Bad Dad Joke of the Day:
- How do brass players fix their instruments?
- Using a tuba glue!

Creds: Sonia
Game Plan

- Container Adaptors
- Assignment 1
- Associative Containers
- Iterators
- Map iterators
Brief Recap
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>`
Sequence Containers

`std::vector<T>`

- `vec.at(i)` throws an exception
- `vec[i]` causes undefined behavior!

We saw this! In practice, `vec[i]` on an out-of-bounds index fails silently on Windows, and continues as though nothing happened on Mac!
Sequence Containers

`std::vector<T>`

- `vec.at(i)` throws an exception
- `vec[i]` causes undefined behavior!

`std::deque<T>`

- Everything a vector can do + `push_front`
- Slower to access middle elements, however
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):
### Which to Use?

#### Sequence containers in the standard library

<table>
<thead>
<tr>
<th>Container</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>basic_string</code></td>
<td>stores and manipulates sequences of characters</td>
</tr>
<tr>
<td><code>array</code> (C++11)</td>
<td>static contiguous array</td>
</tr>
<tr>
<td><code>vector</code></td>
<td>dynamic contiguous array</td>
</tr>
<tr>
<td><code>deque</code></td>
<td>double-ended queue</td>
</tr>
<tr>
<td><code>forward_list</code> (C++11)</td>
<td>singly-linked list</td>
</tr>
<tr>
<td><code>list</code></td>
<td>doubly-linked list</td>
</tr>
</tbody>
</table>

#### Trade-offs / usage notes

<table>
<thead>
<tr>
<th>Container</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>std::array</code></td>
<td>Fast access but fixed number of elements</td>
</tr>
<tr>
<td><code>std::vector</code></td>
<td>Fast access but mostly inefficient insertions/deletions</td>
</tr>
<tr>
<td><code>std::list</code></td>
<td>Efficient insertion/deletion in the middle of the sequence</td>
</tr>
<tr>
<td><code>std::forward_list</code></td>
<td>Efficient insertion/deletion at the beginning and at the end of the sequence</td>
</tr>
</tbody>
</table>

Container Adaptors
Container Adaptors

Recall stacks and queues:

stack

13
41
12
Recall stacks and queues:
Recall stacks and queues:

Container Adaptors

stack

5
13
41
12
Container Adaptors

Recall stacks and queues:

```
12
41
13
```

(pop)
Container Adaptors

Recall stacks and queues:
Recall stacks and queues:
Recall stacks and queues:

Container Adaptors

Stack:
- 13
- 41
- 12

Queue:
- 9
- 16
- 11
- 5

enqueue
Container Adaptors

Recall stacks and queues:

Stack:
- 13
- 41
- 12

Queue:
- 9
- 16
- 11
- 5

enqueue
Recall stacks and queues:
Container Adaptors

Recall stacks and queues:

stack
13
41
12

queue
9
16
11
5
back
Recall stacks and queues:

Container Adaptors

stack

<table>
<thead>
<tr>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
</tr>
<tr>
<td>12</td>
</tr>
</tbody>
</table>

queue

<table>
<thead>
<tr>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

dequeue

back
Recall stacks and queues:

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

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

**dequeue**

**back**
Recall stacks and queues:

Container Adaptors

stack

13
41
12

queue

16
11
5

back

pop
Recall stacks and queues:
Container Adaptors

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

Hint: What containers have the functions,

push_back          push_front
pop_back           pop_front

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

Step 1:

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`.

Step 2:
Only allow access to the “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`. (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 stack;
```

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.

```
```
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.

```
```
Why not just use a vector/deque?

Design philosophy of C++:

- Allow the programmer full control, responsibility, and choice if they want it.
- Express ideas and intent directly in code.
- Enforce safety at compile time whenever possible.
- Do not waste time or space.
- Compartmentalize messy constructs.
Why not just use a vector/deque?

Design philosophy of C++:

- Allow the programmer full control, responsibility, and choice if they want it.
- Express ideas and intent directly in code.
- Enforce safety at compile time whenever possible.
- Do not waste time or space.
- Compartmentalize messy constructs.
Associative Containers
Associated Containers

Have no idea of a sequence.

Data is accessed using a key instead of an index.

Includes:

- `std::map<T1, T2>`
- `std::set<T>`
- `std::unordered_map<T1, T2>`
- `std::unordered_set<T>`
Associative Containers

Have no idea of a sequence.

Data is accessed using a key instead of an index.

Includes:

- `std::map<T1, T2>`
- `std::set<T>`
- `std::unordered_map<T1, T2>`
- `std::unordered_set<T>`

Preview:
- Based on ordering property of keys.
- Keys need to be comparable using `<` (less than) operator.
Associative Containers

Have no idea of a sequence.

Data is accessed using a key instead of an index.

Includes:

- `std::map<T1, T2>`
- `std::set<T>`
- `std::unordered_map<T1, T2>`
- `std::unordered_set<T>`

Preview:
Based on hash function. You need to define how the key can be hashed.
Associative Containers

Have no idea of a sequence.

Data is accessed using a key instead of an index.

Includes:

- `std::map<T1, T2>`
- `std::set<T>`
- `std::unordered_map<T1, T2>`
- `std::unordered_set<T>`

Preview:
You can define `<` and hash function operators for your own classes!
Which to Use?

`std::map<T1, T2>` vs. `std::unordered_map<T1, T2>`

`std::set<T>` vs. `std::unordered_set<T>`

Map/set:
- iterates and prints in sorted order (of keys)
- faster to iterate through a range of elements

Unordered map/set:
- faster to access individual elements by key
Example
Standard C++ Maps
\texttt{std::map\langle T1, T2 \rangle}

Methods mostly same as Stanford map.

See [documentation](#) for full list of methods.
Methods mostly same as Stanford map.

See documentation for full list of methods.

Key Takeaways:

- `mymap.at(key)` vs. `mymap[key]`
Methods mostly same as Stanford map.

See documentation for full list of methods.

Key Takeaways:

- `mymap.at(key)` vs. `mymap[key]`
- Stanford’s `map.containsKey(key)` doesn’t exist (yet)!
  - Instead, use `mymap.count(key)`
  - Preview: there’s a (slightly faster) alternative that we’ll learn next lecture!
std::set<T>

Methods mostly same as Stanford set.

See documentation for full list of methods.

Key Takeaways:

- A set is just a specific case of a map that doesn’t have a value!
  - Or you can think of the value as being true (if present) or false
- Literally all the same functions as the C++ map, minus element access ([] and .at())
Example

Standard C++ Sets
Announcements
Announcements

- **Office hours for Assignment 1:**
  - Before lectures, by appointment
  - After lectures, 2:20-2:50 pm
  - Keep an eye on Piazza for assignment-specific OHs!

- **Apply to section lead!**
  - due **January 30th** (for people who have completed 106B/X)
  - due **February 14th** (for current 106B/X students only)
  - Talk to us about it!
Preview of Assignment 1

due Thursday, January 30

https://web.stanford.edu/class/cs106l/graphviz.html
Assignment 1 Preview

- Pay attention to the “Advice and Common Mistakes” section of the assignment handout!

- General style advice:
  - Use C++, not C! See lecture code as reference.
  - Same as 106B - Decompose and use constants!
  - Use lectures through 1/16 (Sequential Containers) only (avoid iterators and algorithms - you shouldn’t need them for this assignment).
Overview of STL

Containers

Adaptors

Iterators

Functors

Algorithms
Overview of STL

- Containers
- Iterators
- Functors
- Algorithms
Overview of STL

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

Key question: How do we iterate over associative containers?

Remember:

Assoc. containers have no notion of a sequence/indexing!
Iterators

Key question: How do we iterate over associative containers?

Remember:

Assoc. containers have no notion of a sequence/indexing!

```cpp
for(int i = umm?; i < uhh?; i++ maybe?) {
```
Iterators

Key question: How do we iterate over associative containers?

Remember:

Assoc. containers have no notion of a sequence/indexing!

```cpp
for(int i = umm?; i < uhh?; i++ maybe?) {
```
Iterators

Key question: How do we iterate over associative containers?

Remember:

Assoc. containers have no notion of a sequence/indexing!

```cpp
for(int i = ???; i < ???; i++ maybe?) {
  // Code goes here
}
```

C++ has a solution!
Iterators

Iterators allow iteration over any container, whether it is ordered or not.
Iterators

Let’s try and get a mental model of iterators:

Say we have a `std::set<int> mySet`
Iterators

Let’s try and get a mental model of iterators:

Say we have a `std::set<int> mySet`
Iterators

Let’s try and get a mental model of iterators:

Say we have a `std::set<int> mySet`
Let’s try and get a mental model of iterators:

Say we have a `std::set<int> mySet`.

Iterators let us view a non-linear collection in a linear manner.
Iterators

How are they able to represent a non-linear collection in a “sequential” way?

We don’t need to know!

We will just use them like any other thing - assume they just work somehow. This is the power of abstraction!
Iterators - Usage

Let’s try and get a mental model of iterators:
Iterators - Usage

Let’s try and get a mental model of iterators:

We can get an iterator pointing to the “start” of the sequence by calling `mySet.begin()`.

```
1  2  3  4
1
4
mySet.begin();
```
Let’s try and get a mental model of iterators:

```cpp
mySet.begin();
```
Let’s try and get a mental model of iterators:

mySet.begin();

How do we store it in a variable?
Iterators - Usage

Let’s try and get a mental model of iterators:

```cpp
??? iter = mySet.begin();
```
Iterators - Usage

Let’s try and get a mental model of iterators:

```cpp
??? iter = mySet.begin();
```

What is the type of the iterator?
Let’s try and get a mental model of iterators:

```cpp
set<int> mySet;
mySet.beg
```

What is the type of the iterator?

```cpp
??? iter = mySet.begin();
```
Iterators - Usage

Let’s try and get a mental model of iterators:

```cpp
???
iter = mySet.begin();
```
Let’s try and get a mental model of iterators:

```cpp
set<int>::iterator iter = mySet.begin();
```
Iterators - Usage

Let’s try and get a mental model of iterators:

```cpp
set<int>::iterator iter = mySet.begin();
```

It is the iterator type defined in the `set<int>` class!
Let’s try and get a mental model of iterators:

```cpp
set<int>::iterator iter = mySet.begin();
```
Iterators - Usage

Let’s try and get a mental model of iterators:
Let’s try and get a mental model of iterators:

We can get the value of an iterator by using the dereference `*` operator.
Let's try and get a mental model of iterators:

```cpp
cout << *iter << endl; // prints 1
```

We can get the value of an iterator by using the dereference * operator.
Iterators - Usage

Let’s try and get a mental model of iterators:
Let’s try and get a mental model of iterators:

We can advance the iterator one by using the ++ operator (prefix)
Iterators - Usage

Let’s try and get a mental model of iterators:

```
1 ++iter; // advances iterator
```

We can advance the iterator one by using the ++ operator (prefix)
Iterators - Usage

Let’s try and get a mental model of iterators:
Iterators - Usage

Let’s try and get a mental model of iterators:

```
cout << *iter << endl;  // prints 2
```
Iterators - Usage

Let’s try and get a mental model of iterators:

And so on...
Iterators - Usage

Let’s try and get a mental model of iterators:

```
++iter;    // advances iterator
```
Let’s try and get a mental model of iterators:

And so on...
Let’s try and get a mental model of iterators:

```
iter = 1
cout << *iter << endl;  // prints 3
iter = 2
cout << *iter << endl;  // prints 2
iter = 3
cout << *iter << endl;  // prints 1
iter = 4
```

And so on...
And so on...

Iterators - Usage

Let’s try and get a mental model of iterators:

```
1  2  3  4
```

`iter`
Iterators - Usage

Let’s try and get a mental model of iterators:

```
++iter; // advances iterator
```
Let’s try and get a mental model of iterators:

Iterators - Usage

And so on...
Iterators - Usage

Let’s try and get a mental model of iterators:

```
int arr[] = {1, 2, 3, 4};
int iter = 1;
while (iter < 4) {
    cout << *iter << endl;
    iter++;
}
```

// prints 1 2 3
Iterators - Usage

Let’s try and get a mental model of iterators:

And so on...
Iterators - Usage

Let’s try and get a mental model of iterators:

```
++iter;  // advances iterator
```
Iterators - Usage

Let’s try and get a mental model of iterators:
Iterators - Usage

Let’s try and get a mental model of iterators:

We can check if we have hit the end by comparing to mySet.end();
Let’s try and get a mental model of iterators:

```
if (iter == mySet.end()) return;
```
A summary of the essential iterator operations:

- **Create** iterator
- **Dereference** iterator to read value currently pointed to
- **Advance** iterator
- **Compare** against another iterator (especially the `end()` iterator)
Example

Basic Iterator Usage
Our examples have used sets, but (almost) all C++ containers have iterators.

Why is this powerful?

- Many scenarios require looking at elements, regardless of what type of container is storing those elements.
- Iterators let us go through sequences of elements in a standardised way.
- C++ is huge!
Iterators

Example (find number occurrences):

```cpp
int numOccurrences(vector<int>& cont, int elemToCount) {
    int counter = 0;
    vector<int>::iterator iter;
    for (iter = cont.begin(); iter != cont.end(); ++iter) {
        if (*iter == elemToCount)
            ++counter;
    }
    return counter;
}
```
Iterators

Example (find number occurrences):

```cpp
int numOccurrences(vector<int>& cont, int elemToCount) {
  int counter = 0;
  vector<int>::iterator iter;
  for(iter = cont.begin(); iter != cont.end(); ++iter) {
    if(*iter == elemToCount)
      ++counter;
  }
  return counter;
}

Can I make this work for `std::list<int>`?
```
Iterators

Example (find number occurrences):

```cpp
int numOccurrences(vector<int>& cont, int elemToCount) {
    int counter = 0;
    vector<int>::iterator iter;
    for (iter = cont.begin(); iter != cont.end(); ++iter) {
        if (*iter == elemToCount)
            ++counter;
    }
    return counter;
}
```

Can I make this work for `std::list<int>`?
Iterators

Example (find number occurrences):

```cpp
int numOccurrences(list<int>& cont, int elemToCount) {
    int counter = 0;
    list<int>::iterator iter;
    for (iter = cont.begin(); iter != cont.end(); ++iter) {
        if (*iter == elemToCount)
            ++counter;
    }
    return counter;
}
```
Iterators

Example (find number occurrences):

```cpp
int numOccurrences(list<int>& cont, int elemToCount) {
    int counter = 0;
    list<int>::iterator iter;
    for(iter = cont.begin(); iter != cont.end(); ++iter) {
        if(*iter == elemToCount)
            ++counter;
    }
    return counter;
}
```

What about std::set<int>?
Iterators

Example (find number occurrences):

```cpp
int numOccurrences(list<int> & cont, int elemToCount) {
    int counter = 0;
    list<int>::iterator iter;
    for(iter = cont.begin(); iter != cont.end(); ++iter) {
        if(*iter == elemToCount)
            ++counter;
    }
    return counter;
}
```
```cpp
int numOccurrences(set<int>& cont, int elemToCount) {
    int counter = 0;
    set<int>::iterator iter;
    for(iter = cont.begin(); iter != cont.end(); ++iter) {
        if(*iter == elemToCount)
            ++counter;
    }
    return counter;
}
```
This standard interface for looping through things is going to be really **powerful**.

We will cover it sometime this week or next week!
Map Iterators
Map Iterators

Map iterators are slightly different because we have both keys and values.

The iterator of a `map<string, int>` points to a `std::pair<string, int>`.
The `std::pair` Class

A pair is simply two objects bundled together.

Syntax:

```cpp
std::pair<string, int> p;
p.first = "Phone number";
p.second = 6504550404;
```
Map Iterators

Example:

```cpp
map<int, int> m;

map<int, int>::iterator i = m.begin();

map<int, int>::iterator end = m.end();

while (i != end) {
    cout << (*i).first << (*i).second << endl;
    ++i;
}
```