Smart Pointers

Resource Acquisition Is Initialization (RAII)

- A poorly-named paradigm that means: wrap release/delete code in destructors

```
struct handle {
  HANDLE _h;
  handle(HANDLE h) : _h(h) {}
  ~handle() { CloseHandle(_h); }
  operator HANDLE() { return _h; }
};

handle file { CreateFile("temp.txt", ...) };
WriteFile(file, ...);
// handle is closed when `file` goes out of scope
```

Example: Mutex Locking

```
struct mutex_lock {
  handle& _mutex;
  mutex_lock(handle& m) : _mutex(m) {
    WaitForSingleObject(_mutex, INFINITE);
  }
  ~mutex_lock() { ReleaseMutex(_mutex); }
};

std::vector<int> primes;
handle mutex { CreateMutex(...) };
Concurrency::parallel_for(2, 100000, [&](int n) {
  if (is_prime(n)) {
    mutex_lock lock(mutex);
    primes.push_back(n);
  }
});
```

Example: Generic RAII Helper

```
template <typename Resource, typename Deleter>
struct raii {
  Resource res;
  Deleter del;
  raii(Resource r, Deleter d) : res(r), del(d) {}
  ~raii() { del(res); }
  operator Resource() { return res; }
};
template <typename Resource, typename Deleter>
auto make_raii(Resource r, Deleter d) {
  return raii<Resource, Deleter>(r, d);
}
```

```
auto helper = make_raii(CreateFile(...), CloseHandle);
```

Smart Pointers

- STL in C++ 11 has three smart pointer types that provide RAII for pointers
- You should no longer have to write `new` and `delete` (in most cases)

`unique_ptr<T>`

- Moveable RAII wrapper that calls `delete`
- Non-copyable
- Pointer-sized

`shared_ptr<T>`

- Reference-counted smart pointer
- Copyable
- Larger overhead

`weak_ptr<T>`

- Helper for breaking reference cycles with `shared_ptr`

Your Go-To Smart Pointer Type: `unique_ptr<T>`

- Moveable: can be returned and passed by value
- Does not allow shared ownership: there is exactly one owner at any time

```
std::unique_ptr<LargeObject> initialize() {
  auto p = std::unique_ptr<LargeObject>(new LargeObject(...));
  p->additional_initialization();
  return p;
}
struct owner {
  unique_ptr<LargeObject> large;
  owner(unique_ptr<LargeObject> p) : large(std::move(p)) {}
};
```

Usage:

```
auto largey = initialize();
owner o { std::move(largey) };
```

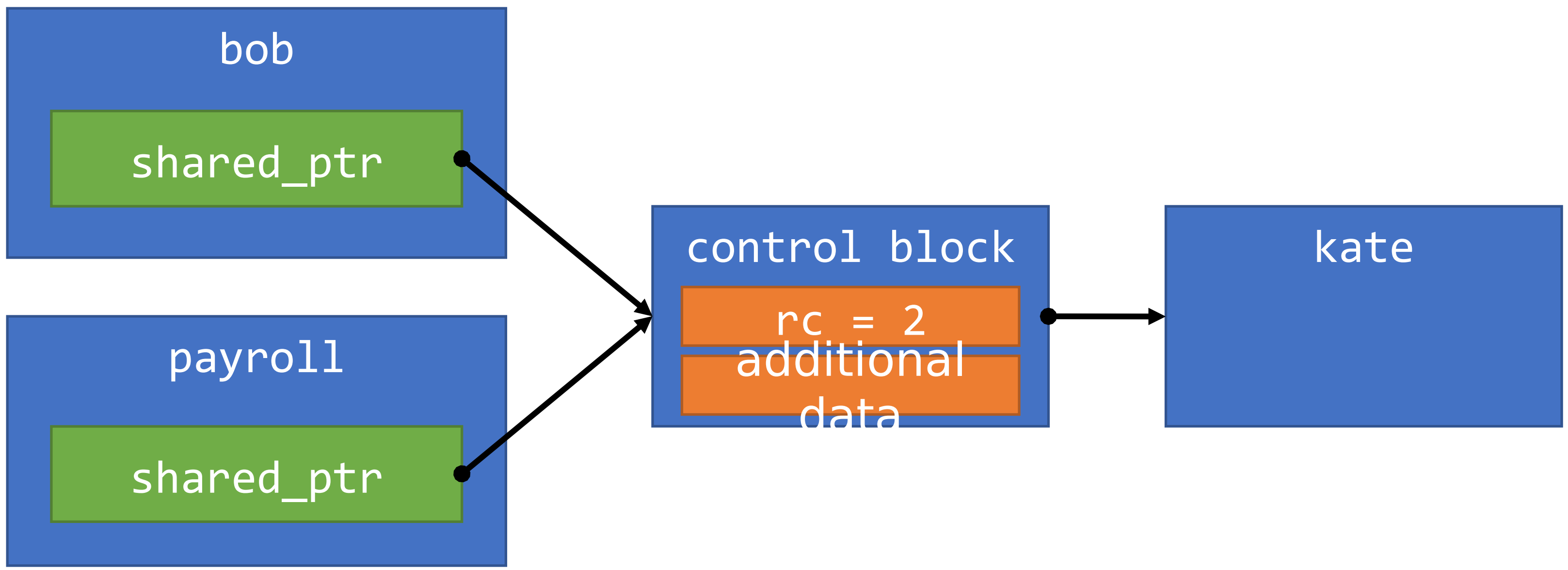
Shared Ownership: `shared_ptr<T>`

- Copyable, maintains a reference count
- Deletion occurs when reference count reaches 0
- The reference count is updated *atomically*

```
struct manager {
  std::vector<std::shared_ptr<employee>> employees;
} bob;
struct payroll_system {
  std::map<std::shared_pointer<employee>, double> salaries;
} payroll;
std::shared_ptr<employee> kate(new employee(...));
payroll.salaries[kate] = 6000.0;
bob.employees.push_back(kate);
```

`shared_ptr<T>` Internals

- Internally, `shared_ptr<T>` contains a pointer to a shared control block that maintains the reference count and points to the target

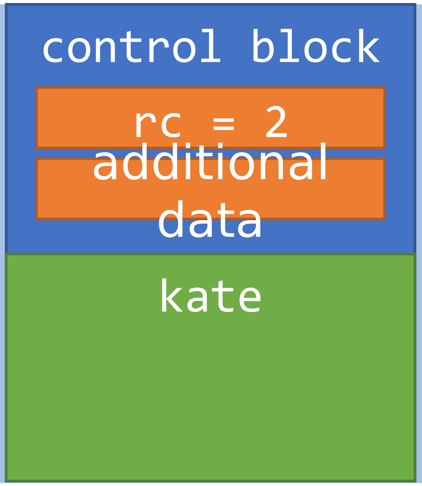


make_shared and make_unique

- make_shared is a helper that:
  - Makes it easier to specify the pointer type
  - Allocates the control block alongside the object
  - One heap allocation means smaller overhead
- make_unique is a similar helper *(in C++ 14)*

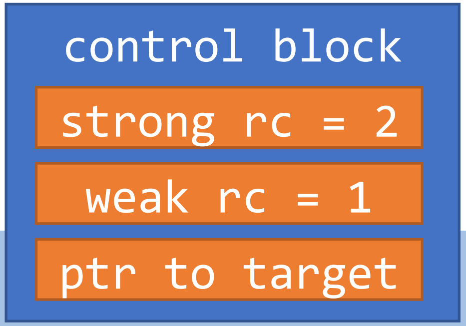
```
auto kate = std::make_shared<employee>(
  "Kate", departments::research);

auto largey = std::make_unique<LargeObject>(
  ...);
```



Shared Pointers, Reference Cycles, and `weak_ptr<T>`

- Shared pointers can create indestructible reference cycles: A -> B, B -> A
- `weak_ptr<T>` helps resolve cycles




```
struct manager {
  std::vector<std::shared_ptr<employee>> employees;
};
struct employee {
  std::weak_ptr<manager> manager;
  void request_leave() {
    if (auto m = manager.lock()) { m->request_leave(*this); }
  } // `m` is std::shared_ptr<manager>, might be null
};
```

When To Use What?

- Use `unique_ptr` to express *ownership*
- If you need *shared* ownership, use `shared_ptr`
- To break reference cycles, use `weak_ptr`

- What about raw pointers and references?
- **Raw pointers and references are perfectly fine for *non-owning* pointers when the object's lifetime is known to exceed the pointer's lifetime**

Examples Where Raw Pointers and References Are Just Fine

```
void performance_review(employee& e) {
  if (e.performance > average) {
    e.salary *= 1.08;
  }
}

int do_reviews(manager* specific_manager = nullptr) {
  if (specific_manager) {
    // ... Perform a review for each employee of that manager
  } else {
    // ... Perform a review for all the employees on payroll
  }
}
```

Bad Examples of Smart Pointer Use

```
void assign_task(std::shared_ptr<employee>& emp, task t) {
  if (emp->tasks.size() < 10) {
    emp->tasks.push_back(t);
  }
}
```

Bad! No reason to force the caller to pass a `shared_ptr`. The function doesn't retain the employee object, so a plain reference is just fine.

```
void assign_task(std::unique_ptr<employee> emp, task t) {
  // ... Same as before
}
```

Even worse! The caller can only pass in a temporary, which is destroyed when the function returns. Clearly bad.

Passing unique_ptr<T> by Value

- Means only one thing: the callee now takes ownership of the object, and is responsible for its lifetime

```
struct window {
  std::unique_ptr<menu> main_menu;
  void set_menu(std::unique_ptr<menu> m)
    { main_menu(std::move(m)); }
};

auto main = std::make_unique<menu>();
main->add(command { "File", { "Open...", "Exit" } });
app::main_window().set_menu(std::move(main));
```

Passing `shared_ptr<T>` by Value

- Means only one thing: the callee will share ownership on the object
- The caller and callee are both responsible for it

```
void manager::add_employee(shared_ptr<employee> emp) {
  // ... Stores the pointer for later use
}

void payroll::add_employee(shared_ptr<employee> emp) {
  manager& mgr = next_available_manager();
  mgr.add_employee(emp); // can also std::move(emp) here
}
```

Passing Smart Pointers by Reference

- Only pass smart pointers by reference if you intend the callee to manipulate the pointer's *target*
  - This should be rare in practice
- Do not return smart pointers by reference; if you really intend to return a reference, return a reference to the object itself

Custom Deleters

- When the default `delete` isn't enough, you can use a custom deleter function
  - Works with both `std::unique_ptr` and `std::shared_ptr`

```
std::shared_ptr<Brush> file {
  OpenBrush(...),
  [](Brush* b) { DestroyBrush(b); delete b; }
};
```

unique_ptr performance

- Dereference, destruction and even move is the same performance as in native pointer.

```
smart_ptr<int>(...); *p = 42;  // same as p->operator*() = 42; compile optimization
  will be *(p->ptr)= 42

int* p = new int ; *p = 42;

smart_ptr<int> p p(new int());
smart_ptr<int> p2 = std::move(p) // swap - same as in native

int* p = new int();
int* p2 = p; p = null; // this is what the move do

p.ptr = nullptr; // compiler save the delete, delete null ignored
delete p.ptr;
```

The Pimpl Idiom, Revisited

- The *pimpl idiom* is typically used to reduce dependencies and shorten compilation times
- Hides implementation details behind a pointer

```
class widget {
public:
  // ... the class’ public interface
private:
  struct impl;
  impl* pimpl_;
};
// All the interface methods touch pimpl_ to access state
```

The Pimpl Pointer Can Be Smart

- Use `std::unique_ptr<>` for the pimpl pointer: easy and supports trivial move

```
class widget {
public: // ... as before
private:
  struct impl;
  std::unique_ptr<impl> pimpl_;
};

// In implementation file:
widget::widget() : pimpl_(std::make_unique<impl>()) {}
widget::widget(widget&&) = default;    // same with op=
// ... and so on
```

Modernizing C++ Code
Getting Rid of Raw Pointers

```
widget* next_widget();
void read_widget() {
  if ((widget* pw = next_widget()) != NULL) {
    pw->process_widget();
    delete pw;
  }
}
unique_ptr<widget> next_widget();
void read_widget() {
  if (auto pw = next_widget()) {
    pw->process_widget();
  }
}
```