Midquarter Review

A review of (hopefully) everything you’ve learned

Today

- **Lambdas Code Demo**
- Review: Streams, Refs, Containers and Iterators!
- Announcements
- Review: All things classes!

Code Demo

Today

- ~~Lambdas Code Demo~~
- **Review: Streams, Refs, Containers and Iterators!**
- Announcements
- Review: All things classes!

Definition

stream: an abstraction for input/output. Streams convert between *data* and the *string representation of data.*

Output Streams

- Have type `std::ostream`
- Can only ***send*** data using the `<<` operator
    - Converts any type into string and ***sends*** it to the stream

- `std::cout` is the output stream that goes to the console

```
std::cout << 5 << std::endl;
// converts int value 5 to string "5"
// sends "5" to the console output stream
```

Output File Streams

- Have type `std::ofstream`
- Only receive data using the `<<` operator
  - Converts data of any type into a string and sends it to the **file stream**

- Must initialize your own `ofstream` object linked to your file

```
std::ofstream out("out.txt", std::ofstream::out);
// out is now an ofstream that outputs to out.txt

out << 5 << std::endl; // out.txt contains 5
```

Input Streams

- Have type `std::istream`
- Can only ***receive*** data using the `>>` operator
    - ***Receives*** a string from the stream and converts it to data

- `std::cin` is the output stream that gets input from the console

```
int x;
string str;
std::cin >> x >> str;
//reads exactly one int then 1 string from console
```

Nitty Gritty Details: `std::cin`

- First call to `std::cin <<` creates a command line prompt that allows the user to type until they hit enter
- Each `>>` ONLY reads until the next *whitespace*
  - Whitespace = tab, space, newline
- Everything after the first whitespace gets saved and used the next time `std::cin <<` is called
  - The place its saved is called a **buffer**!
- If there is nothing waiting in the buffer, `std::cin <<` creates a new command line prompt
- Whitespace is eaten: it won't show up in output

Input Streams: When things go wrong

```
int age; double hourlyWage;
cout << "Please enter your age: ";
cin >> age;
cout << "Please enter your hourly wage: ";
cin >> hourlyWage;
//what happens if first input is 2.17?
```

Stringstreams

Stringstreams

- Input stream: std::istringstream
    - Give any data type to the istringstream, it'll store it as a string!
- Output stream: std::ostringstream
    - Make an ostringstream out of a string, read from it word/type by word/type!
- The same as the other i/ostreams you've seen!

ostringstreams

```
string judgementCall(int age, string name,
                                          bool
                                       lovesCpp)
{
   std::ostringstream formatter;
   formatter << name <<", age " << age;
   if(lovesCpp) formatter << ", rocks.";
   else formatter << " could be better";
   return formatter.str();
}
```

istringstreams

```
Student reverseJudgementCall(string judgement)
{
    std::istringstream converter;
    string fluff; int age; bool lovesCpp; string name;
    converter >> name;
    converter >> fluff;
    converter >> age;
    converter >> fluff;
    string cool;
    converter >> cool;
    if(fluff == "rocks") return Student{name, age, "bliss"};
    else return Student{name, age, "misery"};
}
```

References

References to variables

```
vector<int> original{1, 2};
vector<int> copy = original;
vector<int>& ref = original;
original.push_back(3);
copy.push_back(4);
ref.push_back(5);

cout << original << endl;
cout << copy << endl;
cout << ref << endl;
```

References to variables

```
vector<int> original{1, 2};
vector<int> copy = original;
vector<int>& ref = original;
original.push_back(3);
copy.push_back(4);
ref.push_back(5);

cout << original << endl; // {1, 2, 3, 5}
cout << copy << endl;
cout << ref << endl;
```

References to variables

```
vector<int> original{1, 2};
vector<int> copy = original;
vector<int>& ref = original;
original.push_back(3);
copy.push_back(4);
ref.push_back(5);

cout << original << endl; // {1, 2, 3, 5}
cout << copy << endl;     // {1, 2, 4}
cout << ref << endl;
```

References to variables

```
vector<int> original{1, 2};
vector<int> copy = original;
vector<int>& ref = original;
original.push_back(3);
copy.push_back(4);
ref.push_back(5);

cout << original << endl; // {1, 2, 3, 5}
cout << copy << endl;     // {1, 2, 4}
cout << ref << endl;      // {1, 2, 3, 5}
```

References to variables

```
vector<int> original{1, 2};
vector<int> copy = original;
vector<int>& ref = original;
original.push_back(3);
copy.push_back(4);
ref.push_back(5);

cout << original << endl; // {1, 2, 3, 5}
cout << copy << endl;     // {1, 2, 4}
cout << ref << endl;      // {1, 2, 3, 5}
```

"=" automatically makes a copy! Must use & to avoid this.

The classic reference-copy bug:

```
void shift(vector<std::pair<int, int>>& nums) {
   for (size_t i = 0; i < nums.size(); ++i) {
      auto [num1, num2] = nums[i];
      num1++;
      num2++;
   }
}
```

This creates a copy of the course

This is updating that same copy!

The classic reference-copy bug, fixed:

```
void shift(vector<std::pair<int, int>>& nums) {
    for (auto& [num1, num2]: nums) {
        num1++;
        num2++;
    }
}
```

The classic reference-rvalue error

```
void shift(vector<std::pair<int, int>>& nums) {
    for (auto& [num1, num2]: nums) {
        num1++;
        num2++;
    }
}

shift({{1, 1}});
```

The classic reference-rvalue error

```
void shift(vector<std::pair<int, int>>& nums) {
    for (auto& [num1, num2]: nums) {
        num1++;
        num2++;
    }
}

shift({{1, 1}});
// {{1, 1}} is an rvalue, it can’t be referenced
```

Definition: l-values vs r-values

- **l-values** can appear on the **left** or **right** of an =
- `x` is an **l-value**

```
int x = 3;
int y = x;
```

l-values have names

l-values are <u>**not temporary**</u>

Definition: l-values vs r-values

- l-values can appear on the **left** or **right** of an =
- `x` is an l-value

```
int x = 3;
int y = x;
```

l-values have names

l-values are **not temporary**

- r-values can ONLY appear on the **right** of an =
- `3` is an r-value

```
int x = 3;
int y = x;
```

r-values don't have names

r-values are **temporary**

The classic reference-rvalue error, fixed

```
void shift(vector<pair<int, int>>& nums) {
    for (auto& [num1, num2]: nums) {
        num1++;
        num2++;
    }
}
auto my_nums = {{1, 1}};
shift(my_nums);
```

`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 and Iterators!

Types of containers

All containers can hold almost all elements.

Sequence Containers

- Simple
- Adaptors

Associative Containers

- Ordered
- Unordered

<> vector	(adding + removing elements at end)
⇅ deque	(adding + removing elements anywhere but end)
↓ list	(add/remove elements anywhere, no random access)
() tuple	(different data types, but immutable)

Stanford “Vector” vs STL “vector”

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)

When to use which sequence container?

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`

When to use which sequence container?

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

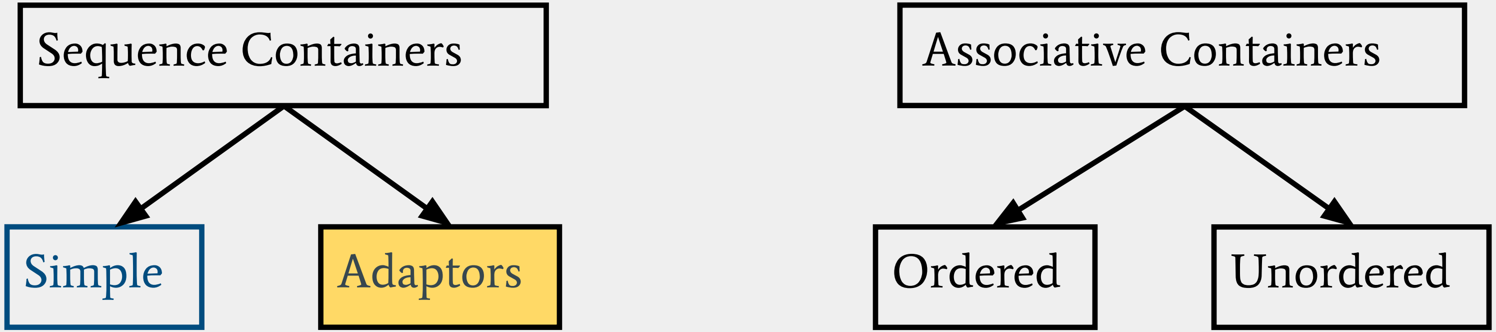
Container Adaptors

What is a container adaptor?

`std::stack` and `std::queue`

Types of containers

All containers can hold almost all elements.



<> vector

⇅ deque

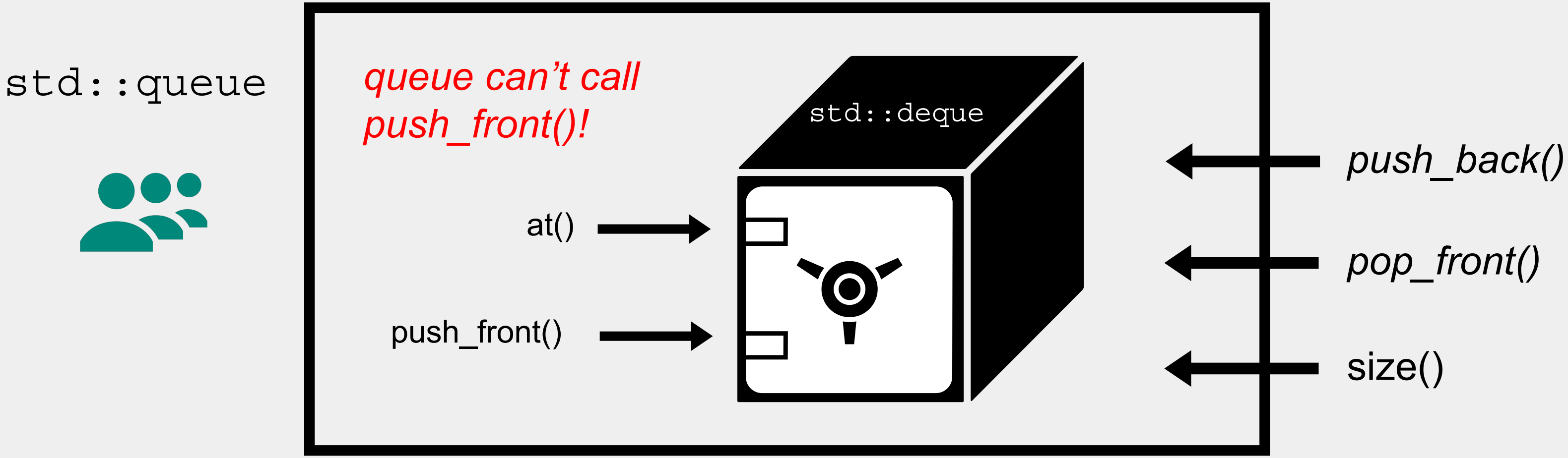
↓ list

() tuple

stack	(adding/removing elements from the front)
queue	(adding elements from the front, removing from the back)
priority_queue	(adding elements with a priority, always removing the highest priority-element)

Container adaptors are wrappers in C++!

- Container adaptors provide a different interface for sequence containers.
- You can choose what the underlying container is!
- For instance, let's choose a deque as our underlying container, and let's implement a queue!



std::stack and std::queue

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.

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 LIFO (last-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 stack pushes and pops the element from the back of the underlying container, known as the top of the stack.

Concrete examples with `std::queue`

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.

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

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

std::queue<int, std::vector<int>> stack_vector; //Container = std::vector?
```

Concrete examples with `std::queue`

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.

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

~~`std::queue<int, std::vector<int>> stack_vector; //Container = std::vector?`~~

removing from the front of a vector is slow!

Some member functions of `std::queue`

Member functions

(constructor)	constructs the queue (public member function)
(destructor)	destructs the queue (public member function)
`operator=`	assigns values to the container adaptor (public member function)

Element access

`front`	access the first element (public member function)
`back`	access the last element (public member function)

Capacity

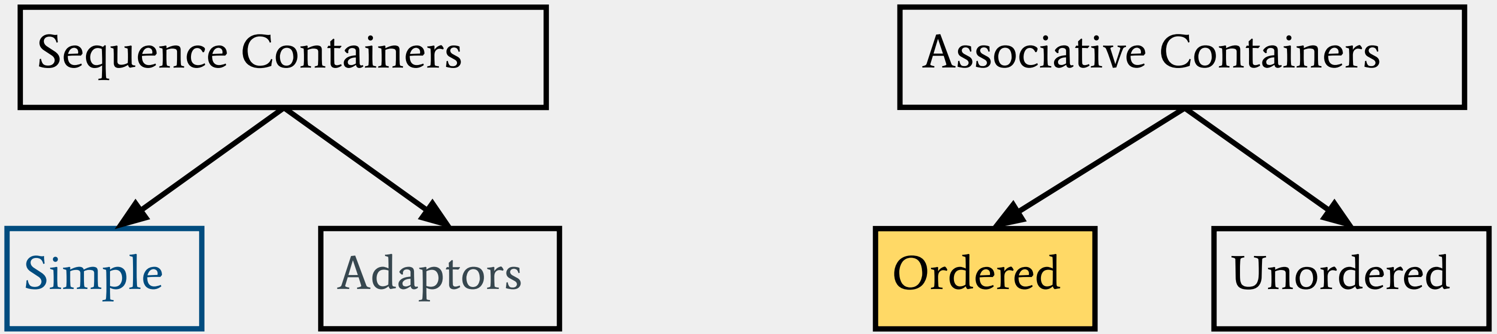
`empty`	checks whether the underlying container is empty (public member function)
`size`	returns the number of elements (public member function)

Modifiers

`push`	inserts element at the end (public member function)
`emplace` (C++11)	constructs element in-place at the end (public member function)
`pop`	removes the first element (public member function)
`swap` (C++11)	swaps the contents (public member function)

Types of containers

All containers can hold almost all elements.



<> vector
⇅ deque
↓ list
() tuple

stack
queue
priority_queue

{} set (unique elements)
{:} map (key value pairs)

Stanford “Set” vs STL “set”

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()) ...`

Stanford "Map" vs STL "map"

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;`

STL Iterators

- Iterators are objects that point to elements inside containers.
- Each STL container has its own iterator, but all of these iterators exhibit a similar behavior!
- Generally, STL iterators support the following operations:

```
std::set<type> s = {0, 1, 2, 3, 4};
std::set::iterator iter = s.begin();                    // at 0
++iter;                                                 // at 1
*iter;                                                  // 1
(iter != s.end());                                      // can compare
iterator equality
auto second_iter = iter;                          // "copy construction"
```

a quick tip:

Why ++iter and not iter++?

Answer: *++iter returns the value after being incremented!*
iter++ returns the previous value and then increments it. (wastes just a bit of time)

Looping over collections

```
std::set<int> set{3, 1, 4, 1, 5, 9};
for (auto iter = set.begin(); iter != set.end(); ++iter) {
  const auto& elem = *iter;
  cout << elem << endl;
}

std::map<int> map{{1, 6}, {1, 8}, {0, 3}, {3, 9}};
for (auto iter = map.begin(); iter != map.end(); ++iter) {
  const auto& [key, value] = *iter;    // structured binding!
  cout << key << ":" << value << ", " << endl;
}
```

Looping over collections

```
std::set<int> set{3, 1, 4, 1, 5, 9};
for (const auto& elem : set) {
 cout << elem << endl;
}


std::map<int> map{{1, 6}, {1, 8}, {0, 3}, {3, 9}};
for (const auto& [key, value] : map) {
 cout << key << ":" << value << ", " << endl;
}
```

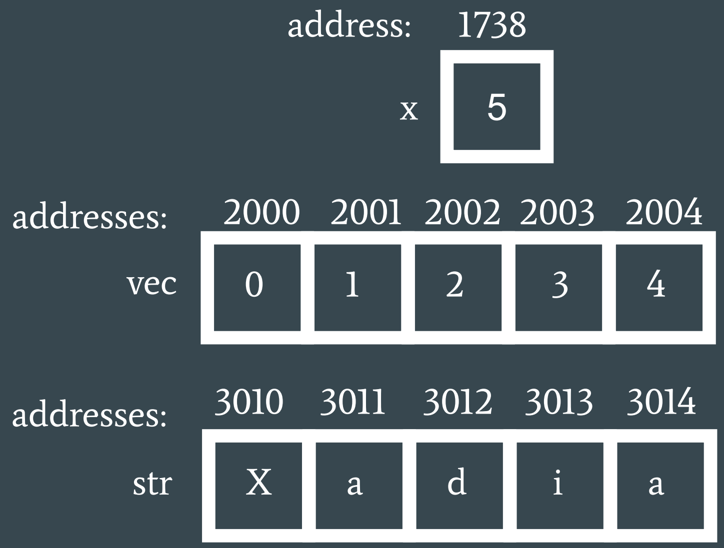
Pointers

- When variables are created, they're given an address in memory.
- Pointers are objects that store an address and type of a variable.

adding a "&" before a variable returns its address, just like passing *by reference*!

```
void f(int& x) ...
```

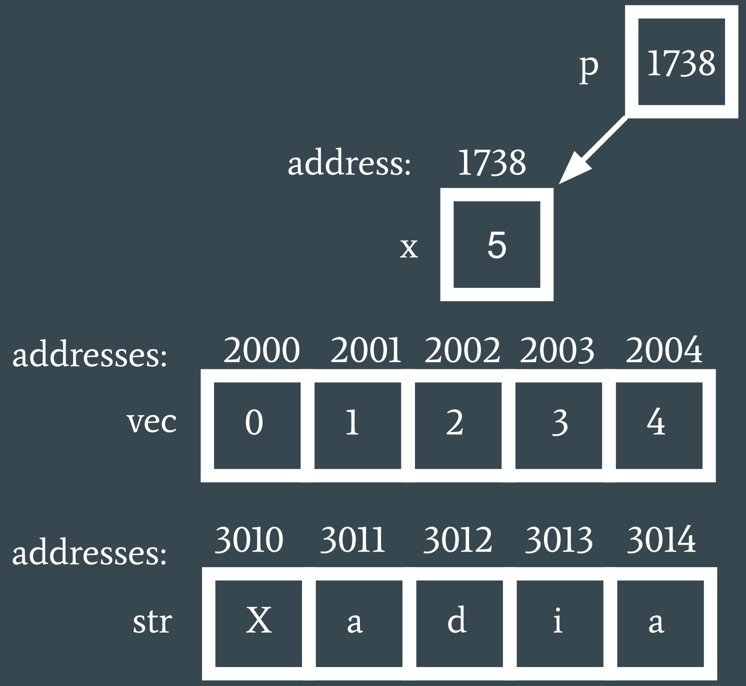
```
int* p = &x;
int* q = &vec[0];
char* r = &str[0];
```



Pointers

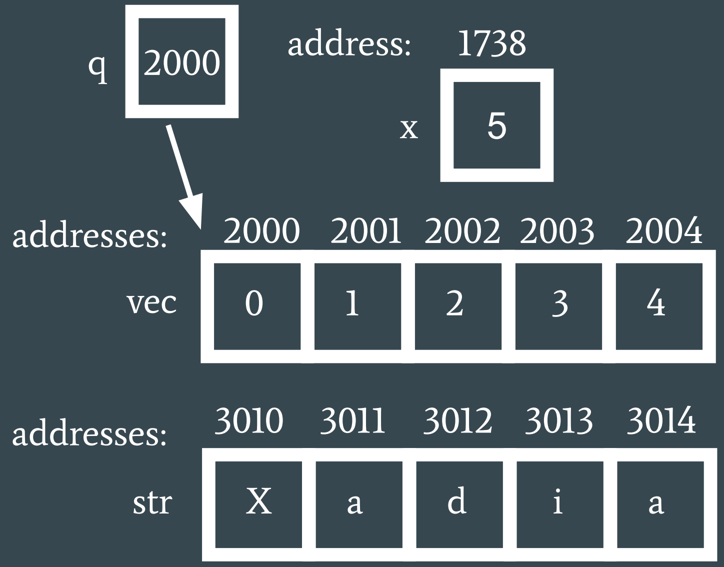
- When variables are created, they're given an address in memory.
- Pointers are objects that store an address and type of a variable.

```
int* p = &x;
int* q = &vec[0];
char* r = &str[0];
```



Pointers

- When variables are created, they're given an address in memory.
- Pointers are objects that store an address and type of a variable.




```
int* p = &x;
int* q = &vec[0];
char* r = &str[0];
```

Pointers

```
int x = 5;
int* pointerToInt = &x;                                        // creates pointer to int
cout << *pointerToInt << endl;                                 // 5

std::pair<int, int> pair = {1, 2};                             // creates pair
std::pair<int, int>* pointerToPair = &pair;                    // creates pointer to
pair
cout << (*pair).first << endl;                                 // 1
cout << pair->first << endl;                                   // 1
```

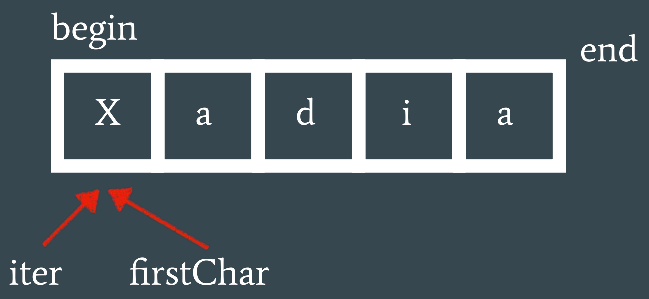
- To get the value of a pointer, we can *dereference* it (get the object *reference*d by the pointer)
- A shorthand for dereferencing a pointer and then accessing a member variable (doing **someObject.variableName**) is using the -> operator.

Pointers vs. Iterators

- Iterators are a form of pointers!
- Pointers are more generic iterators
  - can point to any object, not just elements in a container!

```
std::string lands = "Xadia";
// iterator
auto iter = lands.begin();

// syntax for a pointer. don't worry
about the specifics if you're in 106B!
they'll be discussed in the latter half
of the course.
char* firstChar = &lands[0];
```



Today

- ~~Lambdas Code Demo~~
- ~~Review: Streams, Refs, Containers and Iterators!~~
- **Announcements**
- Review: All things classes!

Announcements!

- Assignment 2 has been released and is due Oct 30
- Fill out the midquarter feedback form and get an extra late day!
- Four more weeks of content!

https://docs.google.com/forms/d/e/1FAIpQLSey2uIWidObNH3P9VhX_jwqGiLAWrAh-vPDZsURZmiayLTDkg/viewform?usp=sf_link

Today

- ~~Lambdas Code Demo~~
- ~~Review: Streams, Refs, Containers and Iterators!~~
- ~~Announcements~~
- **Review: All things classes!**

Definition

Class: A programmer-defined custom type. An abstraction of an object or data type.

Turning `Student` into a class: basic components

//student.h

```
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
};
```

Public section:

- Users of the Student object can directly access anything here!
- Defines **interface** for interacting with the private member variables!

Private section:

- Usually contains all member variables
- Users can't access or modify anything in the private section

//student.cpp

```
#include student.h
std::string Student::getName(){
    return name; //we can access name here!
}
void Student::setName(string name){
    this->name = name; //resolved!
}
int Student::getAge(){
    return age;
}
 void Student::setAge(int age){
    //We can define what "age" means!
    if(age >= 0){
        this -> age = age;
    }
    else error("Age cannot be negative!");
}
```

//student.h

```
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
 };
```

Constructors

- Define how the member variables of an object is initialized
- What gets called when you first create a Student object
- Overloadable!

//student.cpp

```
#include student.h
Student::Student(){...}
Student::Student(string name, int age, string state){
    this->name = name;
    this->age = age;
    this->state = state;
}
```

Putting it all together: Using your shiny new class!

//main.cpp

```
#include student.h
int main(){
    Student frankie;
    frankie.setName("Frankie");
    frankie.setAge(21);
    frankie.setState("MN");
    cout << frankie.getName() << " is from " << frankie.getState() <<
endl;
}
```

Putting it all together: Using your shiny new class!

//main.cpp

```
#include student.h
int main(){
    Student frankie;
    frankie.setName("Frankie");
    frankie.setAge(21);
    frankie.setState("MN");
    cout << frankie.getName() << " is from " << frankie.getState();

    Student sathya("Sathya", 20, "New Jersey");
    cout << sathya.getName() << " is from " << sathya.getState();
}
```

One last thing... Arrays

- Arrays are a primitive type! They are the building blocks of all containers
- Think of them as lists of objects of **fixed size** that you can **index into**
- **Think of them as the struct version of vectors. You should not be using them in application code! Vectors are the STL interface for arrays!**

```
//int * is the type of an array variable
int *my_int_array;

//this is how you initialize an array
my_int_array = new int[10];
//this is how you index into an array
int one_element = my_int_array[0];
```

One last thing... Arrays

```
//int * is the type of an array variable
int *my_int_array;
//my_int_array is a pointer!

//this is how you initialize an array
my_int_array = new int[10];
                   +--+--+--+--+--+--+--+--+--+--+
//my_int_array ->  |  |  |  |  |  |  |  |  |  |  |
                   +--+--+--+--+--+--+--+--+--+--+
//this is how you index into an array
int one_element = my_int_array[0];
```

Destructors

- Arrays are memory **WE** allocate, so we need to give instructions for when to deallocate that memory!
- When we are done using our array, we need to `delete []` it!

```
//int * is the type of an array variable
int *my_int_array;

//this is how you initialize an array
my_int_array = new int[10];
//this is how you index into an array
int one_element = my_int_array[0];
delete [] my_int_array;
```

Destructors

- `delete`ing (almost) always happens in the **destructor** of a class!
- The destructor is defined using Class_name::~Class_name()
- No one ever explicitly calls it! Its called when Class_name object go out of scope!
- Just like all member functions, declare it in the .h and implement in the .cpp!

The problem with StrVector

- Vectors should be able to contain any data type!

~~Solution? Create IntVector, DoubleVector, BoolVector etc..~~

- What if we want to make a vector of `Student`s?
  - How are we supposed to know about every custom class?
- What if we don't want to write a class for every type we can think of?

SOLUTION: Template classes!

Writing a Template Class: Syntax

//mypair.h

```
template<class First, class Second> class MyPair {
    public:
        First getFirst();
        Second getSecond();

        void setFirst(First f);
        void setSecond(Second f);
    private:
        First first;
        Second second;
};
```

Use generic typenames as placeholders!

Implementing a Template Class: Syntax

//mypair.cpp

```
#include "mypair.h"

First MyPair::getFirst(){
    return first;
}
//Compile error! Must announce every member function is templated :/
```

Implementing a Template Class: Syntax

//mypair.cpp

```
#include "mypair.h"

template<class First, typename Second>
First MyPair::getFirst(){
    return first;
}
//Compile error! The namespace of the class isn't just MyPair
```

Implementing a Template Class: Syntax

//mypair.cpp

```
#include “mypair.h”

template<class First, typename Second>
First MyPair<First, Second>::getFirst(){
    return first;
}
```

Implementing a Template Class: Syntax

//mypair.cpp

```
#include "mypair.h"

template<class First, typename Second>
First MyPair<First, Second>::getFirst(){
    return first;
}

template<class Second, typename First>
Second MyPair<First, Second>::getSecond(){
    return second;
}
```

Member Types

- Sometimes, we need a name for a type that is dependent on our template types
- Recall: iterators

```
std::vector a = {1, 2};
std::vector::iterator it = a.begin();
```

Member Types

- Sometimes, we need a name for a type that is dependent on our template types
- Recall: iterators

```
std::vector a = {1, 2};
std::vector::iterator it = a.begin();
```

- iterator is a **member type** of vector

Member Types: Syntax

```
//vector.h
template<typename T> class vector {
    using iterator = …  // something internal

    private:
    iterator front;
}
```

Member Types: Syntax

```
//vector.h
template<typename T> class vector {
    using iterator = …  // something internal

    private:
    iterator front;
}
```

```
//vector.cpp
template <typename T>
iterator vector<T>::begin() {...}
//compile error! Why?
```

Member Types: Syntax

```
//vector.h
template<typename T> class vector {
    using iterator = …  // something internal

    private:
    iterator front;
}
```

```
//vector.cpp
template <typename T>
iterator vector<T>::insert(iterator pos, int value) {...}
//iterator is a nested type in namespace vector<T>::
```

Member Types: Syntax

```
//vector.h
template<typename T> class vector {
    using iterator = …  // something internal

    private:
    iterator front;
}
```

```
//vector.cpp
template <typename T>
typename vector<T>::iterator vector<T>::insert(iterator pos, int
value) {...}
```

Aside: Type Aliases

- You can use `using type_name = type` in application code as well!
- When using it in a class interface, it defines a nested type, like `vector::iterator`
- When using it in application code, like main.cpp, it just creates another name for `type` within that scope (until the next unmatched })

Member Types: Summary

- Used to make sure your clients have a standardized way to access important types.
- Lives in your namespace: **vector<T>::iterator**.
- After class specifier, you can use the alias directly (e.g. inside function arguments, inside function body).
- Before class specifier, use **typename.**

One final compile error....

```
// vector.h
template <typename T>
class vector<T> {
    T at(int i);
};
```

```
// vector.cpp
#include "vector.h"
template <typename T>
void vector<T>::at(int i) {
    // oops
}
```

```
// main.cpp
#include "vector.h"
vector<int> a;
a.at(5);
```

```
g++ -c vector.cpp main.cpp
g++ vector.o main.o -o output
```

Templates don’t emit code until instantiated

What the C++ compiler does with non-template classes

```
// main.cpp
#include "vectorint.h"
vectorInt a;
a.at(5);
```

1. `g++ -c vectorint.cpp main.cpp`: Compile and create all the code in vectorint.cpp and main.cpp. All the functions in vectorint.h have implementations that have been compiled now, and main can access them because it included vectorint.h
2. "Oh look she used vectorInt::at, sure glad I compiled all that code and can access vectorInt::at right now!"

What the C++ compiler does with template classes

```
// main.cpp
#include “vector.h”
vector a;
a.at(5);
```

1. `g++ -c vector.cpp main.cpp`: Compile and create all the code in main.cpp. Compile vector.cpp, but since it’s a template, don’t create any code yet.
2. “Oh look she made a vector<int>! Better go generate all the code for one of those!”
3. “Oh no! All I have access to is vector.h! There’s no implementation for the interface in that file! And I can’t go looking for vector<int>.cpp!”

The fix...

```
// vector.h
template <typename T>
class vector<T> {
    T at(int i);
};
```

```
// vector.cpp
#include "vector.h"
template <typename T>
void vector<T>::at(int i) {
    // oops
}
```

```
// main.cpp
#include "vector.h"
vector<int> a;
a.at(5);
```

```
g++ -c vector.cpp main.cpp
g++ vector.o main.o -o output
```

Include vector.cpp in vector.h!

```cpp
// vector.h
#include "vector.h"
template <typename T>
class vector<T> {
    T at(int i);
};
```

```cpp
// vector.cpp

template <typename T>
void vector<T>::at(int i) {
    // oops
}
```

```cpp
// main.cpp
#include "vector.h"
vector<int> a;
a.at(5);
```

```
g++ -c main.cpp
g++ vector.o main.o -o output
```

What the C++ compiler does with template classes

```
// main.cpp
#include "vector.h"
vector a;
a.at(5);
```

1. `g++ -c vector.cpp main.cpp`: Compile and create all the code in main.cpp. Compile vector.cpp, but since it's a template, don't create any code yet.
2. "Oh look she made a vector<int>! Better go generate all the code for one of those!"
3. "vector.h includes all the code in vector.cpp, which tells me how to create a vector<int>::at function :)"

Recap: Template classes

- Add `template<class T1, T2..>` before class definition in .h
- Add `template<class T1, T2..>`before all function signatures in .cpp
- When returning nested types (like iterator types), put `typename ClassName<T1, T2..>::member_type` as return type, not just `member_type`
- Templates don't emit code until instantiated, so `#include` the .cpp file in the .h file, not the other way around!

Const and Classes

Recall: `Student` class

//student.h

```
class Student {
    public:
    std::string getName();
    void setName(string name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
};
```

//student.cpp

```cpp
#include student.h
std::string Student::getName(){
    return name; //we can access name here!
}
void Student::setName(string name){
    this->name = name; //resolved!
}
int Student::getAge(){
    return age;
}
 void Student::setAge(int age){
    //We can define what “age” means!
    if(age >= 0){
        this -> age = age;
    }
    else error("Age cannot be negative!");
}
```

//student.h

```cpp
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
 };
```

Using a `const Student`

```
//main.cpp
std::string stringify(const Student& s){
    return s.getName() + " is " + std::to_string(s.getAge) +
                                                " years old." ;
}
//compile error!
```

Using a `const Student`

//main.cpp

```
std::string stringify(const Student& s){
    return s.getName() + " is " + std::to_string(s.getAge) +
                                            " years old." ;
}
//compile error!
```

- The compiler doesn't know `getName` and `getAge` don't modify s!
- We need to promise that it doesn't by defining them as **const functions**
- Add `const` to the **end** of function signatures!

Making `Student` const-correct

```
//student.cpp
#include student.h
std::string Student::getName() const{
    return name;
}
void Student::setName(string name){
    this->name = name;
}
int Student::getAge() const{
    return age;
}
 void Student::setAge(int age){
    if(age >= 0){
        this -> age = age;
    }
    else error("Age cannot be
negative!");
}
```

```
//student.h
class Student {
    public:
    std::string getName() const;
    void setName(string name);
    int getAge const();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
 };
```

Definition

const-interface: All member functions marked `const` in a class definition. Objects of type `const ClassName` may only use the const-interface.

Making RealVector's const-interface

```
class StrVector {
public:
    using iterator = std::string*;
    const size_t kInitialSize = 2;
    /*...*/
    size_t size();
    bool empty();
    std::string& at(size_t indx);
    void insert(size_t pos, const std::string& elem);
    void push_back(const std::string& elem);

    iterator begin();
    iterator end();
    /*...*/
```

Making `RealVector`'s const-interface

```
class StrVector {
public:
    using iterator = std::string*;
    const size_t kInitialSize = 2;
    /*...*/
    size_t size() const;
    bool empty() const;
    std::string& at(size_t indx);
    void insert(size_t pos, const std::string& elem);
    void push_back(const std::string& elem);

    iterator begin();
    iterator end();
    /*...*/
```

Should `begin()` and `end()` be `const`?

Consider a function with a const `RealVector` param...

```
void printVec(const RealVector& vec){
    cout << "{ ";
    for(auto it = vec.begin(); it != vec.end(); ++it){
        cout << *it << endl;
    }
    cout << " }" << endl;
}
```

These seem like reasonable calls! Let's mark them const. What could go wrong? :)

Consider a function with a const `RealVector` param...

```
void printVec(const RealVector& vec){
    cout << "{ ";
    for(auto it = vec.begin(); it != vec.end(); ++it){
        *it = "dont mind me modifying a const vector :D";
    }
    cout << " }" << endl;
}
```

This code will compile! begin() and end() don't explicitly change vec, but they give us an iterator that can!

Consider a function with a const `RealVector` param...

```
void printVec(const RealVector& vec){
    cout << "{ ";
    for(auto it = vec.begin(); it != vec.end(); ++it){
        *it = "dont mind me modifying a const vector :D";
    }
    cout << " }" << endl;
}
```

Problem: we need a way to iterate through a const vec just to access it

Solution: `cbegin()` and `cend()`

```
class StrVector {
public:
    using iterator = std::string*;
    using const_ iterator = const std::string*;
    /*...*/
    size_t size() const;
    bool empty() const;
    /*...*/
    void push_back(const std::string& elem);
    iterator begin();
    iterator end();
    const_iterator begin()const;
    const_iterator end()const;
    /*...*/
```

Consider a function with a const `RealVector` param...

```
void printVec(const RealVector& vec){
    cout << "{ ";
    for(auto it = vec.cbegin(); it != vec.cend(); ++it){
        cout << *it << cout;
    }
    cout << " }" << cout;
}
```

Fixed! And now we can't set *it equal to something: it will be a compile error!

const iterator vs const_iterator: Nitty Gritty

```
using iterator = std::string*;
using const_iterator = const std::string*;

const iterator it_c = vec.begin(); //string * const, const ptr to non-const obj
*it_c = "hi"; //OK! it_c is a const pointer to non-const object
it_c++; //not ok! cant change where a const pointer points!


const_iterator c_it = vec.cbegin(); //const string*, a non-const ptr to const obj
c_it++; // totally ok! The pointer itself is non-const
*c_it = "hi" // not ok! Can't change underlying const object
cout << *c_it << endl; //allowed! Can always read a const object, just can't change

//const string * const, const ptr to const obj
const const_iterator c_it_c = vec.cbegin();
cout << c_it_c << " points to " << *c_it_c << endl; //only reads are allowed!
```

Recap: Const and Const-correctness

- Use const parameters and variables wherever you can in application code
- Every member function of a class that doesn't change its member variables should be marked `const`
- auto will drop all const and &, so be sure to specify
- Make iterators and const_iterators for all your classes!
  - **`const iterator`** = cannot increment the iterator, can dereference and change underlying value
  - **`const_iterator`** = can increment the iterator, cannot dereference and change underlying value
  - **`const const_iterator`** = cannot increment iterator, cannot dereference and change underlying value

Recap: Template classes

- Add `template<typename T1, typename T2..>` before class definition in .h
- Add `template<typename T1, typename T2..>` before all function signature in .cpp
- When returning nested types (like iterator types), put `typename ClassName<T1, T2..>::member_type` as return type, not just `member_type`
- Templates don't emit code until instantiated, so `#include` the .cpp file in the .h file, not the other way around!