Final Lecture!

Bad Dad Joke of the Day:
- Me: I'm not sure if I can come up with a bad dad joke.
- Dad: Hi not sure if I can come up with a bad dad joke, I'm dad!

Creds: JX
Game Plan

- Final Lecture Topic
- Finishing Up Smart Pointers
Let’s Talk About...
…Multithreading!
What is a thread?
What is a thread?

Code is usually sequential.
What is a thread?

Code is usually sequential.

Threads are ways to parallelise execution.
What is a thread?
What is a thread?
What is a thread?
What is a thread?
What is a thread?
What is a thread?

Set $a = 2, b = 1$
What is a thread?

Set \( a = 2, \ b = 1 \)

\( b \ += \ a \)

\( a = 5 \)
What is a thread?

Set $a = 2, b = 1$

$b += a$

$a = 5$

$b = ???$
This is known as a data race!

Set $a = 2, b = 1$

$b += a$

$a = 5$

$b = ???
We’ve already seen locks with RAII!

```cpp
void cleanDatabase (mutex& dbLock, map<int, int>& database) {
  databaseLock.lock();
  // other threads will not modify database
  // modify the database
  // if exception, mutex never unlocked!
  databaseLock.unlock();
}
```

```cpp
void cleanDatabase (mutex& dbLock, map<int, int>& database) {
  lock_guard<mutex> lg(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.
Return of the STL!

![Multithreading in C++](http://www.cplusplus.com/reference/multithreading/)

<table>
<thead>
<tr>
<th>Headers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;atomic&gt;</code></td>
<td>Atomic (header)</td>
</tr>
<tr>
<td><code>&lt;thread&gt;</code></td>
<td>Thread (header)</td>
</tr>
<tr>
<td><code>&lt;mutex&gt;</code></td>
<td>Mutex (header)</td>
</tr>
<tr>
<td><code>&lt;condition_variable&gt;</code></td>
<td>Condition variable (header)</td>
</tr>
<tr>
<td><code>&lt;future&gt;</code></td>
<td>Future (header)</td>
</tr>
</tbody>
</table>
Things to Take Away:

- Use atomic types if doing multithreading!
- `std::lock_guard` vs. `std::unique_lock`
- 3 types of “locks”/mutexes: normal, timed, recursive
- Condition variables allow cross-thread communication
  - see CS 110
- `std::async` is one way to use multithreading

- Let’s see how to do multithreading ourselves!

18 November 2019
Example
Multithreading in Action
Announcements

● This is our last class! We’ve loved having you all as students, and loved getting to know you throughout the quarter. :)

● Keep in touch with us after 106L - we’ll gladly still answer questions about C++, CS, Stanford, life, etc.
  ○ Avery: averywang@stanford.edu
  ○ Anna: aszeng@stanford.edu
Announcements

● This is our last class! We’ve loved having you all as students, and loved getting to know you throughout the quarter. :)

● Keep in touch with us after 106L - we’ll gladly still answer questions about C++, CS, Stanford, life, etc.
  ○ Avery: averywang@stanford.edu
  ○ Anna: aszeng@stanford.edu

Let’s finish up RAII!
Perhaps with a guest appearance from multithreading...
Recap
Problem: We can’t guarantee this function will not have a memory leak.

```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;
    return result;
}
```
How do we guarantee classes release their resources?

Regardless of exceptions!
RAII!

Acquire resources in the constructor, release in the destructor.

Use a wrapper class that handles all the resource management for you!
We previously saw how to make file reading RAI1 compliant using a wrapper class:
We previously saw how to make file reading RAII compliant using a wrapper class:

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

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

    input.close();
}
```

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

    // read file
    // no close call needed!
}
// stream destructor
// releases access to file
```
We previously saw how to make locks RAII compliant using a wrapper class:
We previously saw how to make locks RAII compliant using a wrapper class:

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

    databaseLock.lock();
    // other threads will not modify database
    // modify the database
    // if exception, mutex never unlocked!
    databaseLock.unlock();
}
```

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

    // other threads will not modify database
    // modify the database
    // if exception thrown, that’s fine!
    lock_guard<mutex> lg(databaseLock);
    // no release call needed
    } // lock always unlocked when function exits.
```
We previously saw how to make *pointers* RAIi compliant using a *wrapper class*: 
We previously saw how to make pointers RAIII compliant using a wrapper class:

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

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

```cpp
} // Freed!
```
We previously saw how to make pointers RAIU compliant using a wrapper class:

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

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

```cpp
void rawPtrFn() {
    std::shared_ptr<Node> n(new Node);
    // do some stuff with n
}
```
Let’s take a closer look at how we declared a new smart pointer:

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

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

```cpp
void rawPtrFn () {
    std::shared_ptr<Node> n(new Node);
    // do some stuff with n
}
// Freed!
```
Let’s take a closer look at how we declared a new smart pointer:

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

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

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

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

It’s trickier than you might think!
C++ has two main built-in smart pointers:

\texttt{std::unique\_ptr}

\texttt{std::shared\_ptr}
C++ has two main built-in smart pointers:

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

std::shared_ptr<Node> n(new Node);
```
C++ also has built-in smart pointer creators!

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

std::shared_ptr<Node> n(new Node);
```
C++ also has built-in smart pointer creators!

```cpp
std::unique_ptr<Node> n(new Node);
std::unique_ptr<Node> n = std::make_unique<Node>();
std::shared_ptr<Node> n(new Node);
```
C++ also has built-in smart pointer creators!

```cpp
std::unique_ptr<Node> n(new Node);
std::unique_ptr<Node> n = std::make_unique<Node>();
std::shared_ptr<Node> n(new Node);
std::shared_ptr<Node> n = std::make_shared<Node>();
```
C++ also has built-in smart pointer creators!

```cpp
std::unique_ptr<Node> n(new Node);
std::unique_ptr<Node> n = std::make_unique<Node>();
```
C++ also has built-in smart pointer creators!

```cpp
std::unique_ptr<Node> n(new Node);
std::unique_ptr<Node> n = std::make_unique<Node>();
```

Which is better to use?
Which is better to use?
Which is better to use?

3 rules:
● Arguments to a function are evaluated before the function
● Each function is “atomic”
● Arguments may be interleaved otherwise
Which is better to use?

3 rules:
- Arguments to a function are evaluated before the function
- Each function is “atomic”
- Arguments may be interleaved otherwise

\[
\text{f( expr1, expr2 )};
\]
Which is better to use?

3 rules:
- Arguments to a function are evaluated before the function
- Each function is “atomic”
- Arguments may be interleaved otherwise

```c
f(expr1, expr2);
f(g(expr1), h(expr2));
```
Which is better to use?

3 rules:
- Arguments to a function are evaluated before the function
- Each function is “atomic”
- Arguments may be interleaved otherwise

f( expr1, expr2 );

f( g(expr1), h(expr2) );

f( std::unique_ptr<T1>{ new T1 }, std::unique_ptr<T2>{ new T2 } );
Which is better to use?

3 rules:
- Arguments to a function are evaluated before the function
- Each function is “atomic”
- Arguments may be interleaved otherwise

```c++
f( expr1, expr2 );
f( g(expr1), h(expr2) );
```

```c++
f( std::unique_ptr<T1>{ new T1 }, std::unique_ptr<T2>{ new T2 } );
```

What might go wrong here?
Which is better to use?

3 rules:
- Arguments to a function are evaluated before the function
- Each function is “atomic”
- Arguments may be interleaved otherwise

\[
\begin{align*}
\text{f( expr1, expr2 );} \\
\text{f( g(expr1), h(expr2) );} \\
\text{f( std::unique_ptr<T1>{ new T1 }, std::unique_ptr<T2>{ new T2 });} \\
\text{f( std::make_unique<T1>(), std::make_unique<T2>());}
\end{align*}
\]
Which is better to use?

3 rules:
● Arguments to a function are evaluated before the function
● Each function is “atomic”
● Arguments may be interleaved otherwise

Note: The last rule has now been changed in C++17!
But we still prefer the wrapper functions - make_shared has some performance benefits, etc.
C++ also has built-in smart pointer creators!

```cpp
std::unique_ptr<Node> n(new Node);
std::unique_ptr<Node> n = std::make_unique<Node>();
```

Which is better to use?
C++ also has built-in smart pointer creators!

```cpp
std::unique_ptr<Node> n(new Node);
std::unique_ptr<Node> n = std::make_unique<Node>();
```

Which is better to use?

**Always use** `std::make_unique<Node>()`!
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.
In modern C++, we pretty much never use `new` and `delete`!
Where to go from here?
Where to go?

Use C++!
Where to go?

Use C++!

Further C++ reading:

- **Accelerated C++** by Andrew Koenig
- **Effective C++** by Scott Meyers
- **Effective Modern C++** by Scott Meyers
- **Exceptional C++** by Herb Sutter
- **Modern C++ Design** by Andrei Alexandrescu
- **C++ Template Metaprogramming** by Abrahams and Gurtovoy
- **C++ Concurrency in Action** by Anthony Williams
Where to go?

Use C++!

Further C++ reading:

- **Accelerated C++** by Andrew Koenig
- **Effective C++** by Scott Meyers
- **Effective Modern C++** by Scott Meyers
- **Exceptional C++** by Herb Sutter + blog!
- **Modern C++ Design** by Andrei Alexandrescu
- **C++ Template Metaprogramming** by Abrahams and Gurtovoy
- **C++ Concurrency in Action** by Anthony Williams
Thank you!