Lecture 15: RAII and Smart Pointers

CS 106L, Fall '20
Agenda

- Exceptions
- RAII
- Smart pointers
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();
}
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();
}
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();
}
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();
}
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();
}
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

```cpp
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();
}
```
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();
}
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();
}
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();
}
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();
}

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();
}
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();
}
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?

```cpp
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();
}
```
Aside: Exceptions

Exceptions are a way to signal that something has gone wrong

```java
try {
    // code associated with exception handler
} catch ( [exception type] e ) {
    // exception handler
} catch ( [exception type] e ) {
    // exception handler
} // etc.
```
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.
Takeaway: there are often more code paths than meet the eye!
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;
}
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!

```cpp
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.

<table>
<thead>
<tr>
<th>Acquire</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>delete</td>
</tr>
<tr>
<td>open</td>
<td>close</td>
</tr>
<tr>
<td>try_lock</td>
<td>unlock</td>
</tr>
<tr>
<td>socket</td>
<td>close</td>
</tr>
</tbody>
</table>

- Heap memory
- Files
- Locks
- Sockets
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 used by a class should be **acquired** in the constructor.
- All resources used by a class should be **released** in the destructor.
What is RAI?I?

- 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?

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

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

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

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

    input.close();
}
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
🤔 Questions? 🤔
This is also not RAII-compliant!

```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();
}
```
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.

```cpp
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 of your class, release in the destructor.
- Clients of an RAII class won’t have to worry about mismanaged resources.
But what about RAII for memory?
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? 🤔
Smart Pointers
(RAII for memory!)
We just saw how locks could be made RAII-compliant

```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();
}
```
...where the fix was to wrap it in a special object

```cpp
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?

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

    // do some stuff with n...

    delete n;
}
```
You learned this in CS 106B -- is this RAII-compliant?

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

    // do some stuff with n...
    // if exception thrown, n never deleted!
    delete n;
}
```
void rawPtrFn () {
    Node* n = new Node;
    // do some stuff with n…
    // if exception thrown, n never deleted!
    delete n;
}
Solution: built-in “smart” (RAII-compliant) pointers

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

- Uniquely owns its resource and deletes it when the object is destroyed
- Cannot be copied
Before

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

After

```cpp
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

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

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

We then make a copy of our unique_ptr

```cpp
unique_ptr<int> x;

unique_ptr<int> y;

resource

resource

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

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

```
unique_ptr<int> y;
unique_ptr<int> x;
```
This leaves x pointing at deallocated data
We can’t copy an \texttt{std::unique\_ptr}, but what if we could?

If we dereference \texttt{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_ptr
- 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::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?

```cpp
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!

```cpp
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?

```cpp
std::unique_ptr<T> up{new T};
std::unique_ptr<T> up = std::make_unique<T>();
```
Which way is better?

```cpp
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_ptr`s are used often
- `std::shared_ptr`s and `std::weak_ptr`s are not used as often
🤔 Questions? 🤔
Live Code Demo:
(If time) Smart pointers in action