Multithreading

CS 106L Spring 2020 – Avery Wang and Anna Zeng

Game Plan

- Finishing Up Smart Pointers
- Announcements
- Multithreading

Recap

Problem: We can’t 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;
}
```

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 RAII compliant using filestreams:

We previously saw how to make file reading RAII compliant using filestreams:

```
void printFile () {
  ifstream input();
  input.open("hamlet.txt");
  string line;
  while (getline(input, line)) {
    cout << line << endl;
  }
  input.close();
}
```

```
void printFile () {
  ifstream input("hamlet.txt");
  string line;
  while (getline(input, line)) {
    cout << line << endl;
  }
  // no close call needed!
}
// stream destructor
// releases access to file
```

We previously saw how to make locks RAII compliant using lock_guards:

We previously saw how to make locks RAII compliant using lock_guards:

```
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();
}
```

```
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_guard destructor
// releases lock
```

We previously saw how to make pointers RAII compliant using smart pointers:

We previously saw how to make pointers RAII compliant using smart pointers:

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

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

} // Freed!
```

We previously saw how to make pointers RAII compliant using smart pointers:

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

→

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

} // Freed!
```

→

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

} // Freed!
```

And we saw a better way to declare smart pointers:

And we saw a better way to declare smart pointers:

```
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>();
```

Today we'll learn why it's better!

```
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>();
```

But First...

...Multithreading!

What is a thread?

What is a thread?

Code is usually sequential.

What is a thread?

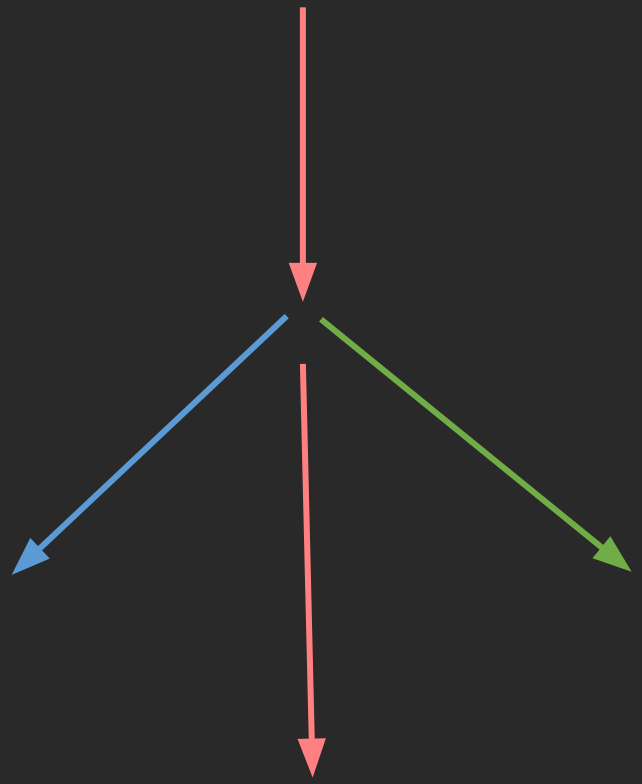
Code is usually sequential.

Threads are ways to parallelise execution.

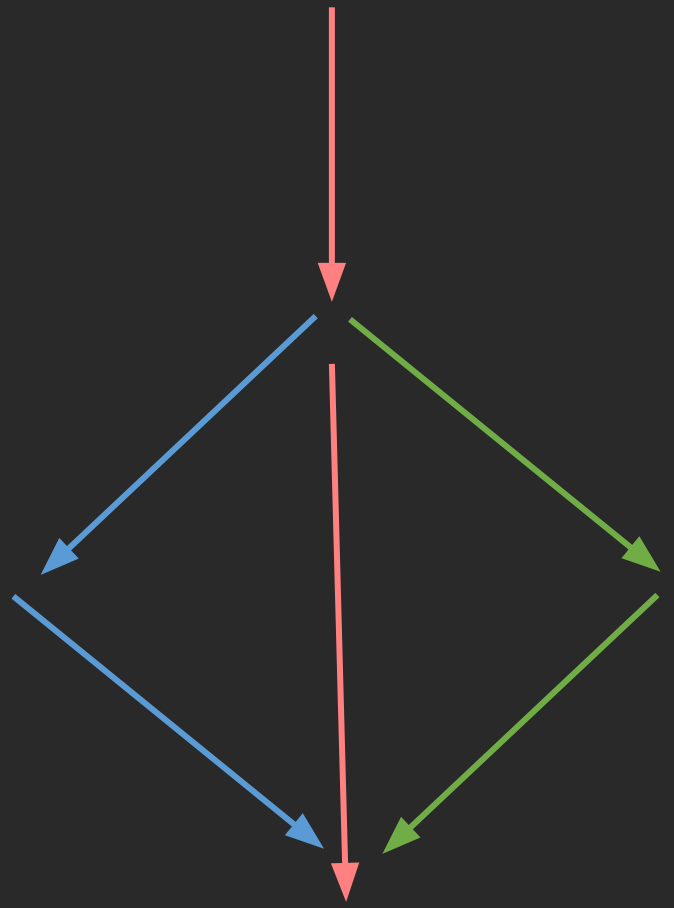
What is a thread?

What is a thread?

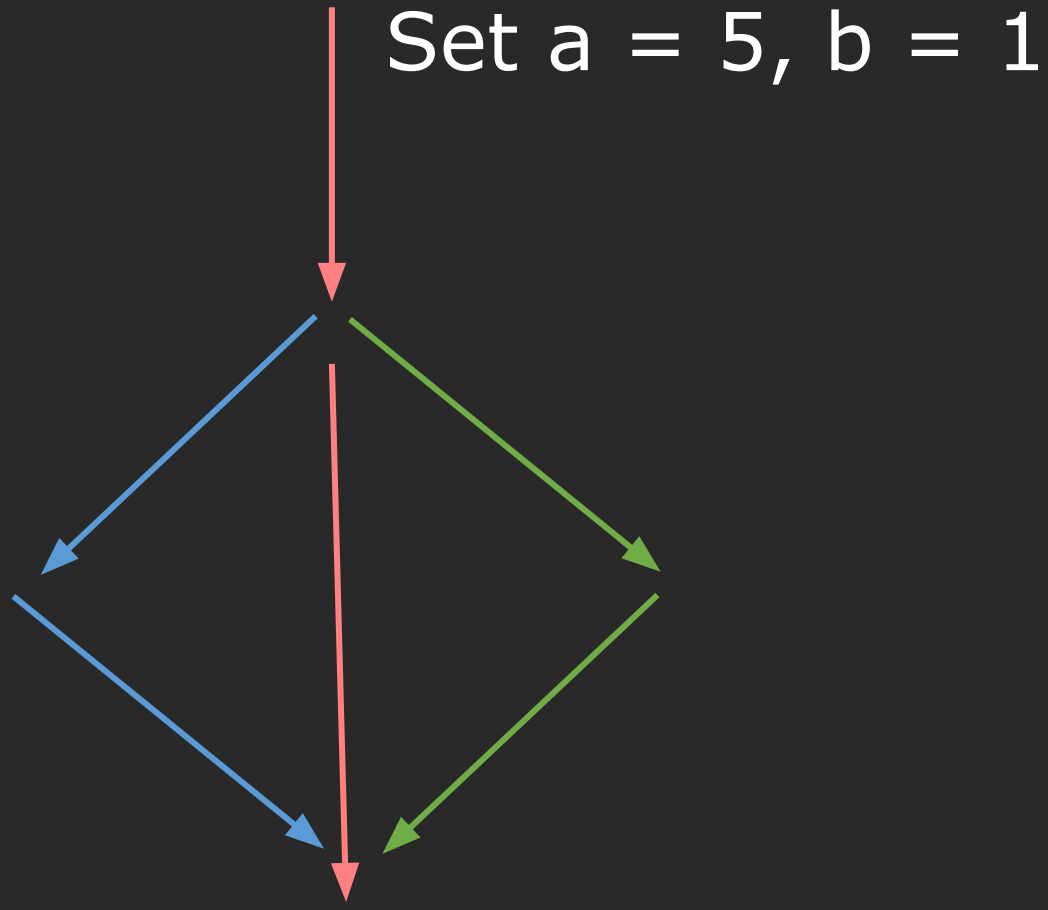
What is a thread?



What is a thread?



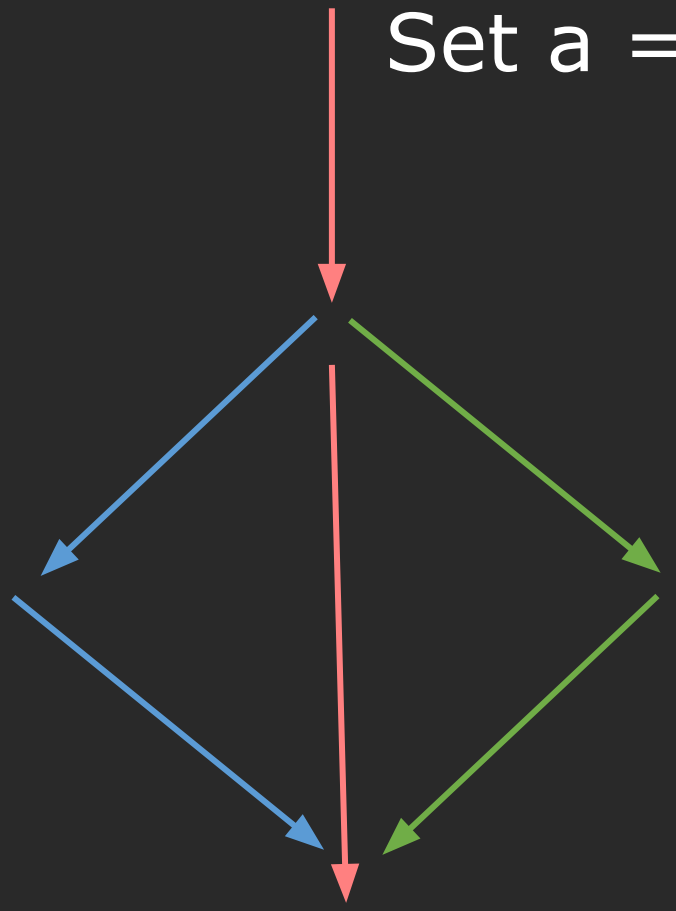
What is a thread?



What is a thread?

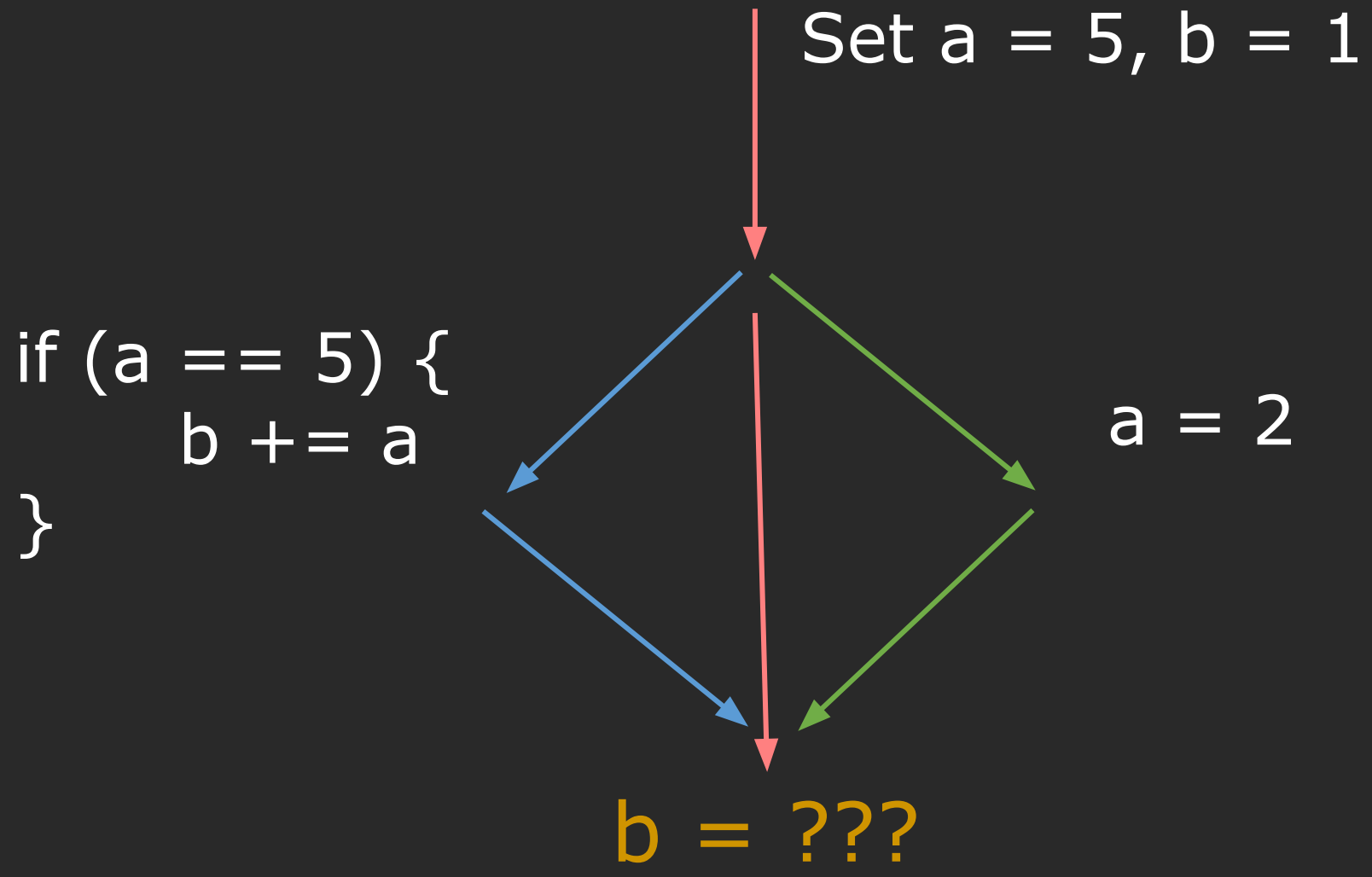
Set a = 5, b = 1

```
if (a == 5) {
      b += a
}
```

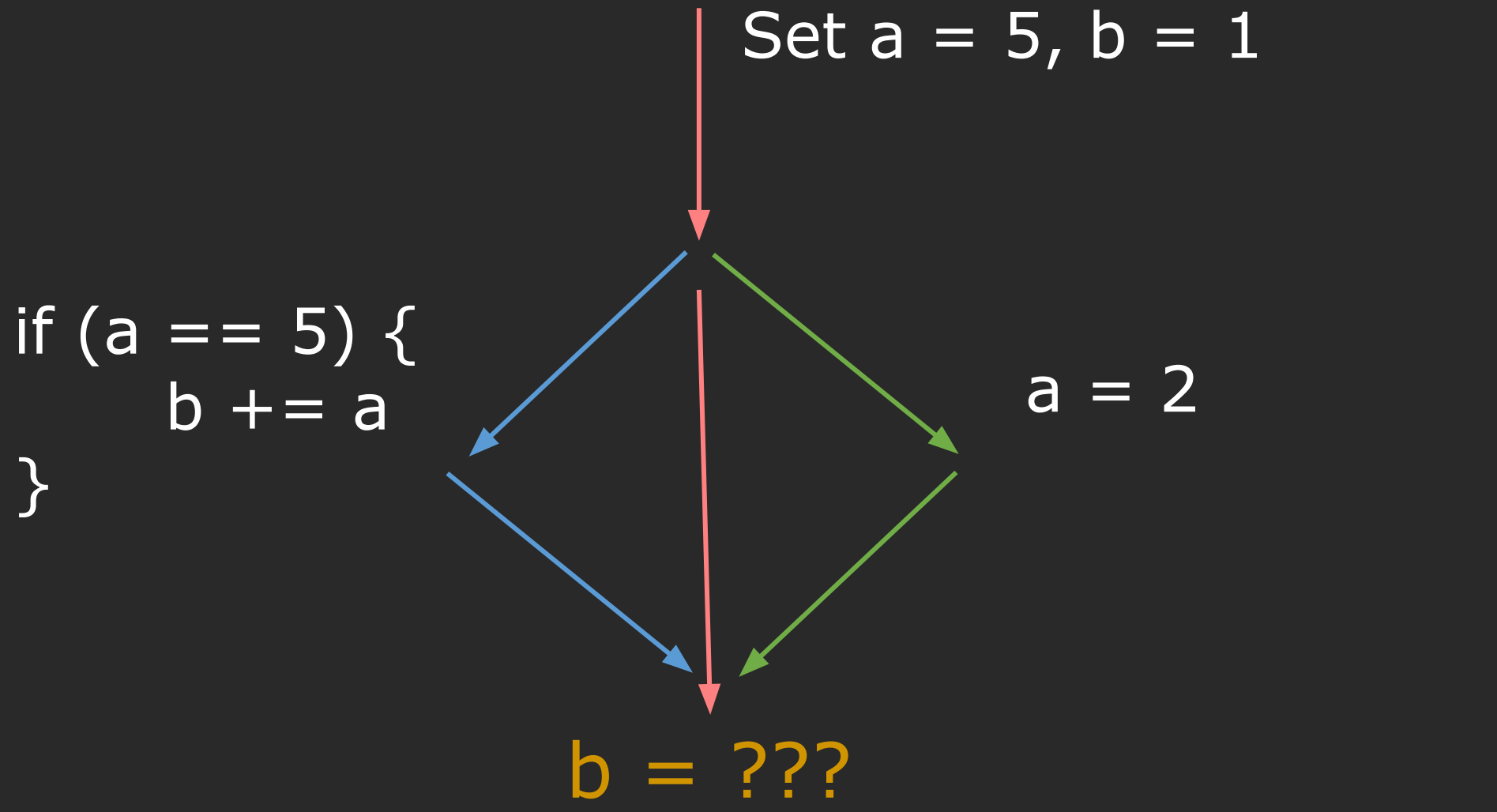


a = 2

What is a thread?



This is known as a data race!

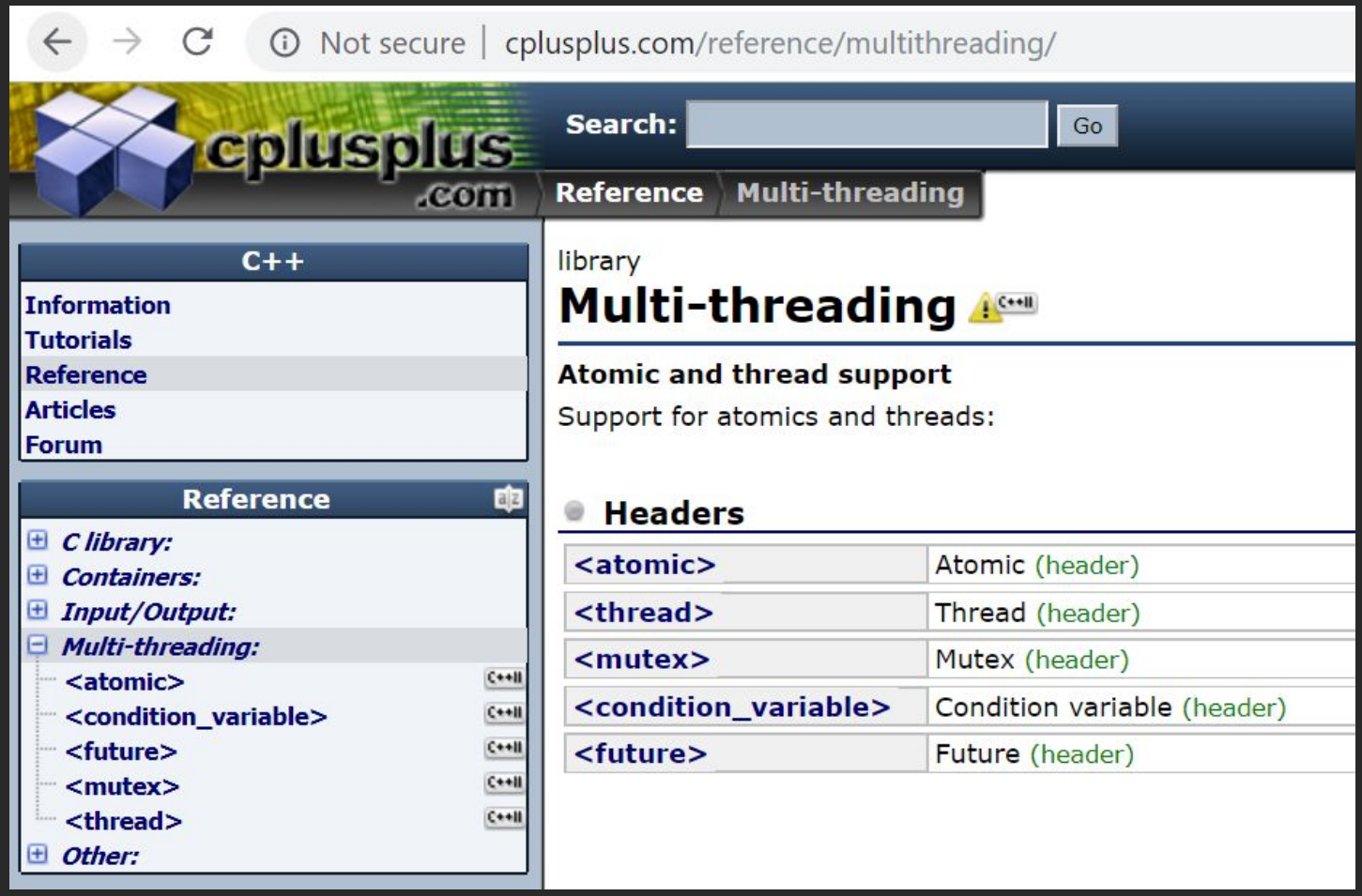


We've already seen locks with RAII!

```
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();
}
```

```
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_guard destructor
// releases lock
```

Return of the STL!



http://www.cplusplus.com/reference/multithreading/

Things to Take Away

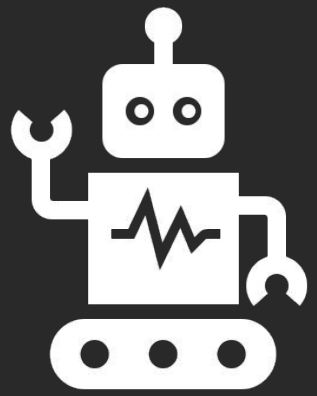
- When do I need to use locks?
  - Internally synchronized: mutex, condition_variable, <atomic>
  - With any other type, you are responsible for synchronizing (i.e. making your code thread-safe)!

- 3 types of locks: normal, timed, recursive
- `std::lock_guard` vs. `std::unique_lock`
- Condition variables allow cross-thread communication
  - see CS 110

Synchronization and Class Design

- In your class design, it’s your responsibility to ensure that internal state is thread-safe!

- See this link for a fuller discussion on this topic!



Example

Multithreading in Action

Bad Dad Joke of the Day 1:

- What did the grape say when it was stepped on?
- Nothing, it just let out a little wine.

Creds: Jared

Bad Dad Joke of the Day 2:

- What do you call a dog that can do magic?
- A labracadabrador.

Creds: Nick

2-min stretch break!

Bad Dad Joke of the Day 1:

- What did the grape say when it was stepped on?
- Nothing, it just let out a little wine.

Creds: Jared

Bad Dad Joke of the Day 2:

- What do you call a dog that can do magic?
- A labracadabrador.

Creds: Nick

2-min stretch break!

Bad Dad Joke of the Day 3:

- Why aren't Koalas considered mammals?
- Because they do not meet the koala-ilifications!

Creds: Wilmer

Bad Dad Joke of the Day 4:

- I don't know any jokes about bad dads

Creds: Anton

Announcements

Announcements

- Reminder to fill out the form for final lecture!
  - Also, come to final lecture to be part of our EOQ screenshot!
- Assignment 2 grades are (almost all) out!
- Assignment 3 due this Sunday, 6/7, 11:59 pm
  - Hard deadline: Wednesday, 6/10, 11:59 pm
- Assignment 3 has been revised!

Smart Pointer Creation

It’s trickier than you might think!

Recap: Smart Pointers

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

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

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

Recap: Smart Pointers

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

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

Recap: Smart Pointers

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

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

Recap: Smart Pointers

```
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>();
```

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>();
```

Rules of Function Calls

```
f( expr1, expr2 );
```

Rule #1:

Arguments to a function are evaluated before the function. Order is not guaranteed!

Rules of Function Calls

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

Rule #2:

Each function is “atomic”.
Arguments may be interleaved otherwise.

Rules of Function Calls

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

Rules:

Arguments to a function are evaluated before the function.
Order is not guaranteed!

Each function is “atomic”.
Arguments may be interleaved otherwise.

Rules of Function Calls

```
f( expr1, expr2 );
```

Rules:

Arguments to a function are evaluated before the function.
Order is not guaranteed!
Arguments may be interleaved otherwise.

Rules of Function Calls

```
f( expr1, expr2 );
```

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

Rules:

Arguments to a function are evaluated before the function.
Order is not guaranteed!
Arguments may be interleaved otherwise.

What might go wrong here?

```
f( expr1, expr2 );
```

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

Rules:

Arguments to a function are evaluated before the function.
Order is not guaranteed!
Arguments may be interleaved otherwise.

What might go wrong here?

```
f( expr1, expr2 );
```

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

Rules:

Arguments to a function are evaluated before the function.
Order is not guaranteed!
Arguments may be interleaved otherwise.

The Fix

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

```
f( std::make_unique<T1>(), std::make_unique<T2>());
```

Rules:

Arguments to a function are evaluated before the function.
Order is not guaranteed!
Arguments may be interleaved otherwise.

Each function is “atomic”.

Aside

This rule is no longer true as of C++17:
Arguments may be interleaved otherwise.

But we still prefer the wrapper functions - make_shared has some performance benefits, etc.

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>();
```

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>();
```

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

So, to repeat a takeaway from last lecture:

> ***Guideline:*** *Don't use explicit* ***new***, ***delete***, *and owning* ******* *pointers, except in rare cases encapsulated inside the implementation of a low-level data structure.*

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

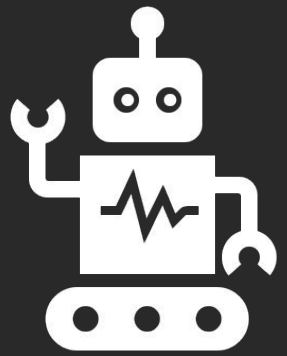
So, to repeat a takeaway from last lecture:

> ***Guideline:*** *Don't use explicit* ***new****,* ***delete****, and owning* ******* *pointers, except in rare cases encapsulated inside the implementation of a low-level data structure.*

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

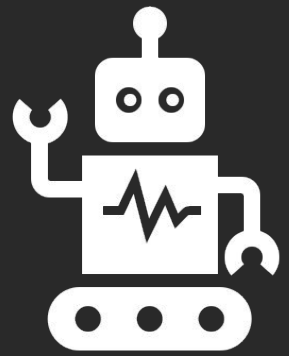
You can read more about this here: https://herbsutter.com/2013/05/29/gotw-89-solution-smart-pointers/ . Overall, highly recommend Herb Sutter's GOTW blog!

If we have time… Last Example: you choose!



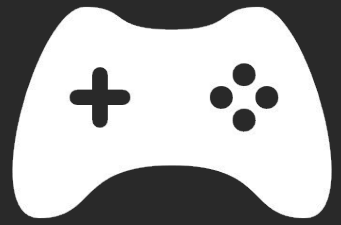
Smart Pointers:

Implementing a Shared Pointer!



Multithreading:

The Classic Ticket Agent Example



Next time

Final Lecture