How I Learned to Stop Worrying and Love C++20

NWCPP
16 February 2022

Topics

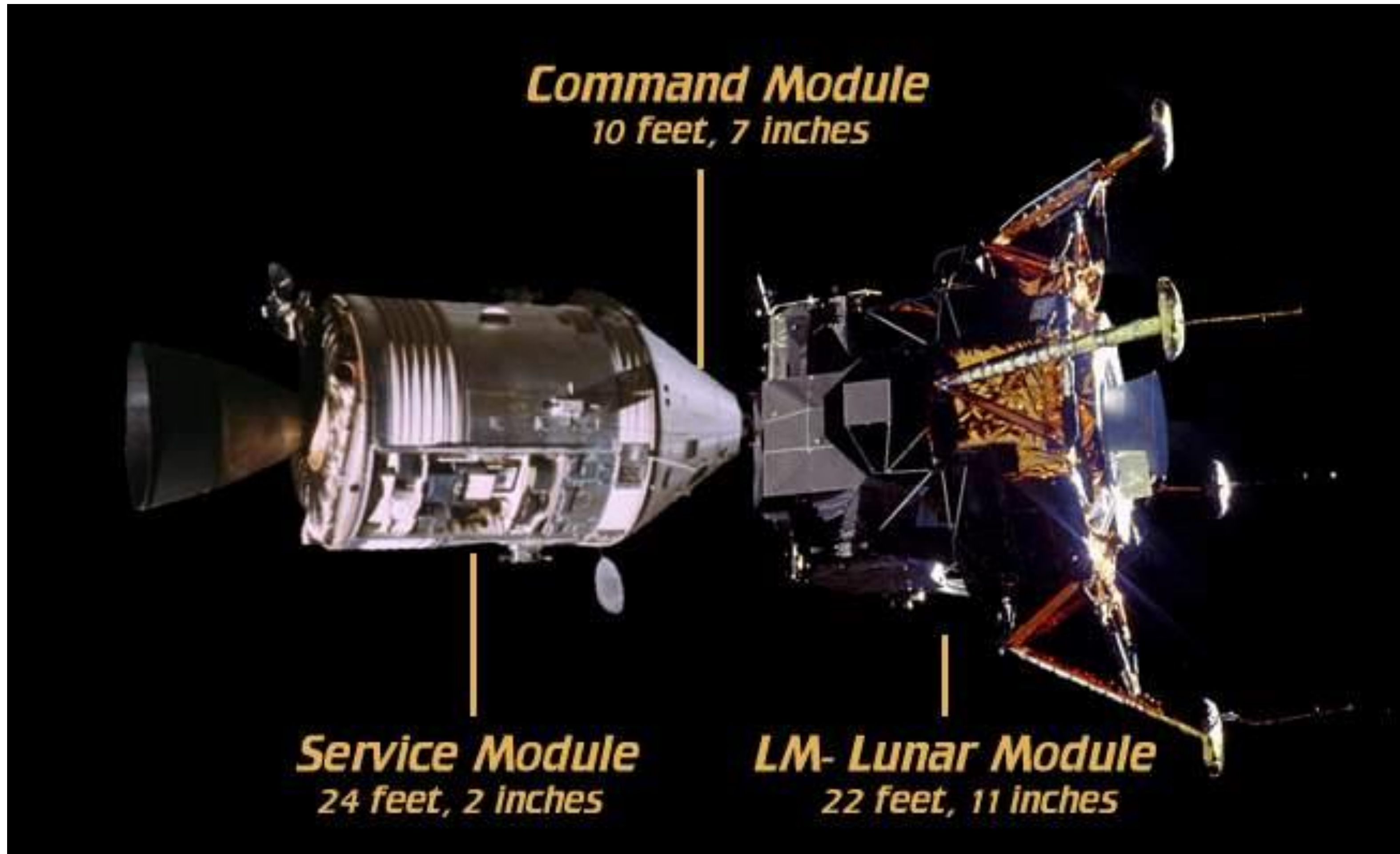
- Modules
- Mathematical Constants
- Spaceship operator
- Ranges and Views
- Dates
- Remarks:
  - **`std::format`** is used in the sample code (will post on nwcpp.org)
  - Concepts – need a separate session for this

Copyright © 2022 Daniel Hanson

Disclaimers and Caveats

- This is a high-level introduction
- I am sharing what I know and have learned about C++20
- There is still plenty more I need to learn

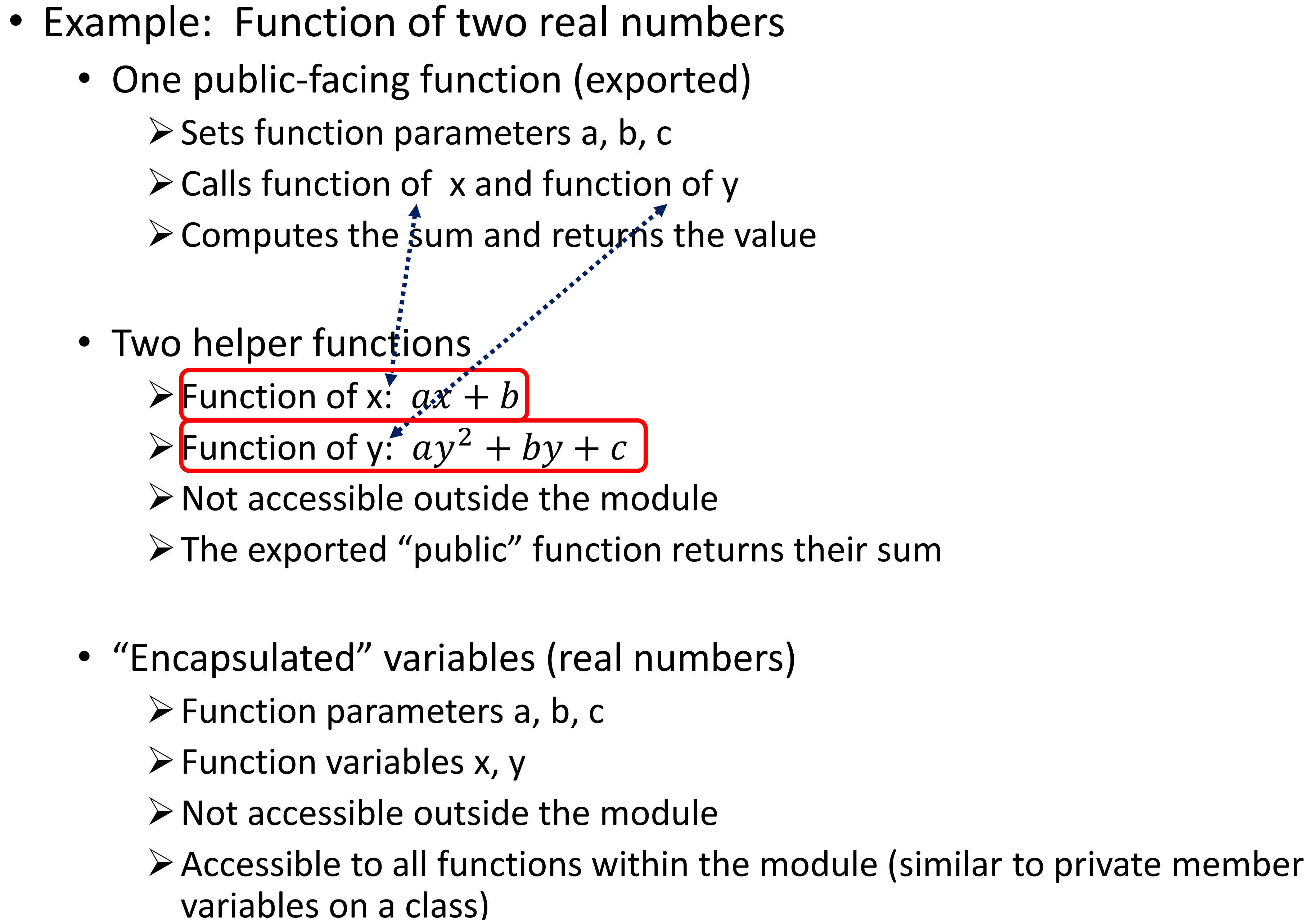
Modules



Modules

- Modules are great!
  - No more header files or include guards (goodbye!)
  - Imported modules don't leak to other modules
  - Can write implementations inside declarations without guilt
  - Function implementations
    - Exported functions visible to the public
    - "Private" module variables (global among module functions)
    - Non-exported helper functions
  - Faster compile times

Modules



Example: Function of two real numbers

```
// This defines the module and allows
// it to be imported elsewhere:
export module FunctionAndHelpersModule;
// Non-exported module variables:
double f_x{ 0.0 }, f_y{ 0.0 };
double a_, b_, c_;
// Non-exported helper function declarations:
void fcn_of_x(double x);
void fcn_of_y(double y);

export double primary_fcn(double a, double b, double c,
          double x, double y)
{
    a_ = a;
    b_ = b;
    c_ = c;

    fcn_of_x(x);
    fcn_of_y(y);

    return f_x + f_y;
}
```

```
// The following functions are "private"
// within the module:
void fcn_of_x(double x)
{
    f_x = a_ * x + b_;
}

void fcn_of_y(double y)
{
    f_y = (a_ * y + b_) * y + c_;
}
```

Copyright © 2022 Daniel Hanson

Standard Library Header Units

- A proposal exists to for reorganize the Standard Library into “a standard-module version” [Stroustrup P2412r0]
  - Was hoped for in C++20
  - Likely now C++23
  - Microsoft has a draft version with the Visual Studio compiler

- In the interim: Header Units [WG 21, p1502r1]
  - Apply to almost all C++ Standard Library declaration files
  - **`import <vector>;        // Example`**
  - Exceptions: headers inherited from C, such as
    - **`<cassert>`**
    - **`<cmath>`**
    - Still need, eg, **`#include <cmath>`** in the *global fragment* of the module

Standard Library Header Units

- The global fragment of a module:

```
module;
#include <cmath>

export module StdLibMod;          // Defines module StdLibMod
                                  // Must follow global fragment (if used)
```

- Note: Header files **#include**-d in the global fragment ***will*** leak into other translation units where the module is imported.

Modules Prevent Leaking into Other Translation Units

- If a module `A` imports another module `B`,

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

- If `A` is imported into another module `C`, module `B` will *not* be implicitly imported – explicit instructions required if `B` is needed:

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

Copyright © 2022 Daniel Hanson

export import

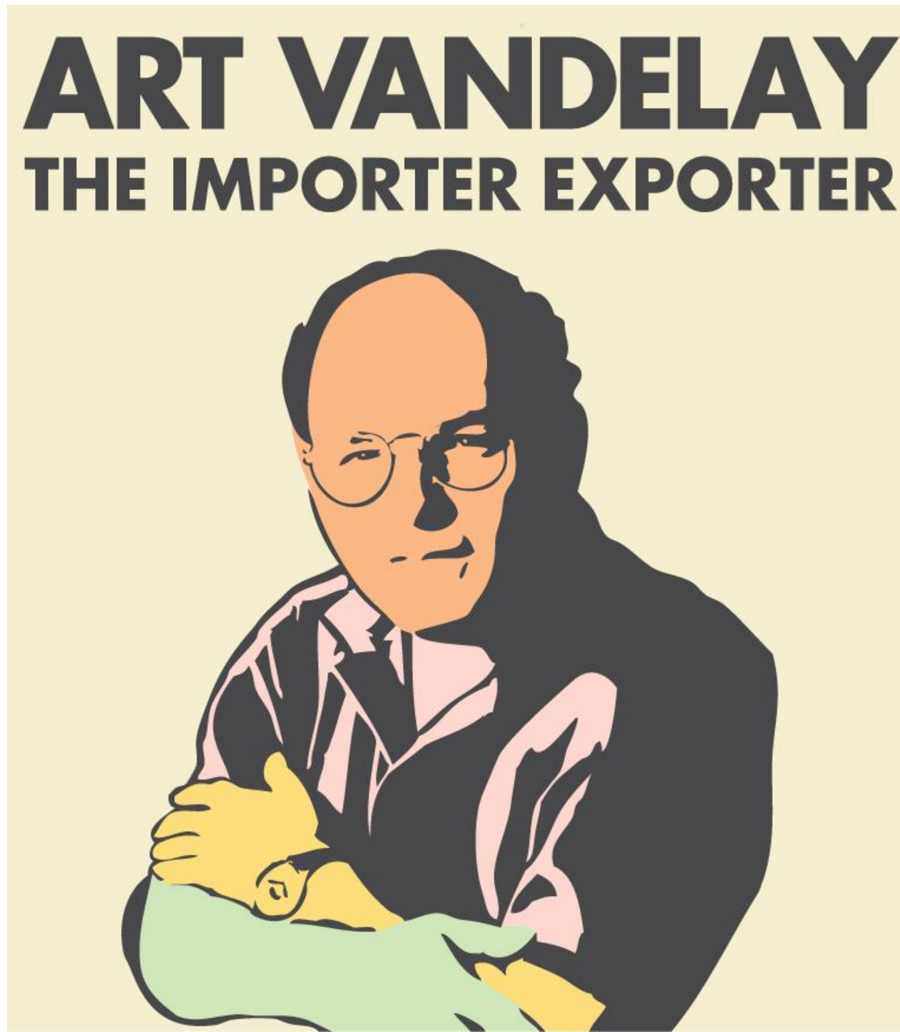
- If we *want* module `B` to be exported with module `A`, put:

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

- Then, it is no longer necessary to explicitly import module `B` into module `C` like before:

```
export module C;
import A;      // Module B is now also imported into module C
```

Mathematical Constants



Mathematical Constants

- Convenient **`constexpr`** mathematical constants
- Some of the most commonly used in math and statistics:

C++ constant	\`e\`	\`pi\`	\`inv_pi\`	\`inv_sqrt_pi\`	\`sqrt2\`
Definition	e	$\boldsymbol{\pi}$	$\frac{1}{\pi}$	$\frac{1}{\sqrt{\pi}}$	$\sqrt{2}$

- **`import <numbers>;`**
- Scoped with **`std::numbers::`**

Mathematical Constants

- Convenient **constexpr** mathematical constants
- For example, to implement the function

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

we could write

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

Mathematical Constants

- Curiously, there is no constant for $\frac{1}{\sqrt{2}}$ or $\frac{1}{\sqrt{2\pi}}$

- Although there is one for $\frac{1}{\sqrt{3}}$

- Despite the first two being far more commonly present in mathematical and statistical calculations

- The Boost Mathematical Constants library contains both

The Spaceship Operator <=>

The Spaceship Operator

- Example: Complex number class **`MyComplex`**

- Two complex numbers $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$,

  - x_k, y_k real numbers (**`double`** types), $i = \sqrt{-1}$

  - Define
    - $z_1 == z_2$ when $x_1 == x_2$ *and* $y_1 == y_2$
    - $z_1 > z_2$ when $|z_1| > |z_2|$, where $|z_k| = \sqrt{x_k^2 + y_k^2}$
    - $z_1 < z_2$ when $|z_1| < |z_2|$

- Operators **==** and **<** determine all remaining comparison operators

- ***No longer need to define all six comparison operators!***

The Spaceship Operator

- **`import <compare>;`**

- The **<=>** operator is no longer Boolean; return types are
  - **`std::strong_ordering`**: means that any two values can be compared, as is the case for *integral types*
  - **`std::weak_ordering`**: non-comparable assignments such as infinity and NaN, as is the case for *floating-point types*
  - **`std::partial_ordering`**: admits incomparable values: a < b, a == b, and a > b may all be false (cppreference.com)

- For **`MyComplex`**, use **`std::weak_ordering`**
  - **`std::weak_ordering::less`**
  - **`std::weak_ordering::equivalent`**
  - **`std::weak_ordering::greater`**

The Spaceship Operator

- To define the spaceship operator

1. Implement **==**

2. Implement **<=>** with definitions of **<** and **==**

3. Return type is **`std::weak_ordering`**

The Spaceship Operator

```
bool MyComplex::operator == (const MyComplex& rhs) const
{
    // tol = tolerance
    double tol =  std::sqrt(std::numeric_limits<double>::epsilon());
    if(std::abs(real_ - rhs.real_) < tol
        && std::abs(imag_ - rhs.imag_) < tol) {
        return true;
    }
    else {
        return false;
     }
}
```

```
std::weak_ordering MyComplex::operator <=> (const MyComplex& rhs) const
{
    if (std::hypot(real_, imag_)
        < std::hypot(rhs.real_, rhs.imag_)) {
        return std::weak_ordering::less;
    }
    else if (*this == rhs) {
        return std::weak_ordering::equivalent;
    }
    else {
        return std::weak_ordering::greater;
    }
}
```

The Spaceship Operator

- Remark: Definition of the **==** operator and the **`equivalent`** condition inside the **<=>** overload may seem redundant, but this is per the specifications in the ISO Standard

- Omission of one or the other can yield results other than those intended

- A default **<=>** operator also exists (use with caution)

Ranges and Views

Ranges and Views

- Ranges
  - Provide abstractions of STL algorithms that are more intuitive
  - In many cases avoid the **`begin`** and **`end`** functions
- Views
  - Allow certain operations to be performed on an STL container without modifying the container
  - Support functional behavior, allowing composition of multiple functions similar to piping in Linux scripting

Copyright © 2022 Daniel Hanson

Compare **std::count_if**

```
import <algorithm>;
import <ranges>;
// . . .

std::vector<int> int_coll{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
bool is_odd(int n)           // Predicate
{
    return ((n % 2) == 1);
}

// Compare the old way vs using the ranges version (num_odd = 5):
auto num_odd = std::count_if(int_coll.begin(), int_coll.end(), is_odd);

num_odd = std::ranges::count_if(int_coll, is_odd);  // More intuitive!
```

Compare **`std::transform`**

```
// Auxiliary function
export template<typename T>
T square(const T& t)
{
    return(t * t);
}

import <algorithm>;
import <ranges>;
// . . .

std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Compare the old way vs using the ranges version (num_odd = 5):
std::transform(v.begin(), v.end(), v.begin(), square<int>);

std::ranges::transform(v, v.begin(), square<int>);  // Cleaner
```

Copyright © 2022 Daniel Hanson

Ranges – Remarks

- Using **`std::back_inserter`** is also straightforward with ranges:

```
std::vector<int> w;
std::ranges::transform(v, std::back_inserter(w), square);
```

- Not yet equipped with range equivalents:
    - **`<numeric>`** algorithms
    - Parallel algorithms

Views

- Good news: Views are really cool
  - A view allows certain operations to be performed on an STL container without modifying the container
  - Also avoids container copies
  - Functional capabilities

- Examples
  - **`std::views::take`**
    - ➢ Takes the first *n* elements of a view and discards (ignores) the rest
  - **`std::views::filter`**
    - ➢ Filters in a set of elements according to a predicate
  - **`std::views::transform`**
    - ➢ Modifies elements of a view based on an auxiliary function
  - **`std::views::drop`**
    - ➢ Removes (ignores) the first *n* elements of a view

- Bad news: Still limited choices – eg, these do not include versions of **`count_if`**, **`find_if`**, **`partial_sum`**, **`adjacent_difference`**, many others…maybe C++23?

Views

- First, proceed naively:

```
std::vector<double> w(10);
std::iota(w.begin(), w.end(), -5.5);  // -5.5, -4.5, . . ., 3.5 (10 elements)

auto take_five = std::views::take(w, 5);
auto two_below = std::views::filter(take_five, [](double x) {return x < -2.0; });
auto squares = std::views::transform(two_below, [](double x) {return x * x; });
auto drop_two = std::views::drop(squares, 2);
auto sum_result = std::accumulate(drop_two.begin(), drop_two.end(), 0.0);
```

- Intermediate results (non-owning):
  - `take_five` = $-5.5, -4.5, -3.5, -2.5, -1.5$
  - `two_below` = $-5.5, -4.5, -3.5, -2.5 < -2.0$
  - `squares` = $30.25,\ 20.25,\ 12.25,\ 6.25$
  - `drop_two` = $12.25,\ 6.25$ (1st two elements of **`squares`** dropped)
  - `sum_result` = 18.5

Views

- (Most) STL algorithms and ranges will accept a view as an argument

- Examples:

```
auto sum_result = std::accumulate(drop_two.begin(), drop_two.end(), 0.0);

std::ranges::transform(squares, std::back_inserter(v),
                  [](double x) {return std::sqrt(x); });

auto max_view = std::ranges::max_element(take_five);
```

- No function yet to copy results of a view in an STL container
  - A **`ranges::to`** function was originally planned for C++20
  - Has been postponed until C++23
  - For now, can just use a range-based **`for`** loop

Copyright © 2022 Daniel Hanson

Views – A Functional Approach

- Alternatively, pipe together
- This is what makes views cool:

```
std::vector<double> w(10);
std::iota(w.begin(), w.end(), -5.5);  // -5.5, -4.5, . . ., 3.5 (10 elements)

auto drop_two = std::views::take(w, 5)
         | std::views::filter([](double x) {return x < -2.0; })
         | std::views::transform([](double x) {return x * x; })
         | std::views::drop(2);

auto sum_result = std::accumulate(drop_two.begin(), drop_two.end(), 0.0);
```

- Result (same):

`sum_result =` 18.5

Dates

Dates

- C++20 finally has date-related classes: header only implementation
- Particularly useful for computational finance (bond/fixed income)
- Expected in C++20 version of **`<chrono>`**; however…
  - Not in Visual Studio 2019
  - Not in Clang (last time I checked)
  - Workaround: Download the source code from Howard Hinnant's GitHub
- Notes
  - Some commonly used operations require multiple steps and casting
  - Lacking in documentation – better luck on Stack Overflow
  - Personal project: Wrap in library that mimics Excel date functions
- Some examples follow

Copyright © 2022 Daniel Hanson

Dates: Construction

- Create date objects:

```
#include <date/date.h>      // Source code from GitHub

date::year_month_day ymd(date::year(2002), date::month(11), date::day(14));
cout << ymd << endl << endl;         // << is overloaded for date::year_month_day
// Alternative:
date::year_month_day ymd_alt(2002y, November, 14d);
```

- Other class formats possible: mm-dd-yyyy, dd-mm-yyyy
    - **date::month_day_year**
    - **date::day_month_year**

Copyright © 2022 Daniel Hanson

Dates: Day Counts

- Example: day count conventions – common in computational finance
  - Default integer value: based on days since UNIX epoch 1970.01.01
  - Facilitates calculating number of days between two dates
  - Convert date to **`date::sys_days`** object
  - Use **`count()`** member function on **`date::sys_days`**
  - (**`sys_days`** is a **`std::chrono::time_point`** (`date::time_point`) that represents the same date as a **`year_month_day`** object)

```
// 50 years since epoch:
date::year_month_day ymd50(date::year(2020), date::month(1), date::day(1));
auto days_since_epoch = date::sys_days(ymd50).time_since_epoch().count();           // 18262

// Number of days between two dates:
date::year_month_day ymd(date::year(2002), date::month(11), date::day(14));
date::year_month_day ymd2(date::year(2003), date::month(11), date::day(14));
auto no_days = (date::sys_days(ymd2) - date::sys_days(ymd)).count();

// ACT/365 and ACT/360 day counts:
auto act_360 = no_days/360.0;  // 1.01389
auto act_365 = no_days/365.0;  // 1
```

- Example: 30/360 day count (more interesting case)
  - Assumes 30 days in each month
  - Assumes 360 days per year
  - Common in fixed income calculations

```
double thirty_360(const date::year_month_day& date1, const date::year_month_day& date2)
{
    // date1.day() returns a date::day type, not integer type!
    // Cannot cast to int, but unsigned works
    auto d1 = static_cast<unsigned>(date1.day());
    auto d2 = static_cast<unsigned>(date2.day());
    if (d1 == 31) d1 = 30;
    if ((d2 == 31) && (d1 == 30)) d2 = 30;
    auto diff = 360 * (date2.year() - date1.year()).count()
        + 30 * (date2.month() - date1.month()).count() + (d2 - d1);

    return diff / 360.0;
}
```

Copyright © 2022 Daniel Hanson

Dates: Other Common Operations

- Check if weekend
- Check if leap year
- Add days/months/years

Dates: Weekends

- Check if a date is a weekend

```
// Check if weekend: sys_days = days since epoch
date::year_month_day ymd1(date::year(2002), date::month(11), date::day(14));  // weekday
date::year_month_day ymd3(date::year(2022), date::month(2), date::day(13));   // weekend

// Again, need to convert to sys_days type:
auto is_weekend = [](date::sys_days t)->bool
{
    const date::weekday wd{ t };
    return wd == date::Saturday || wd == date::Sunday;
};

auto wd1 = is_weekend(date::sys_days(ymd1));   // false
auto wd3 = is_weekend(date::sys_days(ymd3));   // true
```

Dates: Add Days/Months/Years

- Would like something like the following, where 10 is an integer type

```
date::year_month_day ymd1(date::year(2002), date::month(11), date::day(14));
ymd1.add_days(10);
```

- Not so simple: Need to convert to **sys_days** again, and then
  - Add a days *object*
  - Add a months *object*, or
  - Add a years *object*

```
auto temp1 = date::sys_days(ymd1) + date::days(3);
auto temp2 = date::sys_days(ymd1) + date::months(1);
auto temp3 = date::sys_days(ymd1) + date::years(19);
```

- Results:

```
2002-11-17
2002-12-14 10:29:06 – Different object than adding days
2021-11-13 14:34:48 – Date result incorrect! 2021-11-13 14:34:48
```

Copyright © 2022 Daniel Hanson

Dates: Add Days/Months/Years

- Addition assignment, however, is correct
- Note also we don't have to convert to **sys_days** here

```
ymd1 += date::years(19);
```

- Result:

```
2021-11-14 – Correct, and is still a year_month_day object too
```

- Upshot seems to be: use addition assignment when possible

Dates: Add Days/Months/Years

- Also, adding months to end of month does not preserve end of month

```
Add 1 month to end of month 2022-04-30: 2022-05-30 10:29:06

Add 11 months to end of month 2022-04-30: 2023-03-30 19:20:06
```

- There is a **`date::year_month_day_last`** class
  - Only seems to *represent* a date of the form yyyy-mm-**`date::last`**
  - Does not provide the numerical value of last day (28, 29, 30, 31)
  - Would be nice if it did

XLDate

- Work in progress – nearly done
- Wraps the C++20 date library where applicable
- Uses addition assignment for adding days/months/years
- Has an end-of-month check
- Adding months preserves end-of-month
- Integer serial date uses 1900-01-01 epoch like Excel

XLDate

- Public Functions

```
XLDate();
XLDate(unsigned serialDate);
XLDate(unsigned year, unsigned month, unsigned day);

XLDate& addYears(int years);
XLDate& addMonths(int months);
XLDate& addDays(int days);

unsigned daysInMonth() const;
unsigned dayOfWeek() const;

bool endOfMonth() const;
bool leapYear() const;
bool weekday() const;

unsigned year() const;
unsigned month() const;
unsigned day() const;
unsigned serialDate() const;

void setYear(unsigned year);
void setMonth(unsigned month);
void setDay(unsigned day);
void setSerialDate(unsigned serialDate);
```

```
unsigned operator - (const XLDate& rhs) const;

XLDate& operator += (int days);
XLDate& operator -= (int days);

XLDate& operator ++ ();
XLDate& operator -- ();

XLDate operator ++ (int notused);
XLDate operator -- (int notused);

bool operator == (const XLDate& rhs) const;
bool operator != (const XLDate& rhs) const;
bool operator < (const XLDate& rhs) const;
bool operator > (const XLDate& rhs) const;
bool operator <= (const XLDate& rhs) const;
bool operator >= (const XLDate& rhs) const;
```

That's All!

- Woo-hoo!