Bad Dad Joke of the Day:

- Son: “Daddy, were you shot in the Army?”
- Dad: *painful look in his eyes*
- Dad: “No, but I was shot in the leggy.”

Creds: SP
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

\texttt{std::vector<T>}

- `vec.at(i)` throws an exception
- `vec[i]` \textbf{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):
Container Adaptors
Container Adaptors

Recall stacks and queues:

stack

13
41
12
Recall stacks and queues:

```
stack
```

```
push
```
Container Adaptors

Recall stacks and queues:

stack
5
13
41
12
Recall stacks and queues:

Container Adaptors

stack

13
41
12

pop
Container Adaptors

Recall stacks and queues:

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

Container Adaptors

stack

queue

13
41
12
9
16
11
Recall stacks and queues:

Container Adaptors

stack

13
41
12

queue

9
16
11
5

push_back
Recall stacks and queues:

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

- **Queue**
  - 9
  - 16
  - 11
  - 5
  - back
Container Adaptors

Recall stacks and queues:

- Stack:
  - 13
  - 41
  - 12

- Queue:
  - 16
  - 11
  - 5
  - pop_front
  - back
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.

**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

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.
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
Associative Containers

Have no idea of a sequence.

Data is accessed using the key instead of indexes.

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 the key instead of indexes.

Includes:

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

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 the **key** instead of **indexes**.

Includes:

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

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 the key instead of indexes.

Includes:

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

You can define `<` and hash function operators for your own classes!
Associative Containers

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

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

- Map/set: keys in sorted order, 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<T1,T2>}

Methods mostly same as Stanford map.

See \texttt{documentation} for full list of methods.

Key Takeaways:

- \texttt{mymap.at(key)} vs. \texttt{mymap[key]}
Methods mostly same as Stanford map.

See documentation for full list of methods.

Key Takeaways:

- `mymap.at(key) vs. mymap[key]`
- Equivalent of Stanford `.containskey(key)`:
  - `mymap.count(key)`
  - And 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
Example

See website for C++ Sets Example
Announcements
Announcements

• Office hours:
  ○ Anna: **Thursdays, 9-11 am** on the Tresidder 2nd floor patio (facing Dink)
  ○ Avery: **Thursdays, 7-10 pm** in TAP

• Apply to section lead!
  ○ due **October 17th** (for people who have completed 106B/X)
  ○ due **November 1st** (for current 106B/X students only)

• Feedback form released tonight! (+1 late day)
Preview of Assignment 1

due Thursday, October 17

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

● “Advice, Tips, and Tricks” notes:
  ○ “Read graph files correctly”: suggested loop structure in Chapter 3 of the course reader
Assignment 1 Preview

● “Advice, Tips, and Tricks” notes:
  ○ “Read graph files correctly”: suggested loop structure in Chapter 3 of the course reader
  ○ “Reading input from the user”: use GetLine from the Maps Example for inspiration
Assignment 1 Preview

- “Advice, Tips, and Tricks” notes:
  - “Read graph files correctly”: suggested loop structure in Chapter 3 of the course reader
  - “Reading input from the user”: use GetLine from the Maps Example for inspiration

- General style advice:
  - Same as 106B/X - Decompose and use constants!
Overview of STL

- Algorithms
- Iterators
- Containers
- Functors/Lambdas
- Adaptors
- And more!
Overview of STL

- Algorithms
- Iterators
- Containers
- Functors/Lambdas
- Adaptors

And more!
Iterators
Iterators

How do we iterate over associative containers?

Remember:

Assoc. containers have no notion of a sequence/indexing

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

How do we iterate over associative containers?

Remember:

Assoc. containers have no notion of a sequence/indexing

```c++
for(int i = umm?; i < uhh?; i++ maybe?) {
    C++ has a solution!
}
First: A note
A note

We are going on a journey.
We could in theory fly to the end destination.
A note
A note

But then we would miss the experience
A note

We are going on a journey.
A note

We are going on a journey.

At the end lies simplicity, but if we jump to it we miss out on understanding.
We are going on a journey.

At the end lies simplicity, but if we jump to it we miss out on understanding.

So we will walk to it.
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`
Iterators

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

We don’t care right now.*

* They do an in-order traversal on a binary tree.

We will just use them like any other thing - assume they just work somehow.
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 an iterator pointing to the “start” of the sequence by calling `mySet.begin()`.

```cpp
mySet.begin();
```
Iterators - Usage

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

```
mySet.begin();
```
Iterators - Usage

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

1 2 3 4

mySet.begin();

How do we store it in a variable?
Let’s try and get a mental model of iterators:

```??? 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
```

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

```
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.
Iterators - Usage

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

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

cout << *iter << endl;  // prints 1
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 advance the iterator one by using the `++` operator (prefix)
Iterators - Usage

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

We can advance the iterator one by using the `++` operator (prefix)

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

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

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

```
1 2 3 4
```

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

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

1
2
3
4

And so on...
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...
Iterators - Usage

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

```
cout << *iter << endl; // prints 3
```
Iterators - Usage

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

And so on...
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

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

```
1 2 3 4
```

```
cout << *iter << endl; // prints 4
```
Iterators - Usage

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

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

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

![Diagram of an iterator moving through a collection of numbers 1 to 4]
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()`.
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():

```java
if (iter == mySet.end()) return;
```
Iterators - Usage

A summary of the essential iterator operations:

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

Basic Iterator Usage
Iterators

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 numOccurences(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 numOccurences(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 numOccurences(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 numOccurences(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 numOccurences(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 numOccurences(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>`. 
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;
}
```
Next time

Advanced Containers