EECS 482

Lab 3: RAII, Exceptions

Administrivia

Today: 09/19

Project 1 Due 9/22

Project 2 Released Next Week (after P1 due) & Due 10/15 (in ~3 weeks)

++ Difficulty vs. P1

Start Early!!

Student & Staff Social!

Tuesday, 09/23
7:00 - 9:00 PM
BBBB Atrium

After P1 is Due

Board / card games with the staff & Prof. Chen

+ ping pong table
+ pizza & drinks
+ P1 pizza delivery jokes

Bring any games you would like all of us to play

Sign up on Piazza @149

Agenda

1. Q1: Luxury Box
2. RAII
3. Exceptions
4. Q2: lock_guard Exercise

Q1: Luxury Box

Luxury Box: Q2

B1G

PAC 12

Luxury Box: Q2

Only fans from the same conference can be in the box at once.

Implement using monitors.

```
int active_big10 = 0, active_pac12 = 0; // fans in the box

void big10_wants_to_enter();
void big10_leaves();

void pac12_wants_to_enter();
void pac12_leaves();
```

Let's code it! (write just the pac12 functions)

Starvation

When threads are not able to accomplish their work because other threads have control.

How can we fix this in the context of the luxury box problem?

Key Takeaways from Luxury Box

- There are many factors involved when coding with monitors
- Efficiency is important
  - Ask yourself "how many CVs should I use?"
- Fairness should also be considered
  - Difficult to analyze and not as objective
- Evaluating these tradeoffs is key
  - **There is often not one perfect solution**
- **The most important factor is always correctness**
  - Start with a correct solution, and gradually improve it

Resource Acquisition Is Initialization (RAII)

What is a Resource?

- Can be “acquired” and “released”
- Usually expensive to hold
- Examples:
  - Memory (new, delete)
  - Mutex (lock, unlock)
  - File (open, close)

Variable Lifetime

- Tied to the scope it is defined in
- Constructor is called where the variable is introduced into the scope
- When a scope exits the destructors of all contained variables are called in the *reverse* construction order

```
struct A {
   A() { // Constructor
       std::cout << "A ctor" << std::endl;
   }

   ~A() { // Destructor
       std::cout << "A dtor" << std::endl;
   }
};

int main() { A a; }
```

This will print:
A ctor
A dtor

Note: the lifetime of “a” is bound to the function “main”

RAII

Idea: Use variable lifetimes to manage resources

- Make a class to hold the resource
- Acquire/release the resource with the constructor/destructor

Benefits:

- Resources automatically released when exiting scope
  - function return
  - thrown exception
- No worries about manually releasing

Example where RAII would be useful

```
int do_thing() { // Return 0 on success, -1 on failure
  m.lock();
  if (shared.action_that_might_fail_and_return_neg1() < 0) {
    m.unlock();  // <-------
    return -1;
  }
  if (shared.another_action_that_might_fail() < 0) {
    m.unlock();  // <-------
    return -1;
  }
  m.unlock(); // <-------
  return 0;
}
```

A) **What if you forget to unlock m?? (correctness is jeopardized)**

B) **This is ugly and *bad style***

Solution: Use an RAII class

```
int do_thing() { // Return 0 on success, -1 on failure
  // lock_guard calls lock() in ctor, unlock() in dtor
  lock_guard lock{m};
  if (shared.action_that_might_fail_and_return_neg1() < 0) {
    return -1; // lock exits scope, unlocks m
  }
  if (shared.another_action_that_might_fail() < 0) {
    return -1; // lock exits scope, unlocks m
  }
  if (...) { return -1; }
  return 0; // lock exits scope, unlocks m
}
```

Solution: Use an RAII class

```
class lock_guard {
public:
  // acquire resources in constructor
  lock_guard(mutex &in) : my_mutex{in} {
    my_mutex.lock();
  }
  // release resources in destructor
  ~lock_guard() {
    my_mutex.unlock();
  }
private:
  mutex &my_mutex;
};
```

Smart Pointers

Review: Dynamic Memory

- When we want to control the lifetime of an object, we can use dynamic memory
- To use dynamic memory, we use raw pointers
- **operator new** allocates memory for an object
- **operator delete** destroys the object and frees its memory:

```
class Player { ... };
auto* p = new Player{};
...
delete p;
```

Dynamic Memory - Ownership

- For dynamically-allocated objects, ownership of an object means it is yours to keep or destroy as you see fit.
- In C++, by ownership, we mean not just which code gets to refer to or use the object, but mostly what code is responsible for **deleting** it
- When using raw pointers, we implement ownership in terms of where in the code we place the delete that destroys the object
  - This is bug prone. What are some possible errors?

Smart Pointers

- Just as RAII can help us manage the release of resources, is there something that can help us manage the deallocation of dynamic memory?
- **Smart pointers** - class objects that behave like raw pointers but also manage objects that you create with new so that you don't have to worry about when and whether to delete them
- **Acquire memory in constructor; possible deallocate in destructor**
- In C++, `#include <memory>` to use them

```cpp
template<typename T>
class SmartPtr {
public:
    explicit SmartPtr(T *p_in)
        : ptr{ p_in } {}

    ~SmartPtr() { delete ptr; }

    T &operator*() { return *ptr; }

private:
    T *ptr;
};
```

Core Idea: Pointer Abstraction

- A raw pointer can point to valid or invalid memory
- A raw pointer that points to valid memory can be changed

- Raw pointer issues: Invalid Accesses, Memory Leaks

Smart Pointers are **guaranteed to point to valid memory**, and **will clean up resources** when destroyed.

Why smart pointers?

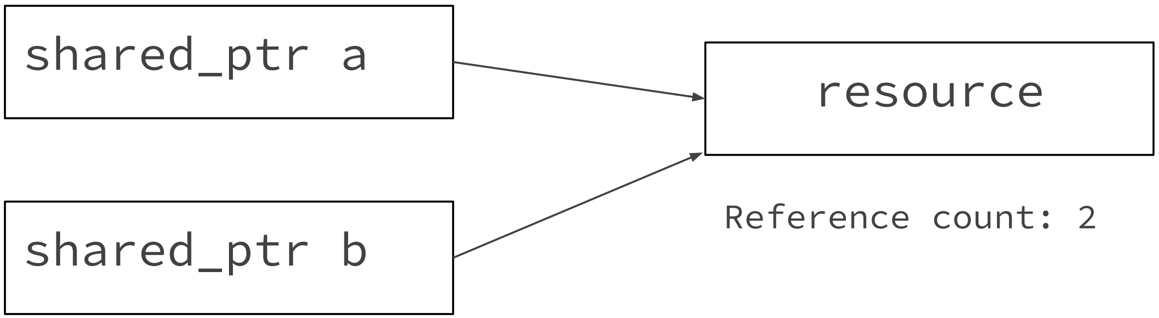
- **Ease of use**: They handle deallocation for you, helping you avoid memory leaks and use-after-free bugs
- **Familiarity with the STL**: They are widely used in C++ codebases and it is be useful to have experience with them
- **Avoiding Bugs**: They can help you avoid bugs by enforcing certain invariants (e.g. unique ownership)

Smart Pointers

- **shared_ptr<T>**
  - Implements **shared ownership**.
  - Any number of shared_ptrs can jointly own an object. When the last shared_ptr pointing to an object is destroyed, the object is **automatically destroyed**.
- **unique_ptr<T>**
  - Implements **unique ownership**.
  - Manages memory like a shared_ptr, but there can only be **one** unique_ptr to a given object at a time.
  - Can't be copied - requires **move semantics**.

shared_ptr<T>

- Multiple shared pointers can point to a resource
- When all shared pointers are destroyed, resource is released
- Maintains a “reference count” for the resource

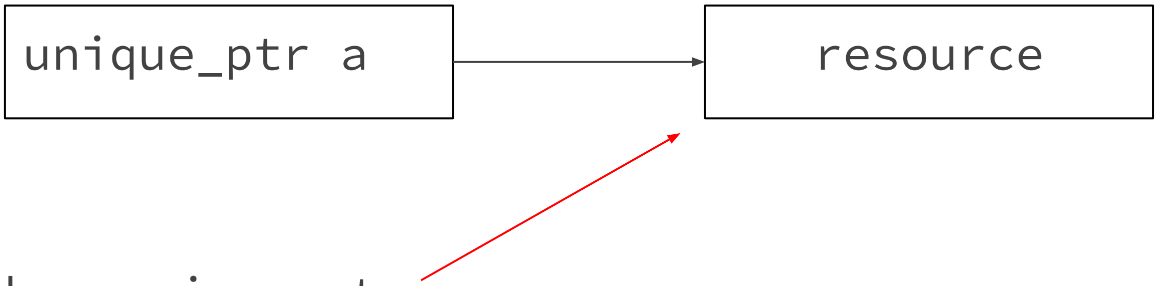


shared_ptr<T> Example

```
{
shared_ptr<int> b;
    {
        shared_ptr<int> a = make_shared(10); //resource allocated
        b = a; //copy of shared pointer
    }
//a is destroyed
}
//b is destroyed
//resource freed
```

unique_ptr<T>

- Ensures only one owner for a given resource
- **Cannot be copied**, ownership must be transferred from one scope to another using std::move()



No other unique ptr

unique_ptr<T> Example

```
{
unique_ptr<int> b;
    {
        unique_ptr<int> a = make_unique(10); //resource allocated
        b = std::move(a); //must move instead of copy
    }
}
//b is destroyed
//resource freed
```

Best Practices

- You can use `.get()` to obtain the raw pointer value of a smart pointer, but try to avoid it– this raw pointer does *not* provide the guarantees that smart pointers do (such as the assurance of no dangling pointers)
- If you are new to smart pointers, start with shared_ptr (easy replacement for raw pointers).

Read:
https://en.cppreference.com/w/cpp/memory/unique_ptr
https://en.cppreference.com/w/cpp/memory/shared_ptr

Converting Code: Raw Pointers

```
---
struct big {  //expensive to copy
    ...
}
big* a = new big();
big* b = a;

delete b; //needed to not leak memory

//a and b still accessible!
*a // segfault or worse
```

Converting Code: Smart Pointers

```
---
struct big {  //expensive to copy
    ...
}
    std::shared_ptr<big> a(new big());
{
    shared_ptr<big> b = a; //copy
}
//impossible to have invalid access
//resource still cleaned up
```

Better interface: make_shared

Exposing the raw pointer is still bad style! (we don’t want to see the **new**. Here’s a better interface:

```
shared_ptr<big> a = std::make_shared<big>();
```

Note: make_shared initializes the object it creates - when might this be a problem?

Solution: Use make_shared_for_overwrite !

Exceptions

Example

```
#include <stdexcept>
#include <iostream>

int main() {
  try {
    int x;
    std::cin >> x;
    if (x == 0) {
      throw std::runtime_error("received 0");
    }
  } catch (std::runtime_error &e) {
    std::cout << e.what() << std::endl; // if runs, prints "received 0"
  }
}
```

Error handling using return value

```
int do_thing() { // Return 0 on success, -1 on failure
  if (action_that_might_fail_and_return_neg1() != 0) {
    return -1;
  }
  if (another_action_that_might_fail() != 0) {
    return -1;
  }
  if (another_action_that_might_not_succeed() != 0) {
    return -1;
  }
  return 0;
}
```

What if we added **another** action that might fail?

Error handling using return value

```
int do_thing() { // Return 0 on success, -1 on failure
  if (action_that_might_fail_and_return_neg1() != 0) {
    return -1;
  } if (another_action_that_might_fail() != 0) {
    return -1;
  } if (another_action_that_might_not_succeed() != 0) {
    return -1;
  } if (yet_another_action_that_could_fail() != 0) {
    return -1;
  }
  return 0;
}
```

What if these **threw exceptions** on failure, instead of returning -1?

Error handling using exceptions

```
void do_thing() {
  try {
    action_that_might_fail_and_throw_exception();
    another_action_that_might_fail();
    another_action_that_might_not_succeed();
  } catch (exception_type_1 &e) {
    error_handler_1(e);
  } catch (exception_type_2 &e) {
    error_handler_2(e);
  }
}
```

What if we added **another** action that might fail?

Error handling using exceptions

```
void do_thing() {
  try {
    action_that_might_fail_and_throw_exception();
    another_action_that_might_fail();
    another_action_that_might_not_succeed();
    yet_another_action_that_might_not_succeed();
  } catch (exception_type_1 &e) {
    error_handler_1(e);
  } catch (exception_type_2 &e) {
    error_handler_2(e);
  }
}
```

Exceptions **without** RAII (bad)

```
mutex m;

void do_thing() try {
  m.lock();
  action_that_might_fail_and_return_neg1();
  another_action_that_might_fail();
  another_action_that_might_not_succeed();
  yet_another_action_that_could_fail();
  m.unlock();
} catch (exception_type &e) {
  m.unlock();
  error_handler(e);
}
```

pairing 1 lock with 2 unlocks :(

Exceptions with RAII (good)

```
mutex m;

void do_thing() try {
  mutex_guard guard{m};
  action_that_might_fail_and_return_neg1();
  another_action_that_might_fail();
  another_action_that_might_not_succeed();
  yet_another_action_that_could_fail();
} catch (exception_type &e) {
  error_handler(e);
}
```

```
if exception is
thrown, guard is
destructed :)
```

Q2: Smart Pointer and Exceptions Exercise

Demo: Dumb Pointers (git clone from GitHub)

```cpp
#include <iostream>
#include <stdexcept>
#include <vector>

constexpr unsigned int VEC_SIZE = 1000000;

struct Resource {
    Resource(const std::string& name)
        : name(name), leaked(VEC_SIZE, 0) {
        std::cout << "Acquired: " << this->name << "\n";
    }
    ~Resource() {
        std::cout << "Released: " << name << "\n";
    }
    void use() {
        std::cout << "Using: " << name << "\n";
    }
    std::string name;

    std::vector<int> leaked;
};
```

```cpp
void riskyOperation() {
    throw std::runtime_error("riskyOperation failed!");
}

int main() {
    while (true) {
        try {
            Resource* res = new Resource("LeakyRes");

            // risky call throws before delete can run
            riskyOperation();

            // never reached
            res->use();
            delete res;
        } catch (const std::exception& e) {
            std::cerr << "Caught exception: " << e.what() << "\n";
            // res is never deleted here - memory leak every iteration
        }
    }
}
```

Excercise: Smart Pointers

Git clone the smart pointers demo from the GitHub

```cpp
// TODO: Implement sharedPtrDemo()
// - Create a std::shared_ptr<Resource>
// - Copy it into more shared_ptrs
// - Print use_count() as copies are made and destroyed
// - Observe reference count changes
void sharedPtrDemo() {
    // TODO
}
```

```cpp
// TODO: Implement uniquePtrDemo()
// - Create a std::unique_ptr<Resource>
// - Move it into another unique_ptr
// - Verify the original is empty after move
// - Call use() on the new owner
void uniquePtrDemo() {
    // TODO
}
```

Questions?