Sequence Containers

Bad Dad Joke of the Day:
- Two dragons are in a bar. One dragon says, “It’s warm in here.”
  The other replies, “Shut your mouth.”

Creds: EW
Game Plan

- Finishing Up C++ Types
- Survey Results!
- Overview of STL
- Sequence Containers
- Container Adaptors
C++ Types (cont.)
structures
C++17 allows structured bindings, allowing you to unpack the variables in a pair.

```cpp
pair<int, int> findPriceRange(int dist) {
    int min = static_cast<int>(dist * 0.08 + 100);
    int max = static_cast<int>(dist * 0.36 + 750);
    return make_pair(min, max);
}

int main() {
    int dist = 6452;
    auto [min, max] = findPriceRange(dist);
    cout << "You can find prices between: " << min << " and " << max;
}
```

Notice the use of auto here – automatically infer the type of each variable.
C++17 allows structured bindings, allowing you to unpack the variables in a pair.

```cpp
pair<int, int> findPriceRange(int dist) {
    int min = static_cast<int>(dist * 0.08 + 100);
    int max = static_cast<int>(dist * 0.36 + 750);
    return make_pair(min, max);
}

int main() {
    int dist = 6452;
    auto [min, max] = findPriceRange(dist);
    cout << "You can find prices between: "
         << min << " and " << max;
}
```

What drawbacks of this approach can you see?
A struct is a collection of named variables grouped together.

```c
struct PriceRange {
    int min;
    int max;
}

struct Course {
    string code;
    Time startTime; Time endTime;
    vector<string> instructors;
}
```
A struct is a collection of named variables grouped together.

```
struct PriceRange {
    int min;
    int max;
}

struct Course {
    string code;
    Time startTime; Time endTime;
    vector<string> instructors;
}
```

Same as a pair<int, int>, but the ints are named.
Structs offer the benefit that the struct itself and the variables inside are named.

```cpp
PriceRange findPriceRange(int dist) {
  int min = static_cast<int>(dist * 0.08 + 100);
  int max = static_cast<int>(dist * 0.36 + 750);
  return PriceRange{min, max};
}

int main() {
  int dist = 6452;
  PriceRange p = findPriceRange(dist);
  cout << "You can find prices between: "
       << p.min << " and " << p.max;
}
```

This is very readable: result is a DatingRange, and you are printing its min and max.
Structs offer the benefit that the struct itself and the variables inside are named.

```cpp
PriceRange findPriceRange(int dist) {
    int min = static_cast<int>(dist * 0.08 + 100);
    int max = static_cast<int>(dist * 0.36 + 750);
    return PriceRange{min, max};
}

int main() {
    int dist = 6452;
    PriceRange p = findPriceRange(dist);
    cout << "You can find prices between: ";
    << p.min << " and " << p.max;
}
```

To access a member inside the struct, use the . notation.
You can also use structured bindings on structs.

```cpp
PriceRange findPriceRange(int dist) {
    int min = static_cast<int>(dist * 0.08 + 100);
    int max = static_cast<int>(dist * 0.36 + 750);
    return PriceRange{min, max};
}

int main() {
    int dist = 6452;
    auto [min, max] = findPriceRange(dist);
    cout << "You can find prices between: " << min << " and " << max;
}
```

The order the binding occurs is the same order as the variables are laid in the struct.
Structures are frequently used in practice.

```cpp
std::tuple<bool, Time, Time> findCourseTime([omitted]) {
    for (int i = 0; i < courseDatabase.size(); ++i) {
        if (courseCode == course.code) {
            return make_tuple(true, course.startTime, course.endTime);
        }
    }
    return make_tuple(false, Time{}, Time{});
}

Note: we’ll clean this up using uniform initialization later!
```
We want to go through all the Courses in the vector and add 1 to start and end time.

```cpp
void transformToDST(vector<Course>& courses) {
    for (auto& course : courses) {
        course.startTime.hour++;
        course.endTime.hour++;
    }
}
```

Remember: auto discards all qualifiers.
Can also use structured binding with references!

```cpp
void transformToDST(vector<Course> & courses) {
    for (auto & [code, start, end, instructors] : courses) {
        start++;  
        end++;  
    }
}
```

In each iteration, unpacks each member as a reference.
void transformToDST(const vector<Course>& courses) {
    for (auto& course : courses) {
        course.startTime.hour++;
        course.endTime.hour++;
    }
}
Use const on local variables that is not meant to change (especially references).

```cpp
void transformToDST(const vector<Course>& courses) {
    for (const auto& course : courses) {
        course.startTime.hour++;
        course.endTime.hour++;    
    }
}
```
initialization
In C++, depending on the type, there were too many ways to initialize a variable.
To solve this, C++11 (ironically) adds one more way: uniform initialization.

```cpp
int main() {
    vector<int> vec{3, 1, 4, 1, 5, 9};
    Course now {“CS106L”, {15, 30}, {16, 30},
                 {“Wang”, “Zeng”} };
}
```

We don’t have to specify the types – automatically deduced.
The return value can also be uniform initialized.

```cpp
pair<int, int> findPriceRange(int dist) {
    int min = static_cast<int>(dist * 0.08 + 100);
    int max = static_cast<int>(dist * 0.36 + 750);
    return {min, max};
}

int main() {
    int dist = 6452;
    auto [min, max] = findPriceRange(dist);
    cout << "You can find prices between: "
         << min << " and " << max;
}
```
A initializer list is a lightweight vector that can be used as a parameter.

```cpp
vector::vector(initializer_list<T> init);
```

Constructor creates a vector with initial elements.

```cpp
vector<int> vec{3, 1, 4, 1, 5, 9};
```
Using the uniform initialization syntax, the initializer list ctor is preferred over constructor.

```cpp
int main() {
    vector<int> vec1{3}; // vector = {3}
    vector<int> vec2(3); // vector = {0, 0, 0}
}
```

First one calls ctor with initializer list, second calls constructor with one parameter.
When should I use a stringstream?

• Processing strings
  • Simplify “./a/b/..” to “/a”

• Formatting input/output
  • uppercase, hex, and other stream manipulators

• Parsing different types
  • stringToInteger() from previous lectures
Caution: Use Thoughtfully

When should I use a stringstream?

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

If you’re just concatenating strings, strappend() is faster than using a stringstream!
Survey Results!
Majors:
- Computer Science
- Aero/Astro
- Linguistics
- Materials Science
- Mechanical Engineering
- Public Policy
Majors:
- Computer Science
- Aero/Astro
- Linguistics
- Materials Science
- Mechanical Engineering
- Public Policy

Why you’re here:
- Industry usages
- C++ practice
- Supplement CS 106B/X
- Specific applications
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

Adaptors
Overview of STL

- Algorithms
- Iterators
- Containers
- Functors/Lambdas
- Adaptors
Overview of STL

- Algorithms
- Iterators
- Containers
- Functors/Lambdas
- Adaptors
- And more!
Example
The Power of the C++ STL
Here is a program that generates a vector with random entries, sorts it, and prints it, all in one go!

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"));
“Another benefit of striving for efficiency is that the process forces you to understand the problem in more depth.”

— also Alex Stepanov, inventor of the STL
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<std::string> vecStr;  // vector of string
std::vector<myStruct> vecStruct;  // vector of myStructs
std::vector<std::vector<std::string>> vecOfVec;  // vector of
                               // vector of
                               // vector<string>
```
# Summary of Stanford `Vector<T>` vs `std::vector<T>`

<table>
<thead>
<tr>
<th>What you want to do</th>
<th>Stanford Vector&lt;int&gt;</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 (verify that i is in bounds)</td>
<td><code>int k = v.get(i);</code></td>
<td><code>int k = v.at(i);</code></td>
</tr>
<tr>
<td></td>
<td><code>int k = v[i];</code></td>
<td><code>int k = v[i];</code></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 (verify that i is in bounds)</td>
<td><code>v.get(i) = k;</code></td>
<td><code>v.at(i) = k;</code></td>
</tr>
<tr>
<td></td>
<td><code>v[i] = k;</code></td>
<td><code>v[i] = k;</code></td>
</tr>
</tbody>
</table>
## 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 (verify that i is in bounds)</td>
<td><code>int k = v.get(i);</code></td>
<td><code>int k = v.at(i);</code></td>
</tr>
<tr>
<td></td>
<td><code>int k = v[i];</code></td>
<td><code>int k = v[i];</code></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 (verify that i is in bounds)</td>
<td><code>v.get(i) = k;</code></td>
<td><code>v.at(i) = k;</code></td>
</tr>
<tr>
<td></td>
<td><code>v[i] = k;</code></td>
<td><code>v[i] = k;</code></td>
</tr>
</tbody>
</table>
### Summary of Stanford Vector\(<T>\) vs `std::vector<T>`

<table>
<thead>
<tr>
<th>What you want to do</th>
<th>Stanford Vector(&lt;\text{int}&gt;)</th>
<th><code>std::vector&lt;\text{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) (verify that (i) is in bounds)</td>
<td><code>int k = v.get(i);</code></td>
<td><code>int k = v.at(i);</code></td>
</tr>
<tr>
<td></td>
<td><code>int k = v[i];</code></td>
<td><code>int k = v[i];</code></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) (verify that (i) is in bounds)</td>
<td><code>v.get(i) = k;</code></td>
<td><code>v.at(i) = k;</code></td>
</tr>
<tr>
<td></td>
<td><code>v[i] = k;</code></td>
<td><code>v[i] = k;</code></td>
</tr>
</tbody>
</table>
## Summary of Stanford Vector<T> vs std::vector<T>

<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>
But there is one key difference!
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.
## 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</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bounds)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Check</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replace the element at index i (verify that i is</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bounds)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What happens if we try to add an element to the **beginning** of a vector?
Why is \texttt{push\_front} slow?
Why is `push_front` slow?

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

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

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

0th index
Why is `push_front` slow?

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.
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.
Suppose \texttt{push\_front} existed and we used it.

Let’s look at a small vector:

\begin{verbatim}
vec.push_front(7);
\end{verbatim}

Why is \texttt{push\_front} slow?

Need to shift these elements up to make space in the 0th position.
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.
Why is `push_front` slow?

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.
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.

0th index
Why is `push_front` slow?

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.
Why is `push_front` slow?

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.
Why is `push_front` slow?

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

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

Why is `push_front` slow?

Need to shift these elements up to make space in the 0th position.
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.
Why is `push_front` slow?

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.
Why is `push_front` slow?

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

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

0th index
Why is \texttt{push\_front} slow?
A vector is the **prime** tool of choice in most applications!

- Fast
- Lightweight
- Intuitive

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

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

C++ has a solution!
std::deque<T>
A deque (pronounced “deck”) is a **double ended queue**. Can do **everything** a vector can do

and also...

Unlike a vector, it is possible (and **fast**) to **push_front** and **pop_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:

```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:

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

```
3 1

NULL

2 6 5

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);
```

![Diagram of deque operations]
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:

```
    7  3  1
       NULL

        2  6  5
```

```
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
std::deque<int> deq;

// Push back an element
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
deq.push_back(5);
```

```plaintext
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);
```

```
7 3 1
```

```
2 6 5 3
```

```
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_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
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: Timing Version

(you’ll use a similar concept with Assignment 1 as well!)
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
Recall stacks and queues:

Container Adaptors
Recall stacks and queues:

Container Adaptors

push

stack

5
13
41
12
Recall stacks and queues:

**Container Adaptors**

stack

- 5
- 13
- 41
- 12
Container Adaptors

Recall stacks and queues:
Recall stacks and queues:

Container Adaptors
Recall stacks and queues:

Stack:
- 13
- 41
- 12

Queue:
- 9
- 16
- 11
Container Adaptors

Recall stacks and queues:

![Diagram showing stack and queue with elements: Stack elements are 12, 41, 13; Queue elements are 9, 16, 11, 5. The stack has a push_back operation. Back is indicated at the end of the queue.](image-url)
Container Adaptors

Recall stacks and queues:

Stack:
- 13
- 41
- 12

Queue:
- 9
- 16
- 11
- 5

(back)
Recall stacks and queues:
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?
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.

**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.
Container Adaptors

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

**std::stack**

Defined in header `<stack>`

```
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++17)
- `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>`

```
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++17)
- `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

```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

```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.
Preview of Assignment 1

due Thursday, October 17
overflow
file streams
ifstream file(filename); // open
string line;

while (getline(file, line)) {
  // do something with line
}

file.close(); // technically don’t need this
Review: read file token by token

```cpp
ifstream file(filename); // open
string token;

while (file >> token) {
    // do something with line
}

file.close(); // technically don’t need this
```
Example

implementing Axess, take 2
manipulators +
overloading >> or <<
There are some keywords that will change the behavior of the stream when inserted.

- **endl**
  - insert newline and flush stream

- **ws**
  - skips all whitespace until it finds another char

- **boolalpha**
  - prints “true” and “false” for bools.

- **hex**
  - prints numbers in hexadecimal

- **setprecision**
  - adjusts the precision of printed numbers
We can use manipulators to pad the output.

```cpp
#include <iostream>
#include <iomanip>

int main() {
    std::cout << "[" << std::setw(10) << "Ito" << "]" << std::endl;
    return 0;
}
```

Output:

```
[        Hi]
```

```cpp
#include <iostream>
#include <iomanip>

int main() {
    std::cout << "[" << std::left << std::setw(10) << "Ito" << "]" << std::endl;
    return 0;
}
```

Output:

```
[Hi        ]
```

```cpp
#include <iostream>
#include <iomanip>

int main() {
    std::cout << "[" << std::left << std::setfill('-') << std::setw(10) << "Ito" << "]" << std::endl;
    return 0;
}
```

Output:

```
[Hi--------]
```
C++ does not know how to convert a custom structure to/from a string.

A Date struct in your program

```
int month
int day
```

string representation

"Sep. 26"

read/write

console
You have to specify how the conversion to/from a string occurs.

```cpp
ostream& operator<<(ostream& os, const Time& time) {
    os << time.hour << ":" << setfill('0') << setw(2) << time.minute;
    return os;
}
```
Other programming languages also allow you to turn an object into a string.

// java
String toString() {
    // create string from this object
}

// python
def __str__(self):
    // create string from self
You have to specify how the conversion to/from a string occurs.

```cpp
ostream& operator<<(ostream& os, const Time& time) {
    os << time.hour << ":";
    os << setfill('0') << setw(2) << time.minute;
    return os;
}
```

[ostream object] << [Time struct];

The header literally means: how should this be interpreted.
Make sure to respect the return value of the `<<` operator.

```cpp
ostream& operator<<(ostream& os, const Time& time) {
    os << time.hour << ":" << setfill('0') << setw(2) << time.minute;
    return os;
}
```

The return value is what allows us to chain the `<<`'s.

```cpp
[ostream] << [Time] << [Time];
```
You have to specify how the conversion to/from a string occurs.

```cpp
ostream& operator<<(ostream& os, const Time& time) {
    os << time.hour << ":" << setfill('0') << setw(2) << time.minute;
    return os;
}
```

“13:04”

string representation

type conversion

A Time struct in your program

<table>
<thead>
<tr>
<th>int hour</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>int minute</td>
<td>4</td>
</tr>
</tbody>
</table>
Manipulators can help you format the string exactly as you want it.

```cpp
ostream& operator<<(ostream& os, const Time& time) {
    os << time.hour << ":";
    os << setfill('0') << setw(2) << time.minute;
    return os;
}
```

"13:04"

string representation

type conversion

A Time struct in your program

```
int hour = 13;
int minute = 4;
```
Example

implementing Axess, stream >> overload
Reading input is harder because you have to worry about error-checking.

```
ostream& operator>>(istream& is, Time& time) {
    ???
    return is;
}
```

“13:04”

string representation  →  type conversion

int hour 13
int minute 4

A Time struct in your program

3 October 2019
Be consistent with other input stream operations!

1. If the fail bit is on, do not do anything.

2. You can only read one token, nothing more, nothing less. (probably a good idea to copy the token into a stringstream).

3. If the operation failed, set the fail bit. Make sure the original stream and object are unchanged!
Be consistent with other input stream operations!

1. If the fail bit is on, do not do anything.

```cpp
istream& operator>>(istream& is, Time& time) {
    if (!is) return is; // if fail do nothing
    // continued
}
```
Be consistent with other input stream operations!

2. You can only read one token, nothing more, nothing less. (probably a good idea to copy the token into a stringstream).

```cpp
string timeString;
if (!(is >> timeString)) { // read exactly one token
    is.setstate(ios::failbit);
    return is;
}
istringstream ss(timeString);
// use ss;
```
Be consistent with other input stream operations!

3. If the operation failed, set the fail bit. Make sure the original stream and object are unchanged!

```cpp
int hour, minute; char colon;
if (/* parse hour, minute, colon using >> */) {
    time = Time{hour, minute};
} else { // we didn’t change time
    is.setstate(ios::failbit);
}
```
Example

implementing Axess, take 2