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 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)
}
```

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?

```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?

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?

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: 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();
}
```

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 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, mute

  // 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

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 🥳

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

Visualizing smart pointers

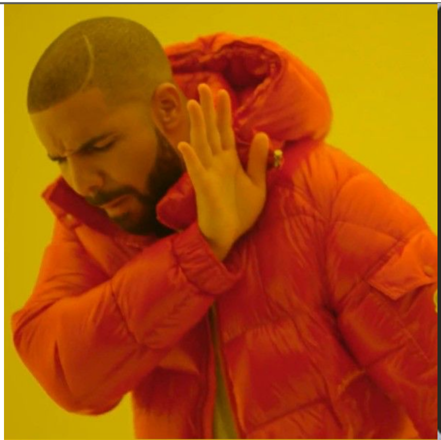
RAII for memory → We can do the same 🥳

- 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
- **`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
}
```

Remember we can’t copy unique pointers

```cpp
void rawPtrFn() {
  std::unique_ptr<Node> n(new Node);

  // this is a compile-time error!
  std::unique_ptr<Node> copy = n;
}
```

Why?

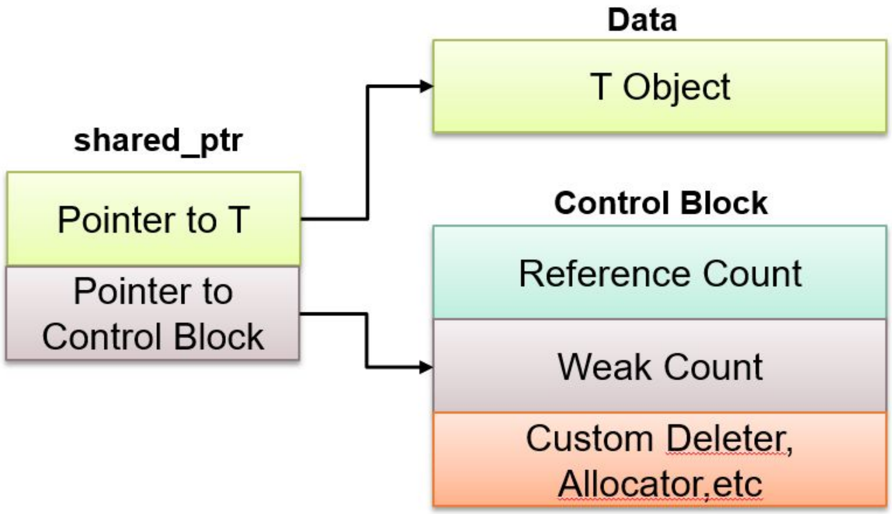
```
void rawPtrFn() {
  std::unique_ptr<Node> n(new Node);

  // 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.

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!



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`**

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.

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

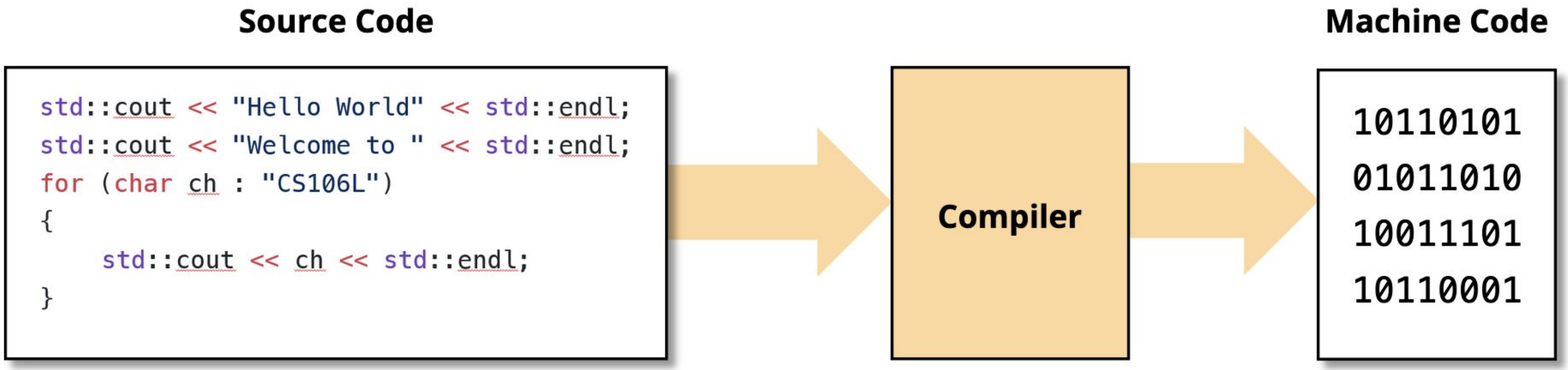
```cpp
#include <iostream>
#include <memory>

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

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`

`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 = ../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

# 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)

to enable tance
CXXFLAGS HERITANCE
$(TARGET)

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

Flags

Targets

Rules

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

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_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!