# Chapter 6. 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 6th 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

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 ex-

ternal 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 newer C++20 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 library 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 for financial applications 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 valuing a coupon-paying bond.

## Representation of a Date

C++11 originally introduced `std::chrono` 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 **{2}**

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 November 14, 2022, we would create it as follows. Note the `<chrono>` header needs to be included.

```
#include <chrono>

. . .

std::chrono::year_month_day ymd{ std::chrono::year{2022},
      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{2022},
      std::chrono::November, std::chrono::day{14} };
```

The `/` operator has also been overloaded to define a `year_month_day` object, with an example here using assignment:

```
ymd = std::chrono::year{ 2022 } / 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{ 2022 } / 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 / 2022;
```

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 library site **{1}** (ibid).

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

```
2022-11-14
```

## 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 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{ std::chrono::year{1970}, std::chrono::month{1}, std::chrono::day{1} };
std::chrono::year_month_day epoch_plus_1{ std::chrono::year{1970}, std::chrono::month{1}, std::chrono::day{2} };
std::chrono::year_month_day epoch_minus_1{ std::chrono::year{ 1969 }, std::chrono::month{ 12 },
      std::chrono::day{ 31 } };
```

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

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

first_days_test =
      std::chrono::sys_days(epoch_plus_1).time_since_epoch().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_epoch().count();     // -1
```

For typical financial trading, it is usually not necessary to go back before 1970, but in some areas, such as actuarial valuations of pension liabilities, many participants 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 2022-11-14, and initializing `ymd_later` to six months later -- 2023-5-14 -- take the difference between the two `sys_days` objects obtained with `sys_days(.)` and apply the `count()` function to the difference:

```
// ymd = 2022-11-14
// ymd_later = 2023-5-14

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

The result is 181 days.

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 these results are immediate with functions conveniently contained in `std::chrono`, but in other cases, there will be a little more work involved.

## 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
```

The number of years, months, or days between two dates `date1` and `day2`, however, can be easily obtained by applying the `count()` function to their differences. The result is an `int` type.

```
(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 integral types, 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;`

---

## 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{2022},
      date::month{11}, date::day{14} };

// torf: "true or false"
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.

## 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} };

torf = ymd_eom == eom_apr;            // Returns true
```

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 **{3}**.

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 library site **{4}**. 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:

```
#include <array>

. . .

unsigned last_day_of_the_month(const date::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 (and briefly discussed in Chapter 1). Because the length of the `array` and its contents are known a priori, `constexpr` instructs the compiler to initialize `normal_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. Live financial data, such as market and trade data, will be inevitably dynamic and thus unknown at compile time, so `constexpr` may be of more limited use in financial applications compared to other domains. The example above, however, demonstrates an example of how and when it can potentially be used effectively.

## 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 each 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 **{5}**.

## Adding Years, Months, and Days

One more set of important date operations in 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 requires a different approach.

### 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 ymd1{ date::year{2002}, date::month{11}, date::day{14} };
```

```
ymd1 += date::years{ 2 };          // ymd1 is now 2004-11-14
ymd1 += date::years{ 18 };         // ymd1 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. Even more end-of-month issues crop up when adding months. Workarounds are discussed next.

### 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 February during 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 ymd2{ date::year{2022}, date::month{2}, date::day{16} };
```

```
ymd += date::months(1);                 // Result: 2022-04-16
ymd += date::months(18);          // Result: 2023-10-16
```

Subtraction assignment is also available:

```
ymd -= date::months(2);                 // Result: 2023-08-16
```

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) };
   }
};
```

### 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 ymd3(date::year(2022), date::month(10), date::day(7));

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

Note that at this point, `ymd3` 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 `ymd3` date to three days later:

```
ymd3 = add_days; // Implicit conversion to year_month_day
                 // ymd3 is now = 2022-10-10
```

More information on adding days can be found in **{6}**.

## 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. It will also hold a data member representing the integral serial date, to be used in day count basis calculations necessary for fixed income investing.

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. 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.

```
ChronoDate(int year, unsigned month, unsigned day);
```

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

```
ChronoDate(const date::year_month_day& ymd);
```

And finally, the compiler-provided default constructor will be included, with in-class member initialization (introduced in Chapter 2) of the `std::chrono` date and serial date member data.

```
ChronoDate() = default;
```

## 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, we will see an application of implementing the C++20 `<=>` operator.

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:
bool end_of_month() const;
unsigned 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

In-class member initialization of the two private member variables is used in tandem with the compiler-provided default constructor. The `year_month_day` object ( `date_` ) is initialized as the UNIX epoch, and the corresponding serial representation of the date ( `serial_date_` ) as 0.

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_{ date::year(1970), date::month{1}, date::day{1} };
    int serial_date_{ 0 };

  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 the private helper function `reset_serial_date_()` . We also have the two user-defined constructor implementations, which is where we will start.

### Constructors

The implementation of the first user-defined 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::runtime_error 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` . So as not to detract from the primary topic here, we will just throw a `std::runtime_error` exception in the event of an invalid argument here and elsewhere in the class.

One more setting that needs to occur when a date is constructed is the serial date integer value. 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.

The second user-defined constructor is just a matter of initializing the the `date_` member with a copy of the single `year_month_day` argument. As `std::chrono` does not guarantee a valid date, we need to check it as well. If it checks out, then the serial date is determined and set:

```
#include <stdexcept>        // std::runtime_error

ChronoDate::ChronoDate(const date::year_month_day& ymd) : date_{ ymd }
{
    if (!date_.ok())            // std::chrono member function to check if valid date
    {
        throw std::runtime_error{ "ChronoDate constructor: Invalid year_month_day input." };
    }

    reset_serial_date_();
}
```

### Member Functions and Operators

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

#### Accessors

To start, implementation of accessors for the serial date and `year_month_day` members is trivial:

```
int ChronoDate::serial_date() const
{
    //    return date::sys_days(date_).time_since_epoch().count();
    return serial_date_;
```

```
}

std::chrono::year_month_day ChronoDate::ymd() const
{
    return date_;
}
```

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 low-level algorithms described in (*Leap Years and Last Day of the Month*).

```
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. 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, again by calling the `reset_serial_date_()` private member function.

```
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. Because the results modifies the underlying `date_` member, we again also need to reset `serial_date_`.

```
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);
	reset_serial_date_();

	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, and once again the serial date is also updated. Further details behind this are also available in **{5 op cit}**.

**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. What this means (in the assumed absence of holidays) is a date falling on a weekend will be moved forward to the next business day (Monday), unless the new date advances to the next month. In this case, the date is rolled back to the previous business day (Friday).

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 three days to the previous Friday of the original month, per the Modified Following rule. This is why the original month is stored first.

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

```
        date::month orig_mth{ date_.month() };

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

        if (orig_mth != date_.month())
        {
              date_ = date::sys_days(date_) - date::days(3);
        }

        reset_serial_date_(); // must be a valid date so no need to check

        return *this;
  }
```

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

**Comparison and Streaming Operators**

It is here we can see a real-world application of implementing the three-way `<=>` (spaceship) operator, introduced first in Chapter 2. First, as we have serial integral representation of a `ChronoDate`, implementation of the equality operator becomes trivial:

```
bool ChronoDate::operator == (const ChronoDate& rhs) const
{
        //return date_ == rhs.date_;
        return this->serial_date() == rhs.serial_date();
}
```

Then, all that needs to be done inside the spaceship operator implementation is to define the meaning of `<`, and the remaining four inequality operators are implied.

```
std::strong_ordering ChronoDate::operator <=> (const ChronoDate& rhs) const
{
	if (this->serial_date() < rhs.serial_date())
	{
		return std::strong_ordering::less;
	}
	if (this->serial_date() == rhs.serial_date())
	{
		return std::strong_ordering::equivalent;
	}
	else
	{
		return std::strong_ordering::greater;
	}
}
```

Gone are the days where we would need separate implementations of all six operators. It is so much easier now, with less to go wrong.

NOTE

The comparison operators `==` and `<=>` are actually already defined on the `year_month_day` class. So, technically speaking, each of these could have just been written as single line definitions in `ChronoDate`. We just need to be sure to use `std::strong_ordering` as the return type for `<=>`, as it is ultimately two integer values – the days since the epoch – that are being compared.

```
bool ChronoDate::operator == (const ChronoDate& rhs) const
{
	return this->serial_date() == rhs.serial_date();
}

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

The user-defined version shown previously was provided to demonstrate what otherwise would be a practical example of implementing the `<=>` operator.

---

For the stream operator, just to keep it simple, we can also piggyback off of the stream operator defined in `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;
}
```

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 years (or sometimes referred to as a *year fraction* for shorter intervals). 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 2022-10-25 and maturing on 2023-12-31. 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:

$t = \mathrm{Act360}(d_1, d_2) = \frac{d_2 - d_1}{360}$

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 counts 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 particular day count basis calculation in each derived class.

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

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

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

Declarations for Actual/365 and 30/360 day count bases, for example, can be written as follows:

```
class Act365 : public DayCount
{
public:
      double operator() (const ChronoDate& date1, const ChronoDate& date2) const override;
};

class Thirty360 : public DayCount
{
public:
      double operator() (const ChronoDate& date1, const ChronoDate& date2) const override;

private:
      unsigned date_diff_(const ChronoDate& date1, const ChronoDate& date2) const;
};
```

As more work is involved for calculating the 30/360 basis, a helper function `date_diff_(.)` is included, to be explained momentarily.

The Actual/365 calculation is trivial:

```
double Act365::operator()(const ChronoDate& date1, const ChronoDate& date2) const
{
    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 as shown here, and then divided by 360:

Let $D_1$ and $D_2$ be two dates, with year, month and day $y_i$, $m_i$, and $d_i$ respectively _ for $i = 1, 2$

The time in years (or year fraction) $t$ between $D_1$ and $D_2$ is then

$t = \frac{360 \times (y_2 - y_1) + 30 \times (m_2 - m_1) + (d_2 - d_1)}{360}$

End-of-month adjustments for the day values $d_1$ and $d_2$ in the numerator will depend on the particular form of the 30/360 basis, 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) **{7}** is commonly used and is implemented in the example below using the private `date_diff_` helper function. The result is divided by 360 in the public operator override.

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

```
unsigned Thirty360::date_diff_(const ChronoDate& date1, const ChronoDate& date2) const
{
	unsigned d1, d2;
	d1 = date1.day();
	d2 = date2.day();

	if (d1 == 31) d1 = 30;
	if ((d2 == 31) && (d1 == 30)) d2 = 30;
	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 next to each functor call. 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 **{8}** (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)                    // dc is polymorphic
{
      // 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, which is comprised of yields for various fixed rate products, all of which have a common *settlement date*. Also called the *settle date*, it is the date on which payments for the products are made. The derived yields extend over a series of increasing maturity dates, no two of which are the same.

## Deriving a Yield Curve from Market Data

A *yield* is essentially 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 2022-10-25 and receive 1035.60 dollars on 2023-12-31. In order to compare this investment with other investments we calculate its yield. Assuming compounded interest and an Actual/365 day count basis, then

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

from which we find the yield

$y = e^{\log(1035.60/1000)\times 365/432}-1 = 3\%$.

In general, the yield curve is a function of time, say $y(t)$, and is constructed from market data, such as Treasury Bill, swap, and bond data. The time value $t$ is in units of years.

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.

Consider buying a bond which makes a single payment of unit amount at maturity. Let the settlement date be $s$, the maturity be $m$, where $s \leq m$, and the price of the bond at $s$ be $P(s, m)$. If the yield is $y(t)$, where $t = \mathrm{Act365}(s, m)$ is the time to maturity in years, then

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

so that

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

Since the bond pays a unit amount at maturity, $P(s, m)$ is known as a *unit price*; it is also the *discount factor* for the interval from settlement date $s$ to maturity date $m$.

To illustrate how the inputs to the yield curve are derived, consider a six month 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 $F_1$, the maturity date is $d_1$, the market yield is $y_m$ for settlement on $s$. Then, the price, $B_1$, of the bill is

$B_1 = F_1(1 - \mathrm{Act360}(s, d_1)y_m)$

The unit price is $P(s, d_1) = B_1/F_1$, from which the continuously compounded Act365 yield can be calculated. _

Values of $y(t)$ for longer maturities can be obtained by bootstrapping. Suppose there is a US Treasury bond, maturing in one year and paying coupons every six months. Let the coupon payments be $C_1$ on date $d_1$ and $C_2$ on date $d_2$, the face value be $F$, and the market price on settlement date $s$ be $B_2$. Then,

$B_2 = C_1 P(s, d_1) + (F_2 + C_2)P(s, d_2)$

Since $P(s, d_1)$ is known, the value of $P(s, d_2)$ can be found. The bootstrapping process can be continued to find unit prices for longer maturities, and the associated yields can be found from

$y(t_i) = -\frac{log(P(s, d_i))}{t_i}$,

where $t_i = \mathrm{Act365}(s, d_i)$, for each $i$

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

$d_1< d_2 < \cdots < d_n$, with $ s < d_1$,

and the associated yields be

$y_1, y_2, \dots, y_n$, where $y_i = y(t_i)$ and $t_i = \mathrm{Act365}(s,d_i)$.

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

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

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

Now, given the fitted yield curve $y(t)$, constructed for settlement date $s$, the discount factor for any period from date $s$ to date $m$ is

$P(s, m) = e^{-t y(t)}$

where $t = \mathrm{Act365}(s, m)$. Since $y(t)$ is a spot yield, this is a spot discount factor. This can now be extended more generally for *forward discount factors*, as discussed next.

## 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 $P(s; d_1,d_2)$, based on market data *as seen at* the settle date $s$. The spot value of the unit payment is

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

To avoid arbitrage opportunities we must have

$P(s,d_2) = P(s,d_1)P(s;d_1,d_2)$, so that

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

Substituting for the spot discount factors:

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

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

The next sections will now describe a framework for yield curves in C++, employing the mathematical motivation just presented. The yield curve class that follows will ultimately be used in the valuation of a bond in the conclusion this chapter.

## 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. As noted previously, there are many ways to fit a curve through the points $(t_1,y_1), (t_2,y_2), \dots, (t_n,y_n)$, including linear interpolation, cubic splines, and methods designed specifcally for yield curves, such as Quartic Forward Splines **{9}** and Monotone Convex Splines **{10}**.

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

1. Provides a common method to calculate forward discount factors between two `ChronoDate` arguments, based on yields obtained from any given curve fitting method.
2. Requires a derived class to implement a particular curve-fitting method as a private member function that interpolates a yield for any time $t \geq s$.

Figure 6-1 shows a class diagram of the abstract base class, `YieldCurve`, with possible derived classes containing the curve-fitting methods referenced above.


*Figure 6-1. Yield Curve Class Hierarchy*

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—based on the curve-fitting method chosen—will provide an interpolated yield for a given time period relative to a specific settle date.

```
#include "ChronoDate.h"
#include "DayCounts.h"

class YieldCurve
{
public:
    // The yields are continuously compounded with day count basis act365:
    double discount_factor(const ChronoDate& d1, const ChronoDate& d2) const; // d1 <= d2 < infinity

    virtual ~YieldCurve() = default;

protected:
    ChronoDate settle_;
    inline static Act365 act365_{};

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

Both the settle date, `settle_`, and the `act365_` day count basis are protected, as they will need to be accessible from the derived class implementing the particular curve-fitting method. Furthermore, `act365_` is static, as the Act/365 basis will be the same for any yield curve instance. Starting in C++17, inlining static members in the declaration became sufficient; defining the variable in the implementation file is no longer required.

The implementation of `discount_factor(.)` then follows the mathematical derivation presented above (*Discount Factors* and *Forward Discount Factors*). Beyond the error conditions from which exceptions are thrown, there are three cases under normal conditions. First, if the two dates are identical, then the discount factor is 1. Second, if the first date `d1` is the settle date, the result defaults to the spot discount factor at date `d2`. Finally, third, is the general case of a forward discount factor over the period from `d1` to `d2`.

Note that the `discount_factor(.)` function doesn't care what the particular curve-fitting method is, as long as it can obtain an interpolated yield from the overridden `yield_curve_(.)` method. The `settle_date()` accessor is trivial but shown here as well for completeness.

```
double YieldCurve::discount_factor(const ChronoDate& d1, const ChronoDate& d2) const
{
	using std::exp;

	if (d2 < d1)
	{
		throw std::runtime_error("Invalid inequality: d2 < d1");
	}

	if (d1 < settle_ || d2 < settle_)
	{
		throw std::runtime_error("Dates must fall on or after settle date");
	}

	// Case 1:  Dates are identical => disc factor = 1
	if (d1 == d2)
	{
		return 1.0;		// exp(0.0)
	}
```

```
        // General formula: P(t1, t2) = exp( -(t2-t1) * f(t1, t2) )

        double t2 = act365_(settle_, d2);
        double y2 = yield_curve_(t2);

        // Case 2:  if d1 == settle_ then P(t1,t2) = P(0,t2) = exp(-t2 * y2 )
        if (d1 == settle_)
        {
                return exp(-t2 * y2);
        }

        double t1 = act365_(settle_, d1);
        double y1 = yield_curve_(t1);

        // Case 3:  settle_ < d1 < d2: (t2-t1) f(t1,t2) = t2 * y2 - t1 * y1
        return exp(t1 * y1 - t2 * y2);
}

double YieldCurve::discount_factor(const ChronoDate& d1, const ChronoDate& d2) const
{
        using std::exp;

        if (d2 < d1)
                throw std::runtime_error("YieldCurve::discount_factor: d2 < d1");

        if (d1 < settle_ || d2 < settle_)
                throw std::runtime_error("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 previously, 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.

```
class LinearInterpYieldCurve final : public YieldCurve
{
public:
        LinearInterpYieldCurve(
                const ChronoDate& settle_date,
                const std::vector<ChronoDate>& maturity_dates,
                const std::vector<double>& unit_prices);

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

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

The constructor will take in a set of maturity dates relative to the settle date, based on the Actual/365 day count basis, along with a set of corresponding unit prices, each as a `vector`. Its implementation will first check whether the yield and maturity date 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. In the concluding `for` loop, the Actual/365 continuous time equivalents (relative to the settle date) are computed and stored in the `maturities_` member vector, and the yields are extracted from the unit price inputs and also stored as a vector.

```
LinearInterpYieldCurve::LinearInterpYieldCurve(const ChronoDate& settle_date,
      const std::vector<ChronoDate>& maturity_dates, const std::vector<double>& unit_prices)
{
      using std::runtime_error;

      settle_ = settle_date;

      if (maturity_dates.size() != unit_prices.size())
            throw exception("LinearInterpYieldCurve: maturity_dates and spot_discount_factors different lengths");

      if (maturity_dates.front() < settle_)
            throw exception("LinearInterpYieldCurve: first maturity date before settle date");

      // Assume maturity dates in are in ascending order
      for (std::size_t i = 0; i < maturity_dates.size(); i++)
      {
            double t = act365_(settle_, maturity_dates[i]);
            maturities_.push_back(t);
            yields_.push_back(-std::log(unit_prices[i]) / t);
      }
}
```

The `maturities_` and `yields_` are then used in the linear interpolation method, implemented in the mandated `yield_curve_` private member function. If the time value at which a yield is to be interpolated exceeds its maximum data value, 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())
	{
		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 used in its valuation:

- Dated date
- First coupon date
- Penultimate coupon date
- Maturity date

The *dated date* is the date on which interest begins to accrue. It can be the same as, or different than, the *issue date*, the date on which the bond goes on sale. A stream of fixed payments is typically paid, for example 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*.

To make for more concise examples to come, the following assumptions will be made:

- The issue date and dated date are the same.
- Each coupon day is strictly less than 29, to avoid complications due to end of month adjustments.
- There are no holidays other than Saturdays and Sundays.

### 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? To show this, let us use a hypothetical bond with a maturity of two years at issue. The contractual dates are shown in the following table.

Table 6-1. Hypothetical Two-Year Bond Contractual Dates and Data

| | |
|---|---|
| **Issue Date** | 2022-09-22 |
| **Dated Date** | 2022-09-22 |
| **First Coupon Date** | 2023-03-22 |
| **Penultimate Coupon Date** | 2025-03-22 |
| **Maturity Date** | 2025-09-22 |
| **Coupon Rate** | 5% |
| **Face Value Amount** | $1,000.00 |

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 (irrespective of whether or not a business day at this point). 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. In our example, the bond has a first coupon date of 2023-3-22 and a penultimate coupon date of 2025-3-22. The two dates differ by 24 months, which is a multiple of 12/2=6, as required.

Since the due dates might not fall on business dates, the bond also has associated *payment dates* that are adjusted for weekends and holidays; however, to simplify matters we have assumed there are no holidays except Saturday and Sunday, so that if a due date falls on a weekend, the regular coupon payment will be made on the following Monday. In addition, because we avoid end of month cases by assuming payments will not occur on any day with value greater than 28, the date will not be rolled into the next month.

The intermediate due and payment dates are thus those shown in the following table. Note that the first two dates fall on business days, but as the the third falls on a Sunday, the payment date has been rolled to the following Monday. A $25 payment is made on each of the payment dates.

Table 6-2. Intermediate Coupon Due and Payment Dates

| Due Date | Payment Date |
|---|---|
| 2023-09-22 | 2023-09-22 |
| 2024-03-22 | 2024-03-22 |
| 2024-09-22 | 2024-09-23 |

If the first and final payments also occur regular periods, they will be \$25 and \$1025 as well, respectively, as is the case in this first example.

The first and last coupon periods may or may not be regular, however, and thus require adjustments that will also need to go into the code. The usually cited case is an irregular first payment period, but there can also be cases over of an irregular final payment period ending with the maturity date. To make things even more complicated, the adjustments depend on whether these irregular periods are shorter or longer than a regular period.

To demonstrate each case of an irregular first payment period, let us again use an example of a \$1000 face value bond paying an annual coupon of 5% semiannually (\$25 regular coupon payments), but for a 10-year bond. The contract data for the case of a short first payment period is as follows.

Table 6-3. Ten-Year Bond with Short First Payment Period Contractual Dates and Data

| | |
|---|---|
| **Issue Date** | 2022-07-12 (Tuesday) |
| **Dated Date** | 2022-07-12 (Tuesday) |
| **First Coupon Date** | 2022-12-21 (Wednesday) |
| **Penultimate Coupon Date** | 2032-12-21 (Tuesday) |
| **Maturity Date** | 2033-06-21 (Tuesday) |
| **Coupon Rate** | 5% |
| **Face Value Amount** | $1,000.00 |

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. If the first payment period had been regular, the dated date would have been 2022-6-21. This date is called the *first prior date*. A visual description is provided in Figure 6-2.

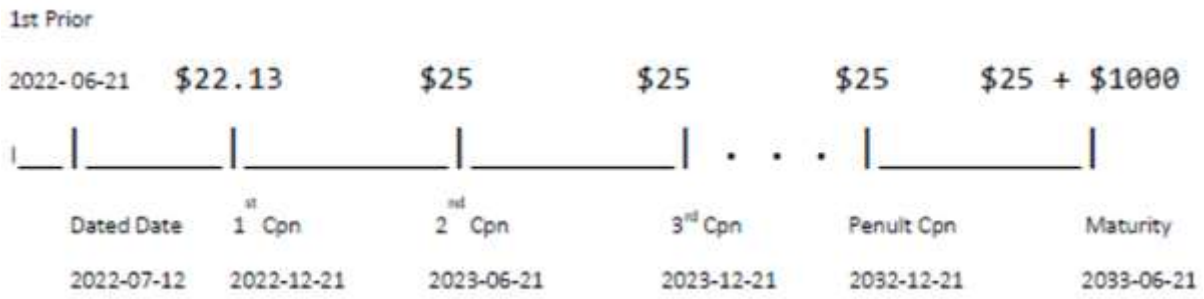


*Figure 6-2. Irregular short first coupon period*

The first payment is then prorated as follows:

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

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

In the case of a long first period, suppose the contract data is the same, except where the dated date is moved to an earlier date of 2022-5-12. A visualization of this example is shown in Figure 6-3.


*Figure 6-3. Irregular long first coupon period*

In this case, the first coupon payment will be the regular coupon of $25 over the period from the first prior date to the dated date, *plus* a partial payment (in red) over the interval between the dated date 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\; dated\; date\; to\; 1st\; prior)}{(number\; of\; days\; from\; 2nd \; prior\; to\; 1st\; prior)} = $ $25(\frac{40}{182}) = \$5.49$

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

Calculating adjusted payments over irregular final periods are similar, except that instead of prior payment periods preceding the dated date, extra payment periods extending later than maturity are utilized.

While this discussion might seem to be a bit tedious, it is a fair approximation of real-world design requirements for financial programmers responsible for fixed income trading operations and risk management. Date calculations are also very common in applications related to defined benefit pension plans and variable annuities. The new C++20 date library addition makes for much more rapid and easier development than before.

## Valuing a Bond

The issuer sells bonds on the issue date, pays the owners of the bonds the coupon amounts starting from the dated date 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.

A bond can be valued as of an arbitrary settle date, falling on or after the yield curve settlement date. This can be done by calculating the discounted value of the bond, using a yield curve of one's choice, and the discount factors calculated off of this curve for the periods from the bond settlement date to the corresponding payment dates.


*Figure 6-4. Bond valuation at settle date*

Revisiting the short first period example in Figure 6-1, suppose a bond is exchanged for cash on 2023-10-24, as indicated by the red hash mark in Figure 6-3 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.

Under the common textbook assumption, the bond and yield curve settle dates are the same. Using the discount factor notation above, the value of the bond on this date will be

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

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.

Discounted valuations are also routine calculations in trading and risk management software. A common task for programmers is to write code to value a bond portfolio using an updated or simulated yield curve in order to calculate P/L (based on current market conditions) and run risk calculations (based on shocked or random scenarios).

## Designing 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)
- Dated 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. Note that we will be using all of the user-defined classes now complete that were developed in the preceding sections: date class, day count classes, and yield curve classes.

```
#include "ChronoDate.h"
#include "YieldCurve.h"
#include "DayCounts.h"
#include <cmath>
#include <vector>
#include <string>

class Bond
{
public:
	Bond(std::string bond_id, const ChronoDate& dated_date, const ChronoDate&
		first_coupon_date, const ChronoDate& penultimate_coupon_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);

	std::string bond_id() const;

private:
	std::string bond_id_;

	std::vector<ChronoDate> due_dates_;		// Dates on which payments are due,
```

```
                                                    // whether business days or not.
        std::vector<ChronoDate> payment_dates_;     // Business dates on which payments are made.
        std::vector<double> payment_amounts_;    // Coupon and redemption payments:
                                                    // assume redemption_payment = face_value.

        void generate_regular_dates_and_pmts_(const ChronoDate& first_coupon_date,
            const ChronoDate& penultimate_coupon_date, int months_in_regular_coupon_period,
            double regular_coupon_payment);

        void amend_initial_irregular_dates_and_pmts_(const ChronoDate& dated_date,
            const ChronoDate& first_coupon_date, const int months_in_regular_coupon_period,
            const double regular_coupon_payment);

        void amend_final_irregular_dates_and_pmts_(const ChronoDate& penultimate_coupon_date,
            const ChronoDate& maturity_date, const int months_in_regular_coupon_period,
            const double regular_coupon_payment, double face_value);
    };
```

Note that all the contractual information is captured in the constructor, but none of these arguments ends up being stored as a data member. Rather, they are used when a `Bond` object is created—as will be seen in the implementations—inside the constructor itself or in the private helper functions that generate the vectors of dates and payments that do get stored as members.

The 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 previous discussion {Bond Payments and Valuation}, 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 multiple (possibly thousands of) times under different yield curve scenarios for risk reporting purposes, as noted previously.

There are three member vectors of equal length, `due_dates_`, `payment_dates_`, and `payment_amounts_`, corresponding to the respective descriptions in {Determining the Payment Schedule} above. All three are necessary for calculating the discounted value of a bond.

A bond ID field of some form 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 construction and the process of setting the vector members in the class implementation step by step. This setup needs to take place first before valuing a bond. The constructor will generate the due and payment dates, and the payment amounts. Recall 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.

```
#include "Bond.h"
#include <iterator>

Bond::Bond(std::string bond_id, const ChronoDate& dated_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;

        generate_regular_dates_and_pmts_(first_coupon_date, penultimate_coupon_date,
                months_in_regular_coupon_period, regular_coupon_payment);

        amend_initial_irregular_dates_and_pmts_(dated_date, first_coupon_date,
                months_in_regular_coupon_period, regular_coupon_payment);

        amend_final_irregular_dates_and_pmts_(penultimate_coupon_date, maturity_date,
                months_in_regular_coupon_period, regular_coupon_payment, face_value);

        // (7) Maturity date is a due date which falls on a business day
        due_dates_.push_back(maturity_date);
        payment_dates_.push_back(maturity_date);
}

void Bond::generate_regular_dates_and_pmts_(const ChronoDate& first_coupon_date,
        const ChronoDate& penultimate_coupon_date, int months_in_regular_coupon_period,
        double regular_coupon_payment)
{
        // (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);
        }
    }

    void Bond::amend_initial_irregular_dates_and_pmts_(const ChronoDate& dated_date,
        const ChronoDate& first_coupon_date,
        const int months_in_regular_coupon_period, const double regular_coupon_payment)
    {
        // (5) If first coupon is irregular, amend the coupon payment
        ChronoDate first_prior{ first_coupon_date };
        first_prior.add_months(-months_in_regular_coupon_period);
        if (first_prior != dated_date) // if true then irregular coupon
        {
            if (first_prior < dated_date) // if true then short coupon period
            {
                double coupon_fraction =
                    static_cast<double>(first_coupon_date - dated_date) /
                    static_cast<double>(first_coupon_date - first_prior);
                payment_amounts_[0] *= coupon_fraction;
            }
            else // dated_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 - dated_date) /
                    static_cast<double>(first_prior - second_prior);
                payment_amounts_[0] += coupon_fraction * regular_coupon_payment;
            }
        }
```

```
}

void Bond::amend_final_irregular_dates_and_pmts_(const ChronoDate& penultimate_coupon_date,
      const ChronoDate& maturity_date, const int months_in_regular_coupon_period,
      const double regular_coupon_payment, double face_value)
{
      // (6)  If final coupon period is irregular, amend the coupon payment.
      ChronoDate maturity_regular_date{ penultimate_coupon_date };
      maturity_regular_date.add_months(months_in_regular_coupon_period);
      double final_coupon{ regular_coupon_payment };

      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_period);
                  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 {Bond Payments and Valuation}, `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 private member function `generate_regular_dates_and_pmts_(.)` will generate the due and payment dates, and set the payment amounts, over each period between the first coupon payment date and the penultimate coupon date.

As each period between these two dates is regular, each date generated and appended to the `due_dates_` vector will be be spaced apart equally—eg every six months -- up to and including the contractual penultimate coupon date. Because the `+=` operator for months on the incremented `year_month_day` object guarantees the same day value, this will satisfy the requirement for the due dates. Bonds have many variations, but since this is not production code it is simplified by assuming the coupon day is less than 29 in order to avoid end of month calculations.

Note that we can encapsulate the `regular_due_date` variable inside the body of the `for` loop, rather than defining it externally first, by incrementing each successive due date as an argument for the iteration. Like a range-based `for` loop, this also obviates using index values.

```
for (ChronoDate regular_due_date{ first_coupon_date };
      regular_due_date <= penultimate_coupon_date;
      regular_due_date.add_months(months_in_regular_coupon_period))
{
      . . .
}
```

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. Finally, the regular coupon payment amount—already determined inside the body of the constructor—is appended as a constant value to the `payment_amounts_` vector for each regular date.

In the previous step, the first coupon payment was naively set to the regular amount. We therefore need to call the `amend_initial_irregular_dates_and_pmts_(.)` member function (5) to verify whether an adjustment is required for the first payment over a long or short period, as discussed in (Figures 6-1 and 6-2). Similarly, this needs to be performed for irregular final payments in the `amend_final_irregular_dates_and_pmts_(.)` member function (6).

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 in `payment_amounts_` is set to the regular coupon amount plus the face value of the bond at this point (7). 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 as with an irregular first period are performed, but using prospective rather than retrospective extensions (8).

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

As noted previously {Valuing a Bond}, 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 yield_curve.discount_factor(yield_curve.settle_date(), bond_settle_date) * pv;
}
```

As you may notice, this valuation function is short and compact, as the due dates, payment dates, and payment amounts were already determined and set by the constructor and the private helper functions at the outset. This included any adjustment to the first or last payment in the event of an irregular short or long payment period.

So now, the valuation code can start by looping 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 `discount_factor(.)` member function on the `yield_curve` input object.

These discounted payment values are summed and assigned to the `pv` variable. As the curve-fitting method is determined at runtime (we are not necessarily limited to linear interpolation), the `discounted_value()` function doesn't "need to care" about how the discount factors or date calculations are obtained. The result is the discounted value of the bond as of the bond settlement date.

## A Bond Valuation Example

We can now put the individual classes previously presented together into an example of pricing a bond. Recall that the `YieldCurve` abstract base class will require 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 the linear interpolation version we have already. In constructing a `Bond` object, we will need to supply the face value, dated date, first coupon date, penultimate payment date, and maturity of the bond, along with its face value. A UML diagram of this assembly showing the relationships between the required objects for this example is shown in Figure 6-5.


*Figure 6-5. Bond Valuation Object Assembly*

As an example, suppose the term sheet of a 20-year bond is as follows:

Table 6-4. Contractual bond terms - example

| | |
|---|---|
| **Face Value** | $1000 |
| **Annual Coupon Rate** | 6.2% |
| **Payment Frequency** | Every six months (semiannual) |
| **Dated Date** | 2023-05-08 (Mon) |
| **First Coupon Date** | 2023-11-07 (Tue) |
| **Penultimate Coupon Date** | 2042-11-07 (Fri) |
| **Maturity Date** | 2043-05-07 (Thu) |

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";                    // 20 year bond
```

```
ChronoDate dated_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{ 2043, 5, 7 };              // Regular last coupon period (Thu)

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, dated_date, first_coupon_date, penultimate_coupon_date,
            maturity_date, coupon_frequency, coupon_rate, face_value };
```

Recall, however, the due dates, payment dates, and payment amounts are all generated and adjusted at the time a `Bond` object is constructed. Each due date will carry a day value of 7, and the payment dates will be the same except for the due dates falling on weekends that are rolled to the following 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

$31(\frac{183}{184}) = \$30.83$

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 the dated date and first coupon date, say Tuesday, October 10, 2023. Suppose also market data as of this date imply the unit prices:

Table 6-5. Spot yields - example (values rounded to six decimal places)

| Period | Maturity | Unit Price |
|---|---|---|
| Overnight | 2023-10-11 | 0.999945 |
| 3 Month | 2024-01-10 | 0.994489 |
| 6 Month | 2024-04-10 | 0.988210 |
| 1 Year | 2024-10-10 | 0.973601 |
| 2 Year | 2025-10-10 | 0.939372 |
| 3 Year | 2026-10-12 | 0.901885 |
| 5 Year | 2028-10-10 | 0.827719 |
| 7 Year | 2030-10-10 | 0.759504 |
| 10 Year | 2033-10-10 | 0.670094 |
| 15 Year | 2038-10-11 | 0.547598 |
| 20 Year | 2043-10-12 | 0.448541 |
| 30 Year | 2053-10-10 | 0.300886 |

Create two vectors containing the times to maturity (in units of years) and discount bond prices above (again, in place of containers that would normally be initialized in an interface):

```
std::vector<double> unit_bond_maturity_dates{ {2023, 10, 11}, {2024, 1, 10},
      . . ., {2053, 10, 10} };

std::vector<double> unit_bond_prices{0.999945, 0.994489, . . ., 0.300886 };
```

And also the settlement date:

```
ChronoDate yc_settle_date{ 2023, 10, 10 };
```

With these, we can create an instance of a linearly interpolated yield curve:

```
LinearInterpYieldCurve yc{ yc_settle_date, unit_bond_maturity_dates, unit_bond_prices };
```

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(yc_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 $1320.38.

## Summary

Date calculations are fundamental to fixed income trading, so having a date class in the Standard should be a very welcome addition in C++20. With the two topics being inexorably linked, this chapter was a long one; however, these are topics that surprisingly are rarely covered in much practical detail either on the C++ programming side or finance side in most textbooks and quantitative finance curricula, so hopefully this information will prove useful.

The core date class in `std::chrono` is the `year_month_day` class. Individual year, month, and day components of a `year_month_day` object are accessible and can be cast to integral types. An integer serial date equal to the number of days since the UNIX epoch of 1970-01-01 can also obtained, and hence it can be used to find the number of days between two dates, which is often required in valuing fixed income products and analytics.

Rather than throwing an exception that can hinder performance, it is possible to set a `year_month_day` object to an invalid date, so it is up to the programmer to determine whether a date is valid using its `ok()` member function. The `is_leap()` member function is also available to check whether a year is a leap year.

Adding months or years to a `year_month_day` object is made possible with its `+=` operator defined; however, adding days requires finding a date's `sys_days` (an alias for `std::chrono::time_point`) equivalent first. Similarly, workarounds are required for determining whether a date is on a weekend, and whether it lands on the end of a month, two more important properties to have when working with date-related calculations in finance.

Because of workarounds and other functionality left to the programmer, it can be easier to wrap everything in a class, encapsulating these extra steps while providing a user-friendly interface. One such rendition was presented in the user-defined `ChronoDate` class. It included public member functions to add years, months, and days as integer values, to calculate the number of days between two dates, to obtain

the serial date, and to return the number of days in a month, or whether a date fell during a leap year. An implementation of the C++20 `<=>` operator, originally introduced in Chapter 1, was also demonstrated, plus a function was defined that rolls a weekend date to the next business day.

Day count bases are also essential in fixed income trading, and with the `ChronoDate` class providing the addition and subtraction methods, we implemented them with very little difficulty as derived classes from an abstract base class. It is important to use the correct day count basis designated in the contractual data of a bond, swap, or fixed income derivative when calculating their valuations.

A yield curve in general is a function of time, $y(t)$, where $t$ is the time from the settle date in units of years and dependent upon a day count basis, although in practice the Actual/365 day count is often used as a uniform method to convert the dates contained in yield curve data to continuous time. As yield curve data is discrete, a curve fitting method needs to be applied. In our example, we defined an abstract base class with a common yield calculation, but where the particular curve fitting method is implemented as a protected member function overriding the pure virtual `yield_curve_(.)` method. Curve fitting methods can vary in sophistication, but as an example we used linear interpolation. The class design, however, allows one to plug in whichever method is desired.

Date, day count basis, and yield curve objects all come together in order to value fixed income securities. In particular, a bond is a series of payments on specific and mostly evenly spaced dates, usually every six months. Most of the payments are a single constant value, but there can be some irregular periods that require the use of the contractual day count basis in order to calculate the proportional coupon value, plus payment dates may need to be rolled forward from weekends, and the final payment also includes repayment of the face amount.

These payments are discounted back from the payment date to the settle date, using the discount factor calculated off of the yield curve. Using the traditional object-oriented features of C++, in conjunction with the new `year_month_day` date class, a clear separation of duties ensures better code maintainability, lower probability of errors, and the ability to reuse these objects when working with different securi-

ties, be they bonds, swaps, or mortgages, or derivatives such as mortgage-backed securities, swaptions, or bond options.

## References


{1} `std::chrono` date GitHub repository: *https://github.com/HowardHinnant/date*

{2} Nicolai Josuttis, *The C++ Standard Library 2E*, Addison Wesley, Sec 5.7.1, pp 143-44 (O'Reilly Learning: *https://learning.oreilly.com/library/view/c-standard-library/9780132978286/*)

{3} *https://stackoverflow.com/questions/59418514/using-c20-chrono-how-to-compute-various-facts-about-a-date* [Howard Hinnant, Stack Overflow (Fact 5)]

{4} `chrono`-Compatible Low-Level Date Algorithms *https://howardhinnant.github.io/date_algorithms.html*

{5} 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*

{6} 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*

{7} ISDA 30/360 Day Count Basis *https://www.iso20022.org/15022/uhb/mt565-16-field-22f.htm*

{8} Steiner pp 40-41

{9} Kenneth J Adams, Smooth interpolation of zero curves, Algo Research Quarterly, 4(1/2):11-22, 2001

{10} Hagan and West, Interpolation Methods for Curve Construction, Applied Mathematical Finance, Vol. 13, No. 2. 89-129, June 2006

Not cited directly but used as a reference: Stigum & Robinson, *Money Market & Bond Calculations*, Irwin, 1996