Multithreading

CS 106L Winter 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 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 RAII compliant using a wrapper class:

We previously saw how to make file reading RAII compliant using a wrapper class:

```
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");

  // 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:

```
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();
}
```

```
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.
```

We previously saw how to make pointers RAII compliant using a wrapper class:

We previously saw how to make pointers RAII compliant using a wrapper class:

```
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 a wrapper class:

```
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!
```

Let’s take a closer look at how we declared a new smart pointer:

```
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!
```

Let’s take a closer look at how we declared a new smart pointer:

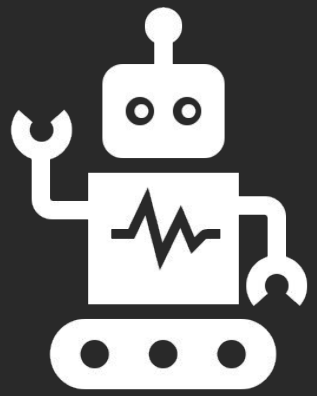
```
void rawPtrFn () {
  Node* n = new Node;
  // do some stuff with n…
  delete n;
}
```

```
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!
```



Example

Implement our own RAII-compliant pointer!

Smart Pointer Creation

It’s trickier than you might think!

C++ has two main built-in smart pointers:

`std::unique_ptr`

`std::shared_ptr`

C++ has two main built-in smart pointers:

```
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!

```
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!

```
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!

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

C++ also has built-in smart pointer creators!

```
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

```
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

```
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

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

```
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

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

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

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!

```
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!

```
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.

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

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`!

Announcements

Announcements

- Reminder to fill out the form for final lecture!
  - Also, come to final lecture to be part of our EOQ selfie!
- Assignment 2 grades will be coming out tomorrow
- Assignment 3 due this Friday, 3/6, 11:59 pm

Let’s Talk About...

...Multithreading!

What is a thread?

What is a thread?

Code is usually sequential.

What is a thread?

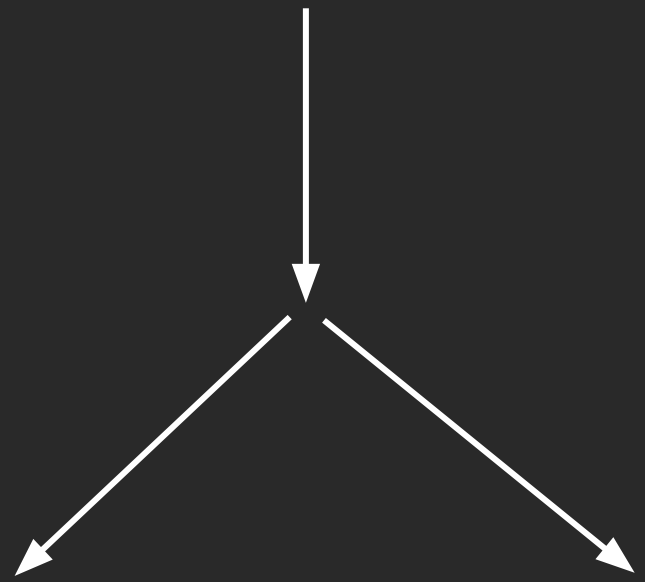
Code is usually sequential.

Threads are ways to parallelise execution.

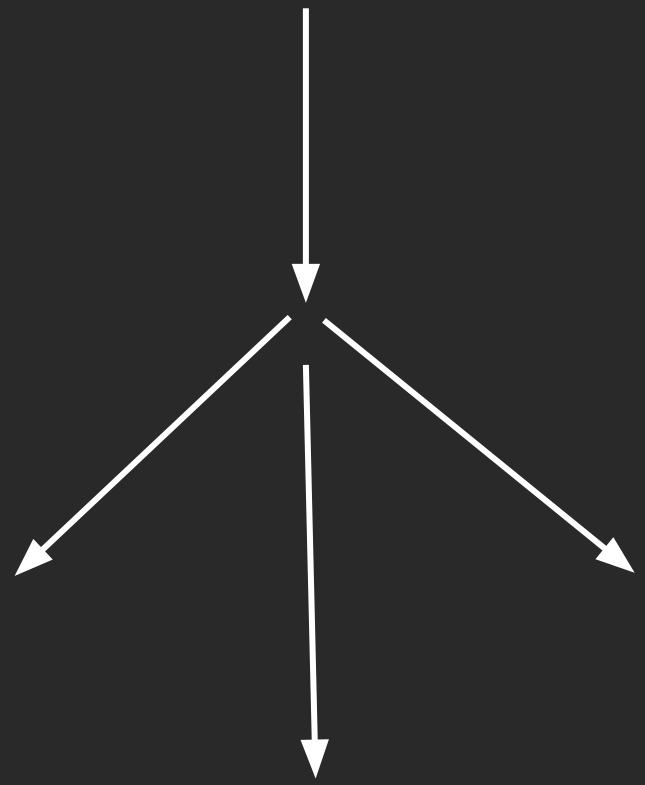
What is a thread?

What is a thread?

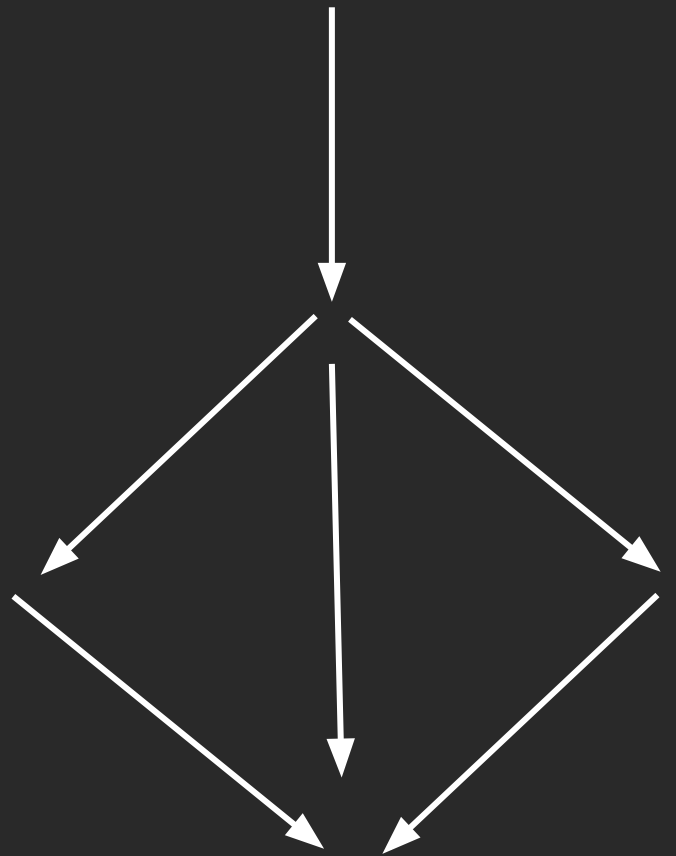
What is a thread?



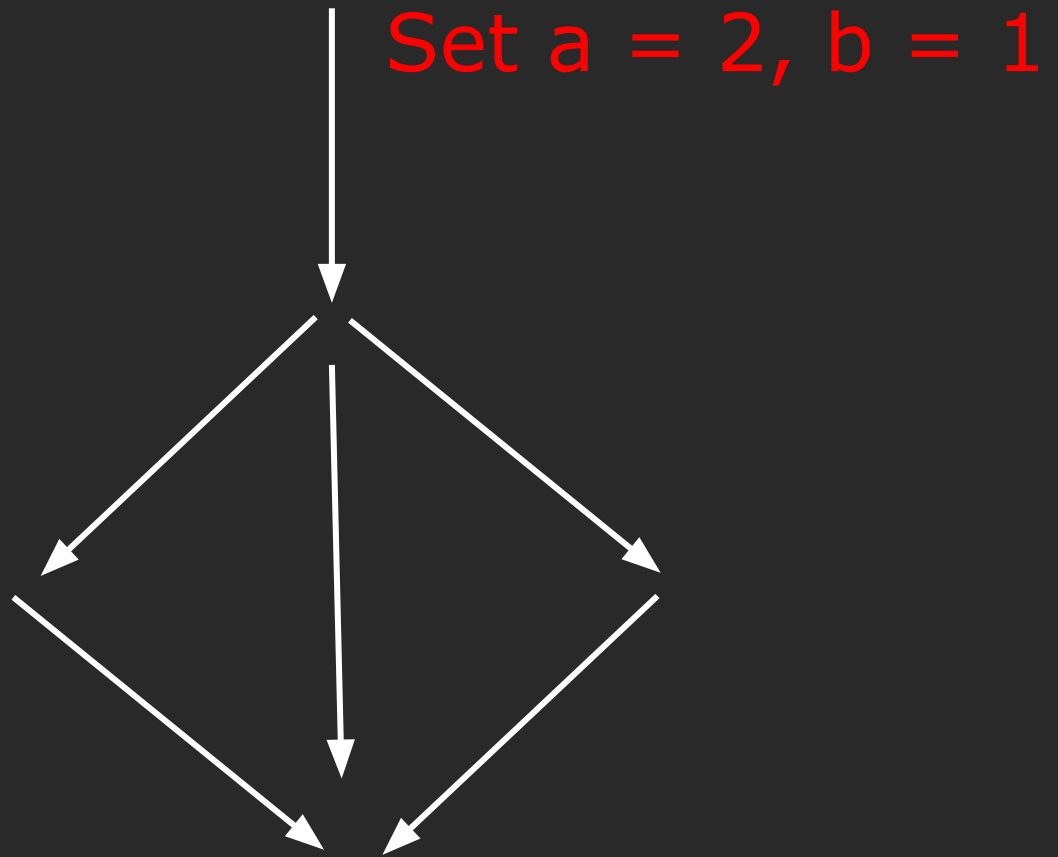
What is a thread?



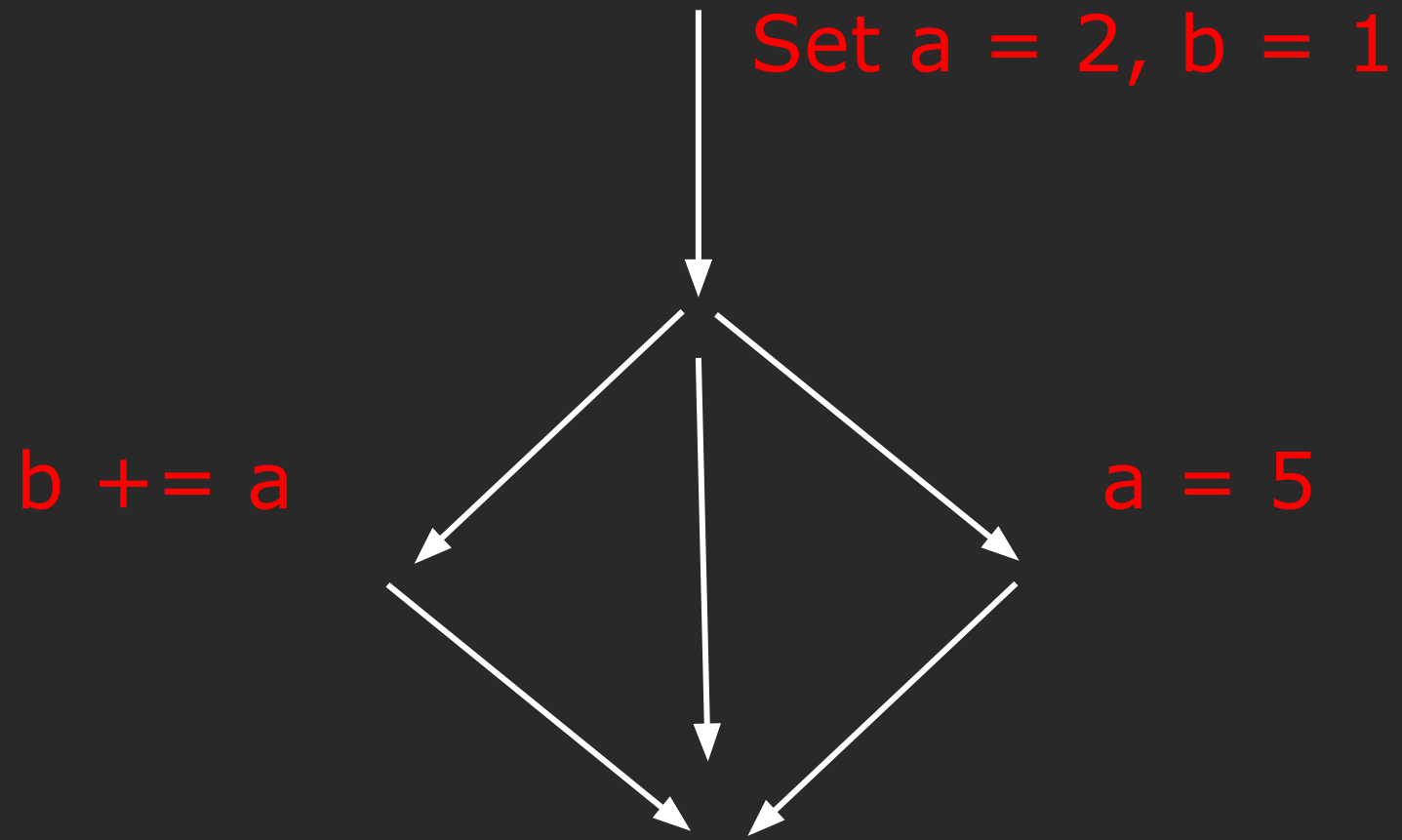
What is a thread?



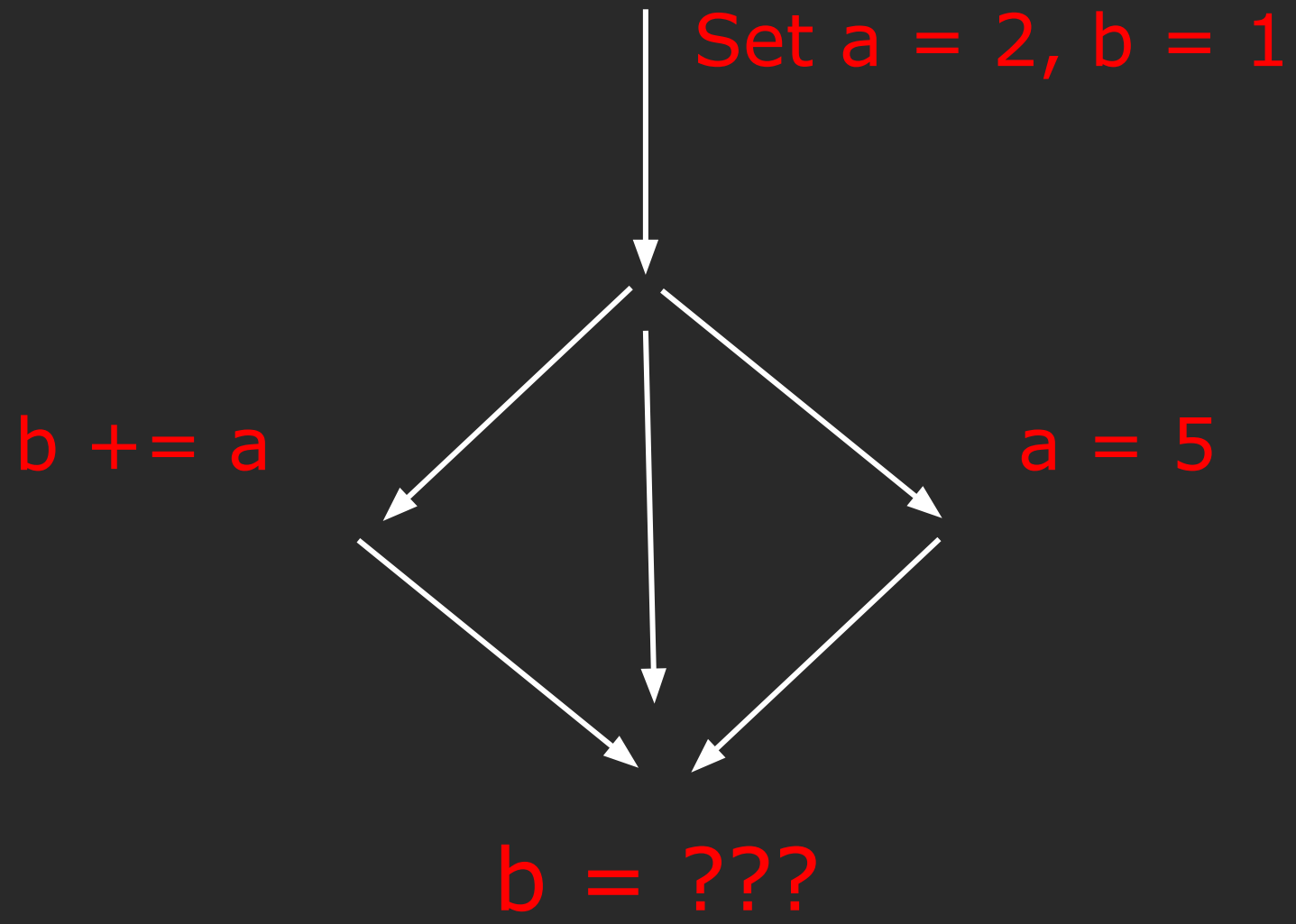
What is a thread?



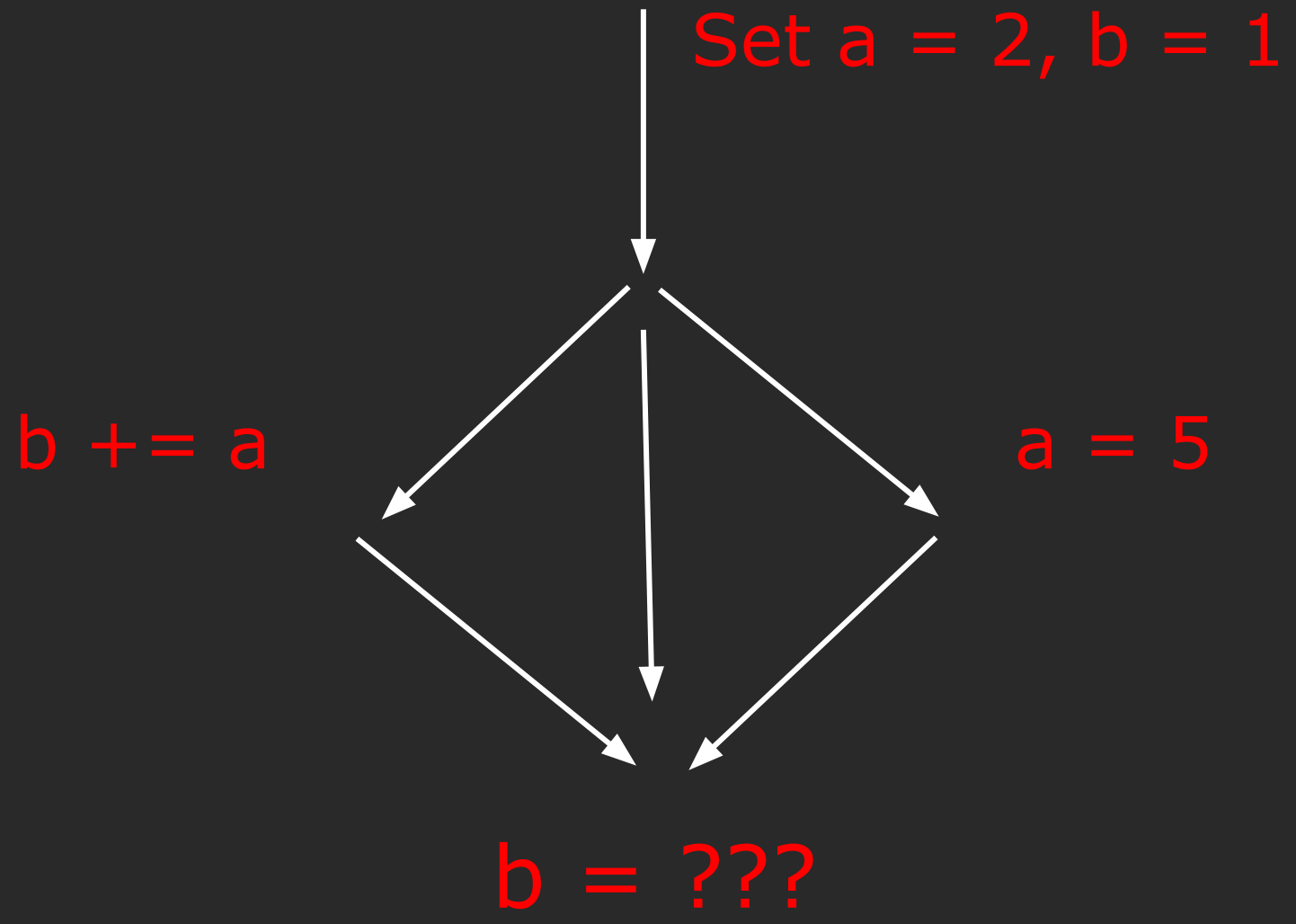
What is a thread?



What is a thread?



This is known as a data race!



We've already seen locks with RAII!

```
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();
}
```

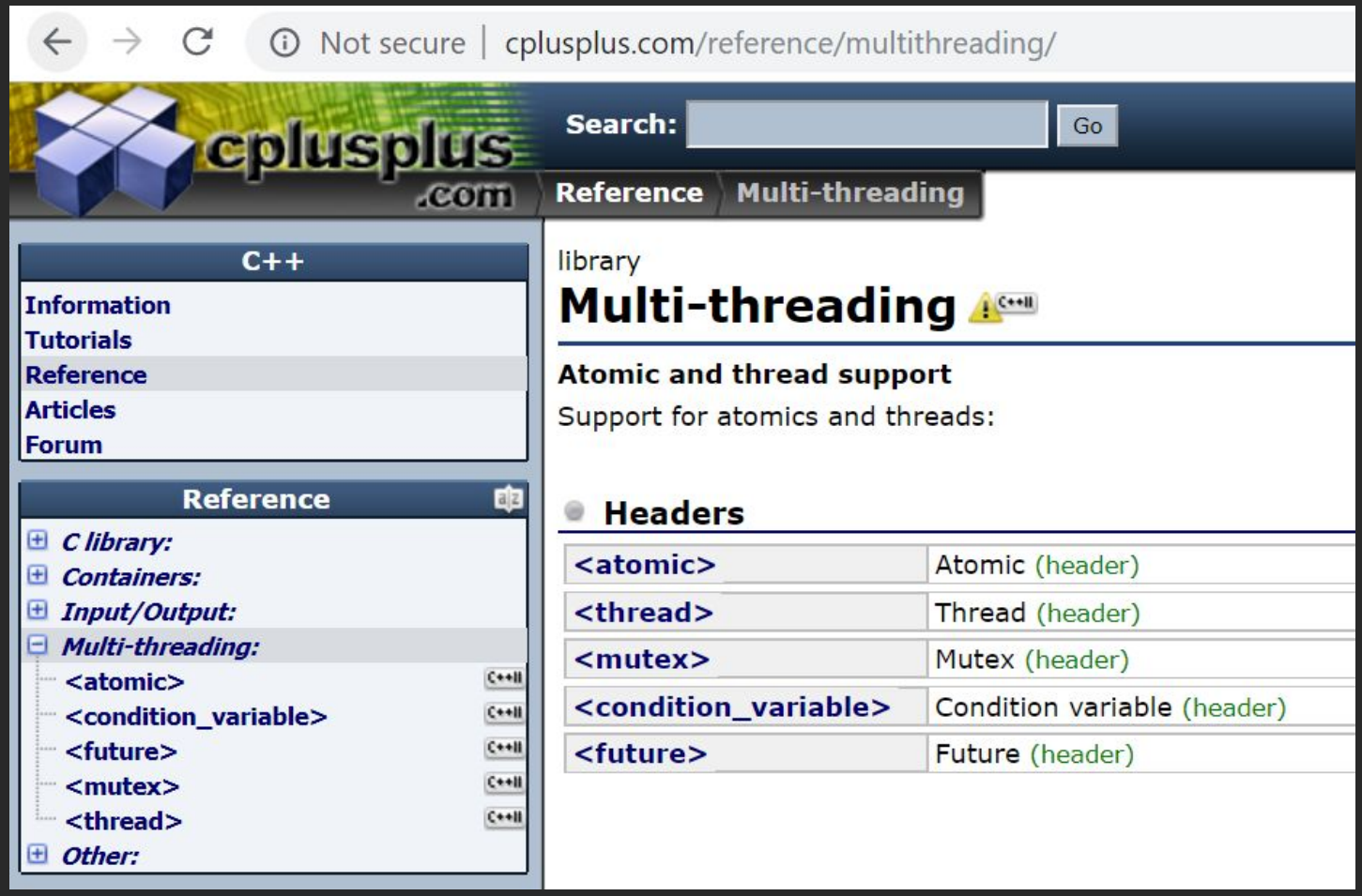
```
void cleanDatabase (mutex& dbLock,
                    map<int, int>& database) {

  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!

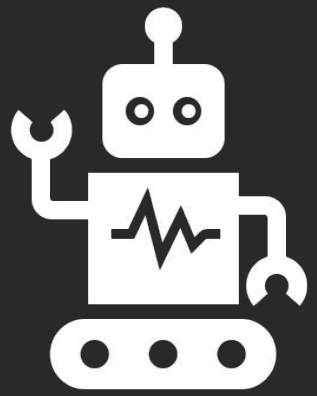


http://www.cplusplus.com/reference/multithreading/

Things to Take Away:

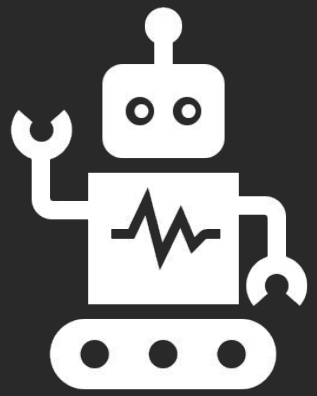
- Use atomic types if doing multithreading!
- `std::lock_guard` vs. `std::unique_lock`
- 3 types of "locks"/mutexes: normal, timed, recursive
- Condition variables allow cross-thread communication
  - see CS 110
- `std::async` is one way to use multithreading

- Let's see how to do multithreading ourselves!



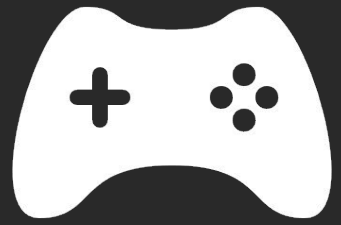
Example

Multithreading in Action



Example

If we have time… the Classic Ticket Agent Example



Next time

Final Lecture