 5

Replacing Items via Equality (1 of 2)

`replace()`

Replaces items in the range [first, last) that match an old value with a new value

```
template <typename ForwardIterator, typename T>
void replace(ForwardIterator first,
             ForwardIterator last,
             const T & oldValue,
             const T & newValue);
```

`replace_copy()`

Similar to **`replace()`**, but copies to a specified result destination

```
template <typename InputIterator, typename OutputOperator, typename T>
OutputIterator replace_copy(InputIterator first,
                            InputIterator last,
                            OutputIterator result,
                            const T & oldValue,
                            const T & newValue);
```

Replacing Items via Equality (2 of 2)

```
std::vector<int> collection{ 1,6,3,6,5,6,7,6,9 };

std::cout << "Initial Series" << std::endl;

PrintCollection(collection);

std::replace(collection.begin(), collection.end(), 6, 66);

std::cout << "After replace" << std::endl;

PrintCollection(collection);

std::list<int> destination(collection.size());

std::replace_copy(collection.begin(), collection.end(), destination.begin(), 9, 99);

std::cout << "After replace_copy" << std::endl;
PrintCollection(collection);
PrintCollection(destination);
```

OUTPUT
Initial Series
1 6 3 6 5 6 7 6 9
After replace
1 66 3 66 5 66 7 66 9
After replace_copy
1 66 3 66 5 66 7 66 9
1 66 3 66 5 66 7 66 99

Removing Items via Equality (1 of 2)

remove()

Removes items in the range [first, last) that match a value

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

remove_copy()

Copies items in the range [first, last) that DON'T match a value, to the result destination

```
template <typename InputIterator, typename
OutputOperator, typename T>
OutputIterator remove_copy(InputIterator first,
                           InputIterator last,
                           OutputIterator result,
                           const T & value);
```

Removing Items via Equality (2 of 2)

```
std::vector<int> collection{ 1,2,1,2,1,2,1,3,1 };

std::cout << "Initial Series" << std::endl;

PrintCollection(collection);

std::remove(collection.begin(), collection.end(), 1);

std::cout << "After remove" << std::endl;

PrintCollection(collection);

std::list<int> destination(collection.size());

std::remove_copy(collection.begin(), collection.end(), destination.begin(), 3);

std::cout << "After remove copy" << std::endl;

PrintCollection(destination);
```

OUTPUT
Initial Series
1 2 1 2 1 2 1 3 1
After remove
2 2 2 3 1 2 1 3 1
After remove copy
2 2 2 1 2 1 1 0 0

Removing Items via Predicates (1 of 2)

There are "predicate" versions of the removal algorithms from the previous slide

- Rather than comparing elements for equality
- … they receive a predicate function (i.e. returns a bool)
- … and they call the predicate function to test for matches

Here are the predicate-based removal algorithms:

```
template <typename ForwardIterator, typename UnaryPredicate>
ForwardIterator remove_if(ForwardIterator first,
                          ForwardIterator last,
                          UnaryPredicate pred);
```

```
template <typename InputIterator, typename OutputOperator,
typename UnaryPredicate>
OutputIterator remove_copy_if(InputIterator first,
                              InputIterator last,
                              OutputIterator result,
                              UnaryPredicate pred);
```

Removing Items via Predicates (2 of 2)

```
std::vector<int> collection{ 1,2,1,2,1,2,1,3,1 };
std::cout << "Initial Series" << std::endl;
PrintCollection(collection);

std::remove_if(collection.begin(), collection.end(), [](int n) { return (n%2) == 1; });

std::cout << "After remove_if" << std::endl;
PrintCollection(collection);

std::list<int> destination(collection.size());

std::remove_copy_if(collection.begin(), collection.end(),
                    destination.begin(),
                    [](int n) { return (n%2) == 1; });

std::cout << "After remove_copy_if" << std::endl;
PrintCollection(destination);
```

OUTPUT
Initial Series
1 2 1 2 1 2 1 3 1
After remove_if
2 2 2 2 1 2 1 3 1
After remove_copy_if
2 2 2 2 2 0 0 0 0

Removing Adjacent Dups via Equality (1 of 2)

`unique()`

Removes adjacent duplicates in the range [first, last)

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

`unique_copy()`

Copies items in the range [first, last) to the result destination, except for adjacent duplicates (which are not copied)

```
template <typename InputIterator, typename
OutputIterator>
OutputIterator unique_copy(InputIterator first,
                           InputIterator last,
                           OutputIterator result);
```

Removing Adjacent Dups via Equality (2 of 2)

```
std::vector<int> collection{ 1,2,2,3,3,3,4,4,4,4,3,3,3,2,2,1 };
std::cout << "Initial Series" << std::endl;
PrintCollection(collection);

std::vector<int>::iterator it;
it = std::unique(collection.begin(), collection.end());

collection.resize(std::distance(collection.begin(), it));

std::cout << "After unique" << std::endl;
PrintCollection(collection);

std::list<int> destination(collection.size());

collection = { 1,2,2,3,3,3,4,4,4,4,3,3,3,2,2,1 };

std::unique_copy(collection.begin(), collection.end(), destination.begin());


std::cout << "After unique_copy" << std::endl;
PrintCollection(destination);
```

OUTPUT
Initial Series
1 2 2 3 3 3 4 4 4 4 3 3 3 2 2 1
After unique
1 2 3 4 3 2 1
After unique_copy
1 2 3 4 3 2 1

Removing Adjacent Dups - Predicates (1 of 2)

There are "predicate" versions of the adjacent-duplicate-removal algorithms from the previous slide

- Rather than comparing elements for equality
- … they receive a predicate function (i.e. returns a bool)
- … and they call the predicate function to test for matches

Here are the predicate-based algorithms:

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

```
template <typename InputIterator, typename
OutputOperator, typename BinaryPredicate>
OutputIterator unique_copy(InputIterator first,
                           InputIterator last,
                           OutputIterator result,
                           BinaryPredicate pred);
```

Removing Adjacent Dups - Predicates (2 of 2)

```
/// As Before
std::vector<int>::iterator it;
it = std::unique(collection.begin(), collection.end(), IsEqual());

collection.resize(std::distance(collection.begin(), it));

std::cout << "After unique (with predicate)" << std::endl;
PrintCollection(collection);

collection = { 1,2,2,3,3,3,4,4,4,4,3,3,3,2,2,1 };
std::cout << "Before unique_copy (with predicate)" << std::endl;
PrintCollection(collection);

it = std::unique_copy(collection.begin(), collection.end(),
                      collection.begin(), IsEquiDivisible());

collection.resize(std::distance(collection.begin(), it));
```

```
struct IsEqual {
bool operator()(int x, int y)
  {
    return (x == y);
  }
};
```

```
struct IsEquiDivisible {
bool operator()(int x, int y)
  {
    return (x % y == 0);
  }
};
```

OUTPUT

Initial Series
1 2 2 3 3 3 4 4 4 4 3 3 3 2 2 1
After unique (with predicate)
1 2 3 4 3 2 1
Before unique_copy (with predicate)
1 2 2 3 3 3 4 4 4 4 3 3 3 2 2 1
After unique_copy (with predicate)
1 2 3 4 3 2

Reversing Items (1 of 3)

`reverse()`

Reverses items in the range [first, last)

```
template <typename BidirectionalIterator>
void reverse(BidirectionalIterator first,
             BidirectionalIterator last);
```

`reverse_copy()`

Similar to **`reverse()`**, but copies to a specified result destination

```
template <typename BidirectionalIterator, typename OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator);
```

Plus:

`rotate()`
`rotate_copy()`

Reversing Items (2 of 3)

```
std::vector<int> collection{ 0,1,1,2,3,5,8,13,21};
std::cout << "Initial Series" << std::endl;
PrintCollection(collection);

std::reverse(collection.begin(), collection.end());

std::cout << "After Reverse" << std::endl;
PrintCollection(collection);

std::list<int> destination(collection.size());

std::reverse_copy(collection.begin(), collection.end(), destination.begin());

std::cout << "After Reverse_copy" << std::endl;
PrintCollection(destination);
```

OUTPUT

```
Initial Series
 0 1 1 2 3 5 8 13 21
After Reverse
 21 13 8 5 3 2 1 1 0
After Reverse_copy
 0 1 1 2 3 5 8 13 21
```

Reversing Items (3 of 3)

```
std::vector<int> collection{ 0,1,1,2,3,5,8,13,21,34,55 };
std::cout << "Initial Series" << std::endl;
PrintCollection(collection);

std::rotate(collection.begin(), collection.begin() + 6, collection.end());

std::cout << "After rotate" << std::endl;
PrintCollection(collection);

std::list<int> destination(collection.size());

std::rotate_copy(collection.begin(), collection.begin() + 3,
                 collection.end(), destination.begin());

std::cout << "After rotate_copy" << std::endl;
PrintCollection(destination);
```

OUTPUT

```
Initial Series
 0 1 1 2 3 5 8 13 21 34 55
After rotate
 8 13 21 34 55 0 1 1 2 3 5
After rotate_copy
 34 55 0 1 1 2 3 5 8 13 21
```

Searching

Overview of linear searching

Linear searches

Linear searches for min and max elements

Overview of binary searching

Binary searches

Overview of Linear Searching

The STL provides various linear search algorithms

- You provide iterators into a data collection
- Algorithms step through items in sequence, from start to finish
- Return an iterator to the located item, or end iterator if not found

E.g. **`find()`** is a linear search, behaves as follows:

```
template <typename InputIterator, typename T>
InputIterator find(InputIterator iter, InputIterator end, const T & value)
{
  for ( ; iter != end; iter++)
  {
    if (*iter == value)
      break;
  }
  return iter;
}
```

Linear Searches (1 of 4)

`find()`

Finds first equal value in a range

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

`adjacent_find()`

Finds first pair of adjacent equal values in a range

```
template <typename ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator start, ForwardIterator last);
```

`find_first_of()`

Finds first occurrence in source range, of an item in another range

```
template <typename ForwardIterator>
ForwardIterator find_first_of(ForwardIterator start1, ForwardIterator last1,
                              ForwardIterator start2, ForwardIterator last2);
```

Linear Searches (2 of 4)

```
std::vector<int> collection{ 0,1,1,2,3,5,8,13,21,34,55 };
std::cout << "Initial Series" << std::endl;
PrintCollection(collection);

std::vector<int>::iterator it;

it = std::find(begin(collection), end(collection), 21);
if (it != end(collection))
        std::cout << "Found " << *it << std::endl;
else std::cout << "Not Found " << std::endl;

it = std::find(begin(collection), end(collection), 32);
if (it != end(collection))
        std::cout << "Found " << *it << std::endl;
else std::cout << "Not Found " << std::endl;
```

OUTPUT
Initial Series
0 1 1 2 3 5 8 13 21 34 55
Found 21
Not Found

Linear Searches (3 of 4)

```
std::vector<int> collection{ 0,1,1,2,3,5,5,5,8,13,21 };
std::cout << "Initial Series" << std::endl;
PrintCollection(collection);

std::vector<int>::iterator it;

it = std::adjacent_find(begin(collection), end(collection));
if (it != end(collection))
        std::cout << "First pair of adjacent items are " << *it << std::endl;

it = std::adjacent_find(++it, end(collection));
if (it != end(collection))
        std::cout << "Next pair of adjacent items are " << *it << std::endl;

it = std::adjacent_find(++it, end(collection));
if (it != end(collection))
        std::cout << "Next pair of adjacent items are " << *it << std::endl;
```

OUTPUT

```
Initial Series
 0 1 1 2 3 5 5 5 8 13 21
First pair of adjacent items are 1
Next pair of adjacent items are 5
Next pair of adjacent items are 5
```

Linear Searches (4 of 4)

```
std::vector<int> squares{ 1,4,9,16,25,36,49,64,81,100 };
std::vector<int> cubes{ 1,8,27,64,125,216,343,512,729,1000 };
std::vector<int>::iterator it;

it = std::find_first_of(begin(squares), end(squares), begin(cubes), end(cubes));

if (it != end(squares))
        std::cout << "The first cube that is also a square is " << *it << std::endl;

it = std::find_first_of(++it, end(squares), begin(cubes), end(cubes));
if (it != end(squares))
        std::cout << "The next cube that is also a square is " << *it << std::endl;
```

OUTPUT

The first cube that is also a square is 1
The next cube that is also a square is 64

Min and Max Elements (1 of 2)

`min_element()`

You can search for the minimum element, using equality tests:

```
template <typename ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator
```

Or you can use a predicate (that returns `true` if arg1 < arg2)

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

`max_element()`

As above, but return the maximum element

Min and Max Elements (2 of 2)

```
std::vector<int> squares{ 36,49,64,81,100,1,4,9,16,25 };
std::vector<int>::iterator it;
PrintCollection(squares);

it = std::min_element(begin(squares), end(squares));

if (it != end(squares))
        std::cout << "The minimum element of squares is " << *it << std::endl;
it = std::max_element(begin(squares), end(squares));

if (it != end(squares))
        std::cout << "The maximum element of squares is " << *it << std::endl;
```

Output

```
36 49 64 81 100 1 4 9 16 25
The minimum element of squares is 1
The maximum element of squares is 100
```

Overview of Binary Searching

The STL provides various binary search algorithms

- Much more efficient than linear searches, for sorted data
- Logarithmic execution time

The binary search algorithms use "equivalence tests" rather than "equality tests"

a and b are considered equivalent if:

```
(!(a < b) && !(b < a))
```

Binary Searches via Equality (1 of 2)

`binary_search()`

Returns `true` if a specified value is present in the range

```
template <typename ForwardIterator, typename T>
bool binary_search(ForwardIterator first, ForwardIterator last, const T &);
```

`lower_bound()`

Returns an iterator to the first item that is not less than value

```
template <typename ForwardIterator, typename T>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
```

`upper_bound()`

Returns an iterator to the first item that is greater than value

```
template <typename ForwardIterator, typename T>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
```

Binary Searches via Equality (2 of 2)

```
std::vector<int> squares{ 16,25,36,49,64,81,100,36,49,9,16,64,81,100,1,4,9,16,25 };

std::sort(begin(squares), end(squares));
bool found = std::binary_search(squares.begin(), squares.end(), 36);

if (found == true) std::cout << "Found 36" << std::endl;
           else std::cout << "Not Found 36" << std::endl;

std::vector<int>::iterator lower, upper;

lower = std::lower_bound(squares.begin(), squares.end(), 16);
upper = std::upper_bound(squares.begin(), squares.end(), 81);

std::list<int> bounded(lower, upper);
std::cout << "Numbers between lower and “
                 <<  “upper" << std::endl;



PrintCollection(bounded);
```

OUTPUT
Found 36
Numbers between lower and upper
16 16 16 25 25 36 36 49 49 64 64 81 81

Sorting

Overview of sorting

Full sorting

Partial sorting

Partitioning

Overview of Sorting

Associative containers are already implicitly sorted

- i.e. **`set`**, **`multiset`**, **`map`**, and **`multimap`**

Sequential containers are not implicitly sorted

- i.e. **`vector`**, **`list`**, **`deque`**, **`stack`**, **`queue`**

You can sort a container if you need to, by using one of the STL sort algorithms

- Consider this example…

```
vector<string> words;
…

// Default sort (applies < to elements).
sort(words.begin(), words.end());

// Explicit sort (using the greater function object, applies > to elements).
sort(words.begin(), words.end(), greater<string>());
```

Full Sorting (1 of 4)

`sort()`

Sorts elements in a range, by applying `operator<`

Equal elements are **NOT** guaranteed to keep original relative order

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

`stable_sort()`

Similar to **`sort()`**, except…

Equal elements **ARE** guaranteed to keep original relative order

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

Full Sorting (2 of 4)

```
std::vector<Item> masterList = {
      Item(3.23, "Windows Lap Top"),
      Item(2.62, "Mouse"),
      Item(123, "Screen"),
      Item(3.02, "MacBook"),
      Item(3.99, "Second Windows Lap Top"),
      Item(1.31, "External Hard Drive"),
};

std::vector<Item> items( cbegin(masterList), cend(masterList) );

std::cout << "Initial Series" << std::endl;

PrintCollection(items);
```

```
struct Item {
  Item(double id, std::string desc) :
     id(id), desc(desc) {}

   double id;
   std::string desc;
};
```

Full Sorting (3 of 4)

```
std::sort (begin(items), end(items), CompareAsDouble);

std::cout << "After sort" << std::endl;

PrintCollection(items);

items = masterList;

std::cout << "Reset Series" << std::endl;

PrintCollection(items);
```

```
bool CompareAsDouble(const Item& x, const Item& y)
{
        return x.id < y.id;
}
```

OUTPUT

Initial Series

3.23	Windows Lap Top
2.62	Mouse
123	Screen
3.02	MacBook
3.99	Second Windows Lap Top
1.31	External Hard Drive

After sort

1.31	External Hard Drive
2.62	Mouse
3.02	MacBook
3.23	Windows Lap Top
3.99	Second Windows Lap Top
123	Screen

```
items = masterList;
std::cout << "Reset Series" << std::endl;

PrintCollection(items);

std::stable_sort(begin(items), end(items), CompareAsInt);

std::cout << "After sort" << std::endl;

PrintCollection(items);
```

```
bool CompareAsInt(const Item& x,  const Item& y)
{
        return static_cast<int>(x.id) <
                static_cast<int>(y.id);
}
```

OUTPUT

Reset Series

3.23	Windows Lap Top
2.62	Mouse
123	Screen
3.02	MacBook
3.99	Second Windows Lap Top
1.31	External Hard Drive

After sort

1.31	External Hard Drive
2.62	Mouse
3.23	Windows Lap Top
3.02	MacBook
3.99	Second Windows Lap Top
123	Screen

Partial Sorting via Operator < (1 of 2)

partial_sort()

Rearranges elements in the range [first,last), such that…
[first,middle) contains smallest elements, sorted in ascending order
[middle,end) contains remaining elements, in no specific order

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

partial_sort_copy()

Copies smallest elements in [first,last) to [result_first, result_last), sorting the elements copied

```
template <typename InputIterator, typename RandomAccessIterator>
RandomAccessIterator partial_sort_copy(InputIterator first,
                                       InputIterator last,
                                       RandomAccessIterator result_first,
                                       RandomAccessIterator result_last );
```

Partial Sorting via Operator < (2 of 2)

```
std::vector<int> squares{ 16,25,36,49,64,81,100,36,49,9,16,64,81,100,1,4,9,16,25 };

std::cout << "Initial Series" << std::endl;

PrintCollection(squares);

std::partial_sort(begin(squares), squares.begin() + 10, end(squares));

std::cout << "After Partial Sort" << std::endl;

PrintCollection(squares);
```

OUTPUT
Initial Series
16 25 36 49 64 81 100 36 49 9 16 64 81 100 1 4 9 16 25
After Partial Sort
1 4 9 9 16 16 16 25 25 36 100 81 81 100 64 64 49 49 36

Partitioning (1 of 2)

Partitioning is the ultimate in partial sorting!

- Divides a range into two, according to a given predicate
- All the elements that satisfy the predicate are placed before all the elements that do not

`partition()` - Relative ordering within each group is NOT guaranteed

```
template <typename BidirectionalIterator, typename UnaryPredicate>
BidirectionalIterator partition(BidirectionalIterator first,
                                BidirectionalIterator last,
                                UnaryPredicate pred);
```

`stable_partition()` - Similar, except relative ordering within each group IS guaranteed

```
template <typename BidirectionalIterator, typename UnaryPredicate>
BidirectionalIterator stable_partition(BidirectionalIterator first,
                                       BidirectionalIterator last,
                                       UnaryPredicate pred);
```

Partitioning (2 of 2)

```
std::vector<int> squares{
16,25,36,49,64,81,100,36,49,9,16,64,81,100,1,4,9,16,25 };


std::cout << "Initial Series" << std::endl;


PrintCollection(squares);


std::partition(begin(squares), end(squares), IsOdd());


std::cout << "After Partition (IsOdd) " << std::endl;


PrintCollection(squares);
```

```
struct IsOdd {
        bool operator()(int x) {
                return (x % 2 == 1);
        }
};
```

```
OUTPUT
Initial Series
 16 25 36 49 64 81 100 36 49 9 16 64 81 100 1 4 9 16 25
After Partition (IsOdd)
 25 25 9 49 1 81 81 9 49 36 16 64 100 100 64 4 36 16 16
```