O'REILLY

Learning Modern C++ for Finance

Foundations for Quantitative Programming

Daniel Hanson

Learning Modern C++ for Finance

Foundations for Quantitative Programming

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.

Daniel Hanson

Learning Modern C++ for Finance

by Daniel Hanson

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

Editors: Jeff Bleiel and Amanda Quinn

Production Editor: Ashley Stussy

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

July 2023: First Edition

Revision History for the First Edition

- 2022-04-27: First Release
- 2022-06-15: Second Release
- 2022-11-29: Third Release
- 2023-02-08: Fourth Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098100803 for release details.

978-1-098-10080-3

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 the C++ Standard Library, and the ways in which C++ continues to have a major presence in quantitative finance.

You may have already felt intimidated by opinions and rumors claiming that C++ is extraordinarily difficult to learn and fraught with minefields. So, in this chapter, we will try to allay these fears by first debunking some of the common myths about C++, and then presenting straightforward examples to help you get up and running.

Most of the content here is likely familiar for most readers, but the discussion here attempts to extend some of the basics with points about quantitative programming and best practices that often are not included in introductory books. We will also have our first look at C++20, namely mathematical constants that have been added to the C++ Standard Library.

By the end of the chapter, you should be able to write, compile, and run simple C++ programs, understand basic numerical types, and employ

mathematical functions in the Standard Library that are fundamental in just about any quantitative discipline, including finance.

C++ and Quantitative Finance

C++ started its rapid growth in the financial sector around the mid-1990's. Many of us who were in the industry around this time had been raised on FORTRAN, particularly for writing numerical routines and scientific applications. While FORTRAN and its supporting libraries were very well-developed in terms of mathematical and linear algebra support, it lacked support for object-oriented programming.

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 term structure of interest rates for each currency
- A market rate feed of live foreign exchange rate quotes
- Volatility curves or surfaces for movements in FX rates and interest rates
- A set of pricing methods, eg closed form, simulation, or other 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 particular challenge for companies with trading operations spread across the major financial centers of New York, London, and Tokyo, as well as other capital markets. 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 around the world. 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.

Around 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 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 was 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++ also offers certain compile-time optimizations that are not available in these other programming languages.

C++ 11: The Modern Era is Born

In 2011, the Standard C++ Foundation 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
- Task-based concurrency that can parallelize computations without the need for manual thread management
- Smart pointers that prevent memory-related program crashes, without affecting performance

These topics and more will be discussed in the chapters ahead. An excellent reference that covers the history and evolution of C++ into the modern era is also available from O'Reilly: *C++ Today: The Beast is Back*, by Jon Kalb and Gasper Azman [1]. It should also be noted that with more attention to, and promotion of best practices[1] and guidelines[2] by the ISO C++ committee, cross-platform development is now a much easier task than in years past.

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 threeyear cadence, with the most recent release being C++20. This book will primarily cover developments through C++20, particularly those that should be of interest to financial quant developers. Proposals currently in the works for future standards are also mentioned where relevant.

Proprietary and high-frequency trading firms have been at the forefront of adopting the C++11 Standard and later, 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 derivatives pricing models utilized by traders and risk managers at investment banks and hedge funds. The recent random number generation and concurrency features in the Standard Library, for example, provide built-in support for efficient Monte Carlo simulation that is a key component in both evaluating trading strategies and pricing complex exotic options. These tasks used to require many more hours of distributional random number generation code development and time-consuming integration of platform-dependent threading libraries.

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 matrix algebra libraries, and machine learning libraries

such as TensorFlow and PyTorch. We will cover Boost and Eigen in more detail later in the book.

Debunking Myths About C++

There are a multitude of myths about C++. Here are several of the more infamous beliefs, and explanations which debunk them.

- *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. Clinging to C style can in fact hinder learning the powerful abstractions and potential benefits of C++.
- *C++ is too difficult*: There is no doubt that C++ is a rich language that provides plenty of the proverbial rope with which one can hang oneself, but 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 developer in C++ very quickly.
- *Memory leaks are always a problem in C++*: With smart pointers available since C++11, this no longer needs to be an issue in most financial model implementations, as we shall see.

Compiled vs Interpreted Code

As alluded to above, C++ is a compiled language, where commands typed into a file by us mere mortals are translated into 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 individually translated to machine code at run-time, thus slowing down execution time for larger applications.

This is by no means a knock on these languages, as their power is evident in their popularity for rapid implementations of models arising in quantitative fields such as finance, data science, and biosciences, with their built-in mathematical and statistical functions are often compiled in C, C++, or

FORTRAN. However, the financial world at least is replete with stories where a model would require days to run in an interpreted language, where run times could be reduced to a matter of minutes when reimplemented in C++.

An effective approach is to use interpreted mathematical languages with C++ in a complementary fashion. For example, 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. The results can then be used inside powerful plotting and other visualization tools in R that are not available in C++.

Another advantage is that 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 in 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++ packages are available for both R and Python, namely Rcpp and pybind11, respectively. Matlab also provides options for C++ interfaces.

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 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 that is shipped with modern compilers.

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

In addition, C++ is not limited to object-oriented programming; rather, the language also supports the other three major programming paradigms, namely procedural programming, generic programming, and functional programming. Each of these will be discussed in subsequent chapters.

C++ is a strongly-typed language, meaning that before we use a variable, we must declare it by its type. The language provides a variety of numerical types; however, those that we will primarily use are as follows:

Type	Description	Minimum Value	Maximum Value
double	Double Precision	+/- 2.2e-308	+/- 1.8e308
int	Integer	-2,147,483,648	2,147,483,647

Others, such as unsigned and extended integer types, will be introduced later when we need them.

The C++ Standard Library

As Nicolai Josuttis describes it in his indispensable text, *The C++ Standard Library - A Tutorial and Reference, 2nd Edition*[3], 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 quantitative model implementations include:

- Array-style containers, particularly the venerable `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 manages threads internally and safely
- Smart pointers that abstract away the dangers associated with memory management
- A class to store and manage character data
- Streaming functions to take input from and display results to the console

Use of Standard Library components, however, 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 file containing the Standard Library declarations of 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).

Compilers and IDE’s

In order to get started with learning C++, you will need to obtain a compiler and a development environment. The three major modern and freely available compilers, which ship with their implementations of the C++ Standard Library, are:

- The Microsoft Visual Studio 2019 compiler
- Clang (LLVM Project)
- GNU gcc compiler

There are also several integrated development environments (IDE’s) available, namely Visual Studio, Apple’s Xcode (which ships with the Clang compiler), and CLion, a product that typically requires purchase from JetBrains. For this book, Microsoft’s Visual Studio compiler and IDE are highly recommended. They are user-friendly options to get up and running quickly on C++, with very powerful debugging tools.

Furthermore, the Visual Studio option also includes a Clang option that allows a programmer to switch between it and the Microsoft compiler, helping to ensure cross-platform compatibility.

Unfortunately, the Visual Studio option for C++ only exists for Windows, as the Mac version does not ship with a C++ option. In this case, one might opt for downloading Apple’s Xcode, which ships with the Clang compiler. Linux users will typically want to opt for the gcc or Clang compiler.

Basic Review of C++

The following will be a quick review of C++ using some simple code examples. We will also have our first look at a new feature in C++20, namely mathematical constants.

Good Old “Hello World!”

First, here is a “Hello World!” example to get started. The following code will return the message to the screen, and then allow the user to input the name of someone to whom to say hello:

```
#include <iostream>
#include <string>
int main()
{
        std::cout << "Hello World!" << '\n';
        std::string person;
        std::cout << "To whom do you wish to say hello? ";
        std::cin >> person;
        std::cout << "Hello "<< person << "!" << '\n';
        return 0;
}
```

If you want to say hello to your mother, then after compiling and running the code, the screen would resemble the following:

```
Hello World!
To whom do you wish to say hello? Mom
Hello Mom!
```

The main review points here are

- `cout` and `cin`, along with the string class, depend upon including the C++ Standard Library declaration files iostream and string.

  Members of the Standard Library need to be scoped by their namespace `std`. An alternative is to put `using` statements with the namespace scopes at the top of the file, indicating that anytime these elements appear in the code, they are understood to be coming from the std namespace. Also, you may find it easier to type `endl` (end of line) rather than '\n’:

  ```
  #include <iostream>
  ```

```
using std::cout;
using std::cin;
using std::endl;
#include <string>
using std::string;
int main()
{
        cout << "Hello World!" << endl;
        string person;
        cout << "To whom do you wish to say hello? ";
        cin >> person;
        cout << "Hello " << person << "!" << endl;
        return 0;
}
```

- Importing the std namespace into the global namespace with

  ```
  using namespace std;
  ```

  is sometimes used to replace the individual using statements; however, this is not considered good practice, as it can result in naming clashes at compile time. The motivation behind namespaces will be presented in Chapter 3.

- Output to and input from the console is almost never used in production-level financial programming. User input data will typically come from graphical user interfaces (GUIs) or web applications, while market data usually comes from live feeds. Results are typically displayed in the user interface and then stored in a database, such as when a trade in executed.

- We will use `cout` and `cin` to sometimes mimic these inputs, but they should be avoided in production code.

Simple Procedural Programming in C++

The structure of a procedural program should be familiar, namely:

- A `main()` function, which is called first in execution of a program, and
- A set of user-defined functions that contain individual tasks that comprise the program.

In the simplest case, these can all be written in a single executable file containing `main()`.

We first *declare* each user-defined function in a function *declaration* statement, prior to the start of program execution in the `main()` function. A function declaration states its name, return type, and input argument types, followed by a semicolon.

The function *implementations* are written beneath `main()`, each containing a series of commands within open and closed braces. User-defined function calls can then be made within the `main` function, or from other user-defined functions.

Single line comments are indicated by two consecutive forward slashes. The high-level format is shown here:

```
// Function declarations ("//" indicates a comment line)
return_type function_01(input arguments);
return_type function_02(input arguments);
return_type function_03(input arguments);
.
.
.
int main()
{
  // Call each function
  function_01(input arguments);
  function_02(input arguments);
  function_03(input arguments);
.
.
.
}
return_type function_01(input arguments)
```

```
{
  // Do stuff
  // Return something (or void return)
}
return_type function_02(input arguments)
{
  // Do stuff
  // Return something (or void return)
}
return_type function_03(input arguments)
{
  // Do stuff
  // Return something (or void return)
}
.
.
.
```

> **NOTE**
>
> For larger and more robust production applications, we will soon look at writing functions in separate *modules*, using a new feature in C++20, in which the same method of declaring and implementing functions will carry over.

Further details on functions follow in the next two subsections.

Function declarations

C++ functions may or may not return a value; furthermore, they may or may not take input arguments. A function that has no return value is indicated by a `void` return type. For example, if we move our “Hello World” example into a separate function, it would simply output a message to the screen without returning a value when called from the `main` function, so it would be declared as a `void` function. In addition, it does not require any input parameters, so its declaration would take on the form

```
void hello_world();
```

Next, suppose we want to write a real-valued function that takes in a single variable and returns twice its value. In this case, our declaration will have a double precision floating type return, indicated by `double`, and an input of the same type. If we name this function `twice_a_real`, and the input variable `x`, our declaration would be written as

```
double twice_a_real(double x);
```

As a final example, as in other programming languages, a function can take in more than one variable. Suppose we wish to add three integers in a function called `add_three_ints` and return the sum of variables `i`, `j`, and `k`. Integer types are indicated by `int`, so our function declaration would be

```
int add_three_ints(int i, int j, int k);
```

Function implementations

Function implementations, also called function *definitions*, are where we implement the actual commands to display a message to the screen, calculate a mathematical result, or to perform other tasks. The *body* of the function is placed inside braces, as shown here for the `hello_world` function. We again need to indicate the `void` return type.

```
void hello_world()
{
  std::cout << "Hello World!\n";
}
```

Next, we can write the implementations of our two simple mathematical functions. As in their declarations, the `double` and `int` return types, respectively, as well as the types of their input variables, must be included:

```
double twice_a_real(double x)
{
        double y = 2.0 * x;
        return y;
}
int add_three_ints(int i, int j, int k)
{
        return i + j + k;
```

```
}
```

In the first case, we initialize a new `double` variable `y` and with the result of the calculation. Because C++ is a strongly typed language, we need to indicate the type of a variable when it is initialized. This variable is then returned to the `main` function with the result. In the second function, we just put the sum operations in the return statement itself; this is also perfectly legal.

Finally, we put this all together with a `main` function that is called when the program starts and makes calls to our user-defined functions. It goes in between the user-defined function declarations and their implementations below, as shown here:

```
#include <iostream>
// Maybe put in using statements here(?)
void hello_world();
double twice_a_real(double x);
int add_three_ints(int i, int j, int k);
int main()
{
        hello_world();
        double prod = twice_a_real(2.5);
        std::cout << "2 x 2.5 = " << prod << std::endl;
        std::cout << "1 + 2 + 3 = " << add_three_ints(1, 2,
3) << std::endl;
        double r;
        std::cout << "Enter a real number: ";
        std::cin >> r;
        std::cout << "2 x " << r << " = " << twice_a_real(r)
<< std::endl;
        return 0;
}
void hello_world()
{
        std::cout << "Hello World!\n";
}
double twice_a_real(double x)
{
        double y = 2.0 * x;
        return y;
```

```
}
int add_three_ints(int i, int j, int k)
{
        return i + j + k;
}
```

C++ Syntax and Style Guidelines

In this section a review of essential C++ syntax is provided, along with guidelines on code formatting and variable naming. The guidelines discussion might not be high on many people's priority list, but this topic is 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 clean and maintainable state.

We will review essential rules about C++ syntax. Even if you are familiar with some of it already, a summary will be presented in one place that you may find useful.

Code Blocks in Braces

Function implementations, also called *function definitions*, are placed inside braces, as shown in each of the function implementations in 6.2.2 above. When control reaches the closing brace, the function terminates. This is also true for other code blocks such as in conditional statements, loops, user-defined functions, and user-defined classes. When the closing brace is encountered, non-static local variables and objects defined within the block are said to _go out of scope_. That is, they are wiped from memory and no longer accessible. Pointers can be an exception to this rule, but we will discuss this in more detail in Chapter XX.

Syntax Review

Commands and declarations in C++ terminate with a semicolon:

```
double y = 2.0 * x;
```

Again, as C++ is a strongly-typed language, numerical variable types should be indicated before initialization.

```
double x1 = 10.6;
int k;  // Defaults to zero
double y1 = twice_a_real(x1);
```

> **NOTE**
>
> C++11 introduced the `auto` keyword that can automatically deduce a variable or object type, as well as uniform initialization (with braces). Varied opinions on their use exist, but many programmers still prefer to explicitly state plain old data (POD) types such as `int` and `double` to avoid ambiguity. This will be the style followed in this book. `auto` and uniform initialization will be discussed later within contexts where they tend to be more useful.

One-line comments are indicated with two forward slashes, eg,

```
// This is a comment
```

Multiple lines of comments in a block can also be commented out, as follows:

```
/*
    Owl loved to rest quietly whilst no one was talking
    Sitting on a fence one day, he was surprised when
    suddenly a kangaroo ran close by.
*/
```

There is no difference to the compiler between a single space or multiple spaces; for example, despite the variations in whitespace, the following code is legal:

```
  int j = 1101;
  int k=   603;
  int sum = j +    k;
  std::cout <<    "j + k = " <<   sum << "\n";
```

A well-known mantra in programming, however, and particularly relevant to C++, is *just because you can do something, doesn't mean you should*. The

above code will be more readable and maintainable if written with clear and consistent spacing:

```
int j = 1101;
int k = 603;
int sum = j + k;
std::cout << "j + k = " << sum << "\n";
```

Again, for more realistic and complex code, this mantra should be kept in mind. It will be a recurring theme throughout this book.

Code may also be continued onto multiple lines without the use of a continuation character, and vertical spaces are ignored. Returning to our previous example, writing

```
int j = 1101;
int k =
        603;

int sum = j + k;
std::cout << "j + k = "
          << sum
          << "\n";
```

would yield the same result. As before, the preceding example, with uniform spacing and each command placed in a single line, would be preferable. However, it should be noted that, in quantitative programming where complex and nested calculations are involved, it often becomes highly advisable to split up formulae and algorithms on multiple lines for clarity and code maintainability. We will see examples of this in subsequent chapters.

Finally, C++ syntax is *case sensitive*. For example, two `double` variables `x` and `X` would be as different as two other variables `kirk` and `spock`. The same applies to function names. In examples above, we used the Standard Library function `std::cout`. Attempting to write `std::Cout` instead would trigger a compiler error.

Naming Conventions

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.
- Other than the underscore character, special characters, such as `@`, `=`, `$` etc are not allowed.
- Spaces are not allowed. Names must be contiguous.
- Language keywords are not allowed in naming, such as `double`, `if`, `while`, etc. A complete listing can be found on https://en.cppreference.com/w/cpp/keyword.

The maximum name length is compiler-dependent, and in at least one case – the GNU gcc compiler – imposes no limitation; however, see the *mantra* discussed above.

Single letter variable and function names are fine for simple examples and plain mathematical functions. However, for quantitative models, 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.

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 the most typical of what is found in C++ function and variable names, and class names are usually in Upper Camel form. In recent years – likely propelled by Google's C++ Style Guide [5] – variable and function names have gravitated more toward the snake case. As such, we will adopt this convention in this book, and use Upper Camel for

class names.

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.

Mathematical Operators, Functions, and Constants in C++

While the previous discussion was loads of fun, our focus in this book is on math and finance. We have already used the mathematical operators for addition and multiplication of built-in C++ numerical types above. These are language features in C++, and a comprehensive discussion of these standard operators follows. Common mathematical functions, however -- such as cosine, exponential, etc -- are provided in the C++ Standard Library rather than in the core language.

Standard Arithmetic Operators

As suggested in the examples above, addition, subtraction, multiplication, and division of numerical types are provided in C++ with the operators `+`, `-`, `*`, and `/`, respectively, as usually found in other programming languages. In addition, the modulus operator, `%`, is also included. 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 = 30.6;
double x2 = 8.74;
double y = x1 + x2;
double z = x1 - x2;
double twice_x2 = 2.0 * x2;
```

The order and precedence of arithmetic operators are the same as found in most other programming languages, 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:

```
x1 + twice_x2/x2
```

Using the above double precision values would result in 30.6 + 2.0 = 32.6

- Use round brackets to change the precedence:

```
(x1 + twice_x2)/x2
```

This would yield $\frac{30.6+8.74}{8.74} = 4.50$

with the same double precision values.

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 finance include the following, where `x` and `y` are assumed to be double precision variables:

`cos(x)`	cosine of *x*

`sin(x)`	sine of *x*
`tan`	tangent of *x*
`exp`	exponential function e^x
`log`	natural logarithm *ln(x)*
`sqrt`	square root of *x*
`cbrt`	cube root of *x*
`pow`	*x* raised to the power of *y*
`hypot`	computes $\sqrt{x^2 + y^2}$ for two numerical values x and y

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:

```
#include <cmath>      // Put this at top of the file.
double trig_fcn(double theta, double phi)
{
  return = std::sin(theta) + std::cos(phi);
}
```

Again, if you don’t feel like typing out `std::` all the time, putting `using` statements after the `include` statement are also fine:

```
#include <cmath>      // Put this at top of the file.
using std::sin;
using std::cos;
double trig_fcn(double theta, double phi)
{
  return = sin(theta) + cos(phi);
```

```
}
```

We can also now write our first finance example. We want to price a zero coupon bond

Ae^{-rt}

where

A = the face value of the bond,

r is the interest rate, and

t is the time to maturity as a year fraction.

In C++, we could then write

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

For a more comprehensive list of Standard Library math functions, again see Josuttis, *The C++ Standard Library (2E)* [4], Section 17.3, or the listing available on the CppReference website [6]. Both are indispensable references for any modern C++ developer and are highly recommended advanced complementary resources for this book. Some additional guidance on the use of Standard Library math functions follows in the next two sections.

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. Instead, one needs to call the Standard Library `std::pow` function in `cmath`. When computing polynomials, however, it is more efficient to apply factoring per Horner's Method and reduce the number of multiplicative operations[6]. For example, if we wish to implement a function

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

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

```
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;
}
```

For the case of a non-integer exponent, say

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

then there is no alternative but to use `std::pow`:

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

` < *cmath* >` *Ensures Consistency Across Compilers*

It may be the case that you can use these math functions without `#include <cmath>`, but one should adhere to including `cmath` and scoping the functions with `std::`. First, because C++ is built upon C, some compilers retain the old math functions from C in what is called the *global namespace*. Other compilers, however, might put `cmath` into the global namespace. As a result, one might actually be calling old C functions rather than the ISO C++ Standard versions, and this could cause unexpected or inconsistent behavior among different compilers.

Another example of inconsistencies that can arise is with the absolute value function. In C, and on older C++ compilers, the `abs` function was only implemented for integer types. In order to calculate the absolute value of a floating point number, 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.

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, we should always put `#include <cmath>`, and scope the math functions with `std::`. This will help ensure cross-compatibility on different compilers and operating system platforms.

> **NOTE**
>
> Note: Regarding C headers and namespace std, this is clarified, for example, in the specifications for the gcc compiler:
>
> *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.*[8]

Constants

In any type of quantitative programming, there is often a need to use constant values in calculations. In C++, one can define a constant by simply appending the keyword `const` when a value is assigned. Furthermore, beginning with C++20, a set of commonly used mathematical constants is now available.

The `const` Keyword

If a variable doesn’t change value, it is safer to declare it as a constant type, by using the `const` keyword. For example, we could use it to store an approximation of earth’s gravitational acceleration constant:

```
const double grav_accel = 9.80665;
```

Then, if later within the same scope someone attempted to reassign it to a different value:

```
grav_accel = 1.625;     // Gravitational constant for the
moon
```

a compiler error would result, with a message indicating an attempt was made to modify the value of a constant. Catching errors at compile time is better than chasing them at runtime and tracking down the cause, especially in a live production environment.

`const` also has other important uses and interesting properties that we will cover later, particularly in an object-oriented programming context.

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.

C++ constant	`e`	`pi`	`inv_pi`	`inv_s
Definition	*e*	π	$\frac{1}{\pi}$	$\frac{1}{\sqrt{\pi}}$

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 some_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 consistencies 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 at least mentioning that these constants are set at *compile time* rather than runtime, using a C++11 designation called `constexpr`. This ties in with the much broader and more advanced subject of *template metaprogramming*, in which calculations of constant values to be used at runtime are performed at compile time. [[Might return to this topic later, although it is of limited used in financial modeling where the computations depend on data only available at runtime]].

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 more commonly present in statistical calculations. [[See later chapter on the Boost libraries – they are included there]].

Conclusion

This concludes our whirlwind overview of C++. We emphasized quantitative programming, along with the mathematical constants now included in C++20.

Our coverage of best practices with respect to coding style will be a consistent theme throughout the book, as C++ is an extremely feature-rich language with plenty of the proverbial rope with which to hang oneself. Adhering to best practices and consistent coding style is vital to ensure code maintainability and reliability.

One other point to remember is that while we use a lot of screen output and input, this is not how C++ is typically used in quantitative development. `std::cout`, and `std::cin` should be thought as placeholders for real-world interfaces. We will continue to use them as devices to check our results, but they will mostly be relegated to use within the test functions that are called from `main()`, rather than within mathematical and models code itself where they should be avoided in practice anyway.

References

[1] Kalb and Azman, *C++ Today: The Beast is Back*, available on https://resources.jetbrains.com/storage/products/cpp/books/Cplusplus_Today.pdf (link)

[2] Guideline Support Library (ISO) (link)

[3] ISO C++ Coding Standards (link)

[4] Nicolai Josuttis, *The C++ Standard Library (2E)* (link)

[5] Google C++ Style Guide
(https://google.github.io/styleguide/cppguide.html)

[6] cppreference.com

[7] Stepanov, Mathematics of Generic Programming (Horner's Method)

[8] GNU gcc Compiler Documentation

(https://gcc.gnu.org/onlinedocs/libstdc++/manual/using_headers.html

Chapter 2. Some Mechanics 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 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*.

Just about any programming language will have some form of an array structure for storing collections of like types. In C++, there are a handful of options for this purpose, but it is the Standard Library `vector` container that is by far the most-used type. In this chapter, we will see how a `vector` can conveniently represent a vector of real numbers in the mathematical sense. We will also go through the basics of creating and using a `vector` and its key member functions, as it will tie in well with iterative statements, such as counted loops and `while` statements, also to be covered in this chapter.

Control structures, include both iterative statements and conditional. Conditional branching in C++ can be implemented in `if` statements, similar to other languages, as well as in what are called `switch` statements. A good complementary topic related to the `switch` statement is that of enumerated types (enums), particularly the more modern *enum classes* that were added in

C++11. Enum classes are also well-suited for facilitating data input to and output from financial models.

Finally, we will wrap up with a summary of aliases that can be used in C++, and why they are important. This includes type aliases that can add clarity to the code, in place of longer and sometimes more cryptic templated types. References and pointers allow you to access and modify objects without the overhead of object copy, although pointers have wider-ranging uses as well.

The `vector` Container

The `vector` container in the C++ Standard Library is the go-to choice for storing and managing an indexed array of like types. It is particularly useful for managing vectors of real numbers that are ubiquitous in quantitative work, with `double` types representing the numerical values.

> **NOTE**
>
> **Historical Note**: A `vector` is more specifically part of what is called the Standard Template Library (STL). The STL was developed independently of Bjarne Stroupstrup's early efforts in the 1980's and 90's to design and produce C++, by researcher Alexander Stepanov. The history behind acceptance the STL and its acceptance into the C++ Standard is a very interesting one [[see Kalb/Azman]], but the upshot is the STL – based on generic programming – was accepted into the theretofore object-oriented focused C++ for its first ISO standard release in 1998.

Being a generic container, `std::vector` can hold elements of a common arbitrary type, ranging from plain old data (POD) types such as `double` and `int`, to objects of user-defined and library classes.

```
#include <vector>
using std::vector;
//. . .
vector <double> x;                  // Vector of real numbers
```

```
vector <BondTrade> bond_trades; // Vector of user-defined
BondTrade objects
```

The type to be held is indicated inside the angle brackets.

> **NOTE**
>
> The angle brackets indicate the *template parameter*. Templates are the means by which C++ implements generic programming. This topic will be discussed in further detail in Chapter 7.

Setting and Accessing Elements of a `vector`

An STL `vector` essentially encapsulates and manages a dynamic array, meaning that elements can be appended to it or removed from it after it is constructed. The `vector` also supports random access, meaning an element can be accessed, and moreover modified, by the index of the element. Like everything else in C++, a `vector` is zero-indexed, meaning that the index of its first position is index 0, and its last position is index *n* - 1, if it holds *n* elements.

Creating a `vector` and Using its Index

The following instruction will create a vector holding three real numbers.

```
vector <double> v(3);   // Will hold three elements
```

The `vector` can be populated element by element as shown here. Note that indexing starts with zero rather than one.

```
v[0] = 10.6;            // Set the first element (index 0)
and assign to 10.6
v[1] = 58.63;           // Set the second element (index 1)
and assign to 58.63
v[2] = 0.874;           // Set the first element (index 2)
and assign to 0.874
```

The index is indicated by square brackets. We can also change the values by simply reassigning an element to a new value; viz,

```
v[1] = 13.68;
```

It is also possible to initialize vectors using *uniform initialization* introduced in C++11. Also known *as braced initialization*, use of the assignment operator is optional:

```
vector <double> w{9.8, 36.8, 91.3, 104.7}; // No assignment
operator
vector <int> q = {4, 12, 15};    // With assignment operator
```

> **NOTE**
>
> The addition of uniform initialization to C++11 has had a significant impact on the language, beyond simply initializing a vector. It has some interesting and convenient properties that will be discussed in Chapter 4.

Member Functions

As `vector` is a class, it holds a number of public member functions, including three: `at`, `size`, and `push_back`.

The `at` Function

The `at` function essentially performs the same roles as the square bracket operator, namely access of an element for a given index, or to modify the element.

```
double val = v.at(2);   // val = 0.874
w.at(1) = val;                  // 36.8 in w[1] is replaced
with 0.874
```

The difference between using the square bracket operator and the `at` function is the latter performs bound checking. Two examples are

- Attempting to access an element that exceeds the maximum index; eg,

  ```
  double out_of_range = v.at(100);        // 2 is the max
  index for v
  ```

- Attempting to use a negative index value

```
w.at(-3) = 19.28;
```

In each case, an exception will be thrown that can be used in error handling. Otherwise, you can just think of `at` and `[.]` as the same.

The `size` Function

The name of this member function makes it fairly obvious what it does: it returns the number of elements held by a `vector`:

```
auto num_elems_w = w.size();    // Returns 5
auto num_elems_q = q.size();    // Returns 3
```

You may notice this is the first time we have used the `auto` keyword. What this does is automatically deduce the type returned from the `size` function. We will see in future cases how useful `auto` can be, but here, it helps us get around the fact that the maximum size of a `std::vector` container will vary depending upon compiler settings and the platform you are using. The type will be some form of an unsigned (non-negative) integer, of which there are multiple sizes.

So as to not get into the weeds here, we don’t need to be concerned with the specific unsigned type here, so we can mostly just use `auto` for the return type of the `size` member function.

The `push_back` Function

This function will append elements to a `vector`; that is, new elements are “pushed onto the back” of the container.

For example, we can append a new element to the `vector v` above, say 47.44:

```
v.push_back(47.44);
```

Now, `v` contains four values: 10.6, 58.63, 0.84, 47.44, with `v[3]` (fourth element, using 0-indexing) equal to the new value.

We can also append values to an empty vector. At the outset, we defined

```
vector <double> x;              // x.size() = 0
```

Now, if we append a value,

```
x.push_back(3.08);      // x.size() = 1
```

`x` now contains the value 3.08 in its index 0 position and contains one element. This can be repeated arbitrarily many times:

```
x.push_back(5.12);      // x.size() = 2
x.push_back(7.32);      // x.size() = 3
//. . . etc
```

To close the discussion on `push_back`, there is one potential gotcha to be aware of,

Suppose we create a `vector` of integers with three elements:

`vector <int> ints(3);`

Now, each element will hold the default value of an `int` type: 0.

If we then apply the `push_back` function to append, say, 5:

`ints.push_back(5);`

this value will be appended as a *new element* following the third zero; ie, the vector now contains four elements.

```
0 0 0 5
```

To put a value into any of the first three positions, you will need to use the index explicitly; eg

```
ints[0] = 2;
ints.at(2) = 4;
```

Concluding Remarks on STL `vector`s

In the examples above, we only used plain old numerical types `double` and `int`. Vectors of real numbers are of course fundamental for computational work, but keep in mind an STL `vector` is generic, in that it can hold elements of any valid type, including objects rather than just numerical data

types, as we shall see in more advanced contexts.

Also, as mentioned earlier, in real-world production level programming, inputs are taken from function arguments that come from market and product data, and user input, not hard-coded values as seen in the previous examples. One might find vectors set with fixed numerical values in test functions, but they should be avoided in production code.

The Standard Library contains additional STL containers, plus a large set of STL *algorithms* that are now a core component of modern C++ programming. These will be discussed in greater detail in Chapter 7, and becoming familiar with the basics of the `vector` container now will make this material more accessible when we get to it.

Finally, to reiterate, prefer using an STL `vector` over a dynamic C-style array using `new` and `delete`. There is no performance benefit to using the latter, and memory management is all encapsulated inside the `vector` class, freeing the developer from risks due to memory leaks.

Enum Constants and Classes

Enumerated constants, more commonly called enums for short, map text to integers. Prior to C++11, enums were a great means of making it clearer for us mere mortals to comprehend integer codes by representing them in (contiguous) words. It was also far more efficient for the machine to process integers rather than 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 *enum classes*. These remove ambiguities that can occur with overlapping integer values when using regular enum constants, while preserving the advantages.

We will discuss the motivation for preferring the more modern enum classes over integer-based enums. In the next section we will see how they can be used to our advantage in conditional statements. Later on, they will prove useful in making data input and output with financial models more robust.

Enum Constants

Enums allow us to pass around identifiers, classifications, indicators etc in text representation, while behind the scenes, the compiler recognizes them as integers.

As an example, we can create an enum called `OptionType` that will indicate the types of option deals that are allowed in a simple trading system, eg European, American, Bermudan, and Asian. The `enum` type is declared; then, inside the braces, the allowable types are defined, separated by commas. By default, each will be assigned an integer value starting at zero and incremented by one (remember that indexing in C++ is zero-based). 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
};
```

We can then verify that in place of each option type, its corresponding integer value is given:

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

Checking the output, we get:

```
European
American
Bermudan
Asian
```

So, we can see how the program treats the text representations as integers. Note that these text labels are *not* enclosed in quotation marks, as they

ultimately represent integer types, not strings.

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 type member. 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 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 ≠ 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.

Enum Classes

A new and more robust way to avoid `enum` overlaps was introduced in C++11 that eliminates the integer representation altogether. The other benefits of enums, such as avoiding cryptic numerical codes and larger string objects, still remain, but the conflicts are avoided by using what is called an *enum class*. As an example, we can define bond and futures contract categories within enum classes, 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:

```
if(Bond::Corporate == Futures_Contract::Gold)
{
```

```
        // . . .
}
```

This works to our advantage, as it is much better to catch an error at compile time rather than runtime. Modern best practices now maintain that we should prefer using enum classes rather than enumerated constants [[refer to ISO Guidelines]].

Control Structures

Control structures consist of two categories:

- Conditional branching, such as `if` statements
- Iterative controls that repeat a set of commands in a loop

In C++, the code that pertains to a given condition or sequence is contained in a block defined by braces. Similar to a function, variables declared within a block will go out of scope when the block terminates. These structures can also be nested within one another.

It was assumed in the previous section on enums and enum classes that you are familiar with the basics of `if` conditions, but here you can read through a more comprehensive review of conditional and iterative constructs that will be utilized heavily from here on out. Both depend on logical operators determining a true or false condition, so before launching into our tour of control structures, a quick review of logical operators and Boolean types are in order.

The C++ boolean type, represented represented by `bool`, can store a value of either `true` or `false`. Behind the scenes, a `bool` type has a size of one byte and may store only `1` for `true`, or `0` for `false`. Note that `true` and `false` are not placed in quotations, as ;ole enums they are not character types. They represent fixed integer values.

The C++ operators for equality and inequalities will return a `bool` type based on whether the result is true or false. They are as follows:

- `<, >` Strict inequality
- `<=, >=` Inclusive inequality
- `==` Equality
- `!=` Not equals
- *And* and *Or* operations are represented by `&&` and `||` respectively.

Examples will follow in the next section on conditional branching.

Conditional Branching

C++ supports both the usual `if` based logic found in most other languages, and `switch`/`case` statements that offer a cleaner alternative to multiple `else if` conditions in special cases.

`if` and Related Conditions

The usual conditional branching statements

- `if (condition) then (action)`
- `if (condition) then (action), else (default action)`
- `if (condition 1) then (action 1),`
- `else if (condition 2) then (action 2)`
- `...`
- `else if (condition n) then (action n)`
- `else (default action)`

are represented by the following C++ syntax. Each condition, whether it be `if`, `else if`, or `else`, the code that gets executed for a `true` condition is contained within a separate body, indicated by open and closed braces.

```
// Simple if
if (condition)
```

```
{
  // action
}
// if/else
if (condition)
{
  // action
}
else
{
  // default action
}
// if/else if.../else
if (condition 1)
{
  // action 1
}
else if (condition 2)
{
  // action 2
}
// ...
else if (condition n)
{
  // action n
}
else
{
  // default action
}
```

TIP

In conditional statements containing `else if`, it is a best practice to include a default `else` block at the end. Without it, code may build without any complaints from the compiler and run just fine, but its execution could very easily result in unexpected behavior that can cause major headaches in larger and more realistic code bases.

Utilizing the inequality operators introduced above, we can then write some

simple examples with all three variations on the `if` statement theme:

```
int x = 1;
int y = 2;
int z = 3;

// Simple if
if (x > 0)
{
        cout << x << " > 0" << endl;
}
// if/else
if (x >= y)
{
        cout << x << " >= " << y << endl;
}
else
{
        cout << x << " is less than " << y << endl;
}
// if/else if.../else
if (x == z)
{
        cout << x << " == " << z << endl;
}
else if (x < z)
{
        cout << x << " > " << z << endl;
}
else if (x > z)
{
        cout << x << " < " << z << endl;
}
else
{
        cout << "Default condition" << endl;
}
```

WARNING

Due to the nature of floating point numerical representation and arithmetic, one should never test for exact equality between two `double` types, nor should

floating point types be compared identically to zero. These cases will be covered later in a separate context.

The operators for logical AND and OR can also be used within conditional arguments. For example:

```
        #include <cmath>
        using std::abs;
        using std::exp;

        // Simple if
        if (x > 0 || y < 1)
        {
                cout << x << " > 0 OR " << y << " < 1 " <<
endl;
        }
        // if/else if.../else
        if (x > 0 && y < 1)
        {
                cout << x << " > 0 AND " << y << " < 1 " <<
endl;
        }
        else if (x <= 0 || y >= 1)
        {
                cout << x << " <= 0 OR " << y << " >= 1 " <<
endl;
        }
        else if (z <= 0 || (abs(x) > z && exp(y) < z))
        {
                cout << z << " <= 0 OR " << endl;
                cout << abs(x) << " > " << z << " AND "
                             << exp(y) << " < " << z <<
endl;
        }
        else
        {
                cout << "Default condition" << endl;
        }
```

Note that in the last `else if` condition, we put the AND condition inside round brackets, as OR takes precedence over AND. [cppreference.com]

Finally, we can assign logical conditions to `bool` variables, and used within `if` conditions, as shown here:

```
        bool cond1 = (x > 0 && y < 1);
        bool cond2 = (z <= 0 || (abs(x) > z && abs(y) < z));
        if (cond1)
        {
                cout << x << " > 0 AND " << y << " < 1 " <<
endl;
        }
        else if (!cond1)
        {
                cout << x << " <= 0 OR " << y << " >= 1 " <<
endl;
        }
        else if (cond2)
        {
                cout << z << " <= 0 OR " << endl;
                cout << abs(x) << " > " << z << " AND "
                        << abs(y) << " < " << z << endl;
        }
        else
        {
                cout << "Default condition" << endl;
        }
```

Note that a boolean variable can be negated simply by preceding it with the `!` operator, as shown in the first `else if` condition above.

WARNING

A common trap is to mistakenly use `=` to test equality instead of `==`. The former is the assignment operator and will cause unexpected behavior in this case. Be sure to use `==` when testing for equality.

Ternary `if` Statement

There is also a convenient one-line shortcut for short and sweet `if-else` combinations. The syntax is as follows:

type `var = ` *logical condition* `? var_val_true`(if `true`) : `var_val_false`(if `false`);

In English, this means if _*logical condition*_ is `true`, assign the value `var` to `var_val_true`; otherwise, assign it to `var_val_false`.

A code example should make this clearer:

```
using std::sin;
using std::cos;
int j = 10;
int k = 20;
double theta = 3.14;
double result = j < k ? sin(theta) : cos(theta);
```

So, in this example, `result` would be assigned the value of sin(3.14), or approximately zero.

The `switch`/`case` Statement

Also known as just a `switch` statement, this control sequence allows us to eliminate some of the clutter that comes with multiple `else if` clauses, but for the particular case of branching on the *state* of a single integer type, or alternatively, either an enum that maps to an integer, or an enum class member.

For each possible `case`, the command that follows the matching state is executed. As with the `else` condition above, a `default` action should be provided at the end to catch cases that do not fall into any of the given categories, or handle the error if no other possibilities are admissible.

As a first example, consider a case where we pretend an integer condition represents a type of option, and in place of each `cout`, the action would be to call a corresponding pricing model. This will render our code more readable and maintainable than using multiple `else if` statements.

```
void switch_statement(int x)
{
        switch (x)
        {
        case 0:
```

```
                cout << "European Option: Use Black-Scholes"
<< endl;
                break;
        case 1:
                cout << "American Option: Use a lattice
model" << endl;
                break;
        case 2:
                cout << "Bermudan Option: Use Longstaff-
Schwartz Monte Carlo" << endl;
                break;
        case 3:
                cout << "Asian Option: Calculate average of
the spot time series" << endl;
                break;
        default:
                cout << "Option type unknown" << endl;
                break;
        }
}
```

After each case, the `break` statement instructs the program to exit the `switch` statement once the corresponding code for a particular state is executed. So if `x` is `1`, a lattice model would be called to price an American option, and then control would pass out of the body of the `switch` statement rather than checking if `x` is `2`.

There are also cases where one might want to drop down to the next step if the same action is desired for multiple states. For example, in (American) football, if a drive stalls, the offense punts the ball on fourth down and no points are scored. If the team scores, however, it might have kicked a field goal for three points, or scored a touchdown with three possible outcomes:

- Miss the extra point(s) -- Result is six points
- Kick the extra point -- Result is seven points
- Score a two-point conversion -- Result is eight points

No matter how a team scores, it kicks the ball off to their opponent, so for cases 3, 6, 7, and 8, we just drop down through each case until we hit the

kickoff. This quasi-Bayesian logic could then be implemented with the following code:

```
void switch_football(int x)
{
        switch (x)
        {
        case 0:         // Drive stalls
                cout << "Punt" << endl;
                break;
        case 3:         // Kick field goal
        case 6:         // Score touchdown; miss extra
point(s)
        case 7:         // Kick extra point
        case 8:         // Score two-point conversion
                cout << "Kick off" << endl;
                break;
        default:
                cout << "Are you at a tennis match?" << endl;
                break;
        }
}
```

An obvious pre-C++11 alternative for the `switch` on option pricing `case`s would be to substitute in the corresponding enums for the integer codes, thus making the logic even easier to understand for human consumption (`cout` messages remain the same):

```
void switch_statement_enum(OptionType ot)
{
        switch (ot)
        {
        case European:          // = 0
                cout << "European Option: Use Black-Scholes"
<< endl;
                break;
        case American:          // = 1
                . . .
        case Bermudan:          // = 2
                . . .
        case Asian:             // = 3
                . . .
```

```
        default:
                cout << "Option type unknown" << endl;
                break;
        }
}
```

However, modern ISO Guidelines now favor using enum classes, for the reasons demonstrated above with integer conflicts. So, we just substitute the `Options_Contract` enum class into the preceding example to get:

```
void switch_enum_class_member(Options_Contract oc)
{
        switch (oc)
        {
        case Options_Contract::European:
                cout << "European Option: Use Black-Scholes"
<< endl;
                break;
        case Options_Contract::American:
                . . .
        case Options_Contract::Bermudan:
                . . .
        case Options_Contract::Asian:
                . . .
        default:
                cout << "Option type unknown" << endl;
                break;
        }
}
```

Iterative Statements

In C++, there are two built-in language features that enable looping logic and iteration:

- `while` and `do...while` loops
- `for` loops (including range-based `for` loops)

These iterative commands will execute a repeated block of code over a set of values or objects based on a fixed count, while a logical condition is true, or

over a range of elements held by a `vector`.

`while` and `do...while` Loops

The essential workflow behind a `while` loop is to repeat a block of code *while* a logical expression is `true` (or alternatively, whilst `false`). The following simple example demonstrates a simple `while` loop, where the incremented value of an integer is output to the screen while its value remains strictly less than some fixed maximum value:

```
int i = 0;
int max = 10;
while (i < max)
{
        cout << i << ", ";
        ++i;
}
```

Our logical condition is for `i` to be strictly less than the value `max`. As long as this condition holds, the value of `i` will be incremented

A `do...while` loop is similar, except that by placing the `while` condition at the end, it guarantees that at least one iteration of the loop will be executed. For example:

```
int i = 0;
int max = 10;
do
{
        cout << i << ", ";
        ++i;
} while (i < max);
```

Note that even if `max` had been set to zero or less, there would still be one trip through the `do...while` loop, as the maximum condition is not checked until the end. This is the distinction that separates it from the simpler `while` loop.

In time, we will see looping examples that involve more interesting mathematics and financial applications.

The `for` Loop

This construct is another form of iteration over a countable range. The form that is employed in C++ can be summarized in the following pseudocode example:

```
for(initial expression executed only once;
exit condition executed at the beginning of every loop;
loop expression executed at the end of every loop)
{
DoSomeStuff;
}
```

The syntax here is important, namely the semicolons separating the three expressions in the `for` argument. Breaking this down into parts a, b, and c, we would have

```
for(a; b; c)
```

Each of these parts is typically dependent on some form of a counter, such as an `int i` counter as seen in the `while` statement; however, we now move this index into the argument of the `for` statement, which allows us to remove the increment from the body of the loop. The (a) part determines the starting value of the counter, (b) indicates where to stop, and (c) enforces how the counter is increased or decreased.

For example, we could rewrite the `while` example above using a `for` loop as follows:

```
int max = 10;
for(int i = 0; i < max; ++i)
{
        cout << i << ", ";              // we no longer need
++i in the body
}
```

The results will be exactly the same as the those in the `while` loop examples.

1. There technically is a difference between the pre- and post- increment

operator that can affect other uses, but either `++i` with `i++` in the `for` will work identically. It is generally preferred to use `++i`

2. It is also legal to have a `for` loop where a decrement (`--`) is used to decrease the index value down to some minimum value.

`break` and `continue`

In iterative loops, it is sometimes necessary to break out of a loop before a maximum or minimum index value is attained, or before the specified logical condition would otherwise terminate the iteration. A prime example in computational finance is barrier option pricing using Monte Carlo simulation. The simulation paths will typically have the same number of time steps; however, in the case of an up-and-out barrier option, for example, we would need to break out of the loop if the underlying asset price rose above the barrier level.

This is accomplished by applying the same `break` command as used in `switch` statements. A simple example is shown here, which also demonstrates nesting an `if` condition inside a `for` block:

```
int max = 10;
for (int i = 0; i < 100; ++i)
{
        cout << i << ", ";
        if (i > max)
        {
            cout << "Passed i = " << max << "; I'm tired, so
let's go home."
                 << endl;
            break;
        }
}
```

Once `i` is incremented to 11, the `if` statement is true, so the `break` command is called, causing the program control to exit the `for` loop.

There is also the `continue` keyword that can be used to continue the process of the loop, but since this is the default behavior of a loop anyway, its usefulness is limited.

Nested Loops

In addition to nesting `if` conditions inside loops, it is also possible to nest iterative blocks inside other blocks, whether they be `for` or `while` loop. In quantitative programming, it is easy to find oneself writing double and sometimes even triple nested loops when implementing common numerical routines and financial models. This type of coding, however, can become complex and error prone in a hurry, so one needs to take special precautions, as well as consider alternatives we will take up later.

Range-Based `for` Loops

Prior to C++11, iterating through a `vector` would involve using the index as the counter, up to the number 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] or v.at(i). . .
}
```

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 `elem` in `v`, do something with it”:

```
for(auto elem : v)
{
        // Use elem, rather than v[i] or v.at(i)
}
```

As a trivial example, calculate the sum of the elements:

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

And we are done. No worries about making a mistake with the index, there is less to type, and the code more obviously expresses what it is doing. The ISO Guidelines in fact tell us to prefer using range-based `for` loops with `vector`

objects, as well as other STL containers that will be discussed in Chapter 7.

Aliases

Aliasing can take on several forms, the first being one of convenience, namely *type aliasing*, where commonly used parameterized type names can be assigned to a shorter and more descriptive alias names.

In addition, *reference aliases* help to avoid copies of objects being created when they are passed into functions, often resulting in significant speedup at runtime.

Pointers can also be considered as aliases, particularly useful for representing an active object in class design (the `this` pointer in Chapter 4). Pointers (and now smart pointers) can also be used for allocating memory that persists, but this is a separate and deeper discussion that will be deferred until Chapter 6.

Both references and pointers can help facilitate the *object-oriented programming* concepts of *inheritance* and *composition* that will be presented in subsequent chapters.

Type Aliases

`std::vector<double>` objects are ubiquitous in quantitative code for fairly obvious reasons. Because it is used so much, it is common to assign a type alias to it, such as `RealVector`. This better expresses what it is mathematically, plus we don’t need to bother with as much typing.

Using modern C++, we can define the alias `RealVector` by simply defining it as follows:

```
using RealVector = vector<double>;
```

Then, we could just write, for example:

```
RealVector v = {3.19, 2.58, 1.06};
v.push_back(2.1);
v.push_back(1.7);
// etc...
```

As long as the alias is defined before it is used in the code, then it’s fair game.

Prior to C++11, this application of the `using` command did not exist, so type aliasing was accomplished by using the `typedef` command; eg,

```
typedef vector<double> RealVector;
```

This is also valid C++ and is still found in many modern code bases, but the `using` form is preferable per the modern ISO Guidelines. The detailed reason for this is outside the scope of this book, but the upshot is `using` can be used to define aliases of generic templated types (eg, not just `double` parameters as above), while `typedef` cannot.

References

A reference, put simply, provides an alias for a *variable*, rather than a type. Once a reference is defined, then accessing or modifying it is exactly the same as using the original variable. A reference is created by placing an ampersand between the type name and the reference name before assigning it to the original variable. For example:

```
int original = 15;
int& ref = original;    // int& means "reference to an int"
```

At this point, both `original` and `ref` would return 15 if accessed in a function or assigned to another variable. However, reassigning `original` to 12 would also mean `ref` now returns 12. Similarly, reassigning `ref` would change the value held by `original`:

```
original = 12;                  // ref now = 12
ref = 4;                               // original also now
= 4
```

It is important to note that a reference must be assigned at the same time it is declared. For example,

```
int& ozone;
```

would be nonsense as there is nothing to which it refers, and the code would

fail to compile. Also, once a reference is defined, it cannot be reassigned to another variable for the remainder of its lifetime.

Using a reference for a plain old numerical type is trivial, but they become important when passing large objects into a function, so as to avoid object copy that can decimate a program’s runtime performance.

Suppose we have a `std::vector` containing 2000 option contract objects? By passing it as a reference into a function, the original object itself can be accessed without copying it.

There is one caveat, however. Remember that if a reference is modified, so is the original variable to which it refers. For this reason, one can just make the reference argument `const`. Then, any attempt to modify the reference will be prevented by the compiler.

For example, here are two functions that take in a `std::vector<int>` object as a reference argument. The first one returns the sum of the elements, so there is no modification of the elements attempted. The second one, however, attempts to reset each element to twice its value and then sum the elements. This will result in a compiler error – much better than a runtime error – and prevent the operations from ever being executed:

```
// This is OK
using IntVector = std::vector<int>;
int sum_ints(const IntVector& v)
{
        int sum = 0;
        for (auto elem : v)
        {
                sum += elem;
        }

        return sum;
}
int sum_of_twice_the_ints(const IntVector& v)
{
        // Will not compile!  const prevents modification
        // of the elements in the vector v.

        int sum = 0;
```

```
        for (auto elem : v)
        {
                elem = 2 * elem;
                sum += elem;
        }

        return sum;
}
```

NOTE

It is also possible to pass a function argument as non-`const` reference, with the intent to modify it in place. In this case, one would typically make the return type `void`, instead of returning a modified variable. This is rarely justified anymore in modern C++ due to *return value optimization* (*RVO*). With RVO, objects by default are returned "in place" rather than as copies from functions. This is now a requirement for compilers per the ISO standards, beginning with C++11.

One final point about references relates to managed languages such as Java and C#, in that the default behavior is to pass objects by non-constant reference. In C++ the default is to pass by value; hence, one must specifically instruct the compiler to expect a function argument as a reference with the `&`. This is an adjustment that a programmer needs to make if switching between C++ and a managed language.

Pointers

A *pointer* in C++ shares some similarities with a reference, in that it can also be an alias to another variable, but rather than being permanently tied to a variable throughout its lifetime, a pointer *points* to a memory address containing the variable's contents, and it can be redirected to another memory address containing another variable.

This unfortunately can be confusing, as a memory address of a variable is also indicated by the `&` operator, in addition to another operator `*` that is used to declare a pointer. A simple example illustrates this. First, declare and assign an integer variable:

```
int x = 42;
```

Next, declare a pointer to an integer, using the `*` operator:

```
int* xp;
```

This says to create a variable that will be a *pointer to an* `int` *type*, but don't point to anything specific yet; this comes in the next step:

```
xp = &x;
```

The `&` operator in this case means *the address of* `x`. `xp` now points at the memory address that contains the contents of `x`, namely 42. Note that this usage of `&` has a different meaning than declaring a reference.

We can now access the contents of this memory address by *dereferencing* `xp` by applying the `*` operator. If we put

```
std::cout << *xp << std::endl;
```

the output would be 42, just as if we had applied `std::cout` to the variable `x`. Note the `*` operator is used in a different context here, accessing the contents of memory rather than declaring a pointer.

We can also means we can change the value of `x`. For example, putting

```
*xp = 25;
```

then both `*xp` and `x` will return the value 25, rather than 42.

It is also possible to *reassign* the pointer `xp` to a different memory address; this is not possible to do with a reference. Suppose we have a different integer variable `y` and we reassign `xp` to point to the address of `y`:

```
int y = 106;
xp = &y;
```

Now, `*xp` will return 106 rather than 25, but `x` is still equal to 25.

NOTE

`xp`, as opposed to `*xp`, will return the hexadecimal value that represents the address of the first byte in memory containing the contents of `y`.

Similar to references, pointers can be used with objects. If we have a class `SomeClass` with a member function, say `some_fcn`, then we can define a pointer to a `SomeClass` object:

```
SomeClass sc;
auto SomeClass* ptr_sc = &sc;
```

As it's obvious that `ptr_sc` will point to a `SomeClass` object, we can use the `auto` keyword without obscuring its context.

Suppose also that `SomeClass` has a member function `some_fcn`. This function can be invoked by dereferencing `ptr_sc` and then calling it in the usual way:

```
(*ptr_sc).some_fcn();
```

More common, however, is to use the indirection operator, indicated by an arrow:

```
ptr_sc->some_fcn();
```

This is all we will need to know about pointers for now. More specifically, these examples take place in stack memory, and they are automatically deleted when the function or control block in which they are defined terminates. More advanced usage will be presented later.

NOTE

Pointers can also point to memory allocated in heap memory, which allows the value or object to persist in memory outside the scope of a function or control block. This becomes relevant in certain situations related to object-oriented programming and requires and extra care. Moreover, C++11 introduced smart pointers into the Standard Library. These topics will be presented in Chapter 5.

Function and Operator Overloading

A key feature of C++, as well as other modern programming languages, is implementing different versions of the same function name, distinguished by different sets of input arguments. This is known as *function overloading*. A related feature that is very convenient to us as quantitative programmers is *operator overloading*, where we can define an operation for specific types, such as vector multiplication. Operator overloading is not supported in as many languages as function overloading; for example, it exists in C++ and C#, but it is not an option in Java.

Function Overloading

To illustrate function overloading, let's look at an example of two versions of a `sum` function, one of which returns a `double` type, while the other returns a `vector<double>`. The first version is trivial, just summing two real numbers.

```
#include <vector>
// . . .
double sum(double x, double y)
{
        return x + y;
}
```

The second version, however, will take in two `std::vector<double>` objects and return a vector containing the sum of its elements.

```
std::vector<double> sum(const std::vector<double>& x, const
std::vector<double>& y)
{
        // NOTE TO SELF: Can we do this with range-based for
loops(?!)
        std::vector<double> vec_sum;
        if(x.size() == y.size())
        {
                for (int i = 0; i < x.size(); ++i)
                {
                        vec_sum.push_back(x.at(i) + y.at(i));
                }
        }
        return vec_sum;         // Empty if size of x and y
```

```
do not match
}
```

As we can see, the two functions perform two distinct tasks, and have different return types, based on the types of arguments.

Overloaded functions can also be distinguished based on the *number of arguments* of the same type, as well as return the same type. For example (trivially), we could define a `sum` function that takes in three real numbers:

```
double sum(double x, double y, double z)
{
        return x + y + z;
}
```

Now, if we put in our `main()` function the following,

```
sum(5.31, 92.26);
sum(4.19, 41.9, 419.0);
```

the respective overloaded functions will be called.

Operator Overloading

C++ provides the standard mathematical operators for integer and floating type numerical values. The Standard Library also provides the `+` operator for `std::string` types, which will concatenate them. However, there are no operators provided for `std::vector`, for example. So, if we want to compute an element-by-element sum of two vectors, or calculate a dot product, we’re on our own.

We could just use the `sum` overload for two vectors as shown above for vector addition, and write a new function called `dot_product` for vector multiplication. However, C++ provides us with a more naturally mathematical approach, namely *operator overloading*.

For a vector sum, the addition operator replaces the `sum` overload as shown below. The body of the function remains the same:

```
std::vector<double> operator + (const std::vector<double>& x,
const std::vector<double>& y)
```

```
{
        std::vector<double> add_vec;
        if (x.size() == y.size())
        {
                for (unsigned i = 0; i < x.size(); ++i)
                {
                        add_vec.push_back(x.at(i) + y.at(i));
                }
        }
        return add_vec;         // Empty vector if x & y
sizes not identical
}
```

Similarly, for the dot product, which returns a scalar (`double`), overload the `*` operator:

```
double operator * (const std::vector<double>& x, const
std::vector<double>& y)
{
        double dot_prod = 0.0;
        if (x.size() == y.size())
        {
                for (int i = 0; i < x.size(); ++i)
                {
                        dot_prod += (x[i] * y[i]);
                }
        }
        return dot_prod;        // Return 0.0 if size of x
and y do not match
}
```

Then, for two vectors `x` and `y`, say

```
std::vector<double> x = {1.1, 2.2, 3.3};
std::vector<double> y = {0.1, 0.2, 0.3};
```

the overloaded operators would perform vector addition and multiplication:

```
auto v_sum = x + y;             // ans: {1.2, 2.4, 3.6}
auto v_dot = x * y;             // ans: 1.54
```

For `double` types, the compiler knows to apply the language-provided operators

```
double s = 1.1 + 0.1;   // s = 1.2
double p = 2.2 * 0.2;   // p = 0.44
```

NOTE

1. For simultaneous iteration over two `vector` objects, at this stage we need to revert to an indexed `for` loop. There are more elegant ways to do this that avoid the index but require additional background that will be presented in Chapter 7.
2. For the error condition where `x.size() != y.size()`, for now we are simply returning an empty vector for the vector sum, and 0 for the dot product. Exceptions would be more appropriate for production code.

As for other examples, if we were to write a `Matrix` class, we would also want to overload operators `+`, `-`, and `*`. For a `Date` class, we could define `-` to return the number of days between two dates. Operator overloading is thus very convenient for mathematical and financial programming. We will utilize it in various contexts going forward.

Summary

This chapter has covered a fairly lengthy list of topics, starting with the `std::vector` container class in the Standard Template Library (STL). `std::vector` is ubiquitous in quantitative programming, for (good) reasons that will be covered in Chapter 7, along with STL iterators and algorithms that can make C++ code more elegant, reliable, and efficient. At this point, however, the goal is to be familiar with `std::vector` as a dynamic array of real numbers.

Aliases come in three different varieties: type aliases (`using`), references, and pointers. `using` saves us from having to type out long type names, such as `RealVector` in place of the oft-used `std::vector<double>`. References in C++ are mostly used in passing `const` reference objects as function arguments, avoiding object copying that can degrade performance, while

preventing the object from being modified inside the function. Pointers have several important applications beyond being mere aliases that will be presented in due course, along with smart pointers that were added to the Standard Library beginning with C++11.

Function overloading is a natural fit for mathematical programming, and operator overloading even more so for objects such as matrices and vectors that are ubiquitous in quantitative programming. This is another topic that will be extended in object-oriented programming in Chapter 4.

References

[1] CppReference: `https://en.cppreference.com/w/cpp/language/operator_precedence]`

[2] Stroustrup 4E (not directly referenced)

Chapter 3. Writing User-Defined Functions and Classes in Modules

A NOTE FOR EARLY RELEASE READERS

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

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

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

In our code examples so far, we have placed all our code into one file with a top-down design, beginning with the `main` function where execution of the program begins. This is of course not practical for realistic production programming. Prior to C++20, the usual approach to writing user-defined (non-templated) functions and classes would first involve placing their declarations placed in header files (usually `.h`, `.hpp`, or `.hxx` extensions). An implementation file (typically `.cpp`, `.cxx`, or `.cc` extensions) would then load the declarations using the `#include` preprocessor command. Each implementation file would then be compiled into a *translation unit* (de facto standard extensions are `.o` for Clang and gcc, and `.obj` for Visual Studio).

C++20 introduced *modules* which, without going into excessive detail here,

offer the advantages of reduced time required to compile and build a code base, elimination of problems that can arise with header files and preprocessor directives, and greater control over what is exported to other code files. One convenient result is it is no longer necessary to wrap declarations in include guard macros. In addition, the problem of header file leaking can also be avoided, as an imported module into another does not carry with it everything that it imports, unlike `#include` macros in one header file that get propagated into another. [[For more details, see Ranier Grimm, and Niall Cooling (https://blog.feabhas.com/2021/08/c20-modules-with-gcc11/)]]

Modules also allow placing both declarations and implementations in a single file, although it is also possible to separate them over multiple files with an interface and multiple implementation files. For this book, we will limit the discussion to the single file case [[at least for now]].

Single file modules can be used to implement functions and classes similar to C# and Java, without separate declarations. But keeping the separation even within one file still provides certain advantages, namely a clear separation of interface and implementation, and the potential to reduce or eliminate the recompilation of other code that consumes a module.

Default file extensions for single file modules at this stage seem to be converging around `.ixx` for Visual Studio, and `.cppm` on Clang, although using `.cpp` is also an option. [[Check Ranier Grimm's warning about using cppm; he recommends using .cpp with Clang, or .ixx with Clang-Cl in MSVC]]. In this book, we will use the `.ixx` extension for module files.

> **NOTE**
>
> At the time of this writing, there are still various points with respect to standardization that are still pending. This could change by the time this book reaches publication.

In this chapter, we will firt show introductory first examples of using

modules for both user-defined non-member functions and classes. This will a good segue into a deeper discussion on class design, including compiler-provided default special functions, user-defined constructors and destructors, and operator overloading. Move semantics, a language feature added in C++11, will also be introduced. For some readers this will be review, but the examples will be shown again within the context of C++20 modules.

Using Modules to Write User-Defined Functions

We will now build up the process of writing functions in modules, starting from a very simple example, and then introducing the details step-by-step.

A First Example with Non-Member Functions

Suppose we want to write a function that sums up the elements in a vector of integers:

```
int vector_sum_of_ints()
{
        vector<int> v = { 1,2,3 };
        int sum = 0;
        for (int elem : v) sum += elem;
        return sum;
}
```

This is placed in a module called `CppMod`, in a separate `.ixx` file, say `CppModule.ixx`. It will be easier to follow by just writing out the entire code in this file and then going through it section by section.

```
module; // The global fragment
#include <vector>
export module CppMod;   // The global fragment ends at this
point, and
                                          // the functionality
of the module starts here.
using std::vector;
// Declare the function first, using the
// export keyword to make it accessible outside the module.
export int vector_sum_of_ints();
```

```
// Implementation of the function.  The export keyword
// is not necessary here as it is already included
// in the declaration.
int vector_sum_of_ints()
{
        vector<int> v = { 1,2,3 };
        int sum = 0;
        for (int elem : v) sum += elem;
        return sum;
}
```

The `module` statement on the first line determines where the *global fragment* of the module begins. This section is [[exclusively]] reserved for pre-processor commands, particularly for `#include` statements of header files in the Standard Library and elsewhere that are needed for the implementation.

Next, the `export module` statement indicates the end of the global fragment and defines the module itself. This will make it available for *import* into other modules and source code. What follows first is the declaration of the ` vector_sum_of_ints ` function, preceded by another use of the `export` keyword. What this does is indicate to the compiler that this function can be called externally from code outside the module. Functions that are not marked as `export` will only be callable from inside the module.

After this, we can then write the function implementation. Note that we do not need to put `export` here, as placing it with the function declaration is sufficient.

To see how we can use this module, we will import it into our usual `Main.cpp` file (not a module itself at this stage). This is accomplished using the `import` keyword before the `main` function. Then, from `main`, we will call ` vector_sum_of_ints `:

```
#include <iostream>             // #include <iostream> as
usual
using std::cout;
using std::endl;
import CppMod;          // Import the CppMod module
                                // containing the
```

```
vector_sum_of_ints function.
int main()
{
        cout << vector_sum_of_ints() << endl;
}
```

Running this code, you can verify the result is 6.

Next, we can have a look at what happens if a non-export function, `add_stuff` is added to the module. Again, a very simple example, this will just double an integer value. If we call it from inside the exported function, then twice the sum of the vector results is returned:

```
module;
#include <vector>
export module CppMod;
using std::vector;
export int vector_sum_of_ints();
int add_stuff (int n);
int vector_sum_of_ints()
{
        vector<int> v = { 1,2,3 };
        int sum = 0;
        for (int elem : v) sum += elem;
        sum = add_one(sum);
        return sum;
}
int add_stuff (int n)
{
        return n + n;
}
```

Compiling and running the program again, the result is not surprisingly (drum roll please), 12. Attempting to call `add_stuff ` from the external `main` function, however, would result in a compiler error.

It is also possible to define a *variable* that is local to the module and not accessible outside. For example, we could declare and initialize a non-exported integer variable to 0, and then if it is reassigned, it will maintain its new value inside the module. For example, we can set it equal to `n` inside the `add_stuff ` function, and then add it again in the exported function where

it retains its reassigned value of 6:

```
module;
#include <vector>
export module CppMod;
using std::vector;
export int vector_sum_of_ints();
int add_one(int n);
int k = 0;
int vector_sum_of_ints()
{
        vector<int> v = { 1,2,3 };
        int sum = 0;
        for (int elem : v) sum += elem;
        sum = add_k(sum);
        return sum + k;         // k still = 6; returns 18
}
int add_stuff(int n)
{
        k = n;                  // k = 6
        return k + n;           // 12
}
```

The result now is 18, but that isn’t as important as the fact that `k` is local to the module and cannot be accessed externally. Similar to a non-exported function, an attempt to use `k` inside the `main` function will fail to compile. Conceptually, both non-exported functions and variables act like private members on a class, but with respect to the module and its non-member functions instead.

It is furthermore possible to make the separation of interface and implementation clearer by rearranging the function implementations in what is called the *private fragment* of a single-file module. This must be the final section in the module, and it is indicated with `module: private` separating the declarations above with the implementations below:

```
module;
#include <vector>
export module CppMod;
using std::vector;
// Interface section is here:
```

```
export int vector_sum_of_ints();
int add_stuff(int n);
int k = 0;
// Implementations are placed in the private fragment:
module:private;
int vector_sum_of_ints()
{
        vector<int> v = { 1,2,3 };
        int sum = 0;
        for (int elem : v) sum += elem;
        sum = add_stuff(sum);
        return sum + k;         // k still = 6; returns 18
}
int add_stuff(int n)
{
        k = n;                  // k = 6
        return k + n;           // 12
}
```

Using a private fragment also purportedly will/can prevent external code using the module from recompiling when a change is made inside the module. There is still some ambiguity on this point that will hopefully be clarified within the next few months.

We will soon see how a module with a private fragment can be useful in a more realistic example.

Standard Library Header Units

Proposals to the ISO C++ Committee for reorganizing the Standard Library into "a standard-module version" [Stroustrup P2412r0] were also drafted and submitted for inclusion in C++20, but this effort has been deferred until the next release planned for 2023. In the interim, as a placeholder, *Standard Library header units* that guarantee "[e]xisting `#include`s of standard library headers transparently turn into module imports in C++20" [ISO P1502R1] are now available. What this essentially means is preprocessor statements such as

```
#include <vector>
#include <algorithm>
```

can be replaced by importing their header equivalents:

```
import <vector>;                // Note these require a
semicolon
import <algorithm>;
```

This applies to all C++ Standard Library declaration files; however, due to complications arising in headers inherited from C – eg, <cassert> and <cmath>, based on the legacy C headers <assert.h> and <math.h> respectively – these are not covered.

In the above example then, we can eliminate the global fragment and import the `vector` header file under the `export module` statement; viz,

```
export module CppMod;
import <vector>;
using std::vector;
```

If other header files outside of the Standard Library need to be `#include`d, as preprocessor directives, *these must* go into the global fragment of the module.

Beyond the convenience of importing Standard Library header units, they also have been shown to decrease build times and binary file bloat, although this is not always guaranteed.

Examples can be found in Stroustrup [P2412r0].

Modules Prevent Leaking into Other Modules

A module will not “leak” imported modules within itself when imported into other models. That is, if a module `A` imports another module `B`,

```
// Define module A that imports module B:
export module A;
import B;
```

then if `A` is imported into another module `C`, or into a `Main.cpp` file, `B` will not be implicitly imported as well unless it also is explicitly imported as well:

```
// Define module C that imports module A:
export module C;
```

```
import A;
import B;                  // Not implicitly imported with
module A.
                           // Must be explicitly imported if
functions
                           // in B are also to be used inside
module C
```

This is in contrast with `#include`d header files that will leak. For example, a header file `MyHeader.h` includes a user-defined `YourHeader.h` and the STL <vector> header:

```
// MyHeader.h
#include “YourHeader.h”
#include <vector>
```

If `MyHeader.h` is `#include`d in `Main.cpp`, then it will also carry functions in “YourHeader.h” and the `std::vector` class will also be present in `Main.cpp`:

// Main.cpp

```
#include “MyHeader.h”
int main()
{
        // This will compile:
        auto y = my_header_fcn(…);
        // But so will these lines:
        auto z = your_header_fcn(…);
        std::vector <double> v;
}
```

With preprocessor `#include` statements, the content contained in `YourHeader.h` and `std::vector` are “leaked” into the `main` function. In realistic situations where many more header files might be involved, losing track of what is included and what is not can potentially lead to unexpected behavior or errors at runtime. In addition, this can also lead to longer build times. With modules, the programmer has greater control over what is imported, and build times can be substantially lower.

More details are available in Ranier Grimm’s very informative ModernesCpp

blog site.

A Black-Scholes Module Example

For something closer to a realistic financial example, the Black-Scholes model for pricing a European equity option can be written inside a module. Before coding up the model, though, we will need a reliable way to indicate the payoff type, call or put. For this, a module called `Enums` will contain an exported enum class `PayoffType`:

For the Black-Scholes functions, the mathematical formulation shown in James, Option Theory will be used.

$$\text{where } \phi = \begin{cases} 1, if\ a\ call\ option \\ -1, if\ a\ put\ option \end{cases}$$

A Module Containing Enum Definitions

First, so as to prevent bogus input values of ϕ, an enum class is used to represent the payoff type, and it resides in its own module and separate file, say `Enums.ixx` so that it can be reused elsewhere.

```
// File Enums.ixx
export module Enums;
export enum class PayoffType
{
        CALL,
        PUT
};
```

In practice, other enum classes representing bond types, futures contract identifiers, currency codes etc could be appended to this module and then imported into other pricing and risk modules as needed.

The Black-Scholes Formula Module

The model requires a natural log function, and a way to compute the cumulative distribution function for a standard normal distribution. We are in luck here, because the CDF can be written as

$$\frac{1 + erf\,\frac{x}{\sqrt{2}}}{2}$$

where erf is the error function, and it is available in `<cmath>`, as is the natural log function `log`. However, because this header does not have a header unit guaranteed by the C++20 Standard [[discussed above]], it needs to be `#include`d in the global fragment of the module:

```
module;
#include <cmath>        // cstuff headers (derived from
stuff.h)
                                  // should be #include(d) in
the global fragment
```

An `export` statement follows next and defines the name of the module, say `BlackScholesFcns`. For convenience, the required `<cmath>` function `using` aliases go next

```
export module BlackScholesFcns;
// <cmath> functions used below:
using std::log;
using std::erf;
```

The formula also uses $\sqrt{2}$, and this is now available as a constant alias in C++20. It also needs to determine the maximum between the option payoff and zero at expiration. For these, the `sqrt2` constant in `<numbers>`, and the `std::max` function in `<algorithms>` are available. Because `<numbers>` and `<algorithms>` are available as header units, they can be `import`ed rather than `#include`d:

```
// Standard Library header units, and using aliases
import <numbers>;
```

```
using std::numbers::sqrt2;
import <algorithm>;
using std::max;
```

`export import`

The `BlackScholesFcns` module will need to `import` the `Enums` module in order to indicate whether an option is a call or a put. But as described earlier, a benefit of using a module is it will not leak an imported module into another location. This means if `BlackScholesFcns` is imported into `Main.cpp`, the programmer will need to know to also `import` the `Enums` module. This is not a desirable approach, as it would require inspecting the source code of a module for others it imports before it could be consumed elsewhere.

Fortunately, the `export import` command is available. It will first `import` the `Enums` module into `BlackScholesFcns`, and then `export` it wherever the latter is consumed.

```
export import Enums;
```

This way, when `BlackScholesFcns` is imported into another module or file, it will not be necessary to hunt through the source code in `BlackScholesFcns` for the imported `Enums`, and then re-import it into the new target. In addition, and in general, the only imported modules that can be carried through are those marked with `export import`. This way, unexpected behavior due to an unintentionally leaked module can be avoided.

> **NOTE**
>
> The `<cmath>` header will leak into the target where `BlackScholesFcns` is imported due to the `#include` statement. This should be rectified with the standard-module version of the Standard Library currently slated for C++23.

The calculations are naturally divided up into an exported Black-Scholes function, and two private functions that compute the $d1$ and $d2$ values, and the Standard Normal CDF. The declarations are:

```
export double black_scholes_price(double strike, double spot,
double rate,
        double sigma, double year_frac, PayoffType pot);
// Internal functions and variables
void dee_fcns(double strike, double rate, double spot,
        double sigma, double year_frac);
double norm_cdf(double x);
```

The salient point here is module users only need to be concerned with the `black_scholes_price` function. The responsibility for calling other two is delegated to the exported function.

Because the *d*1 and *d*2 values are internal to the model, they are declared and initialized, but also not exported. This makes them accessible to the functions in the module but not to the outside world, much like a private member function on a class.

```
// Internal module variables
double d1 = 0.0, d2 = 0.0;              // d1 and d2 values
in Black-Scholes
```

The function implementations go last, with the exported `black_scholes_price` function first, followed by the two helper functions in the private fragment. Note that `dee_fcns` is a `void` function. Rather than have two separate functions to calculate the *d*1 and *d*2 values, this can be done in one function by setting the results on the `d1` and `d2` variables that are common to the module, but hidden from its users.

```
double black_scholes_price(double strike, double spot, double
rate,
        double sigma, double year_frac, PayoffType pot)
{
        double opt_price = 0.0;
        // phi, as in the James book:
        double phi = (pot == PayoffType::CALL) ? 1.0 : -1.0;
        if (year_frac > 0.0)
        {
                dee_fcns(strike, rate, spot, sigma,
year_frac);
                double n_dee_one = norm_cdf(phi * d1);
// N(d1)
```

```
                double n_dee_two = norm_cdf(phi * d2);  //
N(d2)
                double disc_fctr = exp(-rate * year_frac);
                opt_price = phi * (spot * n_dee_one -
disc_fctr * strike * n_dee_two);
        }
        else
        {
                opt_price = max(phi * (spot - strike), 0.0);
        }
        return opt_price;
}
module : private;
void dee_fcns(double strike, double rate, double spot,
        double sigma, double year_frac)
{
        double numer = log(spot / strike) + rate * year_frac
                + 0.5 * year_frac * sigma * sigma;
        double sigma_sqrt = sigma * sqrt(year_frac);
        d1 = numer / sigma_sqrt;
        d2 = d1 - sigma_sqrt;
}
double norm_cdf(double x)
{
        return (1.0 + erf(x / sqrt2)) / 2.0;
}
```

In the `Main.cpp` file, put

```
import BlackScholesFcns;
```

to import the model, and then call the exported function; eg, an in-the-money put with about three months until expiration. Note that the payoff enum type can be assigned because the `Enums` module was exported from `BlackScholesFcns` with the `export import` command.

```
        strike = 200.0;
        porc = PayoffType::PUT;
        spot = 185.0;
        rate = 0.05;
        sigma = 0.25;
        year_frac = 0.25;
        cout << "Put Option price = "
```

```
                << black_scholes_price(strike, spot, rate,
sigma, year_frac, porc )
                << endl;
```

This gives a price of 17.0649.

As can (hopefully) be seen, in addition to their other benefits, modules can be quite useful for implementing financial models where a lot of internal calculations are required but not necessary for the consumer to be concerned with. This way, the intermediate values can be encapsulated similar to a private variable on a class rather than passed around between functions and being exposed to unexpected modification, without the overhead of creating an object. For models more complex than Black-Scholes – and many of course exist – this means fewer moving parts accessible to the outside world, and hence fewer things to go wrong.

This is not to say it would be wrong to use a class to implement a financial model, but the decision to use a module of non-member functions rather than a class will come down to a design decision. There may be cases where it might be preferable to have a set of models conform to a contract set on an abstract base class, as we will discuss in the next chapter., so it really comes down to weighing the pros and cons of each while considering the requirements set in the design phase.

As for writing classes, the modern approach also involves using modules. This is the topic of the next section.

User-Defined Class Implementation in Modules

The traditional way to write a (non-templated) class in C++ has also been to write the declarations in a header file, and the implementations in a separate file. For a class, this could also be beneficial in terms of code maintenance, as the declarations alone – provided the member function and variable naming was informative – essentially presented an outline up front about what the class did without the “clutter” from all the function implementations.

Moving to modules, again the declarations and implementations can now be

placed in a single file, with the same best practice of keeping the class declaration separate from the implementation. This way, compile times can be reduced, plus the outline of the class once found in a header file is preserved inside a module.

With the exception of a new module name to hold a class called `BlackScholes`,

```
export module BlackScholesClass;
```

the same preliminary `import`, `export`, and `export import` statements, as in the non-member function version, remain the same. Things then change with writing the class declaration rather than those for individual functions. Note the entire class declaration is placed inside the scope of the `export class` statement.

```
export class BlackScholes
{
public:
        double black_scholes_price(double strike, double
spot, double rate,
                double sigma, double year_frac, PayoffType
pot);
private:
        void dee_fcns_(double strike, double rate, double
spot,
                double sigma, double year_frac);
        double norm_cdf_(double x);
        double d1_ = 0.0, d2_ = 0.0;
};
```

The member functions are the same as before, but scoped with the class name:

```
double BlackScholes::black_scholes_price(double strike,
double spot, double rate,
        double sigma, double year_frac, PayoffType pot)
{
        double opt_price = 0.0;
        // phi, as in the James book:
        double phi = (pot == PayoffType::CALL) ? 1.0 : -1.0;
        if (year_frac > 0.0) . . .
```

```
        return opt_price;
}
etc. . .
```

As the class declaration is marked `export`, the implementations are implicitly exported, so there is no need to `export` the public `black_scholes_price` function.

Using Namespaces with Modules

Namespaces have often been employed to prevent name clashes between functions with the same name and signature, but from different source files or libraries. They can also prevent similar compiler errors from two different modules.

Suppose there are two modules, `ThisModule` and `ThatModule`, each containing a `maximum` function that returns the maximum value of two real numbers.

```
export module ThisModule;
export double maximum(double x, double y)
{
        double max_val = x > y ? x : y;
        return max_val;
}
export module ThatModule;
export double maximum(double x, double y)
{
        double max_val = 2*x > 2*y ? x : y;
        return max_val;
}
```

If both modules are imported into another translation unit (eg Main.cpp) and a call is made to the `maximum` function, the compiler will not be able to determine which version of the function is intended, resulting in a compiler error.

```
// In some other location:
import ThisModule;
import ThatModule;
```

```
//. . .
double compare_max(double x, double y)
{
        return maximum(x, y);   // Compiler error!
}
```

By wrapping the `maximum` function in a distinct namespace and indicating the namespace scope when called outside the module, the compiler error is avoided.

```
export module ThisModule;
export namespace this_nsp
{
        export double maximum(double x, double y)
        {
                double max_val = x > y ? x : y;
                return max_val;
        }
}
export module ThatModule;
export namespace that_nsp
{
        export double maximum(double x, double y)
        {
                double max_val = 2*x > 2*y ? x : y;
                return max_val;
        }
}
```

Now, scope the function call with one of the namespaces, just as we do for functions in classes in the `std` Standard Library namespace, and the code will compile:

```
import ThisModule;
import ThatModule;
//. . .
double compare_max(double x, double y)
{
        return this_nsp::maximum(x, y); // Will now compile
}
```

An alternative would be to use a `using` alias for one of the namespaces:

```
import ThisModule;
import ThatModule;
using that_nsp::maximum
// …
double compare_max(double x, double y)
{
        return this_nsp::maximum(x, y); // Will also compile
}
```

Note that if in `Main.cpp`, the entirety of each of these namespaces is loaded into the global namespace with `using namespace` statements, and there is no scoping of the `maximum` function, the same problem will ensue, as shown here:

```
import ThisModule;
import ThatModule;
using namespace this_nsp;
using namespace that_nsp;
// . . .
double compare_max(double x, double y)
{
        return maximum(x, y);   // Compiler error!
}
```

As a corollary, you can now see why global use of

`using namespace::std;`

is also not regarded as good practice. With so many classes and functions in the Standard Library, it is entirely possible that someone might use a common name as a user-defined class or function elsewhere, causing a name clash that leads to a compiler error.

NOTE

Many of the newer features in C++, beginning with C++11, have their roots in the Boost Libraries. For example, smart pointers `unique_ptr` and `shared_ptr` use the same names in Boost as well as in the Standard Library. As the Boost libraries are in wide use, opening up both the entire `std` and `boost` namespaces globally could create headaches for code maintenance and build teams

> Both smart pointers in particular, and the Boost libraries more generally, will be covered in later chapters of this book.

Our discussion of namespaces here is related more specifically to modules, but for more information about best practices using namespaces in C++, Sutter and Alexandrescu, Coding Style, chapter X remains an excellent resource.

As a general takeaway, classes and functions that belong to namespaces should either be scoped explicitly; eg,

```
import <vector>;
import ThisModule;
//. . .
std::vector<double> v;
return this_nsp::maximum(x, y);
```

or contained in using aliases; eg,

```
import <vector>;
using std::vector;
import ThisModule;
using this_nsp::maximum;
```

rather than exposing entire namespaces globally.

Summary

TBD…

Some references:

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1502r1.html

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1453r0.html

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0581r1.pdf

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2412r0.pdf

https://devblogs.microsoft.com/cppblog/a-tour-of-cpp-modules-in-visual-

studio/

https://github.com/microsoft/STL/issues/60

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0955r0.pdf

Chapter 4. Dates and Fixed Income Securities

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 9th 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*.

Dates and date calculations might not seem like the most compelling topic to discuss, but they are vitally important in quantitative finance, particularly in fixed income trading and analytics.

As with distributional random number generation in the past, financial C++ programmers were left with similar options: either write their own date classes and functions or use a commercial or open source external library. This has changed with C++20. It includes a date class that is determined by the integer year, month, and day values. This class relies both on the already existing (since C++11) `std::chrono` foundation of durations, timepoints, and the system clock – ie, chronological computations – as well as calendrical computations, which are based on the number of days relative to the epoch and take into account the non-uniform number of days in each month.

While the goal of this chapter is to demonstrate how to use the new date features in financial applications, the author of `std::chrono`, Howard Hinnant, provides more details on his GitHub site containing the original development code for `std::chrono` dates{1}. (This will be referred to as "the GitHub date code site" going forward).

Adding years and months can be effected by using calendrical options in `std::chrono`, but adding days requires conversion to a chronological timepoint. These are important operations to be discussed in due course, but first let us cover how dates are represented and instantiated in C++20. From there, we will look at common date calculations that are required in finance, a class that encapsulates the functions for us, day count conventions and yield curves, and finally an application in valuing a coupon-paying bond.

Representation of a Date

C++11 introduced the `std::chrono` library into the Standard Library, which provided the following abstractions:

- Duration of time: A method of measurement over a given time interval, such as in units of minutes, days, milliseconds, etc
- Timepoint: A duration of time relative to an epoch, such as the UNIX epoch 1970-1-1
- Clock: The object that specifies the epoch and normalizes duration measurements {0}

Dates in `std::chrono` are based on these chronological foundations, but as part of the new C++20 features, conversions to calendrical forms are also now available. These can be used for calculations involving years and months.

A standard date in `std::chrono` is represented by an object of the class `std::chrono::year_month_day`. There are a variety of constructors for this class, among which several are discussed here.

First, a constructor taking in the year, month, and day is provided. But instead of integer values for each, these constructor arguments must be defined as separate `std::chrono::year`, `std::chrono::month`, and `std::chrono::day` objects. For example, to create an object holding the date 14 November 2002, we would create it as follows.

```
import <chrono>;

std::chrono::year_month_day ymd{ std::chrono::year{2002},
        std::chrono::month{11}, std::chrono::day{14} };
```

Alternatively, individual constant `month` objects are defined in `std::chrono` by name, so an equivalent approach to constructing the same month above is to replace the constructed month object in the previous example with the pre-defined `November` instance:

```
std::chrono::year_month_day ymd_alt{
std::chrono::year{2002},
        std::chrono::November, std::chrono::day(14) };
```

For assignment, the `/` operator has also been overloaded to define a `year_month_day` object:

```
ymd = std::chrono::year{ 2002 } / std::chrono::month{11} /
std::chrono::day{14};
```

Different orders can be used, along with integer types, as long as the first argument is obvious. For yyyy/mm/dd format, putting

```
ymd = std::chrono::year{ 2002 } / 11 / 14;
```

would yield the same result, with the compiler interpreting the 11 and 14 as `unsigned` types. mm/dd/yyyy format can also be used: `auto mdy = std::chrono::November / 14 / 2002;`

In this case, the 14 is recognized as `unsigned`, and the year as an `int`. In

`std::chrono`, `month` and `day` types can be cast to `unsigned`, while a `year` can only be cast to an `int`. The examples above are non-exhaustive, and a more comprehensive list can be found on the GitHub date code site{1}.

Note that the output stream operator is overloaded for `year_month_day`, so any of the above can be output to the console with `cout`. For example,

```
cout << ymd << endl;
```

will display the date on the screen as `2002-11-14`

1.1 Serial Representation and Date Differences

A `year_month_day` date can also be measured in terms of the number of days since an epoch, with the `system_clock` default being the UNIX epoch January 1, 1970. Similar to Excel – whose epoch is January 1, 1900 – this representation can be convenient for certain types of date arithmetic in finance, particularly in determining the number of days between two dates. Unlike Excel, however, the UNIX epoch is represented by 0 rather than 1, in the sense that serial dates are measured in *days since the epoch*. Consider the following example, with dates 1970-1-1 and 1970-1-2.

```
std::chrono::year_month_day epoch{ date::year{1970},
date::month{1}, date::day{1} };
std::chrono::year_month_day epoch_plus_1{ date::year{1970},
date::month{1}, date::day{2} };
```

Then, the respective serial dates can be accessed as follows:

```
int first_days_test =
        std::chrono::sys_days(epoch).time_since_epoch().cou
nt();           // 0

first_days_test =
        std::chrono::sys_days(epoch_plus_1).time_since_epoc
h().count();    // 1
```

These return `int` values 0 and 1, respectively. Also unlike Excel, `std::chrono` dates before the epoch are also valid but carry a negative integer value. In the statement that follows, the returned value is -1.

```
first_days_test =
        std::chrono::sys_days(epoch_minus_1).time_since_epo
ch().count();   // -1
```

For typical financial trading, it is usually not necessary to go back before 1970, but in some fields, such as actuarial valuations of pension liabilities, many retirees were born before this date. Historical simulations of markets also might use data going back many decades.

Recalling that the `year_month_day` class is built upon the three `std::chrono` abstractions listed at the outset, technically what is happening here is the `sys_days` operator returns the `ymd` date as a `std::chrono::time_point` object, where `sys_days` is an alias for `time_point`. Then, its `time_since_epoch` member function returns a `std::chrono::duration` type. The corresponding integer value is then accessed with the `count` function.

An important calculation to have in finance is the number of days between two dates. Using `ymd` again as 2002-11-14, and initializing `ymd_later` to six months later – 2003-5-14 – take the difference between the two `sys_days` objects obtained with `sys_days` and apply the `count` function to the difference:

```
// ymd = 2002-11-14
// ymd_later = 2003-5-14

auto diff = (std::chrono::sys_days(ymd_later) –
        std::chrono::sys_days(ymd)).count();            //
181
```

The result of 181 is again returned as an `int`.

Next, when working with dates, there are several checks that are often

necessary to perform, namely whether a date is valid, whether it is in a leap year, finding the number of days in a month, and whether a date is a weekend. Some of the machinery for this is immediate with functions conveniently contained in `std::chrono`, but in other cases, there will be a little more work involved.

1.2 Accessor Functions for Year, Month, and Day

Accessor functions on `year_month_day` are provided for obtaining the year, month and day, but they are returned as their respective `year`, `month`, and `day` objects.

```
year()          // returns std::chrono::year
month()         // returns std::chrono::month
day()           // returns std::chrono::day
```

Suppose we have two `year_month_day` objects `date1` and `date2` Then, conveniently, applying the `count` function on the difference (a `duration`), the results are `int` types.

```
(date2.year() - date1.year()).count()           // returns
int
(date2.month() - date1.month()).count()         // returns
int
(date2.day() - date1.day()).count()                    //
returns int
```

Each of the individually accessed year, month, and day components can also be cast to integers (in the mathematical sense), but an important point to be aware of is a `year` can be cast to an `int`, but for a `month` or `day`, these need to be cast to `unsigned`.

```
auto the_year = static_cast<int>(date1.year());
auto the_month = static_cast<unsigned>(date1.month());
auto the_day = static_cast<unsigned>(date1.day());
```

NOTE

Going forward, for convenience we will use the namespace alias `namespace date = std::chrono;`

1.3 Validity of a Date

It is possible to set `year_month_day` objects to invalid dates. For example, as will be seen shortly, adding a month to a date of January 31 will result in February 31. In addition, the constructor will also allow month and day values out of range. Instead of throwing an exception, it is left up to the programmer to check if a date is valid. Fortunately, this is easily accomplished with the boolean `ok` member function. In the following example, the `ymd` date (same as above) is valid, while the two that follow are obviously not.

```
// date is now an alias for std::chrono

date::year_month_day ymd{ date::year{2002},
        date::month{11}, date::day{14} };

bool torf = ymd.ok();                    // true

date::year_month_day negative_year{ date::year{-1000},
        date::October, date::day{10} };

torf = negative_year.ok();               // true – negative
year is valid

date::year_month_day ymd_invalid{ date::year{2018},
        date::month{2}, date::day{31} };

torf = ymd_invalid.ok();                          // false

date::year_month_day ymd_completely_bogus{
date::year{-2004},
        date::month{19}, date::day{58} };
```

```
torf = ymd_completely_bogus.ok();       // false
```

The `ok` member function will come in handy in subsequent examples, particularly in cases where a date operation results in the correct year and month, but an incorrect day setting in end-of-the month cases. This will be addressed shortly. The upshot is it is up to the consumer of the `year_month_day` class to check for validity, as it does not throw an exception or adjust automatically.

1.4 Leap Years and Last Day of the Month

You can easily check whether a date is a leap year or not. A boolean member function, not surprisingly called `is_leap`, takes care of this for us:

```
date::year_month_day ymd_leap{ date::year{2016},
        date::month{10}, date::day{26} };

torf = ymd_leap.year().is_leap()                // true
```

There is no member function available on `year_month_day` that will return the last day of the month. A workaround exists using a separate class in std::chrono that represents an end-of-month date, `year_month_day_last`, from which the last day of its month can also be accessed as before, and then cast to unsigned.

```
date::year_month_day_last
        eom_apr{ date::year{ 2009 } / date:April /
date::last };

auto last_day = static_cast<unsigned>(eom_apr.day());   //
result = 30
```

This can also be used as a device to check whether a date falls on the end of a month:

```
date::year_month_day ymd_eom{ date::year{2009},
```

```
        date::month{4}, date::day{30} };

bool torf = ymd_eom == eom_apr;         // Returns true
(torf = "true or false")
```

The last day of the month for an arbitrary date can also be determined:

```
date::year_month_day ymd = date::year{ 2024 } / 2 / 21;

year_month_day_last
        eom{ date::year{ ymd.year() } / date::month{
ymd.month() } / date::last };

last_day = static_cast<unsigned>(eom.day());    // result =
29
```

It should also be noted a `year_month_day_last` type is implicitly convertible to a `year_month_day` via reassignment:

```
ymd = eom_apr;                    // ymd is now 2009-04-30
```

More background can be found in {2}.

Although this works, it carries the overhead of creating a `year_month_day_last` each time it is called, and additional object copy if reassigned, as shown in the last line of the code example just above. While your mileage may vary, it is possible this could have a negative performance impact in financial systems managing heavy trading volume and large portfolios containing fixed income securities.

A set of "`chrono`-Compatible Low-Level Date Algorithms" is provided elsewhere on the the GitHub date code site{3}. These alternatives apply methods that are independent of `year_month_day` class methods, and their description in the documentation states these low level algorithms are "key algorithms that enable one to write their own date class". This is the direction in which we are eventually headed.

To determine the last day of the month, a more efficient user-defined function

can be derived from code provided in this set of algorithms, as follows:

```
// User-defined last_day_of_the_month
unsigned last_day_of_the_month(const
std::chrono::year_month_day& ymd)
{
    constexpr std::array<unsigned, 12>
        normal_end_dates{ 31, 28, 31, 30, 31, 30, 31, 31,
30, 31, 30, 31 };

    if (!(ymd.month() == date::February &&
ymd.year().is_leap()))
    {
        unsigned m = static_cast<unsigned>(ymd.month());
        return normal_end_dates[m - 1];
    }
    else
    {
        return 29;
    }
}
```

This is more of a brute force approach in hard-coding the days in each month of a non-leap year, but it does obviate additional object creation and copying.

One other point is the use here of `constexpr`, another language feature added to C++11. Because the length of the `array` and its contents are known a priori, `constexpr` instructs the compiler to initialize `end_dates` at compile time, thus eliminating re-initialization every time the `last_day_of_the_month` function is called. Two related points follow:

1. Using `constexpr` in this specific case may or may not have a significant impact on performance, but it is possible in cases where the function is called many times in computationally intensive code, such as in portfolio risk simulations and calculations.

2. Most financial data, such as market and trade data, will be inevitably dynamic and thus unknown at compile time, so `constexpr` may be of limited use in financial applications. The example above, however,

demonstrates an example of how and when it can potentially be used effectively.

1.5 Weekdays and Weekends

Similar to end of the month dates, there is no member function to check whether a date falls on a weekend. There is again a workaround from which we can derive the result we need.

`std::chrono` contains a `weekday` class that represents the day of the week – Monday through Sunday – not just weekdays per se (the terminology here might be slightly confusing). It can be constructed by again applying the `sys_days` operator in the constructor argument.

```
// Define a year_month_day date that falls on a business
day (Wednesday)

date::year_month_day ymd_biz_day{ date::year{2022},
        date::month{10}, date::day{26} };       //
Wednesday

// Its day of the week can be constructed as a weekday
object:
date::weekday dw{ date::sys_days(ymd_biz_day) };
```

The day of the week can be identified by an `unsigned` integer value returned from the `iso_encoding` member function, where values 1 through 7 represent Monday through Sunday, respectively. The stream operator is overloaded so that the abbreviated day of the week is displayed.

```
unsigned iso_code = dw.iso_encoding();
cout << ymd_biz_day << ", " << dw << ", " << iso_code <<
endl;
```

The output is then `2022-10-26, Wed, 3` This allows us to define our own function, in this case a lambda, to determine whether a date falls on a weekend or not.

```
auto is_weekend = [](const date::year_month_day& ymd)->bool
{
        date::weekday dw{ date::sys_days(ymd) };
        return dw.iso_encoding() >= 6;
};
```

Now, also construct a `year_month_day` date that falls on a Saturday:

```
date::year_month_day ymd_weekend{ date::year{2022},
        date::month{10}, date::day{29} };       // Saturday
```

Then, we can use the lambda to test whether each day is a business day or not.

```
torf = is_weekend(ymd_biz_day);         // false (Wed)
torf = is_weekend(ymd_weekend);         // true (Sat)
```

Supplemental information on weekends in `std::chrono` can be found in {4}.

1.6 Adding Years, Months, and Days

One more set of important date operations of finance is adding years, months, and days to existing dates. These are particularly useful for generating schedules of fixed payments. Adding years or months is very similar – relying on the `+=` operator – but adding days involves a different approach.

1.6.1 Adding Years

Adding years is very straightforward. For example, add two years to 2002-11-14, and then add another 18 years to the result. Note that the number of years being added needs to be expressed as a `std::chrono::years` object, an alias for a `duration` representing one year.

```
// Start with 2002-11-14
date::year_month_day ymd{ date::year{2002},
```

```
date::month{11}, date::day{14} };

ymd += date::years{ 2 };                // ymd is now 2004-
11-14
ymd += date::years{ 18 };               // ymd is now 2022-
11-14
```

We run into a problem, however, if the date is the last day of February in a leap year. Adding two years to 2016-02-29 results in an invalid year.

```
date::year_month_day
        ymd_feb_end{ date::year{2016}, date::month{2},
date::day{29} };

ymd_feb_end += date::years{ 2 };        // Invalid result:
2018-02-29
```

Dates in `std::chrono` will again neither throw an exception or adjust the day, so it is up to the developer to handle the case where years are added to a February 29 date in a leap year.

1.6.2 Adding Months and End-of-the-Month Cases

Adding months to a `year_month_day` object is similar to adding years, but this now requires handling multiple end-of-month edge cases due to different numbers of days in different months, plus again the case of a February in a leap year.

When no end-of-month date is involved, the operation is straightforward, similar to adding years, using the addition assignment operator. Similar to adding years, the number of months needs to be represented as a `duration` object, in this case an alias for a period of one month.

```
date::year_month_day ymd{ date::year{2002},
date::month{11}, date::day{14} };
ymd += date::months(1);                 // Result: 2002-12-
14
ymd += date::months(18);                // Result: 2004-06-
```

```
14
```

Subtraction assignment is also available:

```
ymd -= date::months(2);          // Result: 2004-04-14
```

With end-of-the-month cases as well, the `+=` operation can again result in invalid dates. To see this, construct the following end-of-month dates:

```
date::year_month_day ymd_eom_1{ date::year{2015},
date::month{1},
        date::day{31} };
date::year_month_day ymd_eom_2{ date::year{2014},
date::month{8},
        date::day{31} };
date::year_month_day ymd_eom_3{ date::year{2016},
date::month{2},
        date::day{29} };
```

Naively attempting month addition results in invalid dates:

```
ymd_eom_1 += date::months{ 1 };          // 2015-02-31 is
not a valid date
ymd_eom_2 += date::months{ 1 };          // 2014-09-31 is
not a valid date
ymd_eom_3 += date::months{ 12 };         // 2017-02-29 is
not a valid date
```

Although the results are not valid, the year and month of each is correct. That is, for example, adding one month to 2015-01-31 should map to 2015-02-28.

Going the other way, if we were to start on 2015-02-28 and add one month, the result will be correct: 2015-03-28.

Recalling the `last_day_of_the_month` function defined previously, a workaround is fairly straightforward. Addition assignment is naively applied, but if the result is invalid, it must be due to the day value exceeding the actual number of days in a month. In this case, because the resulting year and month will be valid, it just becomes a case of resetting the day with the number of

days in the month.

```
auto add_months = [](date::year_month_day& ymd, unsigned
mths) -> void
{
    ymd += date::months(mths);    // Naively attempt the
addition

    if (!ymd.ok())
    {
        ymd = ymd.year() / ymd.month() / date::day{
last_day_of_the_month(ymd) };
    }
}
```

1.6.3 Adding Days

Unlike for years and months, there is no `+=` operator defined for adding days. For this reason, we will need to obtain the `sys_days` equivalent before adding the number of days.

```
date::year_month_day ymd{date::year(2022), date::month(10),
date::day(7)};

// Obtain the sys_days equivalent of ymd, and then add
three days:
auto add_days = date::sys_days(ymd3) + date::days(3);  //
ymd still = 2022/10/07
```

Note that at this point, `ymd` has not been modified, and the result, `add_days`, is also a `sys_days` type. To set a `year_month_day` object to the equivalent, the assignment operator provides implicit conversion. Similar to previous applications of `sys_days`, we can just update the original `ymd` date to three days later:

```
ymd = add_days; // Implicit conversion to year_month_day
                                  // ymd is now = 2022-10-10
```

More information on adding days can be found in {5}.

A Date Class Wrapper

As you can probably see by now, managing all the intricacies of `std::chrono` dates can eventually become complicated. For this reason, we will now outline the typical requirements for financial date calculations and declare them in a class based on a `year_month_day` member. This way, the adjustments and `year_month_day` function calls are implemented once behind interfacing member functions and operators that are arguably more intuitive for the consumer. These can be divided into two broad categories, namely checking possible states of a date, and performing arithmetic operations on dates. A summary of what we have covered so far is provided in the list of requirements below. Most of these results will be integrated into the class implementation.

State

- Days in month
- Leap year

Arithmetic Operations

- Number of days between two dates
- Addition
    - Years
    - Days
    - Months

Additional functionality that we will want to have is listed next. These additional requirements will be also be part of the implementation.

Accessors

- Year, Month, Day
- Serial date integer representation (days since epoch)
- `year_month_day` data member

Comparison operators

```
==
<=>
```

To begin, the class declaration will give us an implementation roadmap to follow.

Class Declaration

We will incorporate the requirements listed above into a class called `ChronoDate`. It will wrap a `std::chrono::year_month_day` object along with some of its associated member functions that are useful in financial calculations. The only other data member will be the serial date representation of the date object.

Before working through the member functions, let us start with the constructors.

Constructors

For convenience, a constructor is provided that takes in integer values for year, month, and day, rather than requiring the user to create individual `year`, `month`, and `day` objects. ChronoDate{ int year, unsigned month, unsigned day }; Note that the argument for the year is an `int`, while those for the month and day are `unsigned`. This is due to the design of the `year_month_day` class, as previously discussed.

As we will see for convenience later, a second constructor will take in a `year_month_day` object:

```
ChronoDate{ date::year_month_day };
```

And finally, a default constructor will construct a `ChronoDate` set to the UNIX epoch.

Public Member Functions and Operators

These should mostly be self-explanatory from the member function declarations below. Furthermore, it will mainly be a case of integrating the previously developed functionality into the respective member functions. As for the comparison operators `==` and $<$ $\Rightarrow$, as well as the friend stream operator, these are already defined on the `year_month_day` class, so it is simply a matter of wrapping them into the same operators on `ChronoDate`.

There is one remaining public function in the declaration not yet covered, `weekend_roll`, which will be used to roll a date to the nearest business day in the event a date falls on a Saturday or Sunday. Its implementation will be covered shortly.

```
// Check state:
int days_in_month() const;
bool leap_year() const;

// Arithmetic operations:
unsigned operator - (const ChronoDate& rhs) const;
ChronoDate& add_years(int rhs_years);
ChronoDate& add_months(int rhs_months);
ChronoDate& add_days(int rhs_days);

// Accessors
int year() const;
unsigned month() const;
unsigned day() const;
int serial_date() const;
date::year_month_day ymd() const;

// Modfying function
ChronoDate& weekend_roll();              // Roll to
business day if weekend

// Operators
```

```
bool operator == (const ChronoDate& rhs) const;
std::strong_ordering operator <=> (const ChronoDate& rhs)
const;

// friend operator so that we can output date details with
cout
friend std::ostream& operator << (std::ostream& os, const
ChronoDate& rhs);
```

Private Members and Helper Function

Two private member variables will store the underlying `year_month_day` object and serial representation of the date. One private function will wrap the function calls required to obtain the number of days since the UNIX epoch, so that the serial date can be set at construction, as well as updated anytime the state of an object of the class is modified.

```
private:
    date::year_month_day date_;
    int serial_date_;
    void reset_serial_date_();
```

Class Implementation

As we have almost all of the necessary functionality, what remains is mostly a case of wrapping it into the member functions, plus implementing the `weekend_roll` function and a couple of private helper functions. In addition, two constructors are presented, as well as the private `reset_serial_date_` method that will calculate and set the serial representation of the date, either at construction, or whenever the state of an active `ChronoDate` is modified.

Constructors

The implementation of the first declared constructor allows one to create an instance of `ChronoDate` with integer values (`int` and `unsigned`) rather than require individual instances of `year`, `month`, and `day` objects.

```
ChronoDate::ChronoDate(int year, unsigned month, unsigned
day) :
        date_{ date::year{year} / date::month{month} /
date::day{day} }
{
        if(!date_.ok())         // std::chrono member
function to check if valid date
        {
                throw std::exception e{ "ChronoDate
constructor: Invalid date." };
        }
        reset_serial_date_();
}
```

Recall also that because it is possible to construct invalid `year_month_day` objects, such as February 30, a validation check is also included in the constructor, utilizing the `ok` member function on `year_month_day`. One more setting that needs to occur when a date is constructed is the serial date. This is delegated to the private method `reset_serial_date_`. As shown at the outset of the chapter, this is an application of `sys_days` operator to provide the number of days since the UNIX epoch.

```
void ChronoDate::reset_serial_date_()
{
        serial_date_ =
date::sys_days(date_).time_since_epoch().count();
}
```

This function will also be called from each modifying member function.

Finally, the default constructor just sets the date to the UNIX epoch, and initializes the serial date to 0:

```
ChronoDate::ChronoDate():date_{date::year(1970),
date::month{1}, date::day{1} } :
    serial_date_{0} {}
```

Member Functions and Operators

The following describes implementation of the functions previously introduced in the declaration section.

Accessors

Implementation of accessors for the serial date and `year_month_day` members is trivial, but a little more work is involved in returning integer values for the year, month, and day. a `std::chrono::year` object can be cast to an `int`, while `month` and `day` are castable to `unsigned` types. With this in mind, their accessors are straightforward to implement:

```
int ChronoDate::year() const
{
        return static_cast<int>(date_.year());
}

unsigned ChronoDate::month() const
{
        return static_cast<unsigned>(date_.month());
}

unsigned ChronoDate::day() const
{
        return static_cast<unsigned>(date_.day());
}
```

State Methods

Checking whether a date is in a leap year simply involves wrapping the respective `year_month_day` member function.

```
bool ChronoDate::leap_year() const
{
        return date_.year().is_leap();
}
```

Obtaining the number of days in the month is more involved, but it is just a rehash of the function adapted from the `std::chrono` low-level algorithms in Section [1.4].

```
unsigned ChronoDate::days_in_month() const
{
        unsigned m = static_cast<unsigned>(date_.month());
        std::array<unsigned, 12>
        normal_end_dates{ 31, 28, 31, 30, 31, 30, 31, 31,
30, 31, 30, 31 };

        return (m != 2 || !date_.year().is_leap() ?
normal_end_dates[m - 1] : 29);
}
```

Arithmetic Operations

These are the core member functions that will be used for typical fixed income applications, such as in calculating year fractions and generating payment schedules. To start, let us revisit calculation of the number of days between two dates. As we already store the serial date on the class and update it only at construction or when a date is modified, we can remove the `sys_days` conversions and function calls, and implement the subtraction operator as the difference between the integer equivalents.

```
unsigned ChronoDate::operator - (const ChronoDate& rhs)
const
{
        return this->serial_date_ - rhs.serial_date_;
}
```

Adding years and months are also pretty straightforward, as we now have ways to handle pesky end-of-month issues when they arise. The only issue when adding years is if the resulting date lands on the 29th of February in a non-leap year, so this case is easily addressed by resetting the day value to 28. Note that because the result is based on the underlying `year_month_day` `+=` operator, the state of the object is modified, and thus it becomes necessary to update the serial date as well.

```
ChronoDate& ChronoDate::add_years(int rhs_years)
{
        // Proceed naively:
```

```
        date_ += date::years(rhs_years);

        if (!date_.ok())
        {
                date_ = date_.year() / date_.month() / 28;
        }

        reset_serial_date_();

        return *this;
}
```

When adding months to a date, the situation becomes more problematic with varying days in each month plus a leap year condition in February, but with the `days_in_month` member function now available, it becomes a reasonably easy exercise. The addition of months is again naively attempted, with the number of days adjusted if the resulting month is invalid. The only way this incorrect state can occur is if the naïve result has more days than in its respective month.

```
ChronoDate& ChronoDate::add_months(int rhs_months)
{
        date_ += date::months(rhs_months);    // Naively
attempt the addition

        // If the date is invalid, it is because the
        // result is an invalid end-of-month:
        if (!date_.ok())
        {
                date_ = date_.year() / date_.month() /
date::day{ days_in_month() };
        }

        reset_serial_date_();

        return *this;
}
```

As seen earlier, there is no addition assignment operator for adding days, so in `std::chrono` this will require conversion to `sys_days`.

```
ChronoDate& ChronoDate::add_days(int rhs_days)
{
        date_ = date::sys_days(date_) +
date::days(rhs_days);

        return *this;
}
```

Note that the sum of the `sys_days` and the `days` to be added are implicitly converted back to a `year_month_day` object when assigned to the `date_` member. Further details behind this are also available in {5}.

Business Day Roll Rule

One important function we have not discussed yet is that which will roll a weekend date to the next business date.

In practice, there are various commonly used roll methods. For the purposes of this discussion, we will choose one that is used quite often in practice, namely the Modified Following rule. Before proceeding, let us revisit determining the day of the week using the `weekday` class contained in `std::chrono`.

As mentioned earlier, the term “weekday” may be a little confusing. It does not mean “weekday” as in Monday through Friday, but rather “day of the week”. The `iso_encoding` member function will return an integer code for each day of the week, beginning with 1 for Monday and 7 for Sunday; therefore, a value of 6 or 7 will indicate the date falls on a weekend.

The `weekend_roll` function will just reuse this functionality to first determine if the date falls on a weekend. If it does, it will first naively roll forward to the next Monday. However, if this new date advances to the next month, it will roll back to the previous business date, namely the Friday of the original month. This is why the original month is stored first.

```
ChronoDate& ChronoDate::weekend_roll() {
    date::weekday wd{ sys_days(date_) };
    month orig_mth{ date_.month() };
```

```
    unsigned wdn{ wd.iso_encoding() }; // Mon =  1, ...,
Sat = 6, Sun = 7
    if (wdn > 5) date_ = sys_days(date_) + days(8 - wdn);

    // If advance to next month, roll back; also handle
roll to January
    if (orig_mth < date_.month()
        || (orig_mth == December && date_.month() ==
January))
            date_ = sys_days(date_) - days(3);

    reset_serial_date_();
    return *this;
}
```

A rolled date will be modified, so it is necessary to update the serial date here as well.

Comparison and Streaming Operators

The comparison operators `==` and $<$ $\Rightarrow$ are immediate as these are defined for `year_month_day`. We just need to be sure to use `std::strong_ordering` as the return type for $<$ $\Rightarrow$, as it is ultimately two integer values – the days since the epoch – that are being compared.

```
bool ChronoDate::operator == (const ChronoDate& rhs) const
{
        return date_ == rhs.date_;
}

std::strong_ordering ChronoDate::operator <=> (const
ChronoDate& rhs) const
{
        return date_ <=> rhs.date_;
}
```

We can also piggyback off of the stream operator for `year_month_day` and define it as a `friend` operator on `ChronoDate`.

```
// This is a 'friend' of the ChronoDate class
export std::ostream& operator << (std::ostream& os, const
ChronoDate& rhs)
{
        os << rhs.ymd();
        return os;
}
```

> **NOTE**
>
> Because this operator is a `friend` of the `ChronoDate` class, a separate `export` of the implementation is needed even when included in the same module.

With the `ChronoDate` class now ready to go, we can move on to day count conventions and other components that are typically required for programming related to fixed income trading.

Day Count Bases

Day count bases are used to convert the interval between two dates into time measured in units of years, or *year fractions*, as commonly referred to in fixed income trading. Day count bases are used whenever an interest calculation is made. Interest rates are defined by three attributes: an annual percentage value, e.g. 3%, a type, e.g. simple or compound, and a day count basis. Consider a term deposit where 1000 dollars is invested at 3% compound interest, the investment being made on 25 October 2022 and maturing on 31 December 2023. The formula for calculating F, the value of the investment at maturity, is

$$F = 1000(1 + 0.03)^t$$

The value of *t* depends on the day count basis. Money market calculations in the US and the EU are most likely to use the Actual/360 day count basis:

$$Act360\left(d_1, d_2\right) = \frac{d_2 - d_1}{360}$$

In UK, Canadian, and Australian money markets, the Actual/365 day count basis — where the 360 swapped for 365 — is more common. Other common day count bases used in broader fixed income trading include the 30/360 method, which assumes every month has 30 days, and a year has 360 days. The Actual/Actual method uses the actual number of days in both the numerator and denominator. In equity portfolio management, An Actual/252 basis is often used, where 252 business days per year are assumed.

Implementing day count conventions in C++ is an example of where interface inheritance can be useful. We can define a pure abstract base class that mandates the implementation of the day count-adjusted year fraction, and then leave it to the derived classes to implement the specific calculations.

The interface simply declares a pure virtual `operator()` for the calculations on the derived classes.

```
export class DayCount
{
public:
        virtual double operator()
        (const ChronoDate& date1, const ChronoDate& date2)
const = 0;

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

The Actual/365 year fraction calculation is trivial:

```
export class Act365 : public DayCount
{
public:
        double operator() (const ChronoDate& date1, const
ChronoDate& date2) const
        override
    {
        return (date2 - date1) / 365.0;
```

```
    }
};
```

An `Act360` class would be the same, except with the denominator replaced by 360.

The 30/360 case is a bit more complicated, in that the numerator must first be calculated according to the formula {put here}.

End-of-month adjustments for the day values will depend on the particular form of the 30/360, of which there are several that can depend upon the geographical location of a trading desk. In the United States, the ISDA version (International Swaps and Derivatives Association){6} is commonly used and is implemented in the example below as the private `date_diff_` helper function. The result is then divided by 360 in the public operator override.

```
export class Thirty360 : public DayCount
{
public:
        double operator()
        (const ChronoDate& date1, const ChronoDate& date2)
const override
    {
        return static_cast<double>( date_diff_(date1,
date2)) / 360.0;
    }

private:
        unsigned date_diff_(const ChronoDate& date1, const
ChronoDate& date2) const
        {
            unsigned d1, d2;
        d1 = date1.day();
        d2 = date2.day();

        auto f = [](unsigned& d) {
            if (d == 31)
                {
                        d = 30;
```

```
                }
        }

        f(d1);
        f(d2);

        return 360 * (date2.year() - date1.year()) + 30 *
(date2.month() –
            date1.month()) + d2 - d1;
    }
};
```

Then, for some examples:

```
        Act365 act_365{};
        Act360 act_360{};
        Thirty360 thirty_360{};

        ChronoDate sd1{ 2021, 4, 26 };
        ChronoDate ed1{ 2023, 10, 26 };
        ChronoDate sd2{ 2022, 10, 10 };
        ChronoDate ed2{ 2023, 4, 10 };

        auto yf_act_365_01 = act_365(sd1, ed1); // 2.50137
        auto yf_act_365_02 = act_365(sd2, ed2); // 0.49863

        auto yf_act_360_01 = act_360(sd1, ed1); // 2.53611
        auto yf_act_360_02 = act_360(sd2, ed2); // 0.505556

        auto yf_thirty_01 = thirty_360(sd1, ed1);       //
2.5
        auto yf_thirty_02 = thirty_360(sd2, ed2);       //
0.5
```

The results are shown in the comments. Note that only the 30/360 day count basis yields year fractions to half of a year exactly.

As a quick application of day count bases {5.5} (Steiner), consider obtaining the price of a short-term government Treasury Bill. In the US, these have maturities from four months to a year, and pricing is based on an Actual/365 basis. In the UK, maturities may be up to six months and carry an Actual/360

basis. We can write a valuation function that will accommodate an arbitrary day count basis via runtime polymorphism, so both US and UK cases can priced using the same function.

```
double treasury_bill(const ChronoDate& sett_date,
        const ChronoDate& maturity_date, double mkt_yield,
double face_value,
        const DayCount& dc)
{
        // pp 40-41, Steiner
        return face_value / (1.0 + mkt_yield *
dc(sett_date, maturity_date));
}
```

Yield Curves

A yield curve is derived from market data — a set of fixed points over discrete dates — as of its settle date (the day the prices are observed, eg the current trading day). The valuation process of fixed income positions depends on a normalized yield curve from discount factors can be obtained, so as to calculate present values of each future payment. A review of this process is detailed in the next section. These results will then be implemented within in the design of a yield curve class that follows afterward.

Deriving a Yield Curve from Market Data

Essentially a yield is an interest rate, looked at from a different perspective. If money is invested in a deposit account at a known rate of interest, then the accumulated value of the investment at some future date can be calculated. However, suppose we can invest 1000 dollars on 25 October 2022 and receive 1035.60 dollars on 31 December 2023. In order to compare this investment with other investments we calculate its yield. Assuming compounded interest and Actual/365 day count basis, then

$$1000(1+y)^{432/365} = 1035.60$$

from which we find the yield

$$y = \exp(\ln(1035.60/1000) \times 365/432) - 1 = 3\%$$

In general, the yield curve is a function of time, say $y(t)$, and is constructed from market data, such as Treasury Bills, swaps and bonds. The time values are in units of years (or year fractions).

These products all have known future cash flows, and are known as fixed income securities. In addition, each type of fixed income security has its own yield type (simple, discount or compounded), and its own day count basis, and these may vary within a single product group. To avoid the use of multiple interest types and day count bases, yield curves typically define their yields as continuously compounded with an Actual/365 day count basis.

To illustrate how the inputs to the yield curve are derived, consider a US Treasury Bill; the yield type is discount, the day count basis is Act/360, and the market quote is the yield on the bill. Suppose the face value is *F*, the maturity date is *m*, the market yield is y_m for settlement on *s*. Then, the price, *P*, of the bill is

$$P = F(1 - \text{Act } 360\,(s, m)\ y_m)$$

For the yield curve, the corresponding yield is $y(t)$, where $t = \text{Act } 365(s, m)$, so that

$$Pe^{ty(t)} = F$$

The value of $y(t)$ can be found from these two equations. It is

$$y\,(t) = \frac{-\ln(1 - \text{Act}\,360\,(s, m)\,y_m)}{t}$$

It is essential that any set of interest rate products used to create a yield curve have the same settle date. Let the maturity dates for the products be d_1,< d_2 < … < d_n, with $s < d_1$, and the associated yields be y_1, y_2, …, y_n, where $y_i = y\,(t_i)$ and t_i = Act365(s,d_i). Since the yields are calculated for a date

interval whose first date is the settlement date, these yields are known as spot yields.

There are many continuous curve which pass through the points

$$(t_1, y_1), (t_2, y_2), \cdots, (t_n, y_n)$$

The choice of an appropriate curve is a business decision made by the user of the yield curve. This typically based on a curve-fitting technique employing a particular interpolation method.

Discount Factors

Consider a unit payment amount made at time *m*. What is its value on a settle date, *s* < *m*? Let *P*(*s*,*m*) be the price, paid on the settle date, let *t*=Act365(*s*,*m*) and let *y*(*t*) be the associated yield. Then

$$P(s, m)\, e^{ty(t)} = 1$$

from which $P(s,m) = e^{-ty(t)}$. Now *P*(*s*,*m*) is the present value, as seen on date *s* for unit payment made on date *m*. In other words, it is the discount factor for the period *s* to *m*. Since *y*(*t*) is a spot yield, this is a spot discount factor.

Forward Discount Factors

How do we calculate the discount factor for a period which begins at time d_1 and ends at time d_2, where *s* < d_1 \leq d_2?

Consider a unit payment to be made at time d_2 and let its value at d_1 be represented by *P*(*s*; d_1,d_2). Its spot value – hence the parameter *s* – is

$$P(s, d_1)\, P(s; d_1, d_2)$$

To avoid arbitrage opportunities we must have

$$P(s, d_2) = P(s, d_1)\, P(s; d_1, d_2)\,, sothat$$

$$P\left(s;d_1,d_2\right)=\frac{P(s,d_2)}{P(s,d_1)}$$

Substituting for the spot discount factors:

$$P\left(s;d_1,d_2\right)=\frac{e^{-t_2y(t_2)}}{e^{-t_1y(t_1)}}=e^{t_1y(t_1)-t_2y(t_2)}$$

Since $d_1 > s$, $P(s;\_d\__1,d_2)$ is a forward discount factor.

The next sections will describe a framework for yield curves in C++, followed by a simple example of a yield curve and its use to value a bond

A Yield Curve Class

The essential function on a yield curve class will return a continuously compounded forward discount factor between two arbitrary dates, as detailed in the previous section. Fitting the resulting curve through the points $\left(t_1,y_1\right),\left(t_2,y_2\right),\cdots,\left(t_n,y_n\right)$ can be based on a plethora of numerical methods in the literature. As examples, these can range from a simple linearly interpolated yield curve, to more sophisticated examples such as a cubic spline-interpolated curve, Smoothest Yield Curves {7}, or the Monotone Convex Method {8}.

At a high level then, we could define an abstract base class that

1. Provides a common method to convert market yield data into continuously compounded yields, based on the results above {[4.1]}
2. Requires a derived class to implement its own curve-fitting method as a private member function

{UML Diagram Here}

The base class will contain a non-virtual public function that calculates the forward discount factor between two dates, using interpolated yields from the overridden `yield_curve_` method on each derived class. The interpolated yields determined from each derived class are assumed to be continuously

compounded with the Act/365 day count basis. The overridden `yield_curve_` method will depend on market data, related to a specific settle date.

```
export class YieldCurve
{
public:
        // d1 <= d2 < infinity
        double discount_factor(const ChronoDate& d1, const
ChronoDate& d2) const;
        virtual ~YieldCurve() = default;

protected:
        ChronoDate settle_;

private:
        Act365 act365_{};

        virtual double yield_curve_(double t) const = 0;
};
```

The implementation of `discount_factor` then follows the mathematical derivation presented in [[4.1.2]]. Note that if the first date `d1` is the settle date, the result defaults to the spot discount factor at date `d2`.

```
double YieldCurve::discount_factor(const ChronoDate& d1,
const ChronoDate& d2) const
{
        if (d2 < d1)
                throw
std::exception("YieldCurve::discount_factor: d2 < d1");

        if (d1 < settle_ || d2 < settle_)
                throw
std::exception("YieldCurve::discount_factor: date <
settle");

        // P(t1, t2) = exp( -(t2-t1) * f(t1, t2) )

        // if d1 == settle_ then P(t1,t2) = P(0,t2) = exp(-
```

```
t2 * y2 )
        double t2 = act365_(settle_, d2);
        double y2 = yield_curve_(t2);
        if (d1 == settle_) return exp(-t2 * y2);

        double t1 = act365_(settle_, d1);
        double y1 = yield_curve_(t1);
        // (t2-t1) f(t1,t2) = t2 * y2 - t1 * y1
        return exp(t1 * y1 - t2 * y2);
}
```

A Linearly Interpolated Yield Curve Class Implementation

The simplest curve fitting method – but still sometimes used in practice – is linear interpolation. More sophisticated interpolation methods also exist, such as those referenced above, but these require considerably more mathematical horsepower. So to keep the discussion concise, we will limit the example here to the linearly interpolated case, but it is important to remember that more advanced methods can also be integrated into the same inheritance structure.

```
export class LinearInterpYieldCurve final : public
YieldCurve
{
public:
        LinearInterpYieldCurve(
                const ChronoDate& settle_date,
                const vector<double>& maturities,       //
In Act/365 years.
                const vector<double>& spot_yields);     //
Continuously compounded,

// Act/365 day count basis

private:
        vector<double> maturities_; // maturities in years
        vector<double> yields_;

        double yield_curve_(const double t) const override;
```

```
};
```

The constructor will take in the maturities associated with each of the yield data points relative to the settle date, based on the Actual/365 day count basis. The corresponding spot yield values follow in the vector in the third argument. Its implementation will check whether the maturity and yield vectors are of the same length, and whether the settle date value is negative. If either is true, an exception is thrown. For the purposes of demonstration, we will just assume the maturities are in ascending order, but in production this would be another invariant to check.

```
LinearInterpYieldCurve::LinearInterpYieldCurve(
        const ChronoDate& settle_date,
        const vector<double>& maturities,
        const vector<double>& spot_yields) : maturities_{
maturities },
                yields_{ spot_yields }  // Maybe move
semantics instead?
{
        settle_ = settle_date;

        if (maturities.size() != spot_yields.size())
    throw std::exception("LinearInterpYieldCurve:
maturities and spot_yields are different lengths");

        if (maturities.front() < 0.0 )
          throw std::exception("LinearInterpYieldCurve:
first maturity cannot be negative");

        // Assume maturities are in order
}
```

The linear interpolation method is implemented in the mandated `yield_curve_` private member function. If the year fraction at which a yield is to be interpolated exceeds the maximum time value in the data, the result is just the last yield value. Otherwise, the `while` loop locates the interval of time points that surrounds the input value of time `t`. Then, the proportionally weighted yield is calculated and returned.

```
double LinearInterpYieldCurve::yield_curve_(const double t)
const
{
        // interp_yield called from discount_factor, so
maturities_front() <= t

        if (t >= maturities_.back())
        {
                auto check{ maturities_.back() };
                return yields_.back();
        }

        // Now know maturities_front() <= t <
maturities_.back()
        size_t indx{ 0 };
        while (maturities_[indx + 1] < t) ++indx;
        return yields_[indx] + (yields_[indx + 1] -
yields_[indx])
                / (maturities_[indx + 1] -
maturities_[indx]) * (t - maturities_[indx]);
}
```

A Bond Class

We are now in position to utilize objects of the preceding classes, along with a user-defined `Bond` class, to calculate the value of a coupon-paying bond. Common examples of bonds in the include government-issued Treasury Bonds, agency bonds (issued by a government-sponsored enterprise such as the Government National Mortgage Association (GNMA) in the US), corporate bonds, and local state and municipal bonds. As debt obligations, a series of regular payments over time is made in exchange for an amount borrowed by the issuer. The main difference with a traditional loan is the principal amount, ie the face value, is returned when the bond matures, rather than being amortized over time.

Bond Payments and Valuation

Before proceeding with further code development, it is probably worthwhile

to summarize the mechanics of how bond payments are structured, and how a bond is commonly valued. The details behind this are very important in writing real world bond trading software, yet it is surprising they are so often glossed over in computational finance courses and textbooks. The following discussion will then essentially become the design requirements for a Bond class, to subsequently follow.

The general idea is that a bond pays fixed amounts on dates in a regular schedule; for example, suppose a bond has a face value of $1000 and pays 5% of its face value every six months. Then the payment frequency is twice per year, and the coupon amount would be

$$\frac{0.05(1000)}{2} = 25$$

In general, the formula is

$$regular\ coupon\ amount = (coupon\ rate)\ \frac{(face\ value)}{(coupon\ frequency)}$$

The first task is to create a list of dates when payments are due and the coupon payment amounts for each of those dates. Along with the face value and annual coupon rate, the contractual conditions of the bond also include the following four dates:

- Issue date
- First coupon date
- Penultimate coupon date
- Maturity date

The *issue date* is the date on which the bond is first put on sale. A stream of fixed payments is typically paid, eg every six months, beginning with the *first coupon date*. The second to last payment occurs on the *penultimate coupon date*, and the final payment, consisting of the last coupon payment plus repayment of the face value, occurs on the *maturity date*. We will again

assume the coupon day is less than 29, to avoid problems with end of month cases.

Determining the Payment Schedule

Returning to the example above, of a bond which pays 5% coupon on a face value of $1000 with a coupon frequency of 2, when will the payments be made?

For coupons between the first coupon date and the penultimate date, payment *due dates* fall on a *regular schedule*. This means a constant payment of $25 is due every six months. To ensure these dates belong to a regular schedule of due dates, in general there are restrictions on the first and penultimate coupon dates, and the coupon frequency. These two dates must be business days having the same day of the month, the coupon frequency must be a divisor of twelve, and the two dates must differ by a multiple of 12/(coupon frequency) months.

Since the due dates may not fall on business dates, the bond also has associated *payment dates* that are adjusted for weekends and holidays. To simplify matters we will assume there are no holidays except Saturday and Sunday, as well as assume payments will not occur on any day with value greater than 28, so that if a due date falls on a weekend, the regular coupon payment will be made on the following Monday.

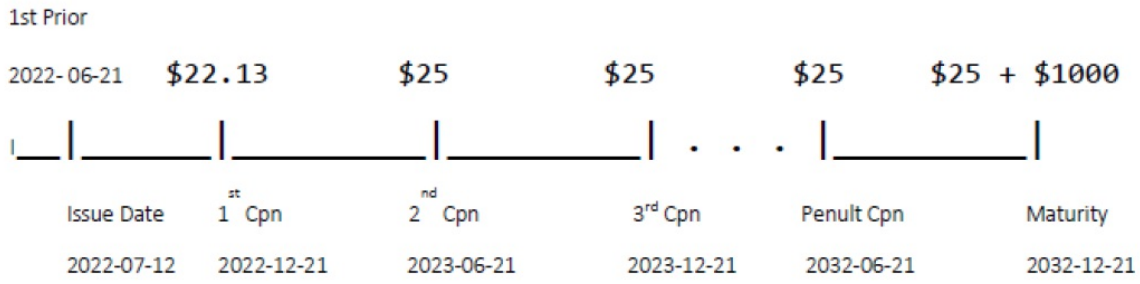
For example, suppose the bond has a first coupon date of 17 Mar 2023 and a penultimate coupon date of 17 Mar 2025. The two dates differ by 36 months, which is a multiple of 12/2=6, as required. Then, the intermediate due dates are 17 Sep 2023, 17 Mar 2024, …, 17 Mar 2025. These due dates will not all fall on business days, so each will need to be checked and rolled forward if necessary to qualify as payment dates.

If the first and final payments also occur regular periods, they will be \$25 and \$1025, respectively. However, first and last coupon periods may or may not be regular. The usually cited case is an irregular first payment period, but there can also be cases over of an irregular period from penultimate to maturity date.

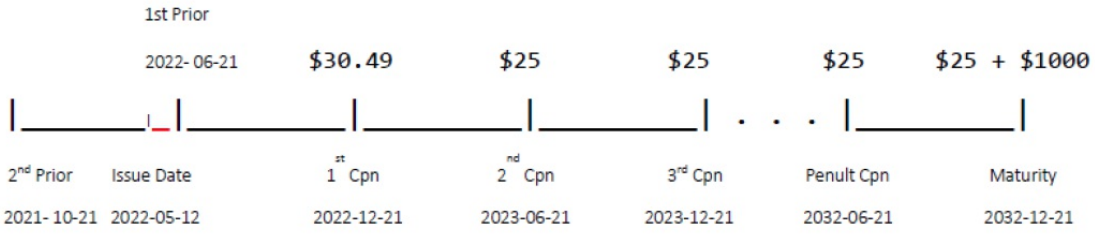
In the case of a short first period, the coupon payment is calculated by multiplying the annual coupon rate by the ratio of actual days in the period over the number of days in what would be a normal first period. Again, let us use the example of a \$1000 face value bond paying an annual coupon of 5% semiannually (\$25 regular coupon payments). Suppose a ten-year bond is issued on 2022-7-12, with a first payment date of 2022-12-21. Subsequent payment dates are then on 21 June and 21 December. If the first payment period had been regular, the issue date would have been 2022-6-21. This date is called the *first prior date*. Now, the first payment is prorated as follows:

$$\frac{(coupon\ payment)(number\ of\ days\ from\ issue\ to\ 1st\ pmt)}{(number\ of\ days\ from\ 1st\ prior\ to\ 1st\ pmt} =$$

$$25\left(\frac{162}{183}\right) = \$22.13$$



In the case of a long first period, suppose we have a bond with the same terms as in the short period example, but where the issue date is now 2022-5-12.



In this case, the first coupon payment will be the regular coupon of \$25 over the period from the first prior date to issue, *plus* a partial payment (in red) over the interval between issue and first prior date. This extra payment is

prorated over the six-month period from the *second prior date* to the first. That is,

$$\frac{(coupon\ payment)(number\ of\ days\ from\ issue\ to\ 1st\ prior)}{(number\ of\ days\ from\ 2ndprior\ to\ 1st\ prior)} =$$

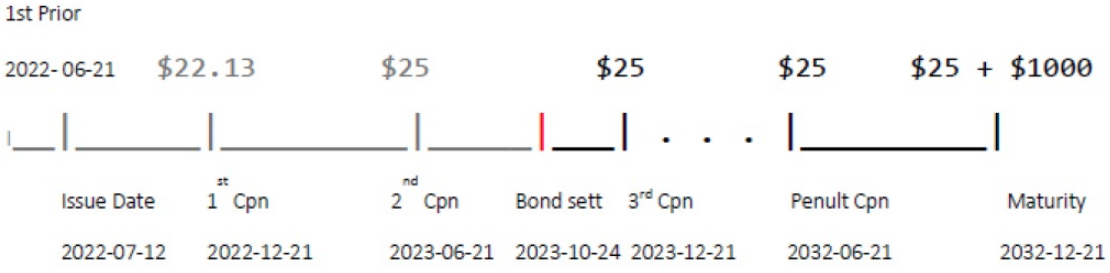
$$25\left(\frac{40}{182}\right) = \$5.49$$

The total first coupon payment is then 25 + 5.49 = 30.49

Calculating prorated payments over irregular final periods are similar, except that prior payment periods preceding issue, extra payment periods extending later than maturity are utilized.

Valuing a Bond

The issuer sells bonds on the issue date, pays the owners of the bonds the coupon amounts and, at maturity, also pays back the face value of the bond. The owner of a bond may then sell it on the secondary market, in which case the buyer and seller agree to a business date, known as the bond settle date, on which the sale will take place. On this settle date the seller receives his money, and the buyer becomes the registered owner, with the right to receive all payments which fall due after the settle date. Should the owner of the bond wish to value the bond at any time, this can be done by calculating the discounted value of the bond, using a yield curve of his own choice, and the discount factors calculated for the periods from the settlement date to the corresponding payment dates.



Revisiting the example in Figure 9-1, suppose a bond is exchanged for cash

on 2023-10-24, as indicated by the red hash mark in Figure 9-3 above. All preceding coupon payments have been paid to the previous owner, so the value of the bond will only depend on payments beginning with the third coupon payment through maturity. If the bond and yield curve settle dates are the same, then using the discount factor notation above, the value of the bond on this date will be

$$25\left(P\left(s,d_3\right)+\cdots+P\left(s,d_p\right)\right)+\left(1000+25\right)P\left(s,d_m\right)$$

where s = 2023-10-24 is the yield curve settlement date, d_p = 2032-6-21 is the penultimate coupon date, and d_m = 2032-12-21 is the maturity date.

It is also possible for a bond settle date to occur after the yield curve settle date. Suppose the bond transaction is to occur two business days after the yield curve settle date. If we let s_b = 2023-10-26, then the bond value becomes

$$25\left(P\left(s;s_b,d_3\right)+\cdots+P\left(s;s_b,d_p\right)\right)+\left(1000+25\right)P\left(s;s_b,d_m\right)$$

Discounted valuations are also routine calculations in trading and risk management software. Modern professional traders use this value as a benchmark for determining the fair (or equilibrium) market price of a bond. Risk managers will calculate bonds under multiple shocked or random yield curve scenarios to derive measures of their portfolio exposures to market risk.

A Bond Class

Our task now is to implement the requirements above in a user-defined `Bond` class, using the contractual terms of a bond as input data. First, let us here consolidate and review the essential data inputs associated with a bond issue.

- Face value
- Annual coupon rate
- Number of coupon payments per year (coupon frequency)

- Issue date
- First coupon date
- Penultimate coupon date
- Maturity date
- Day count basis

Our `Bond` class can be formally summarized as shown in the class declaration below.

```
export class Bond
{
public:
        Bond(string bond_id, const ChronoDate& issue_date,
const ChronoDate& first_coupon_date,
                const ChronoDate& penultimate_couppn_date,
const ChronoDate& maturity_date,
                int coupon_frequency, double coupon_rate,
double face_value);

        double discounted_value(const ChronoDate&
bond_settle_date,
                const YieldCurve& yield_curve);

        string bond_id() const;

private:
        string bond_id_;

        vector<ChronoDate> due_dates_;          // Dates on
which payments are due,

// whether business days or not.
        vector<ChronoDate> payment_dates_;      // Business
dates on which payments are made.
        vector<double> payment_amounts_;
};
```

Note that all the contractual information is captured in the constructor, and

valuation of the bond will be delegated to the public `discounted_value` function. This separates what is essentially the “interface” – namely the input and processing of contractual bond data – from the “implementation” where the bond value is calculated. Per the discussion in {[5.1]}, this valuation function is based on the bond settlement date and the market yield curve, inputs that are independent from the constructor arguments. One specific advantage to this is a single `Bond` instance can be created, and its valuation function called many times under different market scenarios for risk reporting purposes, as noted previously.

There are three vectors of equal length, `due_dates_`, `payment_dates_`, and `payment_amounts_`, corresponding to the respective descriptions in Section {[5.1.1]} {Determining the Payment Schedule} above. All three are necessary for calculating the discounted value of a bond.

A bond ID field is also generally required for both trading and risk applications, so it is added as a constructor argument and data member, along with a public accessor. The `coupon_frequency` parameter represents the number of coupon payments per year – ie 2 for semiannual, and 4 for quarterly – as defined in the bond contract.

Bond Class Implementation

Next, we will work through the class implementation step by step. The constructor will generate the due and payment dates, and the payment amounts. Note that `first_coupon_date`, `penultimate_coupon_date` and `maturity_date` are due dates which fall on business days. The `first_coupon_date` and `penultimate_coupon_date` input objects are also dates that are part of the regular schedule of due dates. The maturity date may or may not be part of the regular schedule of due dates, as discussed previously.

```
Bond::Bond(string bond_id, const ChronoDate& issue_date,
const ChronoDate& first_coupon_date,
        const ChronoDate& penultimate_coupon_date, const
ChronoDate& maturity_date,
```

```
        int coupon_frequency, double coupon_rate, double
face_value) : bond_id_(bond_id)
{

        // (1) Number of months in coupon period:
        const int months_in_regular_coupon_period = 12 /
coupon_frequency;

        // (2) Regular coupon payment:
        const double regular_coupon_payment = coupon_rate *
face_value / coupon_frequency;

        // (3) Generate vectors containing due dates,
payment dates,
        // and regular coupon payment amounts:
        for (ChronoDate regular_due_date{ first_coupon_date
};
                regular_due_date <=
penultimate_coupon_date;

regular_due_date.add_months(months_in_regular_coupon_period
))
        {
                // The due and payment Dates
                due_dates_.push_back(regular_due_date);
                ChronoDate payment_date{ regular_due_date
};

                // (4) Roll any due dates falling on a
weekend:

payment_dates_.push_back(payment_date.weekend_roll());
                // Assume all coupons are regular; deal
with short first period later.

payment_amounts_.push_back(regular_coupon_payment);
        }

        // (5) If first coupon is irregular, amend the
coupon payment:
        // Calculate the first_prior, the last regular date
before first_coupon_date.
        ChronoDate first_prior{ first_coupon_date };
```

```
        first_prior.add_months(-
months_in_regular_coupon_period);
        if (first_prior != issue_date) // if true then
irregular coupon
        {
                if (first_prior < issue_date) // if true
then short coupon period
                {
                        double coupon_fraction =
                                static_cast<double>
(first_coupon_date - issue_date) /
                                static_cast<double>
(first_coupon_date - first_prior );
                        payment_amounts_[0] *=
coupon_fraction;
                }
                else // issue_date < first_prior, so long
coupon period
                {
                        // long_first_coupon =
regular_coupon + extra_interest
                        // Calculate the second_prior, the
last regular date before the first_prior
                        ChronoDate second_prior{
first_prior };
                        second_prior.add_months(-
months_in_regular_coupon_period);
                        double coupon_fraction =
                                static_cast<double>
(first_prior - issue_date) /
                                        static_cast<double>
(first_prior - second_prior);
                        payment_amounts_[0] +=
coupon_fraction * regular_coupon_payment;
                }
        }


  // (6) The maturity date is a due date which falls on a
business day:
        due_dates_.push_back( maturity_date );
        payment_dates_.push_back( maturity_date );
        // Assume maturity date is a regular due date:
```

```
        double final_coupon{ regular_coupon_payment };

  // (7) If final coupon period is irregular amend the
coupon payment
  // Calculate maturity_regular_date, the first regular
date after penultimate_coupon_date
        ChronoDate maturity_regular_date{
penultimate_coupon_date };
        maturity_regular_date.add_months(months_in_regular_
coupon_period);
        if (maturity_regular_date != maturity_date) // if
true then irregular coupon period
        {
                if (maturity_date < maturity_regular_date)
// if true then short coupon period
                {
                        double coupon_fraction =
                                static_cast<double>
(maturity_date - penultimate_coupon_date) /
                                static_cast<double>
(maturity_regular_date - penultimate_coupon_date);
                        final_coupon *= coupon_fraction;
                }
                else  // maturity_regular_date <
maturity_date, do long coupon period
                {
                        // final_coupon =
regular_coupon_amount + extra_interest
                        // Calculate the next_regular_date,
the first regular date
                        // after the maturity_regular_date
                        ChronoDate next_regular_date{
maturity_regular_date };

next_regular_date.add_months(months_in_regular_coupon_perio
d);
                        double extra_coupon_fraction =
                                static_cast<double>
(maturity_date - maturity_regular_date) /
                        static_cast<double>
(next_regular_date - maturity_regular_date);
                        final_coupon +=
extra_coupon_fraction * regular_coupon_payment;
```

```
            }
        }

        // (8) Calculate final payment:
        payment_amounts_.push_back(face_value +
final_coupon);

}
```

First (1), although the `coupon_frequency` value is defined in the bond contract, and often stored in a bond database, it is easier to use the length of the regular coupon period – eg 3 months, 6 months, etc – in the tasks that follow. This equivalent number of months is calculated as shown above and stored as the constant integer value `months_in_regular_coupon_period`. Next (2), following the formula presented above {[5.1]}, `regular_coupon_payment` stores this value as a constant. Recall that regular coupon periods are those which span two adjacent due dates, and all coupon periods except the first and last are guaranteed to be regular.

Generating the Date and Payment Vectors

Now (3), the constructor implementation will generate the due and payment dates, and the payment amounts. The `due_dates_` vector will contain the regular dates that get generated for each respective coupon period – eg every six months – up to the contractual penultimate coupon date. These are not adjusted for weekends. Because the `+=` operator for months on `std::chrono::year_month_day` guarantees the same day value, with the correct year and month result, we are OK as long as the successive date is valid. Bonds have many variations, but since this is not production code it is simplified by assuming the coupon day is less than 29. This avoids end of month calculations.

At point (4), the `weekend_roll` member function is applied to successive copies of each due date and pushed onto the `payment_dates_` vector prior to the penultimate payment date. Thus, any due date falling on a weekend is rolled to the next business date. The regular coupon payment amount is also

appended to the `payment_amounts_` vector corresponding to each regular date.

In the previous step, the first coupon payment was naively set to the regular amount, so in (5), a check is made whether the first payment period is irregular. If so, a nested conditional statement determines whether this period is short or long. In the event of a short period, the first prior date is found by subtracting the number of months in a regular period from the first payment date, and then the prorated coupon is calculated. In the event of a long period, the second prior date is determined, and then the prorated coupon over the interval from issue to the first prior date is calculated. The total first coupon payment is then this prorated amount plus a regular payment {[see 5.1.1]}.

The maturity date is a business day and is appended to each date vector. It is provisionally assumed to follow a normal payment period, and thus the final payment is set to the regular coupon amount plus the face value of the bond at this point (6). Then, one more conditional statement checks if the final period is regular or not. If so, an adjustment is made to the final payment. Similar calculations to an irregular first period are performed, but using prospective rather than retrospective extensions (7).

Finally (8), the final payment consisting of the final coupon payment and return of face value is appended to the vector of payments.

Bond Valuation

Although bonds can be traded at whichever price a buyer and seller agree, traders will usually require having access to the benchmark “fair” price, namely the sum of the discounted payments remaining as of the bond settlement date. As noted in Section {[5.1.2]}, the buyer of a bond becomes entitled to receive all payments that are due strictly *after* the settlement date. This introduces a special case that is addressed in the code, namely if bond settlement occurs on a due date, the coupon payment is paid to the seller. Therefore, only those coupon payments due after settlement add to the bond value in the form of discounted amounts. If a due date falls on a weekend, it cannot be a settle date, and therefore the payment date is rolled to the next Monday and is payable to the buyer. It is for this reason the `Bond` class has

both due date and payment date vectors as data members.

```
double Bond::discounted_value(const ChronoDate&
bond_settle_date,
     const YieldCurve& yield_curve)
{
        // The buyer receives the payments which fall due
after the bond_settle_date
        // If the bond_settle_date falls on a due_date the
seller receives the payment
        double pv{ 0.0 };
        for (size_t i{ 0 }; i < due_dates_.size(); i++)
        {
                if (bond_settle_date < due_dates_[i])
                        pv +=
yield_curve.discount_factor(bond_settle_date,
payment_dates_[i])
                                * payment_amounts_[i];
        }
        return pv;
}
```

The code will loop through the `due_dates_` member vector until the first due date strictly later than settlement is located. At this point, each remaining payment – starting with the same current index as `due_dates_` — is obtained from the `payment_amounts_` vector. Each payment value is discounted from the payment date back to the bond settlement date. The discount factor that multiplies each payment is easily obtained by the eponymous member function on the `yield_curve` input object. The cumulative sum of these discounted payments is then returned as the fair market value of the bond.

As you may notice, this valuation function is short and compact, as the due dates have been generated by the constructor, along with the payment dates adjusted for business days where necessary. The payment amounts – including the final payment comprised of the last coupon payment and return of face value – were also computed by the constructor, including any adjustment to the first payment in the event of an irregular short or long

payment period.

The discount factors back to the bond settlement date are easily obtained from the member function `discounted_value` on the input `yield_curve` object, while all the date functionality is wrapped in the `ChronoDate` class. In essence, the `discounted_value` function doesn't "need to care" about how the discount factors or date calculations are obtained. It just uses public member functions on the objects to get the information it needs and computes the result.

A Bond Valuation Example

We can now put the individual classes previously presented into an example of pricing a bond. Recall that the `YieldCurve` abstract base class will required a derived curve fitting method. Again, there are many different approaches available, ranging from simple to highly advanced, but to keep the example concise, we will use linear interpolation. In constructing a `Bond` object, we will need to supply the face value, issue date, first coupon date, penultimate payment date, and maturity of the bond, along with its face value.

{UML diagram here} As an example, suppose the term sheet of a 20-year bond is as follows:

Table 4-1. Table 9-1: Contractual bond terms - example

Face Value	$1000
Annual Coupon Rate	6.2%
Payment Frequency	Every six months (semiannual)
Issue Date	Mon 8 May 2023

First Coupon Date	Tue 7 Nov 2023
Penultimate Coupon Date	Wed 7 May 2042
Maturity Date	Fri 7 Nov 2042

Data in practice would be taken in from an interface and converted to `ChronoDate` types, but we can replicate the result as follows:

```
std::string bond_id = "20 yr bond"; // normal 20 year bond

ChronoDate issue_date{ 2023, 5, 8 };
// (Mon)
ChronoDate first_coupon_date{ 2023, 11, 7 };            //
Short first coupon (Tue)
ChronoDate penultimate_coupon_date{ 2042, 5, 7 };       //
(Wed)
ChronoDate maturity_date{ 2042, 11, 7 };                //
Long final coupon (Fri)

int coupon_frequency{ 2 };
double coupon_rate{ 0.062 };
double face_value{ 1000.00 };
Construction of the bond is then straightforward:
Bond bond_20_yr{ bond_id, issue_date, first_coupon_date,
penultimate_coupon_date,
                maturity_date, coupon_frequency,
coupon_rate, face_value, day_count };
```

Recall, however, the due dates, payment dates, and payment amounts are all generated and adjusted in the body of the constructor. Each due date will carry a day value of 7, and the payment dates are the same except for the due dates falling on weekends that are rolled to the next Monday:

2026-11-09, 2027-11-08, 2028-05-08,

2032-11-08, 2033-05-09, 2034-05-08,

2037-11-09, 2038-11-08, 2039-05-09

The regular coupon amount is

$$\frac{1000(0.0625)}{2} = \$31.00$$

The only irregular period will be the from settle to the first coupon date, 2023-05-08 to 2023-11-07, resulting in the calculated coupon amount as

$$31\left(\frac{183}{184}\right) = \$24.86$$

where $\frac{183}{184}$ is the ratio of the actual number of days in the first period by the number of days from the first prior date to first coupon date.

Next, suppose we want to value the bond on a date between issue and first coupon date, say Tuesday, 10 October 2023. Suppose also market data as of this date imply the following spot yields:

Table 4-2. Table 9-2: Spot yields - example (figures rounded)

Period	Maturity	Year Fraction	Yield
Overnight	2023-10-11	0.00274	2%
3 Month	2024-01-10	0.25205	2.19%
6 Month	2024-04-10	0.50137	2.37%
1 Year	2024-10-10	1.00274	2.67%
2 Year	2025-10-10	2.00274	3.12%
3 Year	2026-10-12	3.00822	3.43%

5 Year	2028-10-10	5.00548	3.78%
7 Year	2030-10-10	7.00548	3.93%
10 Year	2033-10-10	10.0082	4%
15 Year	2038-10-11	15.0137	4.01%
20 Year	2043-10-12	20.0192	4.01%
30 Year	2053-10-10	30.0219	4%

Create two vectors containing the maturities (as year fractions) and discount bond prices above (again, in place of containers that would normally be initialized in an interface):

```
std::vector<double> maturities{0.00273973, 0.252055, . . ., 30.0219};
std::vector<double> spot_yields{0.0200219, 0.021924, . . ., 0.0400049};
```

And also the settlement date:

```
ChronoDate spot_settle_date{ 2023, 10, 10 };
```

With these, we can create an instance of a linearly interpolated yield curve:

```
LinearInterpYieldCurve yc{ spot_settle_date, maturities , spot_yields };
```

Then, to value the bond as of the same settlement date, provide the settlement date and yield curve data to the corresponding member function on the `Bond` object:

```
double value =
bond_20_yr.discounted_value(spot_settle_date, yc);
```

This function will locate the first due date after settlement (in this case the first coupon date), compute each continuously compounded discount factor from each payment date back to the settle date using interpolated rates off of the yield curve, multiply each payment by this discount factor, and sum the discounted values to determine the discounted value of the bond. In this example, the result is $1315.34.

Note the design separating the bond data "interface" from the "implementation" provides flexibility in two respects. First, as noted above, a `Bond` object can be created once, and then multiple random or shocked yield curve scenarios can be applied to the valuation of the same bond. This can make calculations of risk measures more efficient by obviating the need to create a whole new `Bond` object for each scenario. Thousands of scenarios are often applied in these situations, and there can be thousands of bonds in a portfolio, held in multiple bond portfolios across all the international trading operations within a financial institution. Avoiding new object creation at every step can make a measurable difference in the time required to calculate risk values.

The other case could be where the bond settlement date is set for some point in the (near) future, but an expected valuation as of current market conditions is needed today. As long as the bond settlement date is on or after yield curve settlement, the valuation will be valid.

Summary

TBD

References

{0} Nicolai Josuttis, The C++ Standard Library 2E, Sec 5.7.1, pp 143-44

{1} `std::chrono` date GitHub repository:

https://github.com/HowardHinnant/date

{2} Howard Hinnant, Stack Overflow (Fact 5), *https://stackoverflow.com/questions/59418514/using-c20-chrono-how-to-compute-various-facts-about-a-date*

{3} `chrono`-Compatible Low-Level Date Algorithms *https://howardhinnant.github.io/date_algorithms.html*

{4} Howard Hinnant, Stack Overflow, "C++ chrono: Determine Whether a Day is a Weekend" *https://stackoverflow.com/questions/52776999/c-chrono-determine-whether-day-is-a-weekend*

{5} Howard Hinnant, Stack Overflow, "How Do I Add a Number of Days to a Date in C++20 chrono" *https://stackoverflow.com/questions/62734974/how-do-i-add-a-number-of-days-to-a-date-in-c20-chrono*

{6} ISDA 30/360 Day Count Basis *https://www.iso20022.org/15022/uhb/mt565-16-field-22f.htm*

{6.5} Steiner pp 40-41

{7} Kenneth J Adams, Smooth interpolation of zero curves, Algo Research Quarterly, 4(1/2):11-22, 2001

{8} Hagan and West, Interpolation Methods for Curve Construction, Applied Mathematical Finance, Vol. 13, No. 2. 89-129, June 2006

{9} C++ Add months to chrono::system_clock::time_point, Stack Overflow, *https://stackoverflow.com/questions/43010362/c-add-months-to-chronosystem-clocktime-point/43018120#43018120* (Not referenced directly)

Chapter 5. Linear Algebra

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 10th 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

Linear algebra is an essential part of computational finance, and as such it is a necessary and fundamental component for financial C++ software development. Options existing at present in the Standard Library are mostly limited to the `valarray` container, to be discussed briefly below. Over the last 15 years or so, some very good open source matrix algebra libraries have emerged that have been adopted by the financial software industry, as well as other computationally intensive domains such as data science and medical research. Progress is also being made toward linear algebra capabilities eventually being adopted into the Standard Library in C++23 and C++26.

As C++ did not have all the convenient built-in multidimensional array capabilities that came with Fortran platforms, quantitative programmers making the transition to C++ back in the 1990's often found themselves in an

inconvenient situation with limited options. These included building up this functionality mostly from scratch, wrestling with interfaces to numerical Fortran libraries such as BLAS and LAPACK, or somehow convincing management to invest in a third-party C++ commercial library.

Contrived DIY solutions that were sometimes employed, based on what was available in C++ at the time, included representing a matrix as a `vector` of `vector`(s), or holding data in a two-dimensional dynamic C-array. Neither of these was particular palatable, with the former being cumbersome and inefficient, and the latter exposing the software to the risks associated with raw pointers and dynamic memory management. One seemingly useful feature available in the Standard Library, but not without controversy, was `std::valarray`. It has survived to the current day, and it provides vectorized operations and functions highly suitable for matrix and vector math. Its pros and cons will be presented momentarily.

The situation has improved substantially over the years with the release of multiple open-source linear algebra libraries for C++. Among these, two that have gained considerable critical mass in computational finance are the Eigen library **{1}**, and Armadillo **{2}**. A third option that has risen to some prominence in high-performance computing (HPC) is the Blaze library **{3}**. An earlier library, uBLAS, **{4}** is also available as part of the Boost libraries; however, it does not include the matrix decompositions and other capabilities available in the aforementioned offerings.

As a side note, open source R interface packages are available for each of these libraries. **{5}** These packages enable the integration of C++ code dependent on one or more of these libraries into R packages, usually to enhance run-time performance.

More recently, NVIDIA has released GPU-accelerated C++ linear algebra libraries as part of its HPC SDK. **{6}**.

A comparative list of both open source and commercial C++ linear algebra libraries, including those mentioned here, can be found on Wikipedia.**{7}** A more in-depth view of the Eigen library follows later in this chapter.

New features planned for C++23 and C++26 look set to finally provide the

Standard Library with long overdue robust and well-supported linear algebra capabilities. Central among these new features is the `std::mdspan` multi-dimensional array representation planned for C++23. C++26 should then be updated with both a standardized interface to external BLAS-compatible libraries, as well as its own set of linear algebra facilities.

Below, we will first take a trip back in time and examine convenient mathematical features of `valarray`, and then demonstrate how it can be used as a proxy for a matrix. Following that, we will dive into the Eigen library of the present and demonstrate the basic matrix operations, along with matrix decompositions frequently used in financial modeling. Finally, a glimpse of the proposals for near-future Standard Library releases will also be presented.

valarray and Matrix Operations

The workhorse STL container `std::vector`, as we have seen, is an option for representing a vector in the mathematical sense. Common vector arithmetic, such as inner products, can be performed using STL algorithms. However, having been "designed to be a general mechanism for holding values…and to fit into the architecture of containers, iterators, and algorithms" {2.5 Stroustrup, Tour 2E} **{8}**, the common arithmetic vector operators such as addition and multiplication were not included as members in its implementation.

A Standard Library container class separate from the STL, called `valarray`, does support arithmetic operators and provides "optimizations that are often considered essential for serious numerical work" {ibid}. With slice and stride functions also accompanying the `valarray` class, it can also facilitate representation arrays of higher dimension, in particular a matrix.

While `valarray` has these very useful properties that would seem to make it an obvious choice for matrix math, it has played to mixed reviews. This dates back to its original specification which was never fully complete due to debates over whether to require a new technique at the time, expression

templates (to be introduced shortly), that can significantly optimize performance. In the end, this was not mandated. As a result, "initial implementations were slow, and thus users did not want to rely on it." **{9}**

As of the time of this writing, however, two of the mainstream Standard Library distributions have implemented expression template versions of `valarray`, namely those that accompany the gcc and Clang compilers. In addition, the Intel oneAPI DPC++/C++ Compiler **{10}** ships with its own high-performance implementation of `valarray`. And as an incidental remark, specializations of `begin` and `end` functions were included as enhancements in C++11.

The moral of the story seems to be: know the capabilities of the implementation you intend to use. If its performance is suitable for your needs, then it can potentially be a very convenient option for matrix/vector operations, and vectorized versions of common mathematical functions. In addition, examining the properties of `valarray` may provide some context for future linear algebra enhancements planned for the Standard Library, with similar functionality in some cases, even though the implementations behind the scenes will be considerably different.

Arithmetic Operators and Math functions

The `valarray` container supports the standard arithmetic operators on an element-by-element basis, as well as scalar multiplication.

For example, the vector sum expression $3\mathbf{v}_1 + \frac{1}{2}\mathbf{v}_2$ can be naturally transcribed from a mathematical statement into C++ using `valarray` objects:

```
import <valarray>;
. . .

        std::valarray<double> v1{ 1.0, 2.0, 3.0,
                                  1.5, 2.5 };

        std::valarray<double> v2{ 10.0, -20.0, 30.0,
```

```
                                    -15.0, 25.0 };

          double vec_sum = 3.0 * v1 + 0.5 * v2;    // vec_sum
  is also a valarray <double>
```

The result is

```
  8 -4 24 -3 20
```

Element-by-element multiplication is also implemented with the `*` operator:

```
  double prod = v1 * v2;
```

This gives us

```
  10 -40 90 -22.5 62.5
```

The dot (or inner) product of v_1 and v_2 is easily obtained by summing the preceding result by invoking the `sum()` member function on the `valarray` class:

```
  double dot_prod = prod.sum();   // Result = 100
```

In addition to `sum`, `valarray` also has `max` and `min` functions, along with an `apply(.)` member function that applies an auxiliary function similar to `std::transform`:

```
  double v1_max = v1.max();       // 3.0
  double v1_min = v1.min();       // 1.0

  // u and w are valarray<double> types
  auto u = v1.apply([](double x) -> double {return x * x; });
  // Result: 1, 4, 9, 2.25, 6.25

  auto w = v1.apply([](double x) -> double {return
  std::sin(x) + std::cos(x);});
  // Result: 1.38177 0.493151 -0.848872 1.06823 -0.202671
```

A subset of the `cmath` functions is conveniently defined for vectorized operations on the entirety of a `valarray`. For example, the following operations will return a `valarray` containing the images of the respective functions applied to each element in `v1` and `neg_val` below. Note that we can also negate each element in the same way as a plain numerical type with the subtraction operator.

```
// The result in each is a valarray<double>
auto sine_v1 = std::sin(v1);
auto log_v1 = std::log(v1);
auto abs_v1 = std::abs(neg_val);
auto exp_v1 = std::exp(neg_val);
auto neg_v1 = - v1;
```

Finally, as of C++11, specializations of the `begin` and `end` functions analagous to those provided for STL containers have been implemented for `valarray`. A simple example is as follows:

```
template<typename T>
void print(T t) { cout << t << " "; }

std::for_each(std::begin(w), std::end(w), print<double>);
```

Given the `apply(.)` member function on `valarray` and the built-in vectorized mathematical functions that are already available, STL algorithms `for_each` and `transform` might not be needed as often in the case of `valarray` compared to STL containers, however.

valarray as a Matrix Proxy

`valarray` provides the facilities to represent multidimensional arrays. In our case, we are specifically concerned with representing a two-dimensional array as a proxy for a matrix. This can be achieved with the `slice(.)` member function that can extract a reference to an individual row or column.

To demonstrate this, let us first lighten the notation by defining the alias

```
using mtx_array = std::valarray<double>;
```

Then, create a `valarray` object `val`, with the code formatted in a way to make it look like a 4×3 matrix:

```
mtx_array val{ 1.0, 2.0, 3.0,
                            1.5, 2.5, 3.5,
                            7.0, 8.0, 9.0,
                            7.5, 8.5, 9.5};
```

The first row can be retrieved using the `std::slice` function, defined for a `valarray`, using the square bracket operator.

```
auto slice_row01 = val[std::slice(0, 3, 1)];
```

What this says is:

- Go to the first element of the `valarray`: index 0, value = 1.0.
- Choose 3 elements, beginning with the first
- Using a *stride* of 1, which in this case means to choose three consecutive rowwise elements

Similarly, the second column can be retrieved, using in this case a stride of 3, the number of columns:

```
auto slice_col02 = val[std::slice(1, 4, 3)];            //
The 2nd rowwise element has index 1
```

It is important to note the `slice(.)` function returns a lighter `slice_array` type — that acts as a reference to the selected elements — rather than a full `valarray`. It does not, however, provide the necessary member functions and operators to access individual elements or compute, say, new rows comprising a matrix product. If we want to apply these functions to row or column data, we will need to construct corresponding new `valarray` objects. This will be seen in the next example, computing

the dot product of a row in one matrix by the column in another, a necessary option in carrying out matrix multiplication.

To demonstrate this, suppose we have a 5×3 and a 3×5 matrix, each represented as a `valarray`. Note that we are also storing the number of rows and columns of each in separate variables.

```
mtx_array va01{ 1.0, 2.0, 3.0,
                    1.5, 2.5, 3.5,
                        4.0, 5.0, 6.0,
                        4.5, 5.5, 6.5,
                            7.0, 8.0, 9.0 };

unsigned va01_rows{ 5 }, va01_cols{ 3 };

mtx_array va02{ 1.0, 2.0, 3.0, 4.0, 5.0,
                            1.5, 2.5, 3.5, 4.5, 5.5,
                            5.0, 6.0, 7.0, 8.0, 8.5 };

unsigned va02_rows{ 3 }, va02_cols{ 5 };
```

If we were to apply matrix multiplication, it would require taking the dot product of each row of the first “matrix” by each column of the second. As an example, in order to get the dot product of the third row by the second column, we would first need the slice for each:

```
auto slice_01_row_03 = va01[std::slice(9, va01_cols, 1)];
auto slice_02_col_02 = va02[std::slice(1, va02_rows, 5)];
```

However, neither element-by-element multiplication nor the `sum()` member function is defined on a `slice_array`, so we need to construct corresponding `valarray` objects:

```
mtx_array va01_row03{ slice_01_row_03 };
mtx_array va02_col02{ slice_02_col_02 };
```

The dot product is then computed in the usual way:

```
double dot_prod = (va01_row03 * va02_col02).sum();
```

CAUTION

As previously noted, a `slice_array` acts as a reference to a block within a `valarray`. Operations and member functions such as `\*` and `sum` are not defined on `slice_array`, but assignment operators such as `*=` and `+=` are. Therefore, modification of a `slice_array` such as in the following example will also be reflected in the `valarray` itself. If we take the first row of `va01` as a slice:

```
auto slice_01_row_01 = va01[std::slice(0, va01_cols, 1)];
```

and then apply assignment operators

```
slice_01_row_01[0] *= 10.0;
slice_01_row_01[1] += 1.0;
slice_01_row_01[2] -= 3.0;
```

then the `valarray` contents would be

```
10      3       0
1.5     2.5     3.5
7       8       9
7.5     8.5     9.5
```

In summary, `valarray` conveniently provides the ability to apply mathematical operators and functions on an entire array, similar to Fortran 90, as well as more math-focused languages such as R and Matlab . As a reminder, however, performance can be highly dependent on the implementation used in your Standard Library distribution.

More information about `valarray`, its history, and its pros and cons can be found in the online supplemental chapter accompanying Josuttis, _The C++ Standard Library, second edition). **{11}**

Subsequent to `valarray` and C++98, there have been some very positive developments regarding linear algebra in C++, some of which will now be presented.

Eigen

The first release of the Eigen library became available in 2006. Since then, it has been expanded to version 3.4.0 as of August of 2021. Starting with version 3.3.1, it has been licensed under the reasonably liberal Mozilla Public License (MPL) 2.0.

Eigen is comprised of template code that makes inclusion into other C++ projects very easy, in that in its standard installation there is no linking necessary to external binary files. Its incorporation of expression templates, facilitating lazy evaluation, provides for enhanced computational performance. It also received a a further boost in popularity after being chosen for incorporation into the well-respected TensorFlow **{12}** machine learning library, as well as the Stan Math Library. **{13}** More background on its suitability and popularity in finance is presented in a recent Quantstart article **{14}**.

Finally, the Eigen library is very well documented, with a tutorial and examples to help the newcomer get up and running quickly.

Lazy Evaluation

Lazy evaluation defers and minimizes the number of operations required in matrix and vector operations. Expression templates in C++ are used to encapsulate arithmetic operations – that is, expressions – inside templates such that they are delayed until they are actually needed. This can reduce the total number of operations, assignments, and temporary objects that are created when using conventional approaches.

An example of lazy evaluation that follows is based on a more comprehensive and illustrative discussion in the book by Peter Gottschling on modern C++ for scientific programing. **{15}**

Suppose you have four vectors in the mathematical sense, each with the same fixed number of elements, say

$$\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4$$

and you wish to store their sum in a vector *y*. The traditional approach would be to define the addition operator, take successive sums and store them in temporary objects, ultimately computing the final sum and assigning it to *y*.

In code, the operator could be defined in the generic sense such that vector addition would be defined for any arithmetic type:

```
template <typename T>
std::vector<T> operator + (const std::vector<T>& a,
        const std::vector<T>& b)
{
        std::vector<T> res(a.size());
        for (size_t i = 0; i < a.size(); ++i)
                res[i] = a[i] + b[i];
        return res;
}
```

Computing the sum of four vectors as follows

```
        vector<double> v1{ 1.0, 2.0, 3.0 };
        vector<double> v2{ 1.5, 2.5, 3.5 };
        vector<double> v3{ 4.0, 5.0, 6.0 };
        vector<double> v4{ 4.5, 5.5, 6.5 };

        auto y = v1 + v2 + v3 + v4;
```

results in the following:

- 4 - 1 = 3 `vector` instances created (two temporary plus one final `y` instance)
- (4 - 1) $\times$ 3 assignments of `double` variables

As the number of vectors (say *m*) and the number of elements in each vector (say *n*) get larger, this generalizes to

- m – 1 `vector` objects allocated on the heap: m - 2 temporary, plus one return object (`y`)
- $(m-1)n$ assignments

With lazy evaluation, we can reduce the total number of steps, and thus improve efficiency for "large" *m* and *n*. More specifically, this can be accomplished by delaying the addition until all the data is ready, and then only at that time perform the sums for each element in the result. In code, this could be accomplished by writing a function as follows.

```
template <typename T>
std::vector<T> sum_four_vectors(const std::vector<T>& a,
const std::vector<T>& b,
        const std::vector<T>& c, const std::vector<T>& d)
{
        // Assume a, b, c, and d all contain the same
        // number of elements:
        std::vector<T> sum(a.size());

        for (size_t i = 0; i < a.size(); ++i)
        {
                sum[i] = a[i] + b[i] + c[i] + d[i];
        }

        return sum;
}
```

Now, in this case,

- There are *no* temporary `vector` objects created; only the `sum` result is necessary
- The number of assignments is reduced to *n* = 4

The Eigen documentation provides additional background in the section Lazy Evaluation and Aliasing.

The previous example demonstrates how lazy evaluation can work, but the obvious problem is it would be unrealistic to write individual sum functions

for all possible fixed numbers of vectors. Generalizing with expression templates is a far more challenging problem that will not be included here, but more information can be found in the Gottschling book {ibid 12}, as well as in Chapter 27 of the comprehensive book on C++ templates by Vandevoorde, Josuttis, and Gregor **{16}**.

Like any other optimization tool, it should not be applied blindly in the belief it will automatically make your code more efficient, as again there are cases where performance could actually be degraded.

Finally, for a very interesting presentation of a real-world case study of expression templates in financial risk management, a talk on the subject presented by Bowie Owens at CppCon 2019 **{17}** is very well worth watching.

Eigen Matrices and Vectors

The heart of the Eigen library is, not surprisingly, the `Matrix` template class. It is scoped with the `Eigen` namespace and requires the `Dense` header file be included. At the time of this writing, corresponding module imports have not yet been standardized. This means the header file will need to be included in the global fragment of a module.

The `Matrix` class carries six template parameters, but a variety of aliases are provided as specific types. These include fixed square matrix dimensions up to a maximum of four, as well as dynamic types for arbitrary numbers of rows and columns. The numerical type that a `Matrix` holds is also a template parameter, but this setting is also incorporated into individual aliases. For example, the following code will construct and display a fixed 3 $\times$ 3 matrix of `double` values, and a 4 $\times$ 4 matrix of $\int$s. Braced (uniform) initialization by row can be used to load the data at construction.

```
#include <Eigen/Dense>
. . .

Eigen::Matrix3d dbl_mtx                  // Contains
```

```
'double' elements
{
        {10.6, 41.2, 2.16},
        {41.9, 5.31, 13.68},
        {22.47, 57.43, 8.82}
};

Eigen::Matrix4i int_mtx                 // Contains 'int'
elements
{
        {24, 0, 23, 13},
        {8, 75, 0, 98},
        {11, 60, 1, 3 },
        {422, 55, 11, 55}
};

cout << dbl_mtx << endl << endl;
cout << int_mtx << endl << endl;
```

Note also that the << stream operator is overloaded, so the result can be easily displayed on the screen (in row-major order).

```
 10.6  87.4 58.63
 41.9  53.1 13.68
22.47 57.43  88.2

 24   0  23  13
  8  75   0  98
 11  60   1   3
422  55  11  55
```

Individual rows and columns can also be accessed, using 0-based indexing. The first column of the first matrix, and the third column of the second, for example are obtained with respective accessor functions:

```
cout << dbl_mtx.col(0) << endl << endl;
cout << int_mtx.row(2) << endl << endl;
```

This results in the following screen output:

```
10.6
41.9
22.47

11 60  1  3
```

Technically speaking, the type returned by either the `row` or `col` accessor is an `Eigen::Block`. It is similar to a `slice_array` accessed from a `valarray`, in that it acts as a lighter weight reference to the data. Unlike `slice_array`, it does not carry any mathematical operators such as `+=`.

For most of the financial examples considered in this book, the dimensions of a matrix will not be known a priori, nor will they necessarily be of a square matrix. In addition, the contents will usually be real numbers. For these reasons, we will primarily be concerned with the Eigen dynamic form for `double` types, aliased as `Eigen::MatrixXd`.

NOTE

1. As just mentioned, we will primarily use the dynamic Eigen `MatrixXd` form of a matrix (with `d` indicating `double` numerical elements); however, member and non-member functions will usually apply to any class derived from the `Matrix` template class. Where these functions are discussed, their relations with `Matrix` rather than `MatrixXd` may also be mentioned. Similarly, vector representation in Eigen will use `VectorXd`.
2. Linear algebra will inevitably involve subscripts and superscripts, eg x_{ij}, where in mathematical notation *i* might run from 1 to *m*, and *j* from 1 to *n*. However, C++ is 0-indexed, so a mathematical statement where $i = 1$ will be represented by `i = 0` in C++, $j = n$ by `j = n - 1`, and so forth.

Construction of a `MatrixXd` can take on many forms. Data can be entered as before in row-major order, with the number of rows and columns implied by uniform initialization of individual rows. Alternatively, the dimensions can be used as constructor arguments, with the data input by streaming in

row-major order. And, one more approach is to set each element one-by-one. An example of each is shown here:

```
using Eigen::MatrixXd;
. . .

MatrixXd mtx0
{
        {1.0, 2.0, 3.0},
        {4.0, 5.0, 6.0},
        {7.0, 8.0, 9.0},
        {10.0, 11.0, 12.0}
};

MatrixXd mtx1{4, 3};            // 4 rows, 3 columns
mtx1 << 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0,
11.0, 12.0;

MatrixXd mtx3{2, 2};
mtx3(0, 0) = 3.0;
mtx3(1, 0) = 2.5;
mtx3(0, 1) = -1.0;
mtx3(1, 1) = mtx3(1, 0) + mtx3(0, 1);
```

Note that the round bracket operator serves as both a mutator and an accessor, as demonstrated in the third example.

Two special cases, where either the number of columns or rows is one, are aliased as `VectorXd` and `RowVectorXd`. Construction options are similar to the `MatrixXd` examples above, again as shown in the Eigen documentation:

```
using Eigen::VectorXd;
using Eigen::RowVectorXd;
. . .

VectorXd a      { {1.5, 2.5, 3.5} };            // A
column-vector with 3 coefficients
RowVectorXd b { {1.0, 2.0, 3.0, 4.0} };      // A row-
vector with 4 coefficients
```

```
Eigen::VectorXd v(2);
v(0) = 4.0;
v(1) = v(0) - 1.0;
```

Matrix and Vector Math Operations

Matrix addition and subtraction are are conveniently implemented as overloads of the `+` and `-` operators, not unlike `valarray`. Similarly, these apply to vectors. Unlike `valarray`, however, the multiplication operator `*` refers to matrix multiplication rather than an element-by-element product. A separate set of functions is available for a wide range of element-by-element operations.

To multiply two matrices `A` and `B`, the code follows in natural mathematical order:

```
MatrixXd A
{
        {1.0, 2.0, 3.0},
        {1.5, 2.5, 3.5},
        {4.0, 5.0, 6.0},
        {4.5, 5.5, 6.5},
        {7.0, 8.0, 9.0}
};

MatrixXd B
{
        {1.0, 2.0, 3.0, 4.0, 5.0},
        {1.5, 2.5, 3.5, 4.5, 5.5},
        {5.0, 6.0, 7.0, 8.0, 8.5}
};

MatrixXd prod_ab = A * B;
```

This gives us as output:

```
   19    25    31    37  41.5
22.75 30.25 37.75 45.25    51
 41.5  56.5  71.5  86.5  98.5
```

```
45.25 61.75 78.25 94.75   108
   64    88   112   136 155.5
```

> **CAUTION**
>
> In the Eigen documentation, it is strongly recommended to "not use the auto keywords with Eigen's expressions, unless you are 100% sure about what you are doing. In particular, do not use the auto keyword as a replacement for a `Matrix<>` type."
>
> The reasons behind this require an advanced discussion about templates that tie in with lazy evaluation. Lazy Evaluation provides advantages in efficiency, but it also can involve return types using `auto` that might be in the form of a reference rather than a full `Matrix` type. This can result in unexpected or undefined behavior. It becomes less of an issue as you become more familiar with various Eigen types, but for this introductory presentation, we will mostly heed this admonishment.
>
> More information can be found in the documentation. **{18}**

The * operator is also overloaded for matrix-vector and row vector-matrix multiplication. As an example, suppose we have a portfolio of three funds, with correlation matrix of the returns and the vector of individual fund volatilities given (annualized). As is a typical problem, we might need to construct the covariance matrix given the data in this form in order to calculate the portfolio volatility. First, to form the covariance matrix, we would pre- and post-multiply the correlation matrix by diagonal matrices containing the fund volatilities:

```
MatrixXd corr_mtx
{
        {1.0, 0.5, 0.25},
        {0.5, 1.0, -0.7},
        {0.25, -0.7, 1.0}
};

VectorXd vols{ {0.2, 0.1, 0.4 } };
```

```
MatrixXd cov_mtx = vols.asDiagonal() * corr_mtx *
vols.asDiagonal();
```

Note how the `VectorXd` member function `asDiagonal()` conveniently forms a diagonal matrix with the vector elements along the diagonal.

Then, given a vector of fund weights ω adding to 1, the portfolio volatility is then the square root of the quadratic form

$$\omega^{\mathsf{T}} \Sigma \omega$$

where Σ is the covariance matrix:

```
VectorXd fund_weights{ {0.6, -0.3, 0.7 } };
double port_vol = std::sqrt(fund_weights.transpose() *
cov_mtx * fund_weights);
```

For element-by-element matrix multiplication, use the `cwiseProduct` member function. As an example, to multiply the individual elements in matrices of like dimension, say $\mathbf{A}$ and $\mathbf{B}^{\mathsf{T}}$, we would write:

```
MatrixXd cwise_prod = A.cwiseProduct(B.transpose());
```

There is in fact a set of `cwise...` (meaning *coefficient-wise*) member functions on an Eigen `Matrix` that perform element-by-element operations on two compatible matrices, such as `cwiseQuotient` and `cwiseNotEqual`. There are also unary `cwise` member functions that return the absolute value and square root of each element. These can be found in the Eigen documentation here. **{19}**

The result of the `*` operator when applied to two vectors depends upon which vector is transposed. For two vectors u and v, the dot (inner) product is computed as

$$\mathbf{u}^{\mathsf{T}}\mathbf{v}$$

while the outer product results when the transpose is applied to v:

$$\boldsymbol{uv}^T$$

So, one needs to be careful when using the `*` operator with vectors. Suppose we have:

```
VectorXd u{ {1.0, 2.0, 3.0} };
VectorXd v{ {0.5, -0.5, 1.0} };
```

The respective results of the following vector multiplications will be different:

```
double dp = u.transpose() * v;                  // Returns
'double'
MatrixXd op = u * v.transpose();                // Returns
a Matrix
```

The first would result in a real value of 2.5, while the second would give us a singular 3×3 matrix:

```
  0.5 -0.5   1
    1   -1   2
  1.5 -1.5   3
```

To make life easier, Eigen provides a member function, `dot`, on the `VectorXd` class. By instead writing

```
dp = u.dot(v);
```

it should perhaps make it clearer which product we want. The result would be the same as before, plus the operation is commutative.

STL Compatibility

A very nice feature of both the Eigen `Vector` and `Matrix` classes is their compatibility with the Standard Template Library. This means you can iterate through an Eigen container, apply STL algorithms, and exchange data with

STL containers.

STL and `VectorXd`

As a first example, suppose you wish to generate 12 random variates from a t-distribution and place the results in a `VectorXd` container. The process is essentially the same as what we saw using a `std::vector` and applying the `std::generate` algorithm with a lambda auxiliary function:

```
VectorXd u(12);
// 12 elements
std::mt19937_64 mt(100);                                    //
Mersenne Twister engine, seed = 100
std::student_t_distribution<> tdist(5); // 5 degrees of
freedom
std::generate(u.begin(), u.end(), [&mt, &tdist]() {return
tdist(mt); });
```

> **NOTE**
>
> Prior to the recent Eigen 3.4 release, the `begin` and `end` member functions were not defined. In this case, you would need to instead use the `data` and `size` functions, as follows:
>
> ```
> std::generate(u.data()), u.data() + u.size(),
> [&mt, &tdist]() {return tdist(mt); });
> ```

Non-modifying algorithms such as `std::max_element` are also valid:

```
auto max_u = std::max_element(u.begin(), u.end());
// Returns iterator
```

Numeric algorithms, for example `std::inner_product`, can also be applied:

```
double dot_prod = std::inner_product(u.begin(), u.end(),
```

```
v.begin(), 0.0);
```

The cleaner C++20 range versions are also supported on `VectorXd`:

```
VectorXd w(v.size());
std::ranges::transform(u, v, w.begin(), std::plus{});
```

Constructing a Matrix from STL Container Data

Matrix data can also be obtained from STL containers. This is convenient, as data can often arrive via interfaces from other sources where Eigen might not be included. The key is to use an `Eigen::Map`, which sets up a reference to (view of) the `vector` data rather than taking a copy of it.

As a first example, data residing in a `std::vector` container can be transferred to an `Eigen::Map`, which in turn can be used as the constructor argument for a `MatrixXd`. Note that the `Map` takes in a pointer to the first element in the `vector` (using the `data` member function), and the number of rows and columns, in its constructor.

```
std::vector<double> v{ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0,
        7.0, 8.0, 9.0, 10.0, 11.0, 12.0 };

Eigen::Map<MatrixXd> mtx_map(v.data(), 4, 3);
```

By default, `mtx_map` will provide row/column access to the data. Note that unlike creating a `MatrixXd` as before with an initializer list of data, the order will be column-major rather than row-major. Using `cout` yields the following output:

```
1  5  9
2  6  10
3  7  11
4  8  12
```

As a `Map` is a lighter weight view of the data in `v`, it does not carry with it all of the functionality as found on a `Matrix` object, in a sense similar to a slice

taken from a*va* ← *ay*. If you need this functionality, then you can construct a `MatrixXd` instance by putting the `Map` object in its constructor:

```
MatrixXd mtx_from_std_vector{ mtx_map };
```

The default arrangement of the data in a `Map` will be in *column-major order*, which is different from the earlier `MatrixXd` examples constructed with numerical data. If row-major is required, you can specify row-major at the outset, but this will require an `Eigen::Matrix` template parameter with storage explicitly set to `RowMajor`, as `MatrixXd` does not have its own template parameter for storage method:

```
Eigen::Map<Eigen::Matrix<double, 4, 3, Eigen::RowMajor>>
    mtx_row_major_map{ v.data(), 4, 3 };
```

If the matrix is square, you can just transpose the `Map` in place to put it in row-major order:

```
// Square matrix, place in row-major order:
std::vector<double> sq_data{ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0,
        7.0, 8.0, 9.0 };

Eigen::Map<MatrixXd> sq_mtx_map{ sq_data.data(), 3, 3 };
sq_mtx_map.transposeInPlace();
```

The result is then:

```
1 2 3
4 5 6
7 8 9
```

WARNING

Attempting to transpose a non-square matrix using `Map` can result in a program crash. In this case, you will need to create a full `MatrixXd` object before applying `transposeInPlace`.

Applying STL Algorithms to a Matrix

STL algorithms can also be applied to matrices row by row, or column by column. Suppose we have a 4×3 matrix as follows:

```
MatrixXd vals
{
        { 9.0, 8.0, 7.0 },
        { 3.0, 2.0, 1.0 },
        { 9.5, 8.5, 7.5 },
        { 3.5, 2.5, 1.5 }
};
```

The `rowwise()` member function on an Eigen `Matrix` sets up an iteration by row. Each `row` is a reference to (view of) the respective data, so we can square each element of the matrix in place as follows:

```
for (auto row : vals.rowwise())
{
        std::ranges::transform(row, row.begin(), [](double
x) {return x * x; });
}
```

The `colwise()` member function is similar, in this case sorting each column of the matrix:

```
for (auto col : vals.colwise())
{
        std::ranges::sort(col);
}
```

The end result after applying both algorithms gives us

```
    9     4     1
12.25  6.25  2.25
   81    64    49
90.25 72.25 56.25
```

Each element has been squared, and each column has been rearranged in

ascending order.

Financial programmers often need to write code that will compute the log returns on a set of equity or fund prices. For example, suppose we have a set of 11 monthly prices for three ETF's, in the following three columns:

```
25.5  8.0 70.5
31.0  7.5 71.0
29.5  8.5 77.5
33.5  5.5 71.5
26.5  9.5 72.5
34.5  8.5 75.5
28.5  9.0 72.0
23.5  7.5 73.5
28.0  8.0 72.5
31.5  9.0 73.0
32.5  9.5 74.5
```

The first step in computing log returns is to compute the natural log of each price. This can be done by applying the `transform` algorithm row by row:

```
for (auto row : prices_to_returns.rowwise())
{
        std::ranges::transform(row, row.begin(), [](double
x) {return std::log(x); });
}
```

Then, to get the log returns, we need to subtract from each log price its predecessor. For this, we can apply the `adjacent_difference` numeric algorithm to each column:

```
for (auto col : prices_to_returns.colwise())
{
        std::adjacent_difference(col.begin(), col.end(),
col.begin());
}
```

This result is still an 11×3 matrix, with the first row still containing the logs of the prices in the first row.

```
   3.23868    2.07944    4.25561
  0.195309 -0.0645385 0.00706717
-0.0495969   0.125163  0.0875981
  0.127155  -0.435318 -0.0805805
 -0.234401   0.546544  0.0138891
  0.263815  -0.111226  0.0405461
 -0.191055  0.0571584 -0.0474665
 -0.192904  -0.182322  0.0206193
  0.175204  0.0645385 -0.0136988
  0.117783   0.117783 0.00687288
 0.0312525  0.0540672  0.0203397
```

What we want are the monthly returns alone, so we need to remove the first row. This can be achieved by applying the `seq` function, introduced in Eigen 3.4, which provides an intuitive way of extracting a submatrix view (an `Eigen::Block`) from a `Matrix` object. The example here shows how to extract all rows below the first:

```
MatrixXd returns_mtx{ prices_to_returns(Eigen::seq(1,
Eigen::last),
        Eigen::seq(0, Eigen::last)) };
```

What this says is:

1. Start with the second row (index 1) and include all rows down to the last row: `Eigen::seq(1, Eigen::last)`
2. Take all columns from the first (index 0) to the last: `Eigen::seq(0, Eigen::last)`
3. Use this submatrix data alone in the constructor for the resulting `returns_mtx`

The results held in `returns_mtx` are then the log returns alone:

```
  0.195309 -0.0645385 0.00706717
-0.0495969   0.125163  0.0875981
  0.127155  -0.435318 -0.0805805
 -0.234401   0.546544  0.0138891
```

```
 0.263815  -0.111226  0.0405461
-0.191055  0.0571584 -0.0474665
-0.192904  -0.182322  0.0206193
 0.175204  0.0645385 -0.0136988
 0.117783   0.117783 0.00687288
0.0312525  0.0540672  0.0203397
```

Now, suppose the portfolio allocation is fixed at 35%, 40%, and 25% for each respective fund (columnwise). We can get the monthly portfolio returns by multiplying the vector of allocations by `returns_mtx`:

```
VectorXd monthly_returns = returns_mtx * allocations;
```

The result is

```
 0.0443094
 0.0546058
 -0.149768
   0.14005
 0.0579814
-0.0558726
  -0.13529
 0.0837121
 0.0900555
 0.0376502
```

Matrix Decompositions and Applications

Matrix decompositions are of course essential for a variety of financial engineering problems. Below, we will discuss a few examples.

Systems of Linear Equations and the LU Decomposition

Systems of linear equations are ubiquitous in finance and economics, particularly in optimization, hedging, and forecasting problems. To show how solutions can be found in Eigen, let us just look at a generic problem and implement it in code. The LU (Lower/Upper-triangular matrix) decomposition is a common approach in numerical methods. As opposed to

coding it ourselves, Eigen can get the job done in two lines of code.

Suppose we want to solve the following system of linear equations for x_1, x_2, and x_3:

$$3x_1 - 5x_2 + x_3 = 0$$

$$-x_1 - x_2 + x_3 = -4$$

$$2x_1 - 4x_2 + x_3 = -1$$

This sets up the usual matrix equation

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

where

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^\mathsf{T}$$

The matrix $\mathbf{A}$ contains the coefficients, and the column vector $\mathbf{b}$ contains the constants on the right-hand side of the equations. The LU algorithm decomposes the matrix $\mathbf{A}$ into the product of lower- and upper-triangular matrices $\mathbf{L}$ and $\mathbf{U}$ in order to solve for $\mathbf{x}$.

In Eigen, form the matrix $\mathbf{A}$ and the vector $\mathbf{b}$:

–

```
MatrixXd A              // Row-major
{
        {3.0, -5.0, 1.0},
        {-1.0, -1.0, 1.0},
        {2.0, -4.0, 1.0}
};

VectorXd b
{
        {0.0, -4.0, 1.0}
};
```

The next step is to create an instance of the `Eigen::FullPivLU` class

with template parameter `MatrixXd`. This sets up the LU decomposition. To find the vector containing the solution, all that is left to do is call the `solve(.)` member function on this object. This means two lines of code, as promised:

```
Eigen::FullPivLU<MatrixXd> lu(A);
VectorXd x = lu.solve(b);
```

The solution for `x` is then (from top to bottom x_1, x_2, x_3)

_

```
2.5
1.5
  0
```

Several other decomposition methods are available in Eigen for solving linear systems, where as is often the case a choice between speed and accuracy needs to be considered. The LU decomposition in the example above is among the best for accuracy, although there are others that may be faster but do not offer the same level of stability. The complete list can be found here:

Basic linear solving **{20}**

A comparison of the matrix decomposition methods in Eigen is also available:

Catalogue of decompositions offered by Eigen **{21}**

Fund Tracking with Multiple Regression and the Singular Value Decomposition

Another common programming problem in finance is fund tracking with multiple regression. Examples include

- Tracking whether a fund of hedge funds is following its stated allocation targets by regressing its returns on a set of hedge fund style index returns.

- Tracking the sensitivity of a portfolio to changes in different market sectors.
- Tracking the goodness of fit of mutual funds offered in guaranteed investment products such as variable annuities, with respect to their respective fund group benchmarks.

In multiple regression, one is tasked with finding a vector

$$\widehat{\beta} = [\,\beta_1 \quad \beta_2 \quad \cdots \quad \beta_n\,]^{\mathsf{T}} // \widehat{\beta} = [\,\beta_1 \quad \beta_2 \quad \cdots \quad \beta_n\,]^{\top} // [\,\beta_1 \quad \beta_2 \quad \cdots \quad \beta_n\,]/$$

that satisfies the matrix form of the normal equations

$$\hat{\boldsymbol{\beta}} = [\,\mathbf{X}^{\mathsf{T}}\mathbf{X}\,]^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{Y} // \hat{\boldsymbol{\beta}} = [\mathbf{X}^{\mathsf{T}}\mathbf{X}]^{-1}\mathbf{X}^{\mathsf{T}}\mathbf{y}$$

where $\mathbf{X}$ is the design matrix containing *p* columns of independent variable data, and *n* observations (rows), with the number of observations *n* “comfortably” greater than the number of data columns *p* to ensure stability. This is typically the case in these types of fund tracking applications.

> **NOTE**
>
> For fund tracking applications, the intercept term β_0 can usually be dropped, so it is omitted here. For regression cases where an intercept is required, a final column of ones needs to be appended to the design matrix when using Eigen. _

A commonly employed solution is the compact Singular Value Decomposition (SVD), which replaces the matrix $\mathbf{X}$ with the decomposition $\mathbf{U}\Sigma\mathbf{V}^{T}$, where $\mathbf{U}$ is an $n \times p$ matrix, $\mathbf{V}$ is $p \times p$, and Σ is a $p \times p$ diagonal matrix of strictly positive values. Making this substitution in the original formula for $\widehat{\beta}$, we get

$$\widehat{\beta} = \mathbf{V}\Sigma\mathbf{U}^{\mathsf{T}}\mathbf{y}$$

Eigen provides two SVD solvers that can be used for obtaining the least squares estimates of the regression coefficients. The first of these, as described in the Eigen documentation, is the Jacobi SVD decomposition of a rectangular matrix_ **{22}**. The documentation also states the Jacobi version is recommended for design matrices of 16 columns or less, which is sometimes sufficient for fund tracking problems.

The way this works is given the design matrix predictor data contained in a `MatrixXd X`, Eigen sets up the SVD logic inside the class template `Eigen::JacobiSVD`. Then, given the response data contained in a `VectorXd Y`, solving for $\widehat{\beta}$ is again just two lines of code:

```
Eigen::JacobiSVD<MatrixXd> svd{ X, Eigen::ComputeThinU |
Eigen::ComputeThinV };
VectorXd beta = svd.solve(Y);
```

NOTE

The `Eigen::ComputeThinU | Eigen::ComputeThinV` bitwise-or parameter instructs the program to use the compact version of SVD. If the full SVD is desired, then the above `JacobiSVD` instantiation would be:

```
Eigen::JacobiSVD<MatrixXd> svd{ X, Eigen::ComputeFullU |
Eigen::ComputeFullV };
```

In this case, the $\mathbf{U}$ matrix will be $n \times n$, and Σ will be an $n \times p$ pseudoinverse.

There are no default settings.

As an example, suppose we have three sector ETF's and wish to examine their relationship to the broader market (eg the S&P 500), and suppose we have 30 daily observations.

The design matrix will contain the three ETF returns and is stored in a `MatrixXd` called `X`, as follows:

```
MatrixXd X{ 3, 30 };    // 3 sector funds, 30 observations
(will transpose)
X       <<
        // Sector fund 1
        -0.044700388, -0.007888394, 0.042980064,
0.016416586, -0.01779658, -0.016714149,
        0.019472031, 0.029853293, 0.023126097,
-0.033879088, -0.00338369, -0.018474493,
        -0.012509815, -0.01834808, 0.010626754,
0.036669407, 0.010811115, -0.035571742,
        0.027474007, 0.005406069, -0.010159427,
-0.006145632, -0.0103273, -0.010435171,
        0.011127197, -0.023793709, -0.028009362,
0.00218235, 0.008683152, 0.001440032,

        // Sector fund 2
        -0.019002703, 0.026036835, 0.03782709, 0.010629292,
-0.008382267, 0.001121697,
        -0.004494407, 0.017304537, -0.006106293,
0.012174645, -0.003305029, 0.027219671,
        -0.036089287, -0.00222959, -0.015748493,
-0.02061919, -0.011641386, 0.023148757,
        -0.002290732, 0.006288094, -0.012038397,
-0.029258743, 0.011219297, -0.008846992,
        -0.033738048, 0.02061908, -0.012077677,
0.015672887, 0.041012907, 0.052195282,

        // Sector fund 3
        -0.030629136, 0.024918984, -0.001715798,
0.008561614, 0.003406931, -0.010823864,
        -0.010361097, -0.009302434, 0.008142014,
-0.004064208, 0.000584335, 0.004640294,
        0.031893332, -0.013544321, -0.023573641,
-0.004665085, -0.006446259, -0.005311412,
        0.045096308, -0.007374697, -0.00514201,
-0.001715798, -0.005176363, -0.002884991,
        0.002309361, -0.014521608, -0.017711709,
0.001192088, -0.00238233, -0.004395918;

X.transposeInPlace();
```

Similarly, the market returns are stored in a `VectorXd` array `Y`:

```
VectorXd Y{ 30 };       // 30 observations of market
returns
Y <<
        -0.039891316, 0.00178709, -0.0162018, 0.056452057,
0.00342504, -0.012038314,
        -0.009997657, 0.013452043, 0.013485674,
-0.007898137, 0.008111428, -0.015424523,
        -0.002161451, -0.028752191, 0.011292655,
-0.007958389, -0.004002386, -0.031690771,
        0.026776892, 0.009803957, 0.000886608, 0.01495181,
-0.004155781, -0.001535225,
        0.013517306, -0.021228542, 0.001988701,
-0.02051788, 0.005841347, 0.011248933;
```

Obtaining the regression coefficients is just a matter of compiling and running the SVD using the two lines of code shown at the outset, which gives us for `beta`:

```
  0.352339
-0.0899004
  0.391252
```

The $\mathbf{U}$ and $\mathbf{V}$ matrices can also be obtained if desired, along with the Σ matrix, using the following accessor functions:

```
cout << svd.matrixU() << endl;                  // U: n x p
= 30 x 3
cout << svd.matrixV() << endl;                  // V: p x p
= 3 x 3
cout << svd.singularValues().asDiagonal() << endl;
// Sigma: p x p = 3 x 3
```

An alternative SVD solver, the *Bidiagonal Divide and Conquer SVD*, **{23}** is also available in Eigen. Per the documentation, it is recommended for design matrices with greater than 16 columns for better performance.

The setup is the same as the Jacobi case, but using the `BDCSVD` class in place of `JacobiSVD`:

```
Eigen::BDCSVD<MatrixXd> svd(X, Eigen::ComputeThinU |
Eigen::ComputeThinV);
VectorXd beta = svd.solve(Y);
```

It should be noted Eigen also provides QR decompositions as well as the capability of just coding in the normal equations with Eigen matrices and operations. As noted in the documentation, *Solving linear least squares systems*, these can be faster than SVD methods, but they can be less accurate. **{24}** These are alternatives you might want to consider if speed is an issue, but under the right conditions.

Correlated Random Equity Paths and the Cholesky Decomposition

The Cholesky decomposition is a popular tool in finance for generating correlated Monte Carlo equity path simulations. For example, when pricing basket options, covariances between movements in the basket securities need to be accounted for, specifically in generating correlated random normal draws. This is in contrast to generating a random price path for a single underlying security, as we saw in Chapter 8:

$$S_t = S_{t-1} e^{\left(\frac{r-\sigma^2}{2}\right)\Delta t + \sigma\varepsilon_t\sqrt{\Delta t}}$$

where again $\varepsilon_t \sim N(0,1)$, σ is the equity volatility, and r represents the risk-free interest rate,

In the case of a basket option, we now need to generate a path for each of say m assets at each time t, where the $\sigma\varepsilon_t$ term is replaced by a random term $w_t^{(i)}$ that is again based on a standard normal draw, but whose fluctuations also contain correlations with the other assets in the basket. Therefore, we need to generate a set of prices $\left\{S_t^{(i)}\right\}$ where

$$S_t^{(1)} = S_{t-1}^{(1)} e^{\left(\frac{r-\sigma^2}{2}\right)\Delta t + w_t^{(1)}\sqrt{\Delta t}} \quad (*)$$

.

.

.

$$S_t^{(m)} = S_{t-1}^{(m)} e^{\left(\frac{r-\sigma^2}{2}\right)\Delta t + w_t^{(m)}\sqrt{\Delta t}}$$

for each asset $i = 1, \cdots, m$ at each time step $t_j, j = 1, \cdots, n$. Our task is to calculate the random but correlated vector

$$\begin{bmatrix} w_t^{(1)} & \cdots & w_t^{(m)} \end{bmatrix}^\mathsf{T}$$

for each time t. This is where the Cholesky decomposition – available in Eigen – comes into play. For an $n \times n$ covariance matrix Σ, assuming it is positive definite, will have a Cholesky decomposition

$$\Sigma = LL^\mathsf{T}$$

where L is a lower triangular matrix. Then, for a vector of standard normal variates $\mathbf{z}$,

$$\begin{bmatrix} z_1 & z_2 & \cdots & z_m \end{bmatrix}^\mathsf{T}$$

the $n \times 1$ vector generated by $\mathbf{Lz}^\mathsf{T}$ will provide a set of correlated volatilities that can be used to generate in a single time step a random scenario of prices for each underlying security. For each time step t, we replace $\mathbf{z}$ with $\mathbf{z_t}$ to then arrive at our desired result:

$$\mathbf{w_t} = \mathbf{L}\mathbf{z}_t^\mathsf{T}$$

This can then be extended an arbitrary number of n time steps by placing each vector $\mathbf{z_t}$ into a column of a matrix, say $\mathbf{Z}$. Then, we can generate the entire set of vectors of correlated random variables in one step, and place the results in a matrix

$$\mathbf{W} = \mathbf{LZ}$$

Eigen provides a Cholesky decomposition of a `MatrixXd` object, using the `Eigen::LLT` class template with parameter `MatrixXd`. It again consists

of creating an object of this class, and then calling a member function, `matrixL`, which returns the matrix $\mathbf{L}$ above.

As an example, suppose we have four securities in the basket, with the following covariance matrix.

```
MatrixXd cov_basket
{
    { 0.01263, 0.00025, -0.00017, 0.00503},
    { 0.00025, 0.00138,  0.00280, 0.00027},
    {-0.00017, 0.00280,  0.03775, 0.00480},
    { 0.00503, 0.00027,  0.00480, 0.02900}
};
```

The Cholesky decomposition is set up when the matrix data is used to construct the `Eigen::LLT` object. Calling the member function `matrixL()` computes the decomposition and returns the resulting lower triangle matrix:

```
Eigen::LLT<Eigen::MatrixXd> chol{ cov_basket };
MatrixXd chol_mtx = chol.matrixL();
```

This gives us

```
    0.1124          0          0          0
  0.002226  0.0370332          0          0
-0.0015544  0.0756179   0.178975          0
 0.0447889 0.00464393  0.0252348   0.162289
```

Suppose now there will be six time steps in the Monte Carlo model over a period of one year. This means we will need six vectors containing four standard normal variates each. For this, we can use the Standard Library `<random>` functions and generate a 4×6 matrix, and place the random vectors in consecutive columns.

First, create a `MatrixXd` object with four rows and six columns:

```
MatrixXd corr_norms{ 4, 6 };
```

The first step will be to populate this matrix with uncorrelated standard normal draws. As we did previously (in Chapter 8), we can set up a random engine and distribution, and capture these in a lambda to generate the standard normal variates:

```
std::mt19937_64 mt_norm{ 100 };         // Seed is
arbitrary, just set to 100 again
std::normal_distribution<> std_nd;

auto std_norm = [&mt_norm, &std_nd](double x)
{
    return std_nd(mt_norm);
};
```

Because each column in a `MatrixXd` can be accessed as a `VectorXd`, we can again iterate columnwise through the matrix and apply the `std::ranges::transform` algorithm to each in a range-based `for` loop.

```
for (auto col : corr_norms.colwise())
{
    std::ranges::transform(col,
        col.begin(), std_norm);
}
```

This interim result would resemble the following, with actual results depending on the compiler (in this case using the Microsoft Visual Studio 2022 compiler):

```
   0.201395    0.197482     1.22857     1.40751     1.82789
-0.150014
 -0.0769593   0.0830647     1.86252    0.122389   -0.949222
0.667817
   0.936051     1.16233   -0.642932    0.538005    -1.82688
-0.451039
-0.00916217    -2.79186   -0.434655  -0.0553752     1.46312
0.345527
```

Then, to get the *correlated* normal values, multiply the above result by the Cholesky matrix, and reassign `corr_norms` to the result.

```
MatrixXd corr_norms = chol_mtx * corr_norms;
```

This result, which corresponds to the matrix $\mathbf{W}$ in the mathematical derivation, comes out to be:

```
  0.0226368   0.0221969    0.138091    0.158204    0.205455
-0.0168616
-0.00240174  0.00351574   0.0717097  0.00766557  -0.0310838
0.0243974
   0.161397    0.214002   0.0238613    0.103356   -0.401585
-0.0299926
  0.0307971   -0.414526  -0.0230881   0.0681989    0.268808
0.0410757
```

Each successive column in `corr_norms` will provide a set of four correlated random variates that can be substituted in for $w_t^{(1)} \cdots w_t^{(4)}$ at each time $t = 1, \cdots, 6$ in (*) above. First, we will need the spot prices of the four underlying equities, which can be stored in an Eigen `VectorXd` (pretend they are retrieved from a live market feed). For example:

```
VectorXd spots(4);          // Init spot prices from market
spots << 100.0, 150.0, 25.0, 50.0;
```

Next, we will need a matrix in which to store a random price path for each equity, starting with each spot price, to be stored in an additional first column, making it a 4×7 matrix.

```
MatrixXd integ_scens{ corr_norms.rows(), corr_norms.cols()
+ 1 }   // 4 x 7 matrix
integ_scens.col(0) = spots;
```

Suppose then, the time to maturity is one year, divided into six equal time steps. From this, we can get the Δt value, say `dt`.

```
double time_to_maturity = 1.0;
unsigned num_time_steps = 6;
double dt = time_to_maturity / num_time_steps;
```

Each successive price for a given underlying equity, as shown in (*), can be computed in a lambda, where `price` is the previous price in the scenario, and `vol` is the volatility of the particular equity.

```
auto gen_price = [dt, rf_rate](double price, double vol,
double corr_norm) -> double
{
    double expArg1 = (rf_rate - ((vol * vol) / 2.0)) * dt;
    double expArg2 = corr_norm * std::sqrt(dt);
    double next_price = price * std::exp(expArg1 +
expArg2);
    return next_price;
};
```

Finally, for each underlying equity at each time step, we can set up an iteration and call the lambda at each step:

```
for (unsigned j = 1; j < integ_scens.cols(); ++j)
{
    for (unsigned i = 0; i < integ_scens.rows(); ++i)
    {
        integ_scens(i, j) = gen_price(integ_scens(i, j -
1), vols(i), corr_norms(i, j - 1));
    }
}
```

For this example, the results are

```
100             100.99          101.972         107.952
115.226         125.384         124.6
150             150.086         150.535         155.248
155.976         154.248         156.034
25              26.6633         29.0545         29.2955
30.5129         25.8607         25.5082
50              50.5946         42.6858         42.2536
43.414          48.4132         49.1949
```

Again, results may vary due to differences in the implementation of `<random>` between different Standard Library releases among vendors.

Yield Curve Dynamics and Principal Components Analysis

Principal Components Analysis (PCA) is a go-to tool for determining the sources and magnitudes of variation that drive changes in the shape of a yield curve. Given a covariance matrix of daily changes in yields spanning a range of bond maturities, PCA is employed by first calculating the eigenvalues of this matrix, and ordering them from highest to lowest. Then, the weightings are calculated by dividing each eigenvalue by the some of all the eigenvalues.

Empirical research has shown the contribution of first three eigenvalues will comprise nearly the entirety of the weightings, where the first weighting corresponds to parallel shifts in the yield curve, the second corresponds to variations in its “tilt” or “slope”, and the third to the curvature. The reasons and details behind this can be found in Chapter 18 of the very fine computational finance book by Ruppert and Matteson **{25}**, and in Chapter 3 of the classic text on interest rate derivatives by Rebonato **{26}**.

Rigorous statistical tests exist for measuring significance, but the weights alone can provide a relative estimated measure of each source of variation.

Section 18.2 of the text by Ruppert and Matteson {op cit 25} provides an excellent example on how to apply principal components analysis to publicly available US Treasury yield data {put URL here}. The resulting covariance matrix — input as constructor data for the following `MatrixXd` object, is based on fluctuations in differenced US Treasury yields **{28}** for eleven different maturities, ranging from one month to 30 years. The underlying data is taken from the period from January 1990 to October 2008.

To calculate the eigenvalues, first load the covariance matrix data into the upper triangular region of a `MatrixXd` instance.

```
MatrixXd term_struct_cov_mtx
{
        // 1 month
        { 0.018920,     0.009889, 0.005820,
```

```
0.005103, 0.003813,     0.003626,
                        0.003136, 0.002646, 0.002015,
0.001438, 0.001303 },

                // 3 months
                { 0.0, 0.010107, 0.006123, 0.004796,
0.003532, 0.003414,
                        0.002893, 0.002404, 0.001815,
0.001217, 0.001109},

                // 6 months
                { 0.0, 0.0, 0.005665, 0.004677, 0.003808,
0.003790,
                        0.003255, 0.002771, 0.002179,
0.001567, 0.001400 },

                // 1 year
                        { 0.0, 0.0, 0.0, 0.004830,
0.004695, 0.004672,
                                0.004126, 0.003606,
0.002952, 0.002238, 0.002007},

                // 2 years
                { 0.0, 0.0, 0.0, 0.0, 0.006431, 0.006338,
                        0.005789, 0.005162, 0.004337,
0.003343, 0.003004},

                // 3 years
                { 0.0, 0.0, 0.0, 0.0, 0.0, 0.006524,
                        0.005947, 0.005356, 0.004540,
0.003568, 0.003231 },

                // 5 years
                { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                        0.005800, 0.005291, 0.004552,
0.003669, 0.003352 },

                // 7 years
                { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                        0.0, 0.004985, 0.004346, 0.003572,
0.003288 },

                // 10 years
```

```
                { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                        0.0, 0.0, 0.003958, 0.003319,
0.003085 },

                // 20 years
                { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                        0.0, 0.0, 0.0, 0.003062, 0.002858
},

                // 30 years
                { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                        0.0, 0.0, 0.0, 0.0, 0.002814 }
        };
```

As a real symmetric matrix is trivially self-adjoint (no complex components), Eigen can apply a function `selfadjointView<.>()` to an upper (or lower) triangular matrix to define a view of a symmetric covariance matrix. The eigenvalues (all real) can then be obtained by applying the `eigenvalues()` function on the symmetric result:

```
VectorXd eigenvals =
term_struct_cov_mtx.selfadjointView<Eigen::Upper>
().eigenvalues();
```

In order to determine the weight of each principal component, we need to divide each eigenvalue by the sum of all the eigenvalues:

```
double total_ev = std::accumulate(eigenvals.cbegin(),
eigenvals.cend(), 0.0);
std::ranges::transform(eigenvals, eigenvals.begin(),
        [total_ev](double x) {return x / total_ev; });
```

And finally, to view the results in order of the principal components, the weighted values need to be arranged from largest to smallest:

```
std::ranges::sort(eigenvals, std::greater{});
```

Examining the contents of `eigenvals`, we would find the following results:

```
0.67245 0.213209 0.073749
0.023811 0.00962511 0.00275773
0.0016744 0.00114298 0.000740139
0.000528862 0.000311391
```

From this, we can see the effects of parallel shifts and the “tilt” of the yield curve are the dominant effects, with relative weights of 67.2% and and 21.3%, while the curvature effect is smaller at 7.4%.

Future Directions: Linear Algebra in the Standard Library

Three proposals related to linear algebra have been submitted to the ISO C++ committee.

- `std::mdspan`: A polymorphic multidimensional array reference (P0009) **{28}**
- A free function linear algebra interface based on the BLAS (P1673) **{29}**
- Add linear algebra support to the C++ standard library (P1385) **{30}**

`mdspan` (P0009) can impose a multidimensional array structure on a reference to a container, such as an STL `vector`. Using the example of a `vector` containing the data, and a referring `mdspan` representing a matrix, the number of rows and columns are set at construction of the `mdspan`. An `mdspan` can also take the form of higher dimensional arrays, but for our purposes we will concern ourselves with the two-dimensional case for matrix representations. `mdspan` is officially slated for release in C++23.

The second proposal (P1673) is for a standard interface to external linear algebra “based on the dense Basic Linear Algebra Subroutines (BLAS)”, corresponding “to a subset of the BLAS Standard.” {op cit 29}. In other words, code could be written independently of whichever external linear algebra library is used, making code maintenance much easier and less error-prone. As noted in the proposal, the “interface is designed in the spirit of the

C++ Standard Library's algorithms", and "uses mdspan…, to represent matrices and vectors" {ibid}. Furthermore, the "interface is designed in the spirit of the C++ Standard Library's algorithms"{ibid}. It is currently planned for C++26.

The third proposal (P1385) is to provide actual BLAS-type functionality within the Standard Library. Its primary goal is to "provide a (sic) matrix vocabulary types for representing the mathematical objects and fundamental operations relevant to linear algebra" (op cit {30}). This proposal is also currently planned for release with C++26.

mdspan (P0009)

As mentioned above, for representing a matrix, `mdspan` establishes a reference to a contiguous container and then imposes the number of rows and columns. If these parameters are known at compile time, creating an `mdspan` is easy:

```
vector<int> v{ 101, 102, 103, 104, 105, 106 };
auto mds1 = std::mdspan{ v.data(), 3, 2 };
```

Note that `mdspan` uses the `data()` member function on `vector` to access its contents.

In `mdspan` parlance, rows and columns are referred to as *extents*, with the number of rows and columns accessed by the index of each, 0 for rows and 1 for columns. The total number of extents is referred to as the *rank*, so in this case, the rank is 2. For higher-order multidimensional arrays, the rank would be greater than two.

```
size_t n_rows{ mds1.extent(0) };                // 3
size_t n_cols{ mds1.extent(1) };                // 2
size_t n_extents{ mds1.rank() };                // 2
```

NOTE

> The term *rank* as applied to `mdspan` is not the same as the mathematical definition of the rank of a matrix. This naming might unfortunately seem confusing, but it's something one should be aware of.

Elements of the `mdspan` object will be accessible with another new feature in C++23, namely the square bracket operator with multiple indices:

```
for (size_t i = 0; i < mds1.extent(0); ++i)
{
        for (size_t j = 0; j < mds1.extent(1); ++j)
                cout << mds1[i, j] << "\t";

        cout << "\n";
}
```

This is a welcome improvement, as it will no longer be necessary to put each index in a separate bracket pair, as was the case with C-style arrays:

```
double ** a[3][2];              // Ugh

// Could put a[2, 1] if it were an mdspan instead:
a[2][1] = 5.4;

...

delete [] a;
```

The run-time result of the previous nested loop displays a 3×2 matrix:

```
101     102
103     104
105     106
```

It is also possible to define a matrix with different dimensions to the same data:

```
auto mds2 = std::mdspan(v.data(), 2, 3);                //
```

```
2 x 3
```

Applying the same loop above, but replacing `mds1` with `mds2` would then display as expected:

```
101     102     103
104     105     106
```

One thing to be careful of is modifying data in the `mdspan` object or in the original `vector` will change it in both locations due to the referential relationship between the two. For example, modification of the last element of the vector

```
v[5] = 874;
```

will show up in both of the `mdspan` objects. `mds1` becomes

```
101     102
103     104
105     874
```

and `mds2` is now

```
101     102     103
104     105     874
```

Likewise, changing the value of an element in `mds2` will be reflected in both `mds1` and the vector `v`.

In quant finance applications, it will often be the case where the fixed number of rows and columns are not known at compile time. Suppose `m` and `n` are the numbers of rows and columns to be determined at runtime. These dimensions can be set dynamically by replacing the fixed settings of 2 and 3 in the previous example

```
auto mds2 = std::mdspan(v.data(), 2, 3);
```

with the `std::extents{m, n}` object, as shown in the `mdspan` definition here:

```
void dynamic_mdspan(size_t m, size_t n, vector<double> vec)
{

        std::mdspan md{ vec.data(), std::extents{m, n} };

        . . .

}
```

The `std::extents{m, n}` parameter represents the number of elements in each extent — ie in each row (`m`) and column (`n`) — which are determined dynamically at runtime.

Pretend we have the following data set at runtime:

```
vector<double> w{ 10.1, 10.2, 10.3, 10.4, 10.5, 10.6 };
size_t m{3};
size_t n{2};
```

Using these as inputs to the `dynamic_mdspan(.)` function above will then generate a 3×2 `mdspan` matrix:

```
10.1    10.2
10.3    10.4
10.5    10.6
```

NOTE

The above examples make use of *Class Template Auto Deduction*, commonly referred to as *CTAD*. Both `mdspan` and `extents` are class templates, but because `w` in the example just above is a `vector` of `double` types, and the size of the `vector` being of type `size_t`, when substituted in for `vec` in the `dynamic_mdspan` function, the compiler will deduce that the `mdspan` and `extents` objects are to use `double` and `size_t` as template parameters.

Without CTAD, the function would be written something like:

```
template<typename T, typename S>
void dynamic_mdspan(S m, S n, T vec)
{
        using ext_t = std::extents<S, std::dynamic_extent,
std::dynamic_extent>;
        std::mdspan<T, ext_t> mds_dyn{ vec.data(), m, n };

        . . .

}
```

The discussion in this chapter will rely on CTAD, but in cases where more generality is required, it will be necessary to to write out the template arguments in full.

One more thing to note is in each of the examples so far, `mdspan` will arrange the data in row-major order by default. Arranging it in column-major order requires defining a `layout_left` policy *mapping*, in this case called `col_major`, as shown here:

```
std::layout_left::mapping col_major{ std::extents{ m, n }
};
```

The corresponding column-major matrix can then be defined by substituting this mapping in the extents argument in the `mdspan` constructor.

```
std::mdspan md{ v.data(), col_major };
```

The result is the column-major version of the matrix:

```
10.1    10.4
10.2    10.5
10.3    10.6
```

NOTE

Row-major order can also be set explicitly with the `std::layout_right`

mapping.

The `mdspan` proposal also includes a “slicing” function called `submdspan(.)` to return a reference to an individual row or column from a matrix represented by an mdspan. More generally, this would extend to subsets of higher-dimensional arrays.

Returning to the row-major integer 3×2 `mds1` example, if we wanted to extract a reference to the first row (index 0), it could be obtained as follows, with the index in the first extent (row) argument of `submdspan`:

```
auto row_1 = std::submdspan(mds1, 0, std::full_extent)
```

This would give us:

```
row_1[0] = 101  row_1[1] = 102
```

Rows 2 and 3 could also be referenced by replacing the `0` with `1` and `2`, respectively.

By explicitly setting the second (column) extent argument to the column size less 1, we could also access the last column:

```
auto col_last = std::submdspan(mds1, std::full_extent,
mds1.extent(1)-1);
```

This would then be comprised of:

```
col_2[0] = 102  col_2[1] = 104  col_2[2] = 106
```

As `submdspan` is a reference (view) to an `mdspan` containing a row or column, it will obviate generating an additional `mdspan`, in contrast to the issue with a `slice_array` taken from a `valarray`. Any public member function on `mdspan` can be applied to a `submdspan`. On the downside, this would not include the vectorized mathematical operators or functions that are

provided with `valarray`, although a different approach to the operators will be provided in `P1673`, to be discussed in the next section.

On the flip side, also because it is a reference, modifying an element of a `submdspan` will also modify the underlying `mdspan` object. Suppose the last element in `col_last` is reset:

```
col_2[2] = 3333;
```

The original `mds1` would then become:

```
101     102
103     104
105     3333
```

One final remark is a proposal for a multi-dimensional array (P1684), called `mdarray`, is also in review. As noted in the proposal **{31}**, "`mdarray` is as similar as possible to mdspan, except with container semantics instead of reference semantics". In other words, an `mdarray` object "owns" its data — similar to a `vector` — as opposed to existing as a reference to data "owned" by another container as in the case of `mdspan`. The earliest it would be released is also C++26.

NOTE

The code examples for `mdspan` above can be compiled in C++20 using the working code currently available on the P1673 GitHub site **{31}**. Installation and building instructions are included with the repository, but two particular items to know are first, the code is currently under the namespace `std::experimental`, and second, as the square bracket operator for multiple indices is set for C++23, you can replace it with the round bracket operator for C++20 and earlier; viz,

–

```
namespace stdex = std::experimental;
auto mds1 = stdex::mdspan{ v.data(), 3, 2 };
```

```
// Replace the square brackets here:
for (size_t i = 0; i < n_rows; ++i)
{
        for (size_t j = 0; j < n_cols; ++j)
                cout << mds1[i, j]                      <<
"\t";
}

// with round brackets:
for (size_t i = 0; i < n_rows; ++i)
{
        for (size_t j = 0; j < n_cols; ++j)
                cout << mds1(i, j) << "\t";
}
```

BLAS Interface (P1673)

This proposal is for “a C++ Standard Library dense linear algebra interface based on the dense Basic Linear Algebra Subroutines (BLAS)” {op cit 29}, also simply referred to as “stdBLAS”. BLAS libraries date back a number of decades and were originally written in Fortran, but it evolved into a standard in the early 2000’s, with implementations in other languages such as C (OpenBLAS) and CUDA C++ (NVIDIA) now available, as well as C bindings to Fortran.

Fortran BLAS distributions support four numerical types : `FLOAT`, `DOUBLE`, `COMPLEX`, and `DOUBLE COMPLEX`. The C++ equivalents are `float`, `double`, `std::complex<float>`, and `std::complex<double>`. BLAS libraries contain several matrix formats (standard, symmetric, upper/lower triangular), matrix and vector operations such as element-by-element addition and matrix/vector multiplication.

With implementation of this proposal, it would be possible to apply the same C++ code base to any compatible library containing BLAS functionality, provided an interface has been made available, presumably by the library vendor. This will allow for portable code, independent of the underlying library being used. It remains to be seen at this stage which vendors will

eventually come on board, but one major development is NVIDIA's implementation both of `mdspan` and stdBLAS, now available in their HPC SDK **{33}**.

It should be noted stdBLAS itself would only provide access to a particular subset of matrix operations — to be discussed next — even if the underlying library provides additional features such as matrix decompositions, linear and least squares solvers, etc.

BLAS functions are preceded by the type contained in the matrix and/or vector to which they are applied. For example, the function for multiplication of a matrix by a vector is of the form

```
xGEMV(.)
```

where the `x` can be `S`, `D`, `C`, or `Z`, meaning single precision (`REAL` in Fortran), double precision (`DOUBLE`), complex (`COMPLEX`), and double precision complex (`DOUBLE COMPLEX`) respectively.

The C++ equivalent in the proposal, `matrix_vector_product` would instead take in `mdspan` objects representing a matrix and a vector. For example, we can look at a case involving `double` values, using `m` and `n` for the number of rows and columns as before.

```
std::vector<double> A_vec(m * n);
std::vector<double> x_vec(n);

// A_vec and x_vec are then populated with data...

std::vector<double> y_vec(n);           // empty vector

std::mdspan A{ A_vec.data(), std::extents{m, n} };
std::mdspan x{ x_vec.data(), std::extents{n} };
std::mdspan y{ y_vec.data(), std::extents{m} };
```

Then, performing the multiplication, the vector product is stored in `y`:

```
std::linalg::matrix_vector_product(A, x, y);    // y = A *
```

x

The following table provides a subset of BLAS functions proposed in P1673 that should be useful in financial programming. The BLAS functions are assumed to be double precision, and any given matrix/vector expressions can be assumed to be of appropriate dimensions.

Table 5-1. Selected BLAS Functions in Proposal P1673

BLAS Function	P1673 Function	Description
DSCAL	`scale`	Scalar multiplication of a vector
DCOPY	`copy`	Copy a vector into another
DAXPY	`add`	Calculates $\alpha\mathbf{x} + \mathbf{y}$, vectors $\mathbf{x}$ & $\mathbf{y}$, scalar α
DDOT	`dot`	Dot product of two vectors
DNRM2	`vector_norm2`	Euclidean norm of a vector
DGEMV	`matrix_vector_product`	Calculates $\alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$, matrix $\mathbf{A}$, vector $\mathbf{y}$, scalars α & β
DSYMV	`symmetric_matrix_vector_product`	Same as DGEMV (`matrix_vector_product`) but where $\mathbf{A}$ is symmetric
DGEMM	`matrix_product`	Calculates $\alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C}$, for matrices $\mathbf{A}$, $\mathbf{B}$, & $\mathbf{C}$, and scalars α & β

Linear Algebra (P1385)

The authors of this proposal specifically recognized the importance of linear algebra in financial modeling, along with other applications such as medical imaging, machine learning, and high performance computing. {op cit 28}.

Initial technical requirements for a linear algebra library in C++ were outlined in a preceding proposal (P1166) **{34}**, directly quoted here (in italics):

The set of types and functions should be the minimal set required to perform functions in finite dimensional spaces. This includes:

- *A matrix template*
- *Binary operations for addition, subtraction and multiplication of matrices*
- *Binary operations for scalar multiplication and division of matrices*

Building on this, the P1385 proposal states two primary goals, namely that the library should be easy to use, with "run-time computational performance that is close to what {users} could obtain with an equivalent sequence of function calls to a more "traditional" linear algebra library, such as LAPACK, Blaze, Eigen, etc." {op cit 30}

At a high level, a matrix is generically represented by a *MathObj* type, and an *engine* "is an implementation type that manages the resources associated with a *MathObj* instance." Discussions are ongoing over the details, but "a *MathObj* might own the memory in which it stores its elements, or it might employ some non-owning view type, like `mdspan`, to manipulate elements owned by some other object". {ibid}. An *engine* object is proposed to be a private member on a *MathObj*, and a *MathObj* may have either fixed or dynamic dimensions.

More details should emerge over the next few years on the form the P1385 linear albegra library will assume, but for now, this hopefully provides an

initial high-level glimpse of something that should be a very welcome addition to C++ for financial software developers.

Summary (Linear Algebra Proposals)

Recalling the results we saw with `valarray`, with the above proposals in place, some of the same convenient functionality should be in place by C++26, this time with rigorous, efficient, and consistent specifications that should avoid the problems that plagued `valarray`. While `mdspan` differs from `valarray` as a non-owning reference, it will still allow array storage, such as a `vector`, to be adapted to a matrix proxy by specifiying the number of rows and columns. `submdspan` will assume a similar role as a `valarray` slice, but without the performance penalty due to object copy. P1673 will provide a common interface to libraries containing BLAS functions, and function naming, as as with `matrix_vector_product`, will be more expressive than their cryptic Fortran equivalents such as `DGEMV(.)`. And, the `+`, `-`, and `*` operators in P1385 will provide the ability to to implement linear algebra expressions in a natural mathematical format similar to the results we saw with `valarray`.

This is an exciting development that will finally provide efficient and reliable methods for implementing basic matrix calculations that are long overdue for C++. Hopefully we will also eventually see implementations of P1673 BLAS interfaces for popular open source libraries such as Eigen and others mentioned above, but as of the time of this writing, this remains to be seen.

Chapter Summary

This chapter has examined the past, present, and expected future of two-dimensional array management and linear algebra in C++. `valarray`, dating back to C++98, offered matrix-like capabilities for which quantitative developers certainly would have found ample use cases. It is unforunate it never received the attention and support of the Committee and community, even as C++ was being touted in the late 1990's as the language of the future for computational finance. For specific compilers, it might remain a viable option, but given the inconsistency of its implementations across Standard Library vendors, it can limit code reuse across different platforms.

In the late 2000's, high quality open-source linear algebra libraries such as Eigen and Armadillo came on the scene and were well-received by the financial quant C++ programming community. In the present day, these libraries contain not only much of the same functionality found in the BLAS standard, but also a plethora of matrix decompositions that are frequently used in financial applications.

Finally, the future also looks brighter for ISO C++ with the three proposals — `mdspan` (P0009), BLAS interface (P1673), and linear algebra library (P1385) — slated for inclusion in the Standard Library within the next three years. These are arguably features that are long overdue, but they will provide a big step in satisfying demand not just from the financial industry, but also other computationally intensive domains.

References

US Daily Treasury Par Yield Curve Rates
https://home.treasury.gov/resource-center/data-chart-center/interest-rates/TextView?type=daily_treasury_yield_curve&field_tdr_date_value_month=202211 _

Supplemental Chapter, _The C++ Standard Library, 2E

{2.5} Stroustrup, *A Tour of C++* (2E, but now in 3E)

Quantstart article on Eigen *https://www.quantstart.com/articles/Eigen-Library-for-Matrix-Algebra-in-C/*

Gottschling _Discovering Modern C++

Bowie Owens, CppCon 2019, *https://www.youtube.com/watch?v=4IUCBx5fIv0*

More information on the Jacobi and Bidiagonal Divide and Conquer SVD classes:

Two-sided Jacobi SVD decomposition of a rectangular matrix

Bidiagonal Divide and Conquer SVD

asciidoctor-latex -b html Ch10_v10_Split_LinearAlgebraOnly.adoc

{5} Rcpp Packages:

RcppEigen

RcppArmadillo

RcppBlaze3

Boost Headers, including uBLAS

About the Author

Daniel Hanson spent over 20 years in quantitative development in finance, primarily with C++ implementation of option pricing and portfolio risk models, trading systems, and library development. He now holds a full-time lecturer position in the Department of Applied Mathematics at the University of Washington, teaching quantitative development courses in the Computational Finance and Risk Management (CFRM) undergraduate and graduate programs. Among the classes he teaches is graduate-level sequence in C++ for quantitative finance, ranging from an introductory level through advanced. He also mentors Google Summer of Code student projects involving mathematical model implementations in C++ and R.