CS100 Lecture 18

Smart Pointers

Contents

Ideas

Memory management is difficult!

For raw pointers obtained from `new` / `new[]` expressions, a manual `delete` / `delete[]` is required.

```cpp
void runGame(const std::vector<Option> &options, const Settings &settings) {
  auto pWindow = new Window(settings.width, settings.height, settings.mode);
  auto pGame = new Game(options, settings, pWindow);
  // Run the game ...
  while (true) {
    auto key = getUserKeyAction();
    // ...
  }
  delete pGame;    // You must not forget this.
  delete pWindow; // You must not forget this.
}
```

Will you always remember to `delete`?

Will you always remember to `delete`?

```cpp
void runGame(const std::vector<Option> &options, const Settings &settings) {
  auto pWindow = new Window(settings.width, settings.height, settings.mode);
  auto pGame = new Game(options, settings, pWindow);
  if (/* condition1 */) {
    // ...
    return; // `pWindow` and `pGame` should also be `delete`d here!
  }
  // Run the game ...
  while (true) {
    auto key = getUserKeyAction();
    // ...
    if (/* condition2 */) {
      // ...
      return; // `pWindow` and `pGame` should also be `delete`d here!
    }
  }
  delete pGame;
  delete pWindow;
}
```

Idea: Make use of destructors.

```
struct WindowPtr { // A "smart pointer".
  Window *ptr;
  WindowPtr(Window *p) : ptr(p) {}
  ~WindowPtr() { delete ptr; } // The destructor will `delete` the object.
};
```

When the control reaches the end of the scope in which the `WindowPtr` lives, the destructor of `WindowPtr` will be called automatically.

```
void runGame(const std::vector<Option> &options, const Settings &settings) {
  WindowPtr pWindow(new Window(settings.width, settings.height, settings.mode));
  if (/* condition1 */) {
    // ...
    return; // `pWindow` is destroyed automatically, with its destructor called.
  }
  // ...
  // `pWindow` is destroyed automatically, with its destructor called.
}
```

What if `WindowPtr` is copied?

Now `WindowPtr` only has a compiler-generated copy constructor, which copies the value of `ptr`.

```cpp
{
  WindowPtr pWindow(new Window(settings.width, settings.height, settings.mode));
  auto copy = pWindow; // `copy.ptr` and `pWindow.ptr` point to the same object!
} // The object is deleted twice! Disaster!
```

What should be the behavior of `auto copy = pWindow;`? Possible designs are:

1. Copy the object, as if `WindowPtr copy(new Window(*pWindow.ptr));`.
2. Copy the pointer, as if `WindowPtr copy(pWindow.ptr);`.
    - To avoid disasters caused by multiple `delete`s, some special design is needed.
3. Disable it. If there is no unique reasonable design, disable that operation.

What if `WindowPtr` is copied?

What should be the behavior of `auto copy = pWindow;`? Possible designs are:

1. Copy the object, as if `WindowPtr copy(new Window(*pWindow.ptr));`.
   - **"Value semantics"**
   - Typical example: Standard library containers. When you copy a `std::string`, a new string is created, with the **contents** copied.
   - May be referred to as "deep copy" in some other languages.
2. Copy the pointer, as if `WindowPtr copy(pWindow.ptr);`.
   - To avoid disasters caused by multiple `delete`s, some special design is needed.
   - **"Pointer semantics"**, or **"Reference semantics"**
   - "shallow copy" in some other languages.
3. Disable it. If there is no unique reasonable design, disable that operation.
   - In this case, `pWindow` **exclusively owns** the `Window` object.

Overview of smart pointers

A "smart pointer" is a pointer that manages its resources.

Possible behaviors of copy of a smart pointer:

1. Copy the object. (Value semantics)
    - **Standard library containers.** e.g. `std::string`, `std::vector`, `std::set`, ...
2. Copy the pointer, but with some special design. (Pointer semantics)
    - **`std::shared_ptr<T>`**. Defined in standard library file `<memory>`.
3. Disable it. (Unique ownership)
    - **`std::unique_ptr<T>`**. Defined in standard library file `<memory>`.

The smart pointers `std::shared_ptr<T>`, `std::unique_ptr<T>` and `std::weak_ptr<T>` are **the C++'s answer to garbage collection**.

- `std::weak_ptr` is not covered in CS100.

Overview of smart pointers

The smart pointers `std::shared_ptr<T>`, `std::unique_ptr<T>` and `std::weak_ptr<T>` are **the C++'s answer to garbage collection**.

Smart pointers support the similar operations as raw pointers:

- `*sp` returns reference to the pointed-to object.
- `sp->mem` is equivalent to `(*sp).mem`.
- `sp` is *contextually convertible* to `bool`: It can be treated as a "condition".
  - It can be placed at the "condition" part of `if`, `for`, `while`, `do` statements.
  - It can be used as operands of `&&`, `||`, `!` or the first operand of `?:`.
  - In all cases, the conversion result is `true` **iff** `sp` holds an object (not "null").

[Best practice] In modern C++, prefer smart pointers to raw pointers.

`std::unique_ptr`

Design: Unique ownership of the object

```
class WindowPtr {
  Window *ptr;
public:
  WindowPtr(Window *p = nullptr) : ptr(p) {}
  ~WindowPtr() { delete ptr; }



};
```

A "unique-pointer" saves a raw pointer internally, pointing to the object it owns.

When the unique-pointer is destroyed, it disposes of the object it owns.

Design: Unique ownership of the object

```cpp
class WindowPtr {
  Window *ptr;
public:
  WindowPtr(Window *p = nullptr) : ptr(p) {}
  ~WindowPtr() { delete ptr; }
  WindowPtr(const WindowPtr &) = delete;
  WindowPtr &operator=(const WindowPtr &) = delete;


};
```

The unique-pointer **exclusively** owns the object. Copying a unique-pointer is not allowed.

Design: Unique ownership of the object

```cpp
class WindowPtr {
  Window *ptr;
public:
  WindowPtr(Window *p = nullptr) : ptr(p) {}
  ~WindowPtr() { delete ptr; }
  WindowPtr(const WindowPtr &) = delete;
  WindowPtr &operator=(const WindowPtr &) = delete;
  WindowPtr(WindowPtr &&other) noexcept : ptr(other.ptr) { other.ptr = nullptr; }
  WindowPtr &operator=(WindowPtr &&other) noexcept {
    if (&other != this) {
      delete ptr; ptr = other.ptr; other.ptr = nullptr;
    }
    return *this;
  }
};
```

Move of a unique-pointer: **transfer of ownership**.

- **Move-only type**

`std::unique_ptr`

Like `std::vector`, `std::unique_ptr` is also a class template. It is not a type itself.

- `std::unique_ptr<PointeeType>` is the complete type name, where `PointeeType` is the type of the object that it points to.
- For `T` $\neq$ `U`, `std::unique_ptr<T>` and `std::unique_ptr<U>` are **two different and independent types**.

Same for `std::shared_ptr`, which we will talk about later.

Creating a `std::unique_ptr`: Two common ways

- Pass a pointer created by `new` to the constructor:

```
std::unique_ptr<Student> p(new Student("Bob", 2020123123));
```

  - Here `<Student>` can be omitted. The compiler is able to deduce it.

- Use `std::make_unique<T>`, and pass the initializers to it.

```
std::unique_ptr<Student> p1 = std::make_unique<Student>("Bob", 2020123123);
auto p2 = std::make_unique<Student>("Alice", 2020321321);
```

  - `std::make_unique<T>(args...)` *perfectly forwards* the arguments `args...` to the constructor of `T`, as if the object were created by `new T(args...)`.
  - `std::make_unique<T>` returns a `std::unique_ptr<T>` to the created object.

Default initialization of a `std::unique_ptr`

```
std::unique_ptr<T> up;
```

The default constructor of `std::unique_ptr<T>` initializes `up` to be a "null pointer".

`up` is in the state that does not own any object.

- This is a defined and deterministic behavior! It is **not** holding some indeterminate value.
  - The standard library hates indeterminate values, just as we do.

`std::unique_ptr`: Automatic memory management

```
void foo() {
  auto pAlice = std::make_unique<Student>("Alice", 2020321321);
  // Do something...
  if (some_condition()) {
    auto pBob = std::make_unique<Studnet>("Bob", 2020123123);
    // ...
  } // `Student::~Student()` is called for Bob,
    // because the lifetime of `pBob` ends.
} // `Student::~Student()` is called for Alice,
  // because the lifetime of `pAlice` ends.
```

A `std::unique_ptr` automatically calls the destructor once it gets destroyed or assigned a new value.

- No manual `delete` needed!

`std::unique_ptr`: Move-only

```cpp
auto p = std::make_unique<std::string>(5, 'c');
std::cout << *p << std::endl;                           // Prints "ccccc".
auto q = p;                                             // Error. Copy is not allowed.
auto r = std::move(p);                                  // Correct.
// Now the ownership of this string has been transferred to `r`.
std::cout << *r << std::endl; // Prints "ccccc".
if (!p) // true
  std::cout << "p is \"null\" now." << std::endl;
```

`std::unique_ptr` is not copyable, but only movable.

- Remember, only one `std::unique_ptr` can point to the managed object.
- Move of a `std::unique_ptr` is the transfer of ownership of the managed object.

`std::unique_ptr`: Move-only

```cpp
auto p = std::make_unique<std::string>(5, 'c');
std::cout << *p << std::endl;                      // Prints "ccccc".
auto q = p;                                        // Error. Copy is not allowed.
auto r = std::move(p);                             // Correct.
// Now the ownership of this string has been transferred to `r`.
std::cout << *r << std::endl; // Prints "ccccc".
if (!p) // true
  std::cout << "p is \"null\" now." << std::endl;
```

After `auto up2 = std::move(up1);`, `up1` becomes "null". The object that `up1` used to manage now belongs to `up2`.

The assignment `up2 = std::move(up1)` destroys the object that `up2` used to manage, and lets `up2` take over the object managed by `up1`. After that, `up1` becomes "null".

Express your intent precisely.

You may accidentally write the following code:

```
// Given that `pWindow` is a `std::unique_ptr<Window>`.
auto p = pWindow; // Oops, attempting to copy a `std::unique_ptr`.
```

The compiler gives an error, complaining about the use of deleted copy constructor.

What are you going to do?

A. Change it to `auto p = std::move(pWindow);`.
B. Give up on smart pointers, and switch back to raw pointers.
C. Copy-and-paste the compiler output and ask ChatGPT.

Express your intent precisely.

You may accidentally write the following code:

```
// Given that `pWindow` is a `std::unique_ptr<Window>`.
auto p = pWindow; // Oops, attempting to copy a `std::unique_ptr`.
```

The compiler gives an error, complaining about the use of deleted copy constructor.

1. Syntactically, a `std::unique_ptr` is not copyable, but you are copying it. **(Direct cause of the error)**
2. Logically, a `std::unique_ptr` must exclusively manage the pointed-to object. Why would you copy a `std::unique_ptr`?
    - The **root cause of the error** is related to your intent: What are you going to do with `p`?

Express your intent precisely.

```cpp
// Given that `pWindow` is a `std::unique_ptr<Window>`.
auto p = pWindow; // Oops, attempting to copy a `std::unique_ptr`.
```

What are you going to do with `p`?

- If you want to copy the pointed-to object, change it to `auto p = std::make_unique<Window>(*pWindow);`.
- If you want `p` to be just an ***observer***, write `auto p = pWindow.get();`.
  - `pWindow.get()` returns a **raw pointer** to the object, which is of type `Window *`.
  - Be careful! As an observer, `p` should never interfere in the lifetime of the object. A simple `delete p;` will cause disaster.

Express your intent precisely.

```cpp
// Given that `pWindow` is a `std::unique_ptr<Window>`.
auto p = pWindow; // Oops, attempting to copy a `std::unique_ptr`.
```

What are you going to do with `p`?

- If you want `p` to take over the object managed by `pWindow`, change it to `auto p = std::move(pWindow);`.
  - Be careful! `pWindow` will no longer own that object.
- If you want to `p` to be another smart pointer that ***shares*** the ownership with `pWindow`, `std::unique_ptr` is not suitable here. ⇒ See `std::shared_ptr` later.

Returning a `std::unique_ptr`

```cpp
struct Window {
  // A typical "factory" function.
  static std::unique_ptr<Window> create(const Settings &settings) {
    auto pW = std::make_unique<Window>(/* some arguments */);
    logWindowCreation(pW);
    // ...
    return pW;
  }
};
auto state = Window::create(my_settings);
```

A temporary is move-constructed from `pW`, and then is used to move-construct `state`.

- These two moves can be optimized out by NRVO.

Other operations on `std::unique_ptr`

`up.reset()`, `up.release()`, `up1.swap(up2)`, `up1 == up2`, etc.

Full list of operations supported on a `std::unique_ptr`.

`std::unique_ptr` for array type

By default, the destructor of `std::unique_ptr<T>` uses a `delete` expression to destroy the object it holds.

What happens if `std::unique_ptr<T> up(new T[n]);`?

`std::unique_ptr` for array type

By default, the destructor of `std::unique_ptr<T>` uses a `delete` expression to destroy the object it holds.

What happens if `std::unique_ptr<T> up(new T[n]);`?

- The memory is obtained using `new[]`, but deallocated by `delete`! **Undefined behavior.**

`std::unique_ptr` for array type

A *template specialization*: `std::unique_ptr<T[]>`.

- Specially designed to represent pointers that point to a "dynamic array" of objects.
- It has some array-specific operators, e.g. `operator[]`. In contrast, it does not support `operator*` and `operator->`.
- It uses `delete[]` instead of `delete` to destroy the objects.

```
auto up = std::make_unique<int[]>(n);
std::unique_ptr<int[]> up2(new int[n]{}); // equivalent
for (auto i = 0; i != n; ++i)
  std::cout << up[i] << ' ';
```

~~`std::unique_ptr` for array type~~

~~A *template specialization*: `std::unique_ptr<T[]>` .~~

~~- Specially designed to represent pointers that point to a "dynamic array" of objects.~~
~~- It has some array-specific operators, e.g. `operator[]` . In contrast, it does not support `operator*` and `operator->` .~~
~~- It uses `delete[]` instead of `delete` to destroy the objects.~~

Use standard library containers instead!

They almost always do a better job. `std::unique_ptr<T[]>` is seldom needed.

`std::unique_ptr` is zero-overhead.

`std::unique_ptr` stores nothing more than a raw pointer. [1]

It does nothing more than better copy / move control and automatic object destruction.

Zero-overhead: Using a `std::unique_ptr` does not cost more time or space than using raw pointers.

[Best practice] Use `std::unique_ptr` for exclusive-ownership resource management.

`std::shared_ptr`

Motivation

A `std::unique_ptr` exclusively owns an object, but sometimes this is not convenient.

```cpp
struct WindowManager {
  void addWindow(const std::unique_ptr<Window> &pW) {
    mWindows.push_back(pW); // Error. Attempts to copy a `std::unique_ptr`.
  }
private:
  std::vector<std::unique_ptr<Window>> mWindows;
};

struct Window {
  static std::unique_ptr<Window> create(const Settings &settings) {
    auto pW = std::make_unique<Window>(/* some arguments */);
    logWindowCreation(pW);
    settings.getWindowManager().addWindow(pW);
    return pW;
  }
};
```

Motivation

Design a "shared-pointer" that allows the object it manages to be ***shared***.

When should the object be destroyed?

- A `std::unique_ptr` destroys the object it manages when the pointer itself is destroyed.
- If we allow many shared-pointers to point to the same object, how can we know when to destroy that object?

Idea: Reference counting

Set a **counter** that counts how many `WindowPtr`s are pointing to it:

```
struct WindowWithCounter {
  Window theWindow;
  int refCount = 1;
};
```

When a new object is created by a `WindowPtr`, set the `refCount` to `1`.

Idea: Reference counting

When a `WindowPtr` is copied, let it point to the same object, and increment the counter.

```
class WindowPtr {
  WindowWithCounter *ptr;
public:
  WindowPtr(const WindowPtr &other) : ptr(other.ptr) { ++ptr->refCount; }









};
```

Idea: Reference counting

For copy assignment: the counter of the old object should be decremented.

- If it reaches zero, destroy that object!

```cpp
class WindowPtr {
  WindowWithCounter *ptr;
public:
  WindowPtr(const WindowPtr &other) : ptr(other.ptr) { ++ptr->refCount; }
  WindowPtr &operator=(const WindowPtr &other) {
    if (--ptr->refCount == 0)
      delete ptr;
    ptr = other.ptr;
    ++ptr->refCount;
    return *this;
  }
};
```

*** Is this correct?**

Idea: Reference counting

Self-assignment safe!!!

```
class WindowPtr {
  WindowWithCounter *ptr;
public:
  WindowPtr(const WindowPtr &other) : ptr(other.ptr) { ++ptr->refCount; }
  WindowPtr &operator=(const WindowPtr &other) {
    ++other.ptr->refCount;
    if (--ptr->refCount == 0)
      delete ptr;
    ptr = other.ptr;
    return *this;
  }
};
```

Idea: Reference counting

Destructor: decrement the counter, and destroy the object if the counter reaches zero.

```cpp
class WindowPtr {
  WindowWithCounter *ptr;
public:
  WindowPtr(const WindowPtr &other) : ptr(other.ptr) { ++ptr->refCount; }
  WindowPtr &operator=(const WindowPtr &other) {
    ++other.ptr->refCount;
    if (--ptr->refCount == 0)
      delete ptr;
    ptr = other.ptr;
    return *this;
  }
  ~WindowPtr() {
    if (--ptr->refCount == 0)
      delete ptr;
  }
};
```

Idea: Reference counting

Move: Just "steal" the object. It is also the transfer of ownership.

```
class WindowPtr {
  WindowWithCounter *ptr;
 public:
  WindowPtr(WindowPtr &&other) noexcept : ptr(other.ptr) { other.ptr = nullptr; }
  WindowPtr &operator=(WindowPtr &&other) noexcept {
    if (this != &other) {
      if (--ptr->refCount == 0)
        delete ptr;
      ptr = other.ptr; other.ptr = nullptr;
    }
    return *this;
  }
};
```

`std::shared_ptr`

A smart pointer that uses **reference counting** to manage shared objects.

Create a `shared_ptr`:

```
std::shared_ptr<Type> sp2(new Type(args));
auto sp = std::make_shared<Type>(args); // equivalent, but better
```

For example:

```
// sp points to a string "cccccccccc".
auto sp = std::make_shared<std::string>(10, 'c');

auto pWindow = std::make_shared<Window>(80, 24, my_settings.mode);
```

Create a `shared_ptr`

Note: For `std::unique_ptr`, both of the following ways are ok (since C++17):

```
auto up = std::make_unique<Type>(args);
std::unique_ptr<Type> up2(new Type(args));
```

For `std::shared_ptr`, **`std::make_shared` is preferable to directly using `new`**.

```
auto sp = std::make_shared<Type>(args); // preferred
std::shared_ptr<Type> sp2(new Type(args)); // ok, but less preferred
```

Read *Effective Modern C++* Item 21. (Note that this book is based on C++14.)

[Best practice] Prefer `std::make_shared` to directly using `new` when creating a `std::shared_ptr`.

Operations

`*` and `->` can be used as if it is a raw pointer:

```
auto sp = std::make_shared<std::string>(10, 'c');
std::cout << *sp << std::endl;        // cccccccccc
std::cout << sp->size() << std::endl; // 10
```

`sp.use_count()`: The value of the reference counter.

```
auto sp = std::make_shared<std::string>(10, 'c');
{
  auto sp2 = sp;
  std::cout << sp.use_count() << std::endl; // 2
} // sp2 is destroyed
std::cout << sp.use_count() << std::endl;   // 1
```

Operations

Full list of supported operations on `std::shared_ptr`.

`std::shared_ptr` is relatively easy to use, since you are free to create many `std::shared_ptr`s pointing to one object.

However, `std::shared_ptr` **has time and space overhead**. Copy of a `std::shared_ptr` requires maintenance of reference counter.

Summary

`std::unique_ptr`

- Exclusive-ownership.
- Move-only. Move is the transfer of ownership.
- Zero-overhead.

`std::shared_ptr`

- Shared-ownership.
- Uses reference counting.
  - Copy increments the reference counter.
  - When the counter is decremented to zero, the object is destroyed.

Notes

[1] The **deleter** that a `std::unique_ptr` uses is customizable, which also has to be stored in a `std::unique_ptr`. By default, `std::default_delete` is used that performs `delete ptr;` to destroy the object and release the memory. `std::unique_ptr` often uses some space-saving tricks to store the deleter. If the deleter is "stateless" (e.g. an object with no non-`static` data members), it may be stored with no unique address so that no extra space is required. `std::default_delete` belongs to this kind. Therefore, `sizeof(std::unique_ptr<T>)` is often reasonably equal to `sizeof(T *)`.