CS 111 Assignment 2: Synchronization

Initializer Lists and Destructors

```
class Printer {
public:
    Printer(std::string name);
    ~Printer();
private:
    std::string my_name_;
    int dummy_;
};

Printer::Printer(std::string name)
        : my_name_(name), dummy_(42)
{
    std::cout << "Constructing " << my_name_ << std::endl;
}

Printer::~Printer()
{
    std::cout << "Destructing " << my_name_
              << " with dummy value " << dummy_ << std::endl;
}
```

Initializer list: used to initialize member variables

Destructor: invoked to clean up when object is deleted

When are Destructors Invoked?

- **Stack-allocated objects: when object goes out of scope**
- **Dynamically allocated objects ("new Foo"): when object is deleted**

destruct.cc

```
Printer *test_function()
{
    Printer p1("p1");
    Printer *p2 = new Printer("p2");
    for (int i = 0; i < 5; i++) {

        std::cout << "Loop iteration " << i << std::endl;
    }
    return p2;
}

int
main(int argc, char *argv[])
{
    Printer *p = test_function();
    std::cout << "test_function returned" << std::endl;
    delete p;
    return 0;
}
```

Lock guards: std::unique_lock

- **Constructor and destructor automatically lock and unlock mutex**

  Can't accidentally forget to unlock!

```
class unique_lock {
public:
   unique_lock(std::mutex& mutex);
   ~unique_lock();
private:
   std::mutex& mutex;
}

unique_lock::unique_lock(std::mutex& mutex)
      : mutex(mutex)
{
   mutex.lock();
}

unique_lock::~unique_lock()
{
   mutex.unlock();
}
```

CalTrain Automation

- **Train:**

  Calls load_train(int available) when it arrives at the station

  available is the number of empty seats on the train.

  Must block until either

  - No more seats are available and all passengers have boarded, or
  - No more passengers are waiting at the station and all passengers have boarded

- **Passenger:**

  Calls wait_for_train() when it arrives at the station

  wait_for_train() blocks until a train arrives with open seats

  - i.e. load_train() is called by a train thread

  When wait_for_train returns, passenger starts boarding

  Calls boarded() once safely on board

Lock guards, cont'd

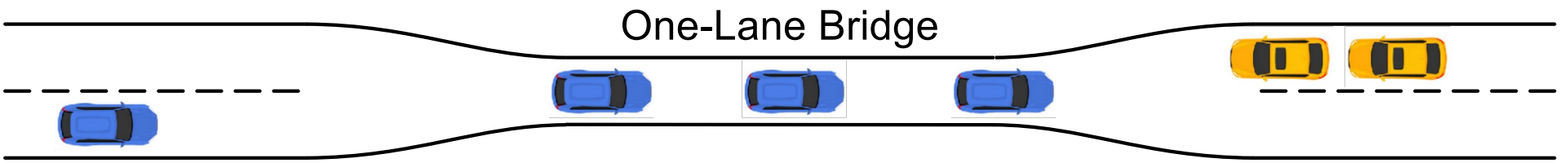
- **Without std::unique_lock:**

```
void foo()
{
    mutex.lock();
    ...
    if (...) {
        mutex.unlock();
        return;
    }
    ...
    mutex.unlock();
}
```

- **With std::unique_lock:**

```
void foo()
{
    std::unique_lock<std::mutex> lock(mutex);
    ...
    if (...) {
        return;
    }
    ...
}
```

Bridge Problem



- **All cars on bridge must be travelling in the same direction**
- **Any number of cars can be on the bridge at once**
- **Once 5 cars have passed in one direction, if there are waiting cars in the other direction they must get a chance**

Code for Cars

Eastbound cars:

```
bridge.arrive_eb();
... drive across bridge ...
bridge.leave_eb();
```

Westbound cars:

```
bridge.arrive_wb();
... drive across bridge ...
bridge.leave_wb();
```

State Variables

- **What information is needed to know whether an eastbound car can enter the bridge?**

First cut at bridge.hh

```
class Bridge {
public:
    Bridge();
    void arrive_eb();
    void leave_eb();
    void arrive_wb();
    void leave_wb();

private:
    // Synchronizes access to all info in
    // this object.
    std::mutex mutex_;

    // Number of cars currently waiting to
    // cross in each direction
    int waiting_eb_;
    int waiting_wb_;

    // Number of consecutive cars that have
    // crossed in each direction (only
    // one of these will be nonzero at any
    // given time).
    int consecutive_eb_;
    int consecutive_wb_;

    // Number of cars currently crossing in
    // each direction (only one of these
    // can be nonzero at any given time).
    int crossing_eb_;
    int crossing_wb_;

    ...
};
```

Maintain state variables

```
Bridge::Bridge()
    : mutex_(), waiting_eb_(0),
      waiting_wb_(0), consecutive_eb_(0),
      consecutive_wb_(0), crossing_eb_(0),
      crossing_wb_(0)
{
}

void Bridge::arrive_eb()
{
    std::unique_lock<std::mutex> lock(mutex_);
    waiting_eb_++;

    /* Wait until safe to cross */

    waiting_eb_--;
    crossing_eb_++;
    consecutive_eb_++;
    consecutive_wb_ = 0;
}
```

```
void Bridge::leave_eb()
{
    std::unique_lock<std::mutex> lock(mutex_);
    crossing_eb_--;

    /* Maybe wake up westbound cars */
}
```

Final bridge.hh

```
class Bridge {
public:
    Bridge();
    void arrive_eb();
    void leave_eb();
    void arrive_wb();
    void leave_wb();

private:
    // Synchronizes access to all info in
    // this object.
    std::mutex mutex_;

    // Number of cars currently waiting to
    // cross in each direction
    int waiting_eb_;
    int waiting_wb_;

    // Number of consecutive cars that have
    // crossed in each direction (only
    // one of these will be nonzero at any
    // given time).
    int consecutive_eb_;
    int consecutive_wb_;

    // Number of cars currently crossing in
    // each direction (only one of these
    // can be nonzero at any given time).
    int crossing_eb_;
    int crossing_wb_;

    // Signaled to indicate that there are
    // no longer any cars crossing
    // in the eastbound direction.
    std::condition_variable done_eb_;

    // Same for the westbound direction.
    std::condition_variable done_wb_;
};
```

Final bridge.cc

```
Bridge::Bridge()
    : mutex_(), waiting_eb_(0),
      waiting_wb_(0), consecutive_eb_(0),
      consecutive_wb_(0), crossing_eb_(0),
      crossing_wb_(0), done_eb(), done_wb()
{
}

void Bridge::arrive_eb()
{
    std::unique_lock<std::mutex> lock(mutex_);
    waiting_eb_++;

    while ((crossing_wb_ > 0) ||
           ((waiting_wb_ > 0)
           && (consecutive_eb_ >= 5)))
        done_wb_.wait(lock);

    waiting_eb_--;
    crossing_eb_++;
    consecutive_eb_++;
    consecutive_wb_ = 0;
}
```

```
void Bridge::leave_eb()
{
    std::unique_lock<std::mutex> lock(mutex_);
    crossing_eb_--;

    if (crossing_eb_ == 0)
        done_eb_.notify_all();
}
```

General Guidelines

- **Your code will not call the methods you implement**
  - Our test harness will spawn threads, construct objects, and invoke your methods
- **You must use the monitor style discussed in lecture**
  - Exactly one mutex per Station or Party object
- **It should be possible have multiple Station or Party objects**
  - Each operates independently
- **Simplicity is crucial:**
  - If it's complex, it probably won't work
  - We'll take off points for overly complex solutions

Party Introductions

- **Program a mechanism that pairs people up at a party based on their Zodiac signs and preferences**

- **Each person invokes:**

```
std::string meet(std::string& my_name, int my_sign,
                 int other_sign)
```

  my_name: name of person

  my_sign: person's Zodiac sign (integer in {0 … 11})

  other_sign: Zodiac sign they'd like to meet

  Must block until a suitable match is available, then return the name of the matching person

Party Introductions

- **Matches must be mutual:**
  - Suppose Bob has sign 3, wants to meet sign 6
  - If Alice has sign 6 and wants to meet sign 3, then they can match
  - Each person must receive the other person's name
  - If Casey has sign 6 and wants to meet sign 4, cannot match with Bob
- **Matches must occur in parallel:**
  - If Bob is waiting for a match, shouldn't prevent others from being matched

CalTrain Automation

- **Don't overbook train!**
  - Once passenger starts boarding, that means one fewer seat available
- **Wait for passengers to finish boarding!**
  - A passenger is not safely on board until it calls `boarded()`
  - No available seats does not mean all passengers are on board
- **Passengers must be able to board concurrently!**
  - Don't board passengers one at a time