http://web.stanford.edu/class/cs106l/

Containers

What are they? How do we use them? How do they differ from their Stanford Library counterparts?

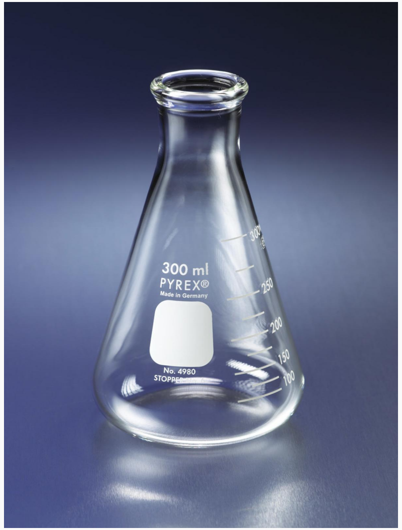
CS106L - Fall 22

Containers

http://web.stanford.edu/class/cs106l/

Attendance!

https://bit.ly/3fS5Dku



Recap:

- **Uniform Initialization**
  - A “uniform” way to initialize variables of different types!
- **References**
  - Allow us to assign aliases to variables
- **Const**
  - Allow us to specify that a variable can’t be modified

Containers

http://web.stanford.edu/class/cs106l/

Recap:

- **Uniform Initialization**
  - A “uniform” way to initialize variables of different types!
- **References**
  - Allow us to assign aliases to variables
- **Const**
  - Allow us to specify that a variable can’t be modified

CRASH COURSE: Const and Const References

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);
c_vec.push_back(3);
ref.push_back(3);
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);    // OKAY
c_vec.push_back(3);
ref.push_back(3);
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);     // OKAY
c_vec.push_back(3);   // BAD - const
ref.push_back(3);
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);     // OKAY
c_vec.push_back(3);   // BAD - const
ref.push_back(3);    // OKAY
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);    // OKAY
c_vec.push_back(3);  // BAD - const
ref.push_back(3);   // OKAY
c_ref.push_back(3); // BAD - const
```

Can’t declare non-const reference to const variable!

```
const std::vector<int> c_vec{7, 8};  // a const variable

// BAD - can't declare non-const ref to const vector
std::vector<int>& bad_ref = c_vec;
```

Can’t declare non-const reference to const variable!

```
const std::vector<int> c_vec{7, 8};  // a const variable

// fixed
const std::vector<int>& bad_ref = c_vec;
```

Can’t declare non-const reference to const variable!

```
const std::vector<int> c_vec{7, 8};  // a const variable

// fixed
const std::vector<int>& bad_ref = c_vec;

// BAD - Can't declare a non-const reference as equal
// to a const reference!
std::vector<int>& ref = c_ref;
```

`const` & subtleties

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};

std::vector<int>& ref = vec;
const std::vector<int>& c_ref = vec;

auto copy = c_ref;             // a non-const copy
const auto copy = c_ref;       // a const copy
auto& a_ref = ref;             // a non-const reference
const auto& c_aref = ref;      // a const reference
```

Containers

http://web.stanford.edu/class/cs106l/

Agenda

01. Defining Containers
What is a container in C++?

02. Containers in the STL vs Stanford
Types of containers and how they work

03. Container Adaptors
Abstracting container implementation

Containers

http://web.stanford.edu/class/cs106l/

Agenda

Containers

http://web.stanford.edu/class/cs106l/

Container: An object that allows us to collect other objects together and interact with them in some way.

Container: An object that allows us to collect other objects together and interact with them in some way.

Think of **vectors**, **stacks**, or **queues**!

Containers

http://web.stanford.edu/class/cs106l/

Why containers?

What is the purpose of container types in programming languages?

Why containers?

What is the purpose of container types in programming languages?

Organization

Related data can be packaged together!

Containers

http://web.stanford.edu/class/cs106l/

Why containers?

What is the purpose of container types in programming languages?

Organization

Related data can be packaged together!

Standardization

Common features are expected and implemented

Containers

http://web.stanford.edu/class/cs106l/

Why containers?

What is the purpose of container types in programming languages?

Organization

Related data can be packaged together!

Standardization

Common features are expected and implemented

Abstraction

Complex ideas made easier to utilize by clients

Motivating containers

We've been using the idea of a Student struct for the past few lectures:

```
struct Student {
    string name; // these are called fields
    string state; // separate these by semicolons
    int age;
};

Student s;
s.name = "Sarah";
s.state = "CA";
s.age = 21; // use . to access fields
```

Motivating containers

We've been using the idea of a Student struct for the past few lectures:

```
struct Student {
    string name; // these are called fields
    string state; // separate these by semicolons
    int age;
};

Student s;
s.name = "Sarah";
s.state = "CA";
s.age = 21; // use . to access fields
```

What if we had a whole class of students?

This is generalizable!

What if we need to store other types of data?

- Class grades
- Coordinates in a graph
- Mountains

What if we want to store it in a different way?

- FIFO vs LIFO
- Ascending order by some value

Containers

http://web.stanford.edu/class/cs106l/

Agenda

01. Defining Containers

What is a container in C++?

02. Containers in the STL vs Stanford

Types of containers and how they work

03. Container Adaptors

Abstracting container implementation

Containers

http://web.stanford.edu/class/cs106l/

The STL has many types of containers:

Both familiar:

- Vector
- Stack
- Queue
- Set
- Map

The STL has many types of containers:

Both familiar:

- Vector
- Stack
- Queue
- Set
- Map

And unfamiliar:

- Array
- Deque
- List
- Unordered set
- Unordered map

The STL has many types of containers:

Both familiar:

- Vector
- Stack
- Queue
- Set
- Map

And unfamiliar:

- Array
- Deque
- List
- Unordered set
- Unordered map

Not a Python list!

New containers

- An **array** is the primitive form of a vector
    - Fixed size in a strict sequence

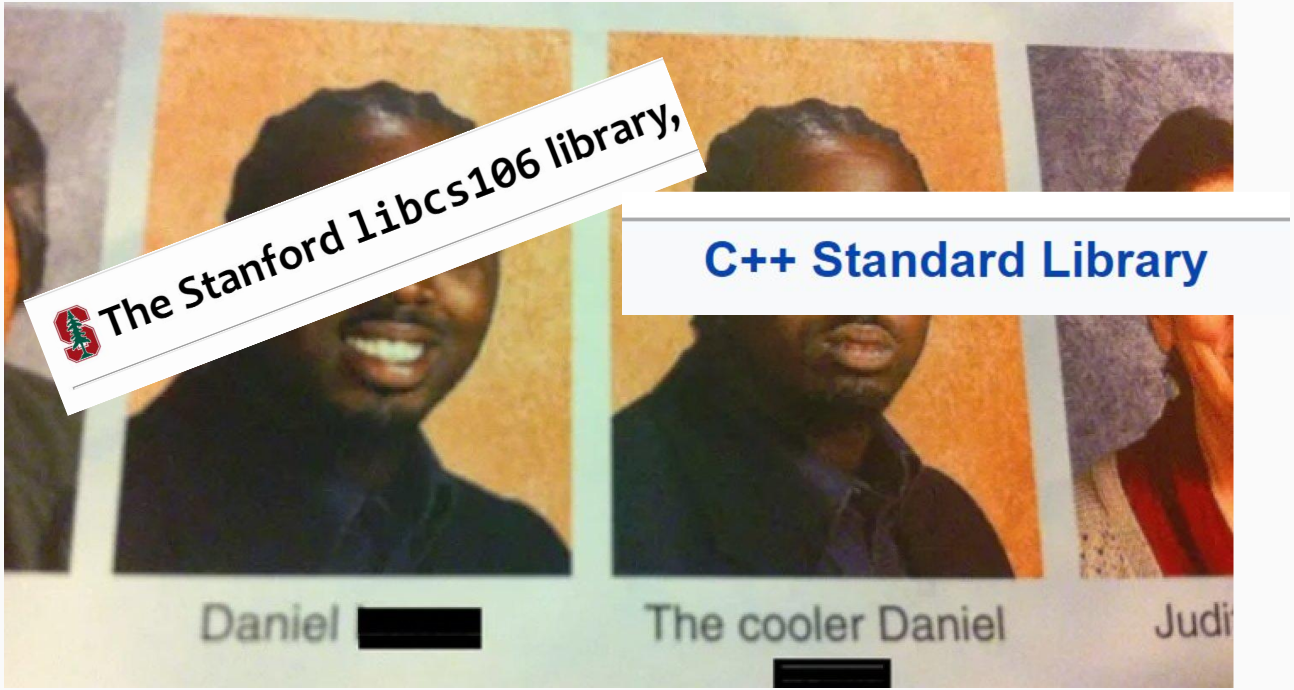
New containers

- An **array** is the primitive form of a vector
    - Fixed size in a strict sequence
- A **deque** is a **d**ouble **e**nded **que**ue

New containers

- An **array** is the primitive form of a vector
  - Fixed size in a strict sequence
- A **deque** is a **d**ouble **e**nded **que**ue
- A **list** is a doubly linked list
  - Can loop through in either direction!

STL vs Stanford



Containers

http://web.stanford.edu/class/cs106l/

STL vs Stanford

The Stanford library and the STL containers have very similar functionality, but there can sometimes be **key differences** in both behavior and syntax!

Containers

http://web.stanford.edu/class/cs106l/

Spot the difference!

What you want to do	Stanford `Vector<int>`	`std::vector<int>`
Create a new, empty vector	`Vector<int> vec;`	`std::vector<int> vec;`
Create a vector with `n` copies of 0	`Vector<int> vec(n);`	`std::vector<int> vec(n);`
Create a vector with `n` copies of a value `k`	`Vector<int> vec(n, k);`	`std::vector<int> vec(n, k);`
Add a value `k` to the end of a vector	`vec.add(k);`	`vec.push_back(k);`
Remove all elements of a vector	`vec.clear();`	`vec.clear();`
Get the element at index `i`	`int k = vec[i];`	`int k = vec[i];` (does **not** bounds check)
Check size of vector	`vec.size();`	`vec.size();`
Loop through vector by index `i`	`for (int i = 0; i < vec.size(); ++i) ...`	`for (std::size_t i = 0; i < vec.size(); ++i) ...`
Replace the element at index `i`	`vec[i] = k;`	`vec[i] = k;` (does **not** bounds check)

Table courtesy of Frankie Cerkvenik and Sathya Edamadaka!

Spot the difference!

What you want to do	Stanford `Vector<int>`	`std::vector<int>`
Create a new, empty vector	`Vector<int> vec;`	`std::vector<int> vec;`
Create a vector with **n** copies of 0	`Vector<int> vec(n);`	`std::vector<int> vec(n);`
Create a vector with **n** copies of a value **k**	`Vector<int> vec(n, k);`	`std::vector<int> vec(n, k);`
Add a value **k** to the end of a vector	`vec.add(k);`	`vec.push_back(k);`
Remove all elements of a vector	`vec.clear();`	`vec.clear();`
Get the element at index **i**	`int k = vec[i];`	`int k = vec[i];` (does **not** bounds check)
Check size of vector	`vec.size();`	`vec.size();`
Loop through vector by index **i**	`for (int i = 0; i < vec.size(); ++i) ...`	`for (std::size_t i = 0; i < vec.size(); ++i) ...`
Replace the element at index **i**	`vec[i] = k;`	`vec[i] = k;` (does **not** bounds check)

Table courtesy of Frankie Cerkvenik and Sathya Edamadaka!

Spot the difference!

What you want to do	Stanford `Vector<int>`	`std::vector<int>`
Create a new, empty vector	`Vector<int> vec;`	`std::vector<int> vec;`
Create a vector with `n` copies of 0	`Vector<int> vec(n);`	`std::vector<int> vec(n);`
Create a vector with `n` copies of a value `k`	`Vector<int> vec(n, k);`	`std::vector<int> vec(n, k);`
Add a value `k` to the end of a vector	`vec.add(k);`	`vec.push_back(k);`
Remove all elements of a vector	`vec.clear();`	`vec.clear();`
Get the element at index `i`	`int k = vec[i];`	`int k = vec[i];` (does **not** bounds check)
Check size of vector	`vec.size();`	`vec.size();`
Loop through vector by index `i`	`for (int i = 0; i < vec.size(); ++i) ...`	`for (std::size_t i = 0; i < vec.size(); ++i) ...`
Replace the element at index `i`	`vec[i] = k;`	`vec[i] = k;` (does **not** bounds check)

What does this mean?

Table courtesy of Frankie Cerkvenik and Sathya Edamadaka!

Safety vs Speed

In choosing a programming language, there's always a tradeoff between **speed**, **power**, and **safety**.

Safety vs Speed

In choosing a programming language, there's always a tradeoff between **speed**, **power**, and **safety**.

C++ is really fast! Why is that?

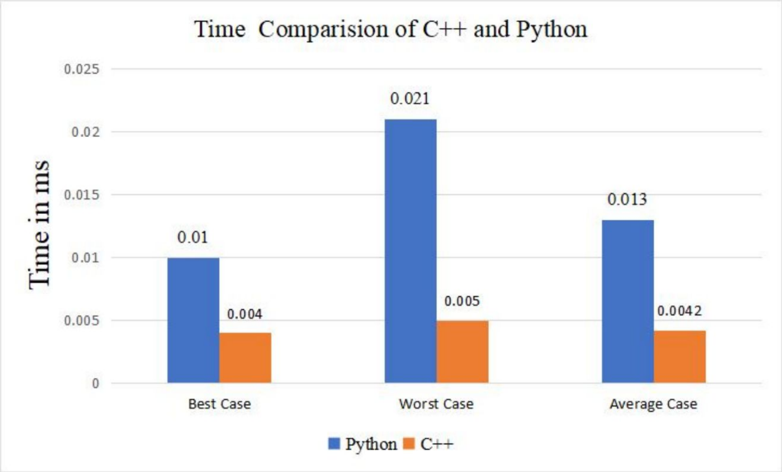


Fig. 13. Comparison of Time Utilization of Deletion Algorithm

C++ Design Philosophy

- Only provide the checks/safety nets that are necessary

C++ Design Philosophy

- Only provide the checks/safety nets that are necessary
- The programmer knows best!

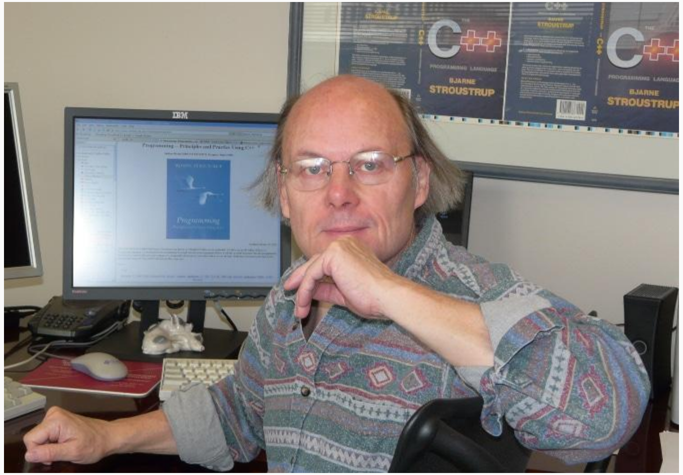
C++ Design Philosophy

- Only provide the checks/safety nets that are necessary
- The programmer knows best!

Making sure what you're doing is allowed is **your** job!

C++ Design Philosophy

- Only provide the checks/safety nets that are necessary
- The programmer knows best!



Making sure what you're doing is allowed is **your** job!

More differences

What you want to do	Stanford Set<int>	std::set<int>
Create an empty set	Set<int> s;	std::set<int> s;
Add a value **k** to the set	s.add(k);	s.insert(k);
Remove value **k** from the set	s.remove(k);	s.erase(k);
Check if a value **k** is in the set	if (s.contains(k)) ...	if (s.count(k)) ...
Check if vector is empty	if (vec.isEmpty()) ...	if (vec.empty()) ...

More differences

What you want to do	Stanford Map<int, char>	`std::map<int, char>`
Create an empty map	`Map<int, char> m;`	`std::map<int, char> m;`
Add key k with value v into the map	`m.put(k, v);` `m[k] = v;`	`m.insert({k, v});` `m[k] = v;`
Remove key k from the map	`m.remove(k);`	`m.erase(k);`
Check if key k is in the map	`if (m.containsKey(k)) ...`	`if (m.count(k)) ...`
Check if the map is empty	`if (m.isEmpty()) ...`	`if (m.empty()) ...`
Retrieve or overwrite value associated with key k *(**error** if key isn't in map)*	`Impossible (but does auto-insert)`	`char c = m.at(k);` `m.at(k) = v;`
Retrieve or overwrite value associated with key k *(**auto-insert** if key isn't in map)*	`char c = m[k];` `m[k] = v;`	`char c = m[k];` `m[k] = v;`

Containers

http://web.stanford.edu/class/cs106l/

There are two types of containers:

Sequence:

- Containers that can be accessed sequentially
- Anything with an inherent order goes here!

There are two types of containers:

Sequence:

- Containers that can be accessed sequentially
- Anything with an inherent order goes here!

Associative

- Containers that don't necessarily have a sequential order
- More easily searched
- Maps and sets go here!

There are two types of containers:

Sequence:

- Containers that can be accessed sequentially
- Anything with an inherent order goes here!

Associative

- Containers that don’t necessarily have a sequential order
- More easily searched
- Maps and sets go here!

All containers can hold all types of information! How do we choose which to use?

There are two types of containers:

Sequence:

- Containers that can be accessed sequentially
- Anything with an inherent order goes here!

Associative

- Containers that don't necessarily have a sequential order
- More easily searched
- Maps and sets go here!

All containers can hold all types of information! How do we choose which to use?

Vector implementation

How do vectors actually work?

Vector implementation

How do vectors actually work?

- At a high level, a vector is an **ordered** collection of elements of the **same type** that can grow and shrink in size.

Vector implementation

How do vectors actually work?

- At a high level, a vector is an **ordered** collection of elements of the **same type** that can grow and shrink in size.

Internally, vectors implement an array!

Vector implementation

How do vectors actually work?

- At a high level, a vector is an **ordered** collection of elements of the **same type** that can grow and shrink in size.

Internally, vectors implement an array!

Vector implementation

We keep track of a few member variables:

Vector implementation

We keep track of a few member variables:

- **_size** = number of elements in the vector

Vector implementation

We keep track of a few member variables:

- **_size** = number of elements in the vector
- **_capacity** = space allocated for elements

Vector implementation

We keep track of a few member variables:

- **_size** = number of elements in the vector
- **_capacity** = space allocated for elements

1	6	1	8	0	3		

Vector implementation

We keep track of a few member variables:

- **_size** = number of elements in the vector
- **_capacity** = space allocated for elements

1	6	1	8	0	3		

Don’t confuse these two!

So why can't we use vectors all the time?

Let's find out!

What about a deque?

Deques can be implemented many different ways! Here’s one:

What about a deque?

Deques can be implemented many different ways! Here's one:

- What if we had an array that stored other arrays?

What about a deque?

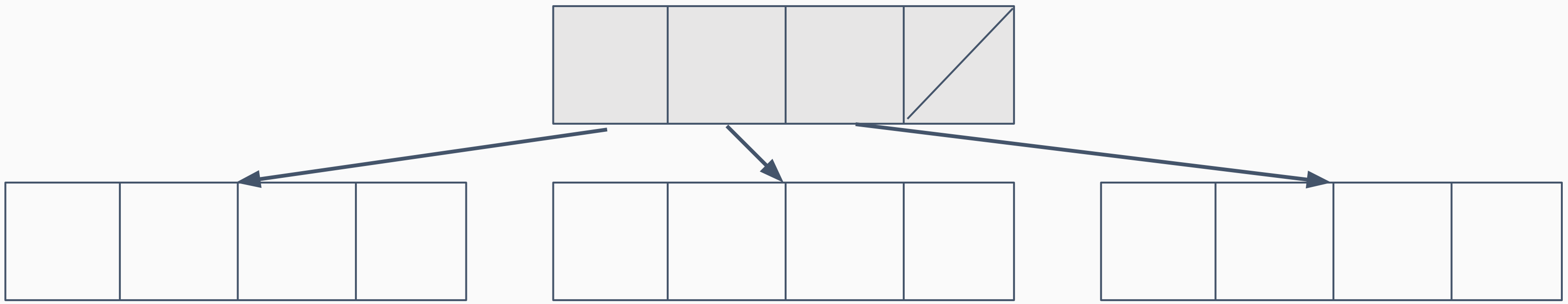
Deques can be implemented many different ways! Here's one:

- What if we had an array that stored other arrays?

What about a deque?

Deques can be implemented many different ways! Here's one:

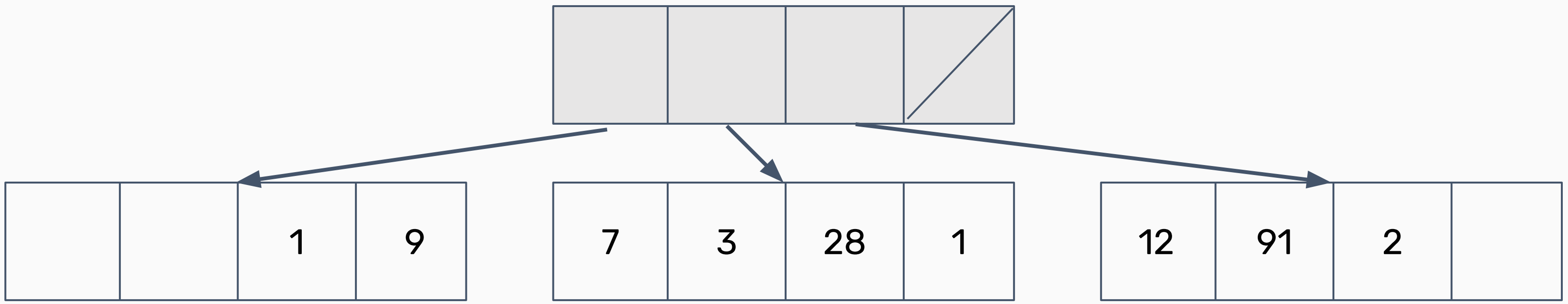
- What if we had an array that stored other arrays?



What about a deque?

Deques can be implemented many different ways! Here's one:

- What if we had an array that stored other arrays?



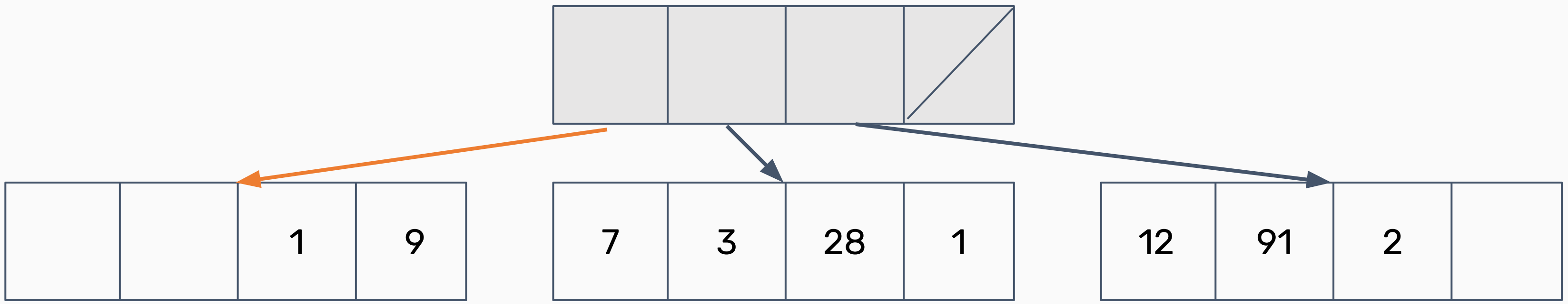
Containers

http://web.stanford.edu/class/cs106l/

What about a deque?

Deques can be implemented many different ways! Here's one:

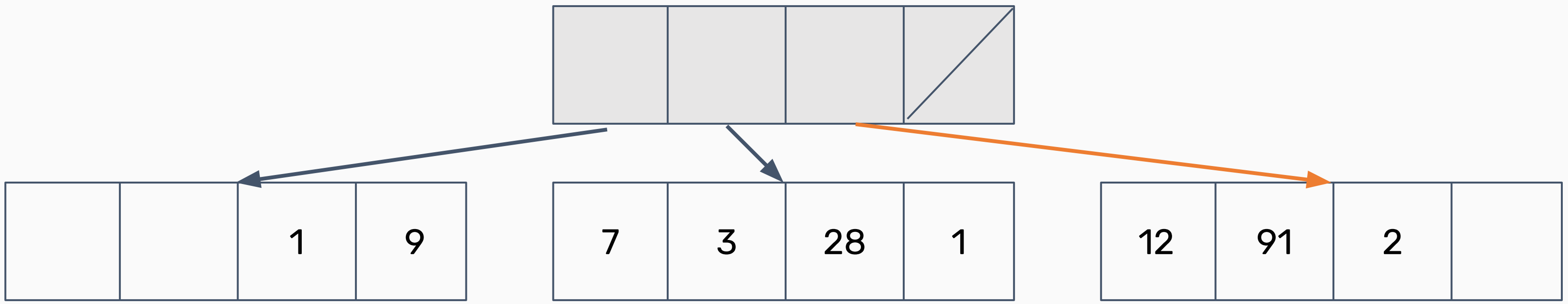
- What if we had an array that stored other arrays?



What about a deque?

Deques can be implemented many different ways! Here's one:

- What if we had an array that stored other arrays?



Choosing sequence containers

What you want to do	`std::vector`	`std::deque`	`std::list`
Insert/remove in the front	Slow	Fast	Fast
Insert/remove in the back	Super Fast	Very Fast	Fast
Indexed Access	Super Fast	Fast	Impossible
Insert/remove in the middle	Slow	Fast	Very Fast
Memory usage	Low	High	High
Combining (splicing/joining)	Slow	Very Slow	Fast
Stability* (iterators/concurrency)	Bad	Very Bad	Good

Sequence Containers: Summary

- Sequence containers are for when you need to enforce some order on your information!

Sequence Containers: Summary

- Sequence containers are for when you need to enforce some order on your information!
- Can usually use an **std::vector** for most anything

Sequence Containers: Summary

- Sequence containers are for when you need to enforce some order on your information!
- Can usually use an **std::vector** for most anything
- If you need particularly fast inserts in the front, consider an **std::deque**

Sequence Containers: Summary

- Sequence containers are for when you need to enforce some order on your information!
- Can usually use an **std::vector** for most anything
- If you need particularly fast inserts in the front, consider an **std::deque**
- For joining/working with multiple lists, consider an **std::list** (very rarely)

There are two types of containers:

Sequence:

- Containers that can be accessed sequentially
- Anything with an inherent order goes here!

Associative

- Containers that don't necessarily have a sequential order
- More easily searched
- Maps and sets go here!

All containers can hold all types of information! How do we choose which to use?

Map implementation

Maps are implemented with pairs! (**std::pair<const key, value>**)

Map implementation

Maps are implemented with pairs! (**std::pair<const key, value>**)

- Note the const! Keys must be immutable.

Map implementation

Maps are implemented with pairs! (**std::pair<const key, value>**)

- Note the const! Keys must be immutable.
- Why a pair and not a tuple?

Unordered maps/sets

Both maps and sets in the STL have an unordered version!

Unordered maps/sets

Both maps and sets in the STL have an unordered version!

- **Ordered** maps/sets require a **comparison operator** to be defined.
- **Unordered** maps/sets require a **hash function** to be defined.

Unordered maps/sets

Both maps and sets in the STL have an unordered version!

- **Ordered** maps/sets require a **comparison operator** to be defined.
- **Unordered** maps/sets require a **hash function** to be defined.

Simple types are already natively supported; anything else will need to be defined yourself.

Unordered maps/sets

Both maps and sets in the STL have an unordered version!

- **Ordered** maps/sets require a **comparison operator** to be defined.
- **Unordered** maps/sets require a **hash function** to be defined.

Simple types are already natively supported; anything else will need to be defined yourself.

Unordered maps/sets are usually faster than ordered ones!

Choosing associative containers

Lots of similarities between maps/sets! Broad tips:

Choosing associative containers

Lots of similarities between maps/sets! Broad tips:

- Unordered containers are **faster**, but can be difficult to get to work with nested containers/collections
- If using **complicated data types**/unfamiliar with hash functions, use an ordered container

Choosing associative containers

Lots of similarities between maps/sets! Broad tips:

- Unordered containers are **faster**, but can be difficult to get to work with nested containers/collections
- If using **complicated data types**/unfamiliar with hash functions, use an ordered container

So far:

- Sequence containers:
  - Arrays, vectors, deques, lists
- Associative containers:
  - Sets and maps
  - Unordered vs. ordered

Containers

http://web.stanford.edu/class/cs106l/

Agenda

01. Defining Containers

What is a container in C++?

02. Containers in the STL vs Stanford

Types of containers and how they work

03. Container Adaptors

Abstracting container implementation

Containers

http://web.stanford.edu/class/cs106l/

Container Adaptors

Container adaptors are “wrappers” to existing containers!

Container Adaptors

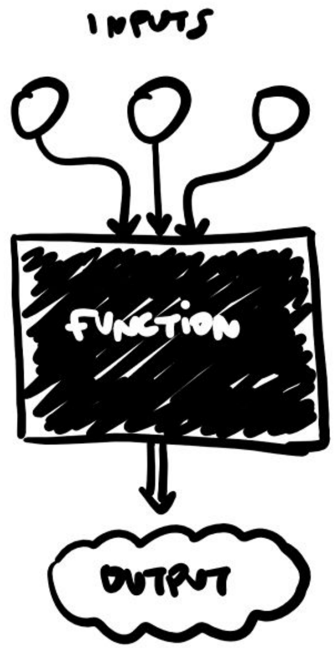
Container adaptors are “wrappers” to existing containers!

- Wrappers **modify the interface** to sequence containers and change what the client is allowed to do/how they can interact with the container.

Container Adaptors

Container adaptors are “wrappers” to existing containers!

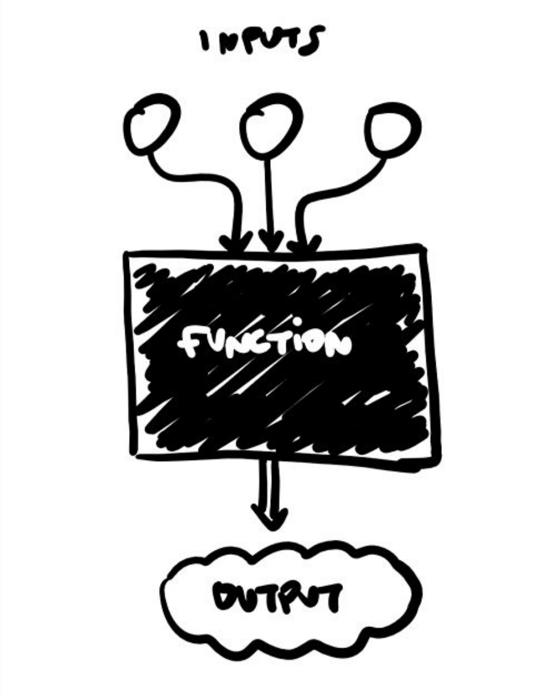
- Wrappers **modify the interface** to sequence containers and change what the client is allowed to do/how they can interact with the container.



Container Adaptors

Container adaptors are “wrappers” to existing containers!

- Wrappers **modify the interface** to sequence containers and change what the client is allowed to do/how they can interact with the container.
- How could we make a wrapper to implement a queue from a deque?



Let’s ask the STL!

```
template <class T, class Container = deque<T> > class queue;
```

queues are implemented as ***containers adaptors***, which are classes that use an encapsulated object of a specific container class as its ***underlying container***, providing a specific set of member functions to access its elements. Elements are ***pushed*** into the *"**back**"* of the specific container and ***popped*** from its *"**front**"*.

The underlying container may be one of the standard container class template or some other specifically designed container class. This underlying container shall support at least the following operations:

```
empty
size
front
back
push_back
pop_front
```

Containers

http://web.stanford.edu/class/cs106l/

Let's ask the STL!

```
template <class T, class Container = deque<T> > class queue;
```

queues are implemented as ***containers adaptors***, which are classes that use an encapsulated object of a specific container class as its ***underlying container***, providing a specific set of member functions to access its elements. Elements are ***pushed*** into the ***"back"*** of the specific container and ***popped*** from its ***"front"***.

The underlying container may be one of the standard container class template or some other specifically designed container class. This underlying container shall support at least the following operations:

```
empty
size
front
back
push_back
pop_front
```

Let’s ask the STL!

```
template <class T, class Container = deque<T> > class queue;
```

queues are implemented as ***containers adaptors***, which are classes that use an encapsulated object of a specific container class as its ***underlying container***, providing a specific set of member functions to access its elements. Elements are ***pushed*** into the ***"back"*** of the specific container and ***popped*** from its ***"front"***.

The underlying container may be one of the standard container class template or some other specifically designed container class. This underlying container shall support at least the following operations:

`empty`
`size`
`front`
`back`
`push_back`
`pop_front`

Containers

http://web.stanford.edu/class/cs106l/

Let's ask the STL!

```
template <class T, class Container = deque<T> > class queue;
```

queues are implemented as ***containers adaptors***, which are classes that use an encapsulated object of a specific container class as its ***underlying container***, providing a specific set of member functions to access its elements. Elements are ***pushed*** into the ***"back"*** of the specific container and ***popped*** from its ***"front"***.

The underlying container may be one of the standard container class template or some other specifically designed container class. This underlying container shall support at least the following operations:

```
empty
size
front
back
push_back
pop_front
```

```
std::queue<int> stack_deque;                    // Container = std::deque
std::queue<int, std::list<int>> stack_list;     // Container = std::list
```

Why?

Abstraction again!

Why?

Abstraction again!

- Commonly used data structures made easy for the client to use

Why?

Abstraction again!

- Commonly used data structures made easy for the client to use
- Can use different backing containers based on use type

Containers

http://web.stanford.edu/class/cs106l/

Summary

- Containers are ways to collect related data together and work with it logically
- Two types of containers: sequence and associative
- Container adaptors wrap existing containers to permit new/restrict access to the interface for the clients.

Containers

http://web.stanford.edu/class/cs106l/

Thanks!

Next up: Iterators and Pointers!