Chapter 2. User-Defined Classes

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 2nd 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

User-defined classes have been a mainstay in financial C++ development from the beginning, as again instruments and data, such as bonds, options contracts, rate curves etc, can be naturally represented by objects. One very useful feature that has existed prior to C++11 is the overloading of the round bracket `()` operator, which enables an object to be utilized as a *function object*, or *functor*. As we will see, this is

particularly convenient in cases where we might need to find the root of a function, such as calculating the implied volatility of a traded options contract.

A newer form of a functor, called a *lambda expression*, was introduced with C++11. Also often referred to as a *lambda function*, or just a *lambda*, one can be written "on the fly" inside other functions. Among other advantages, lambda expressions can be useful in refactoring functionality into a single location, avoiding code duplication. In other cases, they can be used to more clearly separate computational logic from iteration over a container of values (or objects). Like function objects noted above, lambdas can also be passed as function arguments. Both lambda expressions and function objects will also be key within the context of *STL algorithms*, to be covered in Chapter Four.

Move Semantics were another major addition to the C++11 Standard. As opposed to incurring possible performance hits due to initialization of an object's member data by value, move semantics allow an often significant efficiency gain by instead *transferring possession* of a constructor input argument to its respective member variable. Move semantics will also be required when working with unique pointers—another substantial improvement included in C++11—to be covered in the next chapter.

This chapter will also discuss in-class initialization of member data in a header file, the `default` and `delete` keywords, and the *three-way operator*, also commonly called the *spaceship operator*. Each of these additions in C++11 can improve reliability and maintainability in a class design.

A Black-Scholes Class

The Black-Scholes model might sometimes seem like the "Hello World!" of quantitative finance programming, but we can use it to review some important points about writing user-defined classes, as well as introduce some useful features that arrived in C++11 and after.

To start, recall the Black-Scholes pricing formula applies to calculating the value V of a (stock) option with strike price X, where the underlying equity spot price is S, with time remaining until expiration, T, in units of years (or alternatively a year fraction). A very nicely presented version that lends itself well to implementation in code can be found in **{1}** (James) and is adopted here:

$$V = \varphi\left\{Se^{-qT}N\left(\varphi d_1\right) - Xe^{-rT}N\left(\varphi d_2\right)\right\}$$

where

φ = 1 for a call option, -1 for a put option

q = the continuous annual dividend rate

r = the continuous annual risk-free interest rate

σ = the annual volatility

$N(x)$: the standard normal cummulative distribution function (CDF),

and d_1 and d_2 defined as:

$$d_1 = \frac{log\left(\frac{S}{X}\right) + \left(r - q + \frac{1}{2}\sigma^2\right)T}{\sigma\sqrt{T}}$$

$d_2 = d_1 - \sigma\sqrt{T}$

-

Representing the Payoff

Before proceeding to the class design, we can appeal to an enum class to represent a call or a put option. It is still possible to associate integer values with each possibility while still ensuring any two scoped enum types remain non-comparable. This way, we can obtain the values ± 1 "for free".

```
enum class PayoffType
{
      Call = 1,
      Put = -1
};
```

We can recover the value of ϕ by casting the scoped enum to an integer (as explained in Chapter One):

```
auto corp = PayoffType::Call;          // "corp" = "call or put"
. . .

int phi = static_cast<int>(corp);      // phi = 1
```

This value can then be used in the pricing formula.

The Class Declaration

To start, we can propose the following declaration:

```
class BlackScholes
{
```

```
public:
    BlackScholes(double strike, double spot, double rate,
        double time_to_exp, PayoffType pot);

    double operator()(double vol);

private:
    void compute_norm_args_(double vol);        // d_1 and d_2

    double strike_, spot_, rate_, sigma_, time_to_exp_;
    PayoffType pot_;

    double d1_{ 0.0 }, d2_{ 0.0 };
};
```

The constructor will take in each of the required arguments except for the volatility, which will be used by the `()` operator to compute and return the price. The reason for this is it will allow us to reuse the class later when computing the implied volatility numerically (cite Joshi **{2}** and Quantstart article **{3}** - see endnotes). In this example, we will assume there is no dividend (q = 0).

`compute_norm_args_(double vol)` will calculate the d_1 and d_2 (`d1_` and `d2_`) values that will be used as arguments in the standard normal CDF.

One more item to note is the use of *in-class member initialization*, as well as the option to place multiple definitions (or declarations) on a single line, that became available in C++11:

```
double d1_{ 0.0 }, d2_{ 0.0 };
```

With in-class member initialization available in the header, the constructor initializer list only needs to be responsible for data members that depend on the input parameters. In-class member initialization provides an additional benefit when used with a user-defined default constructor, to be discussed later in the chapter.

The Class Implementation

Starting with the constructor, we can implement it as follows:

```
BlackScholes::BlackScholes(double strike, double spot, double rate,
      double time_to_exp, PayoffType pot) :strike_{ strike }, spot_{ spot },
      rate_{ rate }, time_to_exp_{ time_to_exp }, pot_{ pot } {}
```

In general, it is good practice to ensure all member data is initialized at construction, and each member variable initialized in the order in which it is declared. Otherwise, it "can make it hard to see order-dependent bugs" **{4}** (Core Guidelines).

Valuation of the option then begins with the round bracket operator, taking in the volatility as input.

```
double BlackScholes::operator()(double vol)
{
      int phi = static_cast<int>(pot_);        // (1)

      double opt_price = 0.0;
      if (time_to_exp_ > 0.0)                  // (2)
      {
            compute_norm_args_(vol);           // (3)

            auto norm_cdf = [](double x) -> double        // (4)
```

```
            {
                return (1.0 + std::erf(x / std::numbers::sqrt2)) / 2.0;
            };

            double nd_1 = norm_cdf(phi * d1_);          // N(d1_) (5)
            double nd_2 = norm_cdf(phi * d2_);          // N(d2_) (5)
            double disc_fctr = exp(-rate_ * time_to_exp_);        // (6)

            opt_price = phi * (spot_ * nd_1 - disc_fctr * strike_ * nd_2);          // (7)
        }
        else
        {
            opt_price = std::max(phi * (spot_ - strike_), 0.0);          // <algorithm>  // (8)
        }

        return opt_price;
    }
```

The first step (1) determines the value of ϕ by casting `PayoffType::Call` or `PayoffType::Put` to 1 or -1, respectively. If there is positive time remaining ($T - t > 0$) before expiration (2), the Black-Scholes price is calculated by first (3) calculating the d_1 and d_2 values in the private helper function `compute_norm_args_(.)`:

```
    void BlackScholes::compute_norm_args_(double vol)
    {
        double numer = log(spot_ / strike_) + rate_ * time_to_exp_
            + 0.5 * time_to_exp_ * vol * vol;

        double vol_sqrt_time = vol * sqrt(time_to_exp_);

        d1_ = numer / vol_sqrt_time;
```

```
        d2_ = d1_ - vol_sqrt_time;

    }
```

The results are set on the `d1_` and `d2_` member values respectively. $N(d_1)$ and $N(d_2)$ can conveniently be determined by using the *error function* `erf(.)` that is available to us in `<cmath>`.

$$N\left(x\right) = \frac{1+erf\left(\frac{x}{\sqrt{2}}\right)}{2}$$

where

$$erf\left(z\right) = \int_0^z e^{-t^2} dt$$

_ **{5}** (Wolfram Math World)

We could implement this with brute force:

```
double nd_1 = 1.0 + std::erf( (phi * d1_) / std::numbers::sqrt2;          // N(d1)
double nd_2 = 1.0 + std::erf( (phi * d2_) / std::numbers::sqrt2;          // N(d2)
```

But instead, we can make the code a little cleaner by factoring out this expression into a *lambda expression* (4):

```
auto norm_cdf = [](double x) -> double          // (4)
{
    return (1.0 + std::erf(x / std::numbers::sqrt2)) / 2.0;
};
```

As discussed in the previous chapter, lambda expressions allow us to place what is essentially a function as a statement inside another function which helps eliminate duplicate code. It can be called just like any other function (5):

```
double nd_1 = norm_cdf(phi * d1_);          // N(d1)
double nd_2 = norm_cdf(phi * d2_);          // N(d2)
```

Finally, the discount factor from T back to time t is computed (6), followed by the Black-Scholes option price (7).

In the event the option is expired, its value is simply the raw payoff (8). Note that by using the enum class `PayoffType` and assigning its designated integer equivalent to `phi_`, this allows us to avoid several extra lines in an `if` / `else` statement.

As examples, consider an in-the-money (ITM) call function at expiration, and an out-of-the-money (OTM) put option with time remaining.

```
double strike = 75.0;
auto corp = PayoffType::Call;     // corp = "call or put"
double spot = 100.0;
double rate = 0.05;
double vol = 0.25;
double time_to_exp = 0.0;

// ITM Call at expiration (time_to_exp = 0):
BlackScholes
      bsc_itm_exp{ strike, spot, rate, time_to_exp, corp };

double value = bsc_itm_exp(vol);      // Result: 25 (payoff - intrinsic value only)
```

```
// OTM put with time remaining:
time_to_exp = 0.3;
corp = PayoffType::Put;
BlackScholes
        bsp_otm_tv{ strike, spot, rate, time_to_exp, corp };

value = bsp_otm_tv(vol);        // Result: 0.056 (time value only)
```

Using a Functor for Root Finding: Implied Volatility

One situation where functors become very convenient is in numerical analysis applications, for example root finding. A very common case is in calculating the implied volatility of an option, given its market price, as there is no closed-form solution. This is the motivation for making the functor in `BlackScholes` a function of the option volatility.

As an example, we can apply the well-known secant method to determine the root of the function

$f = V(\sigma; \phi, S, r, q, t, T) - V_m$

where V_m is the observed market price of the option. The values to the right of the semicolon can be considered fixed values (parameters in the mathematical sense), while allowing σ to vary. This was the motivation for taking in these parameters at construction of a `BlackScholes` object, and defining the functor as dependent on the volatility.

The secant method says for a function $y = f(x)$, one can find a root of f using the iteration

$x_{i+1} = x_i - f(x_i)\frac{x_i-x_{i-1}}{f(x_i) - f(x_{i-1})}$

given two initial guesses for the volatility (x_0 and x_1), provided it converges.

Because a `BlackScholes` object maintains the state of the parameters above and computes its option value in terms of volatility, this makes implementing the secant method a simpler and cleaner task, not having to bring each individual option valuation parameter into function computing the implied volatility. As an option value is monotonically increasing with respect to volatility, there can only be one root. We can then implement the secant method to locate this root, as follows:

```
double implied_volatility(BlackScholes bsc, double opt_mkt_price, double x0, double x1,
      double tol, unsigned max_iter)          // 1
{
      // x: vol, y: BSc opt price - opt mkt price
      double y0 = bsc(x0) - opt_mkt_price;              // 2
      double y1 = bsc(x1) - opt_mkt_price;

      double impl_vol = 0.0;
      unsigned count_iter;
      for (count_iter = 0; count_iter <= max_iter; ++count_iter)            // 3
      {
            if (std::abs(x1 - x0) > tol)
            {
                  impl_vol = x1 - (x1 - x0) * y1 / (y1 - y0);

                  // Update x1 & x0:
                  x0 = x1;
                  x1 = impl_vol;
                  y0 = y1;
                  y1 = bsc(x1) - opt_mkt_price;
            }
            else
            {
                  break;                  // 4
```

```
            }
        }

        if (count_iter < max_iter)          // 5
        {
            return impl_vol;
        }
        else
        {
            return std::nan(" ");           // std::nan(" ") in <cmath>        // 6
        }
    }
```

The function first takes in a `BlackScholes` object that holds the parameters of V discussed above, followed by the option price `opt_mkt_price` (usually as observed in the market) for which we want to compute the implied volatility. The remaining input values are the two initial guesses (`x0` and `x1`), followed by a convergence tolerance value `tol` and maximum number of iterations `max_iter` . (1)

The initial function values `y0` and `y1` can be obtained by first calling the functor on the `bsc` object at each of `x0` and `x1` , and then subtracting the market option price. (2)

The secant method iteration will continue until either the maximum number of iterations is hit (3), or it converges to the implied volatility per the given tolerance *tol* and exits the loop. (4)

Once the loop is exited, the program checks whether it occurred before the maximum number of iterations was reached, in which case the algorithm has converged to the implied volatility. This value is then returned by the `implied_volatility(.)` function. (5)

If the loop reaches the maximum number of iterations before reaching convergence, `std::nan("")` (in `<cmath>`), representing “not a number” as a `double` type, is returned, indicating an error condition. (6)

There is one further refinement we can make. Notice we have the following lines of code inside the function:

```
double y0 = bsc(x0) - opt_mkt_price;
double y1 = bsc(x1) - opt_mkt_price;

// and then later inside the iteration...
y0 = bsc(x0) - opt_mkt_price;
y1 = bsc(x1) - opt_mkt_price;
```

Although not a grievious example of duplicated code, accidents can happen in more complex code bases, for example something akin to incorrectly modifying one of these statements, eg:

```
y0 = bsc(x0) +      opt_mkt_price;              // Wrong!
```

The code will compile and run but will result in incorrect results. Using a lambda expression, we can refactor this code into one place:

```
auto f = [&bsc, opt_mkt_price](double x) -> double
{
      return bsc(x) - opt_mkt_price;
};
```

Then, we are better assured the results will be accurate by replacing the four lines above with:

```
double y0 = f(x0);
double y1 = f(x1);
```

```
// and then later inside the iteration...
y0 = f(x0);
y1 = f(x1);
```

Move Semantics and Special Class Functions

Move semantics, another major enhancement in C++11, allows one to transfer ownership of an object, as opposed to copying which can entail a nontrivial performance hit. This is very convenient for financial programming, as there are often cases where it is the transfer of ownership that is actually required, rather than maintaining two copies of the same object.

With move semantics came two new special class functions, the *move constructor*, and the *move assignment operator*, analogous to the copy constructor and copy assignment operator. In addition, new and more straightforward ways of disabling or declaring a special class function as its default were also added as language features in C++11.

Move Semantics

One of the most perplexing problems in pre-C++11 development could be something very simple: storing a *subobject member* on an *enclosing* object. A backtest might require a set of intraday prices, for example, and then be used multiple times over a set of parameter values to identify the optimizing values. A `vector` containing the price data might need to be passed into the constructor of an enclosing `Backtest` object, and then be stored as a subobject member on the enclosing object.

One way to do this is to pass the `vector` by `const` reference and initialize the `prices_` member by copy:

```
class Backtest
{
public:
	Backtest(const std::vector<double>& prices, . . .);
	void reset_parameters(. . .);
	. . .

private
	std::vector<double> prices_;
	. . .

};

// Constructor implementation:
Backtest::Backtest(const std::vector& prices, . . .) : prices_{prices}, . . .	// object copy
{. . .}
```

On the upside, the `Backtest` object would have exclusive *ownership* of its `prices_` member. On the downside, this would require the object being copied in the initialization.

Prior to C++11, this object copy could be avoided using a commonly used alternative, storing the subobject `prices_` member by `const` reference:

```
class Backtest
{
public:
	Backtest(const std::vector& prices, . . .);
	void reset_parameters(. . .);
	. . .

private
```

```
        const std::vector& prices_;
        . . .

};

// Constructor implementation:
Backtest::Backtest(const std::vector& prices, . . .) : prices_{prices}, . . .     // avoids object copy now
{. . .}
```

While this approach will likely prevent performance hits due to object copy, it can introduce other problems such as *shallow copy*, creating the potential for the `prices_` container being modified from outside the `Backtest` object. In this case, an enclosing `Backtest` object would be said to be *nonowning*.

What we would like is to have a `Backtest` object retain full ownership of its `prices_` member, but without the overhead of an object copy. This is now possible with *move semantics* introduced in C++11.

An Introduction to Move Semantics

Before discussing move semantics as applied to subobject member initialization on an enclosing object, a first example example will present the basics of what it means when an object is moved. Suppose we revisit our `SimpleClass` from the previous chapter (now written with separate declaration and implementation):

```
class SimpleClass
{
        public:
                SimpleClass(int k);
                int get_val() const;
                void reset_val(int k);
```

```
        private:
            int k_;
};

// Implementation:
SimpleClass::SimpleClass(int k):k_{k} {}
int SimpleClass::get_val() const
{
   return k_;          // Private member on SimpleClass
}

void SimpleClass::reset_val(int k)
{
   k_ = k;
}


// Elsewhere, create an instance:
SimpleClass sc{78};
```

If we needed an identical `SimpleClass`, we could of course copy it:

```
SimpleClass another_sc{sc};             // Copy constructor
```

However, this comes with the overhead of object copy. Suppose instead we won't need `sc` anymore once `another_sc` is created. In this case, we can more efficiently *move* `sc` to `another_sc` instead of copying. This is accomplished by invoking the Standard Library `std::move(.)` function, declared in the `<utility>` header.

```
SimpleClass another_sc{ std::move(sc) };       // Move constructor
```

In this case, if we accessed the `k_` value on `another_sc`, it would return 78.

At a high level, the process for moving an object involves casting it to a temporary state called an *rvalue* reference first, which makes the object *eligible* to be moved. An rvalue reference is indicated by a double ampersand. The mechanics behind the scenes with rvalues and `std::move(.)` can be demonstrated again using a `SimpleClass` object:

```
SimpleClass sc2{ 15 };                         // 1

SimpleClass&& sc_rval = std::move(sc2);        // 2

SimpleClass sc2_move{sc_rval};                 // 3
```

In step (1), another `SimpleClass` object, `sc2`, is constructed, with `k_` initialized to 15. In step (2), an rvalue reference, `sc_rval`, is assigned to the result from `std::move(.)`. This means that `sc2` is now in a state that can be moved. The actual transfer of ownership occurs in step (3), where the (default) *move constructor* on the new `sc2_move` object is called. Similar to a copy constructor, the compiler will provide a default move constructor. This example can be restated with one fewer line:

```
SimpleClass sc2{ 15 };
SimpleClass sc2_move{std::move(sc2)};
```

Behind the scenes, an rvalue reference is created by `std::move(.)` and used as the move constructor argument for `sc_rval`.

A default move assignment operator is also provided by the compiler. The same move above could therefore be accomplished instead as follows:

```
SimpleClass sc2_move = std::move(sc2);
```

Note that attempting to assign a non-rvalue object—that is, an object not cast to a temporary movable state—will not compile:

```
SimpleClass sc3{ 33 };
SimpleClass&& sc_err = sc3;
```

The object `sc3` is said to be an *lvalue*. An *lvalue reference* refers to the same references we have known and loved prior to C++11, as shown here. This of course will compile:

```
SimpleClass& sc_lval = sc3;
```

The topic of rvalues and lvalues can get into a lengthy and complex discussion rather quickly, so what has been presented here has been kept brief and related specifically to the task at hand, namely how to use move semantics. If you wish to read deeper into the subject, Section 16.2.6—Understanding *std::move*—in the text *C++ Primer (5E)* by Lippman and Moo is recommended. **{6}**

Finally, an obvious question might be, what happens to the *moved-from* object `sc2` in the previous example?

```
SimpleClass sc2_move = std::move(sc2);
```

The answer is it remains in a *valid but undetermined* state, as it is not automatically destroyed. In this present case, the `k_` value may or may not remain 15, as its value is no longer guaranteed, but it can be reset. The object could technically then be reused, eg:

```
sc2.reset_val(216);
int sum = sc2.get_val() + 100;          // sum = 316
```

As is typical in C++, flexibility and performance take precedence over safety, and implementations of move semantics among compiler/Standard Library vendors may be different. So ultimately, it is left up to the user as to what to do with a moved-from object, such as resetting it to its default state, or just letting it destroy itself as it goes out of scope. The general rule to follow is "[y]ou can still (re)use them providing you do not make any assumptions about their value". **{7}** (Josuttis-Move), Summary, Sec 2.6, p 33). However, the same source (Josuttis-Move) **{8}** states the recommended practice is that moved-from objects should usually not be reused. For the examples to come in this book, we will adopt this practice and assume moved-from objects will be intended for destruction.

Passing Function Arguments with Move Semantics

It is possible to pass function arguments by move. Recalling that `std::move(.)` returns an rvalue reference, the function parameter will also be an rvalue reference, again indicated by the double ampersand. As a first example, we could have a function that takes in an rvalue to a `SimpleClass`:

```
int square_k(SimpleClass&& sc)
{
      return sc.get_val() * sc.get_val();
}

// Use the function:
```

```
SimpleClass sc_to_square{ 2 };
int squared_result = square_k(std::move(sc_to_square));          // Returns 4
```

This actually doesn't do much for us though, as a `const SimpleClass&` argument would achieve the same result, and would be simpler. However, as we saw earlier with the implied volatility example, we couldn't take in the `BlackScholes` argument as `const` reference, as its `vol_` member had to be modifiable in order to iterate through the secant method. In this case, there is an advantage to be had as we can achieve the same result by obviating pass-by-value (and object copy) and replacing it with move semantics:

```
double implied_volatility_with_move(BlackScholes&& bsc, double opt_mkt_price, double x0, double x1,
      double tol, unsigned max_iter)
{
      auto f = [&bsc, opt_mkt_price](double x) -> double
      {
            return bsc(x) - opt_mkt_price;
      };

      double y0 = f(x0);
      double y1 = f(x1);

      . . . (Code exactly the same as in the previous implied_volatility(.) example...

      if (count_iter < max_iter)
      {
            return impl_vol;
      }
      else
      {
            return std::nan(" ");          // std::nan(" ") in <cmath>
```

```
    }
}
```

By using `std::move(.)` to pass the `BlackScholes` argument, the result will be the same, but now without the overhead of copying the object.

Initializing Constructor Arguments with `std::move(.)`

This now brings us back to the motivation mentioned to the outset, namely finding a more efficient way to initialize a subobject member on an enclosing object, while the latter still retains exclusive ownership of the former, hence avoiding the problems associated with a `const` reference member.

For now, we will just look at a simple example to see how this works, where there is a `SimpleClass` sub-object member. In the next chapter, a financial example will be presented.

Back to our current example, an enclosing class can be designed to take in a constructor argument the "old way", by `const` reference, as well as by rvalue and `std::move(.)`, by including two constructors:

```
class Enclose
{
public:
   Enclose(const SimpleClass& sc);         // Constructor 1
   Enclose(SimpleClass&& sc);              // Constructor 2

private:
   SimpleClass sc_;
};


// Constructor implementations:
```

```
Enclose::Enclose(const SimpleClass& sc) :sc_{ sc } {}            // Constructor 1
Enclose::Enclose(SimpleClass&& sc) :sc_{ std::move(sc) } {}      // Constructor 2
```

In the case of the first constructor, again it is nothing new:

```
SimpleClass sc_enc{ 100 };
Enclose ec{ sc_enc };
```

The `sc_enc` argument is passed by `const` reference, and then a full object copy is made in the initialization of the subobject member `sc_`.

Alternatively, object copy can be avoided by passing an rvalue reference to `sc_enc` with `std::move(.)`, and then the initialization of `sc_` results in the move constructor on `SimpleClass` being called once.

A commonly preferred approach, however, has emerged that can yield essentially equivalent results with just a single constructor. In this case, the `sc` parameter is written to take in its argument by value, but the member initialization is performed by move:

```
class EncloseSingleConstructor
{
public:
   EncloseSingleConstructor(SimpleClass sc);

private:
   SimpleClass sc_;
};
```

```
// Constructor implementation:
EncloseSingleConstructor::EncloseSingleConstructor(SimpleClass sc):sc_{ std::move(sc) }{}
```

Proceeding naively:

```
SimpleClass sc_enc_2{ 200 };
EncloseSingleConstructor ec{ sc_enc_2 };
```

This results in one copy of `sc_enc_2` being generated when bound to the constructor argument, but the initialization of `sc_` is done by move. In terms of performance, this is on par with a `const` reference constructor argument initializing the `sc_` member by value (copy).

We can then do better by passing by rvalue reference:

```
EncloseSingleConstructor ec_move{ std::move(sc_enc_2) };
```

In this case, there are two move operations. First is when the move constructor is called on the `sc` constructor parameter, and second is the move when initializing the `sc_` member. As move operations are extremely cheap, this will not affect performance significantly compared to the constructor on the original `Enclose` class with an explicit rvalue reference parameter.

The upshot here is that initializing subobject members using move semantics will be far less expensive than with object copies associated with value semantics.

Given the efficiencies that can be gained with move semantics, one might be tempted to return objects from functions by move; eg:

```
// Don't do this:
SimpleClass f(int n)
{
	return std::move({n});
}

// or this:
SimpleClass g(int n)
{
	SimpleClass sc{n};
	return std::move(sc);
}
```

Per the Support Guidelines (F.48) **{9}**, this is not good practice and can in fact result in suboptimal results. Instead, one should simply return the object as usual, as guaranteed copy elision **{10}** takes care of the optimization for us.

What this means is if we instead write the functions as follows:

```
SimpleClass f(int n)
{
	return {n};
}

SimpleClass g(int n)
{
	SimpleClass sc{n};
```

```
    return sc;
}
```

then invoking the function calls

```
SimpleClass scf = f(2);

SimpleClass scg = f(3);
```

would be essentially the same as writing:

```
SimpleClass scf{2};
SimpleClass scg{3};
```

That is, no extra `SimpleClass` copies are generated when returning the object from `f(.)` or `g(.)`. In the case of `f(.)`, this is referred to as *Return Value Optimization*, or *RVO* for short, and in the case of `g(.)`, it is called *Named Return Value Optimization*, or *NRVO*, as the named object `sc` is created first before being returned.

Copy elision was permitted (under more specific conditions) beginning with C++11 but not mandatory, although many compilers already provided it. The Standard began requiring it beginning with C++17 and under more general conditions. As with just about everything else in C++, there are subtleties and exceptions, in particular those involving returns from different conditional control paths. Jonathan Boccara's Fluent C++ **{11}** blog provides a straightforward post that will help fill in these details.

Special Class Functions

So far, we have mentioned four of the six special class functions in C++:

- Copy constructor
- Copy assignment operator
- Move constructor
- Move assignment operator

The last two on this list, faciliatating move operations, were added to the language with C++11. The first two, along with two more below not yet referenced, have been in the Standard since its first ISO release in 1998:

- Default constructor
- Destructor

Each of these six special functions has a default provided by the compiler, and so far, these are what we have used. For this reason, we have not had to say much about them. There are cases, however, where they must be explicitly declared (implemented?), plus there are some related newer features from C++11 that can make our lives easier compared to the years prior.

Copy Constructor and Copy Assignment

An object can be copied using either its copy constructor or copy assignment operator. Both of these have defaults provided by the compiler, which are often times perfectly sufficient for what we need and should be preferred when possible:

```
SimpleClass sc{ 300 };
```

```
SimpleClass sc_copy{sc};          // Object sc copied to sc_copy with copy constructor.
                                        // sc is the copy constructor argument.

SimpleClass sc_assgn = sc;        //  Object sc copied to sc_copy with copy assignment.
```

However, there are cases where user-defined copy operations become necessary, particularly if a class contains a pointer as a member. These cases will be discussed in the next chapter, but for now, just note these declarations would also need to be included:

```
class SimpleClass
{
      public:
            SimpleClass(int k);
            int get_val() const;
            void reset_val(int k);

            // Copy constructor and copy assignment operator:
            SimpleClass(const SimpleClass& rhs);      // rhs = "right-hand-side"
            SimpleClass& operator =(const SimpleClass& rhs);

      private:
            int k_;
};
```

There can also be cases where we might just want to disable object copy, again for example where a pointer member might be present and we wish to prevent shallow copy, or in cases where we want to prevent the potential performance hit that often accompanies copying. Prior to C++11, this meant declaring both copy operations private, but without implementations:

```
class SimpleClass
{
      public:
            SimpleClass(int k);
            int get_val() const;
            void reset_val(int k);

      private:
            int k_;

            // Copy constructor and copy assignment operator disabled by
            // declaring them private:
            SimpleClass(const SimpleClass& rhs);      // rhs = "right-hand-side"
            SimpleClass& operator =(const SimpleClass& rhs);
};
```

This prevents external copying, but it would still be possible to implement a copy constructor that could be invoked internally:

```
// Copy constructor can still be implemented even if declared private:
SimpleClass::SimpleClass(const SimpleClass& copy):
   k_{ copy.k_ } {}

// And then, there could be a member function that creates a copy internally:
void SimpleClass::copy_mischief(int k)
{
   SimpleClass sc{ k };
   SimpleClass sc_copy{ sc };
      . . .
}
```

C++11 introduced the `delete` keyword for use in class declarations. This makes it readily obvious copying has been disallowed, plus it is prevented in *all* cases, including internal copying. Note also the declaration is now public:

```
class SimpleClass
{
	public:
		SimpleClass(int k);
		int get_val() const;
		void reset_val(int k);

		// Copy constructor and copy assignment operator are now disabled by
		// assigning them to the delete keyword:
		SimpleClass(const SimpleClass& rhs) = delete;
		SimpleClass& operator =(const SimpleClass& rhs) = delete;

	private:
		int k_;

};
```

In this case, the preceding copy constructor implementation would cause a compiler error, which is a good thing as it helps prevent possible runtime errors and/or unexpected behavior.

Finally, there are cases where we might need to explicitly assign the copy operations to their compiler-provided defaults. Reasons for this will be discussed in the next chapter, but for now, just note that this can be done using the `default` keyword:

```
class SimpleClass
{
```

```
    public:
        SimpleClass(int k);
        int get_val() const;
        void reset_val(int k);

        // Copy constructor and copy assignment operator are now
        // explicitly assigned to their compiler-provided defaults:
        SimpleClass(const SimpleClass& rhs) = default;
        SimpleClass& operator =(const SimpleClass& rhs) = default;

    private:
        int k_;

};
```

This is also convenient, and less error-prone, as we don't have to write our own implementations of the copy constructor and copy assignment operator.

Move Constructor and Move Assignment Operator

The move constructor and move assignment operator are the move analogs to the copy constructor and copy assignment operator, and defaults are again provided by the compiler. For the topics to be covered in this book, we will not need to define our own move operations, so the defaults will suffice. There can be cases, however, where the move constructor and move assignment operator will need to be explicitly assigned to `default`, as will be seen in the next chapter. It is also possible to disable them, if necessary, by using the `delete` keyword.

```
// Move operations disabled:
SimpleClass(SimpleClass&& rhs) = delete;
SimpleClass& operator =(SimpleClass&& rhs) = delete;
```

```
// Explicit default move operations:
SimpleClass(SimpleClass&& rhs) = default;
SimpleClass& operator =(SimpleClass&& rhs) = default;
```

Default Constructor

If no user-defined constructor is provided on a class, a default constructor provided by the compiler can be used.

```
class Minimal
{
public:
    int x() const;          // return x_;
    void set_x(int x);      // x_ = x;
    // No user-defined constructor

private:
    int x_;
};
```

A `Minimal` instance is then constructed by the default constructor provided by the compiler.

```
Minimal min{};              // Use uniform initialization
min.set_x(987);
```

If a user-defined constructor is added, then the compiler-provided default constructor is automatically disabled. If a default constructor is still needed, the programmer becomes responsible for including it.

Prior to C++11, in cases where a default constructor would need to initialize member data, initialization would occur in its implementation. For example, supposing `x_` needed to be initialized to 0 in the `Minimal` class, it would be updated as something like the following:

```
class Minimal
{
public:
	Minimal(int x);			// x_{x};
	Minimal();			// x_{0};
	int x() const;		// return x_;
	void set_x(int x);	// x_ = x;

private:
	int x_;
};

// User-defined default constructor
Minimal::Minimal() : x_(0) {}
```

Since C++11, however, it can be set to `default` in the header, along with in-class member initialization, as discussed previously, resulting in a more modern style:

```
class Minimal
{
public:
	Minimal(int x);			// x_{x};
	Minimal() = default;	// Can use default keyword	(2)
	int x() const;		// return x_;
	void set_x(int x);	// x_ = x;
```

```
private:
    int x_{0};                  // in-class member initialization (1)
};
```

Note that `x_` is initialized in its declaration (1), and the default constructor is explicitly defined as its compiler-provided default (2). This also means there is one fewer constructor requiring implementation, and thus less that can go wrong (particularly in more realistic and involved cases compared with this minimal class example):

```
Minimal::Minimal(int x) : x_{ x } {}
int Minimal::x() const
{
    return x_;
}

// No longer need to write out an implementation for the default constructor...

void Minimal::set_x(int x)
{
    x_ = x;
}
```

The Support Guidelines **{12}** tell us to prefer in-class initialization in cases where one or member needs to be initialized with a literal value.

This ensures consistent behavior in cases where there might be more than one constructor requiring a default value, leading to cleaner code while also being the most efficient form. In addition, a positive knock-on effect is it helps ensure all data members get initialized **{13}** at construction, another best practice that can be found in the Support Guidelines.

The Destructor

The last (but certainly not least) special class function is the destructor, which gets called when an object of the class goes out of scope, such as at the end of a function block, or when `delete` is called on a pointer to the object. The compiler-provided default destructor is typically sufficient in cases where a class does not contain a (raw) pointer member. There is one important case where the destructor should be explicitly defined as `default`, namely as a virtual default destructor on a base class not requiring memory deallocation (ie, no pointer member(s) on the base class).

```
class Base
{
      public:
      . . .

      virtual ~Base() = default;

      . . .
};
```

This will tie into a discussion on class inheritance in the next chapter, but for now it is presented as part of this overview of special class functions.

The Three-Way Comparison Operator (Spaceship Operator)

Suppose we have a `Fraction` class that takes in two integers and stores them as members `n_` and `d_`, representing the numerator and denominator. (Josuttis OOP **{14}**, Stanford course slides **{15}**]). To sim-

plify things, just assume these values are non-negative, and the denominator is not zero.

```
class Fraction
{
public:
      Fraction(unsigned n, unsigned d);     // initializes u_{u}, d_{d}


private:
      unsigned n_, d_;

};
```

Suppose we create two instances, say `a` and `b`:

```
Fraction a{1, 2};
Fraction b{3, 4};
```

Then, if we were to try to compare them, say

```
if(a == b)
{
      // Do something...
}

if(a > b)
{
      // do something...
}
```

The compiler would complain, as it doesn't know what equality or greather than means for a user-defined class. Therefore, it becomes necessary to define these operators ourselves.

Prior to C++11, definitions of all six operators— `==`, `!=`, `<`, `<=`, `>`, and `>=` —were required in order to cover every contingency:

```
bool operator == (const Fraction& rhs) const;
bool operator != (const Fraction& rhs) const;
bool operator < (const Fraction& rhs) const;
bool operator > (const Fraction& rhs) const;
bool operator <= (const Fraction& rhs) const;
bool operator >= (const Fraction& rhs) const;
```

From a logic point of view, if `==` and `<` were defined, then the remaining implementations could be expressed in terms of these two operators. However, separate implementations of all six were still required in the code, even if the remaining four were based on the the first two. With C++20, this is now streamlined with the introduction of the *three-way operator*, aka the *spaceship operator* `<=>`. Now, all we need to do is provide `==` and `<`, and we're essentially done.

Returning to the `Fraction` class, the equality operator is a simple matter by defining `==` as `default`. In this case, each member will be compared with the respective value on the object being compared, ie numerators are compared, and then denominators are compared. If each pair is equal, then the operator returns `true`. Conversely, this also now automatically defines the `!=` operator.

```
#include <compare>

class Fraction
{
public:
```

```
        Fraction(unsigned n, unsigned d);
        bool operator == (const Fraction& rhs) const = default;        // Defines both "==" and "!="

    private:
        unsigned n_, d_;

    };
```

The remaining inequality operators (both exclusive and inclusive) can be defined by the spaceship operator (`<=>`), for which the Standard Library `<compare>` header is required. However, rather than a `bool` return type, `std::strong_ordering` is used for integer values (more about this shortly).

A compiler-provided default is provided for `<=>` as well, but it doesn't help us in this case as it is defines member-by-member lexicographic comparison. That is, if we were to write:

```
#include <compare>

class Fraction
{
public:
        Fraction(unsigned n, unsigned d);
        bool operator == (const Fraction& rhs) const = default;
        std::strong_ordering operator <=> (const Fraction& rhs) const = default;

private:
        unsigned n_, d_;

};
```

then we would get nonsensical results such as the following resulting in a true condition:

$\frac{2}{3} < \frac{4}{13}$, because 4 > 2, and

$\frac{1}{2} < \frac{1}{5}$, because the numerators are equal, and 5 > 2

For this reason, we will need to write our own implementation, which can be done by defining and checking the "less than" (`<`) condition first. If not true, then the next step will be to check for equality based on the definition of `==` already established. If neither of these is true, then the result must be "greater than" (`>`). These are indicated by the `strong_ordering` values `less` , `equivalent` , and `greater` , respectively:

```
std::strong_ordering Fraction::operator <=>(const Fraction& rhs) const
{
	if(n_ * rhs.d_ < rhs.n_ * d_)
	{
		return std::strong_ordering::less;
	}
	else if (*this == rhs)		// Check if the active object is equal to rhs
	{
		return std::strong_ordering::equivalent;
	}
	else
	{
		return std::strong_ordering::greater;
	}
}
```

With definitions for both `==` and `<=>` in place, we can then apply any of the six comparative operators as usual, eg:

```
Fraction f1{ 1, 2 };
Fraction f2{ 3, 4 };

if(f1 <= f2)
{
      // <=> now properly evalauates the inequality 1/2 < 3/4
}
else
{
      . . .
}
```

As for the `strong_ordering` return type, this means any two values can be compared, as is the case for integral types in C++. There is also a `std::partial_ordering` type that should be used for floating point types such as `double` and `float`. This is because a floating type (eg `double`) can hold non-comparable assignments such as infinity and NaN. It should also be noted that two floating types should never be compared directly in defining `==`. Instead, equivalence in this case, say of two `double` values `x` and `y`, should be determined by whether `y` falls within some tolerance of `x`.

A third return type, `weak_ordering`, can be used for cases such as comparing strings (although not limited to this example), where case-insensitive but otherwise like characters are considered equivalant (Grimm C++20) **{16}**.

Although most ordering required for financial applications would probably just be based on a numerical type (eg `double`) and hence already defined in the language, one place where overloads for `==` and `<=>` can be handy is in the design of a date class, where ordering and date arithmetic is based on the number of days since an epoch, such as 1970-01-01 (UNIX epoch). A user-defined date class will be covered in Chapter Six.

Lambda Expressions and User-Defined Class Members

One last point to cover is capturing class member data and member functions in a lambda expression. Suppose we have a class that computes the value of a quadratic function, with the coefficients stored as member variables. A public member function `generate_values` will take in a `vector` of real numbers, compute the value of the function for each element, and store the results in another `vector`:

```
#include<vector>
class QuadraticGenerator
{
public:
	QuadraticGenerator(double a, double b, double c);	// a_{ a }, b_{ b }, c_{ c }
	std::vector<double> generate_values(const std::vector<double>& v);

private:
	double a_, b_, c_;
	std::vector<double> y_;
};
```

Inside the `generate_values(.)` function, we might want to separate out the function evaluation into a lambda expression that gets called each time from a range-based `for` loop iterating over `v`. In order for this, we will need the member variables `a_`, `b_`, and `c_`. We could list these individually in the capture, but it is also possible to gain access to all member data (and member functions) by setting the `this` pointer **{17}** to point to the active object in the lambda capture instead:

```
std::vector<double> QuadraticGenerator::generate_values(const std::vector<double>& v)
{
	auto quadratic_value = [this](double x) -> double
	{
```

```
            return x * (a_ * x + b_) + c_;
        };

        for (double x : v)
        {
            y_.push_back(quadratic_value(x));
        }

        return y_;
    }
```

In certain thread-based situations, it is possible that instead of the `this` pointer, a copy of the active object, `*this`, might be necessary as the lambda capture argument. In C++17, this was added, so that `quadratic_value(.)` could instead by written as:

```
auto quadratic_value = [*this](double x) -> double
{
    return x * (a_ * x + b_) + c_;
};
```

Examples in this book will primarily use the former case of capturing the `this` pointer rather than `*this`.

Summary

The topics presented above may have come across as a "various and sundry" list of mostly newer features that can be employed when writing user-defined classes, so it will probably help to recap what has been discussed.

A user-provided overload of `operator ()` defines a functor that allows us to pass a function object that can hold state and its own member functions. This existed prior to C++11, but it can be a useful tool in quantitative programming, such as for root finding to compute the implied volatility of an option.

A lambda expression is another form of a functor that can be defined "on the fly" inside another function and can help in refactoring code that is called multiple times. Lambda expressions can also capture the `this` pointer to an active object (or the dereferenced `*this` instead if desired) , providing the lambda with access to any class data member or member function. As a side note of things to come, both function objects and lambdas can be passed as an argument to a function or STL algorithm, as we will see in Chapter 4.

Move semantics allow passing objects as arguments to functions and initializing subobject members at construction without the overhead of object copy, while avoiding the problems that can result from older methods such as passing pointers or storing members by reference. We will see more practical examples in subsequent chapters.

The `default` keyword makes it clear and obvious up front that a special class function is to assume its compiler-provided default. As one specific example, combined with in-class member initialization, `default` can obviate the need to implement a default constructor. The `delete` keyword will disable a special function such that it can never be called, even from within an object. These keywords will become more relevant in the next chapter.

In-class member initialization provides several benefits that help reduce the risk of bugs, particularly as a complement to a constructor assigned as `default`. It obviates the necessity of a user-defined implementation by simply allowing members to be initialized in a class declaration. They also help ensure all member data is initialized at construction.

The three-way comparison operator, commonly referred to as the spaceship operator, allows all six equality and comparison operators to be defined in terms of `==` and `<`, so that it is no longer required

to define each in a separate implementation.

References

{1} James, Ch 5

{2} Joshi, implied volatility (cite chapter, page)

{3} Quantstart article, *Implied Volatility in C++ using Template Functions and Interval Bisection*

{4} Core Guidelines, initialize member variables in order.

{5} Wolfram Research Math World - *The Error Function*

{6} Lippman and Moo, *C++ Primer (5E)*, Section 16.2.6: *Understanding* `std::move`

{7} Josuttis-Move, Summary, Sec 2.6, p 33

{8} ibid, Sec 2.1.2, p 26 (Josuttis-Move)

{9} Core Guidelines (F.48) RVO

{10} Jonas Devlieghere, *Guaranteed Copy Elision*

{11} Jonathan Boccara, *RVO*, Fluent C++

{12} Core Guidelines, In-Class Initialization

{13} Core Guidelines: all data members get initialized

{14} Josuttis OOP (cite ch/section/page)

{15} Stanford Fraction class example

{16} Grimm C++20, Weak Ordering, p 137

{17} Stack Overflow: *What is the this pointer?*