RAII, Smart Pointers, and C++ Project Building

CS 106L, Fall ‘21

Let's explore a few interesting C++ concepts and techniques!

Today’s agenda

- Exceptions - Why care?
- RAII
- Smart Pointers
- Building C++ Projects
- Anything you'd like to discuss?

How many code paths are in this function?

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
    cout << p.first() << " "
         << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

How many code paths are in this function?

Code Path 1 - favors neither chocolate nor milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
           << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

How many code paths are in this function?

Code Path 1 - favors neither chocolate nor milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
           << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

How many code paths are in this function?

Code Path 1 - favors neither chocolate nor milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
       p.favorite_drink() == "milkshake") {
     cout << p.first() << " "
          << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

How many code paths are in this function?

Code Path 1 - favors neither chocolate nor milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
    cout << p.first() << " "
         << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

How many code paths are in this function?

Code Path 2 - favors milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
    cout << p.first() << " "
         << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

How many code paths are in this function?

Code Path 2 - favors milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
           << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

How many code paths are in this function?

Code Path 2 - favors milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
     cout << p.first() << " "
          << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

How many code paths are in this function?

Code Path 2 - favors milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
    cout << p.first() << " "
         << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

How many code paths are in this function?

Code Path 2 - favors milkshakes

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
    cout << p.first() << " "
         << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

How many code paths are in this function?

Code Path 3 - favors chocolate (and possibly milkshakes)

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
    cout << p.first() << " "
         << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

How many code paths are in this function?

Code Path 3 - favors chocolate (and possibly milkshakes)

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
           << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

How many code paths are in this function?

Code Path 3 - favors chocolate (and possibly milkshakes)

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
    cout << p.first() << " "
         << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

How many code paths are in this function?

Code Path 3 - favors chocolate (and possibly milkshakes)

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
      p.favorite_drink() == "milkshake") {
    cout << p.first() << " "
         << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

How many code paths are in this function?

Are there any more code paths?

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
           << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Hint: Exceptions

- Exceptions are ways to signal that something has gone wrong during run-time
- Exceptions are "thrown" and can crash the program, but can be "caught" to avoid this

```
try {
  // code that may throw exceptions
} catch ([exception type] e1) {â€šÀè  // exception handler
} catch ([anotherExceptionType] e2) {
  // for the case that e1 was not the exception thrown
} catch {
  // a catch-all (haha)
}
```

How many code paths are in this function?

How many do you think there are now?

```
string get_name_and_print_sweet_tooth(Person p) {
  if (p.favorite_food() == "chocolate" ||
        p.favorite_drink() == "milkshake") {
      cout << p.first() << " "
           << p.last() << " has a sweet tooth!" << endl;
  }
  return p.first() + " " + p.last();
}
```

Hidden Code Paths

There are (at least) 23 code paths in the code before!

- (1) copy constructor of Person parameter may throw
- (5) constructor of temp string may throw
- (6) call to favorite_food, favorite_drink, first (2), last (2), may throw
- (10) operators may be user-overloaded, thus may throw
- (1) copy constructor of string for return value may throw

Takeaway: there are often more code paths than meets the eye!

Takeaway: there are often more code paths than meets the eye!

When writing production code, be sure to have test cases that cover all possible paths (or catch errors that would produce more paths)!

What could go wrong here?

Don't just think about exceptions—try to think of a defect that may not cause an immediate error.

```
string get_name_and_print_sweet_tooth(int id_number) {
  Person* p = new Person(id_number); // assume the constructor fills in variables
  if (p->favorite_food() == "chocolate" ||
      p->favorite_drink() == "milkshake") {
      cout << p->first() << " "
           << p->last() << " has a sweet tooth!" << endl;
  }

  auto result = p->first() + " " + p->last();
  delete p;

  return result;
}
```

What could go wrong here?

Can you guarantee that this function wouldn't leak memory?

```
string get_name_and_print_sweet_tooth(int id_number) {
  Person* p = new Person(id_number); // assume the constructor fills in variables
  if (p->favorite_food() == "chocolate" ||
      p->favorite_drink() == "milkshake") {
     cout << p->first() << " "
          << p->last() << " has a sweet tooth!" << endl;
  }

  auto result = p->first() + " " + p->last();
  delete p;

  return result;
}
```

This problem isn't just unique to pointers!

Resources that need to be returned after use:

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

How do we guarantee resources get released, even if there are exceptions?

RAII

Resource Acquisition Is Initialization

RAII

"The best example of why I shouldn't be in marketing"
"I didn't have a good day when I named that"
— Bjarne Stroustrup (daddy of C++)

What is R·A·Double I?

- All resources used by a class should be **acquired** in the constructor
- All resources used by a class should be **released** in the destructor

Recap: What is R·A·Double I?

- All resources used by a class should be **acquired** in the constructor
- All resources used by a class should be **released** in the destructor

Why?

- Objects should be usable immediately after creation
- There should never be a "half-valid" state of an object, where it exists in memory but is not accessible to/used by the program
- The destructor is always called (when the object goes out of scope), so the resource is always freed!

You learned this in CS 106B. Is it RAII Compliant?

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

  string line;
  while (getline(input, line)) { // might throw exception
    cout << line << endl;
  }

  input.close();
}
```

No! `ifstream` not acquired in ctor/released in door

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


  string line;
  while (getline(input, line)) { // might throw exception
    cout << line << endl;
  }

}
```

This is also not RAII-compliant

We won't go into concurrent programming here, but know that "mutexes" allow for exactly one piece of code to run at the same time!

```
void cleanDatabase (mutex& databaseLock,
                    map<int, int>& database) {
  databaseLock.lock();

  // other threads will not modify database
  // modify the database
  // if exception thrown, mutex never unlocked!

  databaseLock.unlock();
}
```

This fixes it!

The lock_guard is an object whose sole job is to release the mutex when it goes out of scope.

```
void cleanDatabase (mutex& databaseLock,
                    map<int, int>& database) {
  lock_guard<mutex> lg(databaseLock);

  // other threads will not modify database
  // modify the database
  // if exception thrown, mutex is unlocked!

  // no need to unlock at end, as it's handle by the lock_guard
}
```

How would we implement `lock_guard`?

How would we implement `lock_guard`?

Here's a very simple, non-template implementation:

```
class lock_guard {
  public:
    lock_guard(mutex& lock) : acquired_lock(lock){
      acquired_lock.lock();
    }
    ~lock_guard() {
      acquired_lock.unlock();
    }
  private:
    mutex& acquired_lock;
}
```

What about RAII for memory?

This is where we're going with RAII!

From the C++ Core Guidelines:

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.

Smart Pointers

RAII for memory!

We just saw how mutexes can be made RAII-safe...

```
void cleanDatabase (mutex& databaseLock,
                    map<int, int>& database) {
  databaseLock.lock();

  // other threads will not modify database
  // modify the database
  // if exception thrown, mutex never unlocked!

  databaseLock.unlock();
}
```

where the fix was to wrap it in an object with a destructor...

```
void cleanDatabase (mutex& databaseLock,
                    map<int, int>& database) {
  lock_guard<mutex> lg(databaseLock);

  // other threads will not modify database
  // modify the database
  // if exception thrown, mutex is unlocked!

  // no need to unlock at end, as it's handle by the lock_guard
}
```

...so let's do that again!

We saw how this was not RAII-compliant because of the "naked" delete.

```
string get_name_and_print_sweet_tooth(int id_number) {
  Person* p = new Person(id_number); // assume the constructor fills in variables
  if (p->favorite_food() == "chocolate" ||
      p->favorite_drink() == "milkshake") {
      cout << p->first() << " "
           << p->last() << " has a sweet tooth!" << endl;
  }

  auto result = p->first() + " " + p->last();
  delete p;

  return result;
}
```

Solution: built-in "smart" (RAII-safe) pointers

- Three types of smart pointers in C++ that automatically free underlying memory when destructed

Solution: built-in "smart" (RAII-safe) pointers

- Three types of smart pointers in C++ that automatically free underlying memory when destructed
  - **`std::unique_ptr`**
    - Uniquely owns its resource, can't be copied
  - **`std::shared_ptr`**
    - Can make copies, destructed when underlying memory goes out of scope
  - **`std::weak_ptr`**
    - models temporary ownership: when an object only needs to be accessed if it exists (convert to shared_ptr to access)

Solution: built-in "smart" (RAII-safe) pointers

- Three types of smart pointers in C++ that automatically free underlying memory when destructed
  - **`std::unique_ptr`**
    - Uniquely owns its resource, can't be copied
  - **`std::shared_ptr`**
    - Can make copies, destructed when underlying memory goes out of scope
  - **`std::weak_ptr`**
    - models temporary ownership: when an object only needs to be accessed if it exists (convert to shared_ptr to access)

```
::get() -> returns a normal pointer to the object
```

std::unique_ptr

Before

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

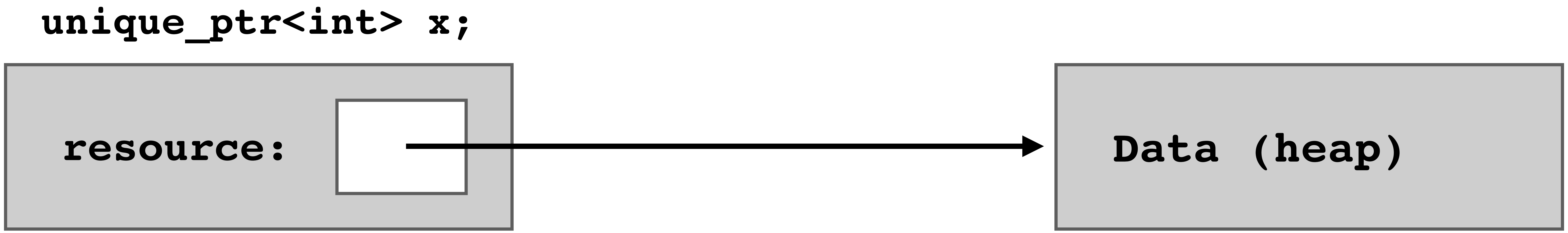
After!

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

What if we could make copies of std::unique_ptr?

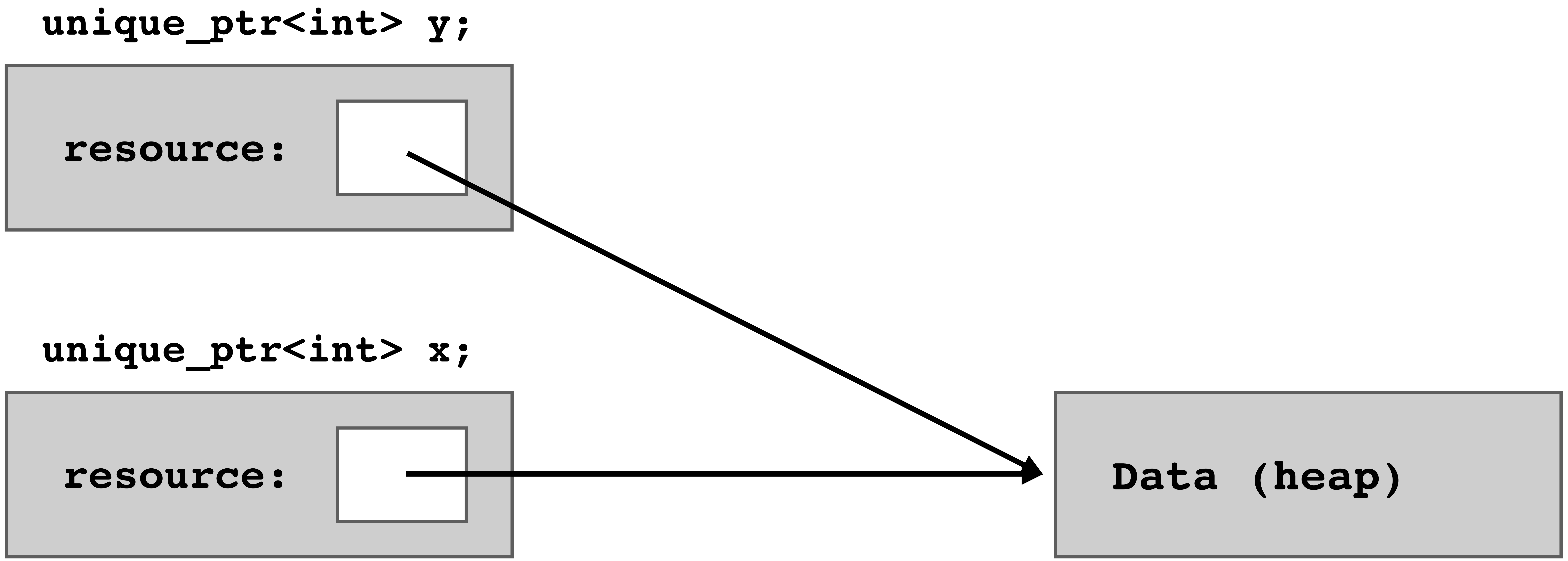
What if we could make copies of std::unique_ptr?

First we make a unique ptr:



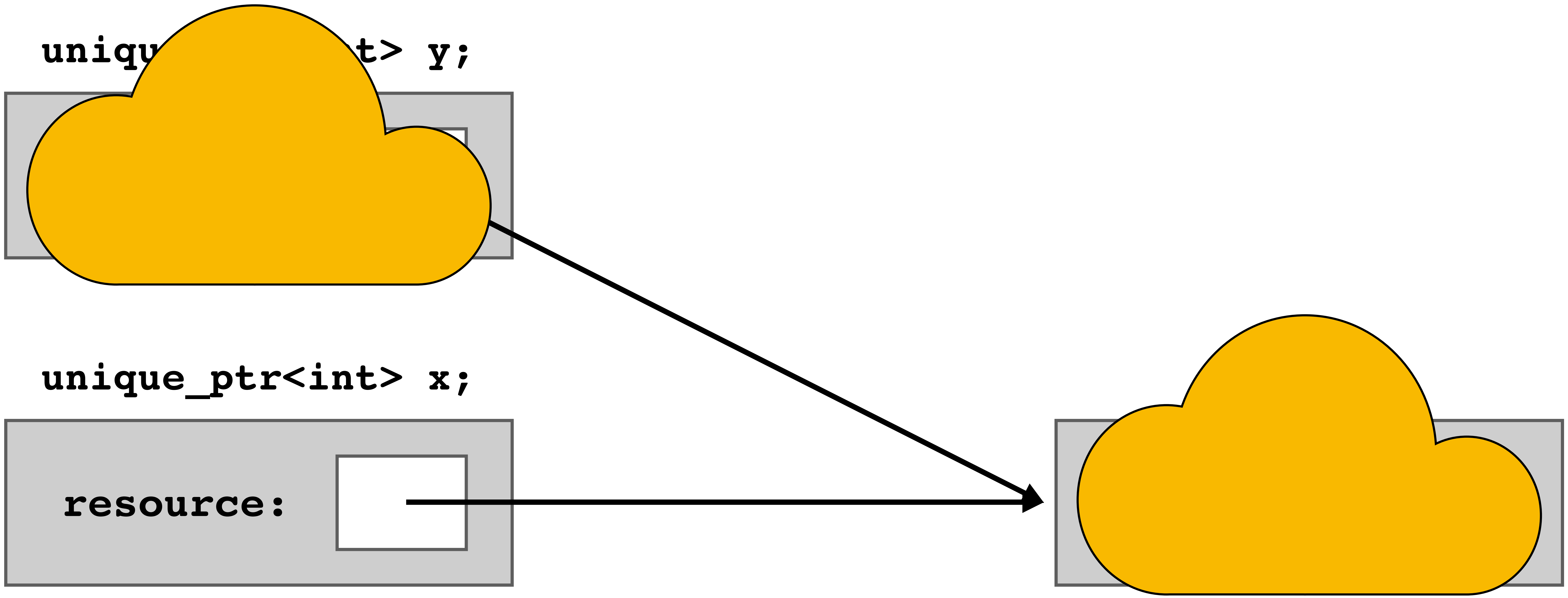
What if we could make copies of std::unique_ptr?

We'd then make a copy of this pointer, pointing to the same resource



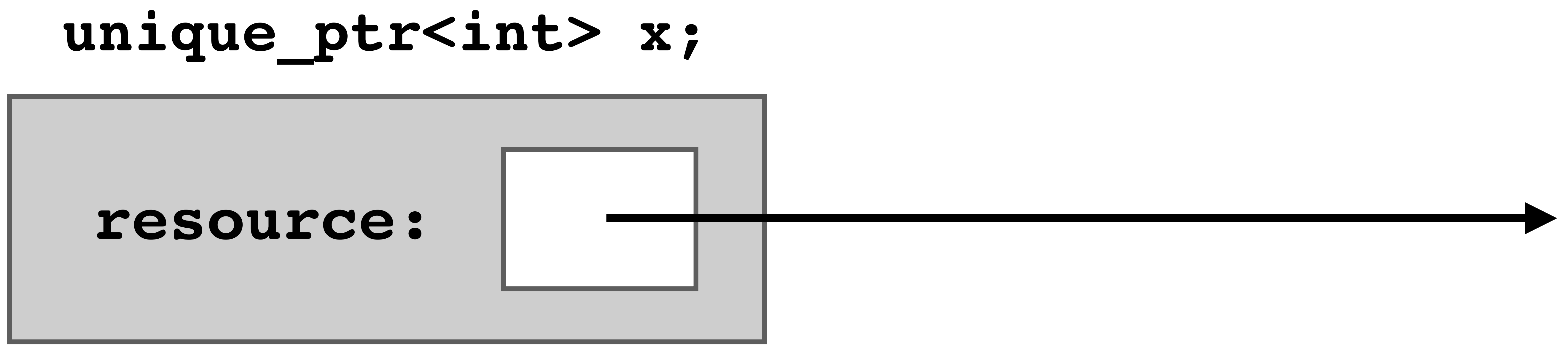
What if we could make copies of std::unique_ptr?

When y goes out of scope, it deletes the heap data



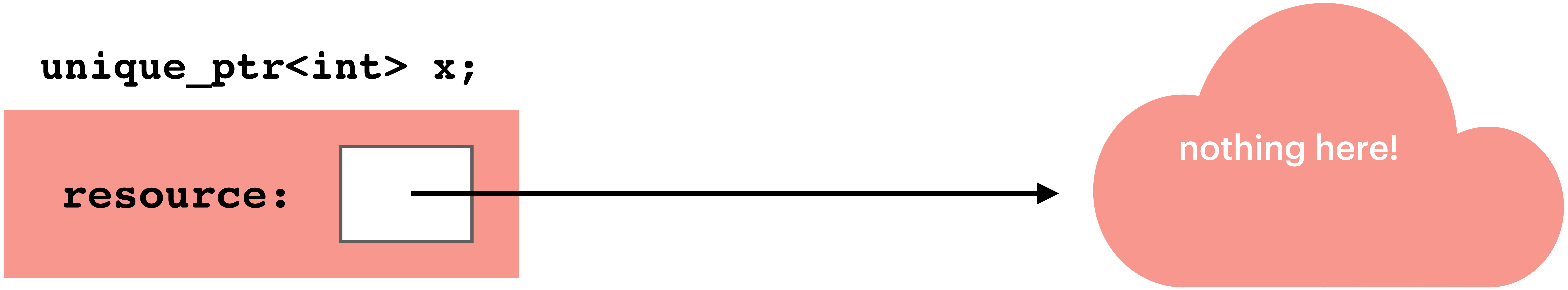
What if we could make copies of std::unique_ptr?

This leaves a hanging pointer x, which points at deallocated data



What if we could make copies of std::unique_ptr?

If we try to access x's data or delete runs the destructor, we crash!



But what if we wanted to have multiple pointers to the same object?

`std::shared_ptr!`

- Resources can be stored by any number of shared_ptrs
- The resource is **`deleted`** when none of the pointers points to the resource!

std::shared_ptr!

- Resources can be stored by any number of shared_ptrs
- The resource is **deleted** when none of the pointers points to the resource!

```
{
  std::shared_ptr<int> p1(new int);
  // use p1
  {
    std::shared_ptr<int> p2 = p1;
    // use p1 and p2
  }
  // use p1, like so
  cout << *p1.get() << endl;
}
// the integer is now deallocated!
```

Smart pointers: RAII Wrapper for pointers

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


std::shared_ptr<T> sp{new T};


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

Smart pointers: RAII Wrapper for pointers

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

std::shared_ptr<T> sp{new T};

std::weak_ptr<T> wp = sp;

// aren't we using "new" explicitly, though?
```

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

There's another option!

```
std::unique_ptr<T> up{new T};
std::unique_ptr<T> up = std::make_unique<T>();

std::shared_ptr<T> sp{new T};
std::shared_ptr<T> sp = std::make_shared<T>();

std::weak_ptr<T> wp = sp;
// can only be copy/move constructed (or empty)!
```

So which way is better?

```
std::unique_ptr<T> up{new T};
std::unique_ptr<T> up = std::make_unique<T>();

std::shared_ptr<T> sp{new T};
std::shared_ptr<T> sp = std::make_shared<T>();
```

Answer:

Always use std::make_unique<T>()!

So which way is better?

```
std::unique_ptr<T> up{new T};
std::unique_ptr<T> up = std::make_unique<T>();

std::shared_ptr<T> sp{new T};
std::shared_ptr<T> sp = std::make_shared<T>();
```

- If we don't use make_shared, then we're allocating memory twice (once for sp, and once for new T)!
- We should be consistent across smart pointers

How are projects built in C++?

What happens when you run our "./build_and_run.sh"?

What do `make` and `Makefiles` do?

- make is a "build system"
- uses g++ as its main engine
- several stages to the compiler system
- can be utilized through a Makefile!
- let's take a look at a simple makefile to get some practice!

Example Makefile—CS111

```
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
```

So why do we use `cmake` in our assignments?

- cmake is a **c**ross-platform **make**
- cmake creates build systems!
- It takes in an even higher-level config file, ties in external libraries, and outputs a Makefile, which is then run.
- Let's take a look at our makefiles!

Example cmake file (CMakeLists.txt)

```
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)
```

Components of C++'s compilation system

How do we go from ASCII code to runnable executables?

Preprocessing	→	**Compiling**	→	**Assembling**	→	**Linking**
in: cpp files		in: full cpp files		in: .s files		in: .o files
out: full cpp files		out: .s files		out: .o files		out: executable

Preprocessing (g++ `-E`)

- The C/C++ preprocessor handles *preprocessor directives*: replaces includes (**`#include`** ...) and and expands any macros (**`#define`** ...)
  - Replace **`#include`**s with content of respective files (which is usually just function/variable declarations, so low bloat)
  - Replaces macros (**`#define`**) and selecting different portions of text depending on **`#if, #ifdef, #ifndef`**
- Outputs a stream of tokens resulting from these transformations
- If you want, you can produce some errors at even this stage (**`#if`**, `#error`)

Preprocessing → Compiling → Assembling → Linking

Compilation (g++ -S)

- Performed on output of the preprocessor (full C++ code)
- Structure of a compiler:
  - Lexical Analysis
  - Parsing
  - Semantic Analysis
  - Optimization
  - Code Generation (assembly code)
- This is where traditional "compiler errors" are caught

Preprocessing → **Compiling** → Assembling → Linking

Assembling (g++ -c)

- Runs on the assembly code as outputted by the compiler
  - Take 107 to see more!
- Converts assembly code to binary machine code
- Assumes that all functions are defined somewhere without checking
- Final output: object files
  - Can't be run by themselves!

Preprocessing → **Compiling** → **Assembling** → **Linking**

Linking (`ld`, g++)

- Creates a single executable file from multiple object files
  - Combine the pieces of a program
  - Figure out a memory organization so that all the pieces can fit together
  - Resolve references so that the program can run under the new memory organization
    - .h files declare functions, but the actual functions may be in separate files from where they're called!
- Output is fully self-sufficient—no other files needed to run

Preprocessing → Compiling → Assembling → **Linking**