Sequential Containers

Ali Malik
malikali@stanford.edu
Game Plan

Recap
Overview of STL
Sequence Containers
std::vector
std::deque
Container Adapters
Announcements
Recap
getLine issue
I am using the function getLine from simpio library

string line = getLine("a)nimate, t)ick, q)uit? ");

When I run the code it will print the prompt but it will not stop to take an input from the console. I can't figure out why this might be, any thoughts?

the students' answer, where students collectively construct a single answer

I have a similar problem of the prompt printing out twice when I use both "getline()" and "cin". Should we stick to using only "cin" or only "getline()" throughout our code?

the instructors' answer, where instructors collectively construct a single answer

If you use both getline() and cin, the problem y'all have will occur. To fix this, just stick to only using the Stanford library functions for input (the ones in the simpio library might not work properly).

~ An instructor (Marty Stepp) endorsed this answer ~
getline vs >> Bug

What happens if we mix cin and getline?

position

favNum =
fullName =

1 7 2 9 \n
What happens if we mix cin and getline?

```
int favNum;
cin >> favNum;
```
What happens if we mix cin and getline?

```cpp
int favNum;
cin >> favNum;
```

```
favNum =
fullName =
```
What happens if we mix cin and getline?

```
position
```

```
favNum   = 1729
fullName =
```
What happens if we mix cin and getline?

```cpp
string fullName;
getline(cin, fullName);
```
What happens if we mix cin and getline?

```cpp
string fullName;
getline(cin, fullName);
```

```
1 7 2 9 \n
position
```
What happens if we mix cin and getline?

favNum = 1729
fullName = ""

1 7 2 9 \n
position
getline vs >> Bug

We can fix this by consuming the newline character after using cin!

```cpp
int favNum;
cin >> favNum;

cin >> ws;  // extracts as many whitespace chars as possible
            // from current position in stream

string fullName;
getline(cin, fullName);  // no residual newline char in stream
```
Useful Aside
You can define your own mini-types that bundle multiple variables together:

```c
struct point {
    int x;
    int y;
};
```

Useful for Assignment 1
struct point {
    int x;
    int y;
};

point p;
p.x = 12;
p.y = 15;
Overview of 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

Allocators

Containers

Iterators

Algorithms

Functors/Lambdas

Adapters
Overview of STL

- Allocators
- Containers
- Iterators
- Algorithms
- Functors/Lambdas
- Adapters
Where we are going...

Here is a program that generates a vector with random entries, sorts it, and prints it, all in one go!

```cpp
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"));
```
Sequence Containers
Sequence Containers

Provides access to sequences of elements.

Examples:

- `std::vector<T>`
- `std::list<T>`
- `std::deque<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::vector<std::vector<std::string>>> vecOfVec; // vector of vector of vector of strings
```
Summary of `std::vector<T>` vs Stanford `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></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></td>
</tr>
</tbody>
</table>
Summary of `std::vector<T>` vs Stanford `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></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></td>
</tr>
</tbody>
</table>
### Summary of `std::vector<T>` vs Stanford `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></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></td>
</tr>
</tbody>
</table>
### Summary of `std::vector<T>` vs Stanford `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>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></td>
</tr>
</tbody>
</table>
## Some Differences - `std::vector<T>` vs Stanford `Vector<T>`

<table>
<thead>
<tr>
<th>Operation</th>
<th><code>std::vector&lt;T&gt;</code></th>
<th><code>Vector&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Get the element at index <em>i</em> without bounds checking</td>
<td><code>int a = x[i];</code></td>
<td><code>// Impossible!</code></td>
</tr>
<tr>
<td>Change the element at index <em>i</em> without bounds checking</td>
<td><code>x[i] = v;</code></td>
<td><code>// Impossible!</code></td>
</tr>
<tr>
<td>Add an element to the beginning of a vector</td>
<td><code>// Impossible! (or at least slow)</code></td>
<td><code>// Impossible! (or at least slow)</code></td>
</tr>
<tr>
<td>Apply a function to each element in <em>x</em></td>
<td><code>x.mapAll(fn)</code></td>
<td><code>// We'll talk about this in another lecture...</code></td>
</tr>
<tr>
<td>Concatenate vectors <em>v1</em> and <em>v2</em></td>
<td><code>v1 += v2;</code></td>
<td><code>// We'll talk about this in another lecture...</code></td>
</tr>
</tbody>
</table>
std::vector<T>

Problem:

Write a program that reads a list of integers and finds the median.

Vector Median

(VecMedian.pro)
Some new stuff there:

```cpp
const int kNumInts = 5;
using vecsz_t = std::vector<int>::size_type;
std::sort(vec.begin(), vec.end());
```
Some new stuff there:

```cpp
const int kNumInts = 5;
using vecsz_t = std::vector<int>::size_type;
std::sort(vec.begin(), vec.end());
```

This is a promise to the compiler that this variable won’t change.
Some new stuff there:

```cpp
const int kNumInts = 5;

using vecsz_t = std::vector<int>::size_type;

std::sort(vec.begin(), vec.end());
```

This let's us use `vecsz_t` as an alias/synonym for the type `std::vector<int>::size_type;`
std::vector<T>

Some new stuff there:

```cpp
const int kNumInts = 5;

using vecsz_t = std::vector<int>::size_type;

std::sort(vec.begin(), vec.end());

This takes a range of the vector and sorts it
```
Some Differences - `std::vector<T>` vs Stanford `Vector<T>`

<table>
<thead>
<tr>
<th>Operation</th>
<th>C++ Code</th>
<th>Java Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get the element at index i without bounds checking</td>
<td><code>// Impossible!</code></td>
<td><code>int a = x[i];</code></td>
</tr>
<tr>
<td>Change the element at index i without bounds checking</td>
<td><code>// Impossible!</code></td>
<td><code>x[i] = v;</code></td>
</tr>
<tr>
<td>Add an element to the beginning of a vector</td>
<td><code>// Impossible! (or at least slow)</code></td>
<td><code>// Impossible! (or at least slow)</code></td>
</tr>
<tr>
<td>Apply a function to each element in x</td>
<td><code>x.mapAll(fn)</code></td>
<td><code>// We'll talk about this in another lecture...</code></td>
</tr>
<tr>
<td>Concatenate vectors v1 and v2</td>
<td><code>v1 += v2;</code></td>
<td><code>// We'll talk about this in another lecture...</code></td>
</tr>
</tbody>
</table>
Some Differences - `std::vector<T>` vs Stanford `Vector<T>`

<table>
<thead>
<tr>
<th>Operation</th>
<th><code>std::vector&lt;T&gt;</code></th>
<th><code>Vector&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Get the element at index i without bounds checking</td>
<td><code>int a = x[i];</code></td>
<td><code>x[i] = v;</code></td>
</tr>
<tr>
<td>Change the element at index i without bounds checking</td>
<td><code>// Impossible!</code></td>
<td><code>// Impossible!</code></td>
</tr>
<tr>
<td>Add an element to the beginning of a vector</td>
<td><code>// Impossible! (or at least slow)</code></td>
<td><code>// Impossible! (or at least slow)</code></td>
</tr>
<tr>
<td>Apply a function to each element in x</td>
<td><code>x.mapAll(fn)</code></td>
<td><code>// We'll talk about this in another lecture...</code></td>
</tr>
<tr>
<td>Concatenate vectors v1 and v2</td>
<td><code>v1 += v2;</code></td>
<td><code>// We'll talk about this in another lecture...</code></td>
</tr>
</tbody>
</table>

Why these differences?
### Some Differences - `std::vector<T>` vs Stanford `Vector<T>`

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result/Code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get the element at index <code>i</code> without bounds checking</td>
<td><code>// Impossible!</code></td>
<td><code>int a = x[i];</code></td>
</tr>
<tr>
<td>Change the element at index <code>i</code> without bounds checking</td>
<td><code>// Impossible!</code></td>
<td><code>x[i] = v;</code></td>
</tr>
<tr>
<td>Add an element to the beginning of a vector</td>
<td><code>// Impossible! (or at least slow)</code></td>
<td><code>// Impossible! (or at least slow)</code></td>
</tr>
<tr>
<td>Apply a function to each element in <code>x</code></td>
<td><code>x.mapAll(fn)</code></td>
<td><code>// We'll talk about this in another lecture...</code></td>
</tr>
<tr>
<td>Concatenate vectors <code>v1</code> and <code>v2</code></td>
<td><code>v1 += v2;</code></td>
<td><code>// We'll talk about this in another lecture...</code></td>
</tr>
</tbody>
</table>

**Why these differences?**
Some Differences - `std::vector<T>` vs Stanford Vector<T>

<table>
<thead>
<tr>
<th>Operation</th>
<th><code>std::vector&lt;T&gt;</code></th>
<th>Stanford Vector&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get the element at index i without bounds checking</td>
<td><code>int a = x[i];</code></td>
<td><code>x[i] = v;</code></td>
</tr>
<tr>
<td>Change the element at index i without bounds checking</td>
<td><code>// Impossible!</code></td>
<td><code>// Impossible!</code></td>
</tr>
<tr>
<td>Add an element to the beginning of a vector</td>
<td><code>// Impossible! (or at least slow)</code></td>
<td><code>// Impossible! (or at least slow)</code></td>
</tr>
<tr>
<td>Apply a function to each element in x</td>
<td><code>x.mapAll(fn)</code></td>
<td><code>// We'll talk about this in another lecture...</code></td>
</tr>
<tr>
<td>Concatenate vectors v1 and v2</td>
<td><code>v1 += v2;</code></td>
<td><code>// We'll talk about this in another lecture...</code></td>
</tr>
</tbody>
</table>

Why these differences?
Why the Differences?

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.
**Some Differences - `std::vector<T>` vs Stanford `Vector<T>`**

<table>
<thead>
<tr>
<th>Operation</th>
<th><code>std::vector&lt;T&gt;</code></th>
<th><code>Vector&lt;T&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Get the element at index i</td>
<td><code>int a = x[i];</code></td>
<td><code>// Impossible!</code></td>
</tr>
<tr>
<td>Change the element at index i</td>
<td><code>x[i] = v;</code></td>
<td><code>// Impossible!</code></td>
</tr>
<tr>
<td>Add an element to the beginning of a vector</td>
<td><code>// Impossible! (or at least slow)</code></td>
<td><code>// Impossible! (or at least slow)</code></td>
</tr>
<tr>
<td>Apply a function to each element in x</td>
<td><code>x.mapAll(fn)</code></td>
<td><code>// We'll talk about this in another lecture...</code></td>
</tr>
<tr>
<td>Concatenate vectors v1 and v2</td>
<td><code>v1 += v2;</code></td>
<td><code>// We'll talk about this in another lecture...</code></td>
</tr>
</tbody>
</table>

**Why these differences?**
**Some Differences** - `std::vector<T>` vs Stanford `Vector<T>`

<table>
<thead>
<tr>
<th>Operation</th>
<th><code>std::vector&lt;T&gt;</code> vs <code>Vector&lt;T&gt;</code></th>
<th>Why these differences?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get the element at index i without bounds checking</td>
<td><code>// Impossible!</code></td>
<td><code>int a = x[i];</code></td>
</tr>
<tr>
<td>Change the element at index i without bounds checking</td>
<td><code>// Impossible!</code></td>
<td><code>x[i] = v;</code></td>
</tr>
<tr>
<td>Add an element to the beginning of a vector</td>
<td><code>// Impossible! (or at least slow)</code></td>
<td><code>// Impossible! (or at least slow)</code></td>
</tr>
<tr>
<td>Apply a function to each element in x</td>
<td><code>x.mapAll(fn)</code></td>
<td><code>// We'll talk about this in another lecture...</code></td>
</tr>
<tr>
<td>Concatenate vectors v1 and v2</td>
<td><code>v1 += v2;</code></td>
<td><code>// We'll talk about this in another lecture...</code></td>
</tr>
</tbody>
</table>
Why is \texttt{push\_front} slow?

Requires shifting over of the other elements in the vector down one by one (\texttt{bad}).

\textbf{Illustration}: Say we have a small vector
Why is `push_front` slow?

Suppose `push_front` existed and we used it.

| 3 | 1 | 4 | 1 | 5 |

0th index
Why is `push_front` slow?

Suppose `push_front` existed and we used it.

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

Initial state:

```
3 1 4 1 5
```

After `push_front(7)`:

```
7 3 1 4 1 5
```
Why is `push_front` slow?

Suppose `push_front` existed and we used it.

```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

```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:

```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

```
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

```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

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

Need to shift these elements up to make space in the 0th position.
Why is \texttt{push\_front} slow?

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

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

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

```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:

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

Need to shift these elements up to make space in the 0th position.
Why is \texttt{push\_front} slow?

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

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

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:

```
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

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

![Diagram showing vector and push_front operation]
Why is `push_front` slow?

Suppose `push_front` existed and we used it:

```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

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

- Vector contents:
  - 0th index: 7
  - 1st index: 3
  - 2nd index: 1
  - 3rd index: 4
  - 4th index: 1
  - 5th index: 5
Why is `push_front` slow?
Why is `push_front` slow?

Let’s get a sense of the difference:

Insertion Speed

(InsertionSpeed.pro)
Why is `push_front` slow?

The results:
Why is `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

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

We can see the efficiency of `push_front` with a `std::deque`

Deque Speed

(DequeSpeed.pro)
`std::deque<T>`

The results:
The results:

std::deque<T>

Same scale as previous graph
The results:

```cpp
std::deque<T>
```

There are the lines!
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:

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

```c++
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:

```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);
```

```
7 3 1
```

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

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

```
std::deque<T>
```

deq.push_back(3);

![Diagram of deque operations]

1. `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 \texttt{std::deque<\text{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:

```
7 3 1
```

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

```
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:

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

![Diagram showing deque operations and data structure](image-url)
How does \texttt{std::deque<T>} work?

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

\begin{verbatim}
deq.push_back(5);
\end{verbatim}

\begin{center}
\begin{tikzpicture}
\node[anchor=center] at (0,0) {
\begin{tabular}{c}
7 & 3 & 1 \\
4 & 1 & 5 & 9 \\
2 & 6 & 5 & 3 & 5
\end{tabular}
};
\end{tikzpicture}
\end{center}
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.

Vector vs Deque

(VecDeqSpeed.pro)
Downsides of `std::deque<T>`

The results:
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):
Questions
Container Adapters
Container Adapters

Recall stacks and queues:

```
stack
13
41
12
```
Container Adapters

Recall stacks and queues:

stack

push

5
13
41
12
Container Adapters

Recall stacks and queues:

stack

5
13
41
12
Container Adapters

Recall stacks and queues:

13
41
12

stack

pop
Container Adapters

Recall stacks and queues:

![Stack Diagram]

- 13
- 41
- 12

**stack**
Recall stacks and queues:

**Stack**
- 13
- 41
- 12

**Queue**
- 9
- 16
- 11
- back
Container Adapters

Recall stacks and queues:
Container Adapters

Recall stacks and queues:

- Stack contains: 13, 41, 12
- Queue contains: 9, 16, 11, 5 (back)
Container Adapters

Recall stacks and queues:

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

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

- `pop_front`
- `back`
Container Adapters

Recall stacks and queues:
Container Adapters

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 Adapters

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

---

**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++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>`

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:
  - `front()`
  - `push_back()`
  - `pop_front()`

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

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

**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++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>`

```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++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 Adapters

For this reason, stacks and queues are known as container adapters.
Next Time

Associative Containers and Iterators