CS106L Lecture 15: RAII, Smart Pointers, Building Projects 🔗

Fabio Ibanez, Jacob Roberts-Baca

Attendance

https://tinyurl.com/lastlectureW25

Plan

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

Plan

1. **RAII (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?

```cpp
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

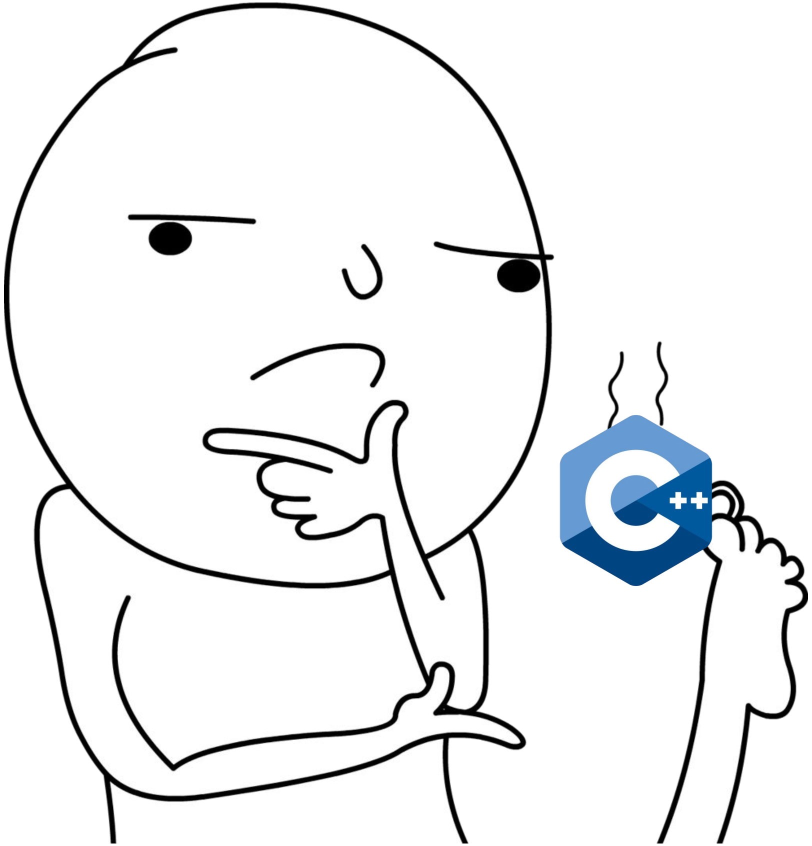
- Exceptions are a way of handling errors when they arise in code
- Exceptions are “thrown”
- However, we can write code that lets us handle exceptions so that we can continue in our code without necessarily erroring.

Exceptions

- Exceptions are a way of handling errors when they arise in code
- Exceptions are “thrown”
- However, we can write code that lets we can 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?

```cpp
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?

```cpp
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?

```cpp
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?

```cpp
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?

```cpp
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;
}
```

exception here means memory leak

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	`new`	`delete`
Files	`open`	`close`
Locks	`try_lock`	`unlock`
Sockets	`socket`	`close`

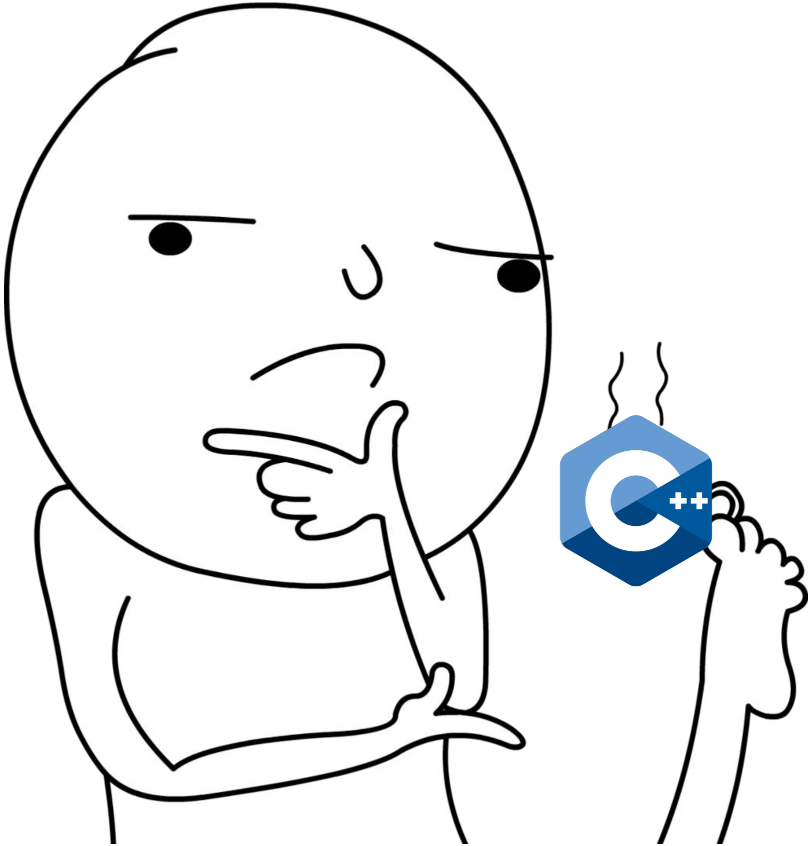
This is not unique to just pointers!

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

	Acquire	Release
Heap memory	`new`	`delete`
Files	`open`	`close`
	`try_lock`	`unlock`
	`socket`	`close`

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

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

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

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 ne

  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?

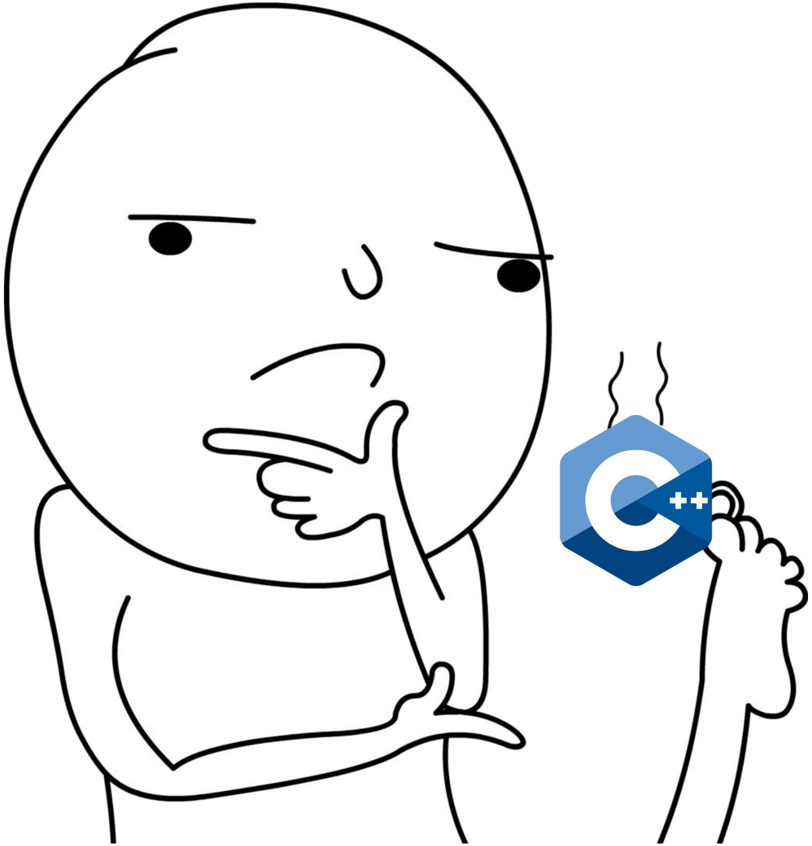
```
void cleanDatabase(mutex& databaseLock, map<int, int>& db) {
  lock_guard<mutex> lg(databaseLock);
  // no other thread or machine can change database
  // modify the database
  // if exception is throw, mutex is UNLOCKED!

  // 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

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

Smart Pointers

RAII for locks → `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?

```cpp
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

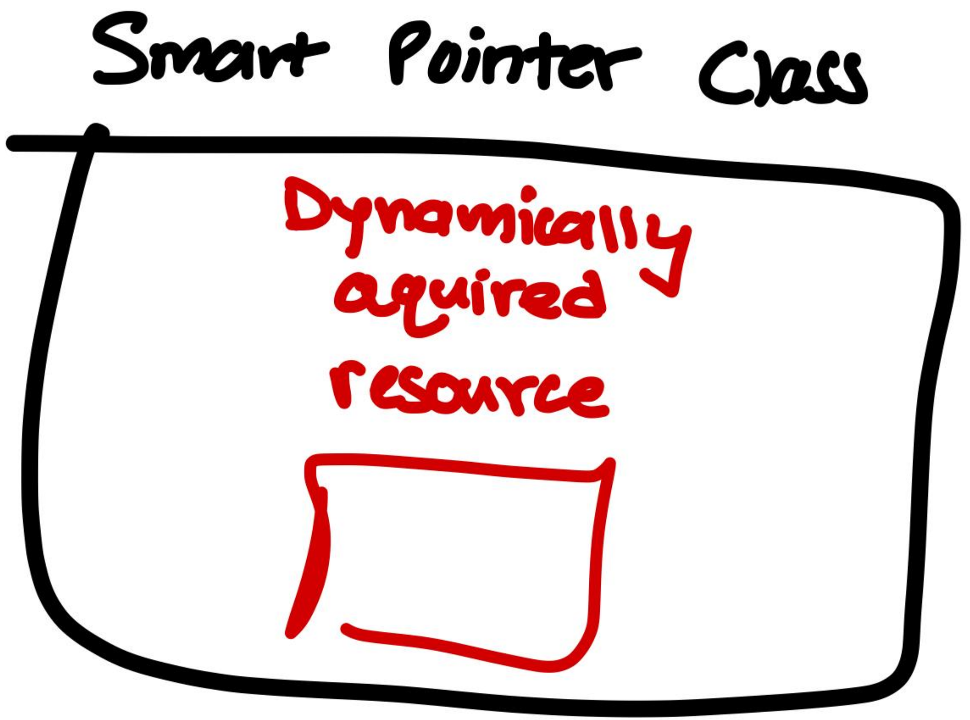
RAII for locks → `lock_guard`

- 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



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 ***<u>underlying memory</u>*** goes out of scope

Visualizing smart pointers

RAII for memory → We can do the same 🥳

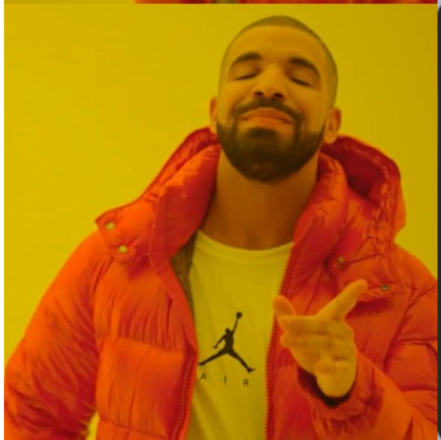
- These “wrapper” pointers are called “smart pointers”!

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 ***<u>underlying memory</u>*** goes out of scope
- **`std::weak_ptr`**
  - A class of pointers designed to ***<u>mitigate circular dependencies</u>***
    - More on these in a bit

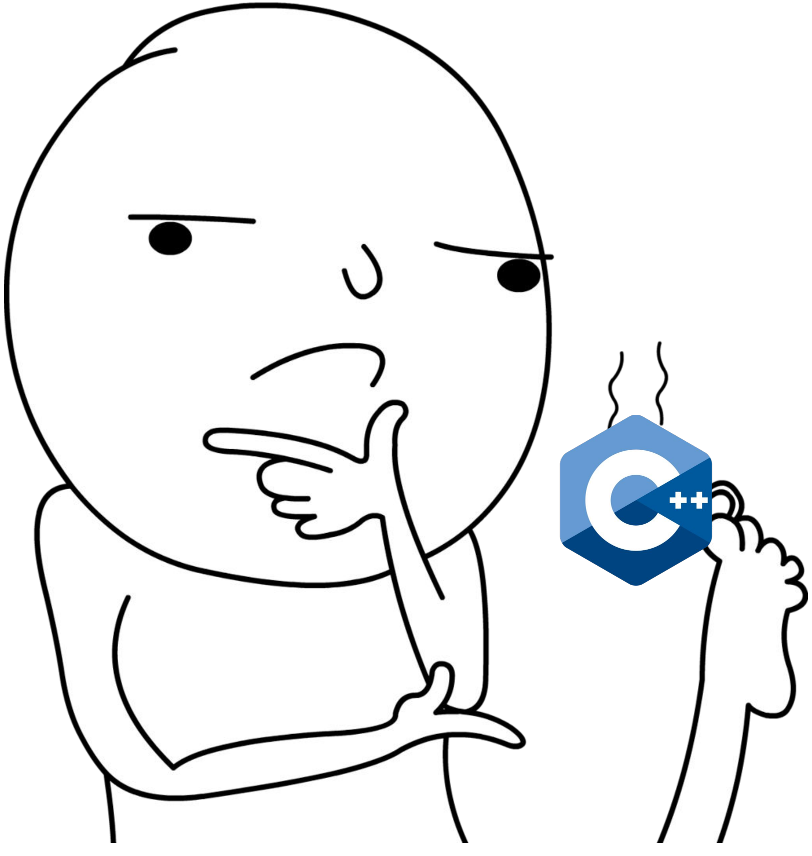
What does this look like?

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



```cpp
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?

```cpp
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?

```cpp
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.

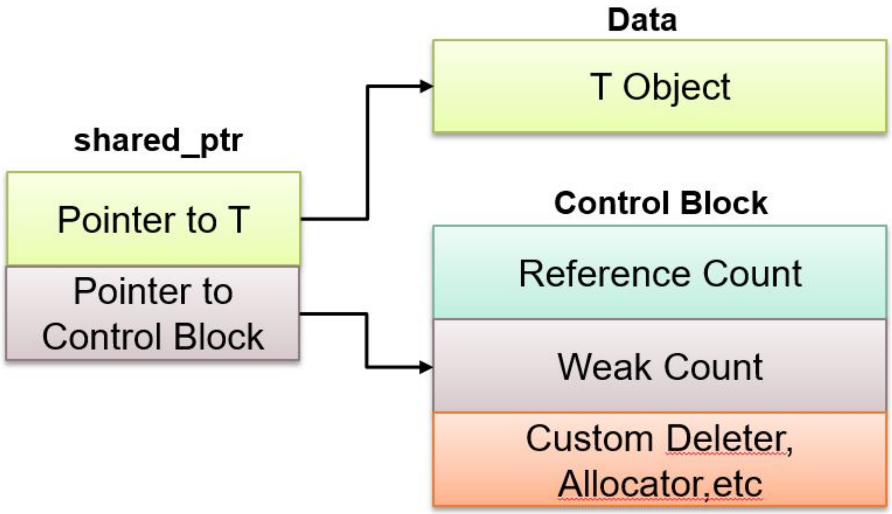
Problem: The copy points to deallocated memory!

`std::shared_ptr`

Shared pointers get around our issue of trying to copy **`std::unique_ptr`**’s by not deallocating the underlying memory until ***<u>all</u>*** 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 ***<u>all</u>*** 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};

hared_ptr<T> sharedPtr{new T};

eak_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.

```cpp
#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

```cpp
#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.

`std::weak_ptr` bad example

```cpp
#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

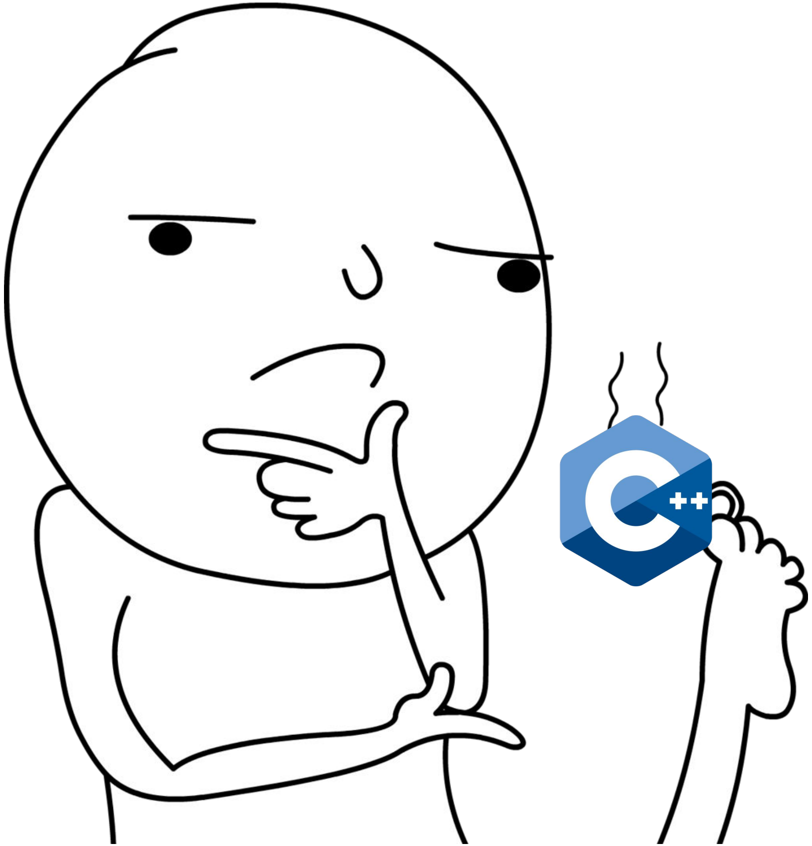
std::weak_ptr good example

```cpp
#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

What questions do we have?



Plan

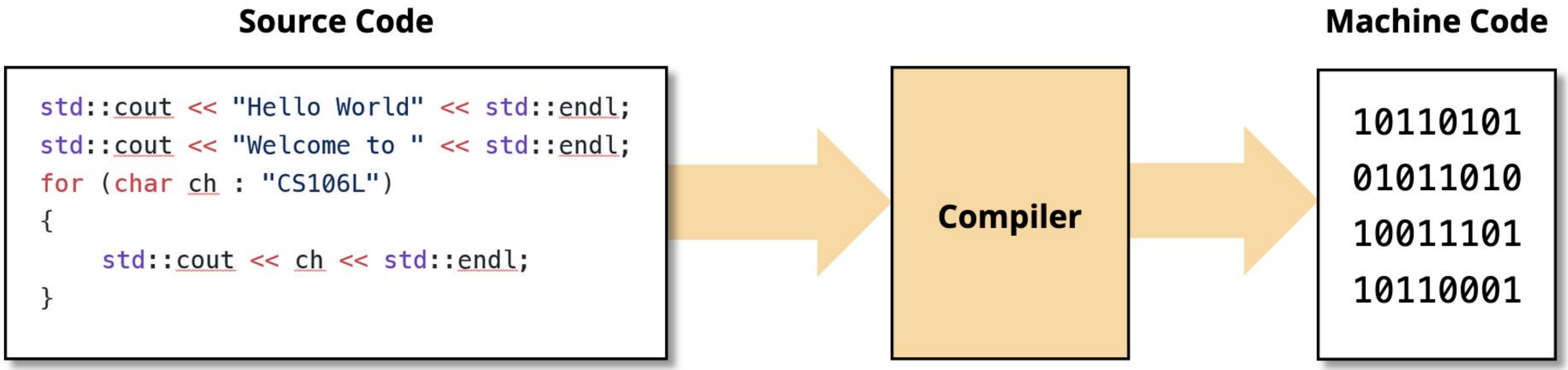
1. RAII (Resource Acquisition Is Initialization)
2. Smart Pointers
3. **Building C++ projects**

Compilation Crash Course

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

Compilation Crash Course

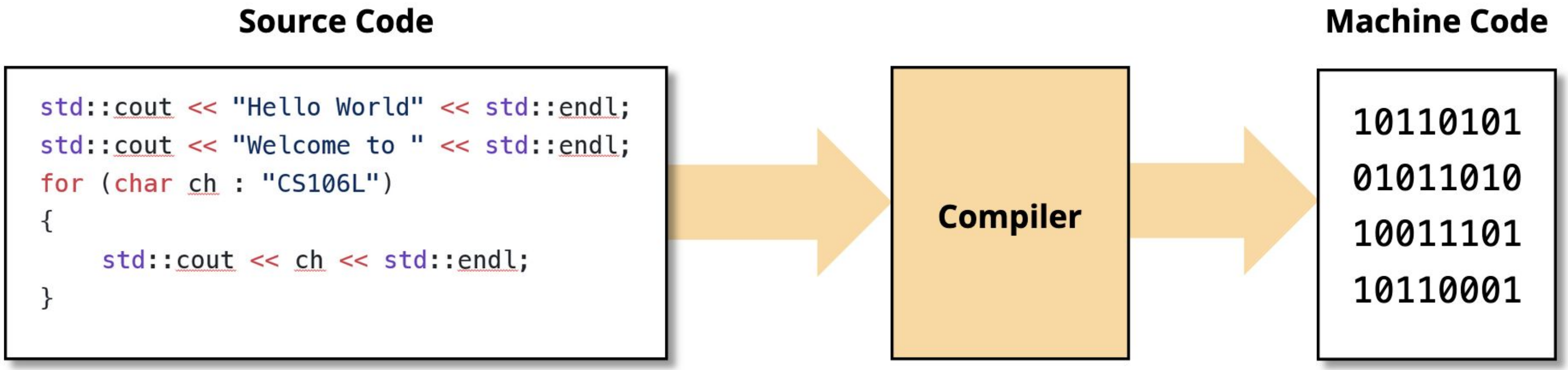
When we write C++ code, it needs to be translated into a form our computer understands it




```
$ 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




```
$ 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

```
$ 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
```

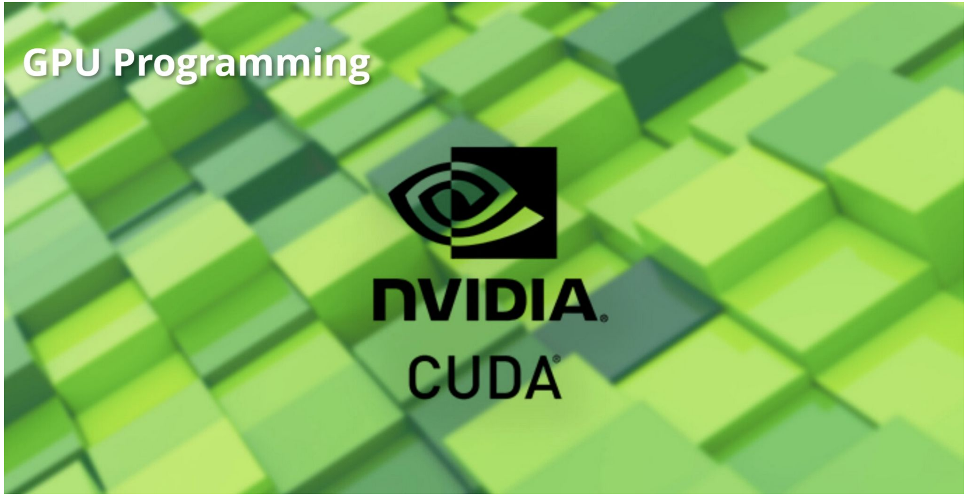
This means that you're going to give a specific name to your executable

Compilation Crash Course

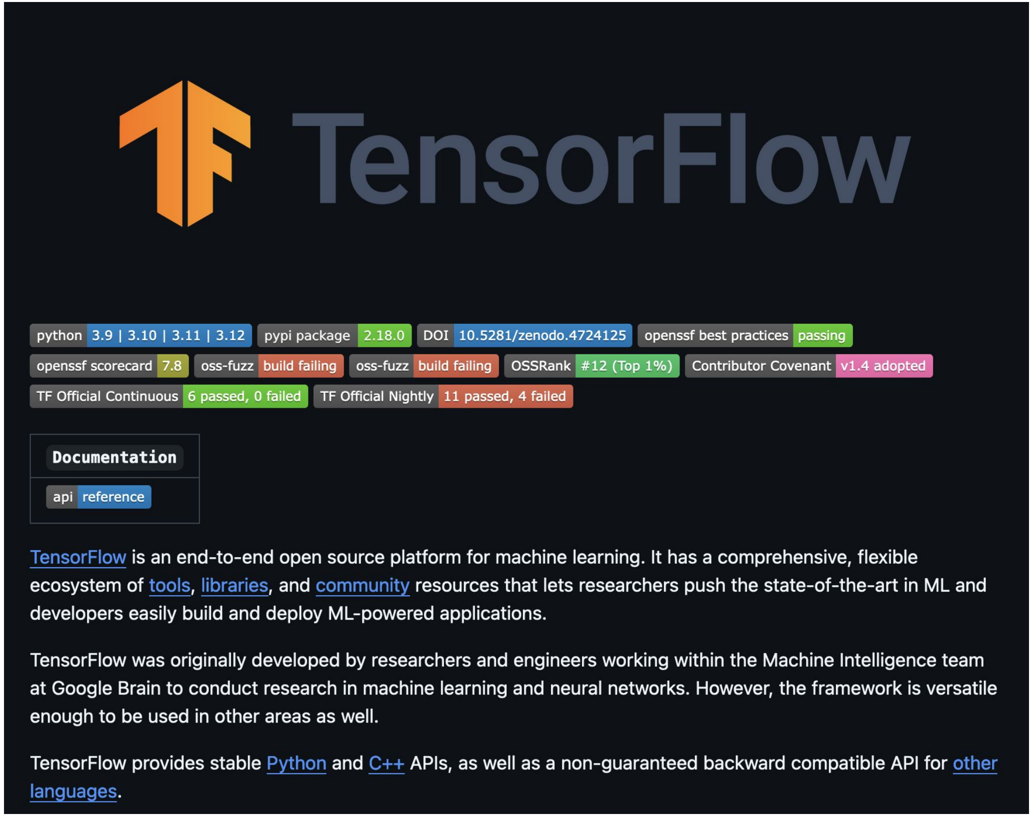
When we write C++ code, it needs to be translated into a form our computer understands it

```
$ 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`



Even the masterpiece
among us



TensorFlow

python 3.9 | 3.10 | 3.11 | 3.12 · pypi package 2.18.0 · DOI 10.5281/zenodo.4724125 · openssf best practices passing · openssf 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

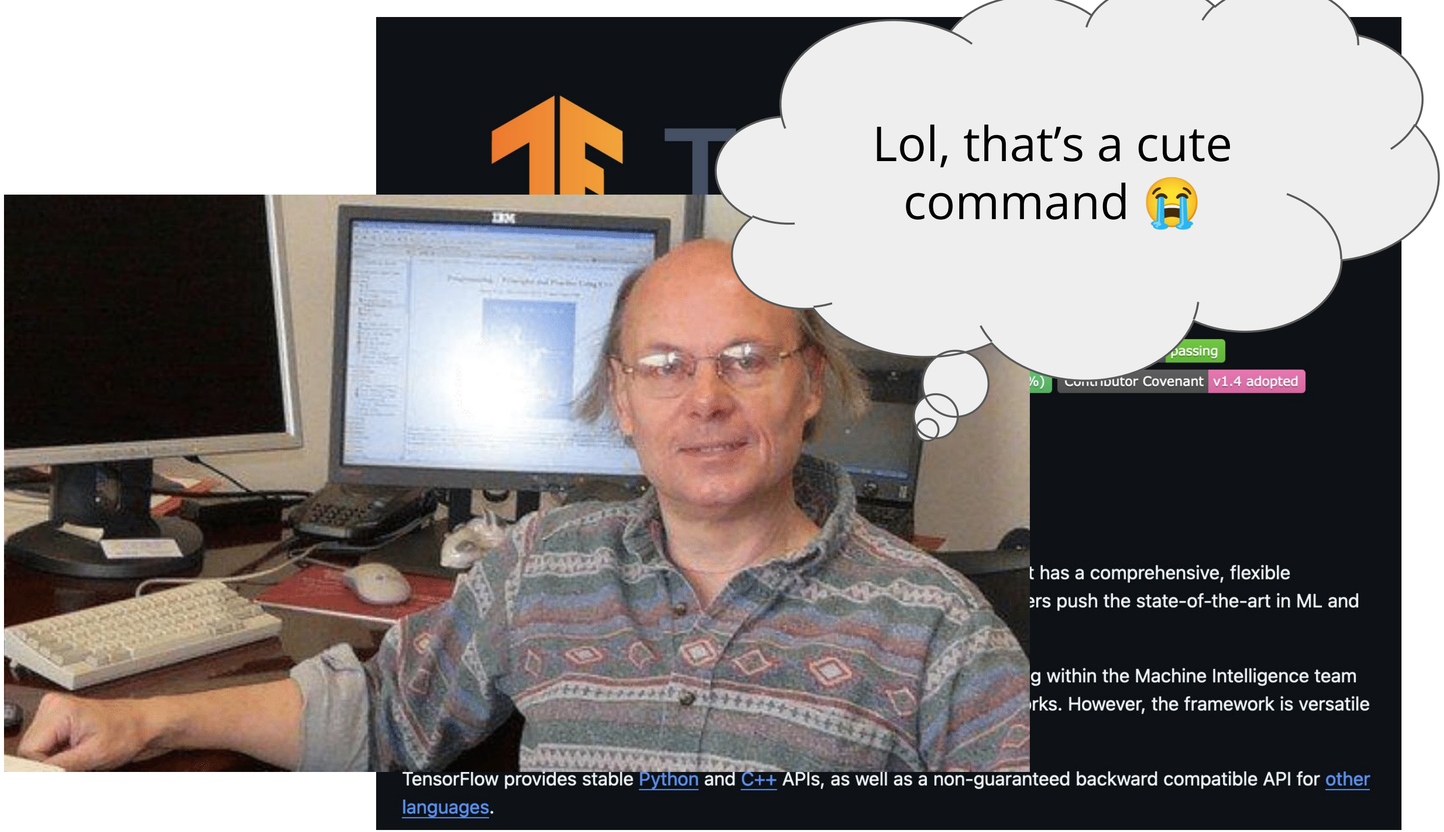
Documentation
api reference

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**




```
$ 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!

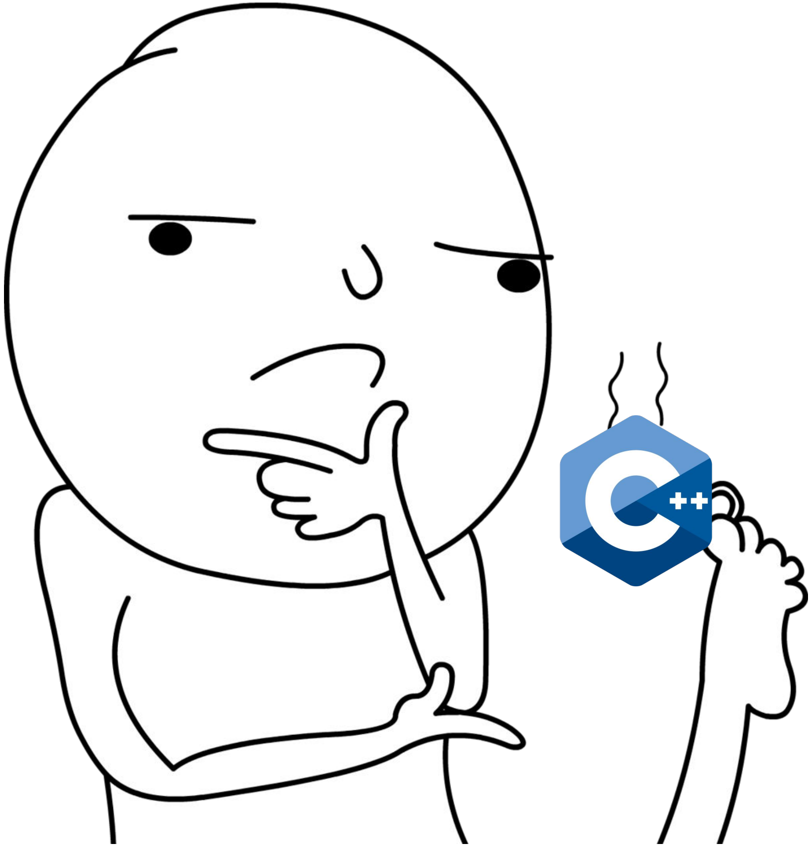
- You can specify what compiler you want to use
- In order to use **`make`** you need to have a **`Makefile`**

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

```makefile
# Compiler
CXX = g++

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

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

# Default target
all:
	$(CXX) $(CXXFLAGS) $(SRCS) -o $(TARGET)

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

This is an example `Makefile` for our lecture 7 code

What questions do we have?



CMake

CMake is a build system generator.

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

Is like a higher level abstraction for `Makefiles`

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(cs106l_classes)

set(CMAKE_CXX_STANDARD 20)

file(GLOB SRC_FILES "*.cpp")

add_executable(main ${SRC_FILES})
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(cs106l_classes)

set(CMAKE_CXX_STANDARD 20)

file(GLOB SRC_FILES "*.cpp")

add_executable(main ${SRC_FILES})
```

This command tells CMAKE to set the C++ compiler to C++20

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(cs106l_classes)

set(CMAKE_CXX_STANDARD 20)

file(GLOB SRC_FILES "*.cpp")

add_executable(main ${SRC_FILES})
```

This GLOB command is telling the CMAKE program to do a wildcard search for all files that have the pattern "`*.cpp`"

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

project(cs106l_classes)

set(CMAKE_CXX_STANDARD 20)

file(GLOB SRC_FILES "*.cpp")

add_executable(main ${SRC_FILES})
```

This command adds all of the source files of our program into the executable

To use `CMAKE`

1. You need to have a `CMakeLists.txt` file in your project's root directory
2. Make a `build` folder (`mkdir build`) within your project!
3. Go into the `build` folder (`cd build`)
4. Run `cmake ..`
    a. This command runs `cmake` using the `CMakeLists.txt` in your project's root folder!
    b. This generates a `Makefile`
5. Run `make`
6. Execute your program using `./main` as usual

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 to for example

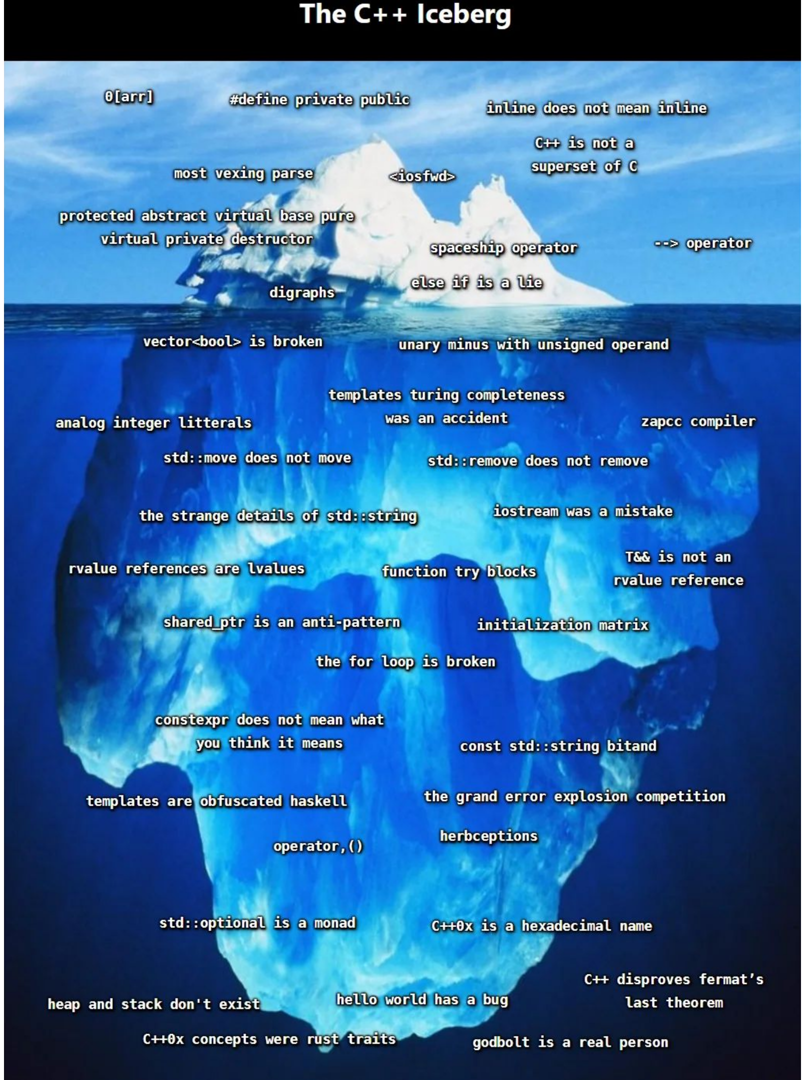
- For compiling our projects we can and should use `Makefiles`

- For making our `Makefiles` we can and should use `CMAKE`

Last lecture

Schedule

Week	Tuesday	Thursday
1	January 7 1. Welcome! Slides Policies	January 9 2. Types and Structs Slides Code Reader
2	January 14 3. Initialization and References Slides Reader	January 16 4. Streams Slides
3	January 21 5. Containers Slides Reader	January 23 6. Iterators and Pointers Slides
4	January 28 7. Classes Slides Code	January 30 8. Template Classes and Const Correctness Slides Code
5	February 4 9. Template Functions Slides Code	February 6 10. Functions and Lambdas Slides Code
6	February 11 11. Operator Overloading Slides Code	February 13 12. Special Member Functions Slides
7	February 18 13. Move Semantics Slides	February 20 14. `std::optional` and Type Safety Slides Code
8	February 25 15. RAII, Smart Pointers, and Building C++ Projects	February 27 Optional: No Class, Extra Office Hours
9	March 4 Optional: No Class, Extra Office Hours	March 6 Optional: No Class, Extra Office Hours
10	March 11 Optional: No Class, Extra Office Hours	March 13 Optional: No Class, Extra Office Hours


The C++ Iceberg
0[arr]
#define private public
inline does not mean inline
C++ is not a superset of C
most vexing parse
<iosfwd>
protected abstract virtual base pure virtual private destructor
spaceship operator
--> operator
else if is a lie
digraphs
vector<bool> is broken
unary minus with unsigned operand
templates turing completeness was an accident
analog integer litterals
zapcc compiler
std::move does not move
std::remove does not remove
the strange details of std::string
iostream was a mistake
rvalue references are lvalues
function try blocks
T&& is not an rvalue reference
shared_ptr is an anti-pattern
initialization matrix
the for loop is broken
constexpr does not mean what you think it means
const std::string bitand
templates are obfuscated haskell
the grand error explosion competition
herbceptions
operator,()
std::optional is a monad
C++0x is a hexadecimal name
C++ disproves fermat's last theorem
heap and stack don't exist
hello world has a bug
C++0x concepts were rust traits
godbolt is a real person

[source]

Thank you for a great quarter!

fabioi@stanford.edu

jtrb@stanford.edu