CS106L Lecture 15:

RAII and Smart Pointers 🔗

Winter 2024

Fabio Ibanez, Haven Whitney

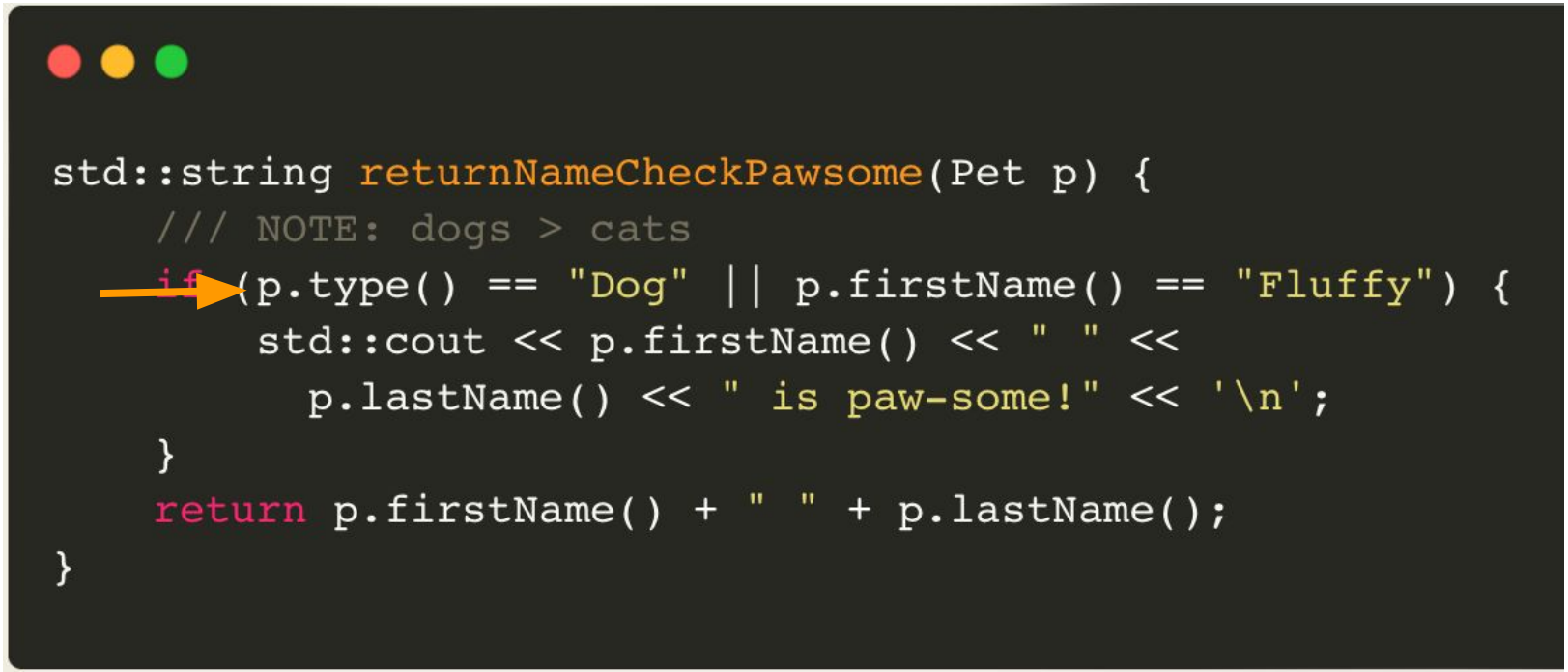
Attendance

http://tinyurl.com/raiiW24

Plan

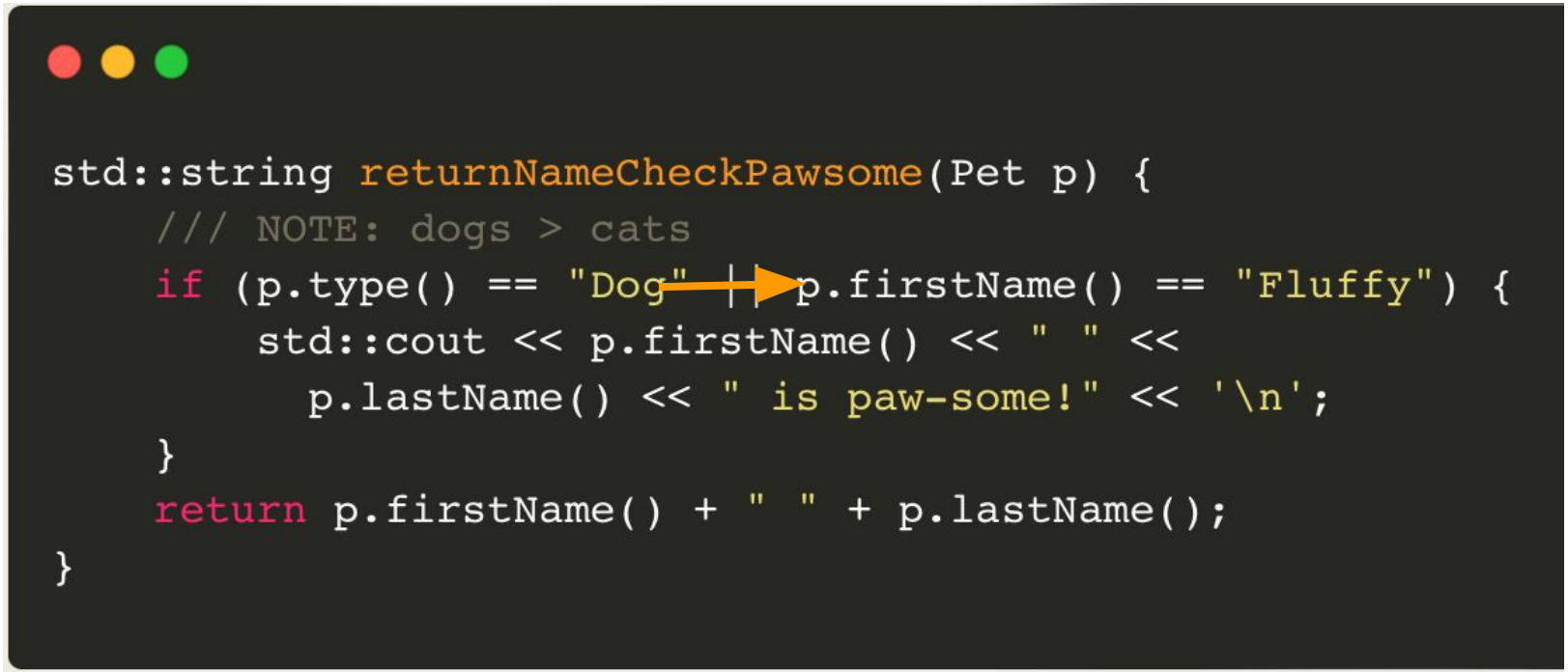
1. RAII (Resource Acquisition Is Initialization)
2. Smart Pointers
3. Building C++ projects

How many code paths?



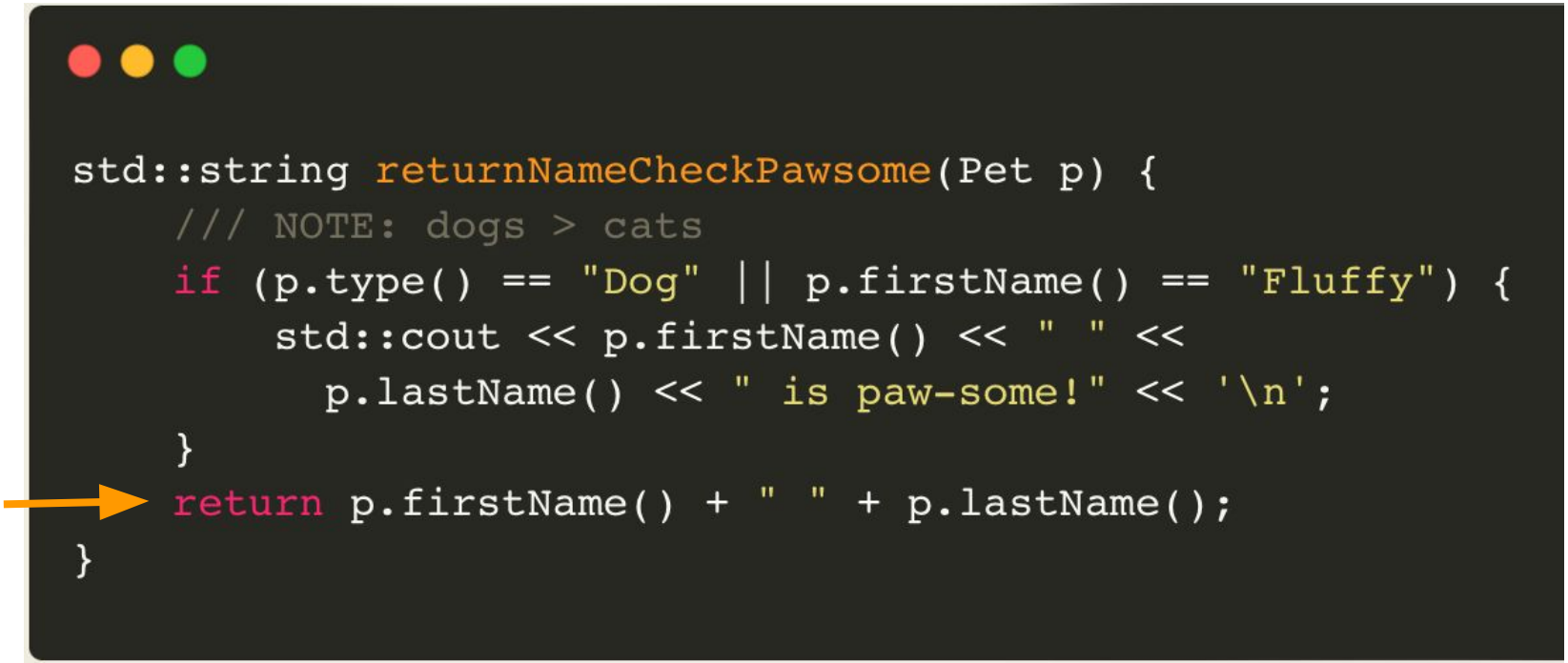
```
std::string returnNameCheckPawsome(Pet p) {
    /// NOTE: dogs > cats
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {
        std::cout << p.firstName() << " " <<
          p.lastName() << " is paw-some!" << '\n';
    }
    return p.firstName() + " " + p.lastName();
}
```

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?

```
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

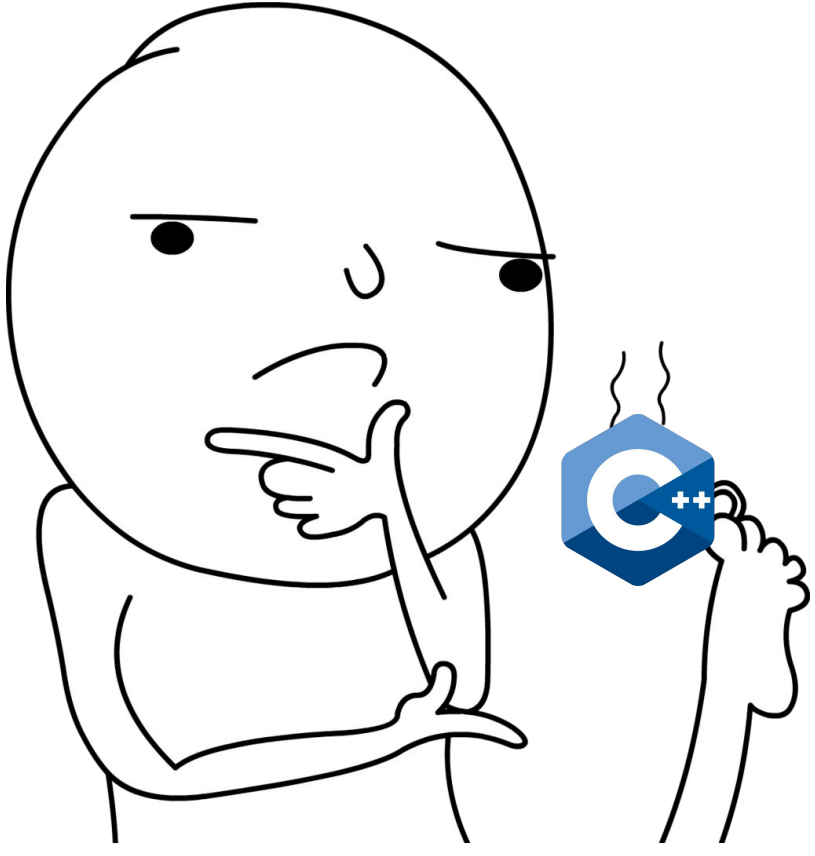
- Exceptions are a way of handling errors when they arise in code
- Exceptions are “thrown”
- However, we can write code that lets us handle exceptions so that we can continue in our code without necessarily erroring.

Exceptions

- Exceptions are a way of handling errors when they arise in code
- Exceptions are “thrown”
- However, we can write code that lets u
  can continue in our code without nece
- 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?

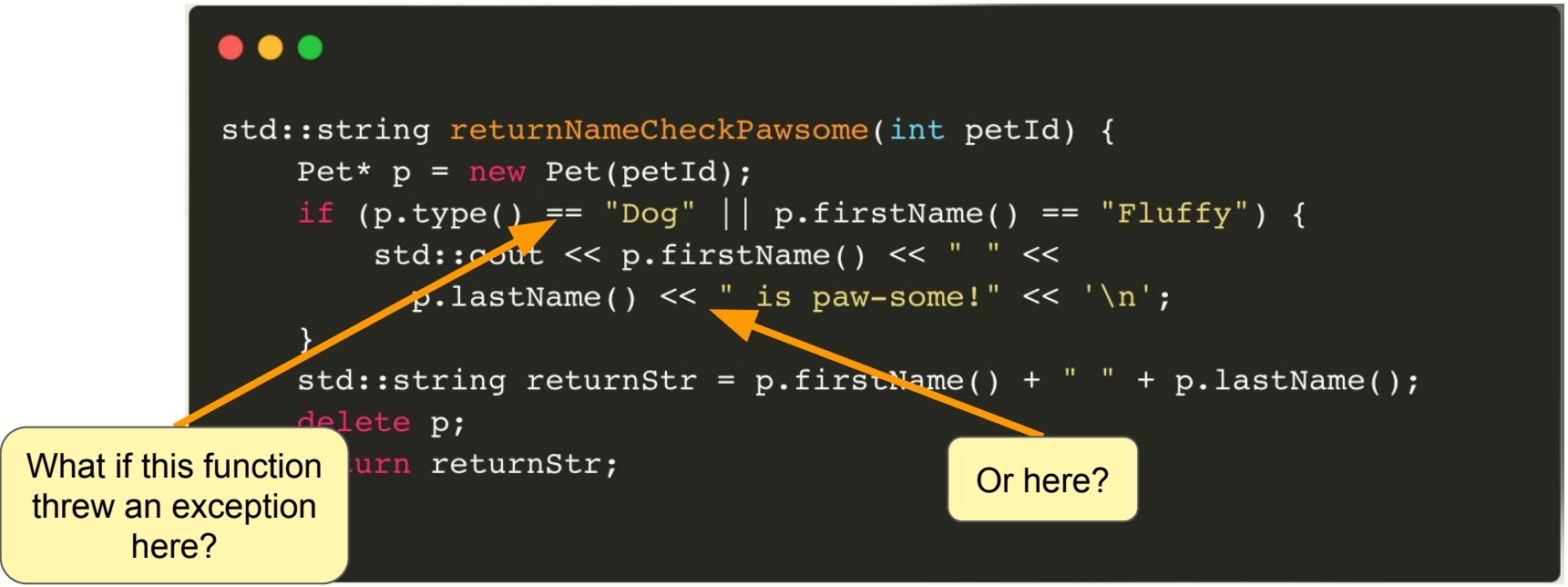
```
std::string returnNameCheckPawsome(int petId) {
    Pet* p = new Pet(petId);
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {
        std::cout << p.firstName() << " " <<
          p.lastName() << " is paw-some!" << '\n';
    }
    std::string returnStr = p.firstName() + " " + p.lastName();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

What could go wrong?

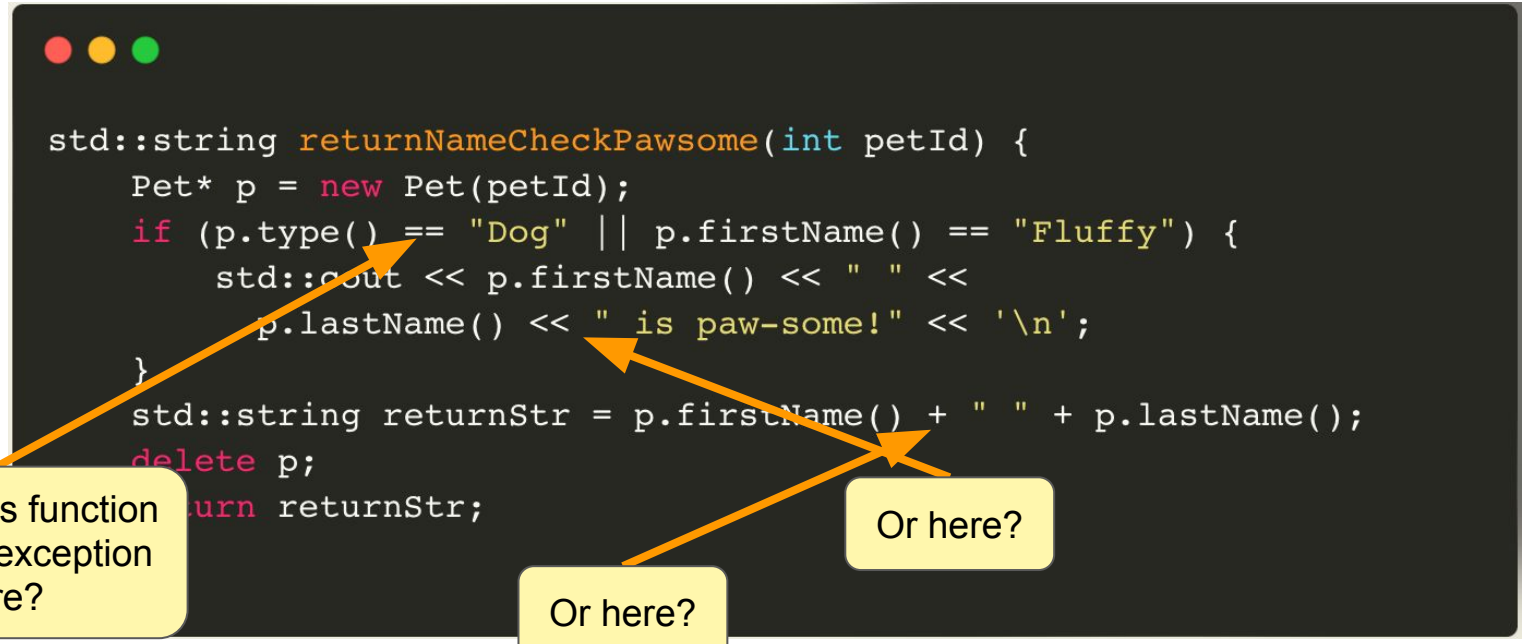
```
std::string returnNameCheckPawsome(int petId) {
    Pet* p = new Pet(petId);
    if (p.type() == "Dog" || p.firstName() == "Fluffy") {
        std::cout << p.firstName() << " " <<
            p.lastName() << " is paw-some!" << '\n';
    }
    std::string returnStr = p.firstName() + " " + p.lastName();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

What if this function threw an exception here?

What could go wrong?



What could go wrong?



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

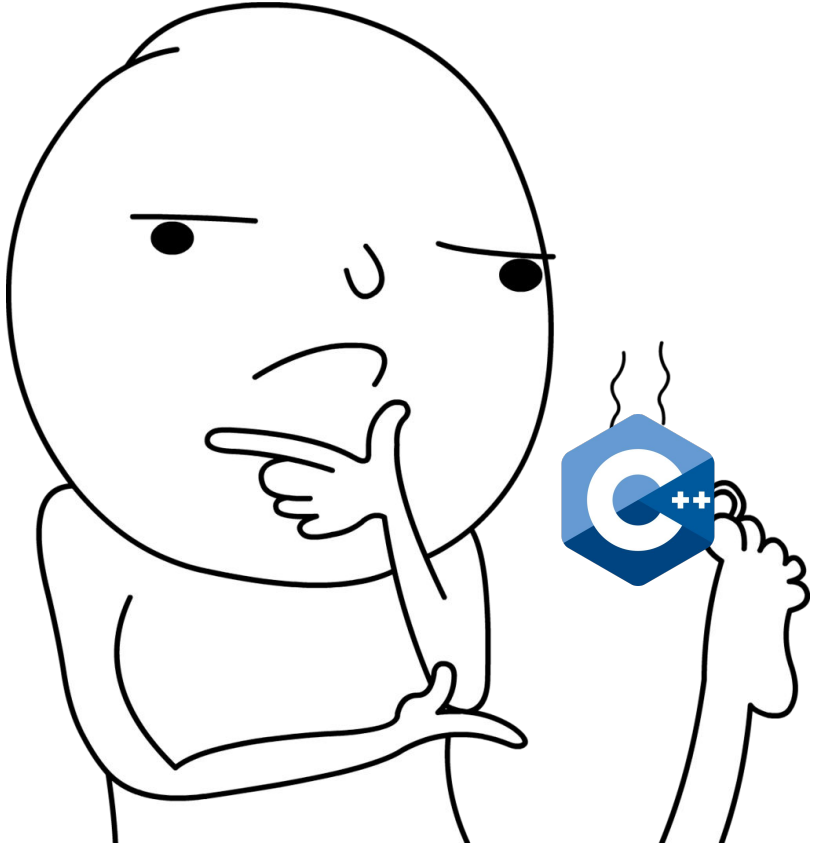
This is not unique to just pointers!

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 (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

RAII: why tho?

RAII: Resource Acquisition is Initialization

It turns out that by abiding by the RAII 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?

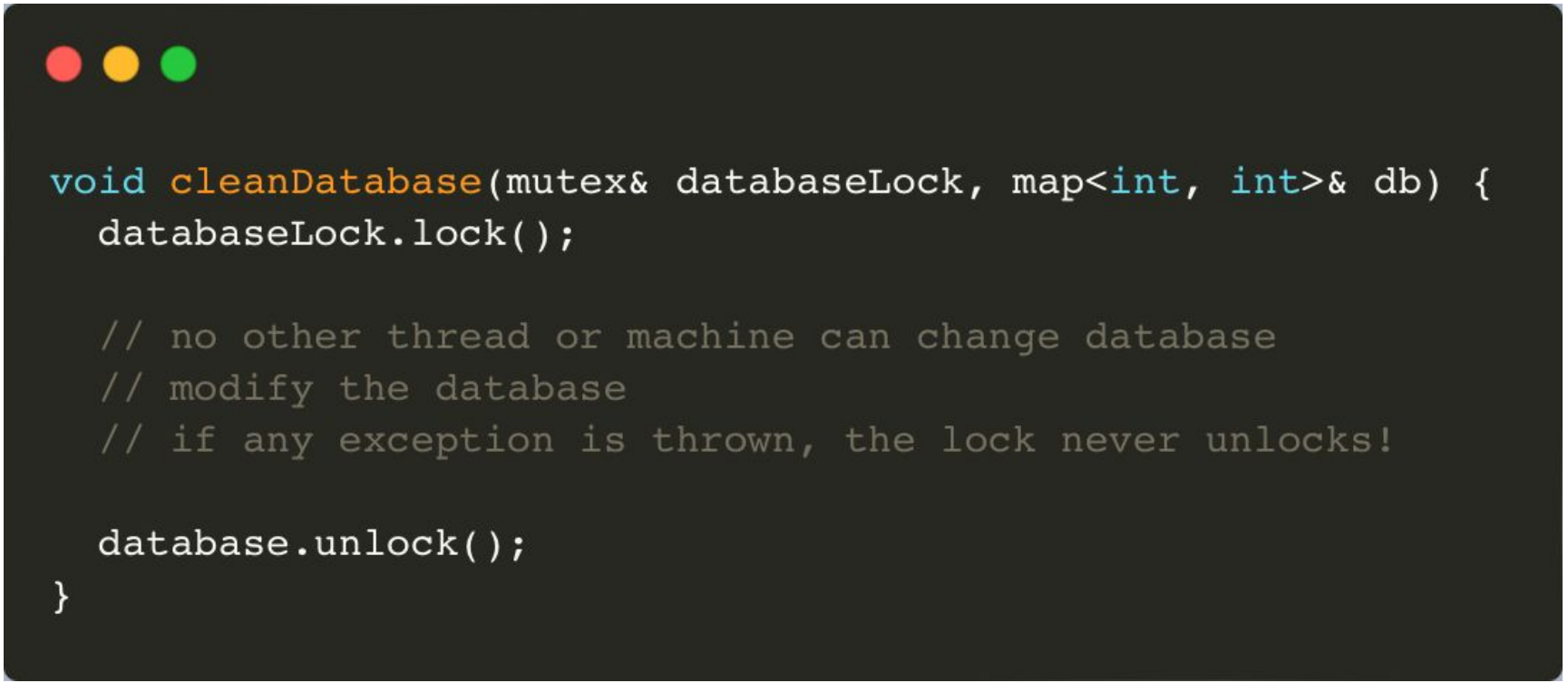
```
void printFile() {
  ifstream input;
  input.open("hamlet.txt");

  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 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?

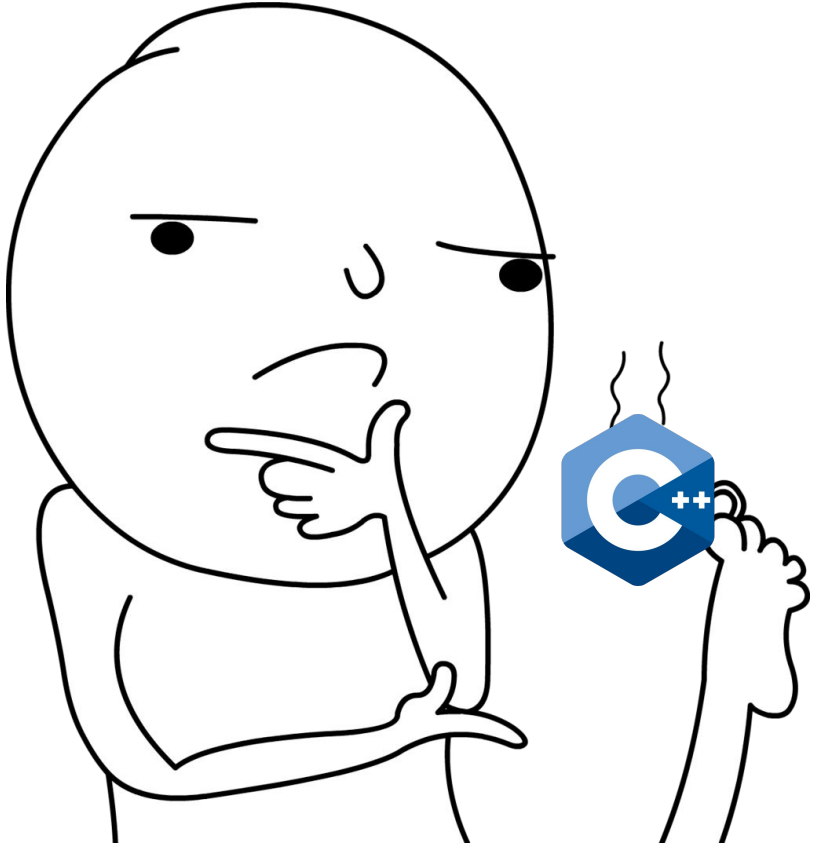
```
void cleanDatabase(mutex& databaseLock, map<int, int>& db) {
  lock_guard<mutex> lg(databaseLock);
  // no other thread or machine can change database
  // modify the database
  // if exception is throw, mutex is UNLOCKED!

  // 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?

```
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

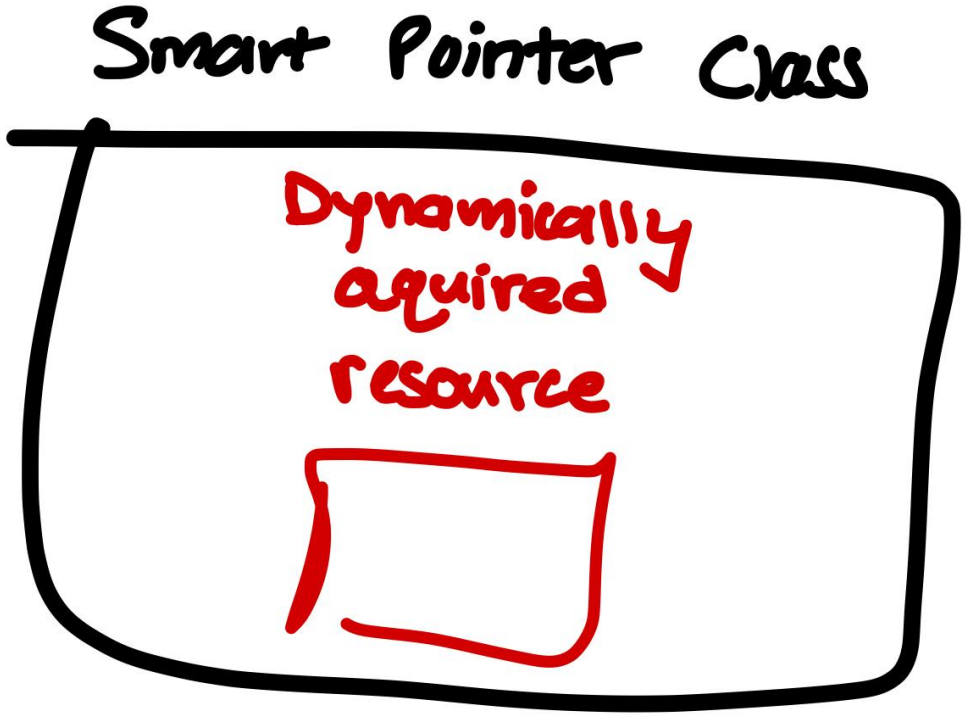
RAII for locks → `lock_guard`

- 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 ***underlying memory*** goes out of scope

Visualizing smart pointers

RAII for memory → We can do the same 🥳

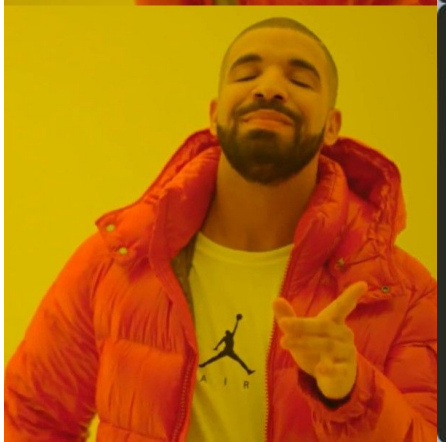
- These “wrapper” pointers are called “smart pointers”!

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`**
  - This is a way to *try* to have ownership over an object that may or may not exist

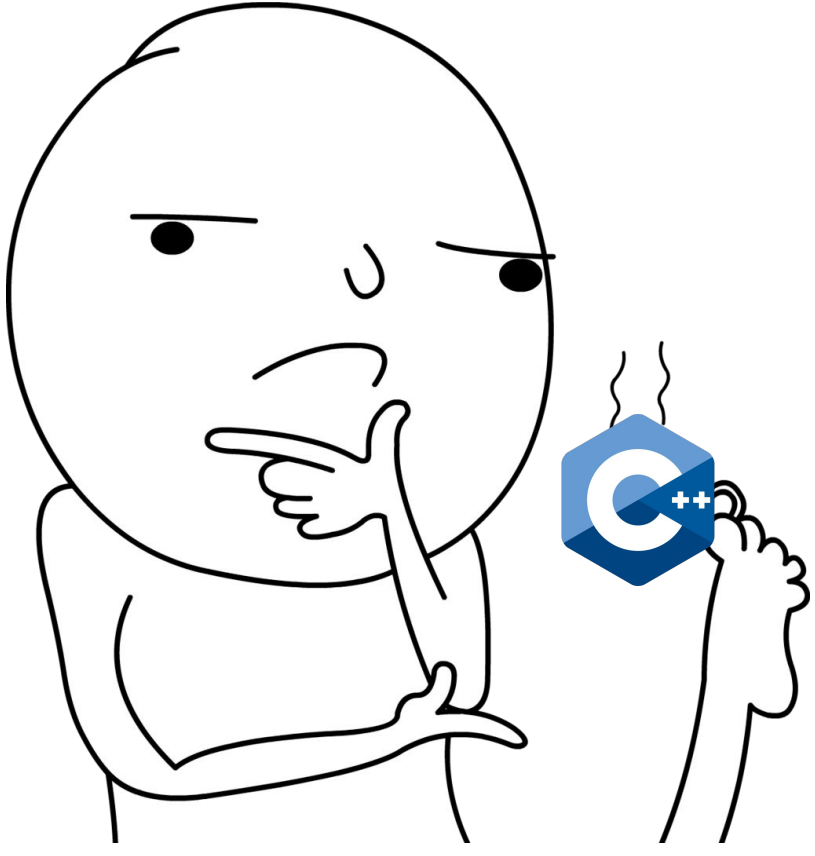
What does this look like?

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

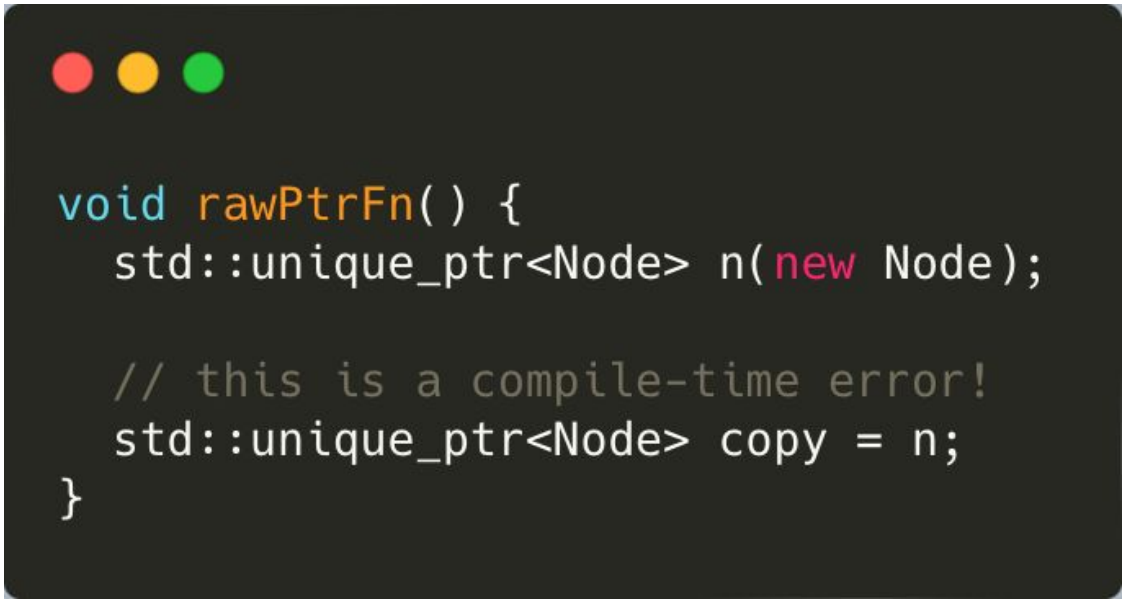


```
void rawPtrFn() {
  std::unique_ptr<Node> n(new Node);
  // do something with n
  // n automatically freed
}
```

What questions do we have?

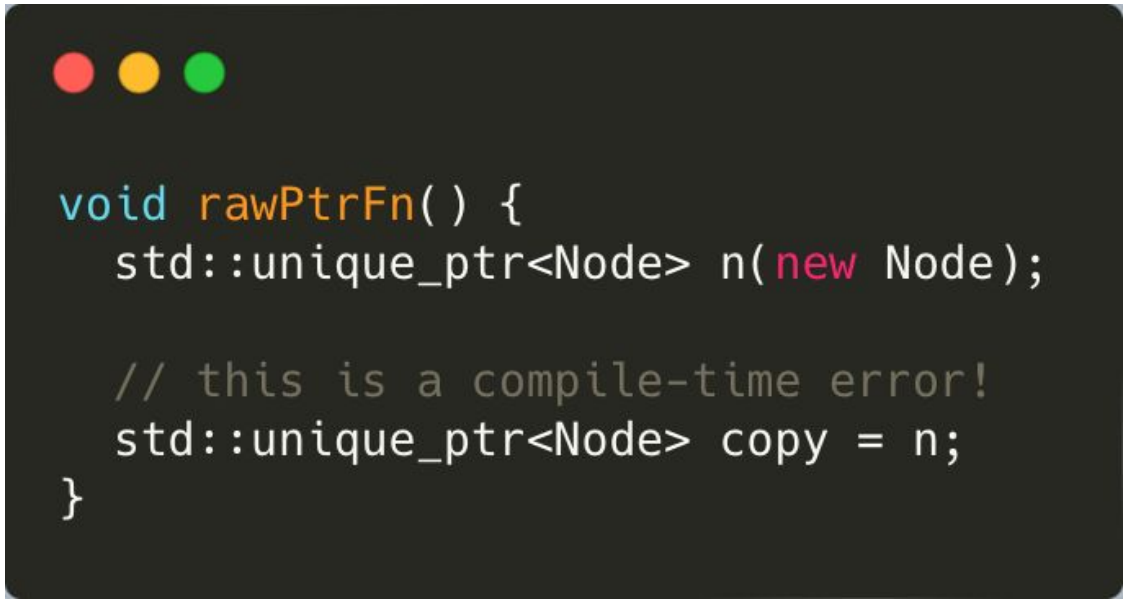


Remember we can’t copy unique pointers



```
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.

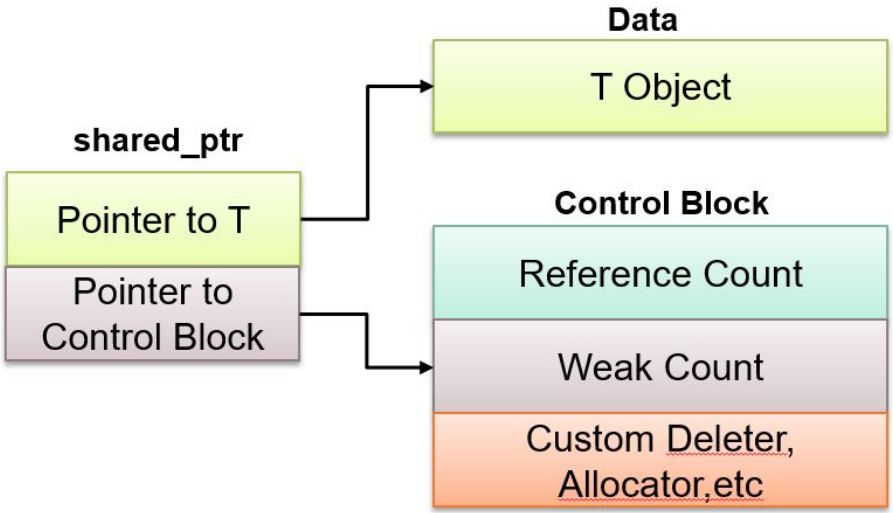
Problem: The copy points to deallocated memory!

`std::shared_ptr`

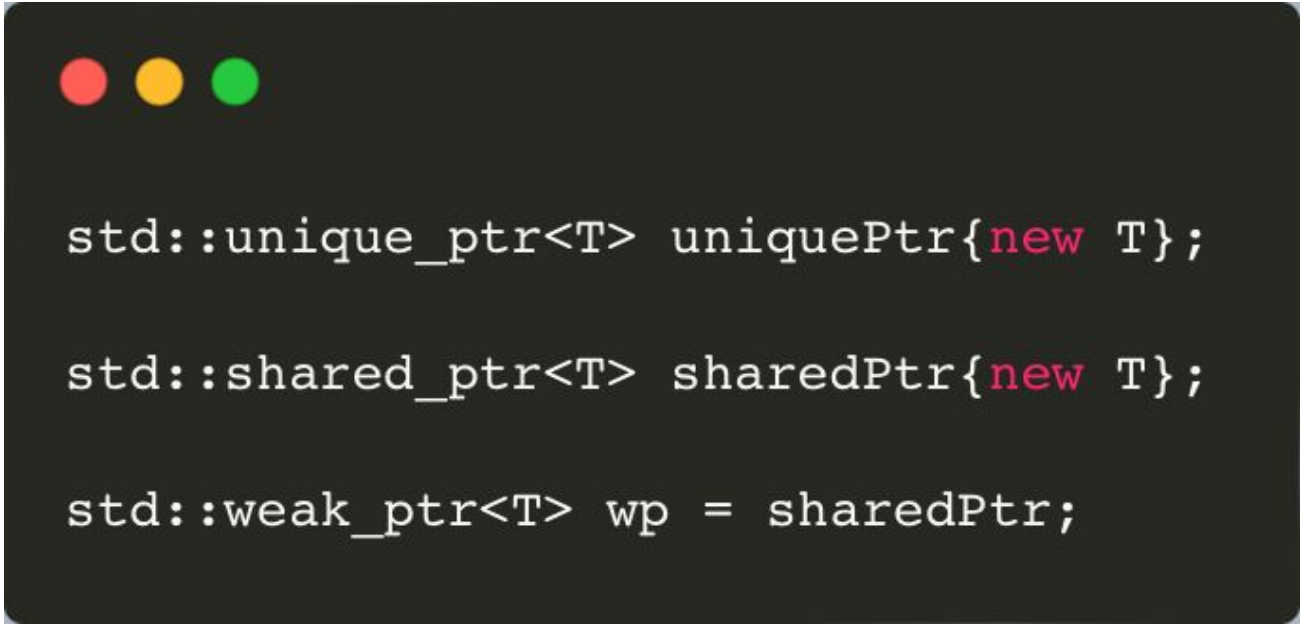
Shared pointers get around our issue of trying to copy `std::unique_ptr`'s by not deallocating the underlying memory until ***all*** shared pointers go out of scope!

`std::shared_ptr`

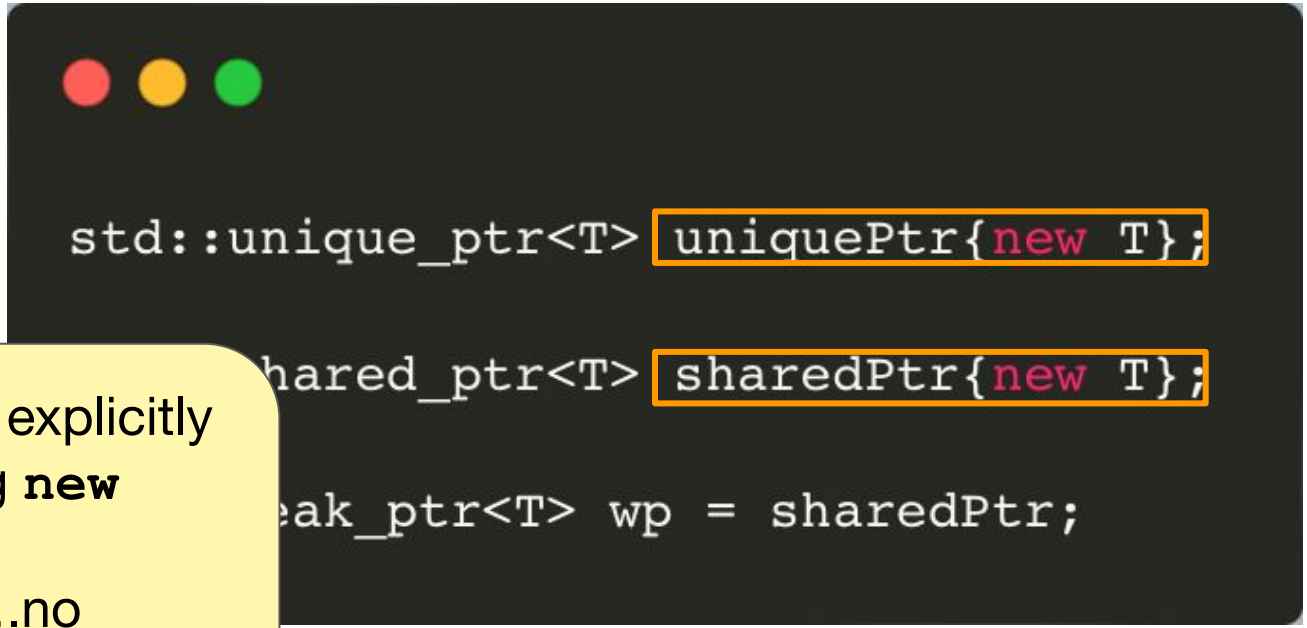
Shared pointers get around our issue of trying to copy `std::unique_ptr`'s by not deallocating the underlying memory until ***all*** shared pointers go out of scope!



Initializing smart pointers!



Initializing smart pointers!



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`**!

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

This is called ***compiling***

Compilation Crash Course

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

This is called ***compiling***

A compiler does this, and a compiler is just a program that translates code from one language to another

Compilation Crash Course

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

This is called ***compiling***

A compiler does this, and a compiler is just a program that translates code from one language to another

A few common ones include g++ and clang.

What this looks like

Preprocessing

Preprocessing

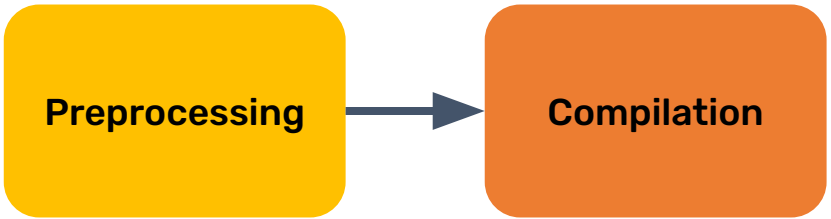
Preprocessing

In this stage, the code is cleaned up before actually compiling

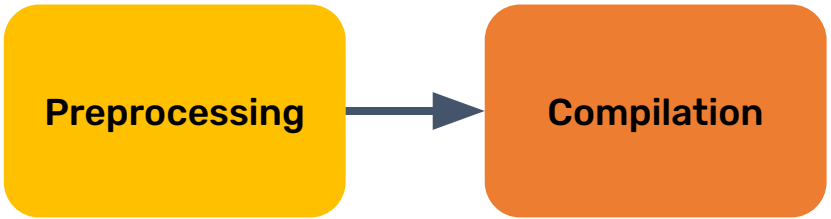
- Any preprocessor commands that begin with # are handled
- Comments and excess whitespace are stripped

main.cpp

The compilation stage

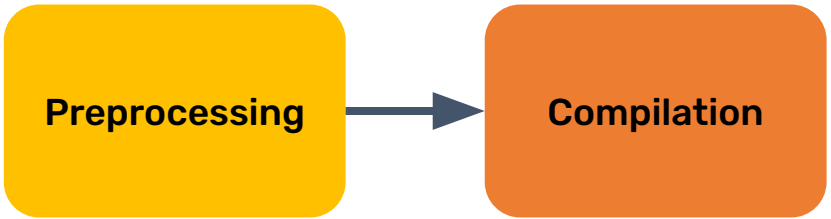


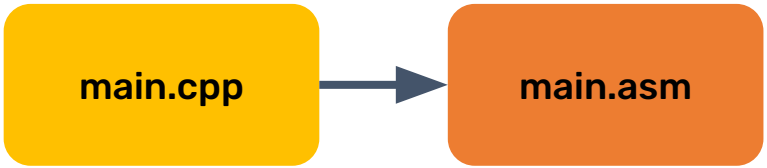
The compilation stage



This is where the translation actually happens – code is translated into assembly which our computer can read.

The compilation stage

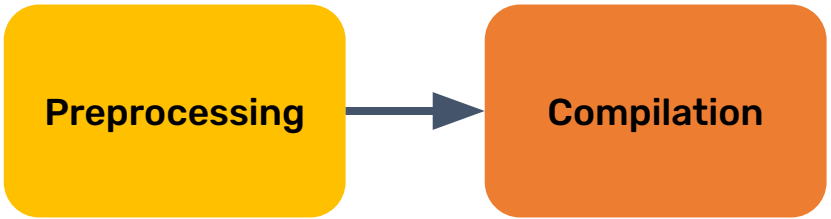


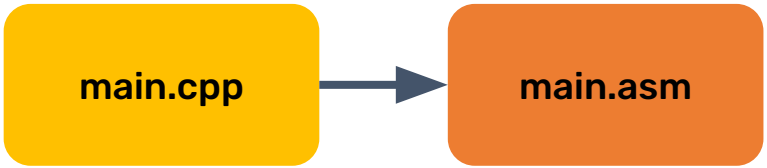


This is where the translation actually happens – code is translated into assembly which our computer can read.

- C++ to assembly translation

The compilation stage

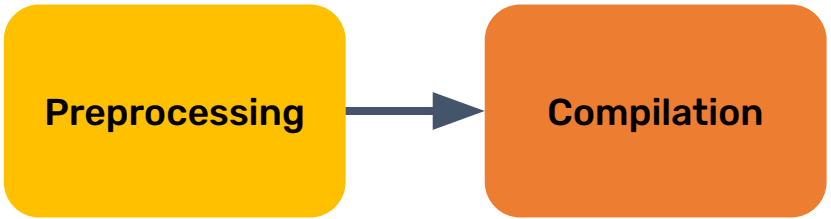


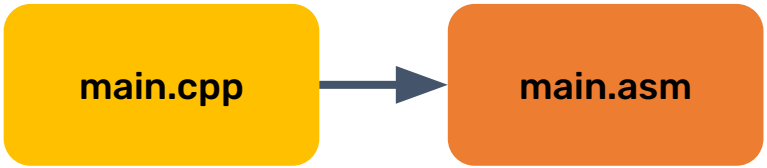


This is where the translation actually happens – code is translated into assembly which our computer can read.

- C++ to assembly translation
- If code is already in assembly, it is not translated

The compilation stage





This is where the translation actually happens – code is translated into assembly which our computer can read.

- C++ to assembly translation
- If code is already in assembly, it is not translated
- Assembly is oftentimes machine-specific.

The assembly stage

The assembly stage

In the assembly stage the assembler converts assembly to object code!

The assembly stage

In the assembly stage the assembler converts assembly to object code!

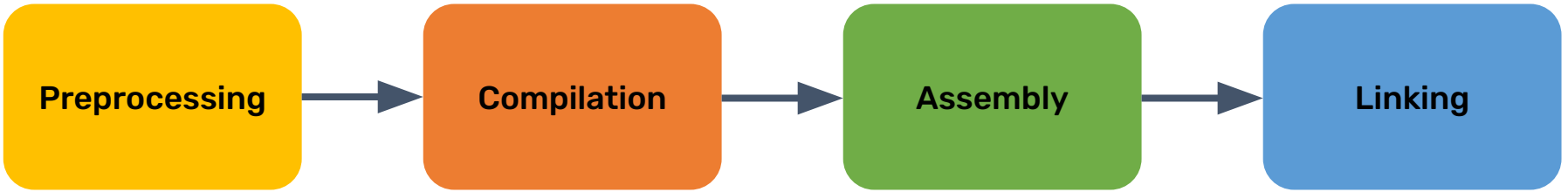
- Object code is actual machine readable code the processor can run.

The assembly stage

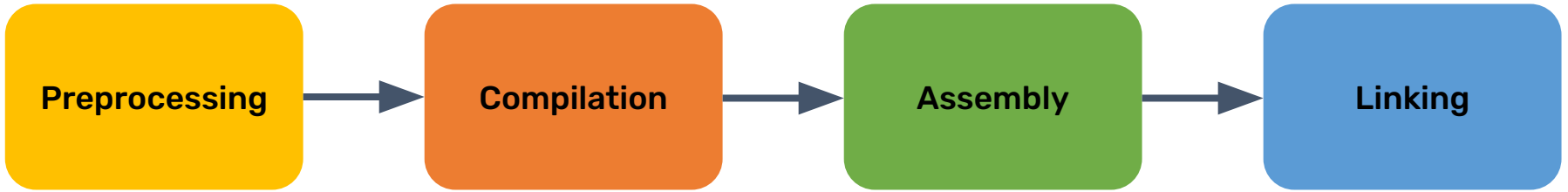
In the assembly stage the assembler converts assembly to object code!

- Object code is actual machine readable code the processor can run.
- Assembly is the human-readable version of the object code

The linking stage

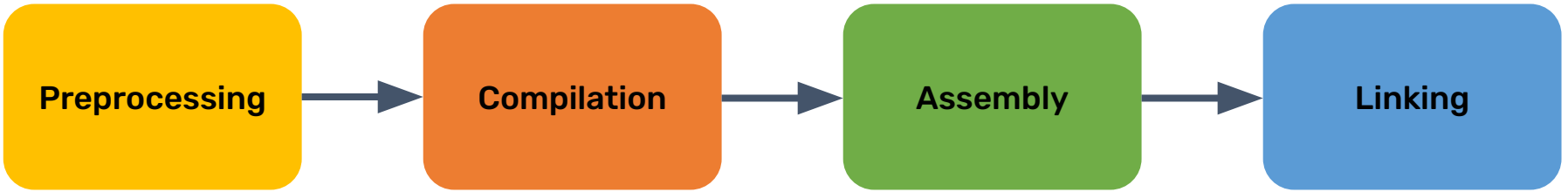


The linking stage



The linker takes each piece of object code and arranges it into one program

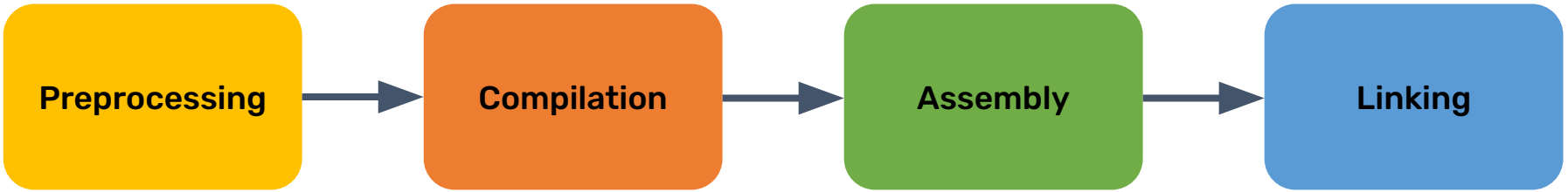
The linking stage



The linker takes each piece of object code and arranges it into one program

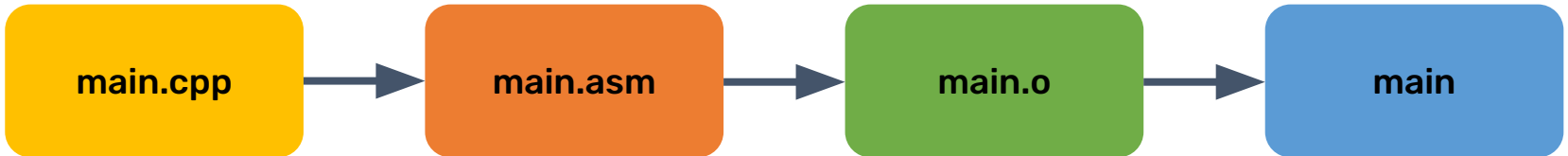
- Think about this stage as “stitching” things together

The linking stage

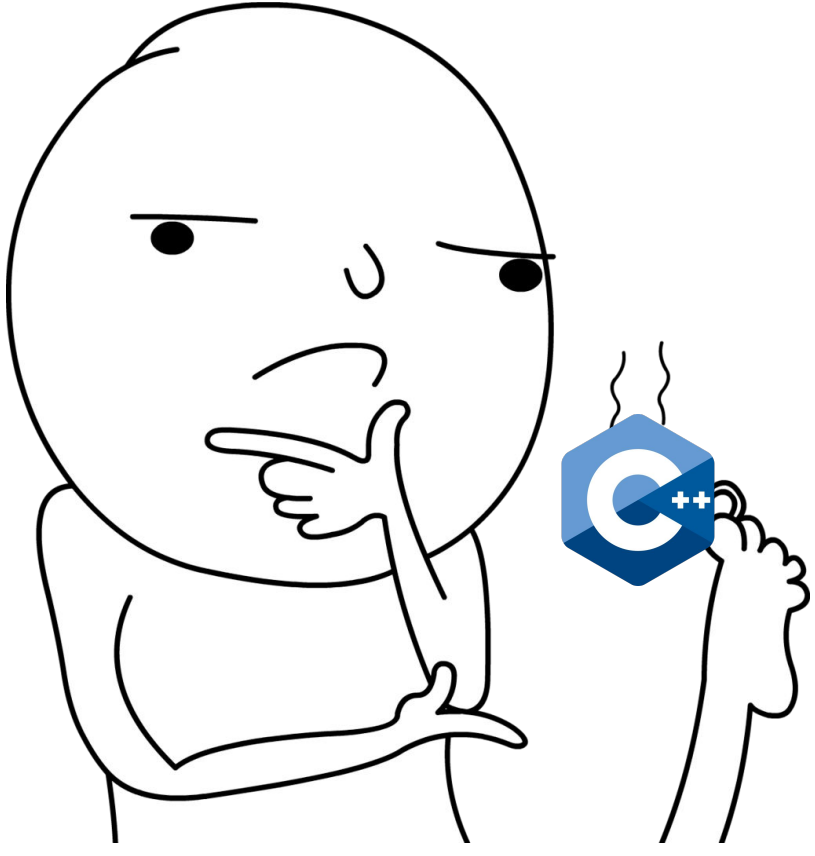


The linker takes each piece of object code and arranges it into one program

- Think about this stage as “stitching” things together
- At this point we get an executable program



What questions do we have?



`Makefiles` and `make`

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

`Makefile` and `make`

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

- It makes use of the g++ compiler

`Makefile` and `make`

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

- It makes use of the g++ compiler
- In order to use **`make`** you need to have a **`Makefile`**

`Makefile` and `make`

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

- It makes use of the `g++` compiler
- In order to use **`make`** you need to have a **`Makefile`**

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

`Makefile` and `make`

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

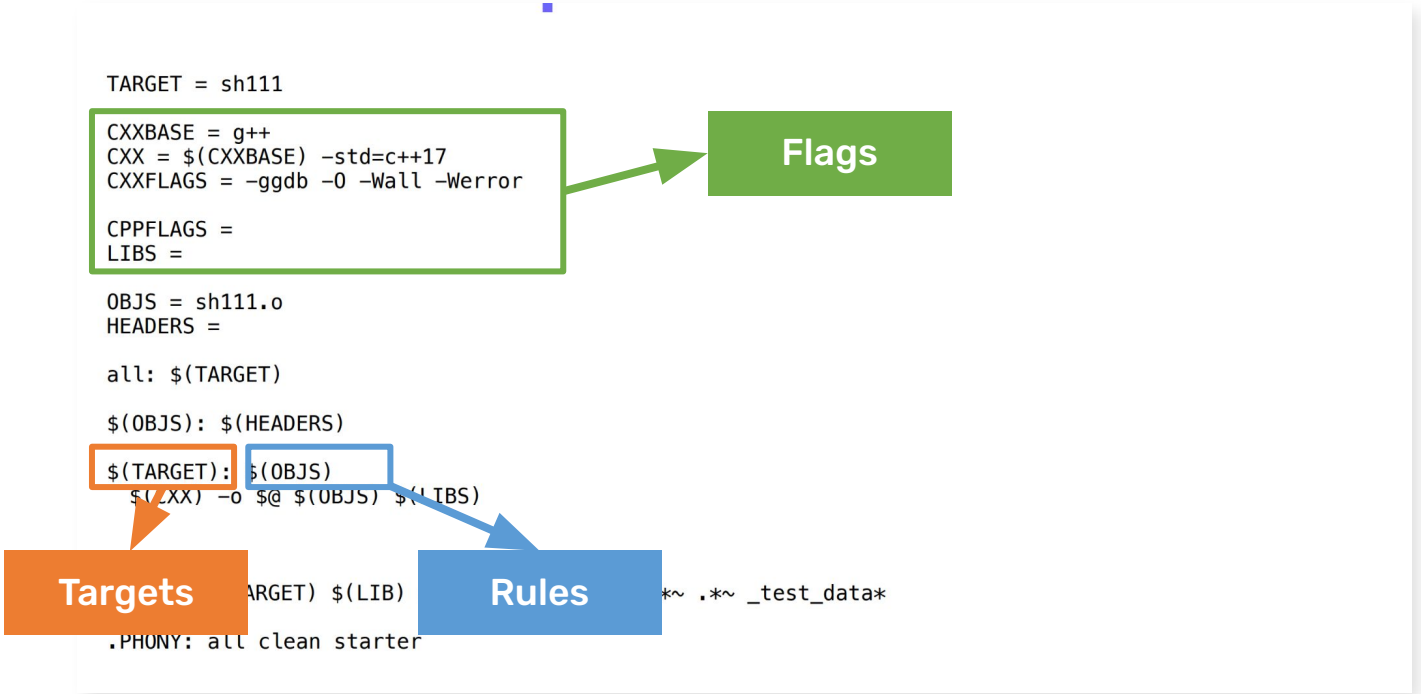
- It makes use of the `g++` compiler
- In order to use **`make`** you need to have a **`Makefile`**

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

Makefile and make

```
TARGET = sh111

CXXBASE = g++
CXX = $(CXXBASE) -std=c++17
CXXFLAGS = -ggdb -O -Wall -Werror

CPPFLAGS =
LIBS =

OBJS = sh111.o
HEADERS =

all: $(TARGET)

$(OBJS): $(HEADERS)

$(TARGET): $(OBJS)
  $(CXX) -o $@ $(OBJS) $(LIBS)


clean:
  rm -f $(TARGET) $(LIB) $(OBJS) $(LIBOBJS) *~ .*~ _test_data*

.PHONY: all clean starter
```

Makefile and make



CMake

CMake is a build system generator.

CMake

CMake is a build system generator.

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

CMake

CMake is a build system generator.

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

Also can be thought about as a cross-platform **make**

CMake

```
cmake_minimum_required(VERSION 3.0)
project(wikiracer)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

find_package(cpr CONFIG REQUIRED)

# adding all files
add_executable(main main.cpp wikiscraper.cpp.o error.cpp)

target_link_libraries(main PRIVATE cpr)
```

Here's an example of of a **cmake** file

CMake

```
cmake_minimum_required(VERSION 3.0)
project(wikiracer)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

find_package(cpr CONFIG REQUIRED)

# adding all files
add_executable(main main.cpp wikiscraper.cpp.o error.cpp)

target_link_libraries(main PRIVATE cpr)
```

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

In summary

- Exceptions in your code during runtime can cause your resources to ‘leak’
- 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
- To build our own projects we can and should use `Makefiles` or another build system.

Last lecture