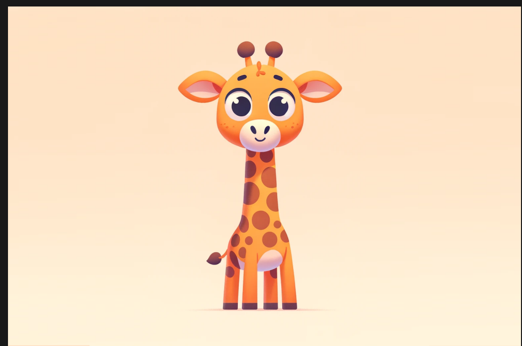
COMP6771

🍪 Smart Pointers

Lecture 5.3

Author(s): Hayden Smith



(Download as PDF)

In This Lecture

- **Why? 🤔**
  - Managing unnamed / heap memory can be dangerous, as there is always the chance that the resource is not released / free'd properly. We need solutions to help with this.

- **What? 📰**
  - Smart pointers
  - Unique pointer, shared pointer
  - Partial construction

⏰ Object Lifetimes

To create safe object lifetimes 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
  - Examples of bad programming practice
    - An owning raw pointer is tied to nothing
    - A C-style array is tied to nothing

🦞 RAII

RAII = Resource Acquisition Is Initialization

In summary, resource management was really about emphasising RAII

- Resource = heap object
- A concept where we encapsulate resources inside objects
  - Acquire the resource in the constructor
  - Release the resource in the destructor
  - eg. Memory, locks, files
- Every resource should be owned by either:
  - Another resource (eg. smart pointer, data member)
  - Named resource on the stack
  - A nameless temporary variable

😷 Making A Pointer Safe

We could write a class to make a pointer safe.

```cpp
// myintpointer.h

class MyIntPointer {
 public:
  // This is the constructor
  MyIntPointer(int* value): value_{value} {}

  // This is the destructor
  ~MyIntPointer() {
    // Similar to C's free function.
    delete value_;
  }

  int* GetValue() {
    return value_
  }

 private:
  int* value_;
};
```

```cpp
void fn() {
  // Similar to C's malloc
  MyIntPointer p{new int{5}};
  // Copy the pointer;
  MyIntPointer q{p.GetValue()};
  // p and q are both now destructed.
  // What happens?
}
```

🧠 Smart Pointers

- Smart pointers are ways of wrapping unnamed (i.e. raw pointer) heap objects in named stack objects to that object lifetimes can be managed much more safely
- Introduced in C++11
- Usually two ways of solving problems
  - unique_ptr + raw pointers
  - shared_ptr + weak_ptr

Type	Shared ownership	Take ownership
std::unique_ptr<T>	No	Yes
raw pointers	No	No
std::shared_ptr<T>	Yes	Yes
std::weak_ptr<T>	No	No

😍 Unique Pointer

- **std::unique_ptr<T>**
  - The unique pointer owns the object
  - When the unique pointer is destructed, the underlying object is too
- **raw pointer (observer)**
  - Unique pointer may have many observers
  - There is an appropriate use of raw pointers (or refereces) in C++
  - Once the original pointer is destructed, you must ensure you don't access the raw pointers (no checks exist)
  - Those observers do not have ownership over the pointer

Also note the use of `nullptr` in C++ instead of `NULL`.

😍 Unique Pointer

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

int main() {
  auto up1 = std::unique_ptr<int>{new int};
  // auto up2 = up1; // no copy constructor
  std::unique_ptr<int> up3;
  // up3 = up2; // no copy assignment

  up3.reset(up1.release()); // OK
  auto up4 = std::move(up3); // OK
  std::cout << up4.get() << "\n";
  std::cout << *up4 << "\n";
  std::cout << *up1 << "\n";
}
```

unique.cpp

👀 Observer Pointer

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

int main() {
  auto up1 = std::unique_ptr<int>{new int{0}};
  *up1 = 5;
  std::cout << *up1 << "\n";
  auto op1 = up1.get();
  *op1 = 6;
  std::cout << *op1 << "\n";
  up1.reset();
  std::cout << *op1 << "\n";
}
```

observer.cpp

❌ Removing New/Delete

We can use another function to remove the need for the `new` keyword. It has other benefits that we will explore later.

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

auto main() -> int {
  // 1 - Worst - you can accidentally own the resource multiple
  // times, or easily forget to own it.
  // auto* silly_string = new std::string{"Hi"};
  // auto up1 = std::unique_ptr<std::string>(silly_string);
  // auto up11 = std::unique_ptr<std::string>(silly_string);

  // 2 - Not good - requires actual thinking about whether there's a leak.
  auto up2 = std::unique_ptr<std::string>(new std::string("Hello"));

  // 3 - Good - no thinking required.
  auto up3 = std::make_unique<std::string>("Hello");

  std::cout << *up2 << "\n";
  std::cout << *up3 << "\n";
  // std::cout << *(up3.get()) << "\n";
  // std::cout << up3->size();
}
```

smart-no-new.cpp

🙏 Shared Pointer

- **`std::shared_ptr<T>`**
- Several shared pointers share ownership of the object
  - A reference counted pointer
  - When a shared pointer is destructed, **if it is the only shared pointer left** pointing at the object, then the **object is destroyed**
  - May also have many observers
    - Just because the pointer has shared ownership doesn't mean the observers should get ownership too
- **`std::weak_ptr<T>`**
  - Weak pointers are used with shared 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

🙏 Shared Pointer

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

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

  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";
  y.reset(); // Deletes the memory, since
  // no one else owns the memory
  std::cout << "use count: " << x.use_count() << "\n";
  std::cout << "value: " << *y << "\n";
}
```

shared.cpp

Weak Pointer

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

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

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

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

weak.cpp

🤷‍♂️ When To Use Which

- **Unique pointer** vs **Shared pointer**
  - You almost always want a unique pointer over a hared pointer
  - Use a shared pointer if either:
    - An object has multiple owenrs, and **you don't know which one will stay around the longest**
    - You need temporary ownership (unlikely)

📋 Examples Of Smart Pointer Usage

- Linked list
- Doubly linked list
- Tree
- Graph

Leak Freedom

"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

Stack Unwinding

- Stack unwinding is the process of exiting the stack frames until we find an exception handler for the function
- This calls any destructors on the way out
  - Any resources not managed by destructors won't get freed up
  - If an exception is thrown during stack unwinding, std::terminate is called

🚰 Stack Unwinding

```cpp
void g() {
  throw std::runtime_error{""};
}

int main() {
  auto ptr = new int{5};
  g();
  // Never executed.
  delete ptr;
}
```

```cpp
void g() {
  throw std::runtime_error{""};
}

int main() {
  auto ptr = new int{5};
  g();
  auto uni = std::unique_ptr<int>(ptr);
}
```

```cpp
void g() {
  throw std::runtime_error{""};
}

int main() {
  auto ptr = std::make_unique<int>(5);
  g();
}
```

🪓 Exceptions And Destructors

- During stack unwinding, std::terminate() will be called if an exception leaves a destructor
- The resources may not be released properly if an exception leaves a destructor
- All exceptions that occur inside a destructor should be handled inside the destructor
- Destructors usually don't throw, and need to explicitly opt in to throwing
  - STL types don't do that

Partial Construction

- What happens if an exception is thrown halfway through a constructor?
  - The C++ standard: "An object that is partially constructed or partially destroyed will have destructors executed for all of its fully constructed subobjects"
  - A destructor is not called for an object that was partially constructed
  - Except for an exception thrown in a constructor that delegates (why?)

```cpp
#include <exception>

class my_int {
public:
   my_int(int const i) : i_{i} {
      (void)i_;
      if (i == 2) {
         throw std::exception();
      }
   }
private:
   int i_;
};

class unsafe_class {
public:
   unsafe_class(int a, int b)
   : a_{new my_int{a}}
   , b_{new my_int{b}}
   {}

  ~unsafe_class() {
    delete a_;
    delete b_;
  }
private:
   my_int* a_;
   my_int* b_;
};

int main() {
  auto a = unsafe_class(1, 2);
}
```

partial-construction-bad.cpp

🧱 Partial Construction: Solution

- Option 1: Try / catch in the constructor
  - Very messy, but works (if you get it right...)
  - Doesn't work with initialiser lists (needs to be in the body)
- Option 2:
  - An object managing a resource should initialise the resource last
    - The resource is only initialised when the whole object is
    - Consequence: An object can only manage one resource
    - If you want to manage multiple resources, instead manage several wrappers , which each manage one resource

```cpp
#include <exception>
#include <memory>

class my_int {
public:
   my_int(int const i)
   : i_{i} {
      (void)i_;
      if (i == 2) {
         throw std::exception();
      }
   }
private:
   int i_;
};

class safe_class {
public:
   safe_class(int a, int b)
   : a_(std::make_unique<my_int>(a))
   , b_(std::make_unique<my_int>(b))
   {}
private:
   std::unique_ptr<my_int> a_;
   std::unique_ptr<my_int> b_;
};

int main() {
  auto a = safe_class(1, 2);
}
```

partial-construction-good.cpp

👂 Feedback

Or go to the form here.