Lecture 15: RAII and Smart Pointers

CS 106L, Fall ‘20

Agenda

- Exceptions
- RAII
- Smart pointers

How many code paths are in this function?

```
string evaluate_sweet_tooth_and_return_name( 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();
}
```

Code Path 1 - favors neither chocolate nor milkshakes

```
string evaluate_sweet_tooth_and_return_name( 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();
}
```

Code Path 1 - favors neither chocolate nor milkshakes

```
string evaluate_sweet_tooth_and_return_name( 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();
}
```

Code Path 1 - favors neither chocolate nor milkshakes

```
string evaluate_sweet_tooth_and_return_name( 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();
}
```

Code Path 1 - favors neither chocolate nor milkshakes

```
string evaluate_sweet_tooth_and_return_name( 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();
}
```

Code Path 2 - favors milkshakes

```
string evaluate_sweet_tooth_and_return_name( 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();
}
```

Code Path 2 - favors milkshakes

```
string evaluate_sweet_tooth_and_return_name( 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();
}
```

Code Path 2 - favors milkshakes

```
string evaluate_sweet_tooth_and_return_name( 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();
}
```

Code Path 2 - favors milkshakes

```
string evaluate_sweet_tooth_and_return_name( 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();
}
```

Code Path 2 - favors milkshakes

```
string evaluate_sweet_tooth_and_return_name( 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();
}
```

Code Path 3 - favors chocolate (and possibly milkshakes)

```
string evaluate_sweet_tooth_and_return_name( 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();
}
```

Code Path 3 - favors chocolate (and possibly milkshakes)

```
string evaluate_sweet_tooth_and_return_name( 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();
}
```

Code Path 3 - favors chocolate (and possibly milkshakes)

```
string evaluate_sweet_tooth_and_return_name( 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();
}
```

Code Path 3 - favors chocolate (and possibly milkshakes)

```
string evaluate_sweet_tooth_and_return_name( 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();
}
```

Are there any more code paths?

```
string evaluate_sweet_tooth_and_return_name( 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, may throw.
- 1 – Copy constructor of string for return value, may throw.

Brief aside: exceptions

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.
```

What might go wrong here? (Answer in chat)

```
string evaluate_sweet_tooth_and_return_name( int id_number ) {
  Person *p = new Person(id_number);

  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;
}
```

Can we guarantee this function won’t leak memory?

```
string evaluate_sweet_tooth_and_return_name( int id_number ) {
  Person *p = new Person(id_number);

  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;
}
```

The “delete” won’t happen if there’s an exception first!

```
string evaluate_sweet_tooth_and_return_name( int id_number ) {
  Person *p = new Person(id_number);

  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;
}
```

Lots of kinds of resources 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

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

~~🤔~~ Questions? ~~🤔~~

What is RAII?

- All resources should be **acquired** in the constructor.
- All resources should be **released** in the destructor.

What is RAII?

- All resources should be **acquired** in the constructor.
- All resources should be **released** in the destructor.

What's the rationale for this?

- There should never be a "half-valid" state of the object--object is usable immediately after creation.
- The destructor is always called (even with exceptions), 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();
}
```

Nope - resource not acquired in ctor/released in dtor

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

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

  input.close();
}
```

This fixes it!

```
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
```

This is also not RAII-compliant!

```
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 resource (unlock the mutex) when it goes out of scope

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

  lock_guard<mutex> lock(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 might `lock_guard` be 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;
}
```

RAII Summary

- Acquire resources in the constructor, release in the destructor.
- Clients of your class won’t have to worry about mismanaged resources.

But what about RAII for memory?

This is where we're going!

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.

~~🤔~~ Questions? ~~🤔~~

Announcements

Assignment 2 has been released!

- For assignment 2, you'll be extending an existing implementation of a HashMap data structure to make it more similar to the std::unordered_map!
- Checkpoint 1 is due Thursday, November 12 at 11:59 PM
- Checkpoint 2 is due Wednesday, November 18 at 11:59 PM
- Avoid using late days for the first checkpoint, but feel free to use them for the second
- You may also work with a partner if you'd like (you'll only submit one assignment for the both of you)

Smart Pointers
(RAII for memory!)

We just saw how locks could be made RAII-compliant

```
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 a special object

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

   lock_guard<mutex> lock(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.
```

… so let's do it again!

You learned this in CS 106B -- is this RAII-compliant?

```
void rawPtrFn () {
  Node* n = new Node;

  // do some stuff with n…

  delete n;
}
```

You learned this in CS 106B -- is this RAII-compliant?

```
void rawPtrFn () {
  Node* n = new Node;

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

You learned this in CS 106B -- is this RAII-compliant?

```
void rawPtrFn () {
  Node* n = new Node

  // do some stuff w
  // if exception th            ever deleted!
  delete n;
}
```

Solution: built-in “smart” (RAII-compliant) pointers

```
std::unique_ptr
std::shared_ptr
std::weak_ptr
// std::auto_ptr is deprecated
```

std::unique_ptr

- Uniquely owns its resource and deletes it when the object is destroyed
- Cannot be copied (but can be moved, if non-const!)

std::unique_ptr

Before

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

After

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

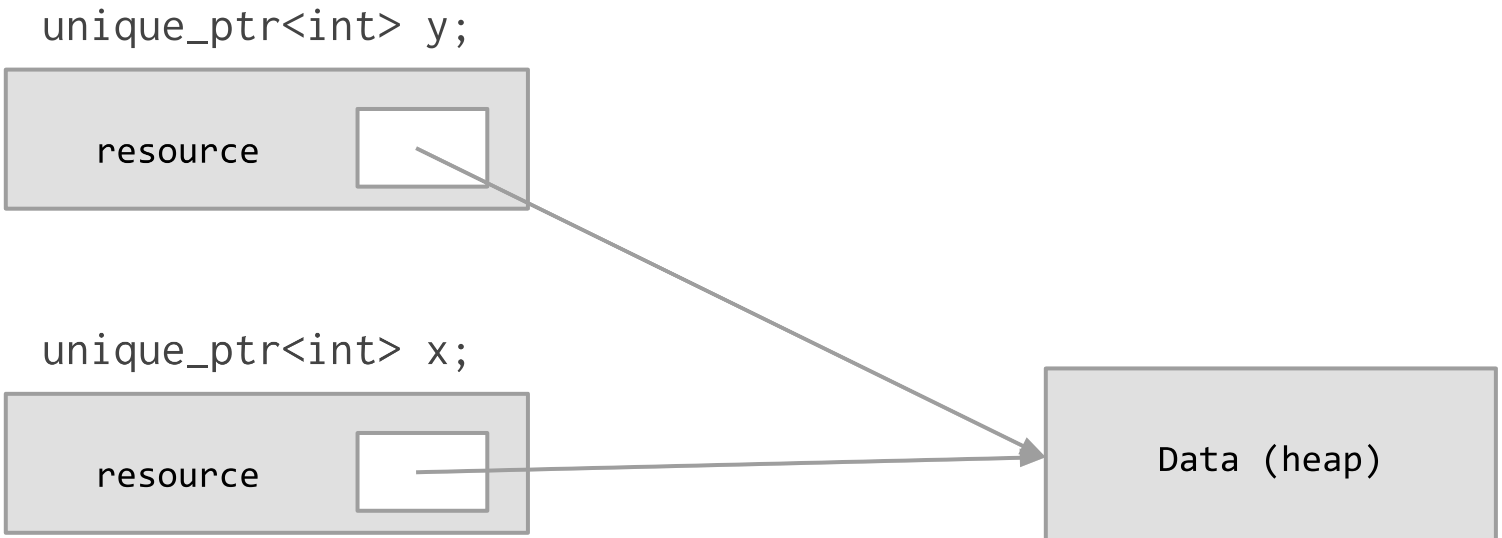
We can’t copy an `std::unique_ptr`, but what if we could?

We can’t copy an `std::unique_ptr`, but what if we could?

First we make a unique_ptr

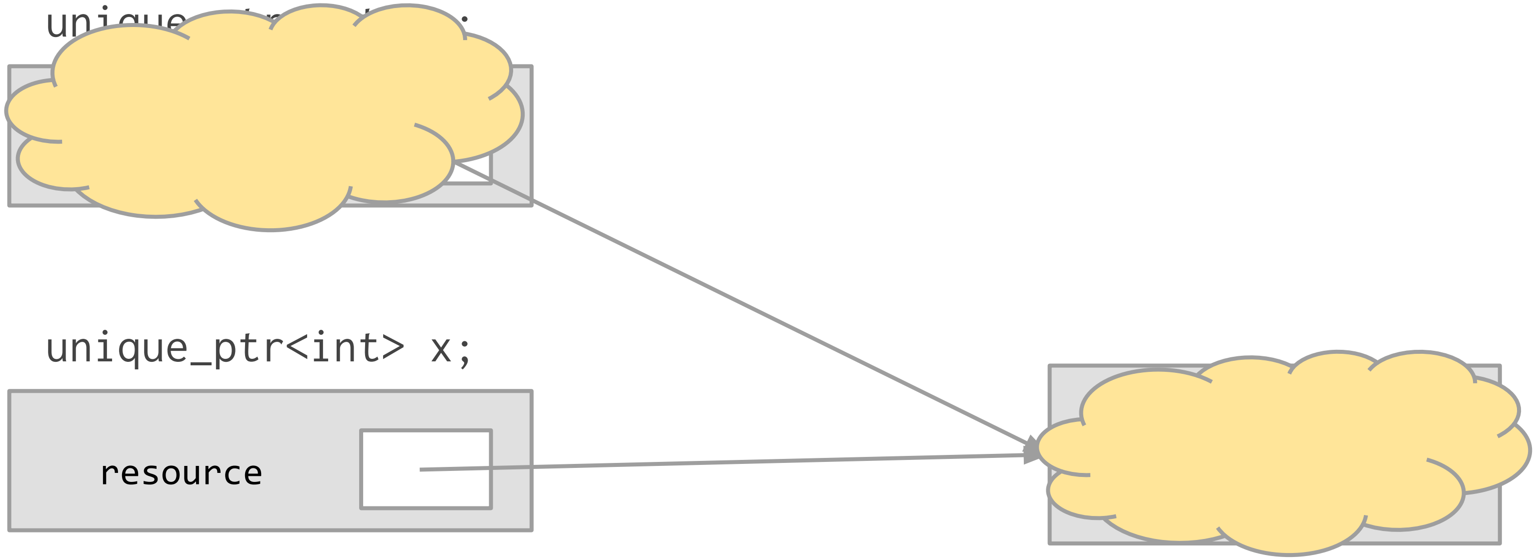
We can't copy an `std::unique_ptr`, but what if we could?

We then make a copy of our unique_ptr



We can't copy an `std::unique_ptr`, but what if we could?

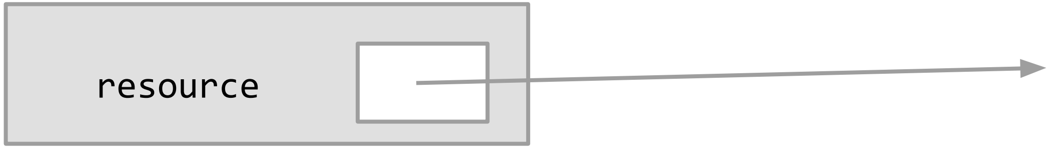
When y goes out of scope, it deletes the heap data



We can’t copy an `std::unique_ptr`, but what if we could?

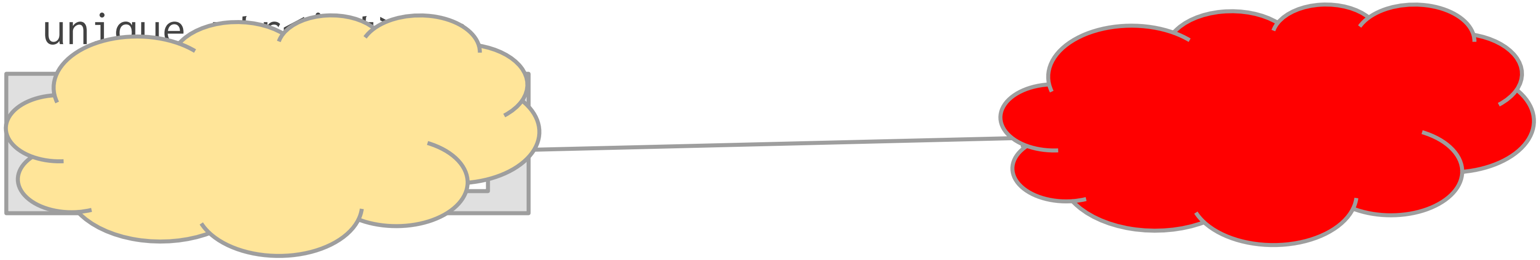
This leaves x pointing at deallocated data

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



We can't copy an `std::unique_ptr`, but what if we could?

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



We can’t copy an `std::unique_ptr`, but what if we could?

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

We can't copy an `std::unique_ptr`, but what if we could?

- The `std::unique_ptr` class hence disallows copying!

🤔 Questions? 🤔

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

`std::shared_ptr`

- Resource can be stored by any number of `std::shared_ptrs`
- The resource is **deleted** when none of them points to it

std::shared_ptr

- Resource can be stored by any number of `std::shared_ptrs`
- The resource is **deleted** when none of them points to it

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

std::shared_ptr

- Resource can be stored by any number of `std::shared_ptrs`
- The resource is **deleted** when none of them points to it

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

Important: This only works if new `shared_ptrs` are made by copying, not by constructing using the raw pointer!

How are `std::shared_ptrs` implemented?

How are these implemented? Reference counting!

- Idea: Store int that tracks how many shared_ptrs currently reference that data
    - Increment in copy constructor/copy assignment
    - Decrement in destructor or when overwritten with copy assignment
- Frees the resource when reference count hits 0

And finally, `std::weak_ptr`

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

To access the referenced data, must convert to `std::shared_ptr`

```
std::shared_ptr<T> tempSharedPtr = weakPtr.lock();
```

And finally, `std::weak_ptr`

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

To access the referenced data, must convert to `std::shared_ptr`

```
std::shared_ptr<T> tempSharedPtr = weakPtr.lock();
```

Used to observe a resource without influencing its lifetime (example use case: use weak_ptr to determine if a shared_ptr has deallocated a resource), or to break shared_ptr cycles.

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;
```

But wait . . . aren't we technically still using new?

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

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

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

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)!
```

Which way is better?

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

Which way is better?

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

Answer:

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

Final notes

- `std::unique_ptrs` are used often
- `std::shared_ptrs` and `std::weak_ptrs` are not used as often

~~🤔~~ Questions? ~~🤔~~

Live Code Demo:
Smart pointers in action

Next time

Guest lecture