

CS106L Lecture 15:

RAII, Smart Pointers,

Building Projects

Autumn 2024

Fabio Ibanez, Jacob Roberts-Baca

Attendance




<https://tinyurl.com/raiiF24>

Plan


1. RAI (Resource Acquisition Is Initialization)
2. Smart Pointers
3. Building C++ projects

How many code paths?




```
std::string returnNameCheckPawsome(Pet p) {  
    /// NOTE: dogs > cats  
    → if (p.type() == "Dog" || p.firstName() == "Fluffy") {  
        std::cout << p.firstName() << " " <<  
            p.lastName() << " is paw-some!" << '\n';  
    }  
    return p.firstName() + " " + p.lastName();  
}
```

How many code paths?




```
std::string returnNameCheckPawsome(Pet p) {  
    /// NOTE: dogs > cats  
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {  
        std::cout << p.firstName() << " " <<  
            p.lastName() << " is paw-some!" << '\n';  
    }  
    return p.firstName() + " " + p.lastName();  
}
```

How many code paths?



```
std::string returnNameCheckPawsome(Pet p) {  
    /// NOTE: dogs > cats  
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {  
        std::cout << p.firstName() << " " <<  
            p.lastName() << " is paw-some!" << '\n';  
    }  
    → return p.firstName() + " " + p.lastName();  
}
```

How many code paths?



```
std::string returnNameCheckPawsome(Pet p) {  
    /// NOTE: dogs > cats  
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {  
        std::cout << p.firstName() << " " <<  
            p.lastName() << " is paw-some!" << '\n';  
    }  
    return p.firstName() + " " + p.lastName();  
}
```

3?

Exceptions

- Exceptions are a way of handling errors when they arise in code

Exceptions

- Exceptions are a way of handling errors when they arise in code
- Exceptions are “thrown”

Exceptions

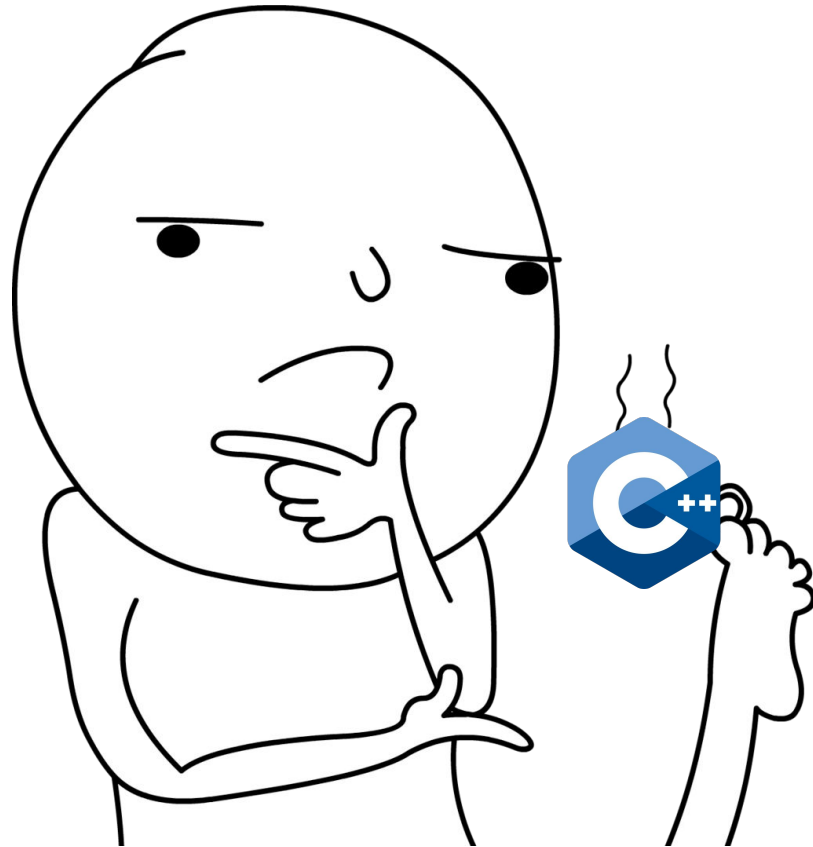
- Exceptions are a way of handling errors when they arise in code
- Exceptions are “thrown”
- However, we can write code that lets us handle exceptions so that we can continue in our code without necessarily erroring.

Exceptions


- Exceptions are a way of handling errors when they arise in code
- Exceptions are “thrown”
- However, we can write code that lets us continue in our code without
- We call this “catching” an exception.

```
try {  
    // code that we check for exceptions  
}  
catch([exception type] e1) { // "if"  
    // behavior when we encounter an error  
}  
catch([other exception type] e2) { // "else if"  
    // ...  
}  
catch { // the "else" statement  
    // catch-all (haha)  
}
```

What questions do we have?



How many code paths?



```
std::string returnNameCheckPawsome(Pet p) {  
    /// NOTE: dogs > cats  
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {  
        std::cout << p.firstName() << " " <<  
            p.lastName() << " is paw-some!" << '\n';  
    }  
    return p.firstName() + " " + p.lastName();  
}
```

At least 23 code paths!

- (1): Copy constructor of Pet may throw
- (5): Constructor of temp strings may throw
- (6): Call to type, firstName(3), lastName(2) may throw
- (10): User overloaded operators may throw
- (1): Copy constructor of returned string may throw


```
std::string returnNameCheckPawsome(Pet p) {  
    /// NOTE: dogs > cats  
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {  
        std::cout << p.firstName() << " " <<  
            p.lastName() << " is paw-some!" << '\n';  
    }  
    return p.firstName() + " " + p.lastName();  
}
```

What could go wrong in this new code?



```
std::string returnNameCheckPawsome(int petId) {  
    Pet* p = new Pet(petId);  
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {  
        std::cout << p.firstName() << " " <<  
            p.lastName() << " is paw-some!" << '\n';  
    }  
    std::string returnStr = p.firstName() + " " + p.lastName();  
    delete p;  
    return returnStr;  
}
```

What could go wrong?



```
std::string returnNameCheckPawsome(int petId) {  
    Pet* p = new Pet(petId);  
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {  
        std::cout << p.firstName() << " " <<  
            p.lastName() << " is paw-some!" << '\n';  
    }  
    std::string returnStr = p.firstName() + " " + p.lastName();  
    delete p;  
    return returnStr;  
}
```

What if this
function threw an
exception here?

What could go wrong?

```
std::string returnNameCheckPawsome(int petId) {  
    Pet* p = new Pet(petId);  
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {  
        std::cout << p.firstName() << " " <<  
            p.lastName() << " is paw-some!" << '\n';  
    }  
    std::string returnStr = p.firstName() + " " + p.lastName();  
    delete p;  
    return returnStr;  
}
```

What if this
function threw an
exception here?

Or here?

What could go wrong?

```
std::string returnNameCheckPawsome(int petId) {  
    Pet* p = new Pet(petId);  
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {  
        std::cout << p.firstName() << " " <<  
            p.lastName() << " is paw-some!" << '\n';  
    }  
    std::string returnStr = p.firstName() + " " + p.lastName();  
    delete p;  
    return returnStr;  
}
```


What if this
function threw an
exception here?

Or here?

Or here?


Or anywhere an exception can be
thrown?

What could go wrong?



```
std::string returnNameCheckPawsome(int petId) {  
    Pet* p = new Pet(petId);  
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {  
        std::cout << p.firstName() << " " <<  
            p.lastName() << " is paw-some!" << '\n';  
    }  
    std::string returnStr = p.firstName() + " " + p.lastName();  
    delete p;  
    return returnStr;  
}
```

What could go wrong?



```
std::string returnNameCheckPawsome(int petId) {  
    Pet* p = new Pet(petId);  
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {  
        std::cout << p.firstName() << " " <<  
            p.lastName() << " is paw-some!" << '\n';  
    }  
    std::string returnStr = p.firstName() + " " + p.lastName();  
    delete p;  
    return returnStr;  
}
```

What could go wrong?

exception
here
means
memory
leak

```
std::string returnNameCheckPawsome(int petId) {  
    Pet* p = new Pet(petId);  
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {  
        std::cout << p.firstName() << " " <<  
            p.lastName() << " is paw-some!" << '\n';  
    }  
    std::string returnStr = p.firstName() + " " + p.lastName();  
    delete p;  
    return returnStr;  
}
```

This is not unique to just pointers!

It turns out that there are many resources that you need to release after acquiring

| | Acquire | Release |
|-------------|-----------------------|---------------------|
| Heap memory | <code>new</code> | <code>delete</code> |
| Files | <code>open</code> | <code>close</code> |
| Locks | <code>try_lock</code> | <code>unlock</code> |
| Sockets | <code>socket</code> | <code>close</code> |

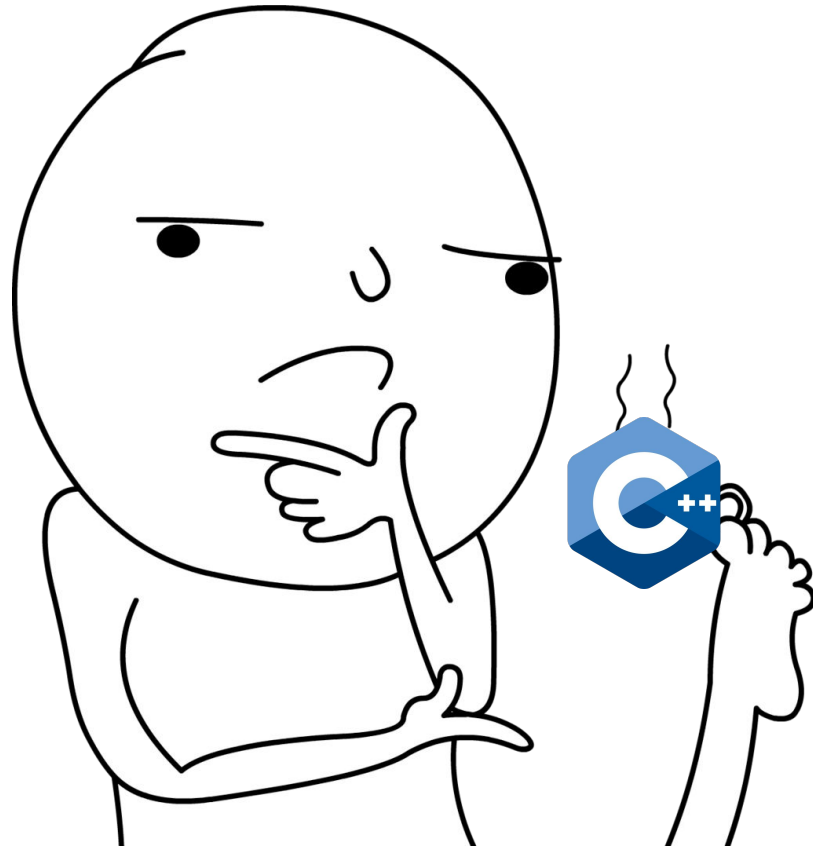
This is not unique to just pointers!

It turns out that there are many resources that you need to release after acquiring

| | Acquire | Release |
|-------------|-----------------------|---------------------|
| Heap memory | <code>new</code> | <code>delete</code> |
| Files | <code>open</code> | <code>close</code> |
| | <code>try_lock</code> | <code>unlock</code> |
| | <code>socket</code> | <code>close</code> |

How to we ensure that we properly release resources in the case that we have an exception?

What questions do we have?



RAII

RAII: Resource Acquisition is Initialization

RAII

RAII: Resource Acquisition is Initialization (What is this name?)

RAII was developed by this lad:



And it's a concept that is very emblematic in C++, among other languages.

RAII

RAII: Resource Acquisition is Initialization (What is this name?)

RAII was developed by this lad:



And it's a concept that is very emblematic in C++, among other languages.

So what is RAII?

- All resources used by a class should be acquired in the constructor!
- All resources that are used by a class should be released in the destructor.

RAII

RAII: Resource Acquisition is Initialization



RAll: why tho?

RAll: Resource Acquisition is Initialization

- By abiding by the RAll policy we avoid “half-valid” states.

RAII: why tho?

RAII: Resource Acquisition is Initialization


- By abiding by the RAII policy we avoid “half-valid” states.
- No matter what, the destructor is called whenever the resource goes out of scope.

RAII: why tho?

RAII: Resource Acquisition is Initialization

- By abiding by the RAII policy we avoid “half-valid” states.
- No matter what, the destructor is called whenever the resource goes out of scope.
- One more thing: the resource/object is usable immediately after it is created.

RAII compliant?



```
void printFile() {  
    ifstream input;  
    input.open("hamlet.txt");  
  
    string line;  
    while(getLine(input, line)) { // might throw an exception  
        std::cout << line << std::endl;  
    }  
  
    input.close();  
}
```


RAII compliant?

```
void printFile() {  
    ifstream input;  
    input.open("hamlet.txt");  
  
    string line;  
    while(getLine(input, line)) { // might throw an exception  
        std::cout << line << std::endl;  
    }  
    input.close();  
}
```

the
ifstream is
opened and
closed in
code, not
constructor
& destructor

Neither is this!




```
void cleanDatabase(mutex& databaseLock, map<int, int>& db) {  
    databaseLock.lock();  
  
    // no other thread or machine can change database  
    // modify the database  
    // if any exception is thrown, the lock never unlocks!  
  
    database.unlock();  
}
```

Neither is this!

```
void cleanDatabase(mutex& databaseLock, map<int, int>& db) {  
    databaseLock.lock();  
  
    // no other thread or machine can change database  
    // modify the database  
    // if any exception is thrown, the lock never unlocks!  
  
    database.unlock();  
}
```


If any code throws an exception in the red area, which we can call the 'critical section', the lock never unlocks!

How can we fix this?



```
void cleanDatabase(mutex& databaseLock, map<int, int>& db) {  
    lock_guard<mutex> lg(databaseLock);  
    // no other thread or machine can change database  
    // modify the database  
    // if exception is throw, mutex is UNLOCKED!  
  
    // no explicit unlock necessary, is handled by lock_guard  
}
```

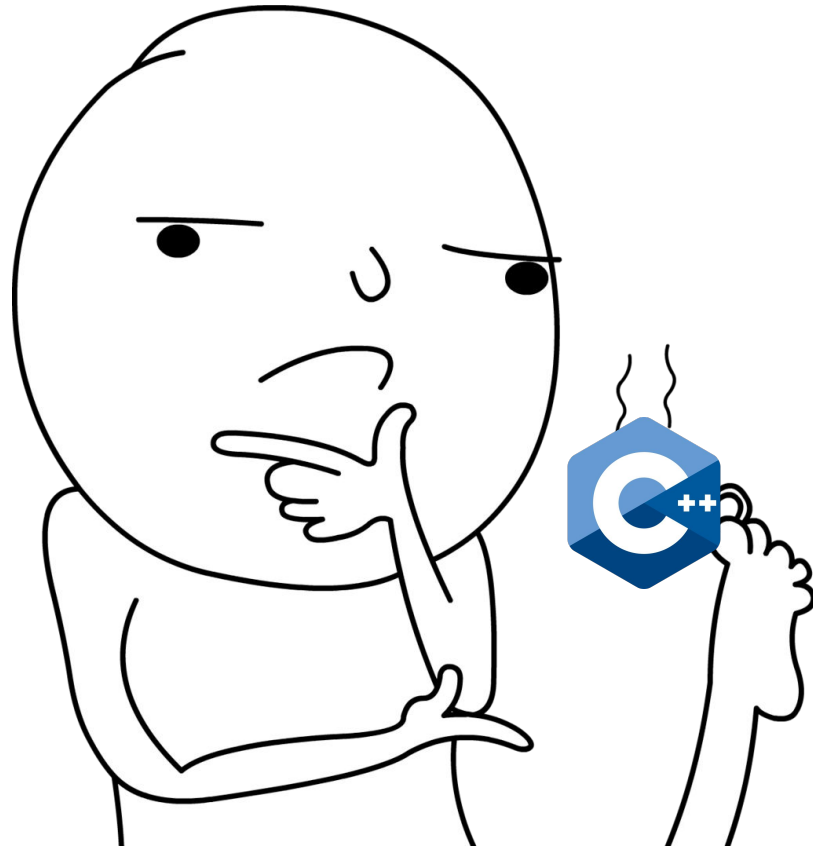
How can we fix this?



```
void cleanDatabase(mutex& databaseLock, map<int, int>& db) {  
    lock_guard<mutex> lg(databaseLock);  
    // no other thread or machine can change database  
    // modify the database  
    // if exception is throw, mutex is not locked  
    // no explicit unlock necessary  
}
```

A lock guard is a RAII-compliant wrapper that attempts to acquire the passed in lock. It releases the the lock once it goes out of scope. Read more [here](#)

What questions do we have?



Plan

1. RAI (Resource Acquisition Is Initialization)
2. **Smart Pointers**
3. Building C++ projects

Smart Pointers

RAII for locks \rightarrow `lock_guard`

Smart Pointers

RAII for locks → `lock_guard`

RAII for memory → 🤔

Smart Pointers

R.11: Avoid calling `new` and `delete` explicitly

Reason

The pointer returned by `new` should belong to a resource handle (that can call `delete`). If the pointer returned by `new` is assigned to a plain/naked pointer, the object can be leaked.

Note

In a large program, a naked `delete` (that is a `delete` in application code, rather than part of code devoted to resource management) is a likely bug: if you have `N` `delete`s, how can you be certain that you don't need `N+1` or `N-1`? The bug may be latent: it may emerge only during maintenance. If you have a naked `new`, you probably need a naked `delete` somewhere, so you probably have a bug.

Enforcement

(Simple) Warn on any explicit use of `new` and `delete`. Suggest using `make_unique` instead.

Remember this?



```
std::string returnNameCheckPawsome(int petId) {  
    Pet* p = new Pet(petId);  
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {  
        std::cout << p.firstName() << " " <<  
            p.lastName() << " is paw-some!" << '\n';  
    }  
    std::string returnStr = p.firstName() + " " + p.lastName();  
    delete p;  
    return returnStr;  
}
```

What did we do for locks?

RAII for locks → `lock_guard`

- Created a new object that acquires the resource in the constructor and releases in the destructor

What did we do for locks?

RAII for locks → `lock_guard`

- Created a new object that acquires the resource in the constructor and releases in the destructor

RAII for memory → We can do the same 🎉

What did we do for locks?

RAII for locks → `lock_guard`

- Created a new object that acquires the resource in the constructor and releases in the destructor

RAII for memory → We can do the same 🎉

- These “wrapper” pointers are called “smart pointers”!

Visualizing smart pointers

RAII for locks → `lock_guard`

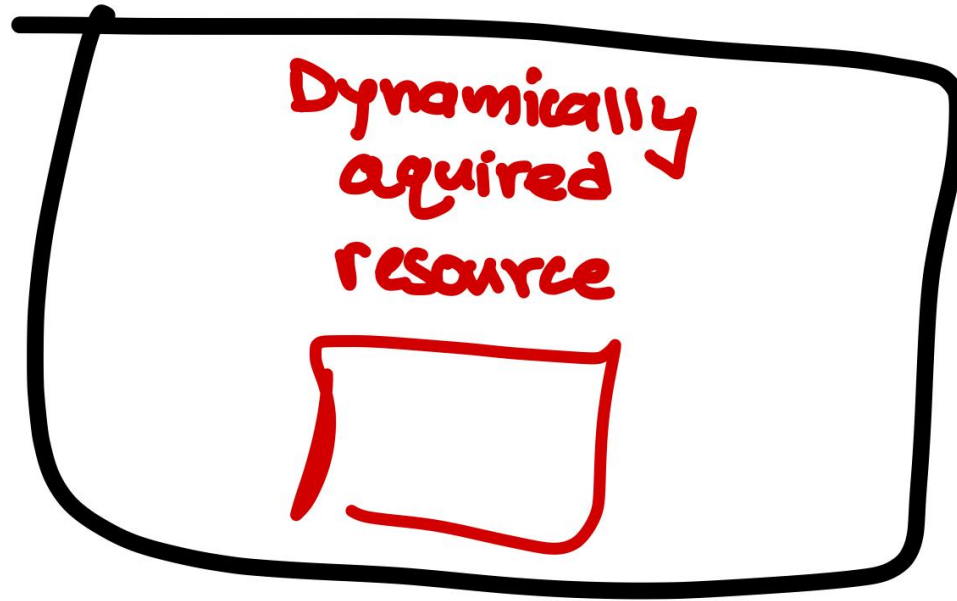
- Created a new object that acquires the resource in the constructor and releases in the destructor

RAII for memory → We can do the same 🎉

- These “wrapper” pointers are called “smart pointers”!

Visualizing smart pointers

Smart Pointer Class



Visualizing smart pointers

RAII for memory → We can do the same 🎉

- These “wrapper” pointers are called “smart pointers”!

There are three types of RAII-compliant pointers:

- **`std::unique_ptr`**
 - Uniquely owns its resource, can't be copied

Visualizing smart pointers

RAII for memory → **We can do the same** 🎉

- These “wrapper” pointers are called “smart pointers”!

There are three types of RAII-compliant pointers:

- **`std::unique_ptr`**
 - Uniquely owns its resource, can't be copied
- **`std::shared_ptr`**
 - Can make copies, destructed when the ***underlying memory*** goes out of scope

Visualizing smart pointers

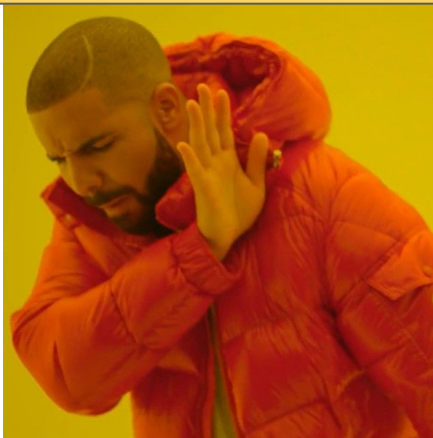
RAII for memory → **We can do the same** 🎉

- These “wrapper” pointers are called “smart pointers”!

There are three types of RAII-compliant pointers:

- **`std::unique_ptr`**
 - Uniquely owns its resource, can't be copied
- **`std::shared_ptr`**
 - Can make copies, destructed when the underlying memory goes out of scope
- **`std::weak_ptr`**
 - A class of pointers designed to mitigate circular dependencies
 - More on these in a bit

What does this look like?

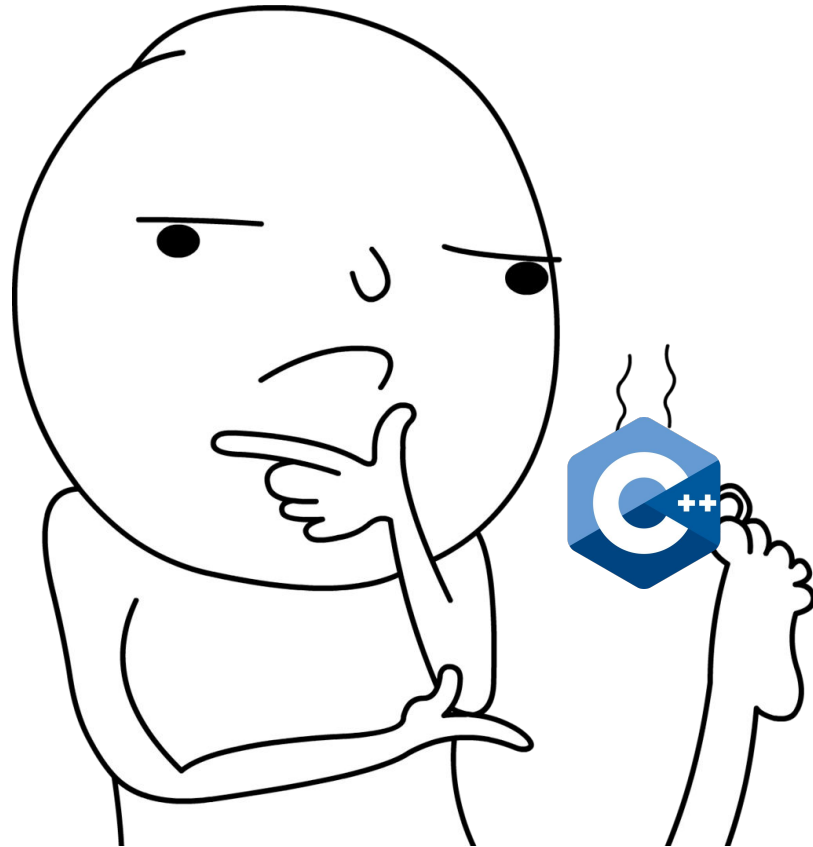


```
void rawPtrFn() {  
    Node* n = new Node;  
    // do smth with n  
    delete n;  
}
```




```
void rawPtrFn() {  
    std::unique_ptr<Node> n(new Node);  
    // do something with n  
    // n automatically freed  
}
```

What questions do we have?



Remember we can't copy unique pointers



```
void rawPtrFn() {  
    std::unique_ptr<Node> n(new Node);  
  
    // this is a compile-time error!  
    std::unique_ptr<Node> copy = n;  
}
```

Why?



```
void rawPtrFn() {  
    std::unique_ptr<Node> n(new Node);  
  
    // this is a compile-time error!  
    std::unique_ptr<Node> copy = n;  
}
```

Imagine a case where the original destructor is called **after** the copy happens.

Why?



```
void rawPtrFn() {  
    std::unique_ptr<Node> n(new Node);  
  
    // this is a compile-time error!  
    std::unique_ptr<Node> copy = n;  
}
```

Imagine a case where the original destructor is called **after** the copy happens.

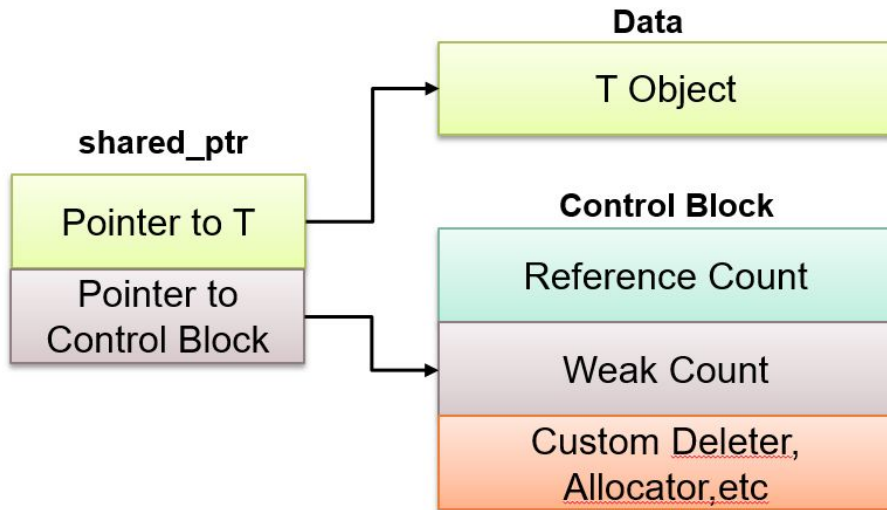
Problem: The copy points to deallocated memory!

`std::shared_ptr`

Shared pointers get around our issue of trying to copy `std::unique_ptr`'s by not deallocating the underlying memory until *all* shared pointers go out of scope!

std::shared_ptr

Shared pointers get around our issue of trying to copy **std::unique_ptr**'s by not deallocating the underlying memory until **all** shared pointers go out of scope!



Initializing smart pointers!



```
std::unique_ptr<T> uniquePtr{new T};
```

```
std::shared_ptr<T> sharedPtr{new T};
```

```
std::weak_ptr<T> wp = sharedPtr;
```

Initializing smart pointers!

```
std::unique_ptr<T> uniquePtr{new T};
```


```
std::shared_ptr<T> sharedPtr{new T};
```

```
std::weak_ptr<T> wp = sharedPtr;
```

We're still explicitly
calling **new**

no....no

Initializing smart pointers!



```
// std::unique_ptr<T> uniquePtr{new T};  
std::unique_ptr<T> uniquePtr = std::make_unique<T>();  
  
// std::shared_ptr<T> sharedPtr{new T};  
std::shared_ptr<T> sharedPtr = std::make_shared<T>();  
  
std::weak_ptr<T> wp = sharedPtr;
```

Initializing smart pointers!

Always use `std::make_unique<T>` and `std::make_shared<T>`

Why?

1. The most important reason: if we don't then we're going to allocate memory twice, once for the pointer itself, and once for the **new T**

Initializing smart pointers!

Always use `std::make_unique<T>` and `std::make_shared<T>`

Why?

1. The most important reason: if we don't then we're going to allocate memory twice, once for the pointer itself, and once for the **new T**
2. We should also be consistent — if you use **make_unique** also use **make_shared**!

std::weak_ptr

Weak pointers are a way to avoid circular dependencies in our code so that we don't leak any memory.


```
#include <iostream>
#include <memory>

class B;

class A {
public:
    std::shared_ptr<B> ptr_to_b;
    ~A() {
        std::cout << "All of A's resources deallocated" << std::endl;
    }
};

class B {
public:
    std::shared_ptr<A> ptr_to_a;
    ~B() {
        std::cout << "All of B's resources deallocated" << std::endl;
    }
};

int main() {
    std::shared_ptr<A> shared_ptr_to_a = std::make_shared<A>();
    std::shared_ptr<A> shared_ptr_to_b = std::make_shared<B>();
    a->ptr_to_b = shared_ptr_to_b;
    b->ptr_to_a = shared_ptr_to_a;
    return 0;
}
```

std::weak_ptr bad example

```
#include <iostream>
#include <memory>

class B;

class A {
public:
    std::shared_ptr<B> ptr_to_b;
    ~A() {
        std::cout << "All of A's resources deallocated" << std::endl;
    }
};

class B {
public:
    std::shared_ptr<A> ptr_to_a;
    ~B() {
        std::cout << "All of B's resources deallocated" << std::endl;
    }
};

int main() {
    std::shared_ptr<A> shared_ptr_to_a = std::make_shared<A>();
    std::shared_ptr<B> shared_ptr_to_b = std::make_shared<B>();
    a->ptr_to_b = shared_ptr_to_b;
    b->ptr_to_a = shared_ptr_to_a;
    return 0;
}
```

Both instance **a** of class **A** and instance **b** of class **B** are keeping a share pointer to each other.

std::weak_ptr bad example

```
#include <iostream>
#include <memory>

class B;

class A {
public:
    std::shared_ptr<B> ptr_to_b;
    ~A() {
        std::cout << "All of A's resources deallocated" << std::endl;
    }
};

class B {
public:
    std::shared_ptr<A> ptr_to_a;
    ~B() {
        std::cout << "All of B's resources deallocated" << std::endl;
    }
};

int main() {
    std::shared_ptr<A> shared_ptr_to_a = std::make_shared<A>();
    std::shared_ptr<A> shared_ptr_to_b = std::make_shared<B>();
    a->ptr_to_b = shared_ptr_to_b;
    b->ptr_to_a = shared_ptr_to_a;
    return 0;
}
```

Both instance **a** of class **A** and instance **b** class **B** are keeping a share pointer to each other.

Therefore, they will never properly deallocate

std::weak_ptr good example

```
#include <iostream>
#include <memory>

class B;

class A {
public:
    std::shared_ptr<B> ptr_to_b;
    ~A() {
        std::cout << "All of A's resources deallocated" << std::endl;
    }
};

class B {
public:
    std::weak_ptr<A> ptr_to_a;
    ~B() {
        std::cout << "All of B's resources deallocated" << std::endl;
    }
};

int main() {
    std::shared_ptr<A> shared_ptr_to_a = std::make_shared<A>();
    std::shared_ptr<A> shared_ptr_to_b = std::make_shared<B>();
    a->ptr_to_b = shared_ptr_to_b;
    b->ptr_to_a = shared_ptr_to_a;
    return 0;
}
```

Here, in class B we are no longer storing **a** as a `shared_ptr` so it does not increase the reference count of **a**.

Therefore **a** can gracefully be deallocated, and therefore so can **b**

Plan

1. RAI (Resource Acquisition Is Initialization)
2. Smart Pointers
3. **Building C++ projects**

Compilation Crash Course

When we write C++ code, it needs to be translated into a form our computer understands it

Compilation Crash Course

When we write C++ code, it needs to be translated into a form our computer understands it

Source Code

```
std::cout << "Hello World" << std::endl;  
std::cout << "Welcome to " << std::endl;  
for (char ch : "CS106L")  
{  
    std::cout << ch << std::endl;  
}
```

Compiler

Machine Code

```
10110101  
01011010  
10011101  
10110001
```

```
$ g++ main.cpp -o main    # g++ is the compiler, outputs binary to main  
$ ./main                 # This actually runs our program
```

Compilation Crash Course

When we write C++ code, it needs to be translated into a form our computer understands it

Source Code

```
std::cout << "Hello World" << std::endl;  
std::cout << "Welcome to " << std::endl;  
for (char ch : "CS106L")  
{  
    std::cout << ch << std::endl;  
}
```

Compiler

Machine Code

```
10110101  
01011010  
10011101  
10110001
```

```
$ g++ main.cpp -o main
```

```
$ ./main
```

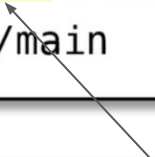
g++ is the compiler, outputs binary to main

This actually runs our program

Compilation Crash Course

When we write C++ code, it needs to be translated into a form our computer understands it

```
$ g++ main.cpp -o main    # g++ is the compiler, outputs binary to main  
$ ./main                 # This actually runs our program
```



This is the compiler
command

Compilation Crash Course

When we write C++ code, it needs to be translated into a form our computer understands it

```
$ g++ main.cpp -o main    # g++ is the compiler, outputs binary to main
$ ./main                 # This actually runs our program
```




This is the source file

Compilation Crash Course

When we write C++ code, it needs to be translated into a form our computer understands it

```
$ g++ main.cpp -o main # g++ is the compiler, outputs binary to main
$ ./main               # This actually runs our program
```



This means that you're going to give a specific name to your executable

Compilation Crash Course

When we write C++ code, it needs to be translated into a form our computer understands it

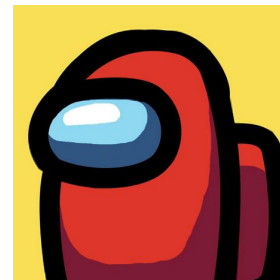
```
$ g++ main.cpp -o main # g++ is the compiler, outputs binary to main
$ ./main               # This actually runs our program
```

In this case it's main

GPU Programming



Even the masterpiece
among us





python [3.9](#) | [3.10](#) | [3.11](#) | [3.12](#) | pyPI package [2.18.0](#) | DOI [10.5281/zenodo.4724125](#) | [opencss best practices](#) [passing](#)
[opencss scorecard](#) [7.8](#) | [oss-fuzz](#) [build failing](#) | [oss-fuzz](#) [build failing](#) | [OSSRank](#) [#12 \(Top 1%\)](#) | [Contributor Covenant](#) [v1.4 adopted](#)
TF Official Continuous [6 passed, 0 failed](#) | TF Official Nightly [11 passed, 4 failed](#)

Documentation

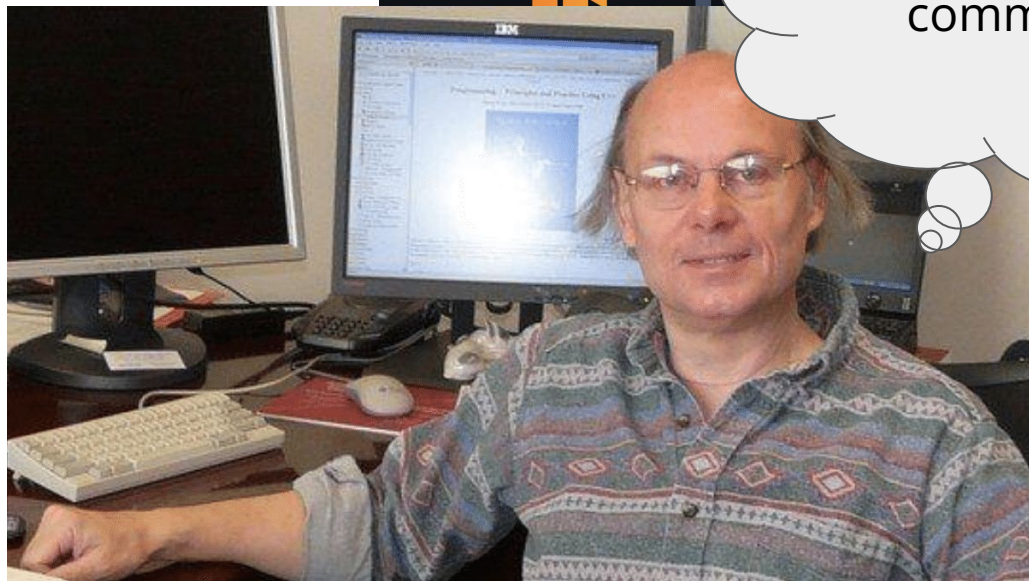
[api](#) [reference](#)

[TensorFlow](#) is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of [tools](#), [libraries](#), and [community](#) resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML-powered applications.

TensorFlow was originally developed by researchers and engineers working within the Machine Intelligence team at Google Brain to conduct research in machine learning and neural networks. However, the framework is versatile enough to be used in other areas as well.

TensorFlow provides stable [Python](#) and [C++](#) APIs, as well as a non-guaranteed backward compatible API for [other languages](#).

The TensorFlow Core is written largely in C++ and it is **composed of 2,000+ source files**



Lol, that's a cute
command 😭

TensorFlow provides stable [Python](#) and [C++](#) APIs, as well as a non-guaranteed backward compatible API for [other languages](#).

```
$ g++ main.cpp -o main    # g++ is the compiler, outputs binary to main
$ ./main                 # This actually runs our program
```

Makefiles and make

make is a “build system” program that helps you compile!

Makefiles and make

make is a “build system” program that helps you compile!

- You can specify what compiler you want to use

Makefiles and make

make is a “build system” program that helps you compile!

- You can specify what compiler you want to use
- In order to use **make** you need to have a **Makefile**

Makefiles and make

make is a “build system” program that helps you compile!

- You can specify what compiler you want to use
- In order to use **make** you need to have a **Makefile**

What does a **Makefile** look like? Let's take a look!

```
# Compiler
CXX = g++

# Compiler flags
CXXFLAGS = -std=c++20

# Source files and target
SRCS = ../main.cpp $(wildcard ../src/*.cpp)
TARGET = main

# Default target
all: $(TARGET)

# Build the executable
$(TARGET): $(SRCS)
    $(CXX) $(CXXFLAGS) $(SRCS) -o $(TARGET)

# Target to enable virtual inheritance
virtual: CXXFLAGS += -DVIRTUAL_INHERITANCE
virtual: $(TARGET)

# Clean up
clean:
    rm -f $(TARGET)
```

```
# Compiler
CXX = g++

# Compiler flags
CXXFLAGS = -std=c++20
```

Flags

```
# Source files and target
SRCS = ../main.cpp $(wildcard ../src/*.cpp)
TARGET = main
```

```
# Default target
all: $(TARGET)
```

```
# Build the executable
```

```
$(TARGET): $(SRCS)
    $(CXX) $(CXXFLAGS) $(SRCS) -o $(TARGET)
```

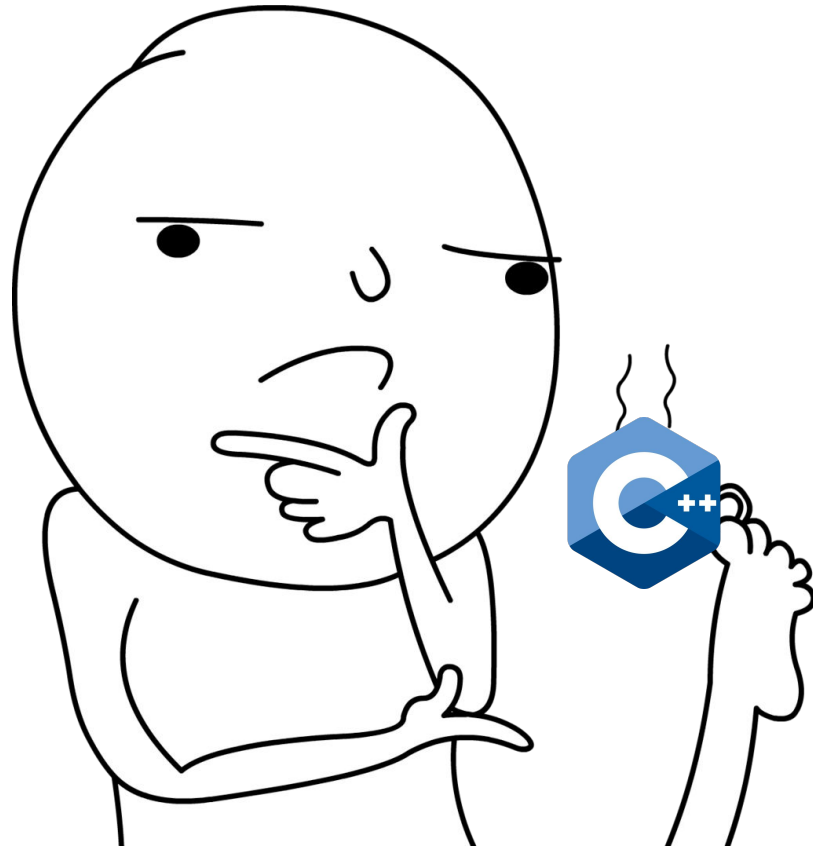
Targets

Rules

```
# Clean up
clean:
    rm -f $(TARGET)
```

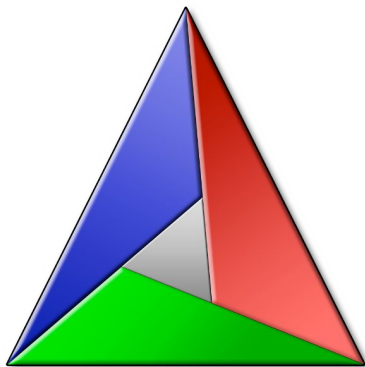
The target, in our case `main` depends on the rules which are really just the source files!

What questions do we have?



CMake

CMake is a build system generator.

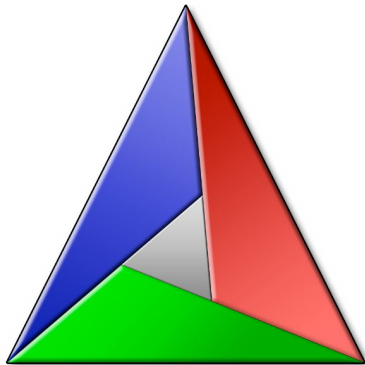


CMake

CMake

CMake is a build system generator.

So you can use **CMake** to generate Makefiles



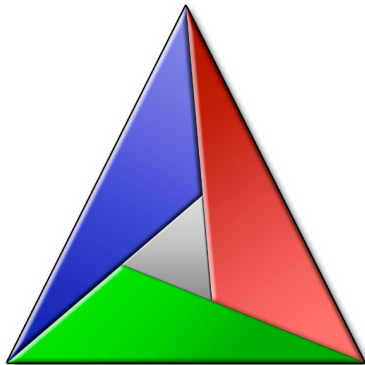
CMake

CMake

CMake is a build system generator.

So you can use **CMake** to generate Makefiles

Is like a higher level abstraction for Makefiles



CMake

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(cs106l_inheritance)

set(CMAKE_CXX_STANDARD 20)

include_directories(include)

add_definitions(-DVIRTUAL_INHERITANCE)

file(GLOB SRC_FILES "src/*.cpp")

add_executable(main main.cpp ${SRC_FILES})
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(cs106l_inheritance)

set(CMAKE_CXX_STANDARD 20)

include_directories(include)

add_definitions(-DVIRTUAL_INHERITANCE)

file(GLOB SRC_FILES "src/*.cpp")

add_executable(main main.cpp ${SRC_FILES})
```

This is the cmake file for our assignment – it looks more like a programming language!

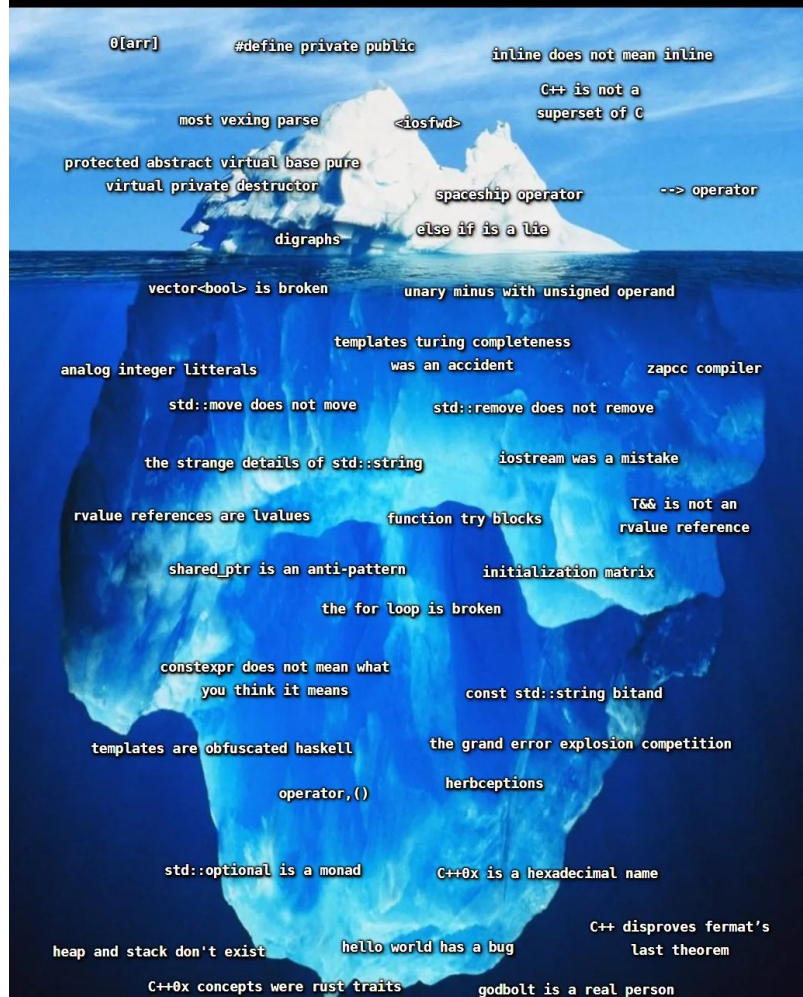
A recap

- RAII says that dynamically allocated resources should be acquired inside of the constructor and released inside the destructor.
 - This is what smart pointers do for example
- For compiling our projects we can and should use `Makefiles`

Last lecture

| Week | Tuesday | Thursday |
|------|--|---|
| 1 | SEPTEMBER 24 1. Welcome! Slides Policies | SEPTEMBER 26 2. Types and Structs Slides Code |
| 2 | OCTOBER 1 3. Initialization and References Slides | OCTOBER 3 4. Streams Slides Code |
| 3 | OCTOBER 8 5. Containers Slides | OCTOBER 10 6. Iterators and Pointers Slides |
| 4 | OCTOBER 15 7. Classes Slides Code | OCTOBER 17 8. Template Classes and Const Correctness Slides Code |
| 5 | OCTOBER 22 9. Template Functions Slides | OCTOBER 24 10. Functions and Lambdas Slides Code |
| 6 | OCTOBER 29 10. Operator Overloading Slides | OCTOBER 31 📅 12. Special Member Functions Slides |
| 7 | NOVEMBER 5 Democracy Day (No Class) | NOVEMBER 7 13. Move Semantics Slides Code |
| 8 | NOVEMBER 12 14. std::optional and Type Safety Slides Code | NOVEMBER 14 15. RAII, Smart Pointers, and Building C++ Projects |
| 9 | NOVEMBER 19 Optional: No Class, Extra Office Hours | NOVEMBER 21 Optional: No Class, Extra Office Hours |
| 10 | DECEMBER 3 Optional: No Class, Extra Office Hours | DECEMBER 5 Optional: No Class, Extra Office Hours |

The C++ Iceberg



[[source](#)]

Thank you for a great quarter!



fabioi@stanford.edu



jtrb@stanford.edu