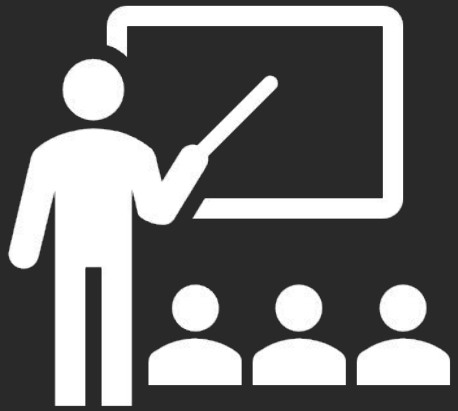
RAII and Smart Pointers

Game Plan



- RAII
- Examples
- Smart Pointers

Without any extra information, how many potential code paths are in this function?

how control flow goes

```
string EvaluateSalaryAndReturnName( Employee e ) {

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();

}
```

Code path 1

```
string EvaluateSalaryAndReturnName( Employee e ) {

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();

}
```

Code path 1

```
string EvaluateSalaryAndReturnName( Employee e ) {

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();

}
```

Code path 1

```
string EvaluateSalaryAndReturnName( Employee e ) {

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();

}
```

Code path 1

```
string EvaluateSalaryAndReturnName( Employee e ) {

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();

}
```

Code path 2

```
string EvaluateSalaryAndReturnName( Employee e ) {

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();

}
```

Code path 2

```
string EvaluateSalaryAndReturnName( Employee e ) {

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();

}
```

Code path 2

```
string EvaluateSalaryAndReturnName( Employee e ) {

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();

}
```

Code path 2

```
string EvaluateSalaryAndReturnName( Employee e ) {

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();

}
```

Code path 2

```
string EvaluateSalaryAndReturnName( Employee e ) {

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();

}
```

Code path 3 (short-circuiting)

```
string EvaluateSalaryAndReturnName( Employee e ) {

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();

}
```

Code path 3 (short-circuiting)

```
string EvaluateSalaryAndReturnName( Employee e ) {

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();

}
```

Code path 3 (short-circuiting)

```
string EvaluateSalaryAndReturnName( Employee e ) {

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();

}
```

Code path 3 (short-circuiting)

```
string EvaluateSalaryAndReturnName( Employee e ) {

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();

}
```

Are there any more code paths?

```
string EvaluateSalaryAndReturnName( Employee e ) {

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  return e.First() + " " + e.Last();

}
```

Hidden Code Paths

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

- 1 – Copy constructor of Employee parameter, may throw.
- 5 – Constructor of temp string/ints, may throw.
- 6 – Call to Title, Salary, First (2), Last (2), may throw.
- 10 – Operators may be user-overloaded, may throw.
- 1 – Copy constructor of string for return value, may throw.

Followup example: what might go wrong?

```
string EvaluateSalaryAndReturnName(int idNumber) {
  Employee* e = new Employee(idNumber);

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  auto result = e.First() + " " + e.Last();

  delete e;
  return result;
}
```

Can you guarantee this function will not have a memory leak?

```
string EvaluateSalaryAndReturnName(int idNumber) {
  Employee* e = new Employee(idNumber);

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  auto result = e.First() + " " + e.Last();

  delete e;
  return result;
}
```

More general concern: resources that need to be released.

Resources that need to be returned.

	Acquire	Release
• Heap memory	new	delete

More general concern:
resources that need to be released.

Resources that need to be returned.

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

Aside: Exceptions

Exceptions are a way to transfer control and information to a (potential) exception handler.

```
try {
    // code associated with exception handler
} catch ( [exception type] e ) {
    // exception handler
} catch ( [exception type] e )  {
    // exception handler
} // etc.
```

Aside: Exceptions

Exceptions are a way to transfer control and information to a (potential) exception handler.

We won't cover in depth how to use exceptions (you've used them, kinda, in 106B, and they're easy to figure out).

However, the idea of exception safety is extremely important in object-oriented programming.

We can’t guarantee the ‘delete’ call is called if an exception is thrown.

```cpp
string EvaluateSalaryAndReturnName(int idNumber) {
  Employee* e = new Employee(idNumber);

  if ( e.Title() == "CEO" || e.Salary() > 100000 ) {
    cout << e.First() << " "
         << e.Last() << " is overpaid" << endl;
  }
  auto result = e.First() + " " + e.Last();

  delete result; // what if we skip this line?
  return result;
}
```

throw errors else where

可能执行不到delete那一行中间就有某行代码因为种种原因中断了，结果delete没有执行造成了内存泄漏

How do we guarantee classes release their resources?

Regardless of exceptions!

Aside: Enforcing exception safety

Functions can have four levels of exception safety:

- Nothrow exception guarantee noexcept
  - absolutely does not throw exceptions: destructors, swaps, move constructors, etc.
- Strong exception guarantee
  - rolled back to the state before function call
- Basic exception guarantee
  - program is in valid state after exception
- No exception guarantee
  - resource leaks, memory corruption, bad…

Aside: avoiding exceptions entirely

Exceptions

We do not use C++ exceptions.

Source: https://google.github.io/styleguide/cppguide.html#Exceptions

Aside: avoiding exceptions entirely

Decision:

On their face, the benefits of using exceptions outweigh the costs, especially in new projects. However, for existing code, the introduction of exceptions has implications on all dependent code. If exceptions can be propagated beyond a new project, it also becomes problematic to integrate the new project into existing exception-free code. Because most existing C++ code at Google is not prepared to deal with exceptions, it is comparatively difficult to adopt new code that generates exceptions.

Given that Google's existing code is not exception-tolerant, the costs of using exceptions are somewhat greater than the costs in a new project. The conversion process would be slow and error-prone. We don't believe that the available alternatives to exceptions, such as error codes and assertions, introduce a significant burden.

Our advice against using exceptions is not predicated on philosophical or moral grounds, but practical ones. Because we'd like to use our open-source projects at Google and it's difficult to do so if those projects use exceptions, we need to advise against exceptions in Google open-source projects as well. Things would probably be different if we had to do it all over again from scratch.

Source: https://google.github.io/styleguide/cppguide.html#Exceptions

tl;dr We forgot to do it initially, so let’s not bother getting started.

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

SBRM

Scope Based Memory Management

CADRE

Constructor Acquires, Destructor Releases

对一个对象，所有资源都在constructor里面准备好；所有资源的销毁都在destructor里面release
不要一半一半。

PIMPL

Pointer to Implementation.

(this is not another name for RAII. just wanted to bring it up since we are talking about bad C++acronyms).

What is RAII?

All resources should be acquired in the constructor.

All resources should be released in the destructor.

What is the rationale?

There should never be a ”half-valid” state of the object. Object <u>useable after its creation.</u>

The destructor is always called (even with exceptions), so the resource is always freed.

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

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

  string line;
  while (getline(input, line)) {
    cout << line << endl;
  }

  input.close();
}
```

Resource not acquired in the constructor or released in the destructor.

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

  string line;
  while (getline(input, line)) {
    cout << line << endl;
  }

  input.close();
}
```

Resource not acquired in the constructor or released in the destructor.

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

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

  // no close call needed!
} // stream destructor, releases access to file
```

一初始就全部准备好

Resource not acquired in the constructor or released in the destructor.

```cpp
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();
}
```

The fix: an object whose sole job is to release the resource when it goes out of scope.

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

  lock_guard<mutex>(databaseLock);

  // other threads will not modify database
  // modify the database
  // if exception thrown, that's fine!

  // no release call needed
} // lock always unlocked when function exits.
```

How do you think lock_guard is implemented?

Here’s a non-template version.

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

Sidenote: mutexes cannot be copied, and cannot be moved!

```
class mutex {
public:
  mutex(const mutex& other) = delete;
  mutex& operator=(const mutex& rhs) = delete;
  mutex(mutex&& other) = delete;
  mutex& operator=(mutex&& other) = delete;
}
```

That’s why we initialize lock_guard using the initializer_list.

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

Summary for RAII

Acquire resources in the constructor, release in the destructor.

Clients of your class won't have to worry about mismanaged resources.

RAII for Memory!

This is what we are approaching (kinda)!

C++ Will No Longer Have Pointers

Published April 1, 2018 - 11 Comments

(note: this is an April Fools joke by a C++ blog, even if it has some truth, please do not take this as a truth!). Do not blindly believe this.

More accurately…

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.

Is automatic memory management a good or bad thing?

Smart Pointers

Up till now we’ve seen how file reading and locks can be non-RAII compliant...

Up till now we've seen how file reading and locks can be non-RAII compliant…

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

  string line;
  while (getline(input, line)) {
    cout << line << endl;
  }

  input.close();
}
```

Up till now we've seen how file reading and locks can be non-RAII compliant…

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

  string line;
  while (getline(input, line)) {
    cout << line << endl;
  }

  input.close();
}
```

```
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 to ensure that the resource is released…

...where the fix was to wrap it in an object to ensure that the resource is released...

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

  // read file

  // no close call needed!
}
// stream destructor
// releases access to file
```

...where the fix was to wrap it in an object to ensure that the resource is released...

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

  // read file

  // no close call needed!
}
// stream destructor
// releases access to file
```

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

  lock_guard<mutex>(databaseLock);

  // other threads will not modify database
  // modify the database
  // if exception thrown, that's fine!

  // no release call needed
} // lock always unlocked when function exi
```

…so let’s do it again!

...so let’s do it again!

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

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

…so let’s do it again!

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

```
void rawPtrFn () {

  Node* n = new Node;

  // do some stuff with n…
  // if exception thrown, n never deleted!
  delete n;

}
```

C++ has built-in “smart” (i.e. RAII-compliant) pointers:

```
std::unique_ptr
std::shared_ptr
std::weak_ptr
```

C++ has built-in “smart” (i.e. RAII-compliant) pointers:

```
std::unique_ptr
std::shared_ptr
std::weak_ptr
boost::scoped_ptr
boost::intrusive_ptr
...
```

C++ has built-in “smart” (i.e. RAII-compliant) pointers:

```
std::unique_ptr
std::shared_ptr
std::weak_ptr
boost::scoped_ptr
boost::intrusive_ptr
...
```

Aside: `std::auto_ptr` is deprecated!

`std::unique_ptr`

Uniquely owns its resource and deletes it when the object is destroyed.

Cannot be copied!

防twice release

std::unique_ptr

Uniquely owns its resource and deletes it when the object is destroyed.

Cannot be copied!

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

std::unique_ptr

Uniquely owns its resource and deletes it when the object is destroyed.

Cannot be copied!

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

```
void rawPtrFn () {
  std::unique_ptr<Node> n(new Node);
  // do some stuff with n

} // Freed!
```

std::unique_ptr

Uniquely owns its resource and deletes it when the object is destroyed.

Cannot be copied!

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

```
void rawPtrFn () {
  std::unique_ptr<Node> n(new Node);
  // do some stuff with n

} // Freed!
```

std::unique_ptr

Uniquely owns its resource and deletes it when the object is destroyed.

Cannot be copied!

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

```
void rawPtrFn () {
  std::unique_ptr<Node> n(new Node);
  // do some stuff with n

} // Freed!
```

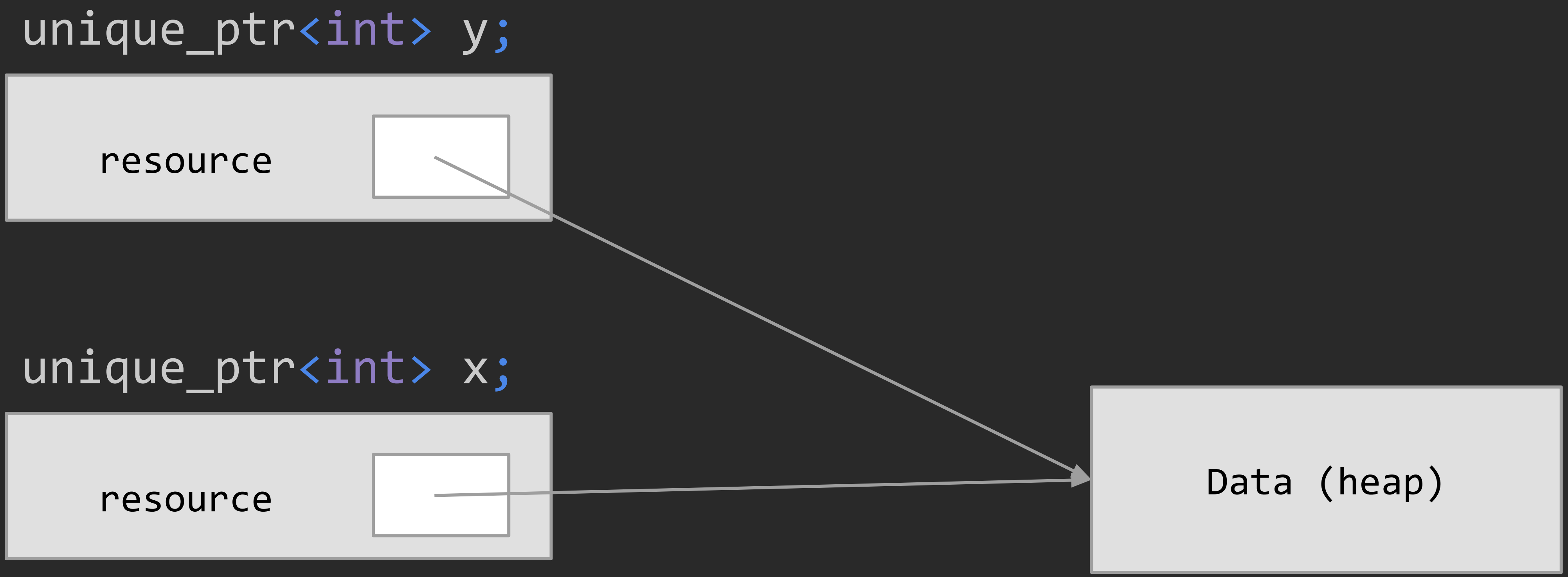
What happens if we try to copy a unique_ptr?

Let’s try it!

First we make a `unique_ptr`

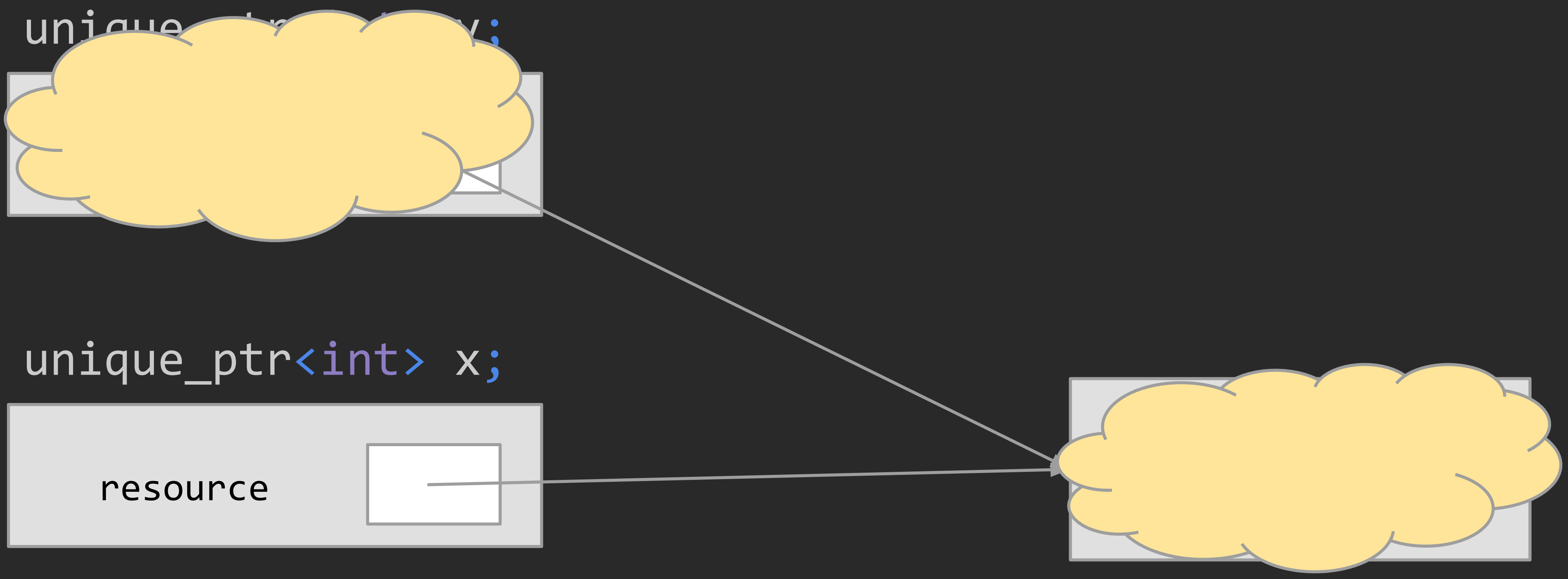
Let’s try it!

We then make a copy of our `unique_ptr`



Let’s try it!

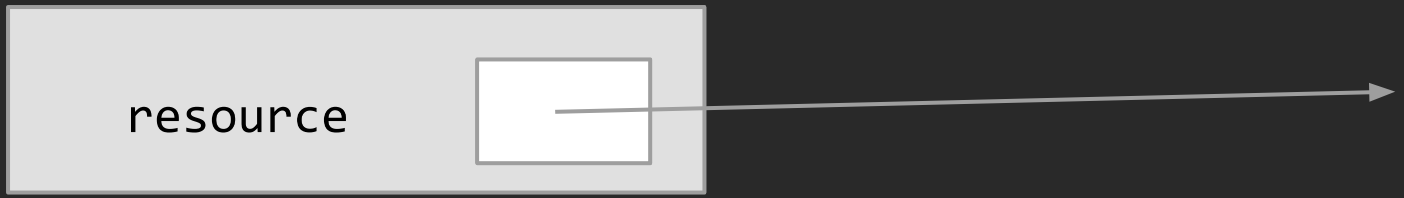
When y goes out of scope, it deletes the heap data



Let’s try it!

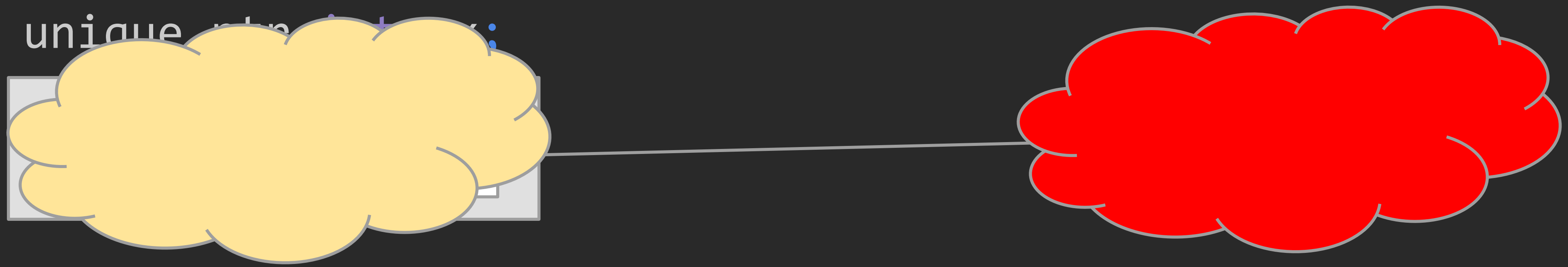
This leaves x pointing at deallocated data

```
unique_ptr<int> x;
```



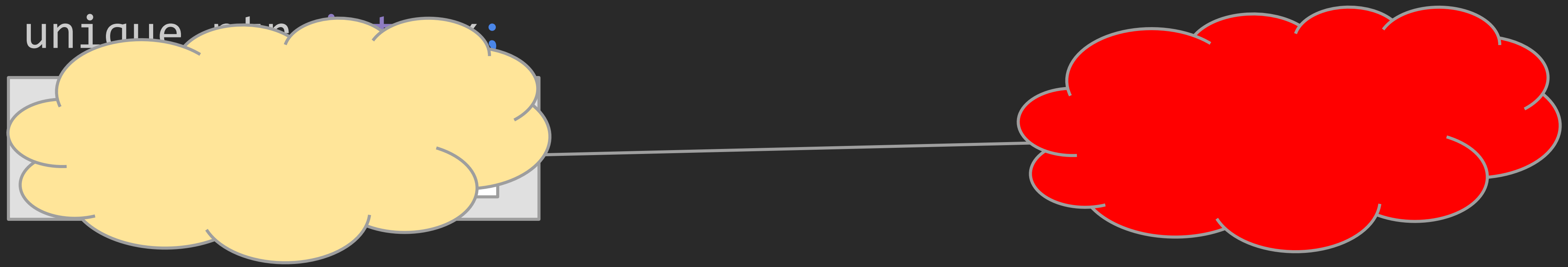
Let’s try it!

If we dereference x or its destructor calls delete, we crash



Let’s try it!

If we dereference x or its destructor calls delete, we crash



`std::unique_ptr`

The unique_ptr class hence disallows copying.

Sanity check: How can you tell a class to disallow copying?

`std::unique_ptr`

The unique_ptr class hence disallows copying.

Sanity check: How can you tell a class to disallow copying?
→ By deleting the copy constructor and copy assignment!

`std::unique_ptr`

The unique_ptr class hence disallows copying.

Yet, we often want to have multiple pointers to the same object.

`std::unique_ptr`

The unique_ptr class hence disallows copying.

Yet, we often want to have multiple pointers to the same object.

→ `std::shared_ptr!`

`std::shared_ptr`

Resource can be stored by any number of `shared_ptr`s.

Deleted when none of them point to it.

std::shared_ptr

Resource can be stored by any number of `shared_ptr`s.

Deleted when none of them point to it.

```
{
 std::shared_ptr<int> p1(new int);
 // Use p1
 {
    std::shared_ptr<int> p2 = p1;
    // Use p1 and p2
 }
 // Use p1
}
// Freed!
```

只有所有指针都用完了shared_ptr所指内存才会删掉

std::shared_ptr

Resource can be stored by any number of `shared_ptr`s.

Deleted when none of them point to it.

```
{
 std::shared_ptr<int> p1(new int);
 // Use p1
 {
     std::shared_ptr<int> p2 = p1;
     // Use p1 and p2
 }
 // Use p1
}
// Freed!
```

Important: this only works if new `shared_ptr`s are made though copying!

`std::shared_ptr`

How are these implemented?

std::shared_ptr

How are these implemented?

Reference counting!

`std::shared_ptr`

How are these implemented?

Reference counting!

复制/给个新的+1次；用完out of scope –1次；0的时候彻底删掉

- Idea: Store an int that keeps track of the number currently referencing that data
  - Gets incremented in copy constructor/copy assignment
  - Gets decremented in destructor or when overwritten with copy assignment
- Frees the resource when reference count hits 0

std::shared_ptr

Notice that our previous example still works!

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

```
void rawPtrFn () {
  std::unique_ptr<Node> n(new Node);
  // do some stuff with n

} // Freed!
```

std::shared_ptr

Notice that our previous example still works!

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

```
void rawPtrFn () {
  std::unique_ptr<Node> n(new Node);
  // do some stuff with n

} // Freed!
```

copy; none

std::shared_ptr

Notice that our previous example still works!

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

➡

```
void rawPtrFn () {
  std::shared_ptr<Node> n(new Node);
  // do some stuff with n

} // Freed!
```

std::shared_ptr

Notice that our previous example still works!

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

```
void rawPtrFn () {
  std::shared_ptr<Node> n(new Node);
  // do some stuff with n

} // Freed!
```

Why?

And finally, `std::weak_ptr`

Similar to a `shared_ptr`, but doesn't contribute to the reference count.

Used to deal with circular references of shared_ptr.

See the documentation for how to use!

C++ has built-in smart pointers:

```
std::unique_ptr
```

```
std::shared_ptr
```

```
std::weak_ptr
```

C++ has built-in smart pointer constructors!

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

std::shared_ptr<T>{new T}

std::weak_ptr<T>{new T}
```

C++ has built-in smart pointer constructors!

```
~~std::unique_ptr<T>{new T}~~
          std::make_unique<T>();
std::shared_ptr<T>{new T}

std::weak_ptr<T>{new T}
```

C++ has built-in smart pointer constructors!

```
~~std::unique_ptr<T>{new T}~~
          std::make_unique<T>();
~~std::shared_ptr<T>{new T}~~
          std::make_shared<T>();
std::weak_ptr<T>{new T}
```

C++ has built-in smart pointer constructors!

~~`std::unique_ptr<T>{new T}`~~

```
std::make_unique<T>();
```

~~`std::shared_ptr<T>{new T}`~~

```
std::make_shared<T>();
```

~~`std::weak_ptr<T>{new T}`~~

Trick question! We’ll see why at the beginning of next lecture.

C++ has built-in smart pointer constructors!

```
std::unique_ptr<T>{new T}
        std::make_unique<T>();
std::shared_ptr<T>{new T}
        std::make_shared<T>();
std::weak_ptr<T>{new T}
```

C++ has built-in smart pointer constructors!

```
std::unique_ptr<T>{new T}
          std::make_unique<T>();
std::shared_ptr<T>{new T}
          std::make_shared<T>();
std::weak_ptr<T>{new T}
```

Which is better to use?

C++ has built-in smart pointer constructors!

```
std::unique_ptr<T>{new T}
        std::make_unique<T>();
std::shared_ptr<T>{new T}
        std::make_shared<T>();
std::weak_ptr<T>{new T}
```

Which is better to use?

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

C++ has built-in smart pointer constructors!

```
std::unique_ptr<T>{new T}
        std::make_unique<T>();
std::shared_ptr<T>{new T}
        std::make_shared<T>();
std::weak_ptr<T>{new T}
```

Which is better to use?

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

We'll see why at the beginning of next lecture.

So, coming full circle:

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.

So, coming full circle:

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 t

Note

In a large p
manageme
latent: it m
you probab

In modern C++, we pretty much never use `new` and `delete`!

Enforcement

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

Next time

Multithreading