Sequence Containers

Tell us a bad dad joke (optional).
Sequence Containers

Bad Dad Joke of the Day:
Bad Dad Joke of the Day:
- What’s another name for oceans?
- Sea++!

Creds: James
Game Plan

- Finishing Up C++ Types
- Survey Results!
- Overview of STL
- Sequence Containers
- Container Adaptors
C++ Types (cont.)
Streams Aside: When do I use...

...cin and cout?

...a filestream (fstream)?

...a stringstream?
Streams Aside: When do I use...

...cin and cout?

...a filestream (fstream)?

...a stringstream?
Streams Aside: When do I use...

...cin and cout?

...a filestream (fstream)?

...a stringstream?
Streams Aside: When do I use...

...cin and cout?

...a filestream (fstream)?

...a stringstream?
stringstream vs. string
When should I use a `stringstream`?

1. Processing strings
   - Simplify "./a/b/.." to "/a"

2. Formatting input/output
   - uppercase, hex, and other stream manipulators

3. Parsing different types
   - `stringToInteger()` from previous lectures
When should I use a `stringstream`?

1. Processing strings
   - Simplify “./a/b/..” to “/a”
2. Formatting input/output
   - uppercase, hex, and other stream manipulators
3. Parsing different types
   - `stringToInteger()` from previous lectures

If you’re just concatenating strings, `str.append()` is faster than using a `stringstream`!
Survey Results!
Majors/Programs:
- Computer Science
- Undecided :)
- Aero/Astro
- Electrical Engineering
- Mechanical Engineering
- SymSys
- And more!
Majors/Programs:
• Computer Science
• Undecided :)
• Aero/Astro
• Electrical Engineering
• Mechanical Engineering
• SymSys
• And more!

Why you’re here:
• Industry usages
• C++ practice
• Supplement CS 106B
• Personal projects
The Standard Template Library (STL)
Overview of STL

“As mathematicians learned to lift theorems into their most general setting, so I wanted to lift algorithms and data structures.”

— Alex Stepanov, inventor of the STL
Overview of STL

- Containers
- Iterators
- Functors
- Algorithms
Overview of STL

Containers

Iterators

Functors

Algorithms

Adaptors
Overview of STL

- Containers
- Iterators
- Functors
- Algorithms
- Adaptors
Sequence Containers
Sequence Containers

Provides access to **sequences** of elements.

Includes:

- `std::vector<T>`
- `std::deque<T>`
- `std::list<T>`
- `std::array<T>`
- `std::forward_list<T>`
std::vector<T>
A vector represents a sequence of elements of any type. You specify the type when using the vector:

```cpp
std::vector<int> vecInt; // vector of ints
std::vector<string> vecStr; // vector of string
std::vector<myStruct> vecStruct; // vector of myStructs
std::vector<std::vector<string>> vecOfVec; // vector of vector<string>
```
Stanford vs. STL: Part 1
/* Stanford C++ Version */

Vector<int> v = { 1, 3, 7 };

v += 271;

cout << v[0] << endl;
cout << v[v.size() - 1] << endl;

Vector<int> first = v.subList(0, 2);
Vector<int> last = v.subList(2);

v.remove(0);
/* Stanford C++ Version */
Vector<int> v = { 1, 3, 7 };

v += 271;
cout << v[0] << endl;
cout << v[v.size() - 1] << endl;
Vector<int> first = v.subList(0, 2);
Vector<int> last = v.subList(2);
v.remove(0);

/* Standard C++ Version */
std::vector<int> v = { 1, 3, 7 };
/* Stanford C++ Version */
Vector<int> v = { 1, 3, 7 };

v += 271;

cout << v[0] << endl;
cout << v[v.size() - 1] << endl;

Vector<int> first = v.subList(0, 2);
Vector<int> last = v.subList(2);
v.remove(0);

/* Standard C++ Version */
std::vector<int> v = { 1, 3, 7 };

v.push_back(271);
std::vector<int> v = { 1, 3, 7 };  
v += 271;  
cout << v[0] << endl;  
cout << v[v.size() - 1] << endl;  
Vector<int> first = v.subList(0, 2);  
Vector<int> last = v.subList(2);  
v.remove(0);  

std::vector<int> v = { 1, 3, 7 };  
v.push_back(271);  
cout << v.front() << endl;  
cout << v.back() << endl;
/* Stanford C++ Version */

Vector<int> v = {1, 3, 7};

v += 271;

cout << v[0] << endl;
cout << v[v.size() - 1] << endl;
Vector<int> first = v.subList(0, 2);
Vector<int> last = v.subList(2);
v.remove(0);

.isNull() No such thing as a sublist

/* Standard C++ Version */

std::vector<int> v = {1, 3, 7};

v.push_back(271);

cout << v.front() << endl;
cout << v.back() << endl;

// no such thing as a sublist
/* Stanford C++ Version */
Vector<int> v = { 1, 3, 7 };

v += 271;
cout << v[0] << endl;
cout << v[v.size() - 1] << endl;
Vector<int> first = v.subList(0, 2);
Vector<int> last = v.subList(2);
v.remove(0);

/* Standard C++ Version */
std::vector<int> v = { 1, 3, 7 };

v.push_back(271);
cout << v.front() << endl;
cout << v.back() << endl;

// no such thing as a sublist
v.erase(v.begin()); // or v.pop_back()
Stanford vs. STL: Part 2
Stanford vs. STL: Part 2

```cpp
/* Stanford C++ Version */
Vector<string> v = { "A", "B", "C" };

/* Counting for loop. */
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << endl;
}

/* Range-based for loop. */
for (string elem : v) {
    cout << elem << endl;
}
```
/* Stanford C++ Version */
Vector<string> v = { "A", "B", "C" };

/* Counting for loop. */
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << endl;
}

/* Range-based for loop. */
for (string elem : v) {
    cout << elem << endl;
}

/* Standard C++ Version */
std::vector<string> v = { "A", "B", "C" };
/* Stanford C++ Version */
Vector<string> v = { "A", "B", "C" };

/* Counting for loop. */
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << endl;
}

/* Range-based for loop. */
for (string elem : v) {
    cout << elem << endl;
}

/* Standard C++ Version */
std::vector<string> v = { "A", "B", "C" };

// Basically the same
for (size_t i = 0; i < v.size(); ++i) {
    cout << v[i] << endl;
}
/* Stanford C++ Version */
Vector<string> v = { "A", "B", "C" };

/* Counting for loop. */
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << endl;
}

/* Range-based for loop. */
for (string elem: v) {
    cout << elem << endl;
}

/* Standard C++ Version */
std::vector<string> v = { "A", "B", "C" };

// Basically the same
for (size_t i = 0; i < v.size(); ++i) {
    cout << v[i] << endl;
}

// The same
for (string elem: v) {
    cout << elem << endl;
}
Example

Standard C++ Vector in (Basic) Action
Why the Difference?

Why doesn't std::vector bounds check by default?

**Hint:** Remember our discussion of the philosophy of C++
Why the Difference?

Why doesn’t `std::vector` bounds check by default?

**Hint:** Remember our discussion of the philosophy of C++

If you write your program **correctly**, bounds checking will just **slow** your code down.
Play around with the `std::vector`!

## Summary of Stanford `Vector<T>` vs `std::vector<T>`

<table>
<thead>
<tr>
<th>What you want to do</th>
<th>Stanford <code>Vector&lt;int&gt;</code></th>
<th><code>std::vector&lt;int&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Create an empty vector</td>
<td><code>Vector&lt;int&gt; v;</code></td>
<td><code>vector&lt;int&gt; v;</code></td>
</tr>
<tr>
<td>Create a vector with n copies of zero</td>
<td><code>Vector&lt;int&gt; v(n);</code></td>
<td><code>vector&lt;int&gt; v(n);</code></td>
</tr>
<tr>
<td>Create a vector with n copies of a value k</td>
<td><code>Vector&lt;int&gt; v(n, k);</code></td>
<td><code>vector&lt;int&gt; v(n, k);</code></td>
</tr>
<tr>
<td>Add k to the end of the vector</td>
<td><code>v.add(k);</code></td>
<td><code>v.push_back(k);</code></td>
</tr>
<tr>
<td>Clear vector</td>
<td><code>v.clear();</code></td>
<td><code>v.clear();</code></td>
</tr>
<tr>
<td>Get the element at index i</td>
<td><code>int k = v.get(i);</code></td>
<td><code>int k = v.at(i);</code></td>
</tr>
<tr>
<td>(* Verify that i is in bounds!)</td>
<td><code>int k = v[i];</code></td>
<td>`int k = v[i]; (*)&amp;</td>
</tr>
<tr>
<td>Check if the vector is empty</td>
<td><code>if (v.isEmpty()) ...</code></td>
<td><code>if (v.empty()) ...</code></td>
</tr>
<tr>
<td>Replace the element at index i</td>
<td><code>v.get(i) = k;</code></td>
<td><code>v.at(i) = k;</code></td>
</tr>
<tr>
<td>(* Verify that i is in bounds!)</td>
<td><code>v[i] = k;</code></td>
<td>`v[i] = k; (*)&amp;</td>
</tr>
</tbody>
</table>
### One Important Similarity

<table>
<thead>
<tr>
<th>What you want to do</th>
<th>Stanford Vector&lt;int&gt;</th>
<th>std::vector&lt;int&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create an empty vector</td>
<td>Vector&lt;int&gt; v;</td>
<td>vector&lt;int&gt; v;</td>
</tr>
<tr>
<td>Create a vector with n copies of zero</td>
<td>Vector&lt;int&gt; v(n);</td>
<td>vector&lt;int&gt; v(n);</td>
</tr>
<tr>
<td>Create a vector with n copies of a value k</td>
<td>Vector&lt;int&gt; v(n, k);</td>
<td>vector&lt;int&gt; v(n, k);</td>
</tr>
<tr>
<td>Add k to the end of the vector</td>
<td>v.add(k);</td>
<td>v.push_back(k);</td>
</tr>
<tr>
<td>Clear vector</td>
<td>v.clear();</td>
<td>v.clear();</td>
</tr>
<tr>
<td>Get the element at index i (verify that i is in bounds)</td>
<td>int k = v.get(i); int k = v[i];</td>
<td>int k = v.at(i); int k = v[i];</td>
</tr>
<tr>
<td>Check if the vector is empty</td>
<td>if (v.isEmpty()) ...</td>
<td>if (v.empty()) ...</td>
</tr>
<tr>
<td>Replace the element at index i (verify that i is in bounds)</td>
<td>v.get(i) = k; v[i] = k;</td>
<td>v.at(i) = k; v[i] = k;</td>
</tr>
</tbody>
</table>
What you want to do | Stanford Vector<int> | std::vector<int>
---|---|---
Create an empty vector | Vector<int> v; | vector<int> v;
Create a vector with n copies of zero | Vector<int> v(n); | vector<int> v(n);
Create a vector with n copies of a value k | Vector<int> v(n, k); | vector<int> v(n, k);
Add k to the end of the vector | v.add(k); | v.push_back(k);
Clear vector | | |
Get the element at index i (verify that i is in bounds) | int k = v.get(i); | int k = v[i];
Check if the vector is empty | if (v.isEmpty()) ... | if (v.empty()) ... |
Replace the element at index i (verify that i is in bounds) | v.get(i) = k; | v[i] = k;
What if we had a `push_front()`?
What if we had a `push_front()`?

Suppose `push_front` existed and we used it.
Let’s look at a small vector:
What if we had a `push_front()`?

Suppose `push_front` existed and we used it.
Let’s look at a small vector:

```cpp
vec.push_front(7);
```

```
7
```

```
3 1 4 1 5
```

0th index
What if we had a `push_front()`?

Suppose `push_front` existed and we used it.
Let’s look at a small vector:

```cpp
vec.push_front(7);
```

Need to shift these elements up to make space in the 0th position.
What if we had a `push_front()`?

Suppose `push_front` existed and we used it. Let’s look at a small vector:

```
vec.push_front(7);
```

Need to shift these elements up to make space in the 0th position.
What if we had a `push_front()`?

Suppose `push_front` existed and we used it. Let’s look at a small vector:

```cpp
vec.push_front(7);
```

Need to shift these elements up to make space in the 0th position.
What if we had a `push_front()`?

Suppose `push_front` existed and we used it.

Let’s look at a small vector:

```cpp
vec.push_front(7);
```

Need to shift these elements up to make space in the 0th position.
What if we had a `push_front()`?

Suppose `push_front` existed and we used it.
Let’s look at a small vector:

```cpp
vec.push_front(7);
```

Need to shift these elements up to make space in the 0th position.
What if we had a **push_front()**?

Suppose **push_front** existed and we used it. Let’s look at a small vector:

```cpp
vec.push_front(7);
```

Need to shift these elements up to make space in the 0th position.
What if we had a `push_front()`?

Suppose `push_front` existed and we used it. Let’s look at a small vector:

```cpp
vec.push_front(7);
```

```
7
```

Need to shift these elements up to make space in the 0th position.
What if we had a `push_front()`?

Suppose `push_front` existed and we used it.
Let’s look at a small vector:

```c++
vec.push_front(7);
```

```
<table>
<thead>
<tr>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
```

Need to shift these elements up to make space in the 0th position.

0th index
What if we had a `push_front()`?

Suppose `push_front` existed and we used it.

Let’s look at a small vector:

```
vec.push_front(7);
```

Need to shift these elements up to make space in the 0th position.
What if we had a `push_front()`?

Suppose `push_front` existed and we used it. Let’s look at a small vector:

```cpp
vec.push_front(7);
```

![Diagram of vector with 0th index highlighted and elements shifted up](image)

Need to shift these elements up to make space in the 0th position.
What if we had a `push_front()`?

Suppose `push_front` existed and we used it. Let’s look at a small vector:

```
vec.push_front(7);
```

Need to shift these elements up to make space in the 0th position.
What if we had a `push_front()`?

Suppose `push_front` existed and we used it. Let’s look at a small vector:

```
vec.push_front(7);
```

Need to shift these elements up to make space in the 0th position.
What if we had a `push_front()`?

Suppose `push_front` existed and we used it. Let’s look at a small vector:

```cpp
vec.push_front(7);
```

Now we can insert the new element.
What if we had a `push_front()`?

Suppose `push_front` existed and we used it. Let’s look at a small vector:

```cpp
vec.push_front(7);
```

```
| 7 | 3 | 1 | 4 | 1 | 5 |
```

0th index
What if we had a `push_front()`?
Conclusion: `push_front()` is slow!

A vector is the prime tool of choice in most applications!

- Fast
- Lightweight
- Intuitive

However, we just saw vectors grow efficiently in only one direction.

Sometimes it is useful to be able to `push_front` quickly!

C++ has a solution!
std::deque<T>
Pronounced “deck”.
Stands for a double ended queue.

Does everything a vector can do
AND

Unlike a vector, it is possible (and fast) to push_front and pop_front!
Syntax of `std::deque<T>`

/* Standard C++ Version */
std::deque<int> d = { 1, 3, 7 };
d.push_back(271);
d.push_front(-1);
cout << d.front() << endl;
cout << d.back() << endl;
d.pop_back();
d.pop_front();

// d = {1, 3, 7}
// d = {1, 3, 7, 271}
// d = {-1, 1, 3, 7, 271}
// prints -1
// prints 271
// d = {-1, 1, 3, 7}
// d = {1, 3, 7}
Example

Vector vs. Deque: push_front
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

```
std::deque<int> deq;
deq.push_front(7);
```

Diagram:

```
     NULL
     |
  2 6 5 | 3 1
     |
     |
     4 1 5 9
```
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

```cpp
deq.push_front(7);
```

![Dequeue Diagram](image-url)
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

```cpp
deq.push_front(7);
```
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

```cpp
deq.push_back(3);
```
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

```cpp
deq.push_back(3);
```
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

```cpp
deq.push_back(3);
```
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

```cpp
deq.push_back(5);
```
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

```cpp
void push_back(T value) {  // Add value to the back of the deque
    std::deque<T>().push_back(value);
}
```
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

```
std::deque<int> deq;
deq.push_back(5);
```

Initial state:

```
4 1 5 9
```

After adding 5:

```
7 3 1
```

And after shifting:

```
2 6 5 3
```
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

```cpp
deq.push_back(5);
```

```
7 3 1
4 1 5 9
```

```
2 6 5 3
```
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:

```cpp
deq.push_back(5);
```
How does `std::deque<T>` work?

There is no single specific implementation of a deque, but one common one might look like this:
Wait a minute...
Question

If deque can do everything a vector can do and also has a fast push_front...

Why use a vector at all?
Downsides of `std::deque<T>`

Deques support fast `push_front` operations.

However, for other common operations like `element access`, vector will always outperform a deque.

Let’s see this in action!
Example

Vector vs. Deque: Element Access
Which to Use?

“vector is the type of sequence that should be used by default... deque is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.”

— C++ ISO Standard (section 23.1.1.2):
Container Adaptors
Container Adaptors

Recall stacks and queues:
Container Adaptors

Recall stacks and queues:

push

stack

5
13
41
12
Container Adaptors

Recall stacks and queues:

```
stack
| 5 |
| 13|
| 41|
| 12|
```
Container Adaptors

Recall stacks and queues:

![Stack diagram with numbers 12, 13, 41 and a pop action](image)
Container Adaptors

Recall stacks and queues:

![Stack Diagram]

- 13
- 41
- 12

stack
Container Adaptors

Recall stacks and queues:

- Stack:
  - 13
  - 41
  - 12

- Queue:
  - 9
  - 16
  - 11
  - Back
Container Adaptors

Recall stacks and queues:
Container Adaptors

Recall stacks and queues:

stack

queue

13
41
12

9
16
11
5

back
Container Adaptors

Recall stacks and queues:

- **Stack**
  - 13
  - 41
  - 12

- **Queue**
  - 16
  - 11
  - 5

- pop_front

- back
Container Adaptors

Recall stacks and queues:

stack

13
41
12

queue

16
11
5

back
Container Adaptors

How can we implement stack and queue using the containers we have?
Container Adaptors

How can we implement stack and queue using the containers we have?

Stack:

Just limit the functionality of a vector/deque to only allow `push_back` and `pop_back`.

Queue:

Just limit the functionality of a deque to only allow `push_back` and `pop_front`.

Plus only allow access to top element
Container Adaptors

For this reason, stacks and queues are known as container adaptors.

\[ \text{std::stack} \]

Defined in header `<stack>`

\[
\text{template}<
\quad \text{T,}
\quad \text{class Container = std::deque<T>}
\> \quad \text{class stack;}
\]

The `std::stack` class is a container adapter that gives the programmer the functionality of a stack - specifically, a FILO (first-in, last-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

**Template parameters**

- \( T \) - The type of the stored elements. The behavior is undefined if \( T \) is not the same type as `Container::value_type`. (Since C++11)

- `Container` - The type of the underlying container to use to store the elements. The container must satisfy the requirements of `SequenceContainer`. Additionally, it must provide the following functions with the usual semantics:
  - `back()`
  - `push_back()`
  - `pop_back()`

The standard containers `std::vector`, `std::deque` and `std::list` satisfy these requirements.

\[ \text{std::queue} \]

Defined in header `<queue>`

\[
\text{template}<
\quad \text{T,}
\quad \text{class Container = std::deque<T>}
\> \quad \text{class queue;}
\]

The `std::queue` class is a container adapter that gives the programmer the functionality of a queue - specifically, a FIFO (first-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The queue pushes the elements on the back of the underlying container and pops them from the front.

**Template parameters**

- \( T \) - The type of the stored elements. The behavior is undefined if \( T \) is not the same type as `Container::value_type`. (Since C++11)

- `Container` - The type of the underlying container to use to store the elements. The container must satisfy the requirements of `SequenceContainer`. Additionally, it must provide the following functions with the usual semantics:
  - `back()`
  - `front()`
  - `push_back()`
  - `pop_front()`

The standard containers `std::deque` and `std::list` satisfy these requirements.
Container Adaptors

For this reason, stacks and queues are known as container adaptors.

**std::stack**

Defined in header `<stack>`

```cpp
template
class T,
class Container = std::deque<T>
> class stack;
```

The `std::stack` class is a container adapter that gives the programmer the functionality of a stack - specifically, a FILO (first-in, last-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

**Template parameters**

- `T` - The type of the stored elements. The behavior is undefined if `T` is not the same type as `Container::value_type`, (since C++11)
- `Container` - The type of the underlying container to use to store the elements. The container must satisfy the requirements of `SequenceContainer`. Additionally, it must provide the following functions with the usual semantics:
  - `back()`
  - `push_back()`
  - `pop_back()`

The standard containers `std::vector`, `std::deque` and `std::list` satisfy these requirements.

**std::queue**

Defined in header `<queue>`

```cpp
template
class T,
class Container = std::deque<T>
> class queue;
```

The `std::queue` class is a container adapter that gives the programmer the functionality of a queue - specifically, a FIFO (first-in, first-out) data structure.

The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The queue pushes the elements on the back of the underlying container and pops them from the front.

**Template parameters**

- `T` - The type of the stored elements. The behavior is undefined if `T` is not the same type as `Container::value_type`, (since C++11)
- `Container` - The type of the underlying container to use to store the elements. The container must satisfy the requirements of `SequenceContainer`. Additionally, it must provide the following functions with the usual semantics:
  - `back()`
  - `front()`
  - `push_back()`
  - `pop_front()`

The standard containers `std::deque` and `std::list` satisfy these requirements.
Container Adaptors

For this reason, stacks and queues are known as **container adaptors**.

### std::stack

Defined in header `<stack>`

```cpp
template class T,
    class Container = std::deque<T>

> class stack;
```

The `std::stack` class is a container adapter that gives the programmer the functionality of a stack - specifically, a FILO (first-in, last-out) data structure. The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

**Template parameters**

- **T** - The type of the stored elements. The behavior is undefined if T is not the same type as `Container::value_type`

**Container** - The type of the underlying container to use to store the elements. The container must satisfy the requirements of `SequenceContainer`. Additionally, it must provide the following functions with the usual semantics:
  - `back()`
  - `push_back()`
  - `pop_back()`

The standard containers `std::vector`, `std::deque` and `std::list` satisfy these requirements.

### std::queue

Defined in header `<queue>`

```cpp
template class T,
    class Container = std::deque<T>

> class queue;
```

The `std::queue` class is a container adapter that gives the programmer the functionality of a queue - specifically, a FIFO (first-in, first-out) data structure. The class template acts as a wrapper to the underlying container - only a specific set of functions is provided. The queue pushes the elements on the back of the underlying container and pops them from the front.

**Template parameters**

- **T** - The type of the stored elements. The behavior is undefined if T is not the same type as `Container::value_type`

**Container** - The type of the underlying container to use to store the elements. The container must satisfy the requirements of `SequenceContainer`. Additionally, it must provide the following functions with the usual semantics:
  - `back()`
  - `front()`
  - `push_back()`
  - `pop_front()`

The standard containers `std::deque` and `std::list` satisfy these requirements.
Next time

Iterators and Associative Containers
Bonus Content...
Example

The Power of the C++ STL
const int kNumInts = 200;
std::vector<int> vec(kNumInts);
std::generate(vec.begin(), vec.end(), rand);
std::sort(vec.begin(), vec.end());
std::copy(vec.begin(), vec.end(),
    std::ostream_iterator<int>(cout, "\n");