COMP6771
Advanced C++ Programming

5.2 – Resource Management

In this Lecture

Why?

- Performance & Control
  - with great power comes great responsibility.
- Leak freedom in C++.
- C++ is not garbage collected.
- While we have ignored heap resources (malloc/free) to date, they are a critical part of many libraries, and we need to understand best practices around usage.

What?

- Resource Management
- RAII
- Smart Pointers

Revision: Objects

- What is an object in C++?
  - An object is a region of memory associated with a type.
  - Unlike some other languages (Java), basic types such as `int` and `bool` are objects.
- For the most part, C++ objects are designed to be intuitive to use.
- Special things can we do with objects:
  - Create them;
  - Destroy them;
  - Copy them; and
  - Move them.

Long Lifetimes

- There are 3 ways you can try and make an object in C++ have a lifetime that outlives the scope it was defined in:
  - Returning it out of a function via copy (can have limitations).
  - Returning it out of a function via references (leads to dangling references).
  - Returning it out of a function as a heap resource (explicit memory management).

```cpp
struct point2i { int x; int y;}

// This function returns a new object,
// Not a reference to the object
point2i make_point_copy(int x, int y) {
  point2i p = {x, y};
  return p;
}

// Returns a reference to a stack-local variable.
// Program has undefined behaviour if used.
point2i &make_point_dangling(int x, int y) {
  point2i p = {x, y};
  return p;
}

// Returns a reference to a heap-allocated variable.
// Now we must manage the memory explicitly.
point2i &make_point_heap(int x, int y) {
  point2i *ptr = new point2i{x, y};
  return *ptr;
}
```

Long Lifetimes & References

- We need to be very careful when returning references.
- *The object* **must** *always outlive the reference.*
- Using dangling references is undefined behaviour!
- **Moral of the story**: Do not return references to stack-local variables.
- Unless we copy, for objects we create INSIDE a function, we must heap-allocate them!

```
auto okay(int &i) -> int& {
  return i;
}

// why is this ok?
// answer: lifetime extension
auto okay(int &i) -> const int& {
  return i;
}

auto not_okay(int i) -> int& {
  return i;
}

auto definitely_not_okay() -> int& {
  auto i = 0;
  return i;
}
```

`new` & `delete`

- Non-global objects are either stored on the *stack or heap*.
- In general, you've created most objects on the stack (without realising!)
- We can allocate heap objects via new and free them via `delete`.
  - new and `delete` call the constructors/destructors of what they are creating.

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

int main() {
  int *a = new int{4};
  auto *b = new std::vector<int>{1,2,3};

  std::cout << *a << "\n";
  std::cout << (*b)[0] << "\n";

  delete a; // frees a
  delete b; // frees b

  return EXIT_SUCCESS;
}
```

A Heap of Memory

- Why do we need heap resources?
  - Heap objects can outlive the scopes they were created in.
  - More useful in contexts where we need explicit control of ongoing memory size e.g. for vector.
  - Stack has limited space on it for storage, heap is much larger.
  - No matter how much we try, it is very difficult to use all of dynamically allocated memory.
    - Primary exception is on embedded systems.

```
#include <iostream>
#include <vector>

int *make_int(int i) {
  int *a = new int{i};
  return a;
}

int main() {
  int *a = make_int(6771);

  // a was defined in a scope that
  // no longer exists
  std::cout << *a << "\n";

  delete a;
}
```

Heap Allocation is Not Free

- Non-trivial programs almost always use the heap in some way.
  - All of our examples using `std::vector` have been using the heap secretly as well.
- Heap allocation is expensive compared to stack allocation.
  - Should be avoided in performance-critical code.
- C++ has value semantics.
  - Is it possible to avoid the cost associated with deep-copying a heap allocated object?
- Answer: Move Semantics!

Move Semantics (C++11 onwards)

- Move semantics are a way for class-types to “steal” the data of a temporary value and reuse it.
- Makes use of a language feature called **rvalue references.**
- Let’s review lvalues and rvalues.
    - Also called ”value categories”.

```
#include <vector>

std::vector<int> make_vec() {
  return std::vector<int>{1, 2, 3, 4, 5};
}

int main() {
   // here, v is default initialised.
   std::vector<int> v;

   // The vector from make_vec() is a temporary.
   // Rather than make v allocate a vector that
   // can fit the contents of make_vec(), why not
   // just steal make_vec()’s data directly?
   // It is going out of scope, so nothing will be
   // affected by this.
   v = make_vec(); // make_vec() is “moved” into v.
}
```

UNSW SYDNEY

lvalue vs rvalue

- **lvalue**: An expression that is an object reference.
  - E.g. named variables.
  - You can always take the address of an lvalue.
- **rvalue**: Expression that is not an lvalue
  - E.g.. Object literals, return result of functions.
  - rvalues are temporary and short lived, while lvalues live a longer life since they exist as variables.

```
int &f();

int main() {
  // 5 is rvalue, i is lvalue
  int i = 5;

  // j is lvalue, i is lvalue
  int j = i;

  // 4 + i produces rvalue then stored in lvalue k
  int k = 4 + i;

  // error: cannot assign to rvalues.
  6 = k;

  // taking address of an lvalue
  int* y = &k;

  // error: cannot take address of an rvalue.
  int* y = &6771;

  // OK: f() returns an lvalue reference
  f() = 3;
}
```

UNSW SYDNEY

lvalue References

- There are multiple types of references:
  - lvalue references look like `T&`.
  - lvalue references to const look like `const T&`.
- Once the lvalue reference goes out of scope, the original lvalue is still usable.
- Constant lvalue references exhibit **lifetime extension**.
  - Any temporaries bound to a `const T&` act like an lvalue.
  - The temporary gets stored in memory and is referred to via the reference

```
// regular lvalue
int y = 10;

// OK: binding lvalue to an lvalue reference.
int &yref = y;

// OK: binding lvalue to a const lvalue
reference.
const int &cref = y;

// !!: cannot bind const lvalue to a non-const
ref
int &ref = static_cast<const int>(y);

// !!: rvalues only bind to const lvalue
references
const int &extended = 10;
```

rvalue References

- rvalue references look like T&&.
- rvalue references extend the lifespan of the temporary object to which they are assigned.
- Non-const rvalue references allow you to modify the rvalue.
- An rvalue reference formal parameter means that the value was disposable from the caller of the function.
    - If the callee modified value, who would notice or care?
    - The caller has promised that it won't be used anymore
    - An rvalue reference parameter is an lvalue inside the function.

```
#include <iostream>

// Declaring an rvalue reference
int &&rref = 20;

void inner(std::string &&value) {
  value[0] = 'H';
  std::cout << value << '\n';
}

void outer(std::string &&value) {
  inner(value); // This call fails? Why?
  std::cout << value << '\n';
}

int main() {
  outer("hello"); // This call succeeds.
  auto s = std::string("hello");

  // This call fails because s is an lvalue.
  inner(s);
}
```

std::move

Uses of rvalue references:

- They are used in working with the move constructor and move assignment special member functions.
- cannot bind non-const lvalue reference of type ‘**int&**‘ to an rvalue of type ‘int’.
- cannot bind rvalue references of type ‘**int&&**‘ to lvalue of type ‘int’.
- A library function that converts an lvalue to an rvalue so that a "move constructor" (similar to copy constructor) can use it.
- This says "I don't care about this anymore”.
- All this does is allow the compiler to use rvalue reference overloads.

```
// std::move looks something like this.
T&& move(T& value) {
  return static_cast<T&&>(value);
}

void inner(std::string&& value) {
  value[0] = 'H';
  std::cout << value << '\n';
}

void outer(std::string &&value) {
  // this now works!
  inner(std::move(value));

  // Value is now in a valid but unspecified state.
  // Although this isn’t an error, this is bad code.
  // Only access moved variables to remake them.
  std::cout << value << '\n';
}

int main() {
  outer("hello"); // This works fine.
  auto s = std::string("hello");
  inner(s); // This fails because i is an lvalue.
}
```

RAII: Resource Acquisition is Initialisation

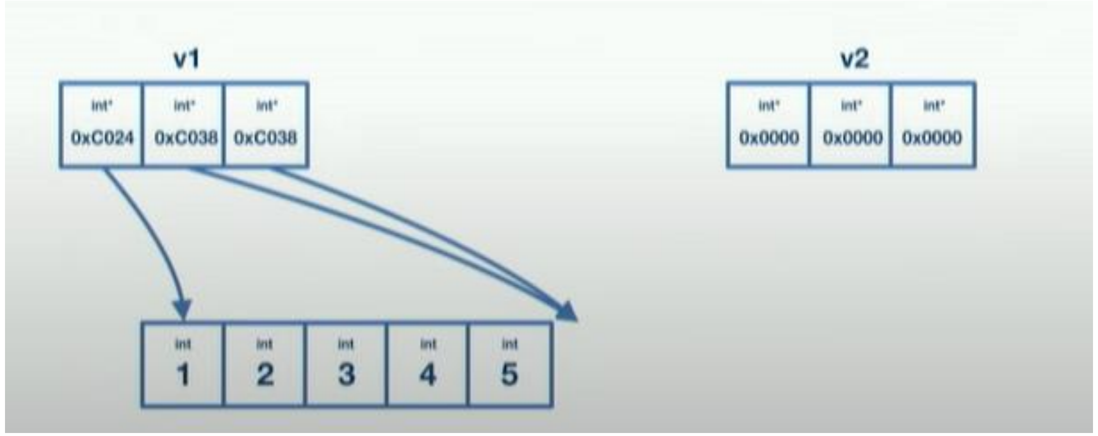
- A resource is anything that is scarce and must be managed.
  - E.g., Memory, locks, file descriptors, time.
- RAII is a concept where we encapsulate resources inside objects.
  - Acquire the resource in the constructor.
  - Release the resource in the destructor.
  - Deep-copy the resource if it makes sense.
  - Transfer ownership if it makes sense.
  - Resource is always released at a known point in the program, which you can control.
- Every resource should be owned by either:
  - Another resource (e.g. smart pointer, data member).
  - Named resource on the stack.
  - A nameless temporary variable.

RAII-aware Classes

- When writing a class, if we can't default all of our operators (preferred), we should consider the **Rule of 5:**
  1. Destructor
  2. Move assignment
  3. Move constructor
  4. Copy constructor
  5. Copy assignment
- The presence or absence of these 5 operations is critical in managing resources.
- If you define one of these special member functions, you should explicitly define or default all of them.

Implementing an RAII Class

- Let's try implementing our own version of `std::vector`.
- It's going to have to manage some heap memory, so it should look something like this.




```
class vec {
public:
  vec(int n)
  : ptr_{new double[n]}, size_{n}, cap_{n} {}

  vec(const vec &other);
  vec(vec &&other);

  vec &operator=(const vec &other);
  vec &operator=(vec &&other) noexcept;

  ~vec();

  // other implementation...
private:
  double *ptr_;
  int size_;
  int cap_;
};
```

RAII Class: Destructor

- What happens when v goes out of scope?
  - Destructors are called on each member…
  - But destructing a pointer type does nothing!
- As it stands, this will result in a memory leak.
  - How do we fix?
  - Use the destructor!

```
class vec {
public:
 // other implementation...
  ~vec() noexcept /* optional noexcept specifier */ {
    delete[] ptr_;
  }
private:
  double *ptr_;
  int size_;
  int cap_;
};
int main() {
  auto v = vec{4};
} // went out of scope...problem?
```

RAII Class: Move Constructor

- Move constructor should be `noexcept`.
- Can use `std::exchange` to exchange the important data members from the temporary.
- Often not much more complex than this.

```
class vec {
public:
 // other implementation...
 vec(vec &&other) noexcept
 : data_{std::exchange(other.data_, nullptr)},
   size_{std::exchange(other.size_, 0)},
   cap_{std::exchange(other.cap_, 0)} {}

private:
  double *ptr_;
  int size_;
  int cap_;
};
```

RAII Class: Move Assignment

- Like the Move Constructor, but the destination has already been constructed.
- The easiest way to write a move assignment is generally to do member-wise swaps, then clean up the original object.
  - Doing so may mean some redundant code, but it means you don't need to deal with mixed state between objects.

```
class vec {
vec &operator=(vec &&other) noexcept {
   if (this != &other) {
     std::swap(data_, other.data_);
     std::swap(size_, other.size_);
     std::swap(cap_, other.cap_);

     delete[] other.data_;
     other.data_ = nullptr;
     other.size_ = 0;
     other.capacity = 0;
   }
 }
 // other implementation...
};
```

Moving Objects

- Always declare your moves as `noexcept` (why?)
  - Failing to do so *might* make your code slower.
- Unless otherwise specified, objects that have been moved from are in a valid but unspecified state.
- Moving is an optimisation on copying
  - The only difference is that when moving, the moved-from object is mutable.
  - Not all types can take advantage of this:
    - If moving an `int`, mutating the moved-from `int` is extra work
    - If moving a vector, mutating the moved-from vector potentially saves a lot of work.
- Moved from objects must be placed in a valid state.
  - Moved-from containers usually contain the default-constructed value.
  - Moved-from types that are cheap to copy are usually unmodified.
  - Although this is the only requirement, individual types may add their own constraints.
- Compiler-generated move constructor / assignment performs member-wise moves.

RAII Class: Copy Constructor

- What does the default synthesised copy constructor do?
  - Member-wise copy.
- What are the consequences?
  - Any modification to our `vec` instance will also change the original.
  - Likely lead to a double-free.
- How can we fix this?
  - Custom copy constructor that does a deep copy!
- Copy constructors are **rarely** `noexcept` (why?).

```
class vec {
public:
 // other implementation...
 vec(const vec &o)
 : data_{new double[o.size_]},
   size_{o.size_},
   cap_{o.cap_} {
  std::copy(o.data_, o.data_ + o.size_, data_);
}

private:
  double *ptr_;
  int size_;
  int cap_;
};
```

RAII Class: Copy Assignment

- Assignment is the same as construction, except there is already a constructed object in your destination.
- You need to clean up the destination first.
- The copy-and-swap idiom makes this trivial.

```
class vec {
public:
 // other implementation...
  vec& operator=(const vec &other) {
  vec(other).swap(*this);
  return *this;
}
private:
  void swap(vec &other) {
    std::swap(data_, other.data_);
    std::swap(size_, other.size_);
    std::swap(capacity_, other.capacity_);
  }
};
```

Explicitly Deleted Copy & Moves

- We may not want a type to be copyable / moveable.
  - Non-copyable types implement "unique ownership" semantics.
    - Only this object owns this resource but can transfer ownership.
  - Non-moveable types are rare.
    - Best example is std::scoped_lock
- If we want to delete them, we can declare `fn() = delete;`

```
struct T {
  T(const T&) = delete;
  T(T&&) = delete;

  T& operator=(const T&) = delete;
  T& operator=(T&&) = delete;
};
```

Implicitly Deleted Copies & Moves

- Under certain conditions, the compiler will not generate the copy and move member functions.
- The implicitly defined copy constructor calls the copy constructor member-wise.
  - If one of its members doesn't have a copy constructor, the compiler can't generate one for you.
  - Same applies for copy assignment, move constructor, and move assignment.
- Under certain conditions, the compiler will not automatically generate copy / move assignment / constructors.
  - If you have manually defined a destructor, the move constructor isn't generated.
  - If you have manually defined a move constructor, the copy constructor isn’t generated.
- If you define one of the Five, you should explicitly delete, default, or define all of the Five.
  - If the default behaviour isn't sufficient for one of them, it likely isn't sufficient for others
  - Explicitly doing this tells the reader of your code that you have carefully considered this
  - This also means you don't need to remember all of the rules about "if I write X, then is Y generated?"

RAII Case Study: Smart Pointers

- Introduced in C++11.
- Ways of wrapping unnamed (i.e., raw pointer) heap objects in named stack objects so that object lifetimes can be managed much easier.
- Supports automatic memory management
  - allocate/deallocate according to RAII.
- Usually two ways of approaching problems:
  - unique_ptr + raw pointers
  - shared_ptr + weak_ptr

Type	**Implied Ownership**
`T*`	None
`std::unique_ptr<T>`	Sole ownership
`std::shared_ptr<T>`	Shared ownership
`std::weak_ptr<T>`	None

std::unique_ptr

- When the unique pointer is destructed, the underlying object is too.
- Slightly more overhead than raw pointers.
- Can be parameterized with a custom deleter
  - default deleter uses `delete` .
- Array version exists:
  - Use `std::unique_ptr<T[]>` instead.

```
#include <memory>
#include <iostream>

int main() {
  auto up1 = std::unique_ptr<int>{new int{42}};

  // error: no copy constructor
  auto up2 = up1;

  std::unique_ptr<int> up3;
  // no copy assignment
  up3 = up2;

  up3.reset(up1.release()); // OK
  auto up4 = std::move(up3); // OK

  std::cout << up4.get() << "\n";
  std::cout << *up4 << "\n";
  std::cout << *up1 << "\n";
} // dynamically allocated int freed here.
```

Raw (Observer) Pointers

- Used to “observe” a `unique_ptr`.
- This is an appropriate use of raw pointers in C++.
- Once the original pointer is destructed, you must ensure you don't access the raw pointers (no checks exist).
- These observers do not have ownership of the pointer.
- Also note the use of `nullptr` in C++ instead of `NULL`.

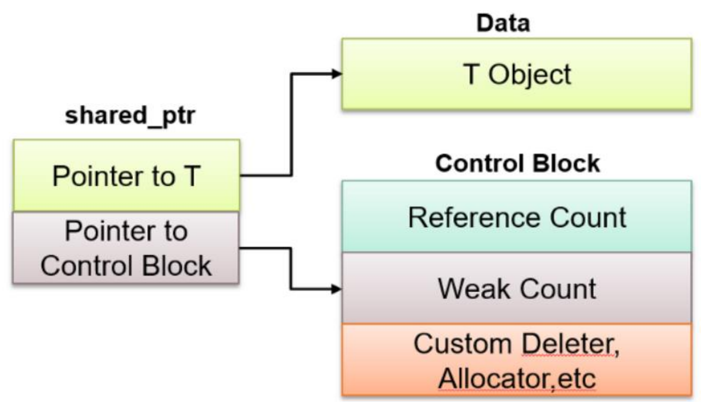
```
#include <memory>
#include <iostream>

int main() {
  auto up1 = std::unique_ptr<int>{new int{5}};
  auto op1 = up1.get();
  *op1 = 6;

  std::cout << *op1 << "\n"; // prints 6
  std::cout << *up1 << "\n"; // also prints 6

  // free the managed int
  up1.reset(nullptr);

  // undefined behaviour
  std::cout << *op1 << "\n";
}
```

std::shared_ptr

- Several objects share ownership of the underlying resource.
- Uses reference counting to avoid premature freeing.
- When a shared pointer is destructed, if it is the only shared pointer left pointing at the object, then the object is destroyed.
- Observers are `std::weak_ptrs` instead of raw pointers.
- Like `unique_ptr`, an array version is also available.

```
#include <iostream>
#include <memory>

auto main() -> in {
  auto x = std::make_shared<int>(5);

  std::cout << "use count: " << x.use_count() << "\n";
  std::cout << "value: " << *x << "\n";

  x.reset(); // Memory still exists, due to y.
  std::cout << "use count: " << y.use_count() << "\n";
  std::cout << "value: " << *y << "\n";

  // Deletes the memory, since no one else owns the memory
  y.reset();

  std::cout << "use count: " << x.use_count() << "\n";

  std::cout << "value: " << *y << "\n";
}
```

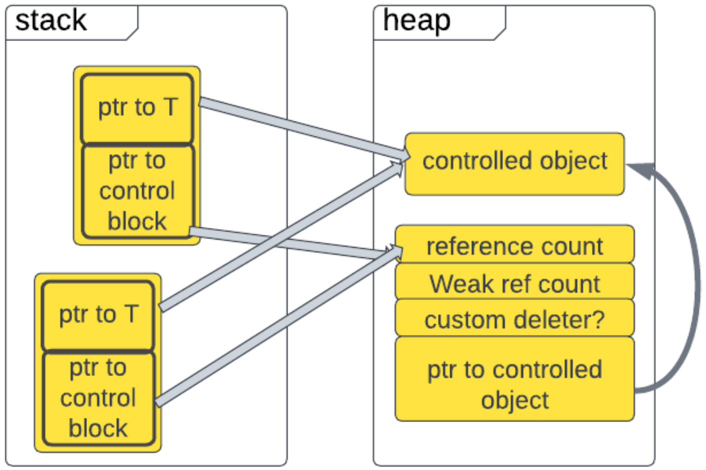
std::shared_ptr Control Block

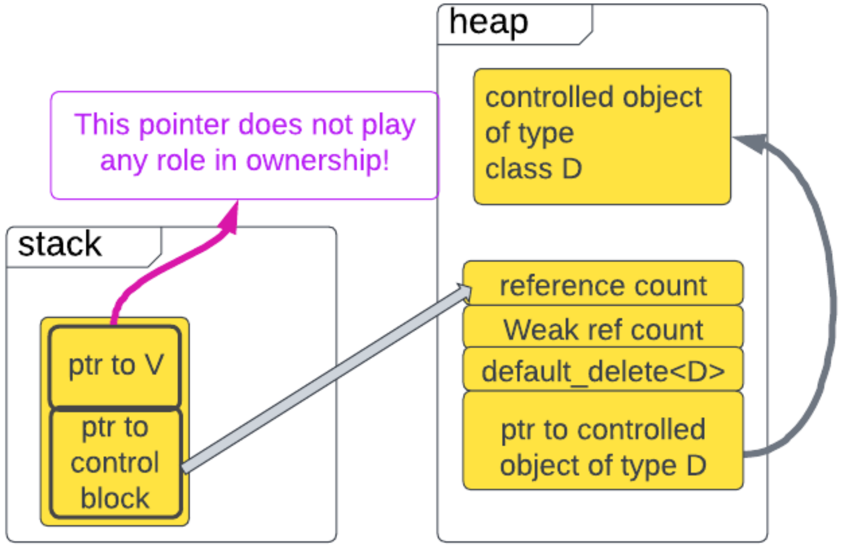


`shared_ptr`, unlike `unique_ptr`, places a layer of indirection between the physical heap-allocated object and the notion of ownership.

`shared_ptr` instances are essentially participating in ref-counted ownership of the control block.

The control block itself is the sole arbiter of what it means to "delete the controlled object."





std::weak_ptr

- Weak pointers are used with share pointers when:
  - You don't want to add to the reference count.
  - You want to be able to check if the underlying data is still valid before using it.
  - Break a circular dependency.
- Must be converted to a `shared_ptr` with the `lock()` method before it can be used.

```
#include <iostream>
#include <memory>

auto main() -> int {
  auto x = std::make_shared<int>(1);

  // x owns the memory
  auto wp = std::weak_ptr<int>(x);

  // now y is a shared_ptr and can be used.
  auto y = wp.lock();

  if (y != nullptr) {
    // x and y own the memory.
    // Do something with y.
    std::cout << "Attempt 1: " << *y << '\n’;
  }
}
```

When to Use Which Pointer

Unique Pointer vs Shared Pointer

- You almost always want a `unique_pointer` over a `shared_pointer`
- Use a `shared_pointer` if either:
- An object has multiple owners, and you don't know which one will stay around the longest; or
- You need temporary ownership
  - This is very rare.
  - Outside scope of this course.

"Leak freedom in C++" poster

Strategy	Natural examples	Cost	Rough frequency
1. Prefer scoped lifetime by default (locals, members)	Local and member objects – directly owned	Zero: Tied directly to another lifetime	O(80%) of objects
2. Else prefer make_unique & unique_ptr or a container, if the object must have its own lifetime (i.e., heap) and ownership can be unique w/o owning cycles	Implementations of trees, lists	Same as new/delete & malloc/free **Automates** simple heap use in a library	O(20%) of objects
3. Else prefer make_shared & shared_ptr, if the object must have its own lifetime (i.e., heap) and shared ownership w/o owning cycles	Node-based DAGs, incl. trees that share out references	Same as manual reference counting (RC) **Automates** shared object use in a library	

Don't use owning raw *'s == don't use explicit *delete*

Don't create ownership cycles across modules by owning "upward" (violates layering)

Use *weak_ptr* to break cycles

Function Signature	Ownership Semantic
`func(value)`	▪ Is an independent owner of the resource ▪ Deletes the resource automatically at the end of `func`
`func(pointer*)`	▪ Borrows the resource ▪ The resource could be empty ▪ Must not delete the resource
`func(reference&)`	▪ Borrows the resource ▪ The resource could not be empty ▪ Must not delete the resource
`func(std::unique_ptr)`	▪ Is an independent owner of the resource ▪ Deletes the resource automatically at the end of `func`
`func(shared_ptr)`	▪ Is a shared owner of the resource ▪ May delete the resource at the end of `func`

Smart Pointer Factory Functions

- `make_shared()` and `make_unique()` wrap raw `new`, just as `~shared_ptr()` and `~unique_ptr()` wrap raw `delete`.
  - Pass the arguments you would have passed to the underlying object's constructor to these functions.
- Never touch raw pointers with hands, and then never need to worry about leaking them.
- `make_unique()` prevents the *unspecified-evaluation-order* leak bug triggered by expressions like:
  - `foo(unique_ptr<T>(new T()), unique_ptr<U>(new U()));`
  - Above, if either the constructor of `T` or `U` throw and the other object has already been allocated, the destructor of `unique_ptr` won't be called.
- `make_shared()` is faster than using `shared_ptr`'s constructors.
  - Able to allocate the managed object *and* control block in a single allocation rather than two.

Resource Safety

To ensure resource safety in C++, we always attach the lifetime of one
object to that of something else.

- **Named objects**:
  - A variable in a function is tied to its scope.
  - A data member is tied to the lifetime of the class instance.
  - An element in a `std::vector` is tied to the lifetime of the vector.
- **Unnamed objects:**
  - A heap object should be tied to the lifetime of whatever object created it.
    - Ideally this should be a smart pointer.
  - Examples of *bad programming practice*:
    - An owning raw pointer is tied to nothing.
    - A heap-allocated C-style array is tied to nothing.

Feedback (stop recording)