

15.5 — std::unique_ptr

BY ALEX ON MARCH 15TH, 2017 | LAST MODIFIED BY ALEX ON JULY 9TH, 2018

At the beginning of the chapter, we discussed how use of pointers can lead to bugs and memory leaks in some situations. For example, this can happen when a function early returns, or throws an exception, and the pointer is not properly deleted.

```
1  #include <iostream>
2
3  void someFunction()
4  {
5      Resource *ptr = new Resource;
6
7      int x;
8      std::cout << "Enter an integer: ";
9      std::cin >> x;
10
11     if (x == 0)
12         throw 0; // the function returns early, and ptr won't be deleted!
13
14     // do stuff with ptr here
15
16     delete ptr;
17 }
```

Now that we've covered the fundamentals of move semantics, we can return to the topic of smart pointer classes. As a reminder, a smart pointer is a class that manages a dynamically allocated object. Although smart pointers can offer other features, the defining characteristic of a smart pointer is that it manages a dynamically allocated resource, and ensures the dynamically allocated object is properly cleaned up at the appropriate time (usually when the smart pointer goes out of scope).

Because of this, smart pointers should never be dynamically allocated themselves (otherwise, there is the risk that the smart pointer may not be properly deallocated, which means the object it owns would not be deallocated, causing a memory leak). By always allocating smart pointers statically (as local variables or composition members of a class), we're guaranteed that the smart pointer will properly go out of scope when the function or object it is contained within ends, ensuring the object the smart pointer owns is properly deallocated.

C++11 standard library ships with 4 smart pointer classes: `std::auto_ptr` (which you shouldn't use -- it's being removed in C++17), `std::unique_ptr`, `std::shared_ptr`, and `std::weak_ptr`. `std::unique_ptr` is by far the most used smart pointer class, so we'll cover that one first. In the next lesson, we'll cover `std::shared_ptr` and `std::weak_ptr`.

std::unique_ptr

`std::unique_ptr` is the C++11 replacement for `std::auto_ptr`. It should be used to manage any dynamically allocated object that is not shared by multiple objects. That is, `std::unique_ptr` should completely own the object it manages, not share that ownership with other classes. `std::unique_ptr` lives in the `<memory>` header.

Let's take a look at a simple smart pointer example:

```
1  #include <iostream>
2  #include <memory> // for std::unique_ptr
3
4  class Resource
5  {
6  public:
7      Resource() { std::cout << "Resource acquired\n"; }
8      ~Resource() { std::cout << "Resource destroyed\n"; }
9  };
10
11 int main()
12 {
13     // allocate a Resource object and have it owned by std::unique_ptr
14     std::unique_ptr<Resource> res(new Resource);
15 }
```

```
16     return 0;
17 } // res goes out of scope here, and the allocated Resource is destroyed
```

Because the `std::unique_ptr` is allocated on the stack here, it's guaranteed to eventually go out of scope, and when it does, it will delete the Resource it is managing.

Unlike `std::auto_ptr`, `std::unique_ptr` properly implements move semantics.

```
1  #include <iostream>
2  #include <memory> // for std::unique_ptr
3
4  class Resource
5  {
6  public:
7      Resource() { std::cout << "Resource acquired\n"; }
8      ~Resource() { std::cout << "Resource destroyed\n"; }
9  };
10
11 int main()
12 {
13     std::unique_ptr<Resource> res1(new Resource); // Resource created here
14     std::unique_ptr<Resource> res2; // Start as nullptr
15
16     std::cout << "res1 is " << (static_cast<bool>(res1) ? "not null\n" : "null\n");
17     std::cout << "res2 is " << (static_cast<bool>(res2) ? "not null\n" : "null\n");
18
19     // res2 = res1; // Won't compile: copy assignment is disabled
20     res2 = std::move(res1); // res2 assumes ownership, res1 is set to null
21
22     std::cout << "Ownership transferred\n";
23
24     std::cout << "res1 is " << (static_cast<bool>(res1) ? "not null\n" : "null\n");
25     std::cout << "res2 is " << (static_cast<bool>(res2) ? "not null\n" : "null\n");
26
27     return 0;
28 } // Resource destroyed here when res2 goes out of scope
```

This prints:

```
Resource acquired
res1 is not null
res2 is null
Ownership transferred
res1 is null
res2 is not null
Resource destroyed
```

Because `std::unique_ptr` is designed with move semantics in mind, copy initialization and copy assignment are disabled. If you want to transfer the contents managed by `std::unique_ptr`, you must use move semantics. In the program above, we accomplish this via `std::move` (which converts `res1` into an r-value, which triggers a move assignment instead of a copy assignment).

Accessing the managed object

`std::unique_ptr` has an overloaded `operator*` and `operator->` that can be used to return the resource being managed. `Operator*` returns a reference to the managed resource, and `operator->` returns a pointer.

Remember that `std::unique_ptr` may not always be managing an object -- either because it was created empty (using the default constructor or passing in a `nullptr` as the parameter), or because the resource it was managing got moved to another `std::unique_ptr`. So before we use either of these operators, we should check whether the `std::unique_ptr` actually has a resource. Fortunately, this is easy: `std::unique_ptr` has a cast to `bool` that returns true if the `std::unique_ptr` is managing a resource.

Here's an example of this:

```
1  #include <iostream>
2  #include <memory> // for std::unique_ptr
```

```

3
4 class Resource
5 {
6 public:
7     Resource() { std::cout << "Resource acquired\n"; }
8     ~Resource() { std::cout << "Resource destroyed\n"; }
9     friend std::ostream& operator<<(std::ostream& out, const Resource &res)
10    {
11        out << "I am a resource\n";
12        return out;
13    }
14 };
15
16 int main()
17 {
18     std::unique_ptr<Resource> res(new Resource);
19
20     if (res) // use implicit cast to bool to ensure res contains a Resource
21         std::cout << *res; // print the Resource that res is owning
22
23     return 0;
24 }

```

This prints:

```

Resource acquired
I am a resource
Resource destroyed

```

In the above program, we use the overloaded operator* to get the Resource object owned by std::unique_ptr res, which we then send to std::cout for printing.

std::unique_ptr and arrays

Unlike std::auto_ptr, std::unique_ptr is smart enough to know whether to use scalar delete or array delete, so std::unique_ptr is okay to use with both scalar objects and arrays.

However, std::array or std::vector (or std::string) are almost always better choices than using std::unique_ptr with a fixed array, dynamic array, or C-style string.

Rule: Favor std::array, std::vector, or std::string over a smart pointer managing a fixed array, dynamic array, or C-style string

std::make_unique

C++14 comes with an additional function named std::make_unique(). This templated function constructs an object of the template type and initializes it with the arguments passed into the function.

```

1  #include <memory> // for std::unique_ptr and std::make_unique
2  #include <iostream>
3
4  class Fraction
5  {
6  private:
7      int m_numerator = 0;
8      int m_denominator = 1;
9
10 public:
11     Fraction(int numerator = 0, int denominator = 1) :
12         m_numerator(numerator), m_denominator(denominator)
13     {
14     }
15
16     friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
17     {
18         out << f1.m_numerator << "/" << f1.m_denominator;
19         return out;

```

```

20     }
21 };
22
23
24 int main()
25 {
26     // Create a single dynamically allocated Fraction with numerator 3 and denominator 5
27     std::unique_ptr<Fraction> f1 = std::make_unique<Fraction>(3, 5);
28     std::cout << *f1 << '\n';
29
30     // Create a dynamically allocated array of Fractions of length 4
31     // We can also use automatic type deduction to good effect here
32     auto f2 = std::make_unique<Fraction[]>(4);
33     std::cout << f2[0] << '\n';
34
35     return 0;
36 }

```

The code above prints:

```

3/5
0/1

```

Use of `std::make_unique()` is optional, but is recommended over creating `std::unique_ptr` yourself. This is because code using `std::make_unique` is simpler, and it also requires less typing (when used with automatic type deduction). Furthermore it resolves an exception safety issue that can result from C++ leaving the order of evaluation for function arguments unspecified.

Rule: use `std::make_unique()` instead of creating `std::unique_ptr` and using `new` yourself

The exception safety issue in more detail

For those wondering what the “exception safety issue” mentioned above is, here’s a description of the issue.

Consider an expression like this one:

```
1 | some_function(std::unique_ptr<T>(new T), function_that_can_throw_exception());
```

The compiler is given a lot of flexibility in terms of how it handles this call. It could create a new `T`, then call `function_that_can_throw_exception()`, then create the `std::unique_ptr` that manages the dynamically allocated `T`. If `function_that_can_throw_exception()` throws an exception, then the `T` that was allocated will not be deallocated, because the smart pointer to do the deallocation hasn’t been created yet. This leads to `T` being leaked.

`std::make_unique()` doesn’t suffer from this problem because the creation of the object `T` and the creation of the `std::unique_ptr` happen inside the `std::make_unique()` function, where there’s no ambiguity about order of execution.

Returning `std::unique_ptr` from a function

`std::unique_ptr` can be safely returned from a function by value:

```

1 | std::unique_ptr<Resource> createResource()
2 | {
3 |     return std::make_unique<Resource>();
4 | }
5
6 | int main()
7 | {
8 |     std::unique_ptr<Resource> ptr = createResource();
9
10 |    // do whatever
11
12 |    return 0;
13 | }

```

In the above code, `createResource()` returns a `std::unique_ptr` by value. If this value is not assigned to anything, the temporary return value will go out of scope and the `Resource` will be cleaned up. If it is assigned (as shown in `main()`), move semantics will be employed

to transfer the Resource from the return value to the object assigned to (in the above example, ptr). This makes returning a resource by `std::unique_ptr` much safer than returning raw pointers!

In general, you should not return `std::unique_ptr` by pointer (ever) or reference (unless you have a specific compelling reason to).

Passing `std::unique_ptr` to a function

If you want the function to take ownership of the contents of the pointer, pass the `std::unique_ptr` by value. Note that because copy semantics have been disabled, you'll need to use `std::move` to actually pass the variable in.

```
1  #include <memory> // for std::unique_ptr
2
3  class Resource
4  {
5  public:
6      Resource() { std::cout << "Resource acquired\n"; }
7      ~Resource() { std::cout << "Resource destroyed\n"; }
8      friend std::ostream& operator<<(std::ostream& out, const Resource &res)
9      {
10         out << "I am a resource\n";
11         return out;
12     }
13 };
14
15 void takeOwnership(std::unique_ptr<Resource> res)
16 {
17     if (res)
18         std::cout << *res;
19 } // the Resource is destroyed here
20
21 int main()
22 {
23     auto ptr = std::make_unique<Resource>();
24
25     // takeOwnership(ptr); // This doesn't work, need to use move semantics
26     takeOwnership(std::move(ptr)); // ok: use move semantics
27
28     std::cout << "Ending program\n";
29
30     return 0;
31 }
```

The above program prints:

```
Resource acquired
I am a resource
Resource destroyed
Ending program
```

Note that in this case, ownership of the Resource was transferred to `takeOwnership()`, so the Resource was destroyed at the end of `takeOwnership()` rather than the end of `main()`.

However, most of the time, you won't want the function to take ownership of the resource. Although you can pass a `std::unique_ptr` by reference (which will allow the function to use the object without assuming ownership), you should only do so when the caller might alter or change the object being managed.

Instead, it's better to just pass the resource itself (by pointer or reference, depending on whether null is a valid argument). This allows the function to remain agnostic of how the caller is managing its resources. To get a raw resource pointer from a `std::unique_ptr`, you can use the `get()` member function:

```
1  #include <memory> // for std::unique_ptr
2  #include <iostream>
3
4  class Resource
5  {
```

```

6     public:
7         Resource() { std::cout << "Resource acquired\n"; }
8         ~Resource() { std::cout << "Resource destroyed\n"; }
9
10        friend std::ostream& operator<<(std::ostream& out, const Resource &res)
11        {
12            out << "I am a resource\n";
13            return out;
14        }
15    };
16
17    // The function only uses the resource, so we'll accept a pointer to the resource, not a reference to t
18    // he whole std::unique_ptr<Resource>
19    void useResource(Resource *res)
20    {
21        if (res)
22            std::cout << *res;
23    }
24
25    int main()
26    {
27        auto ptr = std::make_unique<Resource>();
28
29        useResource(ptr.get()); // note: get() used here to get a pointer to the Resource
30
31        std::cout << "Ending program\n";
32
33        return 0;
34    } // The Resource is destroyed here

```

The above program prints:

```

Resource acquired
I am a resource
Ending program
Resource destroyed

```

std::unique_ptr and classes

You can, of course, use `std::unique_ptr` as a composition member of your class. This way, you don't have to worry about ensuring your class destructor deletes the dynamic memory, as the `std::unique_ptr` will be automatically destroyed when the class object is destroyed. However, do note that if your class object is dynamically allocated, the object itself is at risk for not being properly deallocated, in which case even a smart pointer won't help.

Misusing std::unique_ptr

There are two easy ways to misuse `std::unique_ptr`s, both of which are easily avoided. First, don't let multiple classes manage the same resource. For example:

```

1 Resource *res = new Resource;
2 std::unique_ptr<Resource> res1(res);
3 std::unique_ptr<Resource> res2(res);

```

While this is legal syntactically, the end result will be that both `res1` and `res2` will try to delete the `Resource`, which will lead to undefined behavior.

Second, don't manually delete the resource out from underneath the `std::unique_ptr`.

```

1 Resource *res = new Resource;
2 std::unique_ptr<Resource> res1(res);
3 delete res;

```

If you do, the `std::unique_ptr` will try to delete an already deleted resource, again leading to undefined behavior.

Note that `std::make_unique()` prevents both of the above cases from happening inadvertently.

Quiz time

1) If your class has a smart pointer member, why should you try to avoid allocating objects of that class dynamically?

Hide Solution

Smart pointer members will only deallocate their resource if the class object actually goes out of scope. If you allocate a class object dynamically and do not properly delete it, the class object may never go out of scope, and the smart pointer will not have a chance to clean up its resource.

2) Convert the following program from using a normal pointer to using `std::unique_ptr` where appropriate:

```
1  #include <iostream>
2
3  class Fraction
4  {
5  private:
6      int m_numerator = 0;
7      int m_denominator = 1;
8
9  public:
10     Fraction(int numerator = 0, int denominator = 1) :
11         m_numerator(numerator), m_denominator(denominator)
12     {
13     }
14
15     friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
16     {
17         out << f1.m_numerator << "/" << f1.m_denominator;
18         return out;
19     }
20 };
21
22 void printFraction(const Fraction* const ptr)
23 {
24     if (ptr)
25         std::cout << *ptr;
26 }
27
28 int main()
29 {
30     Fraction *ptr = new Fraction(3, 5);
31
32     printFraction(ptr);
33
34     delete ptr;
35
36     return 0;
37 }
```

Hide Solution

```
1  #include <memory> // for std::unique_ptr
2  #include <iostream>
3
4  class Fraction
5  {
6  private:
7      int m_numerator = 0;
8      int m_denominator = 1;
9
10 public:
11     Fraction(int numerator = 0, int denominator = 1) :
12         m_numerator(numerator), m_denominator(denominator)
13     {
14     }
15 }
```

```

16 friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
17 {
18     out << f1.m_numerator << "/" << f1.m_denominator;
19     return out;
20 }
21 };
22
23 // This function uses a Fraction object, so we just pass the Fraction itself
24 // That way we don't have to worry about what kind of smart pointer (if any) the caller may be using
25 void printFraction(const Fraction* const ptr)
26 {
27     if (ptr)
28         std::cout << *ptr;
29 }
30
31 int main()
32 {
33     auto ptr = std::make_unique<Fraction>(3, 5);
34
35     printFraction(ptr.get());
36
37     return 0;
38 }

```



[15.6 -- std::shared_ptr](#)



[Index](#)



[15.4 -- std::move](#)

Share this:



[C++ TUTORIAL](#) | [PRINT THIS POST](#)

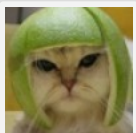
40 comments to 15.5 — std::unique_ptr



DecSco

[July 5, 2018 at 6:31 am](#) · Reply

I think the solution of question 2 makes sense, but the question leaves it very open, whether or not to convert the print function as well.



Alex

[July 9, 2018 at 1:38 pm](#) · Reply

I added "where appropriate", to help hint at the fact that it might not be appropriate to convert all pointers in this case. Thanks for the feedback!