Using `std::chrono` Calendar Dates for Finance

Daniel Hanson

CppCon

16 September 2022

2022 Calendar

January '22

M	T	W	T	F	S	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

February '22

M	T	W	T	F	S	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28						

March '22

M	T	W	T	F	S	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

April '22

M	T	W	T	F	S	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

May '22

M	T	W	T	F	S	S
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

June '22

M	T	W	T	F	S	S
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

July '22

M	T	W	T	F	S	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

August '22

M	T	W	T	F	S	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

September '22

M	T	W	T	F	S	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

October '22

M	T	W	T	F	S	S
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

November '22

M	T	W	T	F	S	S
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30				

December '22

M	T	W	T	F	S	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

New C++20 Features

Concepts

Modules

Ranges

Dates

Dates in C++20

- *Critical* for bond/fixed income trading!

- Addition to **`std::chrono`** (Howard Hinnant – author)

- Quick disclaimer
  - What follows is based on own research and testing
  - Have not found much information yet “in the literature”
  - Sharing what I know and have learned
  - Objective is how to use, rather than technical details

Why Dates are Important in Quant Finance

- Fixed income products usually involve a series of payments

- Based on regular payment schedules
    - Monthly
    - Quarterly
    - Semiannual
    - Annual

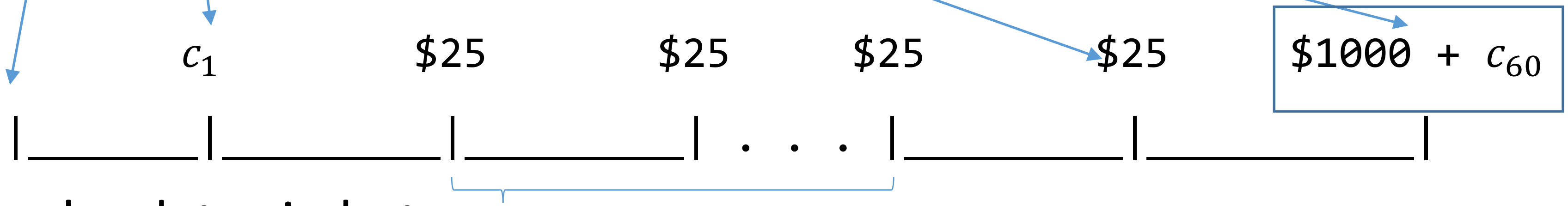
Why Dates are Important in Quant Finance

- Examples
  - Coupon-paying bonds
  - Mortgage and car loans
  - Annuities
  - Interest rate swaps (fixed/float rate payments)
  - Futures and options on bonds and swaps

Bond Pricing

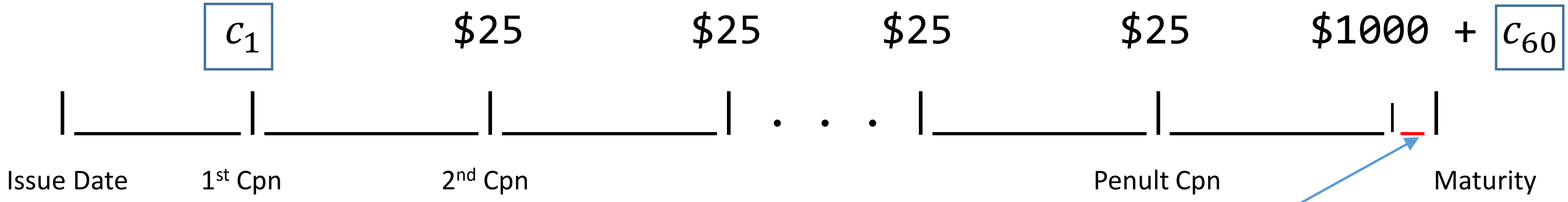
- $1000 Face Value
  - 5% annual coupon paid semiannually over 30 years
  - Regular coupon payment = (.05)(1000)/2 = $25
  - Face value returned on final coupon payment date

- Contractual Dates
  - Issue date
  - First payment date
  - Penultimate payment date
  - Maturity date (final coupon payment and return of $1000 face value)



- Regular dates in between
  - $25 payments
  - Roll to next business day if weekend (or holiday)

Payments and Day Count Conventions



- c_1 and c_{60} might be *irregular* payments
  - $c_1 = 1000(.05)(time\ between\ issue\ date\ and\ 1st\ pmt\ date)$
  - $c_{60} = 25 + \ 1000(.05)(time\ beyond\ reg\ pmt\ period)$

- The time value over each irregular interval is a calculated *year fraction*, based on the contractual day count convention
  - Actual/365
  - 30/360
  - Others

Day Count Conventions

- Actual/365

  Year fraction = $(\# \ of \ days \ between \ date_1 \ \& \ date_2)/365$

- 30/360
  - Assume every month has 30 days
  - Assume each year has 360 days

$$\text{DayCountFactor} = \frac{360 \times (Y_2 - Y_1) + 30 \times (M_2 - M_1) + (D_2 - D_1)}{360}$$

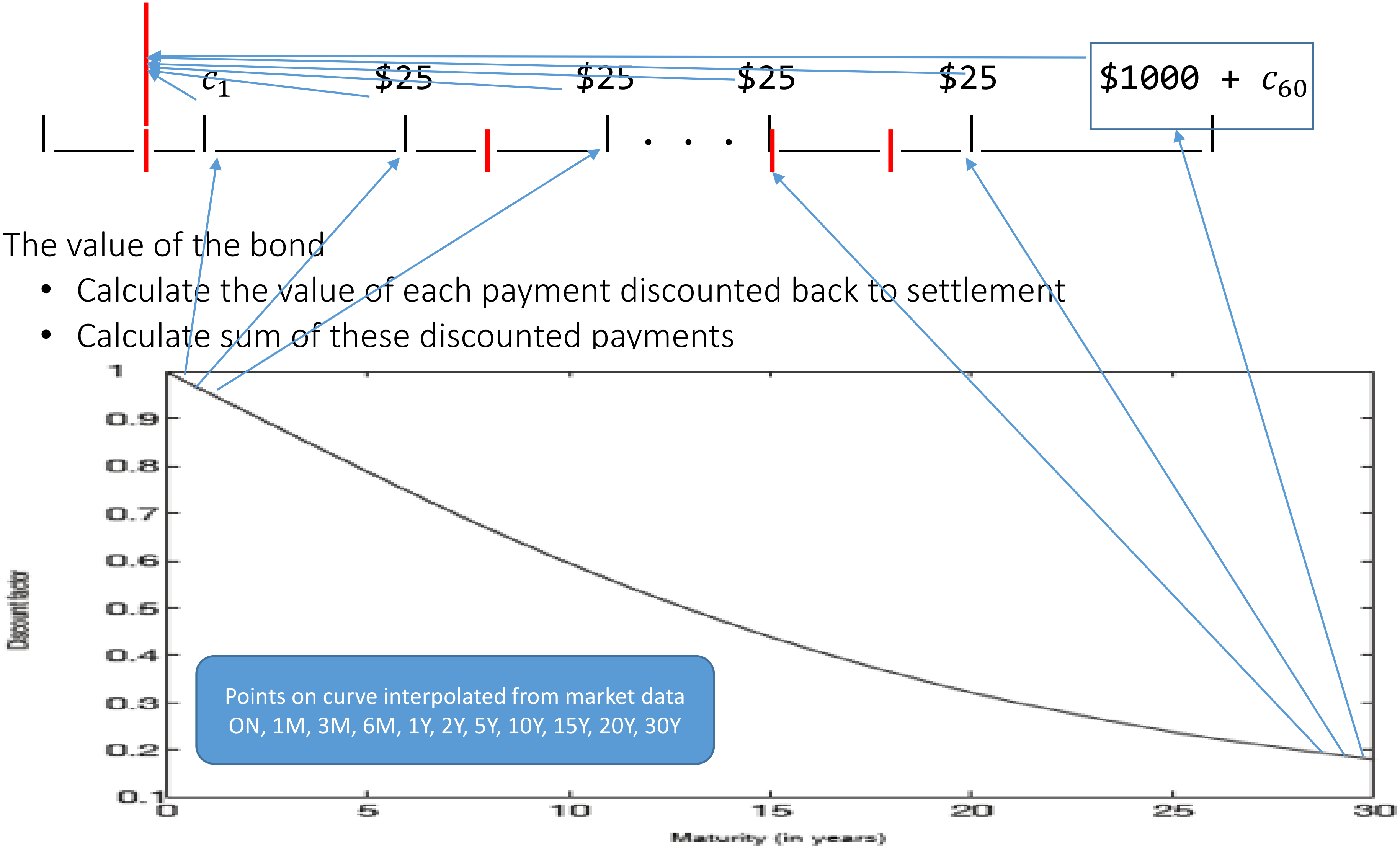
➢ D_1 = MIN (D_1, 30)
➢ If D_1 > 29 Then D_2 = MIN (D_2, 30)
➢ If D_2 is 31 and D_1 is 30 or 31, then change D_2 to 30
➢ If D_1 is 31, then change D_1 to 30

- Example: Year fraction between 2022-9-16 and 2023-3-16
  - Actual/365: 0.49589
  - 30/360: 0.5

https://en.wikipedia.org/wiki/Day_count_convention

Bond Price

- A bond is priced as of its *settlement date*
  - Can be any business date between issue and maturity



- The value of the bond
  - Calculate the value of each payment discounted back to settlement
  - Calculate sum of these discounted payments

Points on curve interpolated from market data
ON, 1M, 3M, 6M, 1Y, 2Y, 5Y, 10Y, 15Y, 20Y, 30Y

Brigo & Mercurio, *Interest Rate Models: Theory and Practice* 2nd Ed (Springer), Ch 1, Fig 1.2

std::chrono::year_month_day

“Only one of us is in the correct time continuum”

Constructing a Date

- A standard date in **`std::chrono`** is represented by an object of the class **`std::chrono::year_month_day`**

```
import <chrono>;    // Header unit for std::chrono (including dates)
```

OR

```
#include <chrono>; // Header unit for std::chrono (including dates)
```

- Options for constructing this class (non-exhaustive), eg 2002-11-14 (y-m-d):

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

  - ➢ **`year`**, **`month`**, and **`day`** are also classes in C++20 **`std::chrono`**

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

  - ➢ Spelled-out months are types in **`std::chrono`**

Assignment

- Assignment with the forward slash operator

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

- Order can be y/m/d or m/d/y with integers otherwise as long as the 1st position is obvious

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

- There are other possible formats (https://github.com/HowardHinnant/date)

std::chrono::year_month_day Serial Representation

- A **`year_month_day`** date can also be measured in terms of the number of days since an epoch, the default being the UNIX epoch:

January 1, 1970.

- Commonly known as a *serial date*

- Dates prior to the epoch are represented by negative integers

std::chrono::year_month_day Serial Representation

- The serial date can be accessed as follows:

```
int days_since_epoch_count =
    std::chrono::sys_days(ymd).time_since_epoch().count();
```

- Technically what is happening here:

  - the **sys_days()** operator returns the **ymd** date as a **sys_days** object (**sys_days** is an alias for **std::chrono::time_point**)

  - Its **time_since_epoch()** member function returns a **std::chrono::duration** type

  - The corresponding integer value is then accessed with the **count()** function

- The serial value equivalent to the date **ymd** (2002-11-14) is 12,005

Date Differences

- Find the number of days between

  ymd (2002-11-14)

  and **ymd_later** (May 14, 2003)

- Take the difference between the **sys_days** equivalents and apply the **count** function to the result:

```
using namespace std::chrono;        // Will be assumed going forward


year_month_day ymd{year{2002}, month{11}, day{14});
year_month_day ymd_later{year{2003}, month{5}, day{14}};


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

Year, month, and day accessors

- Accessor functions on **year_month_day**

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

- Difference operator returns integer types

```
date2.year() – date1.year()
date2.month() – date1.month()
date2.day() – date1.day()
```

Caution: Validity of a Date

- It is possible to set **year_month_day** objects to invalid dates

- Validity is checked with the **ok()** member function that returns a **bool**

```
year_month_day ymd{year{2002}, month{11}, day{14} };
bool torf = ymd.ok();                  // true

year_month_day ymd_invalid{year{2018}, month{2}, day{31} };
torf = ymd_invalid.ok();               // false

year_month_day ymd_completely_bogus{year{-2004}, month{19}, day{58} };
torf = ymd_completely_bogus.ok();      // false
```

Leap Years

- A **year_month_day** date can also be checked easily whether it is in a leap year or not

- The **is_leap()** member function on the **year** class takes care of this for us:

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

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

Last Day of the Month

- There is no member function available

  - We can create a **year_month_day_last** object for a given month and year:

```
year_month_day_last
    eom{ year{ 2009 } / April / std::chrono::last};
```

  - And then, get the day value:

```
auto last_day = static_cast<unsigned>(eom.day());
```

- A **year_month_day_last** type is also implicitly convertible back to a **year_month_day**

```
year_month_day ymd_eom = eom_check;
```

- Might prefer to avoid generating the **year_month_day_last** object, however

Date Algorithms

- ***chrono****-Compatible Low-Level Date Algorithms* are provided on the **`std::chrono`** GitHub site
  - https://howardhinnant.github.io/date_algorithms.html
  - ***"[K]ey algorithms that enable one to write their own date class"***

Date Algorithms

- We can combine these two algorithms
  - **`last_day_of_month_common_year`**
  - **`last_day_of_month`**

```
// User-defined last_day_of_the_month
unsigned last_day_of_the_month(const std::chrono::year_month_day& ymd)
{
    unsigned m = static_cast<unsigned>(ymd.month());
    std::array<unsigned, 12> normal_end_dates{ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    return (m != 2 || !ymd.year().is_leap() ? normal_end_dates[m - 1] : 29);
}
```

- Avoid creation of extra **`year_month_day_last`** object

Adding Years and Months

- Simplest way is to use the **+=** operator

```
year_month_day ymd{year{2002}, month{11}, day{14} };
ymd += std::chrono::months(1);     // Result: 2002-12-14
ymd += std::chrono::months(18);    // Result: 2004-06-14
ymd += std::chrono::years(2);      // Result: 2006-06-14
```

- Note that **`months`** and **`years`** are types in **`std::chrono`**

- Subtraction assignment is also available:

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

Caution: Adding Years and Months

- Can result in invalid dates, however…

```
// 2015-01-31
year_month_day ymd_eom_1{year{2015}, month{1}, day{31} };

// 2014-08-31
year_month_day ymd_eom_2{year{2014}, month{8}, day{31} };

// 2016-02-29
year_month_day ymd_eom_3{year{2016}, month{2}, day{29} };



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

- But note that the year and month are correct

Adding Years and Months

- Adding years – only the last day of February is an issue (leap year)
- Adding months – more special cases (different number of days in different months)
  - Naively attempt as before
  - If valid, return the result
  - If not valid
    - ➢ Year and month are correct (eg **2018 – 2** – 30)
    - ➢ Get proper last day of year and reset (**2018 – 2 – 28**)

```cpp
void add_months_algo(std::chrono::year_month_day& ymd, unsigned mths)
{
    using namespace std::chrono;

    ymd += months(mths);    // Naively attempt the addition

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

Adding Days

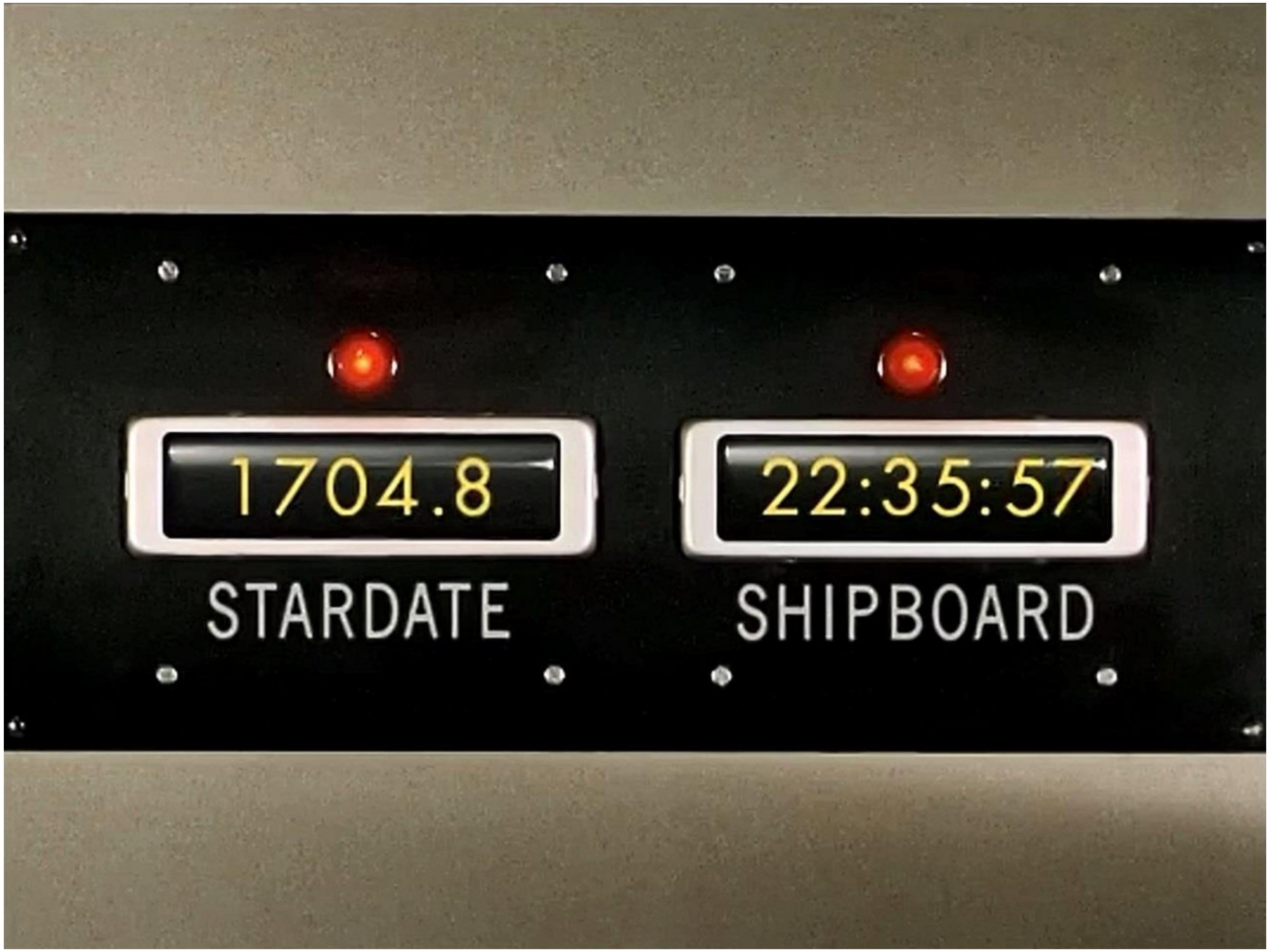
- There is no **+=** operator defined for adding days.
- Need to obtain the equivalent **sys_days** object before adding the number of days:

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

// Obtain the sys_days equivalent of ymd, and then add three days:
auto add_days = sys_days(ymd) + std::chrono::days(3); // no change in ymd
```

- The resulting **sys_days** is implicitly convertible to a **year_month_day**:

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

Date and Day Count Classes



A Date Class Wrapper

- Encapsulate the complexities of **`year_month_day`** in a user-defined class we'll call **`ChronoDate`**

- First, revisit the typical requirements for financial date calculations

A Date Class Wrapper

- State
  - Leap year ○
  - Days in month ○
- Arithmetic Operators
  - Number of days between two dates ◕
  - Addition ○
    - *Years*
    - *Days*
    - *Months*
- Accessors
  - Year, Month, Day ◕
  - Serial date integer representation (days since epoch) ○
  - **`year_month_day`** data member ◕
- Modifying Function: Roll to next business day if weekend
- Comparison operators
  - == ◕
  - <=> ◕

A Date Class Wrapper – Declaration (public methods)

```
import <chrono>;
namespace date = std::chrono;

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

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


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

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

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


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

A Date Class Wrapper – Declaration (private members/fcn)

```
// Store the underlying std::chrono date
date::year_month_day date_;

int serial_date_;

void reset_serial_date_();
```

A Date Class Wrapper – Implementation - Constructors

```
// Integer arguments – convert to std::chrono types in constructor
ChronoDate::ChronoDate(int year, unsigned month, unsigned day) :
    date_{year{year} / month{month} / day{day} }
{
    if(!date_.ok())       // std::chrono member function to check if valid date
    {
       std::exception e("ChronoDate constructor: Invalid date.");
       throw e;
    }
    reset_serial_date_();          // Sets days since epoch (private)
}


// Default:
ChronoDate::ChronoDate():date_{year(1970), month{1}, day{1} },
    serial_date_{1} { }
```

Reset Serial Date

- Just use the earlier result and wrap in a private function:

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

Difference of Two Dates

- Just take the difference of the two serial date members on each object
- Avoid **sys_days(.)** conversion, and **time_since_epoch()** and **count()** function calls each time

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

    // Avoid:
    // return (sys_days(date_).time_since_epoch()
    //     - sys_days(rhs.date_).time_since_epoch()).count();
}
```

Business Day Roll

- If a transaction or contract date falls on a weekend
    - Roll forward to next business date
    - If date is rolled into the next month, roll back to the previous biz date
    - *Modified Forward* rule

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

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

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

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

Day Counts

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

// *** Class Act365 ***
double Act365::operator()(const ChronoDate& date1, const ChronoDate& date2) const
{
    return (date2 - date1) / 365.0;
}
```

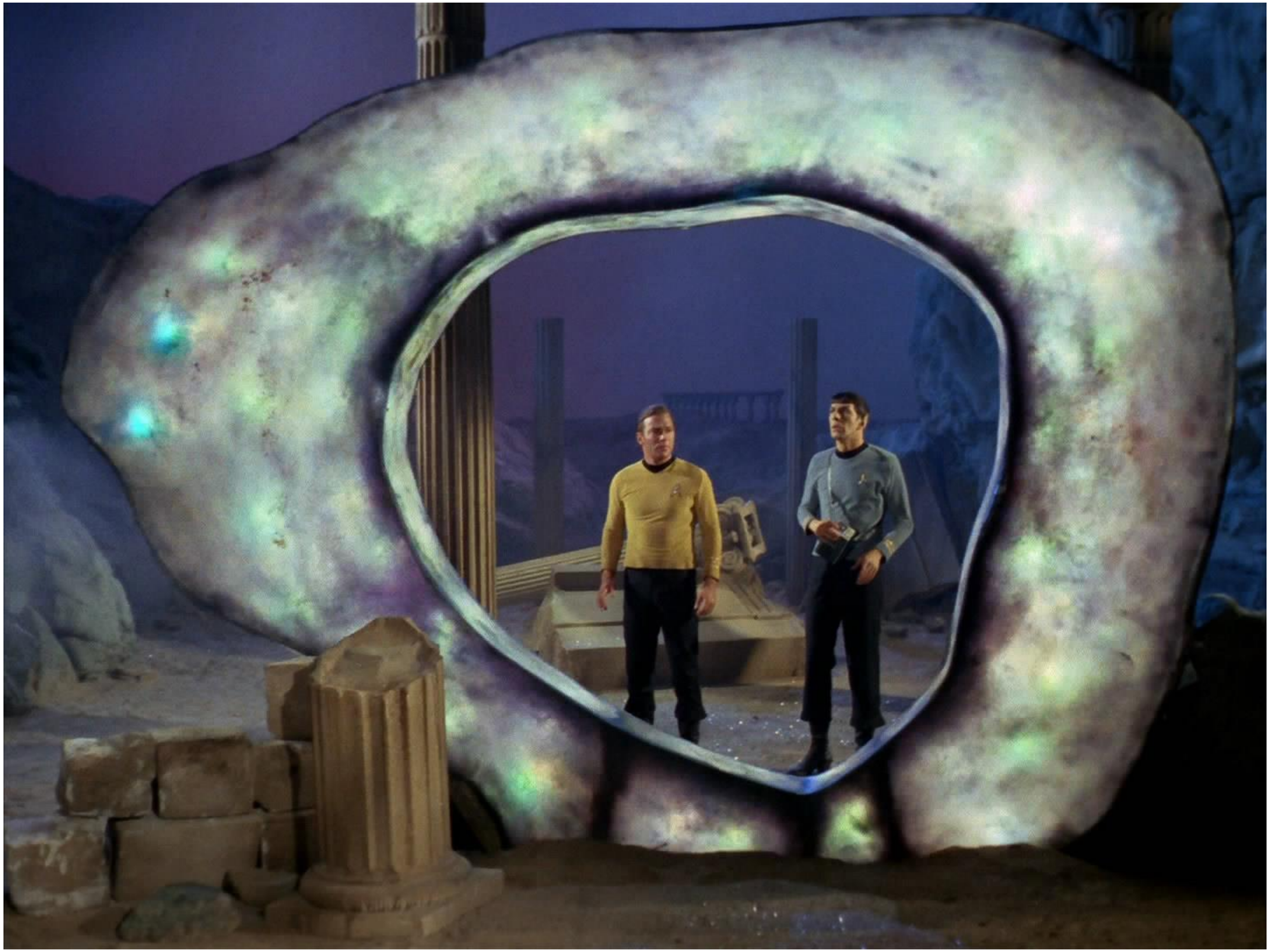
Day Counts

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

unsigned Thirty360::dateDiff_(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;
}
```

$$\text{DayCountFactor} = \frac{360 \times (Y_2 - Y_1) + 30 \times (M_2 - M_1) + (D_2 - D_1)}{360}$$

- D_1 = MIN (D_1, 30)
- If $D_1 > 29$ Then D_2 = MIN (D_2, 30)
- If D_2 is 31 and D_1 is 30 or 31, then change D_2 to 30
- If D_1 is 31, then change D_1 to 30

Wrap-Up



Summary

- The inclusion of dates in C++20
  - Is great to have for computational finance
  - Especially fixed income/derivatives trading
  - Possible to have invalid dates

- Wrap **`year_month_day`** in a user-defined class
  - yyyy/mm/dd representation
  - Serial date representation
  - Is leap year, date valid, number of days in month
  - Accessors for year, month, day
  - Number of days between two dates
  - Add years, months, days
  - Business day rules for weekends
  - More intuitive interface
  - Handles invalid date cases

Summary

- We now have a user-defined date class available to use in
  - Day count classes
  - Yield curve classes and term structure models
  - Bond pricing
  - Interest rate derivatives pricing models

- Slides (pdf) and sample code will be available on GitHub

  https://github.com/QuantDevHacks/CppCon-2022-C-20-Dates-in-Finance

- Contact:
  - https://www.linkedin.com/in/danielhanson/
  - daniel (at) cppcon.org

Thank you!