

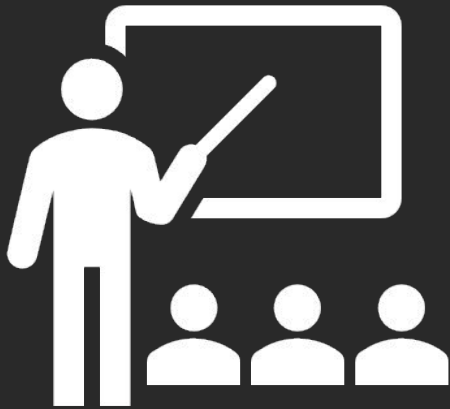
# Associative Containers & Iterators

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

<code>basic_string</code>	stores and manipulates sequences of characters (class template)
<code>array</code> (C++11)	static contiguous array (class template)
<code>vector</code>	dynamic contiguous array (class template)
<code>deque</code>	double-ended queue (class template)
<code>forward_list</code> (C++11)	singly-linked list (class template)
<code>list</code>	doubly-linked list (class template)

## Trade-offs / usage notes

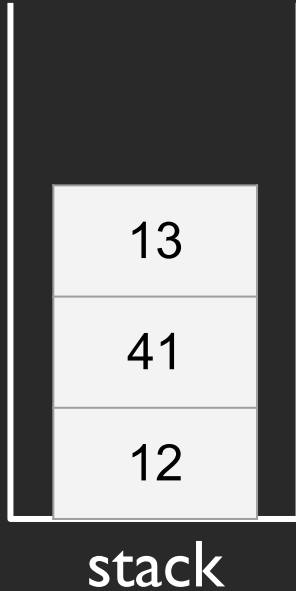
<code>std::array</code>	Fast access but fixed number of elements
<code>std::vector</code>	Fast access but mostly inefficient insertions/deletions
<code>std::list</code> <code>std::forward_list</code>	Efficient insertion/deletion in the middle of the sequence
<code>std::deque</code>	Efficient insertion/deletion at the beginning and at the end of the sequence



# Container Adaptors

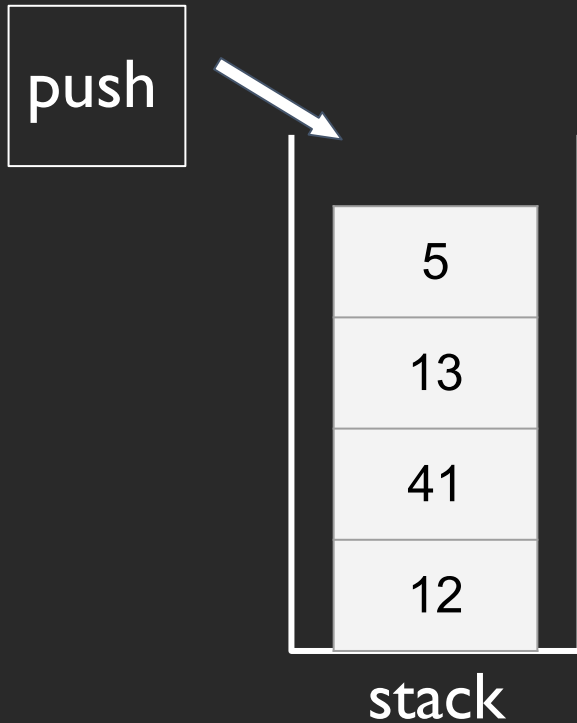
# Container Adaptors

Recall stacks and queues:



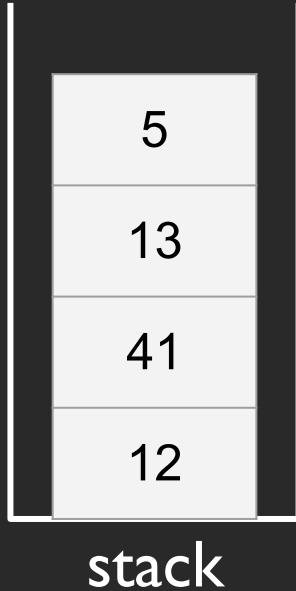
# Container Adaptors

Recall stacks and queues:



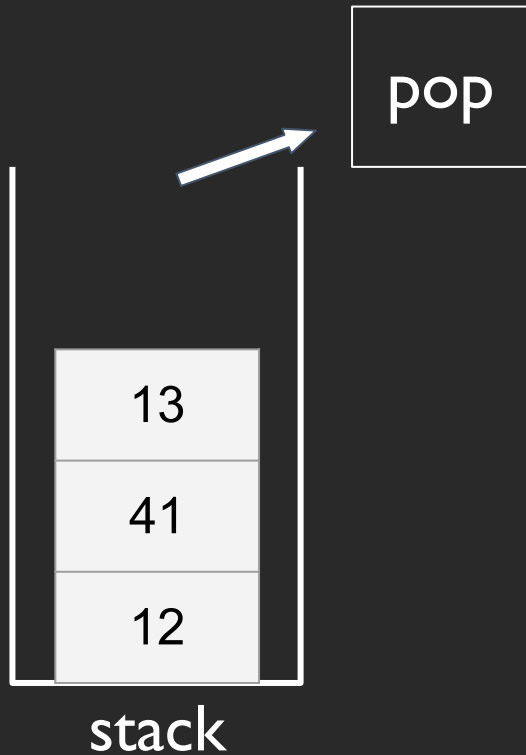
# Container Adaptors

Recall stacks and queues:



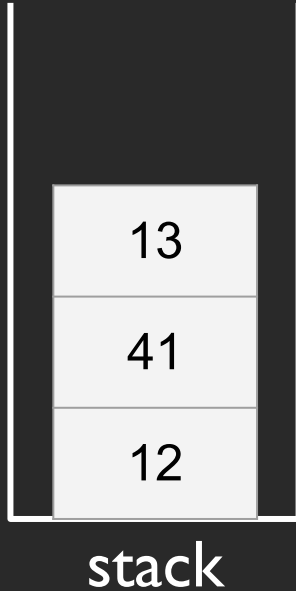
# Container Adaptors

Recall stacks and queues:



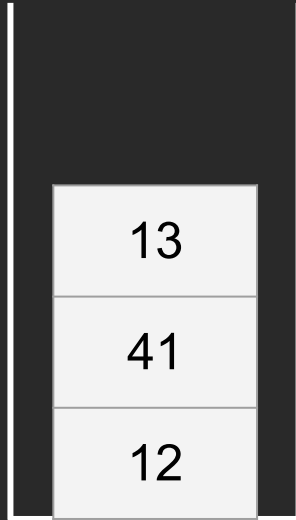
# Container Adaptors

Recall stacks and queues:

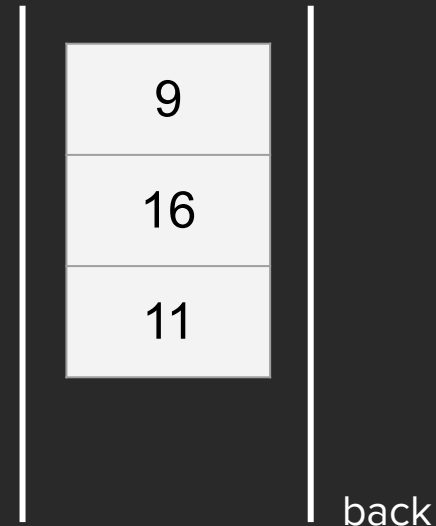


# Container Adaptors

Recall stacks and queues:



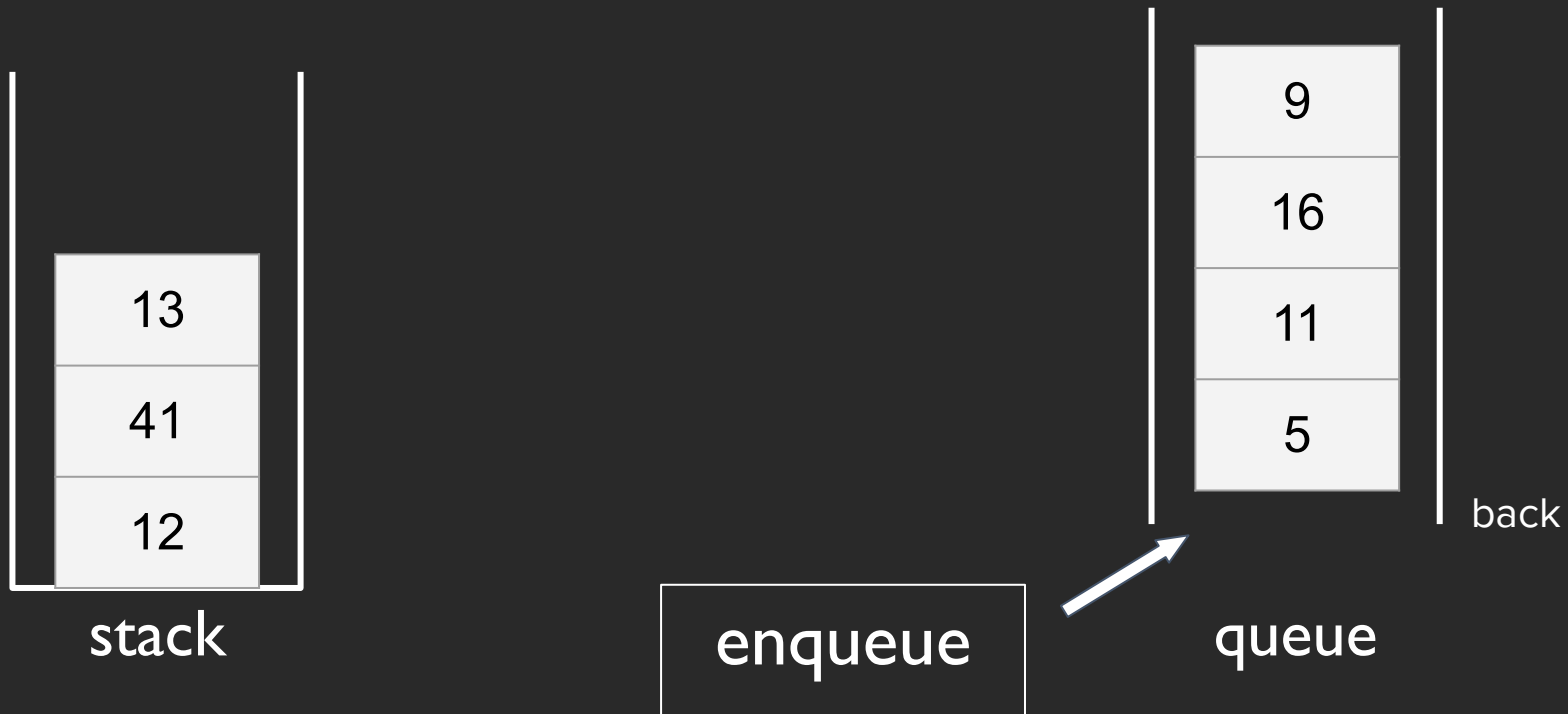
stack



queue

# Container Adaptors

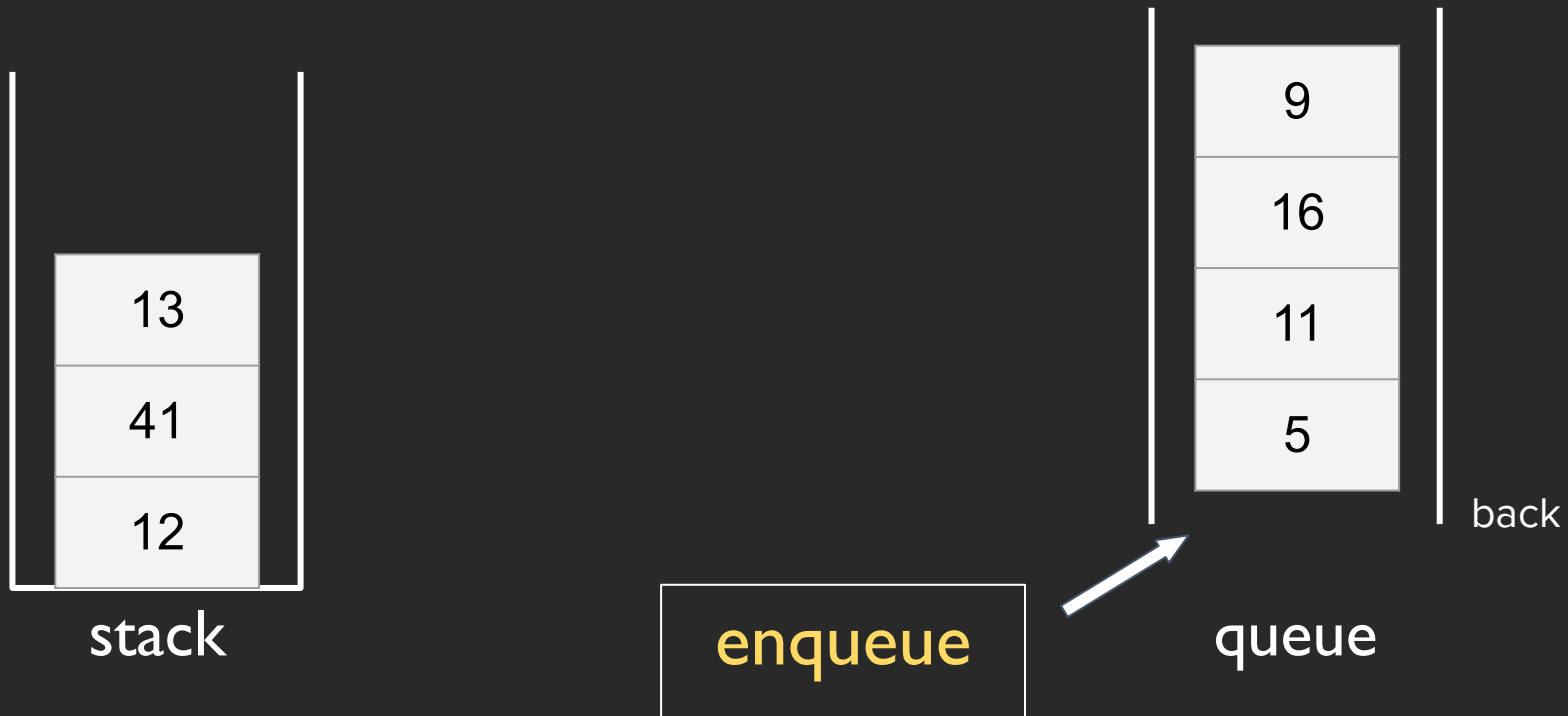
Recall stacks and queues:





# Container Adaptors

Recall stacks and queues:



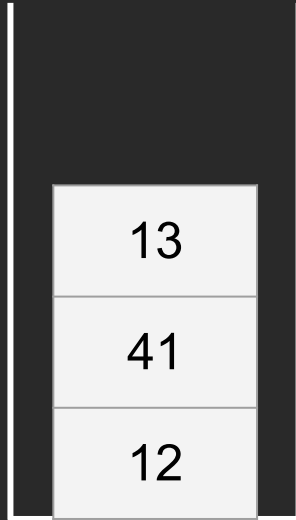
# Container Adaptors

Recall stacks and queues:

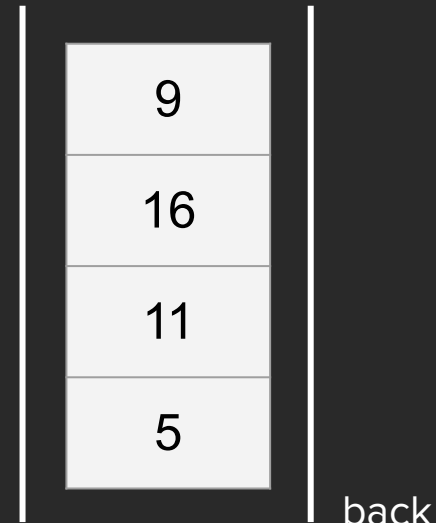


# Container Adaptors

Recall stacks and queues:



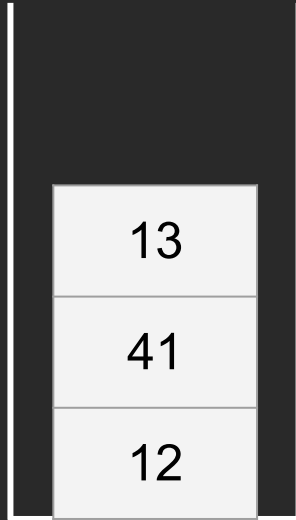
stack



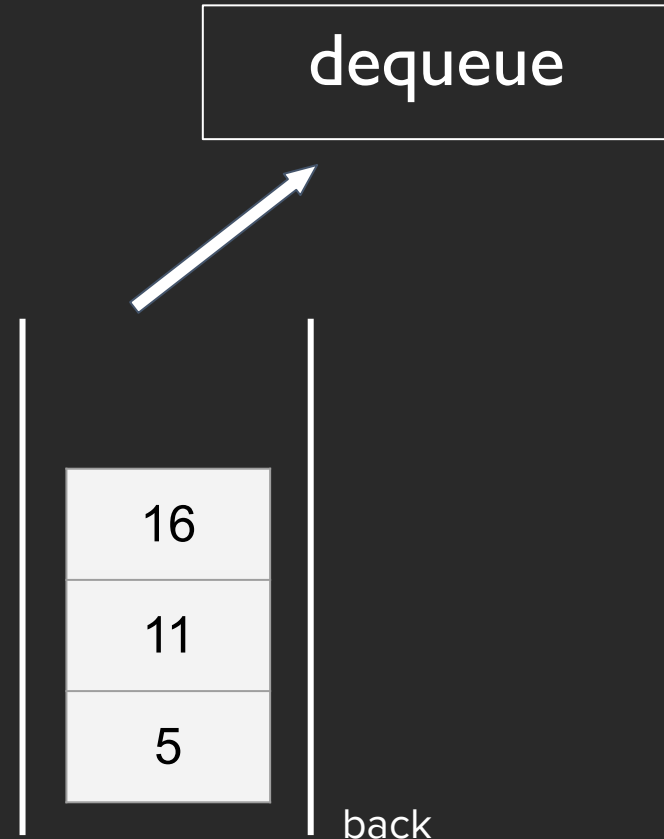
queue

# Container Adaptors

Recall stacks and queues:



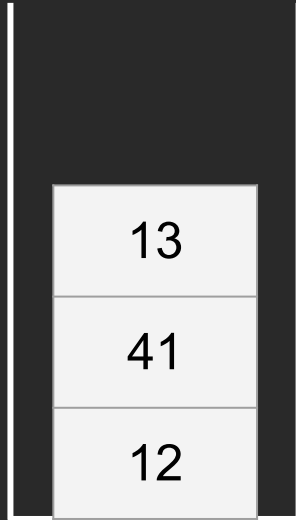
stack



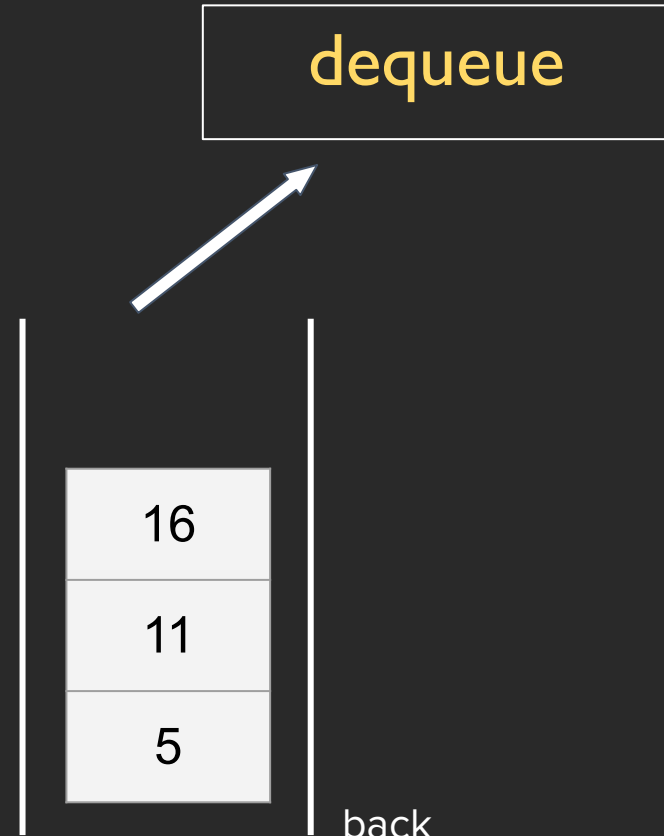
queue

# Container Adaptors

Recall stacks and queues:



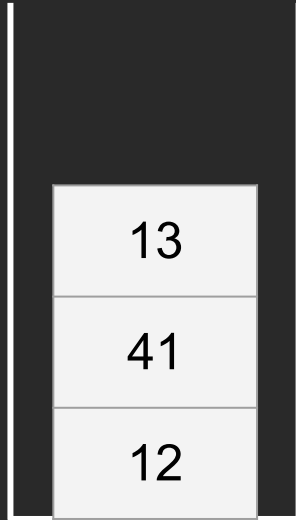
stack



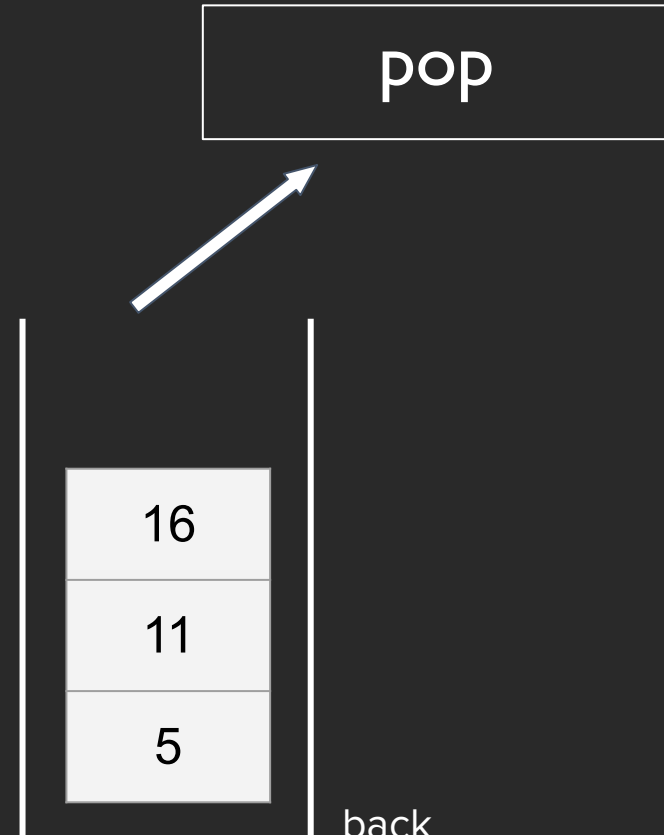
queue

# Container Adaptors

Recall stacks and queues:



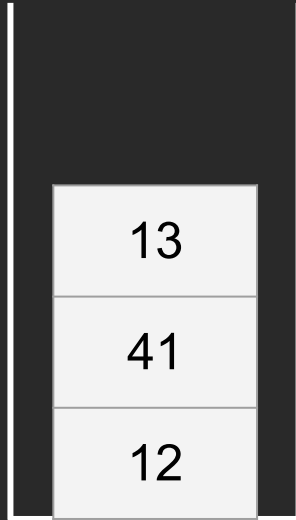
stack



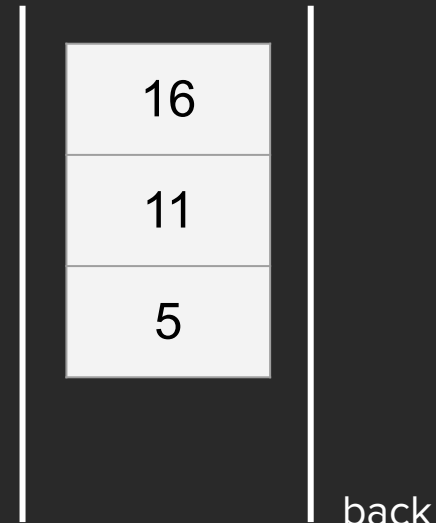
queue

# Container Adaptors

Recall stacks and queues:



stack



queue

# 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`

?



# Container Adaptors

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>`

```
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>`

```
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>`

```
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.

# 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 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>`

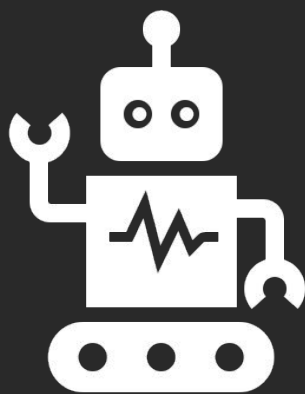
## Map/set:

- 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

```
std::map<T1, T2>
```

Methods mostly same as Stanford map.

See [documentation](#) for full list of methods.

```
std::map<T1, T2>
```

Methods mostly same as Stanford map.

See [documentation](#) for full list of methods.

Key Takeaways:

- `mymap.at(key)` vs. `mymap[key]`

```
std::map<T1, T2>
```

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!



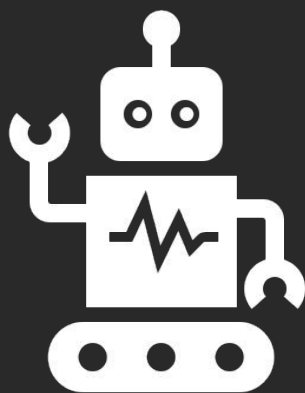
```
std::set<T>
```

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

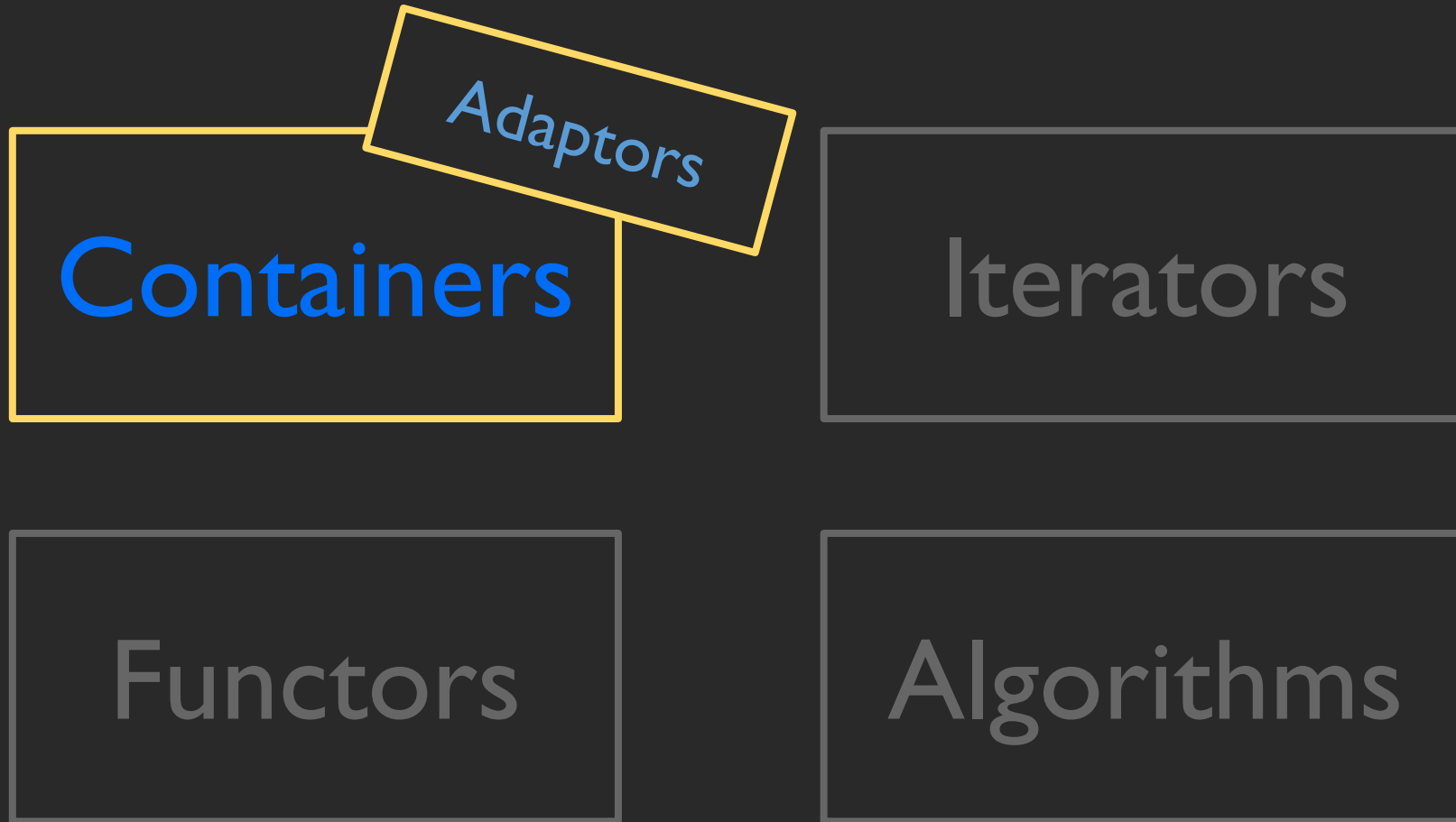
due Thursday, January 30

<https://web.stanford.edu/class/cs106l/graphviz.html>

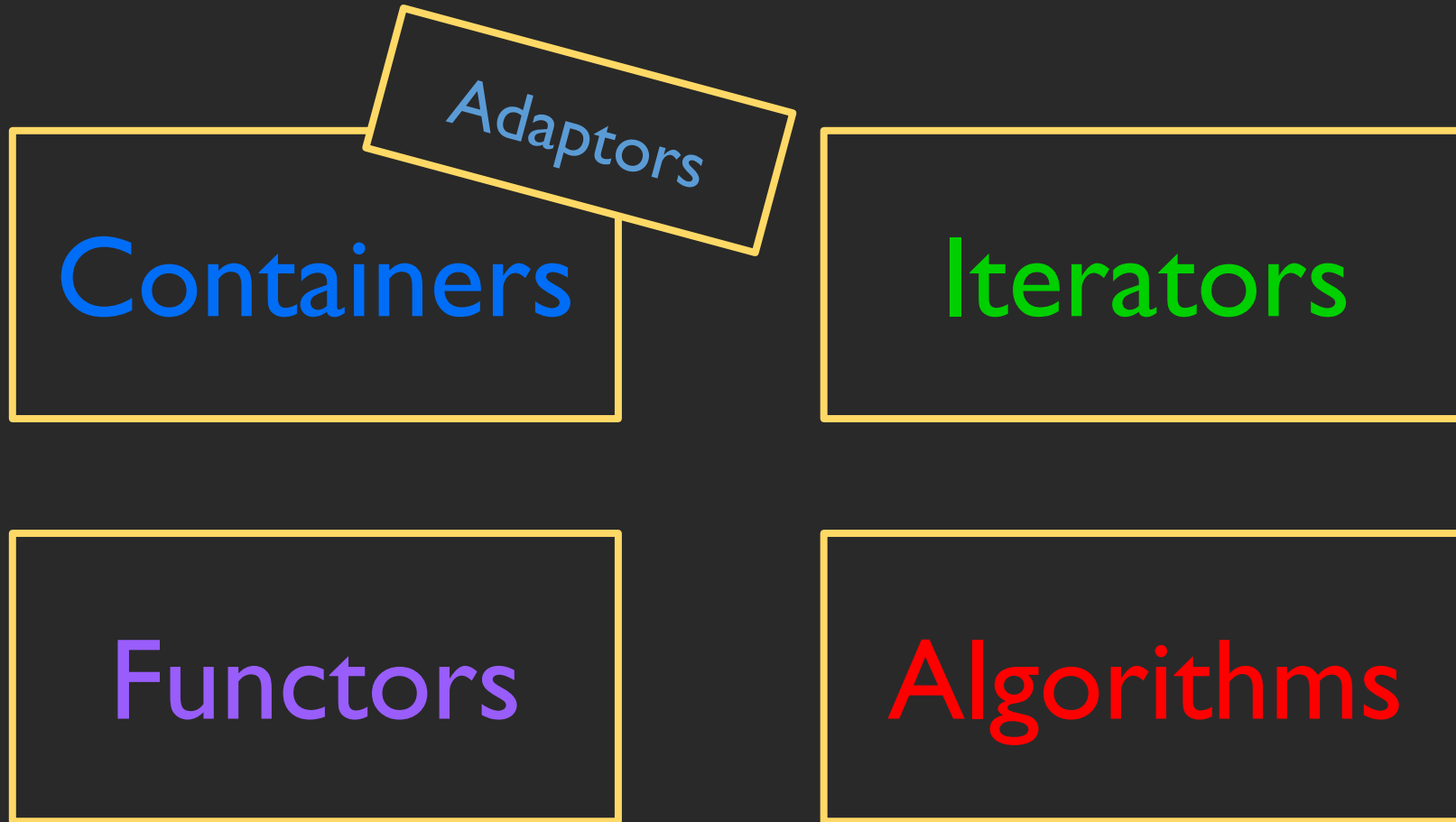
# Assignment 1 Preview

- 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

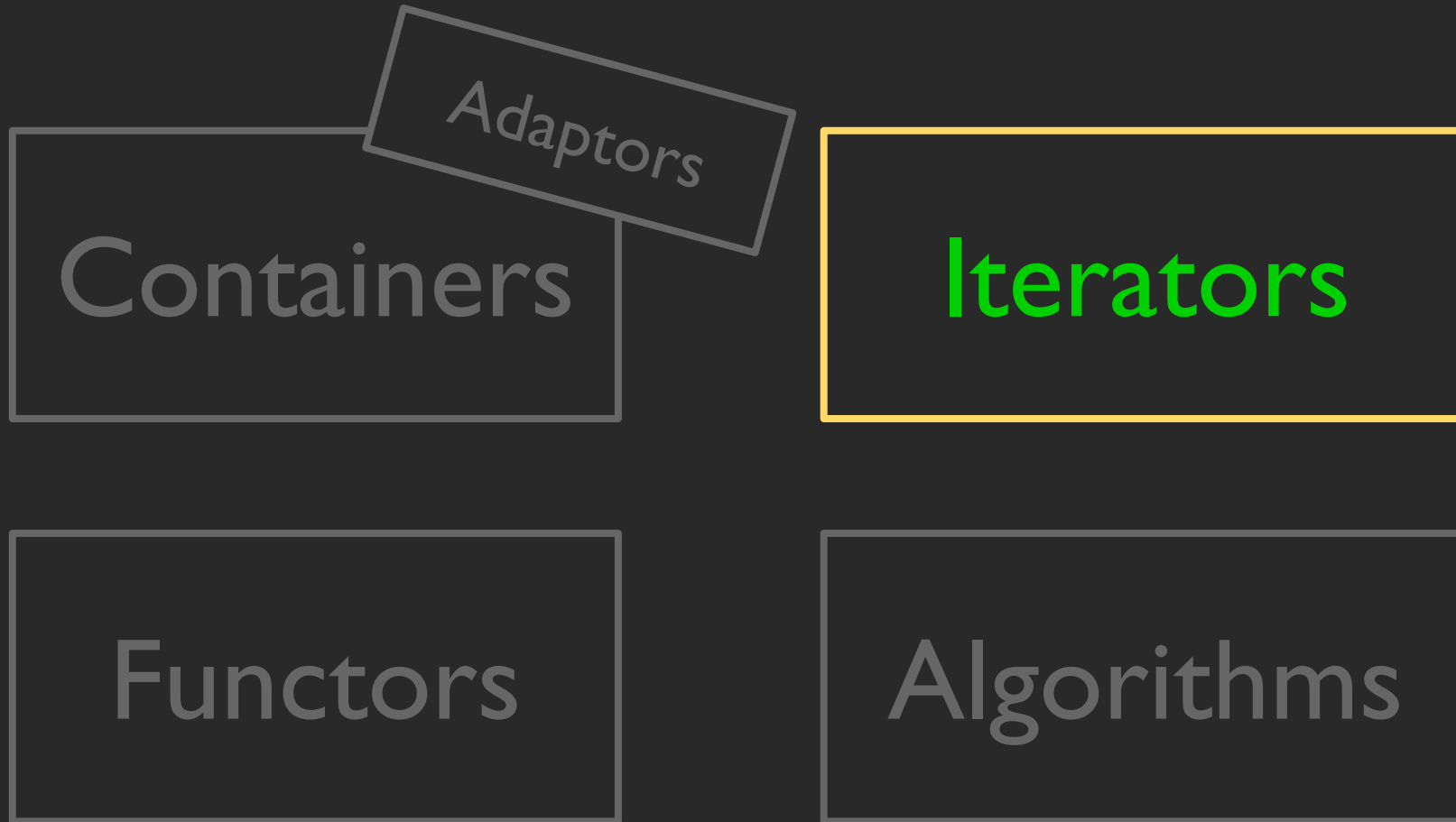


# Overview of STL





# Overview of STL



# 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!

```
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!

```
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!

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

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:

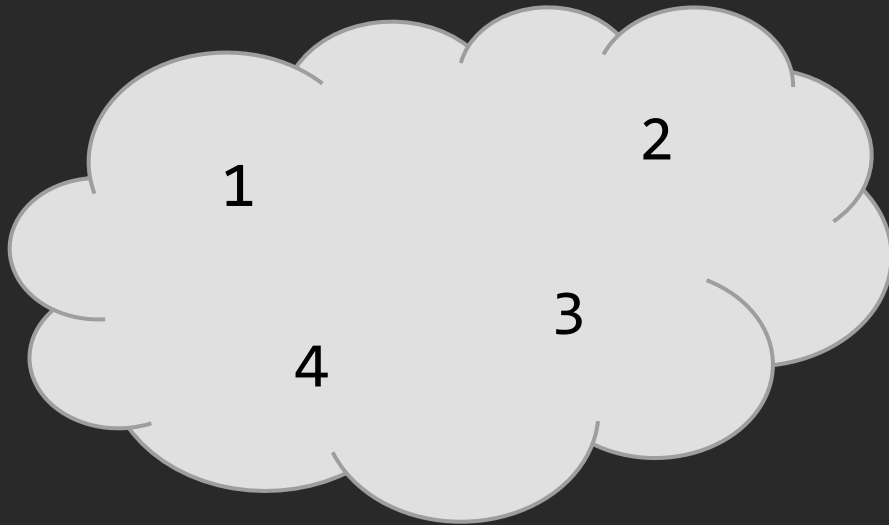
Say we have a `std::set<int> mySet`



# Iterators

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

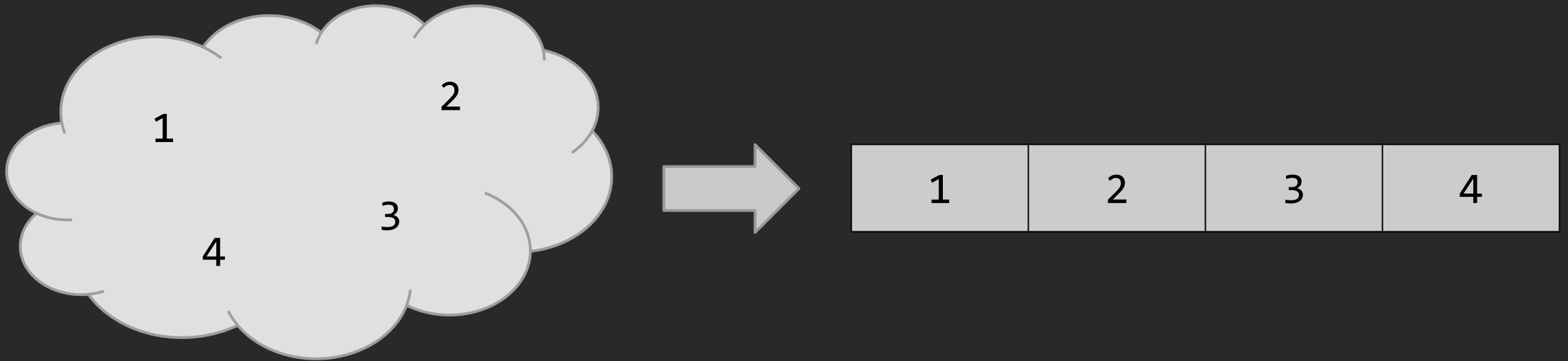
Say we have a `std::set<int> mySet`



# Iterators

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

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



# 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.



# Iterators

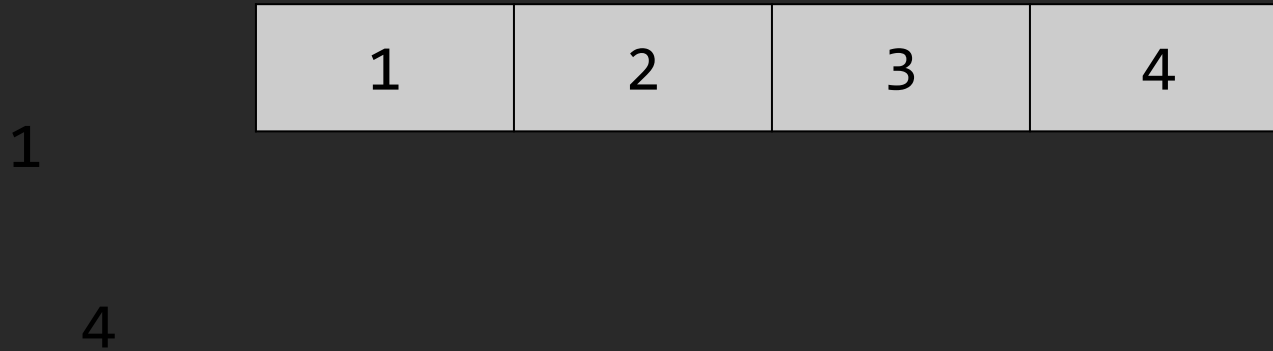
How are they able to represent a non-linear collection in a “sequential” way?

**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

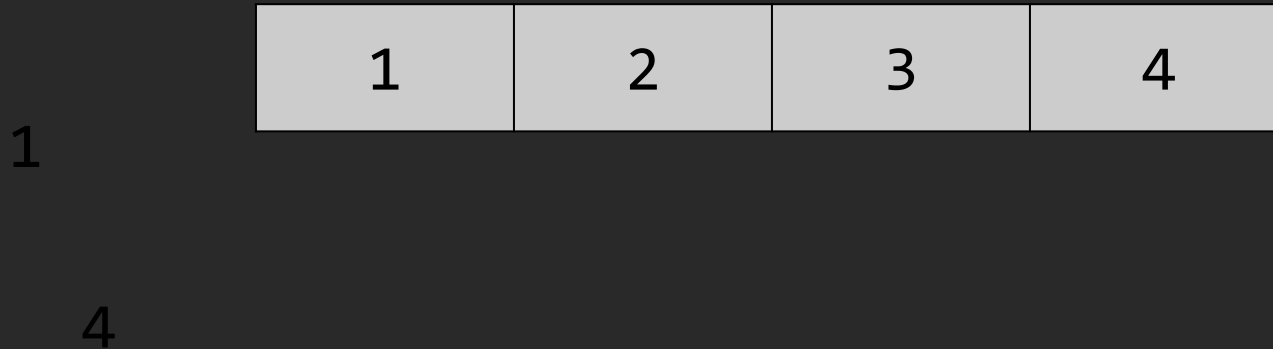
Let's try and get a mental model of iterators:



# Iterators - Usage

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

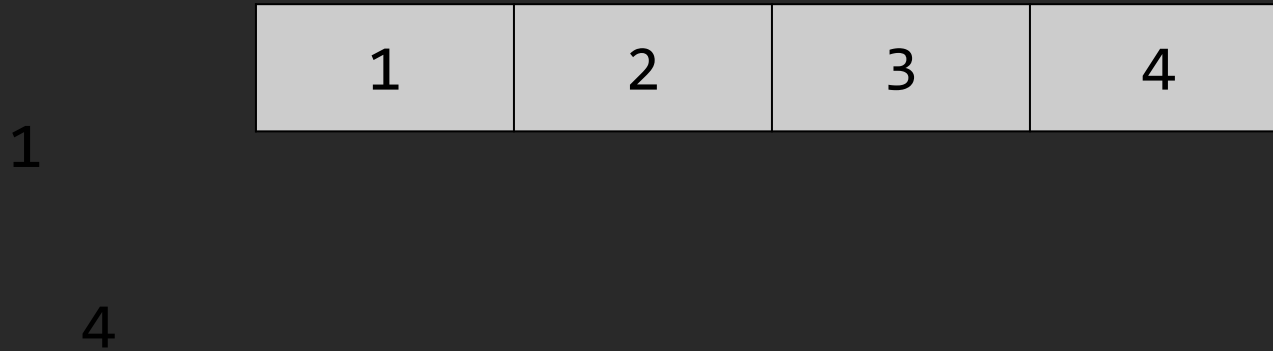
We can get an iterator pointing to the “start” of the sequence by calling `mySet.begin()`



```
mySet.begin();
```

# Iterators - Usage

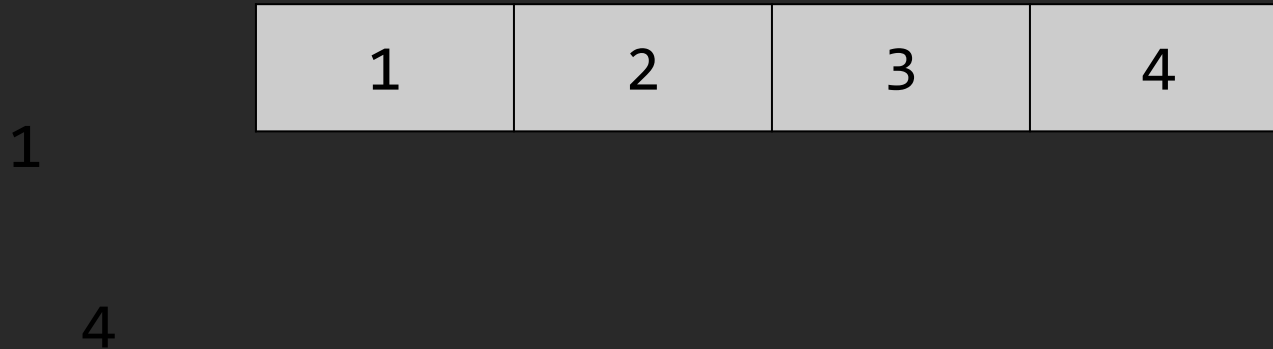
Let's try and get a mental model of iterators:



```
mySet.begin();
```

# Iterators - Usage

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



```
mySet.begin();
```

How do we store it in a variable?



# Iterators - Usage

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



1

4

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

# Iterators - Usage

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



1

What is the type of  
the iterator? 4



??? iter = mySet.begin();

# Iterators - Usage

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



1  
What is the type of  
the iterator? 4

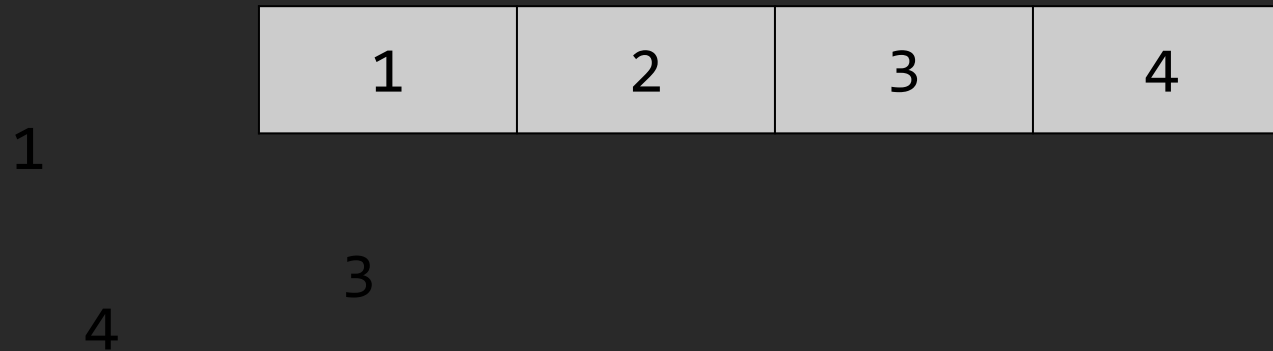


??? iter = mySet.begin();

```
set<int> mySet;  
mySet.begin  
  begin iterator begin()
```

# Iterators - Usage

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:



4

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

# Iterators - Usage

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

It is the `iterator` type defined in the `set<int>` class!



4

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

# Iterators - Usage

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

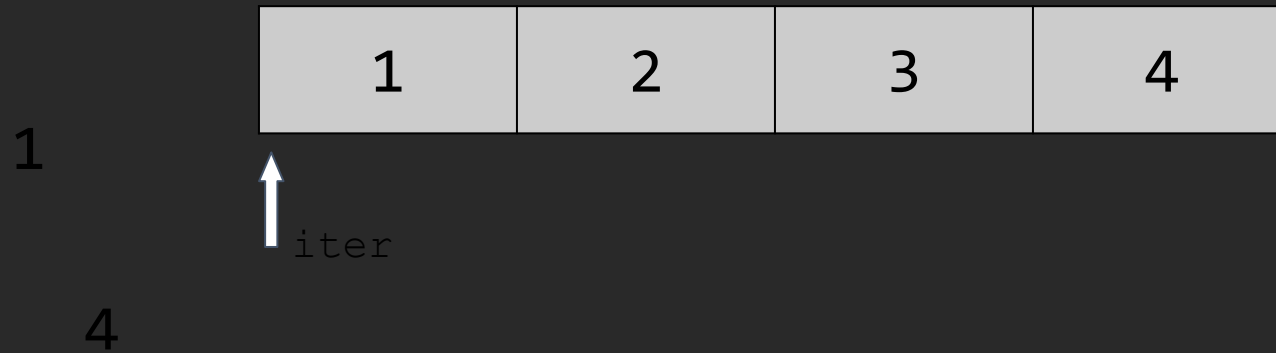


4

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

# Iterators - Usage

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

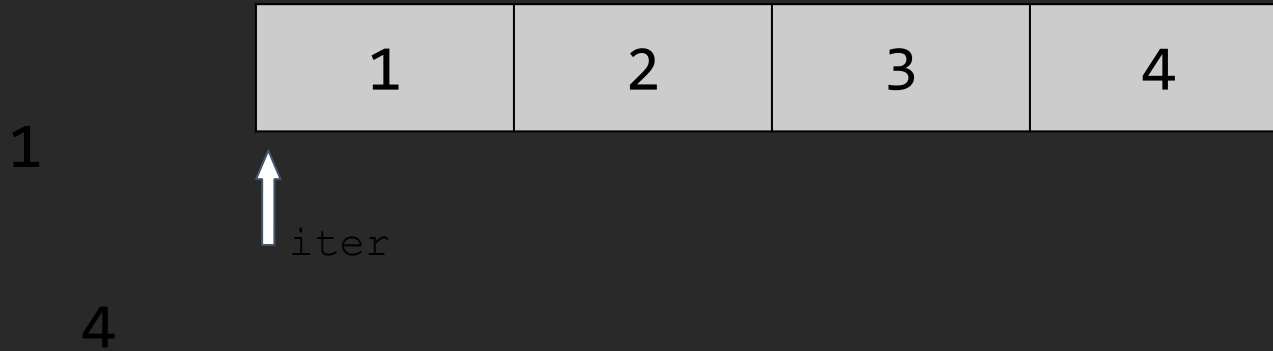




# Iterators - Usage

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

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



# Iterators - Usage

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

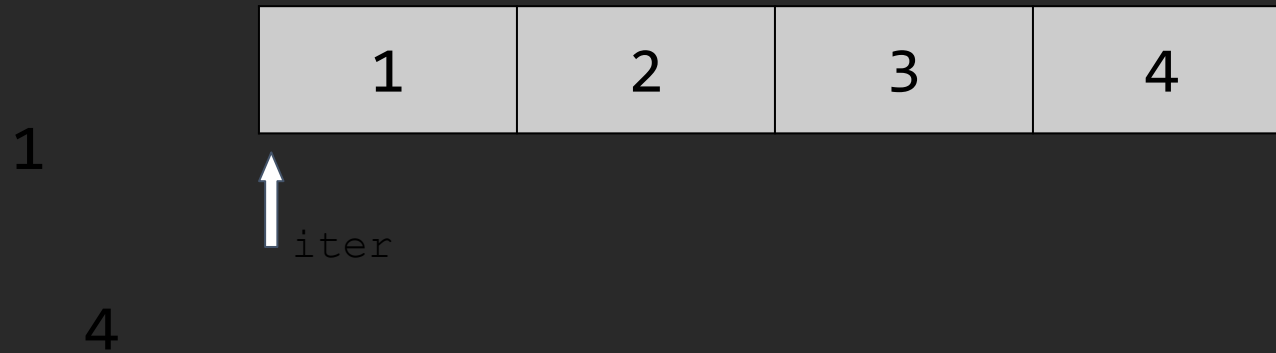
We can get the value of an iterator by using the dereference `*` operator.



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

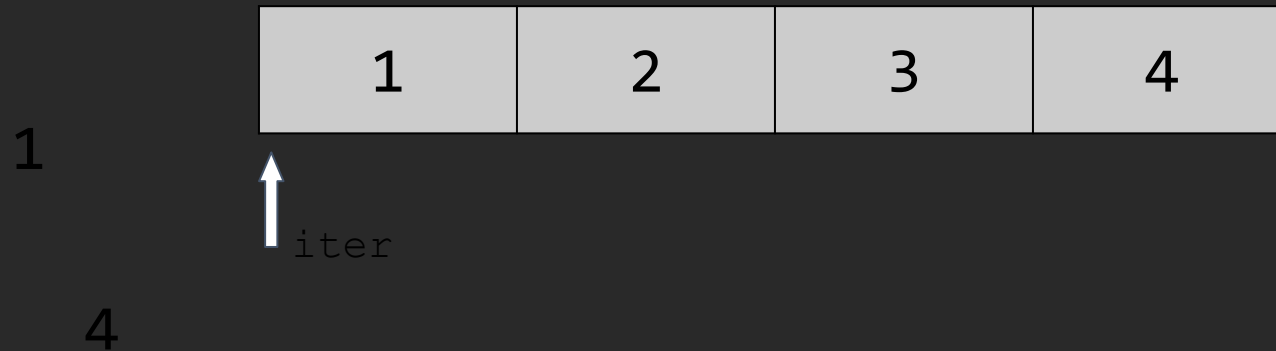
# Iterators - Usage

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



# Iterators - Usage

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

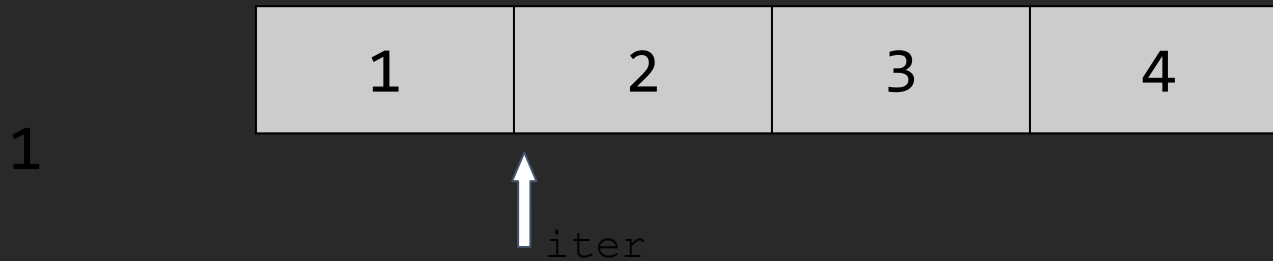


We can advance the iterator one by using the `++` operator (prefix)

# Iterators - Usage

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

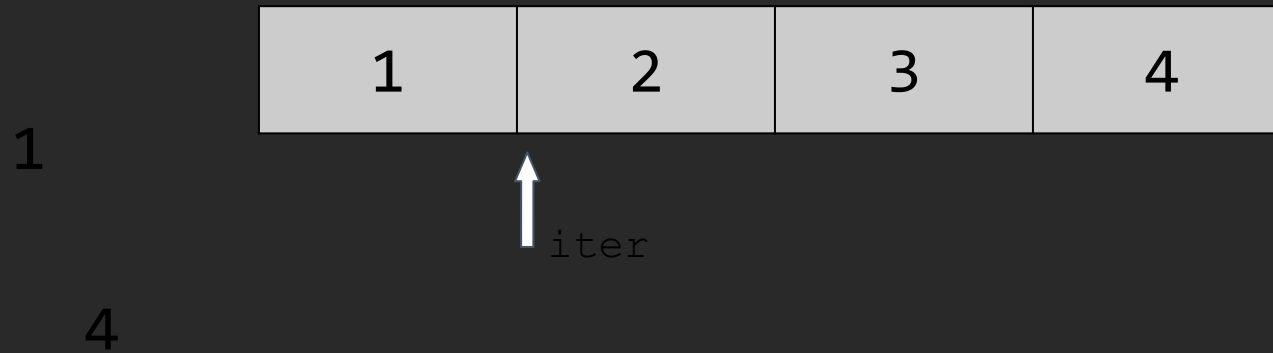
We can advance the iterator one by using the `++` operator (prefix)



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

# Iterators - Usage

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

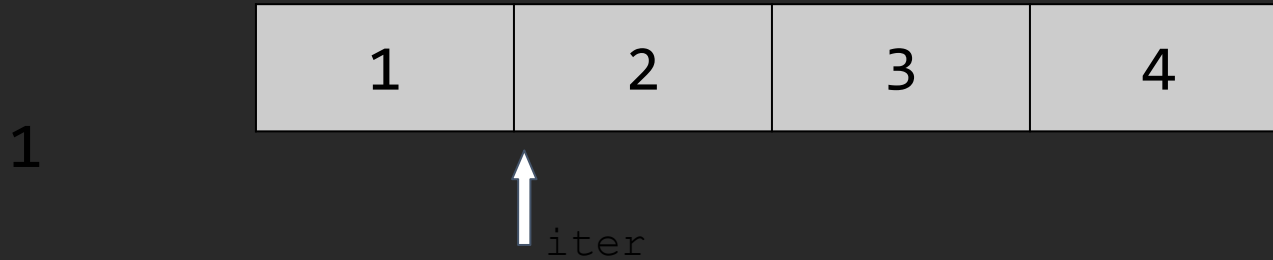


And so on...

# Iterators - Usage

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

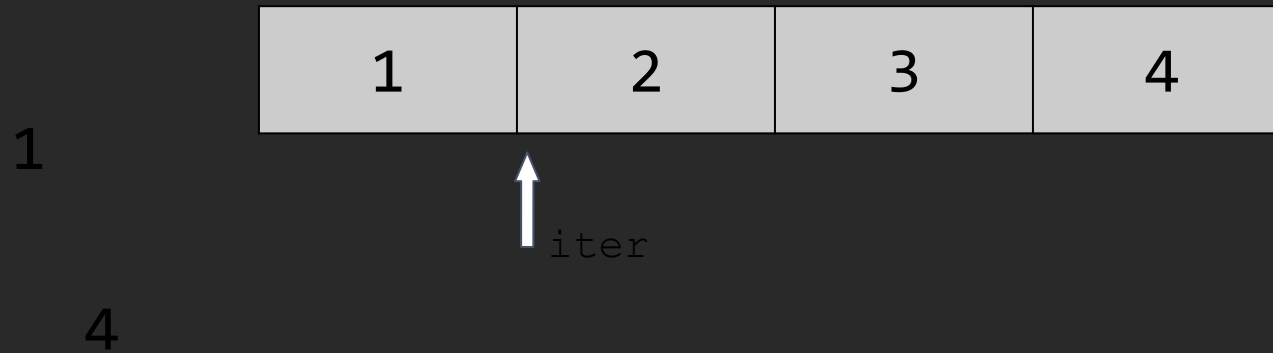
And so on...



```
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:

And so on...



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

# 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:

And so on...



```
cout << *iter << endl;    // prints 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:

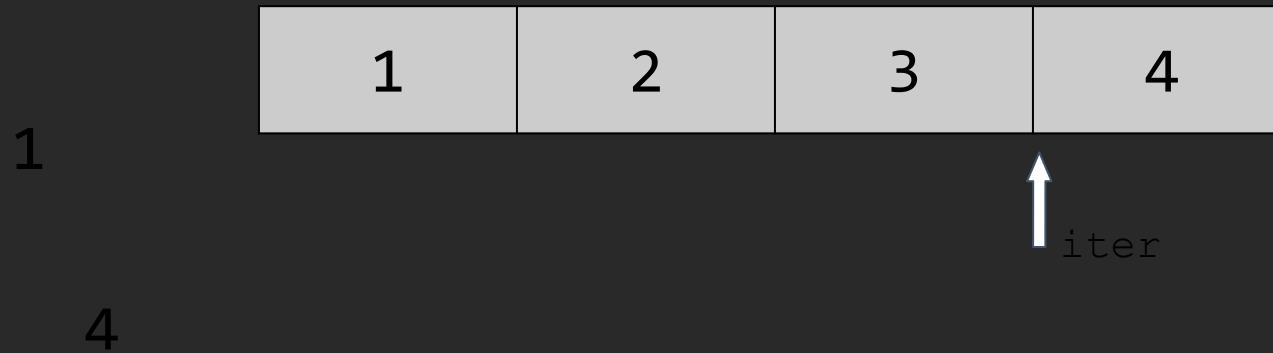
And so on...



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

# 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:

And so on...



```
cout << *iter << endl;    // prints 4
```

# Iterators - Usage

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

1



iter

And so on...



# Iterators - Usage

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

And so on...

1



↑  
iter

```
++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:

1



↑  
iter

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:

1



↑  
iter

We can check if we have hit the end by comparing to `mySet.end()`

```
if (iter == mySet.end()) return;
```

# Iterators - Usage

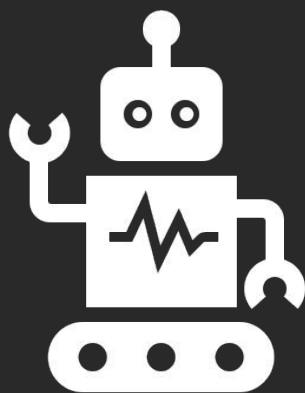
A summary of the essential iterator operations:

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):

```
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):

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

```
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):

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

```
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):

```
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):

What about  
`std::set<int>?`

```
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):

What about  
`std::set<int>?`

```
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):

```
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;  
}
```

# Iterators

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:

```
std::pair<string, int> p;
```

```
p.first = "Phone number";
```

```
p.second = 6504550404;
```

# Map Iterators

Example:

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

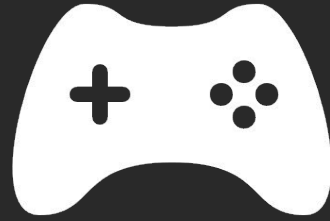
```
map<int, int>::iterator i = m.begin();
```

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

```
while (i != end) {
```

```
    cout << (*i).first << (*i).second << endl;
```

```
    ++i;
```



# Next time

## Advanced Containers