22.5 — std::unique_ptr

At the beginning of the chapter, we discussed how the 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.

```
#include <iostream>

void someFunction()
{
    auto* ptr{ new Resource() };

    int x{};
    std::cout << "Enter an integer: ";
    std::cin >> x;

    if (x == 0)
        throw 0; // the function returns early, and ptr won't be deleted!

    // do stuff with ptr here

    delete ptr;
}
```

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 on the stack (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 (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 following lessons, 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:

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

class Resource
{
public:
	Resource() { std::cout << "Resource acquired\n"; }
	~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
	// allocate a Resource object and have it owned by std::unique_ptr
	std::unique_ptr<Resource> res{ new Resource() };

	return 0;
} // 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.

```
#include <iostream>
#include <memory> // for std::unique_ptr
#include <utility> // for std::move

class Resource
{
public:
        Resource() { std::cout << "Resource acquired\n"; }
        ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
        std::unique_ptr<Resource> res1{ new Resource{} }; // Resource created here
        std::unique_ptr<Resource> res2{}; // Start as nullptr

        std::cout << "res1 is " << (res1 ? "not null\n" : "null\n");
        std::cout << "res2 is " << (res2 ? "not null\n" : "null\n");

        // res2 = res1; // Won't compile: copy assignment is disabled
        res2 = std::move(res1); // res2 assumes ownership, res1 is set to null

        std::cout << "Ownership transferred\n";

        std::cout << "res1 is " << (res1 ? "not null\n" : "null\n");
        std::cout << "res2 is " << (res2 ? "not null\n" : "null\n");

        return 0;
} // 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:

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

class Resource
{
public:
	Resource() { std::cout << "Resource acquired\n"; }
	~Resource() { std::cout << "Resource destroyed\n"; }
	friend std::ostream& operator<<(std::ostream& out, const Resource &res)
	{
		out << "I am a resource";
		return out;
	}
};

int main()
{
	std::unique_ptr<Resource> res{ new Resource{} };

	if (res) // use implicit cast to bool to ensure res contains a Resource
		std::cout << *res << '\n'; // print the Resource that res is owning

	return 0;
}
```

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.

Best practice

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.

```
#include <memory> // for std::unique_ptr and std::make_unique
#include <iostream>

class Fraction
{
private:
	int m_numerator{ 0 };
	int m_denominator{ 1 };

public:
	Fraction(int numerator = 0, int denominator = 1) :
		m_numerator{ numerator }, m_denominator{ denominator }
	{
	}

	friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
	{
		out << f1.m_numerator << '/' << f1.m_denominator;
		return out;
	}
};


int main()
{
	// Create a single dynamically allocated Fraction with numerator 3 and
denominator 5
	// We can also use automatic type deduction to good effect here
	auto f1{ std::make_unique<Fraction>(3, 5) };
	std::cout << *f1 << '\n';

	// Create a dynamically allocated array of Fractions of length 4
	auto f2{ std::make_unique<Fraction[]>(4) };
	std::cout << f2[0] << '\n';

	return 0;
}
```

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, in C++14 it resolves an exception safety issue that can result from C++ leaving the order of evaluation for function arguments unspecified.

Best practice

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:

```
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.

This issue was fixed in C++17, as evaluation of function arguments can no longer be interleaved.

Returning std::unique_ptr from a function

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

```
#include <memory> // for std::unique_ptr

std::unique_ptr<Resource> createResource()
{
     return std::make_unique<Resource>();
}

int main()
{
    auto ptr{ createResource() };

    // do whatever

    return 0;
}
```

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()), in C++14 or earlier, move semantics will

be employed to transfer the Resource from the return value to the object assigned to (in the above example, ptr), and in C++17 or newer, the return will be elided. 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.

```
#include <iostream>
#include <memory> // for std::unique_ptr
#include <utility> // for std::move

class Resource
{
public:
        Resource() { std::cout << "Resource acquired\n"; }
        ~Resource() { std::cout << "Resource destroyed\n"; }
        friend std::ostream& operator<<(std::ostream& out, const Resource &res)
        {
                out << "I am a resource";
                return out;
        }
};

// This function takes ownership of the Resource, which isn't what we want
void takeOwnership(std::unique_ptr<Resource> res)
{
     if (res)
          std::cout << *res << '\n';
} // the Resource is destroyed here

int main()
{
    auto ptr{ std::make_unique<Resource>() };

//    takeOwnership(ptr); // This doesn't work, need to use move semantics
    takeOwnership(std::move(ptr)); // ok: use move semantics

    std::cout << "Ending program\n";

    return 0;
}
```

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 called function 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:

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

class Resource
{
public:
	Resource() { std::cout << "Resource acquired\n"; }
	~Resource() { std::cout << "Resource destroyed\n"; }

	friend std::ostream& operator<<(std::ostream& out, const Resource &res)
	{
		out << "I am a resource";
		return out;
	}
};

// The function only uses the resource, so we'll accept a pointer to the resource,
not a reference to the whole std::unique_ptr<Resource>
void useResource(const Resource* res)
{
	if (res)
		std::cout << *res << '\n';
	else
		std::cout << "No resource\n";
}

int main()
{
	auto ptr{ std::make_unique<Resource>() };

	useResource(ptr.get()); // note: get() used here to get a pointer to the
Resource

	std::cout << "Ending program\n";

	return 0;
} // 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, if the class object is not destroyed properly (e.g. it is dynamically allocated and not deallocated properly), then the std::unique_ptr member will not be destroyed either, and the object being managed by the std::unique_ptr will not be deallocated.

Misusing std::unique_ptr

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

```
Resource* res{ new Resource() };
std::unique_ptr<Resource> res1{ res };
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.

```
Resource* res{ new Resource() };
std::unique_ptr<Resource> res1{ res };
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

Question #1

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

```
#include <iostream>

class Fraction
{
private:
	int m_numerator{ 0 };
	int m_denominator{ 1 };

public:
	Fraction(int numerator = 0, int denominator = 1) :
		m_numerator{ numerator }, m_denominator{ denominator }
	{
	}

	friend std::ostream& operator<<(std::ostream& out, const Fraction &f1)
	{
		out << f1.m_numerator << '/' << f1.m_denominator;
		return out;
	}
};

void printFraction(const Fraction* ptr)
{
	if (ptr)
		std::cout << *ptr << '\n';
	else
		std::cout << "No fraction\n";
}

int main()
{
	auto* ptr{ new Fraction{ 3, 5 } };

	printFraction(ptr);

	delete ptr;

	return 0;
}
```

Show Solution