Chapter 1. An Overview of C++

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 1st 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*.

Before launching into programming in C++, it will be useful to present a brief overview of the language, along with its companion, the C++ Standard Library, and where it continues to have a major presence in quantitative finance.

Some of the content here is likely familiar for most readers, but the discussion attempts to relate some of the basics with points related to quantitative programming, plus it introduces some of the newer additions to C++.

industry around this time had been raised on Fortran, particularly for writing numerical routines and scientific applications. While Fortran and its supporting libraries—BLAS, LAPACK, IMSL—were very well-developed in terms of mathematical and linear algebra support, the language lacked support for *object-oriented programming* at the time.

Financial modeling in the abstract is naturally comprised of different components that interact with each other. For example, to price even a simple derivative contract based on foreign exchange and interest rates, one would typically require the following:

- The yield curve for each currency
- A market rate feed of live foreign exchange rate quotes
- Volatility curves and/or surfaces for option pricing and risk measures
- A set of derivatives pricing methods, eg closed form or numerical approximations

Each of these components can be represented by an object, and C++ provided the means for creating these objects and managing their relationships to each other.

Banks and other financial institutions also needed a way to calculate risk measures at both a regional and global scale. This was a significant challenge for companies with trading operations spread across the major financial centers around the world such as New York, London, and Tokyo. At the start of each trading day, risk reporting was required for a firm's headquarters in, say, New York that took into account the portfolios maintained both locally and globally. This could be a computationally intensive task, but the performance of C++ made it possible and was yet another significant factor in its early adoption in the financial industry.

Following the turn of the century, newer object-oriented languages, such as Java and C#, made software development a relatively simpler and faster process, while more efficient processors became less expensive. However, the same features in these languages that enabled quicker deployment, such as built-in managed memory, garbage collection, and intermediate compilation, could also introduce overhead in terms of run-time performance. Management decisions on which language to adopt often came down to a trade-off between more rapid development and run-time efficiency. Even if one of these language alternatives were employed, computationally intensive pricing models and risk calculations were—and still are—often delegated to existing C++ libraries and called via an interface. It should also be noted that C++ offers certain compile time optimizations that are not available in these other programming languages.

C++11: The Modern Era is Born

In 2011, the ISO C++ Committee **{1}** (usually just called "The Committee") released a substantial revision that addressed long-needed modernization and in particular provided some very welcome abstractions that are immediately useful to quantitative developers. These include:

- Random number generation from a variety of probability distributions
- Lambda expressions that encapsulate mathematical functions that can also be passed as arguments
- Basic task-based concurrency that can parallelize computations without the need for manual thread management (with further enhancements planned for future releases post-C++20)
- Smart pointers that can help prevent memory-related program crashes

These topics and more will be discussed in the chapters ahead. An excellent reference by Jon Kalb and Gašper Ažman **{2}** covers the history and evolution of C++ into the modern era. It should also be noted that with more attention to, and promotion of the ISO C++ Core Guidelines **{3}** and ISO C++ Coding Standards **{4}** by the Committee, cross-platform development can now be a much easier task than in years past. The Core Guidelines in particular will be referenced with some frequency throughout this book.

And following C++11, new releases with more and more modern features addressing the demands of financial and data science industries are being rolled out on a three-year cadence, with the most recent release being C++20. This book will primarily cover developments through C++20, but we will discuss a few items in C++23 and proposed coming attractions in C++26 that should be of interest to financial developers.

Proprietary and high-frequency trading firms have been at the forefront of adopting C++11 and later standards, where the speed of acting on market and trading book signals in statistical strategies can mean a profound difference in profit and loss. Modern C++ is also in keen demand for implementing computationally intensive derivatives pricing models utilized by traders and risk managers at investment banks and hedge funds.

Open Source Mathematical Libraries

Another very welcome development over the past decade has been the proliferation of robust open source mathematical libraries written in standard C++ that therefore do not require the time-consuming C-language interface gymnastics of the past. Primary among these are the Boost libraries, the Eigen and Armadillo linear algebra libraries, and machine learning libraries such as TensorFlow and PyTorch. We will discuss some of these further, specifically Boost and Eigen, as the book proceeds.

Some Myths about C++

Some of the more infamous beliefs that have been perpetuated about C++ are:

- Knowledge of C is necessary for learning C++
    - While the C++ Standard retains most of the C language, it is entirely possible to learn C++ without knowledge of C, as we shall see in the material that follows. Clinging to C style can in fact hinder learning the powerful abstractions and potential benefits of C++.

- C++ is too difficult
    - OK, yes, there is some truth to this, as there is no doubt that C++ is a rich language that provides plenty of the proverbial rope with which one can hang oneself, and indeed it can at times be the source of nontrivial frustration. However, by leveraging *modern* features of the language while holding legacy issues in abeyance at the outset, it is entirely possible to become very productive as a quantitative financial developer in C++ reasonably quickly.
- Memory leaks are always a problem in C++
    - With smart pointers available since C++11—one of the newer *modern* features—and other Standard Library features such as STL algorithms (to be covered in Chapter 4), this no longer needs to be an issue in most financial model implementations.

Compiled vs Interpreted Code

C++ is a compiled language, where commands typed into a file by us mere mortals are first translated into a set of binary instructions, or *machine code*, that a computer processor will understand. This is in contrast to non-typed and interpreted quantitative languages such as Python, R, and Matlab, where each line of code must be reprocessed each time it is executed, thus slowing down execution time for larger applications, especially where there is heavy reliance upon iterative (looping) statements.

This is by no means a knock on these languages, as their power is evident in their popularity for rapid implementations and visualizations of models arising in quantitative fields such as finance, data science, and biosciences, where their available mathematical and statistical functions are in fact often already compiled in C, C++, or Fortran. However, financial programmers may be very well aware of cases where a model would require days to run in an interpreted language, where run times could be reduced to a matter of minutes or less when reimplemented in C++.

An effective approach is to use interpreted mathematical languages with C++ in a complementary fashion. This is when computationally intensive models code is written in a C++ library, and then called either

interactively or from an application in R, for example. C++ efficiently takes care of the number crunching, while the results can be used inside powerful plotting and other visualization tools in R that are not available in the C++ Standard Library.

Another advantage is the models code is written once and maintained in a C++ library that can be deployed across many different departments, divisions, and even international boundaries, and called via interfaces from applications written in different front-end languages, while ensuring consistent numerical results throughout the organization. This can be particularly advantageous for regulatory compliance purposes.

Popular open source C++ integration packages are available for both R and Python, namely Rcpp **{5}** and pybind11 **{6}**, respectively. Matlab also provides options for C++ interfaces, although there can be non-trivial license fees required for their add-on features.

The Components of C++

Standard C++ releases at a high level consist of two components: language features, and the C++ Standard Library. A software library is essentially a set of functions and classes that are not executable on their own but that are called by an application or system. Library development—both open source and commercial—now dominates modern C++ development compared to standalone applications that were more popular in previous decades, and we will discuss some of those later that are useful for computational work. The most important C++ library is the Standard Library which is shipped with modern compilers. Together, the Standard C++ language and the Standard Library are commonly referred to as *the Standard*.

C++ Language Features

C++ language features mostly overlap with the essential operators and constructs one would find in other programming languages, such as:

- Fundamental integer and floating-point numerical types
- Conditional branching: `if` / `else` statements and `switch` / `case` statements
- Iterative constructs: `for` loops and `while` loops
- Standard mathematical and logical operators for numerical types: addition, subtraction, multiplication, division, modulus, and inequalities

C++ language features support the four major programming paradigms: procedural programming, object-oriented programming, generic programming, and functional programming. Specific features that help provide each are, respectively:

- Standlone functions
- Classes and inheritance
- Templates
- Lambda expressions (since C++11)

Finally, as C++ is a strongly-typed language, the language provides a plethora of built-in numerical and logical types. Those that we will primarily use are as follows:

`double` (double precision) for floating point values

`int` for positive and negative integers

`unsigned` for non-negative integers

`bool` for boolean representation (`true` or `false`)

Ranges for each of the numerical types can vary across platforms, but on modern compilers the types noted here are sufficient for modern financial applications. A comprehensive guide that provides these ranges and details can be found on cppreference.com **{7}**.

The C++ Standard Library

As Nicolai Josuttis describes it in his indispensable text, *The C++ Standard Library - A Tutorial and Reference, 2nd Edition* **{8}**, the C++ Standard Library "enable(s) programmers to use general components and a higher level of abstraction without losing portability rather than having to develop all code from scratch." Up through the latest C++20 release, highly useful library features for financial modeling include:

- Single dimension array container classes, particularly the dynamically resizeable `vector` class
- A wide set of standard algorithms that operate on these array containers, such as sorting, searching, and efficiently applying functions to a range of elements in a container
- Standard real-valued mathematical functions such as square root, exponential, and trigonometric functions
- Complex numbers and arithmetic
- Random number generation from a set of standard probability distributions
- Task-based concurrency that can provide return values from functions run in parallel
- Smart pointers that abstract away the dangers associated with memory allocation and management
- A `string` class to store and manage character data

Use of Standard Library components requires the programmer to explicitly import them into the code, as they reside in a separate library rather than within the core language. The idea is similar to importing a NumPy array into a Python program or loading an external package of functions into an R script. In C++,

this is a two-step process, starting with loading the header files containing the Standard Library functions and classes we wish to use, and then scoping these functions with the Standard Library namespace name, `std`, often pronounced as "stood" by C++ developers.

As a quick first example, create a `vector` of `int` values and a `string` object, and output them to the console from a simple executable program inside the `main()` function:

```
#include <vector>             // vector class
#include <string>       // string class
#include <iostream>           // cout

int main()
{
      std::vector<int> x{ 1, 2, 3 };
      std::string s{ "This is a vector: " };
      std::cout << s << x[0] << ", " << x[1] << ", " << x[2] << "\n";

}
```

Note that the Standard Library classes `vector` and `string`, and the console output `cout` are scoped with the Standard Library `std` namespace. If you wish to save yourself typing `std::`, you can use `using` statements, preferably within individual functions, although placed at the top of a file can be acceptable in certain limited situations, such as writing small sets of test functions.

```
#include <vector>             // vector class
#include <string>       // string class
#include <iostream>           // cout

int main()
{
```

```
    using std::vector, std::string, std::cout;
    vector<int> x{ 1, 2, 3 };
    string s{ "This is a vector: " };
    cout << s << x[0] << ", " << x[1] << ", " << x[2] << "\n";
}
```

The output in each case is not surprisingly:

```
This is a vector: 1, 2, 3
```

The `using` statement is only required once, with the three Standard Library components following on the same line. This is a newer feature as of C++11. It also holds for assignments, such as:

```
double a = 1.0, b = 2.0, c = 4.2;
int i = 1, j = 2, k = 42;
```

NOTE

For examples in this book, it will often just be assumed `using` statements have been applied to commonly used Standard Library classes and functions such `vector`, `string`, `cout`, and `format`.

NOTE

Importing the `std` namespace into the global namespace with

```
using namespace std;
```

is sometimes found in code to replace the individual using statements or explicit scoping with the `std` namespace; however, this is not considered good practice, as it can result in naming clashes at compile time. More details can be found in the ISO C++ coding standards FAQ, *Should I use* `using namespace std` *in my code?* **{9}**.

NOTE

Two new features (since C++11) can be seen in the code example above:

1) Multiple commands and declarations can be placed on a single line; eg,

```
using std::vector, std::string;
```

```
int i = 0, j = 1, k = 2;
```

2) *Uniform initialization* (also called *braced initialization*), used in some of the above examples, is also a feature that was added with C++11:

```
vector<int> x{ 1, 2, 3 };
string s{ "This is a vector: " };
```

This is a useful feature in general that will be discussed momentarily (see *Some New C++ Features*—next section).

NOTE

`cout` is generally not used in production code. We will just use it as a placeholder where a result in reality would more likely be passed to a GUI or database interface, or another section of code.

To close out this introductory discussion of the Standard Library, one more type to be aware of is `std::size_t` —*size type*—in addition to the built-in types discussed as part of the C++ language. It is instead contained in the Standard Library and "is defined to be an unsigned integer with enough bytes to represent the size of any type" **{10}**. However, we will mainly be concerned with the fact that it's the return type for the size of a `vector`, and that you will need to use it as a return type as above, and possibly in indexed loops involving a `vector` or other STL container type:

```
#include <vector>
#include <cstdlib>              // std::size_t

. . .

std::vector<int> v{1, 2, 3};

std::size_t v_size = v.size();
```

As for the Standard Library distributions accompanying the three major compilers—Visual Studio, Clang, and gcc—`size_t` is equivalent to a 64-bit `unsigned long` type. All you really need to know about it for this book is it is an astronomically larger vector length than anything we would ever need.

Some New Features since C++11

This section will cover some useful features and syntax introduced since C++11 (inclusive), and then will conclude with general style guidelines that will be used in this book. Although it has been over 10 years since C++11 hit the scene, there is an unfortunate reality that it is still not covered in many university and quantitative finance programs, let alone later releases such as C++17.

Guidelines on code formatting and variable naming are in fact quite important when writing critical production code in financial systems, in a feature-rich language such as C++. Bugs, runtime errors, and program crashes are much more easily avoided or addressed if the source code is written in a consistent, clean, and maintainable state.

The `auto` Keyword

C++11 introduced the `auto` keyword that can automatically deduce a variable or object type. Some first simple examples are:

```
auto k = 1;                 // int
auto x = 419.531;       // double
```

In this case, `k` is deduced as an `int` type, and `x` a `double` type.

Varied opinions on the use of `auto` exist, but many programmers still prefer to explicitly state fundamental types such as `int` and `double` to avoid ambiguity. This will be the style followed in this book.

`auto` becomes more useful though when the return type is reasonably obvious from the context. As we will see in Chapter 3, a unique or a shared pointer is created with the Standard Library function `make_unique<T>(.)` or `make_shared<T>(.)`:

```
auto call_payoff = std::make_unique<CallPayoff>(75.0);
auto mkt_data = std::make_shared<LiveMktData>("CattleFutures");
```

In the first case, `make_unique<CallPayoff>` makes it fairly obvious a unique pointer to a `CallPayoff` object is being created (with a strike of 75). In the second, `make_shared<LiveMktData>` says it is creating a shared pointer to a `LiveMktData` object (providing cattle futures prices). These sufficiently expressive and a lot easier to maintain than the following:

```
std::unique_ptr<CallPayoff> call_payoff = std::make_unique<CallPayoff>(75.0);
std::shared_ptr<LiveMktData> mkt_data = std::make_shared<LiveMktData>("CattleFutures");
```

It is also handy if the return type is a long and nested class template type, such as from a function as follows:

```
std::map<std::string, std::complex<double>> map_of_complex_numbers(. . .)
{
    . . .

    return map_key_string_val_complex;
```

Then, instead of the pre-C++11 way:

```
std::map<std::string, std::complex<double>> cauchys_revenge = map_of_complex_numbers(. . .);
```

we can more cleanly and clearly call the function and assign the result using `auto`:

```
auto cauchys_revenge = map_of_complex_numbers(. . .);
```

Range-Based `for` Loops

Prior to C++11, iterating through a `vector` (or other STL container) would involve using the index as the counter, up to the number of its elements.

```
vector<double> v;
// Populate the vector v and then use below:
for(unsigned i = 0; i < v.size(); ++i)
{
    // Do something with v[i]
}
```

```
vector<double> v{ 1.0, 2.0, 3.0, 4.0, 5.0 };

for (unsigned i = 0; i < v.size(); ++i)
{
    cout << v[i] << " ";
}
```

Alternatively, you could use an iterator-based `for` loop:

```
for (auto iter = v.begin(); iter != v.end(); ++iter)
{
    cout << *iter << " ";
}
```

As an aside here, note the `auto` keyword means we can save ourselves from explicitly specifying the iterator type.

Range-based `for` loops, introduced in C++11, make this more functional and elegant. Instead of explicitly using the `vector` index, a range-based for loop simply says "for every element `x` in `v`, do something with it", similar to what you would find using Python or R:

```
for (double x : v)
{
      cout << x << " ";
}
```

Range-based `for` loops can also be used in applying operations to the elements of a `vector`. As a trivial example, calculate the sum of the elements:

```
double sum = 0.0;
for(double elem : v)
{
      sum += elem;
}
```

And we are done. No worries about making a mistake with the index, and the code more obviously expresses what it is doing. The Core Guidelines **{11}** in fact tell us to prefer using range-based `for` loops with `vector` objects, as well as with other STL containers that will be discussed in Chapter 4.

Use the `using` Keyword instead of `typedef`

Another way to ease the pain with cryptic template types prior to C++11 was to use a `typedef`. Revisiting the complex map example above, we could define an alias, `complex_map`:

```
typedef std::map<std::string, std::complex<double>> complex_map;
```

Beginning with C++11, the `using` statement was extended to perform the same task, but in a more natural syntax:

```
using complex_map = std::map<std::string, std::complex<double>>;
```

Furthermore, and perhaps more importantly, employing `typedef` in function and class templates required some additional and pesky coding gymnastics, while the `using` version can be used the same way in both non-template and template cases.

Uniform Initialization

C++11 introduced *uniform initialization*, also called *braced initialization*. There are several use cases, beginning with the simple case of initializing a numeric variable:

```
int i{ 106 };
```

This isn't terribly interesting, as it simply replaces putting `int i = 106;`. But, what if we had the following?

```
double x = 92.09;

int k = x;        // Compiles with warning
```

This will compile, albeit with some warning to the effect that the decimal part of `x` will be truncated, leaving `k` holding 92 alone. With uniform initialization, the compiler would issue a *narrowing conversion* error and halt the build, thus preventing unexpected behavior at run-time. This is a good thing, as it's better to catch errors at compile time rather than runtime.

```
int n{ x };             // Compiler ERROR: narrowing conversion
```

Uniform initialization can save you from problems with narrowing conversions during initialization of an object. We will return to this later in the book, but for now, consider the following simple class:

```
class SimpleClass
{
public:
      SimpleClass(int k) :k_{ k } {}      // Braced initialization in constructor also
      SimpleClass() = default;            // C++11 form of default constructor
                                                // (to be discussed in more detail in Ch XX)

private:
      int k_;
};
```

If we have

```
double x = 2.58;          // Assigned or input somewhere

. . .

SimpleClass sc_01(x);
```

This will compile successfully, but again with a warning that the `double` value of `x` is being converted to `int`, which will truncate the decimal values. However, by using uniform initialization:

```
SimpleClass sc_02{ x };              // Compiler ERROR
```

the compiler will halt with an error, again protecting against unexpected behavior at run-time.

Using the correct type will successfully result in an instance of the class:

```
int k = 319;          // Assigned or input somewhere

. . .

SimpleClass sc_03{ k };
```

Note there is also a default constructor. Prior to C++11, calling the default constructor would be without round brackets:

```
SimpleClass sc_04;
```

This could sometimes be an issue if you mistakenly put instead:

```
SimpleClass sc_04();
```

This would compile, but the compiler would treat it as a declaration of a `SimpleClass` *function*. With uniform initialization, both constructors use braces, so we now have consistency between the case where arguments are used, and the default constructor:

```
SimpleClass sc_05{};
```

One more point about uniform initialization is it can reduce verbosity with type deduction in the same vein as `auto`. For example, suppose we have a function `some_function(.)` that returns a `SimpleClass` object. Since the compiler knows the return type, all it will need is the constructor argument, indicated by braces rather than typing out `SimpleClass` again. The return object can simply be constructed using uniform initialization.

```
SimpleClass some_function(int m)
{
    if (m >= 0)
    {
        return {2 * m};          // Returns SimpleClass{ 2 * m }
    }
    else
    {
        return {-2 * m};    // Returns SimpleClass{ -2 * m }
    }
}
```

This also can be used when populating a `vector` or other container:

```
vector<SimpleClass> v{ {1}, {2}, {3} };
```

A `vector` of `SimpleClass` objects can be initialized with uniform initializations of each `SimpleClass` element of `v`.

In the examples above, the `SimpleClass` constructor takes in a single argument, but this can be extended to cases where a constructor has more than one argument. This will be seen in subsequent examples later in the book.

NOTE

An alternate equivalent form of uniform initializion is to put an equals sign in between the variable name and the left brace, eg:

```
int i = { 106 };

vector<SimpleClass> v = { {1}, {2}, {3} };
```

In this book, the original form above without the equals sign will be used.

NOTE

There is an exception to the rule with uniform initialization in the case of a `vector`. Going by the discussion above alone, one would probably expect the following to create a vector `u` of two integers:

```
vector<int> u{ 2 };
```

However, as we saw prior to the current discussion, this will actually initialize a `vector` with one `int` element, 2. To create a vector `v` of two elements, you still need to use the old round bracket form to indicate the 2 is a constructor argument and not a data value:

```
vector<int> v(2);
```

Formatting Output

A new feature as of C++20 worth mentionining is a new `format` command that can be streamed in a `cout` statement. It is similar to output commands found in other languages, particularly `Console.WriteLine(.)` in C#. Where output of fundamental types or strings is concerned, it makes it easier to add in descriptive text.

For example, suppose we have two variables `u` and `v` that have been assigned some values:

```
double u = 1.5;
double v = 4.2;
```

If we wanted to output these values with the variable name labels, we could write:

```
cout << "u = " << u << ", v = " << v << "\n';
```

Chaining the chevrons together, however, can become tiresome. Instead, we can now use:

```
#include <format>
. . .

cout << std::format("u = {0}, v = {1}", u, v) << "\n";
```

This says to put `u` in the first (0-indexed) position after `"u = "`, and then `v` in the following position:

```
u = 1.5, v = 4.2
```

When the order is from left to right, though, the index values can be dropped:

```
cout << std::format("u = {}, v = {}", u, v) << endl;
```

In some cases, however, the index values are needed; eg:

```
    double w = std::sin(u) + v;
    cout << "\n"
         << std::format("u = {0}, v = {1}, sin({0}) + {1} = {2}", u, v, w) << "\n";
```

To reiterate, console output in production code would be rare, if ever, but can be useful as a tool when learning C++ on its own. As such, it will be used throughout the book for demonstrating output results.

Class Template Auto Deduction (CTAD)

C++17 introduced *Class Template Auto Deduction*, or *CTAD* for short. Similar to `auto`, it will deduce the type of a template parameter based upon the data being initialized. So, in place of the earlier example

```
std::vector<int> x{ 1, 2, 3 };
```

we could instead just drop the `int` template parameter to arrive at the same result:

```
std::vector x{ 1, 2, 3 };
```

The examples above are again trivial, just using hard-coded values, but CTAD in more realistic situations can lighten the notation and make your code more readable. We will see such examples later on in the book, particular in Chapter Seven when working with the multi-dimensional array view `std::mdspan`, set for release in C++23.

Enumerated Constants and Scoped Enumerations

Prior to C++11, enumerated constants—more commonly called enums for short—were a great means of making it clearer for us mere mortals to comprehend integer codes by representing them as named constants. It was also far more efficient for the machine to process integers rather than passing bulkier `std::string` objects that take up more memory. And finally, errors caused by typos in quoted characters and stray strings could be avoided.

The C++11 Standard improved on this further with scoped enumerations (using `enum class`). These remove ambiguities that can occur with overlapping integer values when using regular enum constants, while preserving the advantages.

In what follows, the motivation for preferring the more modern scoped enumerations over integer-based enums is presented.

Enumerated Constants

To start with an example, we could create an enum called `OptionType` that will represent the types of option deals that are allowed in a simple trading system, eg European, American, Bermudan, and Asian. The `enum` name (`OptionType`) is declared; then, inside the braces, the particular constant values are defined, separated by commas. By default, each will be assigned an integer value starting at zero and incremented by one (consistent with zero-based indexing in C++). The closing brace must be followed by a semicolon. In code, we would write:

```
enum OptionType
{
   European,      // default integer value = 0
   American,      // default integer value = 1
   Bermudan,      // default integer value = 2
   Asian        // default integer value = 3
};
```

Just to verify each corresponding integer value, output it to the screen:

```
cout << " European = " << European << endl;
cout << " American = " << American << endl;
cout << " Bermudan = " << Bermudan << endl;
cout << " Asian = " << Asian << endl;
cout << endl;
```

Checking the output, we get:

```
European = 0
American = 1
Bermudan = 2
Asian = 3
```

Potential Conflicts with Enums

As discussed at the outset, for any `enum` type, the default integer assignments start at zero and then are incremented by one for each value. Therefore, it is possible that two enumerated constants from two different types could be numerically equal. For example, suppose we define two different `enum` types, called `Football` and `Baseball`, representing the defensive positions in each sport. By default, the baseball positions start with 0 for the pitcher and are incremented by one for each in the list. The same goes for the (American) football positions, starting with defensive tackle. The integer constants are provided in the comments.

```
enum Baseball
{
      Pitcher,         // 0
      Catcher,         // 1
      First_Baseman,     // 2
      Second_Baseman, // 3
      Third_Baseman,    // 4
      Shortstop,          // 5
      Left_Field,     // 6
      Center_Field, // 7
      Right_Field          // 8
};

enum Football
{
```

```
        Defensive_Tackle, // 0
        Edge_Rusher,      // 1
        Defensive_End,          // 2
        Linebacker,             // 3
        Cornerback,             // 4
        Strong_Safety,          // 5
        Weak_Safety             // 6
};
```

Then, we could compare `Defensive_End` and `First_Baseman`:

```
if (Defensive_End == First_Baseman)
{
    cout << " Defensive_End == First_Baseman is true" << endl;
}
else
{
    cout << " Defensive_End != First_Baseman is true" << endl;
}
```

Our result would be nonsense:

```
Defensive_End == First_Baseman is true
```

This is because both positions map to an integer value of 2. A quick fix, and one that was often employed prior to C++11, would be to reindex each set of enums; eg,

```
enum Baseball
{
      Pitcher = 100,
      Catcher,       // 101
      First_Baseman,     // 102
      . . .
};

enum Football
{
      Defensive_Tackle = 200,
      Edge_Rusher, // 201
      Defensive_End,     // 202
      . . .
};
```

Now, if we compare `Defensive_End` and `First_Baseman`, they will no longer be equal, because $202 \neq 102$. Still, in large code bases there might be hundreds of enum definitions, so it would not be out of the question for an overlap to slip in and cause errors. Enum classes, introduced in C++11, eliminate this risk.

Scoped Enumerations with Enum Classes

In more modern C++, a more robust approach eliminates integer values altogether. The other benefits of enums still remain, such as avoiding cryptic raw numerical codes or relying on string objects, but the numerical conflicts like those shown above are avoided by using what is called a scoped enumeration, accomplished with an enum *class*, added in C++11. As an example, we could define bond and futures contract categories, as shown here:

```
enum class Bond
{
      Government,
      Corporate,
      Municipal,
      Convertible
};

enum class Futures_Contract
{
      Gold,
      Silver,
      Oil,
      Natural_Gas,
      Wheat,
      Corn
};

enum class Options_Contract
{
   European,
   American,
   Bermudan,
   Asian
};
```

Notice that we no longer need to manually set integer values to avoid conflicts as we did with regular enums.

Attempting to compare members of two different enum classes—such as a `Bond` and a `Futures_Contract` position, will now result in a compiler error. For example, the following will not even

compile (using the default `==` operator):

```
if(Bond::Corporate == Futures_Contract::Silver)
{
    // . . .
}
```

This works to our advantage, as again it is much better to catch an error at compile time rather than run-time. The Core Guidelines **{12}** now maintain that we should prefer using enum classes rather than enumerated constants, "[t]o minimize surprises".

It is still possible to cast scoped enums to integer index values if desired. For example, each of `Bond::Corporate` and `FuturesContract::Silver` is the second member in its respective enum class, so by default each can be cast to a value of `1`, even though they are not comparable.

```
cout << format("Corp Bond index: {}", static_cast<int>(Bond::Corporate)) <<  "\n";
cout << format("Silver Futures index: {}", static_cast<int>(Futures_Contract::Silver)) << "\n";
```

The output would be

```
1
1
```

It is also possible to assign particular index values:

```
enum class Option_Type
{
```

```
        European      = 100,
        American      = 102,
        Bermudan      = 104,
        Asian         = 106
};
```

But again, there is no risk of equivalence with other `enum class` members, as two class types are not comparable and prevented by the compiler. There are, however, cases where having numerical representation can be convenient, as will be seen in later chapters.

Finally, enums in general, and enum classes in particular, are natural complements to `switch` / `case` statements.

```
void switch_statement_enum(OptionType ot)
{
        switch (ot)
        {
        case OptionType::European:
                std::cout << "European: Use Black-Scholes" << "\n";
                break;
        case OptionType::American:    // AMERICAN case matches....
                cout << "American: Use a lattice model" << "\n";
                break;
        case OptionType::Bermudan:
                cout << "Bermudan: Use the Longstaff-Schwartz LSMC model" << "\n";
                break;
        case OptionType::Asian:
                cout << "Asian: Calculate average of the returns time series" << "\n";
                break;
        default:
                cout << "The SEC might want to talk with you" << "\n";
```

```
            break;
        }
    }
```

Lambda Expressions

A *lambda expression* is often referred to as an *anonymous function object*, a term ostensibly coined from its original design proposal back in 2006 **{13}**. Also known in the vernacular as a *lambda function*, or just a plain *lambda*, it can define a functor on the fly within the body of another function. Additionally, like class functors we already discussed, a lambda can be passed as an argument into another function. This last property is what makes them so powerful for use within STL algorithms, to be covered in Chapter Four.

Lambda expressions are a welcome addition to the modern C++ arsenal. Introduced first in C++11, various enhancements were made in each subsequent release, including the most recent C++20 Standard. An excellent summary of the evolution of these improvements can be found in **{14}** (Jonathan Boccara).

To begin the discussion, a lambda expression that prints out good old "Hello World!" can be written as follows inside an enclosing function:

```
void worlds_simplest_function()
{
    auto f = []
    {
        std::cout << "Hello World!" << "\n";
    };

    f();
}
```

A lambda can also take in function arguments by using optional round brackets, a la a regular C++ function:

```
auto g = [](double x, double y)
{
	return x + y;
};

double z = g(9.2, 2.6); // z = 11.8
```

Three things to note at this stage:

First, lambdas have a return type of `auto`, but one has the option of indicating an explicit return type is an option, as follows:

```
auto g = [](double x, double y) -> double
{
	return x + y;
};
```

This also applies to returning an object of a class:

```
auto some_lambda = [](int n) -> SimpleClass
{
	return { n };		// Braced initialization of a SimpleClass type
};
```

Second, be sure to place a semicolon at the end of the lambda block. When placed on a single line as in the first example, this is somewhat obvious as it looks just like any other one-line C++ statement. However, it can be easily overlooked if your lambda implementation spans several lines, in which case your code will fail to compile.

Third, in cases of no function arguments, such as in the first example, round brackets are optional, but the square brackets are mandatory in order to define a lambda. The square brackets in the above examples are empty, but in general they provide the *capture* for the lambda expression.

The capture of a lambda expression does what it says, namely capture (non-static) external variables, including objects, allowing them to be used inside the body of the lambda. The capture data may be taken in by value or by non-`const` reference, with the latter option potentially resulting in modification. Before forming an example, suppose a mutator function is added to `SimpleClass`:

```
class SimpleClass
{
    public:
        . . .

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

    . . .
```

Then, in the following example, a `SimpleClass` object `sc` is constructed and then allowed to be captured in the lambda by reference, while the `alpha` variable is initialized.

```
void lambda_capture_reference()
{
	SimpleClass sc{ 7 };		// k_ = 7
	int alpha = 1;

	auto change_value = [&sc, alpha](int n) -> void
	{
		sc.reset_val(n + alpha);
	};

	change_value(2);		// Modified inside the lambda.
	// Now k_ =  n + alpha = 2 + 1 = 3
}
```

When the lambda function `change_value` is invoked, it captures `sc` by reference and `alpha` by value simultaneously. `sc` is modified with its new state, which is reflected externally, due to its being captured by reference. `alpha`, captured by value, is only modified inside the lambda. The capture can contain an arbitrary number of variables, but you need to be sure not to designate a single variable by both value and reference; eg, putting

`[sc, &sc, alpha]`

above would also result in a compiler error.

NOTE

Using the wildcard `[=]` or `[&]` as a lambda capture will allow any preceding external variable or object to be captured by value or reference, respectively. There is some debate among C++ developers whether this is good practice, however. Scott Meyers, for example, points out that (direct quote in italics) **{15}**:

- *Using* [&] *can result in dangling references or unexpected behavior*
- *Using* [=] *could cause problems due to unexpected copying of pointers, including smart pointers*
- *Avoiding these defaults can also increase the probability of catching problems at compile time rather than at run time*

Mathematical Operators, Functions, and Constants in C++

Standard mathematical operators for numerical types are available as language features in C++, and a comprehensive discussion follows below. Common mathematical functions, however—such as cosine, exponential, etc—plus a newer set of special functions, are provided in the C++ Standard Library rather than in the core language. These are also presented below.

Standard Arithmetic Operators

Addition, subtraction, multiplication, and division of numerical types are provided in C++ with the operators `+`, `-`, `*`, and `/` respectively, as usually found in other programming languages. Furthermore, the modulus operator, `%`, is also defined for integer types (`int`, `unsigned`, `long` etc). Examples are as follows:

```
// integers:
int i = 8;
int j = 5;
int k = i + 7;
int v = j - 3;
int u = i % j;

// double precision:
double x1 = 3.06;
double x2 = 8.74;
double x3 = 0.52;
double y = x1 + x2;
double z = x2 * x3;
```

The order and precedence of arithmetic operators are the same as found in most other programming languages, as well as middle school math for that matter, namely:

- Order runs from left to right:

```
i + j - v
```

Using the above integer values would result in 8 + 5 - 2 = 11

- Multiplication, division, and modulus take precedence over addition and subtraction, eg:

```
x1 + y / z
```

Using the above double precision values would result in

$$3.06 + \frac{11.8}{4.5448} = 3.06 + 2.5964 = 5.6564$$

Use round brackets to change the precedence, eg:

```
(x1 + y) / z
```

would yield

$$frac14.864.5448 = 3.2697$$

Compound assignment operators are also included, for example

```
x1 = x1 + x2;
```

can be replaced with addition assignment:

```
x1 += x2;
```

The remaining operators `-`, `*`, `/`, and `%` also have their respective versions.

Mathematical Functions in the Standard Library

Many of the usual mathematical functions one finds in other languages have the same or similar syntax in C++. Functions commonly used in computational applications include those listed in the following table, in which `x`, `y`, and `z` are assumed to be double precision variables:

Table 1-1. Standard Library Mathematical Functions

C++	Description
`cos(x)`	cosine of x
`sin(x)`	sine of x
`tan(x)`	tangent of x
`exp(x)`	exponential function e^x
`log(x)`	natural logarithm $ln(x)$
`sqrt(x)`	square root of x
`cbrt(x)`	cube root of x
`pow(x, y)`	x raised to the power of y
`hypot(x, y)`	computes $\sqrt{x^{2+y}2}$
`hypot(x, y, z)`	computes $\sqrt{x^{2+y}2+z^2}$ (since C++17)

A comprehensive list of common mathematical functions **{16}** is available on cppreference.com, an essential resource for any C++ developer. Josuttis, Section 17.3 ((op cit) **{8}**), is also very informative.

As these are contained in the Standard Library rather than as language features, the `cmath` header file should always be included, with the functions scoped by the `std::` prefix, viz,

```
#include <cmath>     // Put this at top of the file.

double trig_fcn(double theta, double phi)
{
 return = std::sin(theta) + std::cos(phi);
```

```
}

// Or, alternatively

double zero_coupon_bond(double face_value, double int_rate, double year_fraction)
{
      using std::exp;
   return face_value * exp(-int_rate * year_fraction);
}
```

Two points to note regarding `<cmath>` functions are as follows. First, there is no power operator in C++. Unlike other languages, where an exponent is typically indicated by a `^` or a `\**` operator, this does not exist as a C++ language feature (the operator `^` does exist but is not used for this purpose). Instead, one needs to call the Standard Library `std::pow(.)` function in `<cmath>`. When computing polynomials, however, it may be more efficient to apply factoring per Horner's Method and reduce the number of multiplicative operations (Stepanov) **{17}** . For example, if we wish to implement a function

$$f(x) = 8x^4 + 7x^3 + 4x^2 - 10x - 6$$

it can be preferable to write it in C++ as

```
double f(double x)
{
 return x * (x * (x * (8.0 * x + 7.0) + 4.0 * x) - 10.0) - 6.0;
}
```

rather than

```
double f(double x)
{
 return 8.0 * std::pow(x, 4) + 7.0 * std::pow(x, 3) +
   4.0 * std::pow(x, 2) + 10.0 * x - 6.0;
}
```

Performance results can depend on the compiler used as well as using different compiler optimizations, but you will almost surely be no worse off using Horner's Method.

For the case of a non-integer exponent, say

$$g\left(x,y\right)=x^{-1.368x}+4.19y$$

then there is no available alternative to using `std::pow(.)`:

```
double g(double x, double y)
{
 return std::pow(x, -1.368 * x) + 4.19 * y;
}
```

Second, you might be able to use some of these functions without `#include <cmath>`. This is unfortunately one of the quirks in C++ due to its long association with C; however, the moral of the story is quite simple: to keep C++ code ISO-compliant, and thus help ensure compatibility across different compilers and operating system platforms, one should always put `#include <cmath>`, and scope the math functions with `std::`. Do not include the `math.h` C header.

A prime example is the absolute value function. In some C++ compilers, the default (global namespace) `abs(.)` function might be a carryover from `math.h` that was only implemented for integer types. In order to calculate the absolute value of a floating point number in `math.h`, one would need to use the `fabs(.)`

function. However, `std::abs(.)` is overloaded for *both* integer and floating point (eg `double`) arguments and should be preferred.

It is also the case that the `<cmath>` functions have been evolving separately from their C counterparts, including optimizations particular to C++. This is one more reason to prefer the Standard Library versions. Per the *GNU C++ Library Manual* **{18}**,

The standard specifies that if one includes the C-style header (`math.h` in this case), the symbols will be available in the global namespace and perhaps in namespace std, but this is no longer a firm requirement. On the other hand, including the C++-style header (`<cmath>`) **guarantees** *that the entities will be found in namespace std and perhaps in the global namespace* **{18}** (direct quote in italics, emphasis in bold by the author).

So, long story short: use `#include <cmath>` and scope math functions with `std::` .

Mathematical Special Functions

This is a set of special functions such as Legendre polynomials, Hermite Polynomials, Bessel functions, and the exponential integral. These functions are often employed in physics, but as quantitative finance has a strong historical ties to the physics world, it is possible you may come across them in advanced derivatives modeling. In one of the most well-known and cited papers on options pricing by Longstaff and Schwartz, *Valuing American Options by Simulation: A Simple Least-Squares Approach* **{19}**, both Legendre and Hermite polynomials can be used as a basis (among others) for the underlying model.

Further discussion would be beyond the scope of this book, but for more details re the special math functions, **{20}** (Josuttis: {cpp17}) and **{21}** (cppreference.com) are good resources.

Standard Library Mathematical Constants

A handy addition to the C++20 Standard Library is a set of commonly used mathematical constants, such as the values of π, e, $\sqrt{2}$, etc. Some of those that are convenient for quantitative finance are shown in the following table.

Table 1-2. Standard Library Mathematical Functions

C++ Constant	e	pi	inv_pi	inv_sqrt_pi	sqrt2
Definition	e	π	$\frac{1}{\pi}$	$\frac{1}{\sqrt{\pi}}$	$\sqrt{2}$

To use these constants, one must first include the `numbers` header in the Standard Library. At the time of this writing, each must be scoped with the `std::numbers` namespace. For example, to implement the function

$$f(x) = \frac{1}{\sqrt{2\pi}}\left(sin(\pi x) + cos\left(\frac{y}{\pi}\right)\right)$$

we could write

```
#include <cmath>
#include <numbers>
. . .
double math_constant_fcn(double x, double y)
{
      double math_inv_sqrt_two_pi =
            std::numbers::inv_sqrtpi / std::numbers::sqrt2;
      return math_inv_sqrt_two_pi*(std::sin(std::numbers::pi * x) +
            std::cos(std::numbers::inv_pi*y));
}
```

This way, whenever π is used in calculations, for example, its value will be consistent throughout the program, rather than leaving it up to different programmers on a project who might use approximations out to varying precisions, resulting in possible inconsistencies in numerical results.

In addition, the value of $\sqrt{2}$, which can crop up somewhat frequently in mathematical calculations, does not have to be computed with

```
std::sqrt(2.0)
```

each time it is needed. The constant

```
std::numbers::sqrt2
```

holds the double precision approximation itself. While perhaps of trivial consequence in terms of one-off performance, repeated calls to the `std::sqrt` function millions of times in computationally intensive code could potentially have some effect.

NOTE

While not essential to know at this point, it is worth mentioning that these constants are fixed at compile time rather than computed with each call runtime, using a C++11 designation called `constexpr`.

As a closing note, it is somewhat curious that the set of mathematical constants provided in C++20 include the value $\frac{1}{\sqrt{3}}$, but not $\frac{1}{\sqrt{2}}$ or $\frac{1}{\sqrt{2 \pi}}$, despite the latter two being far more commonly present mathematical and statistical functions, including those used

in finance. These latter two are, however, included in the Boost Math Toolkit library, to be covered in Chapter Eight.

Naming Conventions

So far, we started by just plowing ahead with examples without discussing this topic. The examples are fairly simple at this outset, but as code gets more involved, it is a good idea to step back and conclude with a brief discussion about naming conventions and coding style, and the style guidelines that will be used for in this book. More importantly, in real-life financial C++ development work, the code you work with and write will take a quantum leap in complexity, so these issues become a necessity.

Now, just first to review, variable, function, and class names can be any contiguous combination of letters and numbers, subject to the following conditions:

- Names must begin with a letter or an underscore; leading numerals are not allowed (non-leading numerals are).
- Other than the underscore character, special characters, such as `@`, `=`, `$` etc are not allowed.
- Spaces are not allowed. Names must be contiguous.
- Language keywords are reserved and are not allowed to also be names, such as `double`, `if`, `while`, etc. A a complete listing of reserved keywords can be found in **{22}** (cppreference.com).

NOTE

Regarding the first bullet point, technically speaking, names beginning with an underscore are legal; however, in reality, they are often reserved for the compiler to use and so are typically discouraged. Some coding styles do use trailing underscores for private class members, which will be the case for this book.

Single letter variable and function names are fine for simple examples and plain mathematical functions. However, for quantitative financial models implementation, trading and risk systems, etc, it will usually be better to pass function arguments with more descriptive names. Function and class names as well should also provide some indication of what they do. Furthermore, it is important to decide as a group or company on a set of naming and style rules in order to enhance code maintainability and reduce the risk of bugs getting into the code.

Several naming styles have been common over the years, namely

- Lower Camel case; eg, `optionDelta`, `riskFreeRate`, `efficientFrontier`: Letter of first word in lower case, and following words capitalized
- Upper Camel, aka Pascal case; eg, `OptionDelta`, `RiskFreeRate`, `EfficientFrontier`: Letter of each word is in upper case
- Snake case; eg, `option_delta`, `risk_free_rate`, `efficient_frontier`: Each word begins with lower case, separated by an underscore character

Lower Camel and Snake cases are typical of what is often found in C++ function and variable names, and class names are usually in Upper Camel form. Microsoft **{23}** prefers the Lower Camel case, while the C++ Standard uses the Snake case, as does Google's C++ Style Guide **{24}**. The user should adopt the style of the team he or she is working on, and in general be comfortable with both styles. In this book, we will use the Snake case for function and variable names, and Upper Camel for class names. In addition, trailing underscores will be used for private member variables and member functions, eg:

```
class Blah
{
      public:
            Blah(double x, . . .);
            void calc_blah(double y) const;
```

```
        . . .

    private:
        double x_;

        double do_something_();
};
```

In cases where single characters are used for integral counting variables, it is still common to use the Fortran convention of letters `i` through `n`, although this is not required. We will also adopt this practice for the most part.

For an example of how *not* to write code, the article *How To Write Unmaintainable Code Ensure a job for life ;-)* **{25}** provides a humorous but not irrelevant viewpoint. To quote Homer Simpson, "it's funny because it's true" (d'oh) **{26}**.

Summary

C++ is broadly divided into two components, namely language features, and the Standard Library. Taken together, they are typically referred to as *the Standard*, and the major compiler vendors—particularly "the big three", Microsoft Visual Studio, LLVM Clang, and GNU gcc will include their respective Standard Library distributions with their compiler releases. Implementation of the Standard Library is mostly left up to the individual vendor, as long as they comply with the ISO Standard requirements.

There is some history behind the rise and fall in popularity of C++ in financial software development. After some struggles competing with Java and C#, however, C++ started to experience a bit of a renaissance with the release of C++11 in this domain. With subsequent releases every three years—currently C++20, and C++23 set for release soon— new features and inexpensive abstractions have supplanted a

lot of coding that once had to be done from scratch, resulting in cleaner code when used effectively. Some of these, such as the `auto` keyword, range-based `for` loops, scoped enums, lambda expressions, and mathematical constants were discussed above. Further specific examples as applied to financial software will follow in subsequent chapters.

References

{1} ISO C++ Committee

{2} C++ Today: The Beast is Back, by Jon Kalb and Gašper Ažman

{3} ISO C++ Core Guidelines

{4} ISO C++ Coding Standards

{5} Hanson, Integration of C++ code in R packages with Rcpp

{6} pybind11. Note: Can also find working examples of pybind11 here (Steven Zhang)

{7} cppreference.com[C++ Numerical Types, cppreference.com

{8} Nicolai Josuttis, The C++ Standard Library - A Tutorial and Reference, 2nd Edition. Also available on O'Reilly Learning

{9} Should I use `using namespace std` in my code?, ISO C++ Coding Standards

{10} Christopher Di Bella, Why Do Some C++ Programs use size_t?

{11} ISO Core Guidelines, Prefer Range-Based for Loops

{12} ISO Core Guidelines, enum class

{13} Stroustrup et al, Lambda Expressions Proposal N1968.

{14} Jonathan Boccara, The Evolution of Lambdas in C++14, C++17, and C++20

{15} Scott Meyers, Effective Modern C++, O'Reilly, Item 31 (p 217 and p 220)

{16} Common Mathematical Functions, cppreference.com

{17} Stepanov and Rose, *From Mathematics to Generic Programming* (Horner's Method)

{18} The GNU C++ Library Manual, _ *The C Headers and namespace std*

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

{20} Josuttis, *C++17 The Complete Guide* (28.3.3)

{21} cppreference.com, Special Math Functions _

{22} cppreference.com, Reserved Language Keywords

{23} Microsoft Coding Style Conventions

{24} Google C++ Style Guide

{25} How To Write Unmaintainable Code Ensure a job for life ;-)

{26} Homer Simpson, *Homer vs. Lisa and the 8th Commandment*, The Simpsons Season 2, Episode 13 (Fox)