Chapter 3. Smart Pointers, Class Inheritance, and Polymorphism

---

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *learnmodcppfinance@gmail.com*.

Introduction

As discussed in his *Back to Basics* talk on object-oriented programming **{1}** at CppCon 2019, Jon Kalb states a modern day reality that "object-oriented programming is not what the cool kids are doing in

C++". More attention recently seems focused on things like functional programming and executing instructions at compile-time with template methods.

Still, there are realities that make object-oriented programming still highly relevant for financial programming. As mentioned in the overview in Chapter 1, components of financial models can be effectively represented as objects. For example,

- An options contract *has a* payoff
- A call payoff *is a* type of payoff
- A coupon-paying bond uses a yield curve to compute its valuation
- The price of a swaption depends on the value of the underlying swap

Each of these entities—options contract, payoff, bond, yield curve, swaption, and swap—can be represented as an object. Dependencies also often exist among particular objects.

There is also the fact that a lot of models libraries and financial systems were written at a time when object-oriented programming was in vogue, so writing enhancements or extending from existing code may very well require having a solid foundation in class design.

In the CppCon video referenced above, Mr Kalb in fact covers modern best practices in object-oriented programming and is recommended viewing. We will integrate some of these points into the discussion in this chapter that follows, along with some modern alternatives for managing a polymorphic resource on an option, namely a call or put payoff that inherits from a payoff base class. This is where smart pointers will eventually make our lives easier and our code less error-prone.

Before proceeding, a quick review of inheritance and polymorphism will be presented, but within a more modern context.

NOTE

Technically speaking, the term *polymorphism* in this chapter refers to *dynamic polymorphism* (also called *runtime polymorphism*), where the specific type of a derived object does not need to be known until runtime. This is in contrast to *static polymorphism*, which can help improve performance at runtime but comes with some limitations compared to the dynamic form, plus it involves the additional complexity and sophistication of template programming, as well as possibly increased compile times.

The focus of this chapter will be on the dynamic case. For more information on static polymorphism, an important topic for more advanced C++ programmers, a good source is **{2}** (Vandevoorde, Josuttis, Gregor (2E))

Polymorphism: A Review in a Modernized Context

A common example of polymorphism is of a base `Shape` class from which two concrete shape classes, such as `Circle` and `Rectangle` are *derived* (or *inherit*). Along the way, we can also observe some language features, including some post-C++11, that should be preferred (Kalb, **{1}**).

To start, suppose our `Shape` class has two *virtual functions* **{3}**. The first, `what_am_i()`, will return a string containing the name of a particular shape, namely in this case "Circle" or "Rectangle", or simply "Shape" if called on a base object.

```
class Shape
{
public:

    virtual std::string what_am_i() const;          // Returns "Shape"
    virtual double area() const = 0;
```

```
    virtual ~Shape() = default;
};
```

The `area()` function is *pure virtual*, indicated by the assignment to `0` in the class declaration. This sets up a contract that mandates its implementation on derived classes while having no implementation of its own on the base class. This makes the `Shape` an *abstract base class* that cannot be instantiated without a derived object.

Figure 3-1. Abstract Base Class and Inheritance

As a result, an abstract base class prevents a pesky problem that can arise in object-oriented programming, namely object slicing **{4}**.

Note that a virtual default destructor is also defined, using the `default` keyword. This ensures destructors on derived objects are also called when the destructor on a base object is invoked.

The declaration for the `Circle` class can be written as follows, with new keywords to be explained in context:

```
class Circle final : public Shape
{
```

```
public:
    Circle(double radius);
    double area() const override;
    std::string what_am_i() const override;          // Returns "Circle"

private:
    double radius_;
};
```

Similarly, the declaration for `Rectangle` follows:

```
class Rectangle final : public Shape
{
public:
   Rectangle(double length, double width);
   double area() const override;
   std::string what_am_i() const override;       // Returns "Rectangle"

private:
   double length_, width_;
};
```

Note here the presence of the C++11 `final` and `override` keywords. The `final` keyword is essentially the same thing as `final` in Java or `sealed` in C#, namely ensuring a class with this designation prevents another class inheriting from it. Any attempt to do so will result in a compiler error. It should also be noted the `final` keyword can help improve performance through "devirtualization" **{5}**.

The `override` keyword explicitly indicates a function is overriding a virtual function of the same name. Suppose in the `Circle` class above, the `what_am_i()` function name was intended to override the virtual function on the base class but was mistakenly written as `what_i_am()`, with no `override` keyword.

```
class Circle final : public Shape
{
public:
        Circle(double radius);
        double area() const override;

        std::string what_i_am() const;          // Problem!  Should be what_am_i()

private:
        double radius_;
};
```

Suppose further a `Circle` instance is created, and we call the `what_am_i()` member function:

```
Circle circle{1.0};

cout << circle.what_am_i() << "\n";
```

The code would compile and run, but the output to the screen would be `Shape` rather than `Circle`. In contrast, if the override keyword were present:

```
std::string what_i_am() const override;
```

a compiler error would be generated, saying there is no virtual function `what_i_am()` to override, which is more desirable than unexpected behavior which can cause serious problems in more complex real-world software applications. The results would be similar for the `Rectangle` class.

For completeness, the implementations for each class are as follows:

```
// Shape implementation
std::string Shape::what_am_i() const
{
	return "Shape";
}

// Circle implementation
Circle::Circle(double radius) :radius_{ radius } {}

double Circle::area() const 			// override
{
	return std::numbers::pi * radius_ * radius_;
}

std::string Circle::what_am_i() const  // override
{
	return "Circle";
}


// Rectangle implementation
Rectangle::Rectangle(double length, double width) :
		length_{ length }, width_{ width } {}

double Rectangle::area() const 			// override
{
	return length_ * width_;
}

std::string Rectangle::what_am_i() const  // override
{
```

```
        return "Rectangle";
    }
```

Handling Polymorphic Member Resources with Raw Pointers

The second edition of the book *C++ Design Patterns and Derivatives Pricing*, by Mark Joshi **{6}**, was published in 2008. For its time, it presented an easily comprehendible and straightforward example where a polymorphic payoff object was pointed to from an enclosing option contract object, with two specific goals in mind. First, the actual payoff -- call or put -- would not need to be known until runtime. Second, exclusive ownership of the pointer member would be retained by the enclosing option object. This design obviated the need for an `if ... then` condition to set the payoff, and prevented modififying or deleting the payoff object from outside the option object. This resulted in cleaner and more reliable code.

Sadly, Joshi passed away at a young age in 2017 **{7}** (Wilmott Magazine). His contributions to the financial field were substantial, with publications of two additional books on mathematical finance, over 70 published research papers, and development of practical open source utilities for C++ financial developers. Although this last edition of his C++ design patterns book was published several years prior to C++11, it remains a very useful text for financial programmers wishing to better understand object-oriented design patterns in C++.

We will revisit his example of designing an equity option class with a payoff member whose type is determined at runtime, using a method known as "virtual construction" **{6}** (Sec 4.3). This resulting object would then provide all the information necessary as an argument in pricing models that rely on numerical approximations such as Monte Carlo simulation and binomial lattices (both of which will be discussed in subsequent chapters), or perhaps other alternatives such as a model based on the numerical solution of a partial differential equation (PDE). Later on in the present chapter, we will then update this example with

C++11 features that were not yet available at the time of his book's publication that can improve upon the "virtual construction" approach, namely using *smart pointers*.

The Problem

The problem we wish to solve is as follows. We first have an abstract `Payoff` base class from which are derived a `CallPayoff` and `PutPayoff` class, as shown in the following declaration. To reiterate here, we are assuming we are in the pre-C++11 world where we do not yet have smart pointers available. However, so as not to detract from the main point, we will assume we do have the `default` and override keywords.

```
class Payoff
{
public:
	virtual double operator()(double price) const = 0;
	virtual ~Payoff() = default;	// Assume we have the default keyword
};

class CallPayoff : public Payoff
{
public:
	CallPayoff(double strike);
	double operator()(double price) const override;

private:
	double strike_;
};

class PutPayoff : public Payoff
{
public:
	PutPayoff(double strike);
```

```
        double operator()(double price) const override;          // Assume we have the override keyword

    private:
        double strike_;
    };
```

The `VanillaOption` class will need to hold the time to expiration in units of years (or a year fraction), as well as handle a `Payoff` member. This second task is where the problem lies. To manage a polymorphic member, it will require indirection, meaning that it will need to be either a (`const`) reference or a pointer. In the case of `const` reference, we will run into the same problems of shallow copy and possible modification of the object outside of the enclosing object (see Chapter Two). In the case of a pointer, we might naively have chosen to just take in a pointer to a `Payoff` as a constructor argument and assign it to the `payoff_ptr_` member:

```
    class VanillaOption
    {
    public:
        VanillaOption(Payoff* payoff, double time_to_exp);      // This will be a problem

        double option_payoff(double spot) const;       // Will need for numerical pricing approximations
       double time_to_expiration() const;              // Will need for numerical pricing approximations

    private:
        Payoff* payoff_ptr_;
        double time_to_exp_;
    };
```

In this case, however, we will not only have the same issues as with a `const` reference member, but also things could go very wrong if the pointer gets deleted from outside the active enclosing object. For

example:

```
Payoff* call_payoff = new CallPayoff{25.0};          // Call with strike = 25
VanillaOption van_opt{call_payoff, 0.5};         // Time to exp = six months

delete call_payoff;                                                  // OOPS!
```

Now, `van_opt` points to deleted memory on the heap. This is not good.

Another issue is if the memory that is allocated and pointed to by `call_payoff` is never cleaned up before `van_opt` goes out of scope when a function using it is exited:

```
void f(.)
{
      Payoff* call_payoff = new CallPayoff{25.0};          // Call with strike = 25
      VanillaOption van_opt{call_payoff, 0.5};

      . . .

      // Function exits, delete never called

}
```

In this case, we get a memory leak, and/or unexpected behavior. This is also not good.

The (Pre-C++11) Solution

What we would like to have is for an enclosing `VanillaOption` object to have exclusive control over a `Payoff` pointer member, starting with its acquisition, preventing its external modification during its life-

time, and then properly deallocating the memory pointed to by `payoff_ptr_` before being destroyed itself.

Ultimately, we wish to assemble a `VanillaOption` with a `Payoff` member *resource* that can be passed to a numerical pricing function or object. A resource, as described in Scott Meyers *Effective C++*, "is something that, once you're done using it, you need to return to the system. If you don't, bad things happen. In C++ programs, the most commonly used resource is dynamically allocated memory (if you allocate memory and never deallocate it, you've got a memory leak)...." **{8}**

In our case, the resource is the `payoff_ptr_` member that will point to either a `CallPayoff` or `PutPayoff` object in memory allocated on the heap **{9}**. The way it will work is to use the *Resource Acquisition Is Initialization (RAII)* technique **{10}**, which essentially says:

1. Initialize the `payoff_ptr_` resource with a pointer to a call or put payoff object exclusively acquired in heap memory.
2. Use this resource during the lifetime of the enclosing `VanillaOption` object, without concern it could be modified or deleted externally.
3. In the destructor, call the `delete` operator to deallocate the memory pointed to by `payoff_ptr_`, thus avoiding problems due to memory leaks and/or unexpected behavior. So in this case, we will need to add our own implementation of the destructor; we cannot just use the default.

Cloning the Payoff

Resource acquisition in pre-C++11 form could be accomplished by adding a virtual `clone()` method to each payoff object. It creates a copy of itself (`*this`) in heap memory and returns a pointer to it. This pointer then becomes the resource on the enclosing `VanillaOption` object. The declaration of the base class and the `CallPayoff` class now become:

```
class Payoff
{
public:
    virtual double operator()(double spot) const = 0;
    virtual Payoff* clone() const = 0;
    virtual ~Payoff() = default;

};

class CallPayoff final : public Payoff
{
public:
    CallPayoff(double strike);
    virtual double operator()(double spot) const;
    virtual CallPayoff* clone() const;

private:
    double strike_;
};
```

The `clone()` implementation then returns a pointer to a copy of the active object, `*this`:

```
CallPayoff* CallPayoff::clone() const
{
    return new CallPayoff{*this};
}
```

Both the declaration and the implementation for `PutPayoff` are similar.

Creating an Instance of `VanillaOption`

To employ RAII with a `VanillaOption`, instead of taking in a pointer to a `Payoff`—as was naively attempted in the previous declaration—the constructor will be replaced with one that will take in a `const` reference argument:

```
VanillaOption(const Payoff& payoff, double time_to_exp);
```

Initialization in the constructor implementation will call the `clone()` virtual function from the reference to whichever payoff type argument is present, and then initialize a pointer to the result:

```
VanillaOption::VanillaOption(const Payoff& payoff, double time_to_exp):
    payoff_{ payoff.clone() }, time_to_exp_{ time_to_exp } { }
```

In this case, the `VanillaOption` object will have *exclusive control* over its `payoff_ptr_` resource, as a copy of the derived `payoff` argument has been created in heap memory, and the only pointer to it is *private*. This way, the resource can not be modified or deleted outside of the active `VanillaOption` object.

The payoff of the option can then be easily obtained by providing an accessor that calls the `()` operator on the dereferenced `payoff_ptr_`, given the spot price of the underlying security as its input parameter.

```
double VanillaOption::option_payoff(double spot) const
{
    return (*payoff_ptr_)(spot);
}
```

Returning the time to expiration on the other hand is trivial:

```
double VanillaOption::time_to_expiration() const
{
    return time_to_exp_;
}
```

Now, a `VanillaOption` can be constructed and used inside a numerical option pricing model such as a binomial lattice **{11}**, where the payoff and time to expiration will be required at each node. Other market data such as volatility, the dividend rate, and the risk-free interest rate, and the number of time steps for the lattice are also input into the constructor of, say, a `BinomialLattice` object, as outlined in the following declaration. The price is then obtained from the `calc_price(.)` member function that takes in the underlying spot price and the option type (eg European or American, as a scoped enum). The example that follows is a high level sketch just using a `BinomialLattice` class declaration. A full implementation of a binomial lattice model will be covered in Chapter Eight.

```
enum class OptType
{
    European,
    American
};


class BinomialLattice
{
    public:
        BinomialLattice(const VanillaOption& opt, double vol, double int_rate,
            unsigned time_steps, double div_rate);

        double calc_price(double spot, OptType opt_type);
};
```

The same `VanillaOption` class could then also be reused in other numerical pricing models, such as Monte Carlo methods or a PDE method. A full implementation of a Monte Carlo model will be presented in Chapter Five. The primary goal of this chapter, as hopefully seen so far, is developing the assembly of an option with its payoff which can then be passed to the desired numerical pricing method, as shown in the following diagram.

Figure 3-2. High Level Object Model

That is, we will concentrate on designing and composing the code in the red oval. This will ultimately be updated using features now available in modern C++ and then reused in the models developed in Chapters Five and Eight.

Example

Putting it all together in an example, we can create a `CallPayoff` object with a strike of, say, 75. We can then use it to construct a `VanillaOption`, with, for example, half a year remaining until expiration.

```
class BinomialLattice
{
      public:
            BinomialLattice(const VanillaOption& opt, double vol, double int_rate,
                  unsigned time_steps, double div_rate);
```

```
        double calc_price(double spot, OptType opt_type);
};

CallPayoff call_payoff{ 75.0 };
VanillaOption call_opt{ call_payoff, 0.5 };
```

Then, the value of an American call option with a strike of 75 and six months until expiration could then be calculated using a binomial lattice model by providing it with the market data arguments for the annual volatility and risk-free rates, the number of time steps desired for the lattice, and the annual continuous dividend rate of the underlying equity. The `calc_price(.)` member function is responsible for the valuation, given the market spot price for the underlying equity and option type:

```
BinomialLattice binomial_lattice{ call_opt, 0.25, 0.04, 100, 0.02 };
double opt_val = binomial_lattice.calc_price(85.0, OptType::American);
```

There are two points to note here.

1. The state of the payoff cannot be manipulated outside of `call_opt`, and the `payoff_ptr_` resource is guaranteed to persist until `call_opt` is destroyed (RAII).
2. The `BinomialLattice` constructor doesn't care whether the option is a call or a put—the `call_opt` argument bound to the `VanillaOption` parameter is solely responsible for the payoff calculation, which will be invoked inside the lattice model.

This avoids tight coupling of the classes representing the payoff, the option, and the pricing model, which in turn allows for code reuse, one of the most important fundamental motivations for utilizing object-oriented programming.

Implementing the `VanillaOption` Class Destructor

There is one more important and final step to cover to complete the RAII technique. Consider what would happen if a `VanillaOption` object were to go out of scope in its current form. Its default destructor would be called, but the memory containing the payoff member would not be deallocated. For this reason, a user-defined destructor that calls `delete` on the `payoff_ptr_` member becomes necessary. There is not much involved, but failure to implement it could evenutally lead to problems associated with insufficient heap memory, or potentially worse yet, unexpected behavior. The declaration and implementation of the destructor are shown here:

```
// Declaration of the destructor:
public:
      . . .
      ~VanillaOption();

// Destructor implementation:
VanillaOption::~VanillaOption()
{
      delete payoff_ptr_;
}
```

Copy Operations on `VanillaOption`

When coding financial models, a lot of the time it will probably not be necessary to copy an object. As alluded to in the previous example, a derivative pricing model might get called just once, calculate the price and associated risk values, and then go out of scope. If a trader needs to revalue as asset, a new object just gets created.

If for some reason a copy were attempted, however, the default copy constructor would be called, resulting in a shallow copy sharing the same `Payoff` resource. One solution is just to disable the copy constructor and copy assignment operator. This is easily accomplished given our assumption we do have the `delete` keyword from C++11 available:

```
class VanillaOption
{
public:
        VanillaOption(const Payoff& payoff, double time_to_exp);
        VanillaOption(const VanillaOption& rhs) = delete;
        VanillaOption& operator =(const VanillaOption& rhs) = delete;

        ~VanillaOption();

        double option_payoff(double spot) const;
    double time_to_expiration() const;

private:
        Payoff* payoff_ptr_;
        double time_to_exp_;
};
```

In this case, the compiler will prevent any attempt to copy a `VanillaOption` object, whether by the copy constructor or copy assignment operator.

There can be other cases, however, where we might need to take a copy of an object, such as in a trade entry or risk system. In order to assure a *deep copy*, **{12}** we will need to write our own implementations of the two copy operations. To start, this means we need to declare them, sans the `delete` assignments:

```
VanillaOption(const VanillaOption& rhs);
VanillaOption& operator =(const VanillaOption& rhs);
```

Implementation of the copy constructor would then need to initialize each data member in the copied object. Copying `time_to_exp_` would be trivial, but we again would need to create a copy of the `Payoff` member on the heap and define a new pointer to it. Most of this work is done already in the virtual `clone()` method on each derived payoff object type:

```
VanillaOption::VanillaOption(const VanillaOption& rhs):
    payoff_ptr_{ rhs.payoff_ptr_->clone() }, time_to_exp_{ rhs.time_to_expiration() } {}
```

The copy assignment operator implementation is similar, except that instead of initializing each member, we use the individual assignment operators. We also first need to check if a copy of the same object is being attempted; otherwise, it would delete its own `payoff_ptr_` member:

```
VanillaOption& VanillaOption::operator =(const VanillaOption& rhs)
{
    // Need to check if attempting to copy same object:
    if (this != &rhs)
    {
        time_to_exp_ = rhs.time_to_expiration();
        delete payoff_ptr_;
        payoff_ptr_ = rhs.payoff_ptr_->clone();
    }
    return *this;
}
```

The (Old) Rule of Three

The above illustrates an example of the "Rule of Three" that was common practice prior to C++11. It says if any one of a destructor, copy constructor, or copy assignment operator is implemented, then we should implement the other two (or alternatively just disable the copy operations). The implication is that if any of these has been implemented, there is probably a member resource that needs special attention, in particular a raw pointer. In this case, all three would need to be covered in order to prevent memory leaks and/or shallow copies. This is the predecessor of the "Rule of Five" that came into being after the release of C++11, with the extra two rules pertaining to the move constructor and move assignment operator. This will be discussed after a short prerequisite digression introducing smart pointers.

NOTE

Handling a data member pointed to allocated memory with a clone method as described above simulates what is called "virtual construction", as part of a textbook object-oriented design pattern called an *abstract factory pattern*.

The classic foundational text, *Design Patterns*, by Gamma et al **{13}**, contains an in-depth presentation of this pattern. A subsequent text, Design Patterns Explained (Shalloway and Trott) **{14}**, is also a good source.

Introducing Smart Pointers

Smart pointers that eliminate a lot of the risks associated with raw pointers and heap memory allocation were introduced in C++11. One of these, the unique pointer (`std::unique_ptr`), assures there can be only one pointer to a particular memory location at one time. It is an example of a very low-cost abstraction, as there is little to no appreciable difference in performance compared to a conventional raw pointer. Possession of a unique pointer is transferred using move semantics.

Along with unique pointers, two other smart pointers, namely shared pointers and weak pointers, were introduced in C++11. These also help prevent problems associated with standard raw pointers when used to allocate and manage memory. In fact, as stated in the Core Guidelines(C.149), a unique or shared pointer should be preferred over a raw pointer when possible in order "to avoid forgetting to delete objects" **{15}**.

Our discussion of smart pointers will primarily focus on unique pointers, and they will be the primary go-to smart pointers for this book. In this chapter, we will discuss how to swap them in for the raw pointers in the previous example.

We will also have a look at shared pointers, but keep in mind these should only be used where shared ownership of a resource is required, such as with an object that provides live updated market data. Shared pointers are also used in multithreading situations where a pointed-to object is shared by at least two threads and where you don't know which one will be responsible for the object's destruction.

A weak pointer is used for breaking cyclic dependencies that can arise where the reference count of a shared pointer never reaches zero. This is a more advanced topic that is only mentioned here for completeness.

Unique Pointers

To recap, a unique pointer has sole ownership of a pointed-to object. It cannot be copied, but possession can be transferred with move semantics. The Core Guidelines in fact recommend using a unique pointer to hold pointers as a general rule. More specifically, using a unique pointer "is the simplest way to avoid leaks. It is reliable, it makes the type system do much of the work to validate ownership safety, it increases readability, and it has zero or near zero run-time cost." **{16}**

A unique pointer is defined by a class template in the Standard Library, indicated by `std::unique_ptr<T>`, with template parameter `T`. For introductory examples, consider a simple case of pointers to `string` ob-

jects. The preferred method to create a unique pointer is to use function template `std::make_unique<T>`, as shown below. As in the case of a raw pointer, this can be done in two lines, or combined in one line (typically preferred) using `auto` for implicit typing, as the right-hand side of the assignment makes it obvious what is being created. The Standard Library `<memory>` header needs to be included to use smart pointers.

```
#include <memory>     // For Standard Library smart pointers
#include <utility> // std::move

std::unique_ptr<std::string> ptr1;
ptr1 = std::make_unique<std::string>("Climb in your old umbrella");

// Alternatively, on one line:
auto ptr2 =
      std::make_unique<std::string>("Does it have a nasty tear in the dome?");
```

The actual string object can be obtained by dereferencing the unique pointers in the usual way, with the `*` operator:

```
std::cout << std::format("Contents of ptr1: {}", *ptr1) << std::endl;
std::cout << std::format("Contents of ptr2: {}", *ptr2) << std::endl;
```

Next, we can transfer ownership of the allocated memory to a different unique pointer, say `ptr3`. This is accomplished using move semantics:

```
auto ptr3 = std::move(ptr2);
```

The unique pointer `ptr2` is now set to a null state, as `ptr3` now assumes exclusive ownership of the second `std::string` object. Attempting to output `*ptr2` or use it elsewhere can result in a runtime error, and possibly even crash your program, due to unexpected results slipping through the cracks undetected. Although unique pointers provide a safe yet efficient means of managing allocated heap memory, this shows that you still need to exercise caution when using them. If there ever is any possible doubt, you can check whether a pointer is indeed pointing to a memory block by checking whether it is null or not. This can be done the same as you would with a raw pointer:

```
if (ptr2)
{
      cout << format("ptr2 is not null")
            << "\n";
}
else
{
      cout << format("ptr2 is null")
            << "\n";
}
```

Attempting to dereference `ptr2` without checking first, eg:

```
auto s = *ptr2;                  // ptr2 is null -- DON'T do this!
```

can result in runtime problems as noted above, although depending on your compiler settings, you might at least get a compile-time *warning* first.

Assigning new data to the pointed-to memory is also the same as with raw pointers, by dereferencing the pointer first:

```
*ptr1 = "Your shirt's all dirty";
*ptr3 = "There's a man here from the BBC";
```

Finally, if a unique pointer is no longer needed before going out of scope, it can be deleted and set to null explicitly with its `reset` member function.

```
ptr1.reset();
if (!ptr1)
{
	cout << "ptr1 is null..." << "\n";
}
```

It should be noted, however, that if you find yourself resetting a unique pointer before a function exits, it might be a good indication that your function is too long.

Shared Pointers

In contrast to a unique pointer—which, as implied by its name, is the *uniquely* responsible for a pointed-to object in heap memory—a shared pointer can share this responsibility with another shared pointer, similar to a raw pointer. It is a safer alternative because contents of the memory will not be deleted until the last remaining shared pointer goes out of scope. This is accomplished behind the scenes by *reference counting* which keeps count of the number of active shared pointers to a common block of heap memory. They will persist until the count reaches zero, and only at this point the memory will be deallocated, thus preventing a memory leak and possible unexpected results.

Similar to a unique pointer, a shared pointer is created by invoking the `make_shared` function. This can again be performed in two separate lines, or on one line (again, typically preferred), as shown in the ex-

ample code here:

```
std::shared_ptr<std::string> sp_one;
sp_one = std::make_shared<std::string>("This city drains me");

// Single line
auto sp_two =
      std::make_shared<std::string>("Maybe it's the smell of gasoline");
```

Again like a raw pointer, the pointed-to objects can be accessed by deferencing:

```
cout << format("Contents of ptr1: {}", *sp_one) << "\n";
cout << format("Contents of ptr2: {}", *sp_two) << "\n\n";
```

To see how reference counting works, we can first call the `use_count()` member function on `shared_ptr`. This will return a value of 1, as at this stage there is only one pointer pointing to the `string` object in heap memory.

```
cout << format("How many pointers are there to sp_one? {}",
      sp_one.use_count()) << "\n";
```

With a shared pointer, we can create multiple pointers to the same object, similar to a raw pointer:

```
auto sp_one_2 = sp_one;
auto sp_one_3 = sp_one_2;
```

Now, `use_count()` will return a value of 3. Unlike a raw pointer, however, there is no `delete` operator. Instead, if the `reset()` function is called on `sp_one_2`, then that pointer becomes null, and the reference count will fall to 2.

```
sp_one_2.reset();

long count = sp_one.use_count();          // = 2
```

If the code examples here for `shared_ptr` were inside a function, then when `sp_one` and `sp_one_3` go out of scope at the end of the function block, both are destroyed, and the memory occupied by the pointed-to `string` object is deallocated as the reference count will hit zero.

For an example using a shared pointer in a financial setting, common commercial trading system software allows a trader to enter trades, monitor and revalue positions, and run risk analysis on trading books. Each of these tasks, visualized on the screen in separate client windows, can be open concurrently, and multiple trade screens might be open at the same time. They will also typically rely on live market data, provided by an internal service or external vendor, updated in real-time from a price feed.

The same piece of market data might be used in multiple places, and you don't want to delete the memory until all the various windows have finished using the data. Several windows might use market data in a trading system, including for instance a daily chart (eg a graph of stock price over time), an order entry window which might show the level 1 price (top of book) and level 2 (depth of book) prices, also possibly a position window with profit/loss (P&L) value calculations, and/or a risk checker that might be configured have a risk rule to prevent limit orders from being sent with limit prices which are too far away (eg >10% away) from the current market price.

A shared pointer might also point to a trade execution. Trading systems would display and calculate with partial execution records (aka *partial fills*) for an order. A few different windows would display this infor-

mation in different ways:

- The average price of each order, which needs to be recalculated from the child orders each time a new partial fill comes in
- a list of the executions themselves, so the trader can drill down on the parent order if he wishes to see how and when it was filled
- a positions window, which keeps a running total of the user's position, typically aggregated by symbol and account number
- An execution for an order can also be canceled after it has happened, if it were deemed incorrect for some reason and would need to be reversed out of the displays and calculations mentioned in the points above

In summary, although shared pointers provide the very significant safety advantages noted here, there are two caveats to consider. First, there will be a greater performance overhead compared to raw pointers and unique pointers, so some cost/benefit analysis would be in order rather than proceeding naively. One more point is that shared pointers are also often found in multithreaded applications where data is no longer just read-only, and the underlying object is shared by two or more threads. In general, certain precautions need to be in place when working with multithreaded code, and in particular, conditioning on the value of `use_count()` is rarely usable in this case. Details on proper use of shared pointers in multithreaded applications can be found in **{17}**.

Replacing Raw Pointers with Unique Pointers in Class Design

Returning now to the case of a `VanillaOption` object with a `Payoff` member resource, we can build upon the preceding C++98 solution by replacing the raw pointer member with a unique pointer, facilities that did not exist in the Standard before 2011. This will provide a couple of advantages. First, instead of call-

ing the `clone()` method that generates a copy of the polymorphic payoff object, we can improve performance by just moving a unique pointer at construction of the option object. Second, although RAII is generally safe for managing a raw pointer member resource, there are still things that can go wrong, such as forgetting to call `delete` in the destructor.

We will go through this in the same sequence as with the C++98 era "virtual constructor" method previously discussed, namely starting with the case where we will not require copies, and then where we will need to implement a copy constructor and copy assignment operator.

Case 1:

To start, if a copy of a `VanillaOption` object is not a requirement, we will not need a `clone()` method for the payoff class definitions (we will need to bring it back for Case 2, however). So for now, the payoff class declarations become simpler:

```
class Payoff
{
public:
      virtual double operator()(double price) const = 0;
      virtual ~Payoff() = default;
};

class CallPayoff final : public Payoff
{
public:
      CallPayoff(double strike);
      double operator()(double spot) const override;

private:
      double strike_;
};
```

```
class PutPayoff final : public Payoff
{
public:
    PutPayoff(double strike);
    double operator()(double spot) const override;

private:
    double strike_;
};
```

The implementation for the call payoff is the same as before, but sans the `clone()` function. All we are left with is the payoff functor:

```
CallPayoff::CallPayoff(double strike) :strike_{ strike } {}

double CallPayoff::operator()(double spot) const
{
    return std::max(spot - strike_, 0.0);
}
```

The put payoff is similar, except for the spot and strike prices being reversed.

The changes to the `VanillaOption` class, however, are significant. Note that instead of the raw pointer resource for the payoff, a unique pointer to the `Payoff` base class is used as the constructor argument for a call or put derived type, and that it is stored as a `unique_ptr<Payoff>` data member. Move semantics will be used for initialization in the implementation that follows, obviating object copy while still ensuring the enclosing object has exclusive (unique) control over its resource. Note also we don't need to implement the destructor, as there is nothing for us to clean up—`unique_ptr` takes care of memory de-

allocation for us when a `VanillaOption` object goes out of scope. Its compiler-provided default destructor is sufficient now, and thus we eliminate problems associated with forgetting to delete a raw pointer.

```
class VanillaOption
{
public:
        VanillaOption(std::unique_ptr<Payoff> payoff, double time_to_exp);
        double option_payoff(double spot) const;
        double time_to_expiration() const;

private:
        std::unique_ptr<Payoff> payoff_ptr_;
        double time_to_exp_;
};
```

The constructor implementation takes in the payoff as a unique pointer and initializes it with `std::move`:

```
VanillaOption::VanillaOption(std::unique_ptr<Payoff> payoff, double time_to_exp) :
        payoff_ptr_{ std::move(payoff) }, time_to_exp_{ time_to_exp } {}
```

The `payoff(.)` method is exactly the same, as we can dereference a unique pointer just as we do with a raw pointer:

```
double VanillaOption::option_payoff(double spot) const
{
        return (*payoff_ptr_)(spot);
}
```

So now, we can construct a `VanillaOption` without generating a copy of the payoff object as we did previously. A more efficient move operation is used instead:

```
auto call_payoff = std::make_unique<CallPayoff>(75.0);
VanillaOption call_opt{ std::move(call_payoff), 0.5 };
```

Also notable is we don't have to worry about shallow copies either, as the compiler prevents an attempt to copy an object with a `unique_ptr` member, which is again a good thing.

```
VanillaOption call_copy{ call_opt };        // Will not compile!
```

On the other hand, because a unique pointer is movable, moving a `VanillaOption` object is completely legal, by simply using the default move constructor or move assignment operator.

```
VanillaOption call_move{ std::move(call_opt) };
VanillaOption call_move_assgn = std::move(call_move);
```

So, replacing the raw pointer payoff resource with a unique pointer, we get:

- More efficient code, as object cloning is not necessary for initialization of the payoff (replaced by *moving* a unique pointer)
- Safer code
- Avoidance of unexpected results and memory leaks associated with raw pointers
- Prevention of shallow copies at compile-time.
- More maintainable code with less to go wrong
- The enclosing `VanillaOption` object having exclusive control of its `Payoff` resource *
- No implementation of the destructor required

- No implementation of the move constructor or move assignment required—just use the compiler-provided defaults

Case 2: Copy Operations Required

As mentioned previously, there are cases where copy operations on a class are not necessary, and in some cases they might just be disabled for performance reasons. On the other hand, there are times when copying is useful. Consider again a trading system, where traders might be entering multiple orders for the same security and possibly the same client. In these cases, traders will prefer to copy an existing order, and then enter only the remaining data rather than create a whole new order and re-enter all the information.

Clone Methods, Revisited

While cloning the payoff remains unnecessary at construction of a `VanillaOption`, it will still be necessary for copy operations, which probably comes as no surprise. So, for the class declarations, we have kind of come full circle to where we were before, requiring virtual `clone()` methods again. There will be one semantic difference, namely that its return type of the payoff resource will be a `unique_ptr<Payoff>` rather than a raw pointer to a particular derived payoff type (`CallPayoff`, `PutPayoff`). Retooling the declarations leads us to updated versions such as the following:

```
class Payoff
{
public:
      virtual double operator()(double price) const = 0;
      virtual std::unique_ptr<Payoff> clone() const = 0; //    clone() now returns a unique_ptr
      virtual ~Payoff() = default;
};
```

```
class CallPayoff final : public Payoff
{
public:
    CallPayoff(double strike);
    double operator()(double spot) const override;
    std::unique_ptr<Payoff> clone() const override;        // Not unique_ptr<CallPayoff>

private:
    double strike_;
};

class PutPayoff final : public Payoff
{
public:
    PutPayoff(double strike);
    double operator()(double spot) const override;
    std::unique_ptr<Payoff> clone() const override;        // Not unique_ptr<PutPayoff>

private:
    double strike_;
};
```

The payoff functor implementations remain the same, but the `clone()` method for a call payoff can now be written as follows, with the put payoff similar:

```
std::unique_ptr<Payoff> CallPayoff::clone() const
{
    return std::make_unique<CallPayoff>(*this);        // replaces "new CallPayoff{ *this }"
}
```

This allows us to update `VanillaOption` with a copy constructor and copy assignment operator, as indicated in its expanded declaration:

```
class VanillaOption
{
public:
        VanillaOption(std::unique_ptr<Payoff> payoff, double time_to_exp);
        double option_payoff(double spot) const;
        double time_to_expiration() const;

        VanillaOption(const VanillaOption& rhs);                    // Copy constructor
        VanillaOption& operator =(const VanillaOption& rhs);  // Copy assignment

private:
        std::unique_ptr<Payoff> payoff_ptr_;
        double time_to_exp_;
};
```

Implementation of the copy constructor and copy assignment operator are pretty much painless as they are almost identical to the previous C++98 versions, with the new `clone()` method swapped in:

```
VanillaOption::VanillaOption(const VanillaOption& rhs) :
        payoff_ptr_{ rhs.payoff_ptr_->clone() }, time_to_exp_{ rhs.time_to_expiration() } {}

VanillaOption& VanillaOption::operator =(const VanillaOption& rhs)
{
        // Need to check if attempting to copy same object:
        if (this != &rhs)
        {
                time_to_exp_ = rhs.time_to_expiration();
                payoff_ptr_ = rhs.payoff_ptr_->clone();
```

```
        }
        return *this;
    }
```

The only difference with the C++98 version of the copy assignment operator is we no longer need to call the `delete` operator on the existing `payoff_ptr_` member before reassigning it to the clone of the member on the `VanillaOption` object being copied.

So although this took a little work, we have gained some nontrivial benefits in cases where copy operations are required. Our code is safer, simpler, and easier to maintain.

The Rule of Zero/The Rule of Five

Recall from the Case 1 version of `VanillaOption`, we discussed The Rule of Three, which said if any one of the copy constructor, copy assignment operator, or destructor is implemented, then user-defined versions of the remaining two should also be present (including the option of disabling the copy constructor/assignment operator). With the addition of move constructors and move assignment operators in C++11, this means there are two more special functions that need to be considered.

The fact is, however, it becomes more complex, with default behavior dependent on which of the five is/or implemented:

- If either the copy constructor or copy assignment operator is implemented (or both), the compiler-provided default move operations are disabled.
- If the destructor is implemented, we get the same result.

Even worse, the code will compile and run, but move operations will be replaced by their copy counterparts, meaning the optimizations from move semantics will be voided. Some compilers may issue a warning, but others do not, so this is another case of a problem possibly going undetected.

For these reasons, the Rule of Five supplants the Rule of Three, as follows: If any one of these five special functions has a user-defined implementation, then include definitions for the remaining four. This can include defining them as `default` or `delete`.

So in order to assure move operations are made available for `VanillaOption`, we would need to explicitly set the move operations to `default`, and for good measure the destructor as well:

```
class VanillaOption
{
public:
        VanillaOption(std::unique_ptr<Payoff> payoff, double time_to_exp);
        double option_payoff(double spot) const;
        double time_to_expiration() const;

        VanillaOption(const VanillaOption& rhs);
        VanillaOption& operator =(const VanillaOption& rhs);

        // Remaining Rule of Five:
        VanillaOption(VanillaOption&& rhs) = default;                   // Default move constructor
        VanillaOption& operator =(VanillaOption&& rhs) = default;       // Default move assignment
        ~VanillaOption() = default;                                     // Default destructor

private:
        std::unique_ptr<Payoff> payoff_ptr_;
        double time_to_exp_;
};
```

There are two corollaries to the Rule of Five in the Core Guidelines that are also germane here, taken from the Core Guidelines:

1. **If you can avoid defining default operations, do.** . This was the case for the `VanillaOption` version where no user-defined copy operations were implemented. This is called the *Rule of Zero*, and it should be preferred when possible. (Core Guidelines) **{18}**
2. **If you `define` or `delete` any copy, move, or destructor function, `define` or `delete` them all.** The explanation in this guideline provides the best explanation for the reason: *The semantics of copy, move, and destruction are closely related, so if one needs to be declared, the odds are that others need consideration too. Declaring any copy/move/destructor function, even as `default` or `delete` , will suppress the implicit declaration of a move constructor and move assignment operator. Declaring a move constructor or move assignment operator, even as `=default` or `=delete` , will cause an implicitly generated copy constructor or implicitly generated copy assignment operator to be defined as deleted. So as soon as any of these are declared, the others should all be declared to avoid unwanted effects like turning all potential moves into more expensive copies, or making a class move-only.* (Core Guidelines) **{19}**

Two informative references that provide more details are posts on Jonathan Boccara's Fluent C++ blog site **{20}**, and Rainer Grimm's *Modernes C++* **{21}**.

Using the Result in a Pricing Model

For the Case 1 example using a "virtual constructor", we concluded with the sketch of an example using a binomial lattice pricing model. Just to provide a little variety, we can examine the case where a Monte Carlo model **{22}** is used instead, with a proposed declaration as follows (again, this is a high-level sketch; an implementation example is covered in Chapter Five, as noted previously):

```
MCOptionValuation
{
public:

    MCOptionValuation(VanillaOption opt,
        double vol, double int_rate, unsigned time_steps,
```

```
            double div_rate);

        double calc_price(double spot, int unif_start_seed, unsigned num_scenarios);

    private:
        VanillaOption opt_;

        . . .

    };
```

Note the `VanillaOption` argument type can be used as a model constructor argument just as it was with the binomial lattice case. It carries with it the information about its payoff (call or put member resource) and the time left to expiration, meaning the model is indifferent to these settings. All other market data other than the underlying spot price is again taken into the model object constructor, along with the user argument for the number of time steps for each simulation. The underlying spot argument is also again delayed until the option value is calculated with the `calc_price(.)` member function, where the remaining user arguments for the random seed and number of simulated scenarios are set.

One side note, however, is the basic Monte Carlo method cannot handle optimal early exercise, so there is no `OptType` parameter in the `calc_price(.)` member function. It does, however, require a starting seed value and the number of random equity price path scenarios. This is just a sketch of the model design for now, as the details will be taken up in Chapter Five.

As a further aside, for a Monte Carlo-based model that does evaluate optimal early exercise, the Longstaff and Schwartz model noted in Chapter One is quite often cited **{23}**.

To close this section, the common themes between the Monte Carlo and Binomial Lattice cases are that we can again use the same `VanillaOption` type—carrying only its associated payoff member and time to expiration—while remaining market data is only required at construction of a `BinomialLattice` object and

when the `calc_price(.)` member function is called. This again means we can separate out the division of labor among the different classes and reuse them within the context of pricing models, without having to duplicate code. This hopefully demonstrates a key motivation behind object-oriented programming, and why it has remained popular in financial programming.

Summary

The main focus of this chapter was to show how more modern methods—namely employing `std::unique_ptr`—can make managing a polymorphic member resource on an enclosing object safer and more efficient, as there is little to no performance penalty involved. Again, as financial software development involves the interaction of multiple components, this situation is not uncommon, and before the advent of smart pointers, it could come down to a decision of safer code vs higher performance.

Object-oriented programming remains a viable choice even as more attention in the C++ world is being devoted to functional and generic programming. And even if one is inclined to dive into generic programming with templates, some of the same high-level object-oriented programming concepts can still be relevant.

Shared pointers—the other smart pointer covered here—provide a safer alternative to a conventional raw pointer, allowing multiple pointers to the same object but with reference counting. This can help prevent the associated problems with raw pointers, namely memory leaks, dangling pointers, and unexpected behavior lurking about. There is, however, a trade-off with performance that needs to be taken into consideration, and the situation also changes when used within a multithreaded context.

References

{1} "Back to Basics" talk on object-oriented programming

{2} Vandevoorde, Josuttis, and Gregor, *C++ Templates: The Complete Guide (2E)*, (Section 18.2, pp 372-377). O'Reilly Learning: Book website: *http://tmplbook.com/*

{3} Microsoft Learn: Virtual Functions[*https://learn.microsoft.com/en-us/cpp/cpp/virtual-functions?view=msvc-170*]

{4} Stack Overflow, *What is Object Slicing?*

{5} Niall Cooling, Sticky Bits blog, Using final in C++ to improve performance, November 14, 2022

{6} Mark Joshi, *C++ Design Patterns and Derivatives Pricing* (2E), 2008, Chapter Four, pp 38-57.

{7} Mark Joshi obituary, 2017 (Wilmott Magazine)

{8} Scott Meyers, *Effective C++* (3E, Ch 3, "Resource Management")

{9} Stack Overflow, *https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap*

{10} *Resource Acquisition Is Initialization (RAII)* (put link/reference)

{11} Peter James, *Option Theory* (Wiley 2003), Chapter 7

{12} Stack Overflow, Deep Copy vs Shallow Copy

{13} Gamma, Helm, Johnson, Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1994

{14} Shalloway and Trott, Design Patterns Explained (2E)

{15} ((C.149) C++ Core Guidelines

{16} C++ Core Guidelines, Unique Pointers

{17} *C++ Concurrency in Action* (2E), Anthony Williams

{18} C.20 Avoid defining default operations, C++ Core Guidelines

{19} C.21 Rule of Five

{20} Jonathan Boccara, *Fluent C++* Rule of Zero/ Rule of Five

{21} Rainer Grimm, *Modernes C++*, Rule of Zero/ Rule of Five

{22} Peter James, *Option Theory* (Wiley 2003), Chapter 10, Sections 10.1 - 10.4

{23} Longstaff and Schwartz, Valuing American Options by Simulation: A Simple Least-Squares Approach