Project 4: Lock and Cond. Var. implementation

- **Task: Implement the `Mutex` and `Conditon` types from project 2**
  - Add fields to structures in `thread.hh`
  - Implement methods in `sync.cc`
- **Some non-standard interface choices to eliminate bugs:**
  - Mutex keeps track of thread that owns it (helps diagnose bugs)
  - Specify Mutex when *constructing* a Condition not waiting (should always use same Mutex with same Condition anyway)
- **Ideally this project should be easier than project 2**
  - Allows time to fix problems in your project 2 submission

Classes to implement

```
// A standard mutex providing mutual exclusion.
class Mutex {
public:
    void lock();
    void unlock();
    bool mine(); // True if lock held by current thread
private:
    // You may want to add private fields
};

class Condition {
public:
    explicit Condition(Mutex &m) : m_(m) {}
    void wait();       // Go to sleep until signaled
    void signal();     // Signal at least one waiter (if any)
    void broadcast(); // Signal all waiting threads
private:
    Mutex &m_;
    // You may want to add private fields
};
```

Common problems in project 2

- **Data races: access data from multiple threads w. $\geq$1 write**
  - If you can't protect with a mutex, must disable interrupts
  - In project it's easy, just put `IntrGuard ig;` on stack
- **Memory leaks linear in number of threads allocated**
  - Free everything you allocate except 1 garbage stack/Thread
- **Other problems**
  - Confusion over memory allocators, construction/deletion
  - Confusion over `unique_ptr` and `Bytes`
  - Confusion over static keyword

Review: Object construction and destruction

- **Objects constructed two ways: in lexical scope, and with `new`**
  - With `new`, dynamically allocates memory with `operator new`
  - Always invokes one of constructor functions `T(...)`
  - Also constructs fields and base classes, initializes vptr
- **Objects destroyed on leaving scope or with `delete`/`delete[]`**
  - Invokes destructor function `~T()`
  - Also destroys fields and supertypes
  - With `delete`, also deallocates memory w. `operator delete`
- **Note different scopes**
  - Block scope: destroyed upon exiting block
  - Global scope: destroyed upon program exit (beware ordering)
  - Static in function: destroyed upon program exit
  - `thread_local`: destroyed upon thread exit
- **Example:** `construct.cc`

Notes on constructors

- **A single-argument constructor can be *explicit* or not**

```
struct MyType {
  MyType(std::nullptr_t);   // implicit
  explicit MyType(int);     // explicit
};
void f(MyType mt);

f(nullptr);     // ok--implicit
f(3);           // error--no such function
f(MyType{3});   // ok--explicit
f(0);           // ok--0 is a valid nullptr_t
```

  - Can implicitly construct a `std::function` from normal function
- **Braces (`MyType{3}`) vs. parens (`MyType(3)`)**
  - Braces disallow narrowing (`short s{my_long};` is illegal)—safer
  - Braces allow list initialization (`int x[] = {1, 2, 3};`)
  - Braces do aggregate/default initialization for simple types without user-defined constructors (`int x{};` makes `x == 0`)

The `static` keyword

- **In a class, `static` means not tied to an instance**
  - Static field ≈ global variable (with `::` in name)
  - Static method ≈ global function that can access private fields

```
struct MyType {
  // Note: struct can't contain itself non-statically
  static MyType static_member;
  static void inc(MyType *mtp) { ++mtp->private_int_; }
private:
  int private_int_ = 0;
};
MyType MyType::static_member{};
```

- **In block scope, static variables last until program exit**
  - Like a global, but constructed first time definition crossed
- **At global scope, `static` only affects "linkage"**
  - Linker doesn't expose statics to other object files

Copying objects

- **Many situations require copying objects in C++:**
  - Constructors: `MyType(Obj o) : o_(o) {}`
  - Declarations: `MyType obj1 = obj2;  MyType obj3(obj4);`
  - Assignment: `obj1 = obj2;`
  - Function return: `obj1 = my_function();`
- **These invoke copy constructor, assignment operators:**

```
struct MyType {
    MyType(const MyType &other) { ... }
    MyType& operator=(const MyType &other) { ... }
};
```

  - C++ supplies implicitly declared versions in many cases
  - Can explicitly delete or request auto-generated ones

```
MyType(const MyType &) = delete;
MyType(const MyType &) = default;
```

- **Problem: copying objects may be expensive or impossible**
  - E.g., copying `std::unique_lock` would unlock twice!

Moving objects

- **Since C++11, some copies can be avoided by *moving* objects**
- **After you move `src` into `dst`, means:**
  - `dst` contains the old value of `src`
  - `src` contains a value but unspecified value (e.g., empty)
- **Why is this useful? Solves problems w. copying in many cases**
  - Large container (e.g., `std::map`)? Just copy pointers, not contents
  - Move `std::lock_guard`? Leave `src` with no mutex
- **Declare move constructors with *rvalue reference* `&&`**

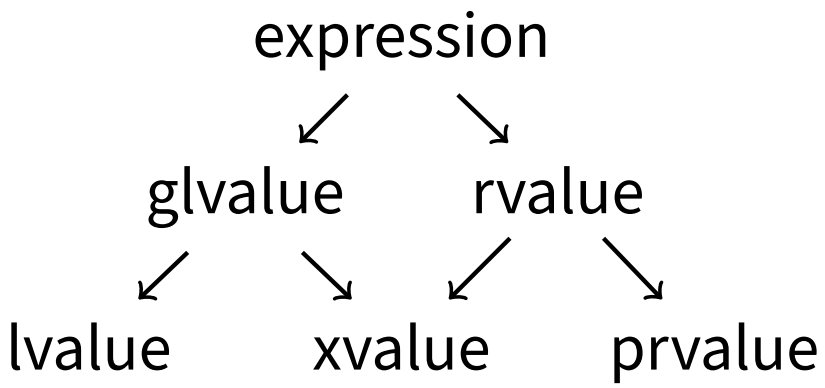
```
struct MyType {
    MyType(MyType &&other) { ... }
    MyType& operator=(MyType &&other) { ... }
};
```

- **Example:** `copymove.cc`

lvalues and rvalues in most languages

- ***lvalue* expression are those that can go to left of =:**
  - `x = 5;  v[3] = 5;  *p = '\n';  (b?p1:p2) = NULL;`
- ***rvalue* expressions are valid only to the right of =:**
  - `x = 5;  x = n+1;  p = malloc(16);`
  - Invalid code: `5 = x;  n+1 = x;  f() = 3;`
- **C++ complicates the picture**
- `const int x = 5;` **– `x` is still an lvalue**
  - May be able to assign anyway with `const_cast` or `mutable`
  - In particular, you still take expression's address (`&x`)
  - So lvalue ≈ you could assign to non-const version
- `int &f();` **– makes function result `f()` an lvalue**
  - So `int&` is called an *lvalue references*

C++ expression categories



- **Every C++ expression (not type) has one of three categories**
- **lvalue: Has address, can assign to if non-const**
  - `x`, `x[a]`, `s.i`, `*p`, `++n`, function call returning `T&`
- **xvalue: An "expiring" value – safe to give arbitrary contents**
  - `MyType{}.field`, `static_cast<T&&>(t)`, `std::move(t)`
- **prvalue: "pure rvalue" used to initialize objects/arguments**
  - `10`, `nullptr`, `n++`, `n+1`, `MyType{}`, `f()` (when not reference), void
  - Might never be materialized (e.g., `MyType x = MyType{3};`)
- **Example:** `category.cc`

Categories and reference types

- `MyType&` **can be initialized only from lvalues**
- `MyType&&` **can be initialized only from rvalues**
- `const MyType&` **be initialized from both**
  - For variable, extends lifetime of rvalue to enclosing block
  - Overload resolution prefers `MyType&&` for rvalues
- **Example: move object when you don't care about original**

```
struct Label {
  std::unique_lock<std::mutex> lk_;
  std::string bigstring_;
  Label(std::unique_lock<std::mutex> lk, std::string b)
    : lk_(std::move(lk)), bigstring_(std::move(b)) {}
};
```

- **Example 2: return large** `std::map` **from function**

Understanding `std::unique_ptr`

- **A `std::unique_ptr` is a pointer that can be moved, not copied**
  - Ensures only one `std::unique_ptr` to any given object
  - Destructor of `std::unique_ptr` deletes object
- **Helps avoid memory leaks**
  - Can't forget to free a `std::unique_ptr`
  - Even properly frees memory when exceptions thrown
- **Beware exceptions in evaluating function arguments**

```
int f(std::unique_ptr<A>, std::unique_ptr<B>);
// Can leak memory if either A::A() or B::B() throws
void g() { f(new A, new B); }
```

- **Solution: `std::make_unique<T>` safely allocates object**

```
f(std::make_unique<A>(), std::make_unique<B>());
```

- **Example: `unique.cc`**

Resource Acquisition Is Initialization (RAII)

- **RAII is a technique to prevent resource leaks**
  - Always tie resources to lifetime of an object
  - Object destructor can then release all resources
- **This is a really good way to avoid unpleasant bugs**
  - Safe to return whenever there is an error
  - Also safely releases resources when you have an exception
- **Slight annoyances:**
  - Constructors can only fail by throwing exceptions
  - Destructors cannot safely throw exceptions
  - You have to give unique names to variables you don't care about

```
if (IntrGuard ig; true) {
    /* do something with interrupts disabled */
}
```

- **Example:** `finally.cc`