

# Lecture 10: Classes I

CS 106L, Fall '20

# Warmup

In breakout groups, walk through the .h and .cpp files of our vector class. Collectively identify:

- things you've seen before, and that you can understand why they make sense in this context
- things you've seen before, but for which you're not sure why they make sense in this context
- things you haven't seen before

# CS 106B covers the **barebones** of C++ classes

we'll be covering the rest

template classes • const-correctness • operator overloading  
special member functions • move semantics • RAII

# Class Design – Today

- File Organization
- Template Classes
- Type Aliases
- Member Templates

# MQE feedback

- **Speaking too fast**
  - we will take a bit more time—please feel free to ask us to slow down or repeat stuff in the chat!
- **Going to 60 min for more challenging topics**
  - we will do that!

# MQE feedback

**Please** continue filling out the MQE—it's due tonight!

- 1 late day
- We get to learn more about how you feel
- Win-win

# Assignment 1

- Four people have already submitted :O
- Come to **office hours**
  - **Ethan:** Sat 1-2 p.m. PST, plus by appointment
  - **Nikhil:** Mon 1-2 p.m. PST, Sun 7 a.m. PST
  - Plus extra office hours this Thursday, Feb 18 at 6 AM PST

Submitted ↕

Mon, Feb 15, 2021 12:23 AM

Tue, Feb 9, 2021 9:28 PM

Mon, Feb 15, 2021 5:24 PM

Mon, Feb 15, 2021 12:09 AM

# Review: Classes



# Classes and structs are almost the same

## Structs



User-defined type



Store multiple values



**Public** by default

## Classes



User-defined type



Store multiple values



**Private** by default

**all other differences are *solely by convention***

	public	private
variable	<b>NO</b> <i>don't do it</i>	<b>private members</b> internal state variables
method	<b>public methods</b> methods part of interface, called by client	<b>private methods</b> internal helper methods

# Why not public variables?

Control the way info is **accessed** and **edited**.

```
class Student {  
    public:  
        int phone;  
        int age;  
        int birthYear() { return currentYear() - age; }  
}
```

```
Student jaxon(6507234000, 5);  
jaxon.age = -5;                // oh no  
jaxon.birthYear() // 2025      // oh noooo
```



Interface should be separate from implementation. (So it can be changed.)

# Before we start: namespaces

- Put code into logical **groups**, to avoid name clashes
- Each class has its own namespace
- Syntax for calling/using something in a namespace:

```
namespace_name::name
```

# File Organization

```
// vector.h  
class vector {  
    void at(int i);  
};
```

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

```
// vector.cpp  
#include "vector.h"  
void vector::at(int i) {  
    // do something  
}
```

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

# **Live Code Demo:**

## Vector.cpp (review classes)

# Template Classes

# Template classes allow for flexibility

## Before

```
class intvector {  
    int& get(int index);  
}  
  
class stringvector {  
    string& get(int index);  
}
```

## After

```
template<typename T>  
class vector {  
    T& get(int index);  
}  
  
vector<int> vs;  
vector<string> vs;
```



# Location of <typename T>

```
// vector.h
```

```
template <typename T>
class vector {
    public:
        value_type& front();
        value_type& back();
}
```

```
// vector.cpp
```

```
template <typename T>
TYPE vector<T>::front() { }
```

```
template <typename T>
TYPE vector<T>::back() { }
```

```
// we'll explain what goes in
TYPE soon
```

# **Live Code Demo:** Template Vector.cpp

# **5-min detour: File organization**

No need to memorize this, but important for future coding

# This doesn't work

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

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

```
// 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);
```

# Why?

Recall: **templates don't emit any code until instantiated**

```
// vector.cpp
#include "vector.h"
void vector<T>::at(int i) {
    // doesn't emit anything
}
```

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

can't instantiate  
**vector<int>::at**  
bc no access to .cpp!

# Resolving this

- We see that **main.cpp** needs to include the file with the method definition (not just the header).
- We can either put the **method body** in the **.h file** (common, but we won't do this...)
- or **include the .cpp file as part of the .h instead**



# How this looks in practice

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

#include "vector.cpp"
```

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

```
// vector.cpp
void vector<T>::at(int i) {
    // this works
}
```

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

# Recap: Template Classes

- Syntax is similar to template functions (typename T)
- **Include .cpp in .h, rather than the other way around**



 **Questions?** 

# Member Types

# Type Aliases

Give a type another name:

```
using another_name = existing_type;  
using charvctr = std::vector<char>;
```

# Member Types

Use a type alias to store a **dependent** type related to our class

```
class vector {  
    using iterator = ... // something internal  
}
```

```
vector::iterator front;
```

```
namespace mycollection {  
    class vector {  
    public:  
        // Type aliases  
        using value_type = int;  
        using iterator = int*;  
    }  
}
```

# Using Member Types

Trying to return an **iterator** out of our function:

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



Does anyone know  
why this might not  
work?

# Using Member Types

Can't use a **vector::iterator** before we've declared we're in **vector**!

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

}
```

```
In file included from main.cpp:1:
In file included from ./vector.h:57:
./vector.cpp:72:1: error: unknown type name 'iterator'
iterator vector<T>::insert(iterator pos, const value_type& value) {
^
```

# Using Member Types

Compiler error, but the compiler tells us how to fix it!

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

}
```

In file included from ./vector.h:57:  
./vector.cpp:72:1: **error:** missing 'typename' prior to dependent type name 'vector<T>::iterator'  
vector<T>::iterator vector<T>::insert(iterator pos, const value\_type& value) {  
~~~~~  
typename

# Using Member Types

Compiler error, but the compiler tells us how to fix it!

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

}
```

```
In file included from ./vector.h:57:
./vector.cpp:72:1: error: missing 'typename' prior to dependent type name 'vector<T>::iterator'
vector<T>::iterator vector<T>::insert(iterator pos, const value_type& value) {
^~~~~~
typename
```



# Why `value_type`?

Why have a separate type alias **`value_type`** that points to **`T`**? (Hint: think about dicts!)

- Not all data types store the type that is their template parameter!

# 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**.

```
typename vector<T>::iterator iter = vec.begin();
```

 **Questions?** 

# Where we're going

```
void print_size(const vector<int>& vec) {  
    cout << vec.size() << endl;  
} // doesn't compile
```

```
void print_front(vector<int>& vec) {  
    cout << vec[0] << endl;  
} // doesn't compile
```

```
void print_front(vector<int> vec) {  
    // something  
} // causes a memory leak and a crash
```

1) **const-correctness**

2) **operator  
overloading**

3) **copy semantics**