1 Introduction

Dates and date calculations are hardly the sexiest topic in quantitative finance, but they are vitally important, particularly in the areas of fixed income analytics and related derivatives pricing. A prime example is in applying the correct day-count convention to pricing assets such as bonds and interest rate derivatives: failure to do so can be sniffed out by competing trading operations and thus turn your error an arbitrage opportunity against you, resulting in potentially significant losses.

Up until C++20, C++ quant developers in finance had to rely on an external library for dates, or write customized code. This has now fortunately changed with the addition of the `std::chrono` date classes. The following sections present how to perform typical date calculations in trading areas such as fixed income and mortgage-backed securities. It concludes with a small collection of classes that can then be applied to bond pricing and other interest-bearing securities.

2 Representation of a Date

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

First, the year, month, and day can be ultimately indicated by their respective integer values, but the constructor arguments require these to be represented by `std::chrono::year`, `std::chrono::month`, and `std::chrono::day` objects. For example, to create an object holding the date November 14, 2002, we would create it as follows.

```
import <chrono>;

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

Alternatively, the individual months can be represented as their own types, in written-out form, so an equivalent approach is to replace the month as shown here:

```
std::chrono::year_month_day ymd2{ std::chrono::year(2002),+
    std::chrono::November, std::chrono::day(14) };
```

And finally, the `/` operator has been overloaded to define a `year_month_day` object:

```
std::chrono::year(2002)
    / std::chrono::November / std::chrono::day(14)
```

With dates in this form, the copy constructor and assignment operator can be employed to create new `year_month_day` objects:

```
std::chrono::year_month_day ymd4{ std::chrono::year(2002)
        / std::chrono::November / std::chrono::day(14) };
```

2.1 Integer Representation and Date Differences

A `year_month_day` date can be measured in terms of the number of days since an epoch, the default being the UNIX epoch January 1, 1970. Dates before the epoch are also valid but carry a negative integer value.

To access these serial integer values, we will need to first convert the date to a `std::chrono::timepoint` object using the `std::chrono::sys_days` operator, and then call its `time_since_epoch` member function. To get the number of days, invoke the `count` function on the result, as follows:

```
auto days_since_epoch =
    std::chrono::sys_days(ymd).time_since_epoch().count();
```

For `ymd` and its equivalents, this value is 12,005, and the return type is `int`. Now, to find the number of days between `ymd` and six months later – May 14, 2003 – this can be done by taking the difference between the timepoints since epoch and applying the `count` function again:

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

Note: Putting the `std::chrono` scope before every related type obviously becomes a bit unwieldy, so going forward, we will simply use the namespace alias

```
namespace date = std::chrono;
```

2.2 Validity of a Date

It is possible to set `year_month_day` objects to invalid and even completely bogus dates. For example, it is possible a date of January 31 could be rolled forward a month to February 31. In addition, the constructor will also allow a negative year with month and day values out of range. A date can be verified with the `ok` member function that returns `true` or `false`. In the following example, the `ymd` date (same as above) is valid, while the two that follow are obviously not.

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

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

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

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

Now, we can verify whether these dates are valid or not:

```
bool torf = ymd.ok();              // true
torf = ymd_invalid.ok();           // false
torf = ymd_completely_bogus.ok()   // false
```

The `ok` member function will come in handy in subsequent examples

2.3 Leap Years

A date can be checked easily as well whether it is in 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
```

2.4 End of the Month

There unfortunately is no member function available to check whether a date is at the end of its month, although there is a separate class that represents an end-of-month date, named `year_month_day_last`. Using the example of the last day of April 2009, its `year_month_day` version is constructed as before:

```
date::year_month_day ymd_eom{ date::year{2009},
    date::month{4}, date::day{30} };
```

We can then create a `year_month_day_last` object for the same month:

```
date:: year_month_day_last
    eom_check{ ymd_eom.year() / ymd_eom.month() / date::last};
```

And then, compare the two:

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

It should also be noted a `year_month_day_last` type is implicitly convertible to a `year_month_day`. In particular, reassigning the `year_month_day` object `ymd_eom` to `eom_check` is valid:

```
ymd_eom = eom_check;
```

This is another useful result to have in subsequent applications.

2.5 Weekends

Similar to end of the month dates, there is no member function to check whether a date falls on a weekend. There is, however, a way to determine the day of the week, from which we can derive the result we need. Furthermore, this provides a convenient application of the `std::optional` wrapper type.

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

```
// Define a weekday (Wednesday) and a weekend year_month_day

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

date::year_month_day ymd_weekend{ date::year{2020},
    date::month{10}, date::day{29} };    // Saturday

// Create `weekday` (day of the week) objects:
date::weekday biz_weekday{ date::sys_days(ymd_biz_day) };
date::weekday week_end{ date::sys_days(ymd_weekend) };
```

Now, these can be compared with days of the week defined in `std::chrono`.

```
bool is_wednesday = biz_weekday == date::Wednesday;     // true
bool is_saturday = week_end == date::Saturday;          // true
```

The stream operator is also defined:

```
if(is_wednesday)
    cout << biz_weekday << endl;
if(is_saturday)
    cout << week_end << endl;
```

The output to the screen will be abbreviated days of the week:

```
Wed
Sat
```

2.6 Adding Years, Months, and Days to Dates

One more set of important date operations of finance is adding years, months, and days to existing dates. These are particularly useful for generating schedules of payments, and adjusting for weekends and end of month. Adding years or months each rely on the `+=` operator, but adding days requires a different approach.

2.6.1 Adding Years and Months

Apart from a few edge cases (to be discussed later), adding years or months is fairly straightforward. Both `year` and `month` types can added to a `year_month_day` object using addition assignment. Examples of adding years are shown here:

```
// Start with 2002/11/14
date::year_month_day ymd1{date::year{2002}, date::month{11}, date::day{14}};

ymd1 += date::years{ 2 };
cout << "ymd1 + 2 years = " << ymd1 << endl;

ymd1 += date::years{ 18 };
cout << "ymd1 + 18 more years = " << ymd1 << endl << endl;
```

The cumulative results are then displayed as

```
ymd1 + 2 years = 2004-11-14

ymd1 + 18 more years = 2022-11-14
```

For months, it is similar:

```
// Start with 2022/02/16
date::year_month_day ymd2{date::year{2022}, date::month{2}, date::day{16}};

ymd2 += date::months{ 2 };
cout << "ymd2 + 2 months = " << ymd2 << endl;

ymd2 += date::months{ 18 };
cout << "ymd2 + 18 more months = " << ymd2 << endl << endl;
```

The results are then as expected:

```
ymd2 + 2 months = 2022-04-16

ymd2 + 18 more months = 2023-10-16
```

More interesting is what happens if we start with a date at the end of the month. In the case of a payment or cash flow schedule, adding months to an end of the month should result in a date also at the end of the month. This is where the `year_month_day_last` type comes in handy. Suppose our start date is April 30, 2022. This can be instantiated as

```
date::year_month_day_last
    ymd_last{date::year{2022} / date::month{4} / date::last};
```

Now, add eight months:

```
ymd_last += date::months{ 8 };
cout << "ymd_last + 8 months = " << ymd_last << endl;
```

Note that in this case, the date is displayed as a last day of the month, rather than in yyyy/mm/dd format.

```
ymd_last + 8 months = 2022/Dec/last
```

But because this type is implicitly convertible to a `year_month_day` type – as we saw in Section 1.4 – we can recover the usual format:

```
// Implicit conversion back to year_month_day:
date::year_month_day ymd_eom = ymd_last;
cout << "ymd_eom = " << ymd_eom << endl;
```

```
ymd_eom = 2022-12-31
```

This also means we can handle February gracefully.

```
ymd_last += date::months{ 2 };    // Advance to February
ymd_eom = ymd_last;
cout << "ymd_eom = " << ymd_eom << endl;
```

The result is the last day of February, as desired:

```
ymd_eom = 2023-02-28
```

2.6.2 Adding Days

To add days to a `year_month_day` date, we need to obtain the equivalent `time_point` -- using the `sys_days` operator – before adding the number of days.

```
// Start with 2022.10.07
date::year_month_day ymd3{ date::year(2022), date::month(10), date::day(7) };

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

Note that at this point, `ymd3` has not been modified, and the result, `add_days`, is also a `time_point` type. A `time_point` is also implicitly convertible back to a `year_month_day` type, so we can assign `add_days` to `ymd3` if desired:

```
ymd3 = add_days;   // Implicit conversion to `year_month_day`
cout << "ymd3 now = " << ymd3 << endl;          // ymd3 now = 2022/10/10
```

3 A Date Wrapper Class

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.

Some of these requirements have already been covered in the non-member sense above. These can be divided into two categories, namely checking possible states of a date, and performing arithmetic operations on dates. A summary of what we have covered follows in the list below. Note that the various states of a date are not necessarily mutually exclusive, and also that arithmetic operations may depend on a date object's state.

- State
  - Valid date
  - Leap year
  - End of the month
  - Weekend
- Arithmetic Operations
  - Number of days between two dates
  - Addition
    - *Years*
    - *Months*
    - *Days*

Additional functionality that we will want to have is listed next, some of which will become more apparent as the class implementation is written.

- Accessors
  - Year, Month, Day
  - Integer representation (days since epoch)
  - `year_month_day` data member
- Increment/Decrement operators
  - `++` (pre-increment)
  - `--` (pre-decrement)

- Comparison operators
  - `==`
  - `<=>`

Next, we can consolidate all of the above requirements into a class declaration.

3.1 Class Declaration

Declaring a class called `ChronoDate`, the requirements above will be included. Only one private data member will be necessary, namely a `year_month_day` object.

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

3.1.1 Class 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. This latter task will be handled by the constructor implementation.

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

Note that the argument for the year is an `int`, while those for the month and day are `unsigned`. This is due to the design of the `year_month_day` class.

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

```
ChronoDate(date::year_month_day);
```

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

3.1.2 Public Member Functions and Operators

These are self-explanatory, with descriptions provided in the requirements above.

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

// Modifying methods and operators:
ChronoDate& add_years(int rhs_years);
ChronoDate& add_months(int rhs_months);
ChronoDate& add_days(int rhs_days);

unsigned operator - (const ChronoDate& rhs) const;

ChronoDate& operator ++ ();
ChronoDate& operator -- ();

// Comparison operators
bool operator == (const ChronoDate& rhs) const;
std::strong_ordering operator <=> (const ChronoDate& rhs) const;

// Check state:
bool end_of_month() const;
int days_in_month() const;
bool leap_year() const;
std::optional<date::weekday> weekend() const;
```

3.2 Class Implementation

3.2.1 Constructors

The implementation of the first declared constructor allows one to create an instance of `ChronoDate` without the manual overhead of first creating instances of `std::chrono` objects `year`, `month`, and `day`. This is all taken care of by the constructor:

```
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
    {
         std::exception e("ChronoDate constructor: Invalid date.");
         throw e;
    }
}
```

Recall also that it is possible to construct invalid `year_month_day` objects, such as February 30, so a validation check is also included in the constructor, utilizing the `ok` member function on `year_month_day`.

For the constructor taking in a `year_month_day`:

```
ChronoDate::ChronoDate(const Date& ymd):date_{ymd}  {}
```

And finally, the default constructor sets the date to the UNIX epoch:

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

3.2.2 State Methods

These methods formally wrap code similar to the examples at the outset. Although they are public and are standard fare for date objects, they will also be used in date arithmetic methods.

```
bool ChronoDate::end_of_month() const
{
    return date_ == date_.year() / date_.month() / date::last;
}

int ChronoDate::days_in_month() const
{
    auto num_days_in_mth =
        date::sys_days{ date_.year() / date_.month() / date::last }
        - date::sys_days{ date_.year() / date_.month() / 1 } + date::days{ 1 };

    return num_days_in_mth.count();
}

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

When checking for weekends, we can again use `std::optional`, returning a `weekday` object if it is a Saturday or a Sunday, and null otherwise. The rationale for this will become apparent in financial applications.

```
std::optional<date::weekday> ChronoDate::weekend() const
{
    date::weekday day_of_week{ date::sys_days(date_) };
    if (!(day_of_week == date::Saturday || day_of_week == date::Sunday))
    {
         return std::nullopt;
    }
    else
    {
         return day_of_week;            // Saturday or Sunday
```

```
    }
}
```

Remember that `weekday` in `std::chrono` means “day of the week”, including weekend days.

3.2.3 Arithmetic Operators

These are the core member functions that will be used for typical fixed income applications, such as applying day count conventions, generating cash flow schedules, and constructing interest rate term structures.

To start, let us revisit calculation of the number of days between two dates but formally incorporate the code into the subtraction operator:

```
unsigned ChronoDate::operator - (const ChronoDate& rhs) const
{
    return (date::sys_days(date_).time_since_epoch()
         - date::sys_days(rhs.date_).time_since_epoch()).count();
}
```

Adding years and months to a date can become problematic when accounting for leap years and, in the case of month addition, end-of-month adjustments. First, our implementation of `add_years` could be written as shown here.

```
ChronoDate& ChronoDate::add_years(int rhs_years)
{
    bool prev_is_eom = end_of_month();
    date_ += date::years(rhs_years);

    if (month() == 2 && prev_is_eom)
    {
         // Implicit conversion to year_month_day type:
         date_ = date_.year() / date_.month() / date::last;
    }

    return *this;
}
```

A problem will arise if the original date (`*this`) falls on the last day of a February. This state is saved in the temporary variable `prev_is_eom` just before the number of years are added. The resulting date in this case should also be the last day of February, whether or not it is a leap year. Redefining the date as a `year_month_day_last` object takes care of this for us, irrespective of leap year status. Then, the object is implicitly convertible back to a `year_month_day` object that is used to reset the `date_` data member.

When adding months to a date, the situation becomes even more problematic, as we need to deal with the following edge cases:

- An end-of-month date (eg November 30) is rolled forward a number of months to the following February. We need to ensure the result is either February 28 or 29.
- The starting date is the end of February (eg the 28th). If it is rolled forward, say by three months, we need to ensure the result is May 31, and not May 28.
- The original date is the 30th or 29th of a 31-day month (eg October 30) and is rolled forward to the following February. We need to ensure again the result is either February 28 or 29.

As a result, the code will become somewhat convoluted, but one way to solve it is as follows.

- Beginning with the case where the starting date is not end-of-month, we can first check whether it falls on a 29th or 30th.
    - If not, then just increment the current date by the desired number of months, say \n, and be done with it.
    - If it does fall on one of these two days, then roll forward to the end of the \nth month. If this new month is February, set it to the end of the month. If not, just use the current date incremented by \n months.

- If the starting date is the end of the month, roll forward to the end of the \nth month, and then assign the result to `date_`, which will again implicitly convert the `year_month_day_last` type back to `year_month_day`.

A possible implementation then follows.

```
ChronoDate& ChronoDate::add_months(int rhs_months)
{
    if (!end_of_month())
    {
         if (!(this -> day() == 30 || this->day() == 29))
         {
              date_ += date::months(rhs_months);
         }
         else
         {
              date::year_month_day_last
              temp{ date_.year() / date_.month() / date::last };
              temp += date::months{ rhs_months };
              if (temp.month() == date::month{ 2 })
              {
                   date_ = temp;
              }
              else
              {
                   date_ += date::months{ rhs_months };
              }
         }
    }
    else
    {
         date::year_month_day_last
             eom_last{ date_.year() / date_.month() / date::last };
         eom_last += date::months(rhs_months);
         date_ = eom_last;   // Implicit conversion to date::year_month_day type
    }

    return *this;
}
```

This will make subsequent tasks much easier.

Adding days to a date, as seen earlier, is trivial in comparison. Note however there is no addition assignment operator defined for days as there is for years and months, so we will need to convert to `time_point`(s) before computing the sum.

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

    return *this;
}
```

Note that the sum of the `time_point` (`sys_days`) and the `days` to be added are implicitly converted back to a `year_month_day` object when assigned to the `date_` member.

Pre-increment and decrement operators follow immediately from the `add_days` member function:

```
ChronoDate& ChronoDate::operator ++ ()
{
    return add_days(1);
}

ChronoDate& ChronoDate::operator -- ()
{
    return add_days(-1);
```

```
}
```

3.2.4 Comparison and Streaming Operators

The comparison operators `==` and `<=>` are immediate as these are already defined for `year_month_day`. 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 date_ == rhs.date_;
}

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

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

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

The `ChronoDate` class is now ready to be utilized in financial classes and calculations.

4 Cash Flow Schedules

An immediate application of the infrastructure now provided by our `ChronoDate` class is the generation of schedules of periodic payments that are ubiquitous in fixed income investing. To keep the code lighter and clearer for demonstration purposes, let us assume the value date occurs prior to the first payment or rate-fixing date.

[[Diagram(s) here – 1)bond payments, 2)floating rate payments (swap)]]

One more simplifying assumption will be that the payment frequency (tenor) will be in terms of months only.

Let us now design a `Schedule` class, with the following requirements:

- Take as constructor input:
  - The value date
  - The first regular period date (eg the first coupon payment of a bond, or the first rate reset date for a series of floating interest payments)
  - The final regular period date (or upper limit)
  - The payment tenor (in months)
- Generate a vector of dates forming a cash flow schedule based on the constructor inputs

The class declaration specifies what we will need. The schedule generation will commence by invoking the `generate_schedule_` private member function from inside the body of the constructor.

```
export class Schedule
{
public:
    Schedule(const ChronoDate& val_date, const ChronoDate& first_reg_date,
         const ChronoDate& last_date, int tenor);

    std::vector<ChronoDate> operator()() const;
```

```
private:
    ChronoDate val_date_, first_reg_date_, last_date_;
    int tenor_;
    std::vector<ChronoDate> schedule_;
    void generate_schedule_();
    ChronoDate mod_following_(const ChronoDate& check_date,
         const date::weekday& day_of_week);
};
```

The trickiest part of the design will be applying a business day convention, or roll method, to handle the case where a date does not fall on a business day. Further complicating this is when a non-business day falls at the end of the month. There are four commonly used roll methods, but again to avoid cluttering the example, assume the Modified Preceding rule is used:

- Roll to the next business day, except if the day would roll to the preceding month, in which case roll forward to the next business day

Due to edge cases such as a February date, or end-of-month, there are a couple of pesky problems we need to address. First, in the absence of any adjustments, the day of any following date should be the same as that of the first regular date. However, if we start from, say, December 30, 2022, and roll forward two months, this would result in an invalid date of February 30, 2023.

Second, the Modified Following business day rule can become somewhat convoluted as we need to naively roll the date forward first to check if it advances to the next month, in which case we need to go back and make adjustments.

In the class implementation, the constructor initializes the three dates that define the schedule, and the tenor (in months). The private `generate_schedule_` is then called upon construction to start the process.

```
Schedule::Schedule(const ChronoDate& val_date, const ChronoDate& first_reg_date,
    const ChronoDate& last_date, int tenor) : val_date_{ val_date },
    first_reg_date_{ first_reg_date }, last_date_{ last_date }, tenor_{tenor}
{
    generate_schedule_();
}
```

For purposes of demonstration, the first two dates are assumed to be valid and are appended to the `schedule_` vector.

```
void Schedule::generate_schedule_()
{
    // Assume the first two dates are valid:
    schedule_.push_back(val_date_);
    schedule_.push_back(first_reg_date_);
```

Now, for the remaining dates to be generated, each previous date (`next_date`) is advanced forward by the tenor and formally set to the same day value as the first regular date.

```
    auto next_date = first_reg_date_;
    while (next_date < last_date_)
    {
         next_date.add_months(tenor);

         auto is_wknd = next_date.weekend();

         if (!is_wknd)
         {
              schedule_.push_back(next_date);
         }
         else
         {
              schedule_.emplace_back(mod_following_(next_date, *is_wknd));
         }
    }
}
```

Each `next_date` is then tested whether it is a weekend. If this is the case, a helper function `mod_following_` checks whether it falls at the end of the month. This will determine whether the date is rolled forward or back, per the Modified Following business day rule.

```
ChronoDate Schedule::mod_following_(const ChronoDate& check_date,
    const date::weekday& day_of_week)
{
    // This function is only called if the date is a weekend.
    auto copy_check = check_date;
    copy_check.add_months(tenor_);
    if (check_date.month() == copy_check.month())      // Does not advance to next month
    {
         if (day_of_week == date::Saturday)
         {
              auto adj = Date{ date::sys_days(check_date.ymd()) + date::days(2) };
              ChronoDate ret{ adj };
              return ret;
         }
         else // day_of_week == date::Sunday
         {
              auto adj = Date{ date::sys_days(check_date.ymd()) + date::days(1) };
              ChronoDate ret{ adj };
              return ret;
         }
    }
    else
    {
         if (day_of_week == date::Saturday)     // Does advance to next month
         {
              auto adj = Date{ date::sys_days(check_date.ymd()) - date::days(1) };
              ChronoDate ret{ adj };
              return ret;
         }
         else // day_of_week == date::Sunday
         {
              auto adj = Date{ date::sys_days(check_date.ymd()) - date::days(2) };
              ChronoDate ret{ adj };
              return ret;
         }
    }
}
```

As an example, generate a schedule of monthly payments with settlement, first, and last dates as shown here.

```
ChronoDate val(2023, 8, 14);
ChronoDate first_reg(2023, 8, 30);
ChronoDate last(2024, 4, 30);

Schedule sched(val, first_reg, last, 1);
```

The function object `sched()` holds the generated vector of dates, and thus we can inspect the results with a range-based `for` loop:

```
for (auto& d : sched())
{
    cout << ++i << ": " << d << ", " << date::weekday(date::sys_days(d.ymd()))
        << endl;
}
```

These results are as follows:

```
2023-08-14, Mon
2023-08-30, Wed
```

```
2023-09-29, Fri
2023-10-30, Mon
2023-11-30, Thu
2023-12-29, Fri
2024-01-30, Tue
2024-02-29, Thu
2024-03-29, Fri
2024-04-30, Tue
```

5 Day Count Conventions

Day count conventions are used for computing year fractions between two dates. Depending on the type of bond or instrument, the day count convention will vary. For example, in the United States, Treasury Bonds and Notes use the Actual/Actual method, which takes the fractions of the number of days in leap and non-leap years and sums them. The 30/360 method, which assumes every month has 30 days and a year 360 days, is common for corporate and municipal bonds, and mortgage-backed securities. Money market securities typically assume the Actual/360 day count, where the actual number of days between two dates is divided by 360. There are also several variations on the 30/360 theme. The Actual/365 day count is often found in Canada, Australia, and the UK.

An Actual/252 day count is common in equity portfolio management, where 252 business days per year are assumed.

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

The interface is simple. We can just declare a pure virtual `operator()` for the calculations on the derived classes.

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

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

For our examples, let us take two day count examples, Actual/360, and 30/360. The particular 30/360 method used is the ISDA version (International Swaps and Derivatives Association – see References at end).

```
export class Act360 : public DayCount
{
public:
    double operator() (const ChronoDate& date1, const ChronoDate& date2) const
        override
    {
        return (date2 - date1) / 360.0;
    }
};

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

private:
```

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

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

        f(d1);
        f(d2);

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

Then, for some examples:

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

    auto yf_act01 = act_360(sd1, ed1);          // 2.46389
    auto yf_act02 = act_360(sd2, ed2);          // 0.502778

    auto yf_thirty_01 = thirty_360(sd1, ed1);   // 2.425
    auto yf_thirty_02 = thirty_360(sd2, ed2);   // 0.5
```

The results are shown in the comments. Note that the 30/360 day count yields a six-month year fraction as exactly 0.5.

6 Term Structure of Interest Rates

A term structure of interest rates takes in yield curve data from the market and provides the mechanisms for computing forward interest rates and discount factors. There are various forms of discrete forward rate calculations, but in financial modeling, the assumption of continuous rates is often used, and the computations are easier. As such, we will make this assumption as well to demonstrate the implementation of a term structure.

Mathematically speaking, these formulae are provided here. Discount factors are used on their own as well as intermediate calculations to compute forward rates. Each rate is based on current market rates as of the current date [[indicated by time 0]]. The forward interest rate from a time t to time T as seen in the market at time 0 is represented by

$$F(0; t, T)$$

Discount factors from t to T, as seen at time 0, are defined as

$$P(t,\ T) = P(0,\ T)/P(0,\ t)$$

where

$$P(0,\ t) = \ e^{-F(0;\ 0,\ t)\tau(0,\ t)}$$

Forward rates from t to T, as seen at time 0, are then:

$$F(0; t, T) = \ -\ ln(P(t,\ T))/\tau(t,\ T)$$

The function $\tau(t,\ T)$ represents the day count-adjusted year fraction from t to T.

To start, the declaration for a proposed `TermStructure` class can be outlined as follows.

```
export module TermStructure;
// . . .

import ChronoDate;
import DayCounts;
import Interpolation;

export class TermStructure
{
public:
    TermStructure(std::vector<ChronoDate>&& dates, std::vector<double>&& rates,
         std::unique_ptr<DayCount> day_count);
    double forward_rate(const ChronoDate& date1, const ChronoDate& date2) const;
    double disc_factor(const ChronoDate& date1, const ChronoDate& date2) const;
    double year_fraction(const ChronoDate& date1, const ChronoDate& date2) const;

private:
    LinearInterpolation lin_interp_;
    bool check_dates_(const ChronoDate& date1, const ChronoDate& date2) const;
    std::unique_ptr<DayCount> day_count_;
};
```

Noting that the yield curve data points are made up of distinct date/rate pairs, in order to obtain an arbitrary market interest rate $F(0; 0,\ t)$, it will need to be interpolated. To keep the example simpler, linear interpolation is assumed, but in practice this could be an arbitrary interpolation method handled polymorphically.

6.1 A Linear Interpolation Class

Before completing the `TermStructure` class, we will first write a class to handle linear interpolation. For dates beyond the right endpoint, the rate will remain constant as $F(0;\ 0, T_n)$, where T_n is the last date in the market yield curve data. And for completeness, to the left, the rate will be fixed at the instantaneous rate at $t = 0$.

The constructor will take in the vectors of dates and rates, and the round bracket operator will return the interpolated rate for a given input date.

```
export class LinearInterpolation
{
public:
    LinearInterpolation(std::vector<ChronoDate>&& dates,
    std::vector<double>&& rates):
         dates_{std::move(dates)}, rates_{std::move(rates)}
    {
         // In practice, should test if dates are sorted or not.
    }
```

In the round bracket operator, the endpoints are checked first and trivially addressed if necessary. For internal points, linear interpolation is applied. In the implementation that follows, three Standard Library features are used: `std::lower_bound`, `std::distance`, and `std::lerp` -- a new C++20 Standard Library function. Remark: `lower_bound` will return the position of an iterator of the first element in a container that is *greater* than or equal to some value, so in fact it ironically will serve as the upper bound on the interpolation interval.

`std::lerp` (in `<cmath>`) is a new C++20 Standard Library function. For a given interval say $[y_{i-1},\ y_i]$, and a parameter t in the interval [0, 1], it returns the interpolated value

$$y\ =\ y_{i-1}\ +\ t(y_i -\ y_{i-1})$$

In our case, t is obtained from the known date input and its immediate upper and lower bounds. With the indices of the interval bounds, we can then apply `std::lerp` to the rates. If there is an exact match of the date with the upper

bound, then the corresponding rate is immediate. The complete implementation of the round bracket operator could then be written as shown in the following code example.

```
double operator()(const ChronoDate& rhs) const
{
    if (rhs <= dates_.front())
    {
         auto check{ dates_.front() };
         return rates_.front();
    }
    else if (rhs >= dates_.back())
    {
         auto check{ dates_.back() };
         return rates_.back();
    }
    else
    {
         // iter is actually an *upper* bound
         auto iter = std::lower_bound(dates_.cbegin(), dates_.cend(), rhs);
         auto up_idx = std::distance(dates_.begin(), iter);
         if (rhs == *iter)
         {
              return rates_.at(up_idx);
         }
         else
         {
              // Run the interpolation...
              auto down_idx = up_idx - 1;
              auto t = (static_cast<double>(rhs - dates_.at(down_idx)))
                   / (static_cast<double>(dates_.at(up_idx) -
                       dates_.at(down_idx)));
              return std::lerp(rates_.at(down_idx), rates_.at(up_idx), t);
         }
    }
}
```

6.2 Term Structure Class Implementation

Returning to the term structure class, now that we have `LinearInterpolation`, we can define it as a member object (`lin_interp_`), and then instantiate it with arguments in the `TermStructure` constructor:

```
TermStructure::TermStructure(std::vector<ChronoDate>&& dates,
    std::vector<double>&& rates, std::unique_ptr<DayCount> day_count):
        lin_interp_{ std::move(dates), std::move(rates) },
        day_count_{ std::move(day_count) } {}
```

The `DayCount` member is handled polymorphically, deferring the particular type until run time, stored on the `day_count_` member. This takes care of the \tau(T_0, T_i) and \tau(T_{i-1}, T_i) terms in the discount factor and forward rate formulae above.

The discount factor function is used for its own calculation, plus it is a necessary step for computing the forward interest rate off of the term structure.

```
double TermStructure::
    disc_factor(const ChronoDate& date1, const ChronoDate& date2) const
{
    double rate1 = lin_interp_(date1);
    double rate2 = lin_interp_(date2);

    double disc_fctr1 =
        std::exp(-rate1 * (*day_count_)(lin_interp_.value_date(), date1));
```

```
    double disc_fctr2 =
        std::exp(-rate2 * (*day_count_)(lin_interp_.value_date(), date2));

    return disc_fctr2 / disc_fctr1;
}
```

The `LinearInterpolator` functor finds the yields for each spot discount factor and uses the result to get the forward discount factor. It also provides the value date that is stored as the first element in it member vector of dates. To calculate a forward rate, the corresponding (forward) discount factor is plugged into the forward rate formula to get the result.

```
double TermStructure::forward_rate(const ChronoDate& date1,
    const ChronoDate& date2) const
{
    return -std::log(disc_factor(date1, date2)) / (*day_count_)(date1, date2);
}
```

7 Pricing a Bond

The machinery above now allows us to easily calculate the present value, or price, of a bond:

```
import ChronoDate;
import DayCounts;
import TermStructure;
import Schedule;

export class Bond
{
public:
    Bond(const ChronoDate& sett_date, const ChronoDate& first_cpn_date,
         const ChronoDate& mat_date, int tenor, double coupon_rate,
         double face_value, TermStructure&& term_struct);

    double operator()() const;          // Returns calculated price


private:
    ChronoDate sett_date_, first_cpn_date_, mat_date_;
    int tenor_;
    double coupon_rate_, face_value_;
    TermStructure term_struct_;
    double yield_{ 0.0 };               // To be determined from the term structure
    vector<double> disc_fctrs_;
    Schedule sched_;
    double price_{ 0.0 };               // To be calculated

    // Private functions:
    void calculate_price_();
    void retrieve_yield_();
    void calculate_disc_factors_();
};

Bond::Bond(const ChronoDate& sett_date, const ChronoDate& first_cpn_date,
    const ChronoDate& mat_date, int tenor, double coupon_rate, double face_value,
    TermStructure&& term_struct):
    sett_date_{ sett_date }, first_cpn_date_{ first_cpn_date },mat_date_{ mat_date },
    tenor_{ tenor }, coupon_rate_{ coupon_rate },
    face_value_{ face_value }, term_struct_{ std::move(term_struct) },
    sched_{ sett_date, first_cpn_date, mat_date, tenor }
{
    calculate_price_();
```

```
}

double Bond::operator()() const
{

    return price_;
}

void Bond::calculate_price_()
{
    retrieve_yield_();
    calculate_disc_factors_();

    double cpn_amount = (face_value_ * coupon_rate_ * tenor_) / 12.0; // 12 mths/yr
    double pv_cpn_pmts{ 0.0 };
    for (double df : disc_fctrs_)
    {
         pv_cpn_pmts += df * cpn_amount;
    }

    price_ = disc_fctrs_.back() * face_value_ + pv_cpn_pmts;
}

void Bond::retrieve_yield_()
{
    yield_ = term_struct_.forward_rate(first_cpn_date_, mat_date_);
}

void Bond::calculate_disc_factors_()
{

    for (unsigned i = 1; i < sched_().size(); ++i)
    {
         disc_fctrs_.emplace_back(term_struct_.disc_factor(sched_().at(i - 1),
              sched_().at(i)));
    }
}
```

Time permitting, may include swap and swaption pricing.

8 References

https://stackoverflow.com/questions/62734974/how-do-i-add-a-number-of-days-to-a-date-in-c20-chrono (Answer from Howard Hinnant)

Fact 5 in https://stackoverflow.com/questions/59418514/using-c20-chrono-how-to-compute-various-facts-about-a-date, relevant to `days_in_month` -- answer from Howard Hinnant]]

Day counts: [[https://www.iso20022.org/15022/uhb/mt565-16-field-22f.htm]].

Josuttis: *The C++ Standard Library*, 2E