

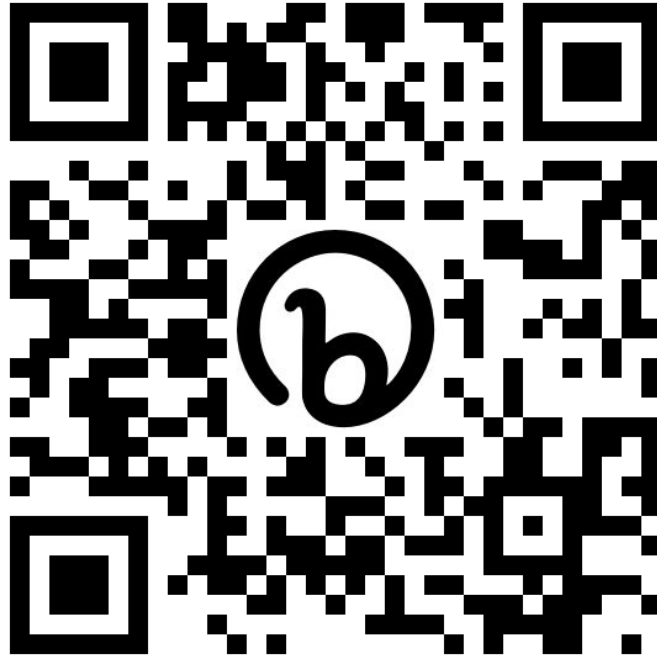
# **CS106L Lecture 8:**

# **Template Classes**

**Fall 2023**

Fabio Ibanez, Haven Whitney

# Attendance

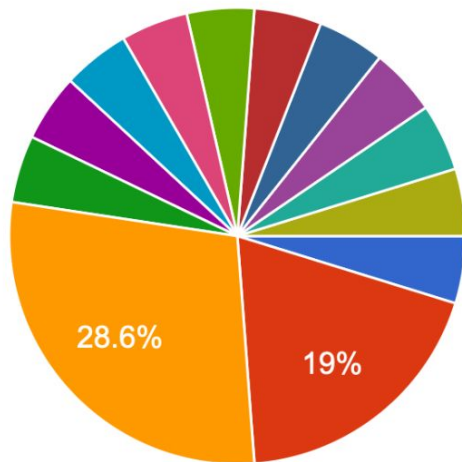


<https://bit.ly/templatesF23>

# Anime!

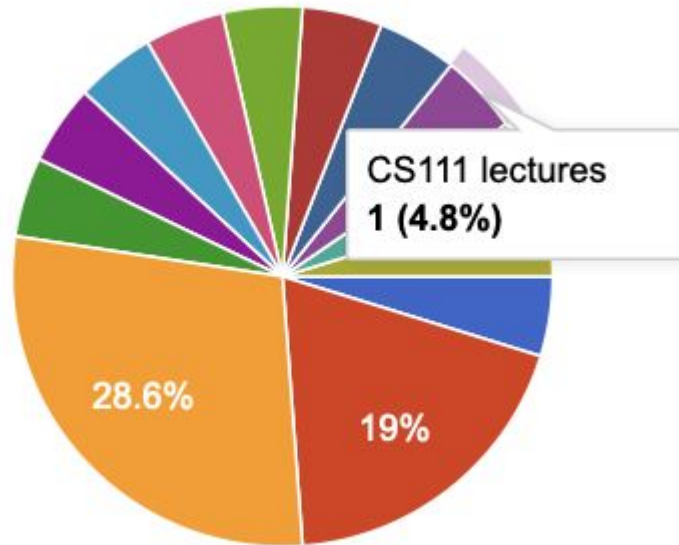
What's your favorite anime? If you don't have a favorite anime, what's your favorite show?

21 responses



- One piece 🐉
- Attack on Titan
- Naruto
- Dragonball Z
- One punch man
- The Mandalorian
- Your lie in April
- Lucky Star

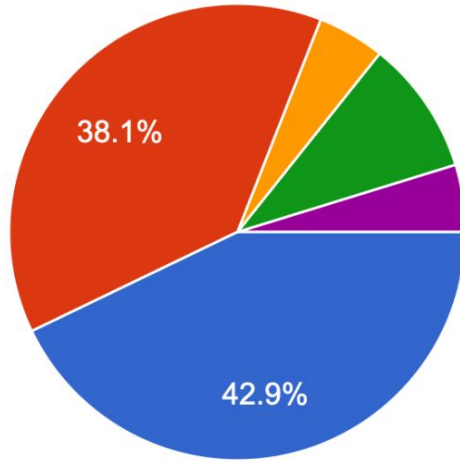
# CS111 🥲



# People are here!

Which describes your current status?

21 responses



- Present and caffeinated!! ☕
- Here, but I would like caffeine 😓
- My cat is attending on my behalf 🐱
- Physically here, mentally in Hogwarts ✨
- omg is this real life

81% of you are here! 🙌

# Plan

1. A quick recap on classes
2. Template classes
3. Const correctness

# A point on classes

```
class Point {  
public:  
    Point(int x, int y, std::string color);  
    int getX();  
    int getY();  
    std::string getColor();  
    void setX(int x);  
    void setY(int y);  
    void setColor(std::string color);  
  
private:  
    int x;  
    int y;  
    std::string color;  
};
```

```
#include "Point.hh"  
  
Point::Point(int x, int y, const std::string &color) {  
    // ?  
}  
  
int Point::getX() {  
    // ?  
}  
  
int Point::getY() {  
    // ?  
}  
  
std::string Point::getColor() {  
    // ?  
}  
  
void Point::setX(int x) {  
    // ?  
}  
  
void Point::setY(int y) {  
    // ?  
}  
  
void Point::setColor(std::string &color) {  
    // ?  
}
```

# The importance of this



```
std::string Point::getColor() {  
    return color;  
}
```



```
std::string Point::getColor() {  
    return this->color;  
}
```



# The importance of this



```
std::string Point::getColor() {  
    return color;  
}
```



```
std::string Point::getColor() {  
    return this->color;  
}
```

The same



# The importance of this



```
void Point::setX(int x) {  
    x = x;  
}
```



```
void Point::setX(int x) {  
    this->x = x;  
}
```

# The importance of this



```
void Point::setX(int x) {  
    x = x;  
}
```



```
void Point::setX(int x) {  
    this->x = x;  
}
```

Not the same ❌

# What C++ thinks about this



Naming conflict

```
void Point::setX(int x) {  
    x = x;  
}
```



Ahh you mean this point

```
void Point::setX(int x) {  
    this->x = x;  
}
```

In the `setX` function we're changing the member-variable value of `x` based on some passed in value.

# What C++ thinks about this



## Naming conflict

```
void Point::setX(int x) {  
    x = x;  
}
```



Ahh you mean this point

```
void Point::setX(int x) {  
    this->x = x;  
}
```

In `getX` we're not modify anything, we're just *getting* some value.

# What C++ thinks about this

```
Point.hh

class Point {
public:
    Point(int x, int y, std::string color);
    int getX();
    int getY();
    std::string getColor();
    void setX(int x);
    void setY(int y);
    void setColor(std::string color);

private:
    int x;
    int y;
    std::string color;
};
```

```
Point.cpp

#include "Point.hh"
Point::Point(int x, int y, std::string &color) {
    this->x = x;
    this->y = y;
    this->color = color;
}

int Point::getX() {
    return x;
}

int Point::getY() {
    return y;
}

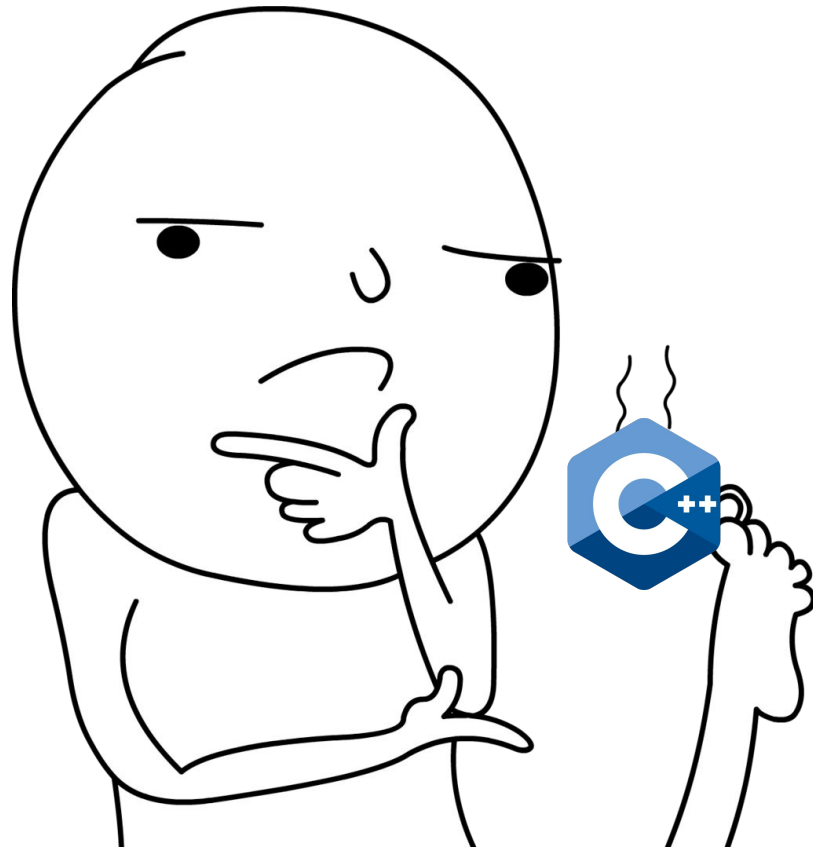
std::string Point::getColor() {
    return color;
}

void Point::setX(int x) {
    this->x = x;
}

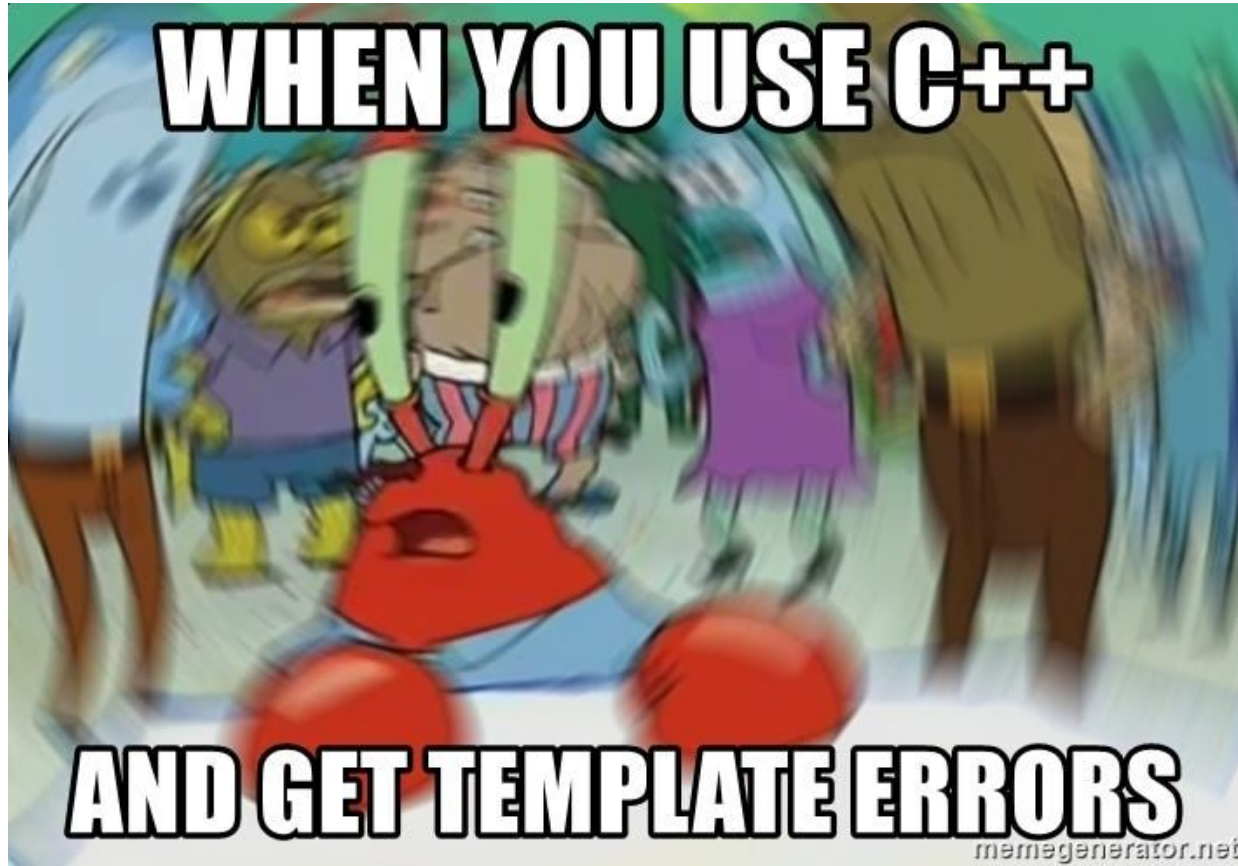
void Point::setY(int y) {
    this->y = y;
}

void Point::setColor(std::string &color) {
    this->color = color;
}
```

# What questions do we have?



# Template Classes





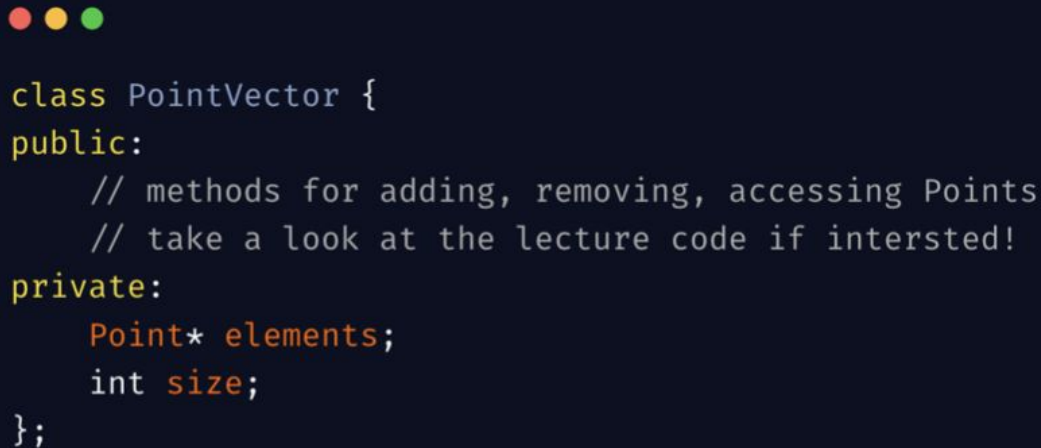
# Plan

- ~~1. A quick recap on classes~~
2. Template classes
3. Const correctness

# Motivation

You were working at a graphics startup in 1998, and you needed a simple data structure to manage **Point** objects.

Wallah, with your knowledge of pointer arithmetic and classes you create this **PointVector**:



```
class PointVector {  
public:  
    // methods for adding, removing, accessing Points  
    // take a look at the lecture code if interested!  
private:  
    Point* elements;  
    int size;  
};
```

# Motivation

**You realized that you also need to keep track of** `bool`, `char`, `int`, `short`, `long`, `long long`, `unsigned int`, `unsigned char`, `unsigned short`, `unsigned long`, `unsigned long long`, `float`, `double`, `long double`, `std::size_t`, `ptrdiff_t`, `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `char16_t`, `char32_t`, `wchar_t`, `void`, `nullptr`, and finally, `enum` types,

**That's A LOT of types!**



# Templates entered the chat



**Template Class:** A class that is parametrized over some number of types; it is comprised of member variables of a general type/types.

# TL;DR

We can now  
allow classes to  
be general, and  
not redundant!



# Solved!

**You realized that you also need to keep track of** `bool`, `char`, `int`, `short`, `long`, `long long`, `unsigned int`, `unsigned char`, `unsigned short`, `unsigned long`, `unsigned long long`, `float`, `double`, `long double`, `std::size_t`, `ptrdiff_t`, `int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `char16_t`, `char32_t`, `wchar_t`, `void`, `nullptr`, and finally, `enum` types,

**That's A LOT of types!**



# A {template} container .h



```
class IntContainer {  
public:  
    IntContainer(int val);  
    int getValue();  
  
private:  
    int value;  
};
```



```
template <typename T>  
class Container {  
public:  
    Container (T val);  
    T getValue();  
  
private:  
    T value;  
};
```



# A {template} container .h



```
class IntContainer {  
public:  
    IntContainer(int val);  
    int getValue();  
  
private:  
    int value;  
};
```

Only for ints



```
template <typename T>  
class Container {  
public:  
    Container (T val);  
    T getValue();  
  
private:  
    T value;  
};
```

A template



# A {template} container .h

```
template <typename T>
```


This is a **template declaration** and allows us to create template classes

```
template <typename T>
class Container {
public:
    Container (T val);
    T getValue();

private:
    T value;
};
```

A template

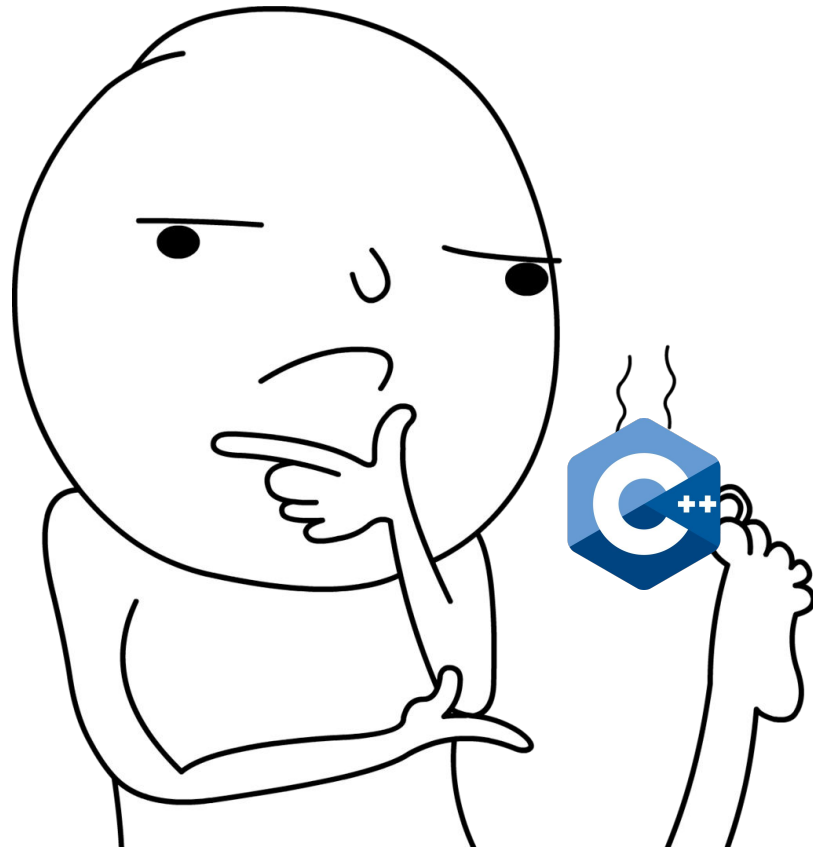
# Template declarations



```
template <typename T, typename U>
```

This is a **template declaration** list which can have various template parameters representing different types.

# What questions do we have?



# A {template} container .cpp

```

#include "IntContainer.hh"

IntContainer::IntContainer(int val) {
    this->value = val;
}

int IntContainer::getValue() {
    return value;
}
```

```

#include "Container.hh"

template <class T>
Container<T>::Container(T val) {
    this->value = val;
}

template <typename T>
T Container<T>::getValue() {
    return value;
}
```

# A {template} container .cpp

```
#include "IntContainer.hh"

IntContainer::IntContainer(int val) {
    this->value = val;
}

int IntContainer::getValue() {
    return value;
}
```

Only for ints

```
#include "Container.hh"

template <class T>
Container<T>::Container(T val) {
    this->value = val;
}

template <typename T>
T Container<T>::getValue() {
    return value;
}
```

A template

# Template functions

When doing our implementation for our template classes, we need to create **template functions**

```
#include "Container.hh"

template <class T>
Container<T>::Container(T val) {
    this->value = val;
}

template <typename T>
T Container<T>::getValue() {
    return value;
}
```

A template

# What's the difference here?

```
● ● ●  
  
#include "Container.hh"
```

```
template <class T>  
Container<T>::Container(T val) {  
    this->value = val;  
}
```

```
template <typename T>  
T Container<T>::getValue() {  
    return value;  
}
```

```
● ● ●  
  
#include "Container.hh"
```

```
template <class T>  
Container::Container(T val) {  
    this->value = val;  
}
```

```
template <typename T>  
T Container::getValue() {  
    return value;  
}
```

# What's the difference here?

```
#include "Container.hh"
```

```
template <class T>
```

```
Container<T>::Container(T val) {  
    this->value = val;  
}
```

```
template <typename T>
```

```
T Container<T>::getValue() {  
    return value;  
}
```

```
#include "Container.hh"
```

```
template <class T>
```

```
Container::Container(T val) {  
    this->value = val;  
}
```

```
template <typename T>
```

```
T Container::getValue() {  
    return value;  
}
```



# What's the difference here?

```
#include "Container.hh"
```

```
template <class T>
```

```
Container<T>::Container(T val) {  
    this->value = val;  
}
```

```
template <typename T>
```

```
T Container<T>::getValue() {  
    return value;  
}
```

On the right we  
don't pass in  
the template  
parameter in  
the class  
namespace

```
#include "Container.hh"
```

```
template <class T>
```

```
Container::Container(T val) {  
    this->value = val;  
}
```

```
template <typename T>
```

```
T Container::getValue() {  
    return value;  
}
```

# Why is this important? 🤔

C++ wants us to specify our template parameters in our namespace because, based on the parameters our class may behave differently!

Note: there is no "one" Container, there is one for an int, bool, char, etc., there is one for an int, bool, char, etc.

```
#include "Container.hh"

template <class T>
Container<T>::Container(T val) {
    this->value = val;
}

template <typename T>
T Container<T>::getValue() {
    return value;
}
```

# Why is this important? 🤔



```
Container::
```



```
Container<T>::
```

# Some syntax stuff



```
template <typename T>
class Container {
public:
    Container (T val);
    T getValue();

private:
    T value;
};
```

**class** and **type name** are interchangeable in the template declaration list



```
#include "Container.hh"

template <class T>
Container<T>::Container(T val) {
    this->value = val;
}

template <typename T>
T Container<T>::getValue() {
    return value;
}
```

# Some syntax stuff

```
template <typename T>
class Container {
public:
    Container (T val);
    T getValue();

private:
    T value;
};
```

**class** and **type name** are interchangeable in the template declaration list

```
#include "Container.hh"

template <class T>
Container<T>::Container(T val) {
    this->value = val;
}

template <typename T>
T Container<T>::getValue() {
    return value;
}
```

# Another quirk of templates

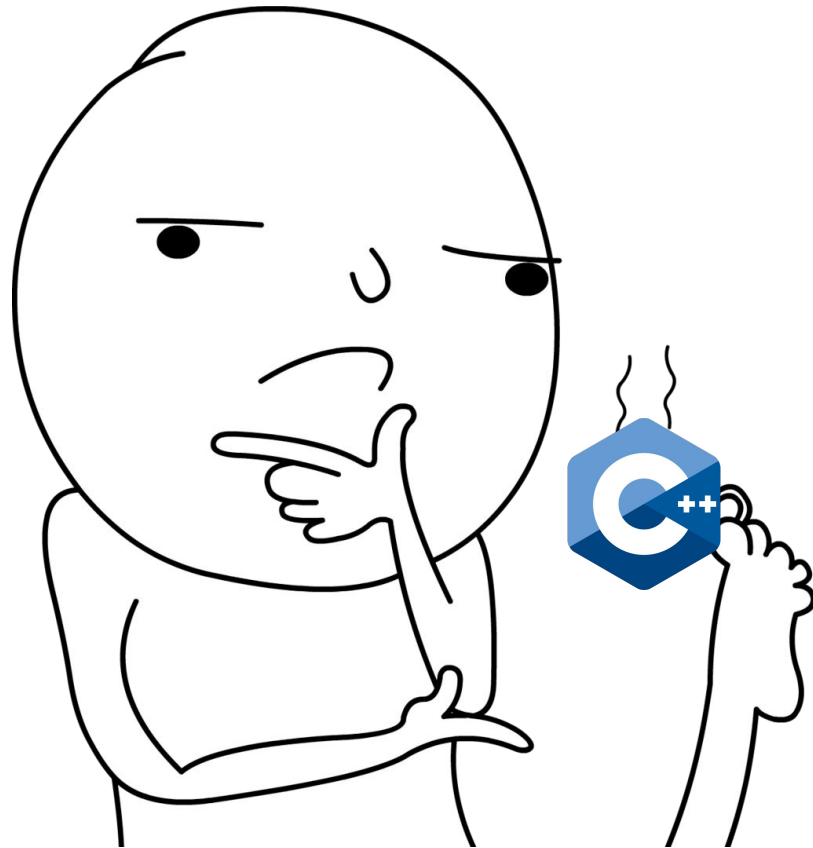
When making template classes you *need* to **#include** the .cpp implementation in the .h file.

# Another quirk of templates

When making template classes you *need* to `#include` the .cpp implementation in the .h file.

This is a compiler quirk — not super important for this class, but just keep it in mind when working with templates.

# What questions do we have?



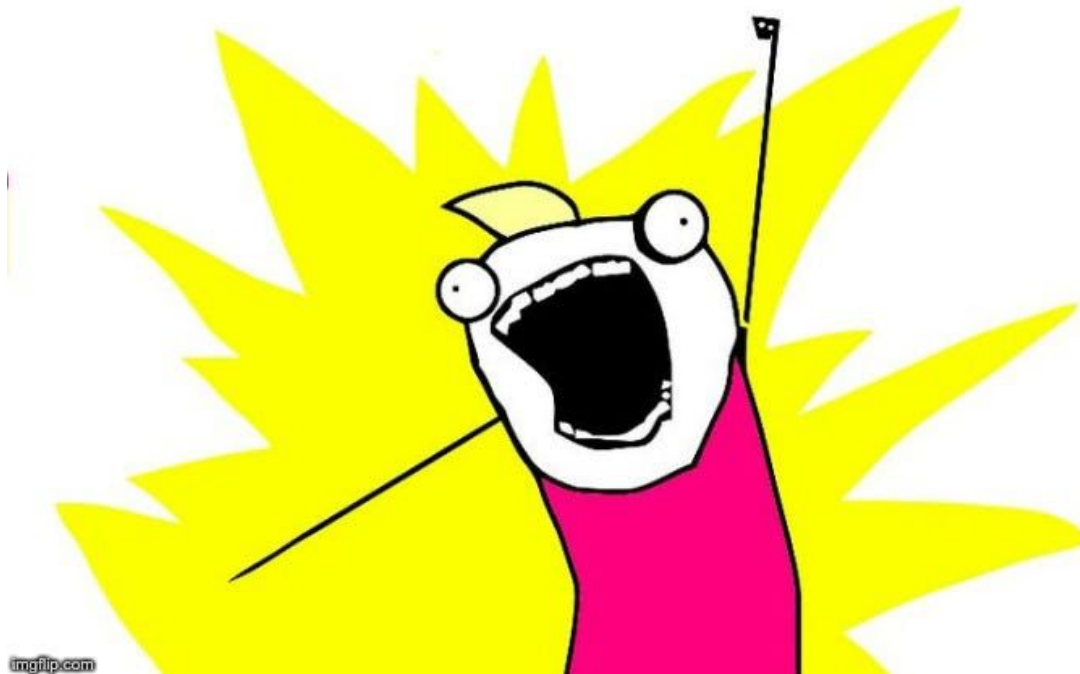


# Plan

- ~~1. A quick recap on classes~~
- ~~2. Template classes~~
3. Const correctness

# Const Correctness!

**COMPILE-TIME ALL THE THINGS!!**



# Const Correctness!

We saw a few weeks ago that `const` is a qualifier for objects that states they cannot be changed!

# Const Correctness!

We saw a few weeks ago that `const` is a qualifier for objects that states they cannot be changed!

But const correctness is much more than that!

# Remember our student class

## .h file

```
class Student {  
Private:  
    /// An example of type aliasing  
    using String = std::string;  
    String name;  
    String state;  
    int age;  
  
public:  
    Student(String name, String state, int age);  
    /// Added a 'setter' function  
    void setName(String name);  
    String getName();  
    String getState();  
    int getAge();  
}
```

# Remember our student class

## .h file

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

# Remember our student class

## main() function

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

- By passing in `s` as `const` we made a promise to *not* modify `s`.

# Remember our student class

## main() function

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

- By passing in `s` as `const` we made a promise to *not* modify `s`.
- The compiler doesn't know whether or not `getName()` and `getAge()` modify `s`



# Remember our student class

## main() function

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

- By passing in `s` as `const` we made a promise to *not* modify `s`.
- The compiler doesn't know whether or not `getName()` and `getAge()` modify `s`
- Remember, member functions *can* access and modify member variables

# Remember our student class

## .h file

```
class Student {  
Private:  
    /// An example of type aliasing  
    using String = std::string;  
    String name;  
    String state;  
    int age;  
  
public:  
    Student(String name, String state, int age);  
    /// Added a 'setter' function  
    void setName(String name);  
    String getName() const;  
    String getState();  
    int getAge() const;  
}
```

# Implemented members

## .cpp file (implementation)

```
#include "Student.h"
#include <string>

std::string Student::getName() {
    return this->name;
}

std::string Student::getState() {
    return this->state;
}

int Student::getAge() {
    return this->age;
}
```

# Implemented members

## .cpp file (implementation)

```
#include "Student.h"
#include <string>

std::string Student::getName() const {
    return this->name;
}

std::string Student::getState() {
    return this->state;
}

int Student::getAge() const {
    return this->age;
}
```

# Implemented members

## .cpp file (implementation)

```
#include "Student.h"
#include <string>

std::string Student::getName() const {
    return this->name;
}

std::string Student::getState() {
    return this->state;
}

int Student::getAge() const {
    return this->age;
}
```

You also have to make the implementation **const**. Otherwise the compiler will scream.

# Const interface

**Definition:**

Objects that are `const` can only interact with the const-interface.

The const interface is simply the functions that are const/do not modify the object/instance of the class.

# Const interface

## Definition:

## Objects that are `const` can only interact with the `const`-interface.

The const interface is simply the functions that are const/do not modify the object/instance of the class.

In our example:

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

# Const interface

## Definition:

## Objects that are `const` can only interact with the `const`-interface.

The const interface is simply the functions that are const/do not modify the object/instance of the class.

In our example:

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



# Const interface

## Definition:

Objects that are `const` can only interact with the const-interface.

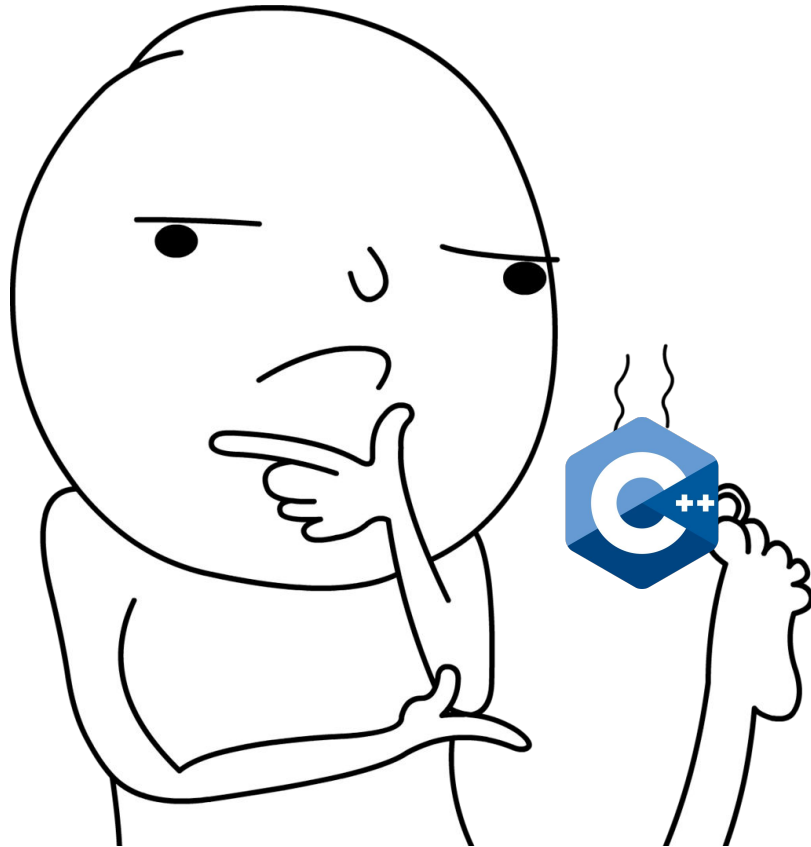
The const interface is simply the functions that are const/do not modify the object/instance of the class.

In our example:

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

This works now!

# What questions do we have?



# A new class IntArray

## .h file

```
class IntArray {  
Private:  
    int* _array  
    size_t _size  
  
Public:  
    // constructor  
    IntArray(size_t size);  
    ~IntArray();  
    int& at(size_t index);  
    int size();  
}
```

# A new class IntArray

## .cpp file

```
IntArray::IntArray(size_t size) : _size(size), _array(new int[size]);

IntArray::~IntArray() {
    delete [] _array;
}

/// assumes that index is valid
int& at(size_t index) {
    return _array[index];
}

int size() {
    return this->_size;
}
```

# A new class IntArray

main()

```
#include "IntArray.h"
#include <iostream>

static void printElement(const IntArray& arr, size_t index) {
    std::cout << arr.at(index) << std::endl;
}

int main() {
    IntArray arr = IntArray(10);
    int& secondVal = arr.at(1);
    /// actually changes the value of arr at index '1'.
    secondVal = 19;
    printElement(arr, 1);
    return 0;
}
```

# A new class IntArray

main()

```
#include "IntArray.h"
#include <iostream>

static void printElement(const IntArray& arr, size_t index) {
    std::cout << arr.at(index) << std::endl;
}

int main() {
    IntArray arr = IntArray(10);
    int& secondVal = arr.at(1);
    /// actually changes the value of arr
    secondVal = 19;
    printElement(arr, 1);
    return 0;
}
```

Anyone see a  
problem here?

# A new class IntArray

main()

```
#include "IntArray.h"
#include <iostream>

static void printElement(const IntArray& arr, size_t index) {
    std::cout << arr.at(index) << std::endl;
}

int main() {
    IntArray arr = IntArray(10);
    int& secondVal = arr.at(1);
    /// actually changes the value of arr at index 1
    secondVal = 19;
    printElement(arr, 1);
    return 0;
}
```



Hint

# A new class IntArray

## .cpp file

```
IntArray::IntArray(size_t size) : _size(size), _array(new int[size]);

IntArray::~IntArray() {
    delete [] _array;
}

/// assumes that index is valid
int& at(size_t index) {
    return _array[index];
}

int size() {
    return this->_size;
}
```



Hint #2




# A new class IntArray

main()

```
#include "IntArray.h"
#include <iostream>

static void printElement(const IntArray& arr, size_t index) {
    std::cout << arr.at(index) << std::endl;
}

int main() {
    IntArray arr = IntArray(10);
    int& secondVal = arr.at(1);
    /// actually changes the value of arr at index 1
    secondVal = 19;
    printElement(arr, 1);
    return 0;
}
```



It turns out that we're passing arr as a constant. What did we say about const objects?

# A new class IntArray

main()

```
#include "IntArray.h"
#include <iostream>

static void printElement(const IntArray& arr, size_t index) {
    std::cout << arr.at(index) << std::endl;
}

int main() {
    IntArray arr = IntArray(10);
    int& secondVal = arr.at(1);
    /// actually changes the value of arr at index 1
    secondVal = 19;
    printElement(arr, 1);
    return 0;
}
```

const objects must  
use the const  
interface.

# A new class IntArray

## .cpp file

```
IntArray::IntArray(size_t size) : _size(size), _array(new int[size]);

IntArray::~IntArray() {
    delete [] _array;
}

/// assumes that index is valid
int& at(size_t index) {
    return _array[index];
}

int size() {
    return this->_size;
}
```

Our .at() function  
is not const!

# A new class IntArray

## .cpp file

```
IntArray::IntArray(size_t size) : _size(size), _array(new int[size]);

IntArray::~IntArray() {
    delete [] _array;
}

/// assumes that index is valid
int& at(size_t index) {
    return _array[index];
}

int size() {
    return this->_size;
}
```

Ok. What do we do?  
Any ideas? 🤔

# A new class IntArray

## .cpp file

```
IntArray::IntArray(size_t size) : _size(size), _array(new int[size]);

IntArray::~IntArray() {
    delete [] _array;
}

/// assumes that index is valid
int& at(size_t index) {
    return _array[index];
}

int size() {
    return this->_size;
}
```

Ok. What do we do?  
Any ideas? 🤔

# A new class `IntArray`

## `.cpp` file

Other member functions

...

`/// assumes that index is valid`

```
int& at(size_t index) {  
    return _array[index];  
}
```

`/// we can make a const version`

```
int& at(size_t index) const {  
    return _array[index];  
}
```

We can create a  
const version of our  
`.at()`

# A new class `IntArray`

## `.cpp` file

Other member functions

`. . . . .`

`/// assumes that index is valid`

```
int& at(size_t index) {  
    return _array[index];  
}
```

`/// we can make a const version`

```
int& at(size_t index) const {  
    return _array[index];  
}
```

This might be  
slightly painful  
though

# A new class `IntArray`

## `.cpp` file

Other member functions

`. . . . .`

`/// assumes that index is valid`

```
int& at(size_t index) {  
    return _array[index];  
}
```

`/// we can make a const version`

```
int& at(size_t index) const {  
    return _array[index];  
}
```

In this example we  
only have to  
rewrite a function  
that has one line of  
code



# A new class `IntArray`

## .cpp file

```
int& findItem(int value) {  
    for (auto& elem: arr) {  
        if (elem == value) return elem;  
    }  
  
    throw std::out_of_range("value not found")  
}
```

What if we had  
another function  
that we needed a  
const version of  
like this

# A new class IntArray

## .cpp file

```
int& findItem(int value) {  
    for (auto& elem: arr) {  
        if (elem == value) return elem;  
    }  
  
    throw std::out_of_range("value not found")  
}  
  
const int& findItem(int value) const {  
    for (auto& elem: arr) {  
        if (elem == value) return elem;  
    }  
  
    throw std::out_of_range("value not found")  
}
```

Like sure. We *could* do this, but the level of pain/annoying scales linearly with the length of the function

# A slight (but useful) aside

## **Casting:**

The process of converting types, there are many ways to do this.

# A slight (but useful) aside

## Casting:

The process of converting types, there are many ways to do this.

## `const_cast`:

- `const_cast<target-type> ( expression )`
- We can use `const_cast` to cast away `const`-ness.
- So why is this useful?

# A new class IntArray

## .cpp file

```
int& findItem(int value) {  
    for (auto& elem: arr) {  
        if (elem == value) return elem;  
    }  
  
    throw std::out_of_range("value not found")  
}
```

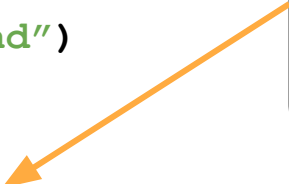
Less pain here

```
const int& findItem(int value) const {  
    /// one-liners ftw :)  
    return const_cast<IntArray&>(*this).findItem(value)  
}
```

# A new class IntArray

## .cpp file

```
int& findItem(int value) {  
    for (auto& elem: arr) {  
        if (elem == value) return elem;  
    }  
  
    throw std::out_of_range("value not found")  
}  
  
const int& findItem(int value) const {  
    /// one-liners ftw :)  
    return const_cast<IntArray&>(*this).findItem(value)  
}
```



This is dense  
though, let's break  
this down

# A new class IntArray

## .cpp file

```
int& findItem(int value) {  
    for (auto& elem: arr) {  
        if (elem == value) return elem;  
    }  
  
    throw std::out_of_range("value not found")  
}  
  
const int& findItem(int value) const {  
    /// one-liners ftw :)  
    return const_cast<IntArray&>(*this).findItem(value)  
}
```

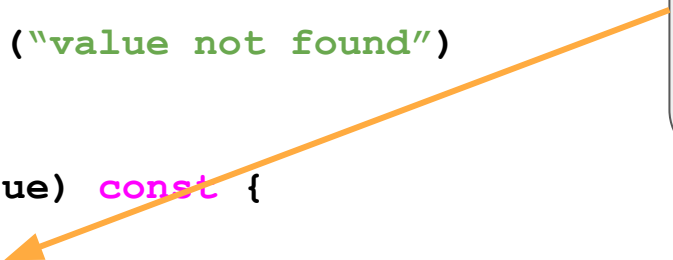
The `const_cast` is casting away the `const`

# A new class IntArray

## .cpp file

```
int& findItem(int value) {  
    for (auto& elem: arr) {  
        if (elem == value) return elem;  
    }  
  
    throw std::out_of_range("value not found")  
}  
  
const int& findItem(int value) const {  
    /// one-liners ftw :)  
    return const_cast<IntArray&>(*this).findItem(value)  
}
```

This text here is the key this is our target. A non-const reference





# A new class IntArray

## .cpp file

```
int& findItem(int value) {  
    for (auto& elem: arr) {  
        if (elem == value) return elem;  
    }  
  
    throw std::out_of_range("value not found")  
}  
  
const int& findItem(int value) const {  
    /// one-liners ftw :)  
    return const_cast<IntArray&>(*this).findItem(value)  
}
```

What in the bjarne is this?

# A new class IntArray

## .cpp file

```
int& findItem(int value) {  
    for (auto& elem: arr) {  
        if (elem == value) return elem;  
    }  
  
    throw std::out_of_range("value not found")  
}  
  
const int& findItem(int value) const {  
    /// one-liners ftw :)  
    return const_cast<IntArray&>(*this).findItem(value)  
}
```

Lol. We didn't really talk about it, but **this** is really a pointer.

# A new class IntArray

## .cpp file

```
int& findItem(int value) {  
    for (auto& elem: arr) {  
        if (elem == value) return elem;  
    }  
  
    throw std::out_of_range("value not found")  
}  
  
const int& findItem(int value) const {  
    /// one-liners ftw :)  
    return const_cast<IntArray&>(*this).findItem(value)  
}
```

Here we're  
dereferencing **this**  
so that we can cast  
it as non-const.

# A new class IntArray

## .cpp file

```
int& findItem(int value) {  
    for (auto& elem: arr) {  
        if (elem == value) return elem;  
    }  
  
    throw std::out_of_range("value not found")  
}  
  
const int& findItem(int value) const {  
    /// one-liners ftw :)  
    return const_cast<IntArray&>(*this).findItem(value);  
}
```

1. Cast so that it is pointing to a non-const object
2. Call the non-const version of the function (this is legal now)
3. Then cast the non-const return from the function call to a const version

# What questions do we have?

