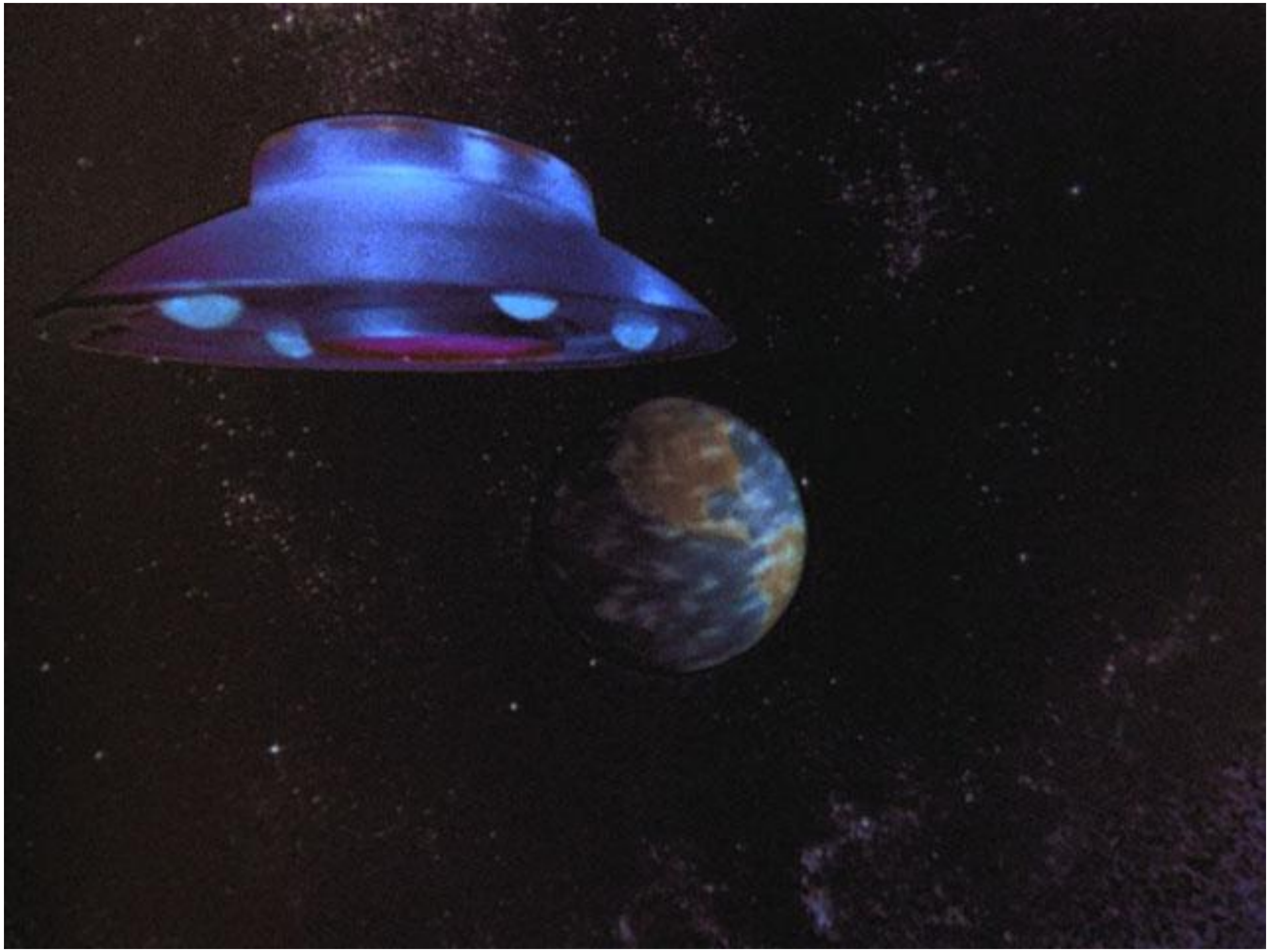
Back to Basics
The Three-Way Comparison Operator
(The Spaceship Operator <=>)



Daniel Hanson, NWCPP 17 April 2025

Overview

- C++20 brought us the three-way, or “spaceship”, operator <=>
- Not too long ago, it was necessary to declare and implement:

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

- These can now be replaced with a single operator†

```
ordering_type operator <=>(const Blah& rhs) const;
```

Overview

- Also with C++20, defining the equality operator now automatically implies the **!=** operator

- If you define equality:

```
bool operator ==(const Blah& rhs) const;
```

- You also get

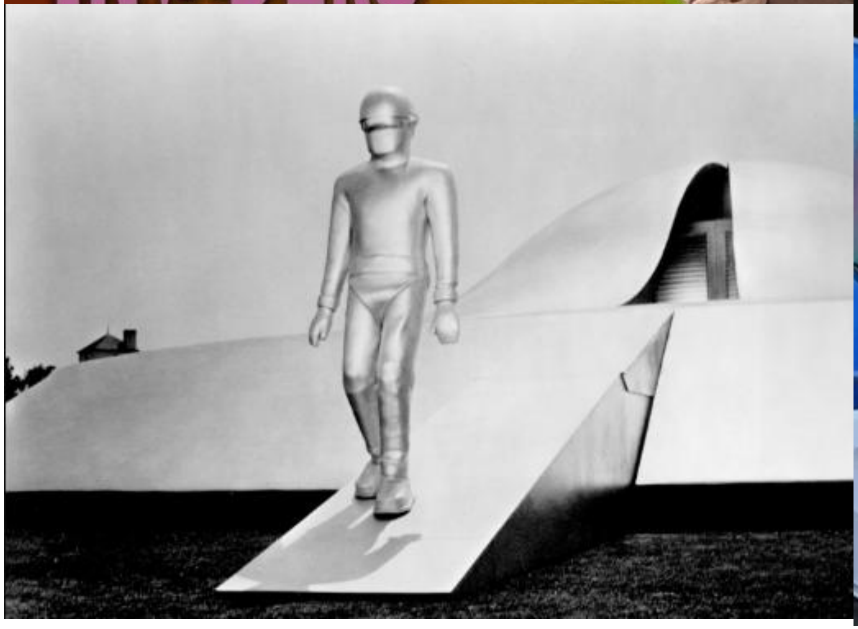
```
bool operator !=(const Blah& rhs) const;
```

for free

- † There may also be cases where you will want a separate **operator ==** defined, in addition to **operator <=>**

Why “the Spaceship” Operator?

- <=> kind of looks like a flying saucer



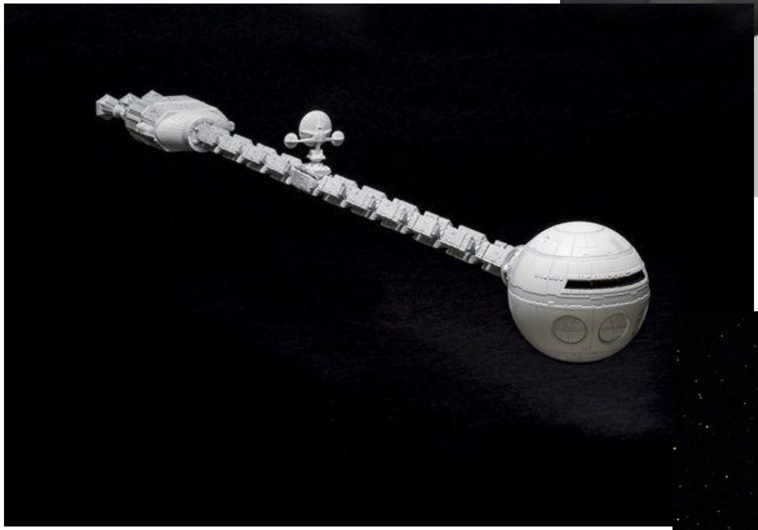
- Some rejected proposals:

`>=<`

`>=>`

`>===|=====0`

`<==V===7`



Typical Textbook Example

```
export module Point;
import <iostream>;
export class Point

{

public:

    Point(int x, int y, int z) : x_{x}, y_{y}, z_{z} {}

    auto operator <=>(const Point& rhs) const = default;

    friend std::ostream& operator<<(std::ostream& os, const Point& p);


private:

    int x_, y_, z_;
};


export std::ostream& operator<<(std::ostream& os, const Point& p)
{
    return os << "(" << p.x_ << ", " << p.y_ << ", " << p.z_ << ") ";
}
```

Typical Textbook Example

- The default is a lexicographical comparison…

```
Point class examples:
(1, 0, 6)  (8, 7, 4)  (4, 1, 9)

(1, 0, 6) == (8, 7, 4) ? false
(8, 7, 4) != (4, 1, 9) ? true
(4, 1, 9) == (4, 1, 9) ? true


(1, 0, 6) <= (8, 7, 4) ? true
(8, 7, 4) > (4, 1, 9) ? true
(4, 1, 9) < (4, 2, 5) ? true
```

- End of textbook example

- Move on to the next chapter on unique pointers or something…

Something a Little More Illustrative

- A Fraction class
- Assume for simplicity
  - Numerator and denominator are non-negative integers
  - Denominator is never zero
  - Fraction is simplified at construction

```
export class Fraction
{
public:
    Fraction(int n, int d) :n_{n}, d_{d}
    {
        // Assume numerator non-negative and numerator strictly positive:
        assert(!(n < 0) && !(d <= 0));

        // Simplify at construction...
    }

    // Old Way:
    bool operator ==(const Fraction& rhs) const;
    bool operator !=(const Fraction& rhs) const;
    bool operator <(const Fraction& rhs) const;
    bool operator >(const Fraction& rhs) const;
    bool operator <=(const Fraction& rhs) const;
    bool operator >=(const Fraction& rhs) const;
    // ...
};
```

Default form of <=>

- Proceed naively using the (textbook) default:

```
export class Fraction
{
public:
    Fraction(int n, int d) :n_{n}, d_{d}
    {
        // . . .
    }

    // Proceed naively:
    bool operator <=>(const Fraction& rhs) const = default;

    friend std::ostream& operator<<(std::ostream& os, const Fraction& p);
    // . . .
};


Fraction frac_01{1, 2}, frac_02{1, 5};
cout << frac_01 << "< " << frac_02 << "? " << (frac_01 < frac_02) << "\n";
```

```
1/2 < 1/5 ? true
```

- This is obviously wrong

Fraction Class: Implement <=>

- Replace the six comparison operators with two:

```
bool operator ==(const Fraction& rhs) const = default;

std::strong_ordering operator <=>(const Fraction& rhs) const
{
    if (n_ * rhs.d_ < rhs.n_ * d_)
    {
        return std::strong_ordering::less;
    }
    else if (*this == rhs)  // `==` defined with `default` above
    {
        return std::strong_ordering::equivalent;
    }
    else
    {
        return std::strong_ordering::greater;
    }
}
```

Fraction Class: Implement <=>

- Now, repeat the same check:

```cpp
Fraction frac_01{1, 2}, frac_02{1, 5};
cout << frac_01 << "< " << frac_02 << "? " << (frac_01 < frac_02) << "\n";
```

```
1/2 < 1/5 ? false
```

- Now we're cooking with dilithium crystals…

Ordered return values: `less`, `greater`, `equivalent`

```
auto comp = x <=> y
```

Possible return values for `comp` (compare) are:

```
std::strong_ordering::greater (x > y)

std::strong_ordering::less (x < y)

std::strong_ordering::equivalent (x == y)
```

less, greater, equivalent and Ordering Type:

- These can be illustrated in this simple example:

```cpp
auto compare_type = [](const Fraction& frac_lhs, const Fraction& frac_rhs,
    const std::strong_ordering& comp) {
        cout << "Comparison result: " << frac_lhs << " <=> " << frac_rhs << "\n";
        if (comp == std::strong_ordering::less) {
            cout << "strong_ordering::less" << "\n";
        }
        else if (comp == std::strong_ordering::greater) {
            cout << "strong_ordering::greater" << "\n";
        }
        else {
            cout << "strong_ordering::equivalent" << "\n";
        }
  };
auto comp = frac_01 <=> frac_02;        // 1/2, 1/5
cout << "Type of comp: " << typeid(comp).name() << "\n";
compare_type(frac_01, frac_02, comp);

comp = frac_02 <=> frac_01;
compare_type(frac_02, frac_01, comp);

Fraction frac_03{1, 2};
comp = frac_01 <=> frac_03;
compare_type(frac_01, frac_03, comp);
```

```
Type of comp: struct std::strong_ordering
Comparison result: 1/2  <=> 1/5
strong_ordering::greater
```

```
Comparison result: 1/5  <=> 1/2
strong_ordering::less
```

```
Comparison result: 1/2  <=> 1/2
strong_ordering::equivalent
```

Three ordering types can be returned from the <=> operator

`std::strong_ordering`

- Use case: exact comparison, such as with integer types (no rounding, NaN's…)
- Return values are **`greater`**, **`less`**, and **`equivalent`**
- Exactly one of **`a < b`**, **`a == b`**, or **`a > b`** must be **`true`**

`std::partial_ordering`

- Return values also include **`greater`**, **`less`**, and **`equivalent`**
- Use case: with floating point types such as **`double`** and **`float`**
- A floating type can
  - Have rounding error
  - hold non-comparable assignments such as infinity and NaN
- Exactly one of **`a < b`**, **`a == b`**, or **`a > b`** must be **`true`**
- **`<=>`** returns **`std::partial_ordering::unordered`** if both variables are NaN

`std::weak_ordering`

- Possible for **`a < b`**, **`a == b`**, and **`a > b`** to all be **`false`**
- Can (purportedly) be used for cases such as comparing case-insensitive strings
- However, the default return type for **`std::string`** comparisons is **`std::strong_ordering`**

Ordering Types: Default Examples

```cpp
int m = 0, n = 1;
double x = 1.06, y = 8.74;
std::string bass = "Rickenbacker", BASS = "RICKENBACKER";

auto int_compare = m <=> n;
auto real_compare = x <=> y;
auto str_compare = bass <=> BASS;

cout << format("int compare type: {}\nfloating compare type: {}\nstring compare type: {}\n",
typeid(int_compare).name(), typeid(real_compare).name(), typeid(str_compare).name());
```

```
int compare type: struct std::strong_ordering
floating compare type: struct std::partial_ordering
string compare type: struct std::strong_ordering
```

Ordering Values: `equal` vs `equivalent`

There is also an **`equal`** return value

`std::strong_ordering::equal`

- Identical to **`std::strong_ordering::equivalent`**
- Means two values have exactly the same bitwise representation
- Perfectly valid for integer values (e.g., in our **`Fraction`** class)

`std::partial_ordering::equal`

- Again, means two values have exactly the same bitwise representation
- Becomes meaningless with roundoff error, NaN's, etc
- Also identical to **`std::partial_ordering::equivalent`**
- But, it indicates the intent…
  - Floating type values should never be tested for exact equality
  - Valid floating type values can be defined to be equivalent within a given tolerance (as will be seen shortly)

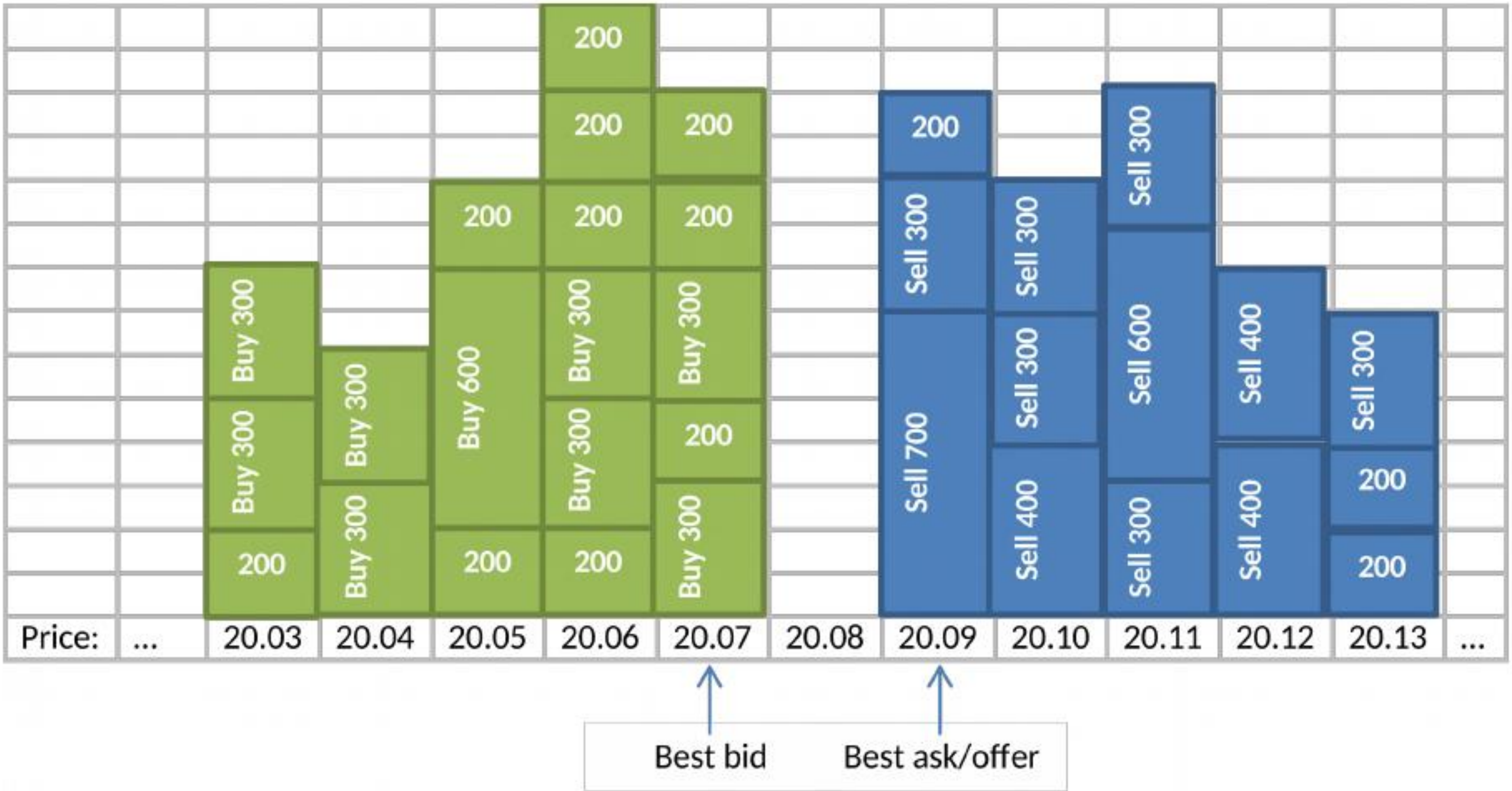
Personal preference: Always use **`equivalent`**, even with **`strong_ordering`**, to avoid possible confusion

An Example from Financial Equity Trading

- A limit order for buying or selling shares of a stock
  - An order to buy or sell a desired number of shares at a specified limit price or better
  - Ensures execution will not be at price worse than the pre-specified price
    - ➢ Buy order will not be executed above the limit price
    - ➢ Sell order will not be executed below the limit price
  - Limit orders are placed in an *order book* on an exchange

An Order Book

- An example of an order book consists of limit orders as shown†:

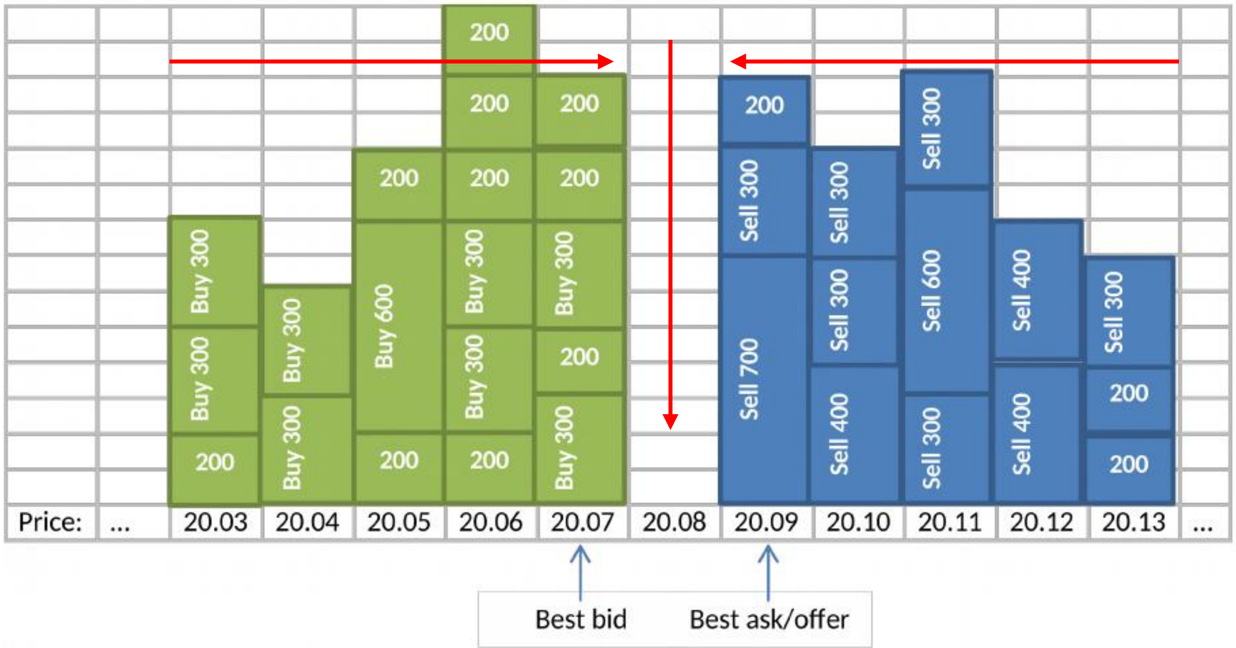


- Orders are queued by priority (to be discussed next)

____

†Doug Service, 2016

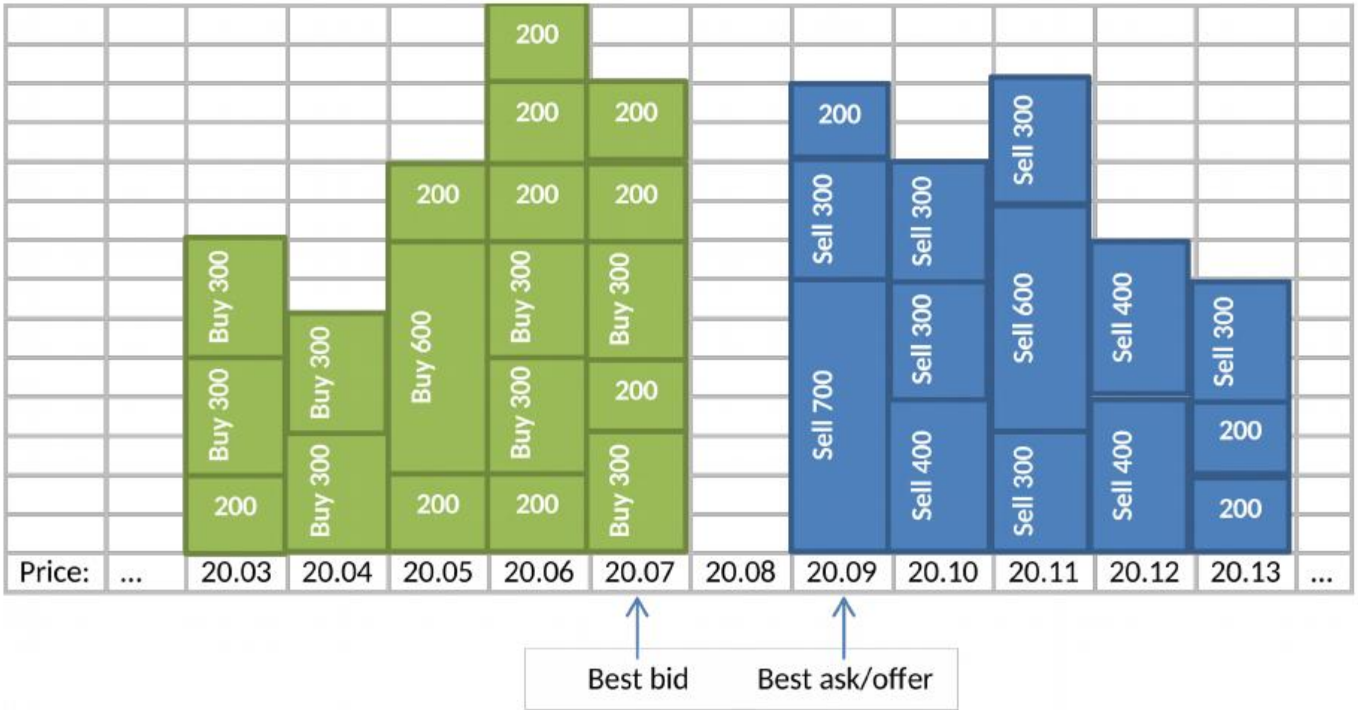
Queuing Priority (a Simplified Example)



- Given two limit orders *a* and *b*, assume that both are of like type
  - Buy (bid) or
  - Sell (offer)
- An order to <u>buy</u>, *a*, will have priority over *b*, if
  - The limit price of a is greater than that of b
  - If equal, then if *a* is a round lot (multiple of 100 shares) and *b* is not
  - If the same, then if order a has an earlier timestamp, it has priority
  - Otherwise (to be considered unlikely in this example), the two are equivalent
  - Similar for an order to sell, except the limit value of *a* is less than that of *b*

Order a Buy 200 @20.07	Order b Buy 200 @20.04
Order a Buy 200 @20.07	Order b Buy 125 @20.07
Order a Buy 200 @20.07	Order b Buy 200 @20.07

Queuing Priority (a Simplified Example)



- Instead of *a* < *b* and *b* > *a* meaning less than or greater than,
- We will take them to indicate priority
- *b* < a => *a* is has priority in the order book over *b*

A LimitOrder Class

- Start with an enum class indicating a buy or sell limit order:

```
module;

#include <cassert>

export module LimitOrder;
import std;

export enum class Buy_or_Sell
{
    Buy,
    Sell
}
```

A `LimitOrder` Class

- For private data members, we will have:

```
export class LimitOrder
{
   // . . .
private:
    Buy_or_Sell buy_or_sell_;
    double price_;                // Bid (buy) or offer (sell) price
    int num_shares_;
    std::chrono::time_point<std::chrono::system_clock> time_stamp_;
    inline static const double epsilon_ =
        std::sqrt(std::numeric_limits<double>::epsilon());
    inline static int trade_id_ = 0;    // Will be incremented
    // . . .
```

- Use a **std::chrono::time_point** for the time stamp (simple approach)
- The epsilon value follows the Boost convention for comparing two real (**double**) values

A `LimitOrder` Class

- At construction:

```
public:
    LimitOrder(Buy_or_Sell buy_or_sell, double price, int num_shares,
        std::chrono::time_point<std::chrono::system_clock> time_stamp) :
        buy_or_sell_{buy_or_sell}, price_{price}, num_shares_{num_shares},
        time_stamp_{time_stamp}
    {
        assert(num_shares > 0);
        assert(price > 0.0);
        ++trade_id_;
    }
```

  - The members are initialized
  - Enforce the constraint that the number of shares and bid/offer price are > 0
  - Increment the trade ID (also simplified – integer value)

A `LimitOrder` Class

- Implementation of **<=>** Step 1

```
std::partial_ordering operator <=>(const LimitOrder& rhs) const
{
    assert(this->buy_or_sell_ == rhs.buy_or_sell_);
    // First: Compare prices first, but flip depending on Buy or Sell
    std::partial_ordering cmp_price = (buy_or_sell_ == Buy_or_Sell::Buy)
        ? (this->price_ <=> rhs.price_)   // Case: Bid order  (higher price wins)
        : (rhs.price_ <=> this->price_);  // Case: Sell order (lower price wins)

    // Check if the absolute difference between the prices is within epsilon;
    // if not, return the result of the <=> operator for the two order prices
    if (std::abs(this->price_ - rhs.price_) > epsilon_)
    {
        return cmp_price;
    }

    // More to follow . . .
}
```

Make sure both trades are in the same direction (buy or sell)

- Use <=> default as applied to double types; for example:
- If an order to buy, this->price < rhs.price_ => this < rhs in priority
- If an order to sell, rhs.price_ < this->price => this < rhs in priority
- Returns std::partial_ordering::less in each case

Make sure the difference between the two limit prices is greater than the tolerance defining equivalence

Return either partial_ordering::less or partial_ordering::greater

A LimitOrder Class

- Implementation of **<=>** Step 2

```
std::partial_ordering operator <=>(const LimitOrder& rhs) const
{
    // Continued from previous slide . . .

    // Second: Prefer round lots (multiples of 100 shares)
    bool this_round_lot = (this->num_shares_ % 100) == 0;
    bool rhs_round_lot = (rhs.num_shares_ % 100) == 0;

    if (this_round_lot != rhs_round_lot)
    {
        return this_round_lot ? std::partial_ordering::greater : std::partial_ordering::less;
    }

    // More to follow . . .
}
```

Check each order, whether it is a round lot of 100 shares or not

Only in the case where one is a round order and the other is not, return a result, greater if *this is a round order, less if rhs is.

A LimitOrder Class

- Implementation of **<=>** Step 3 (last)

```
std::partial_ordering operator <=>(const LimitOrder& rhs) const
{
    // Continued from previous slide . . .

    // Third (and final comparison criterion): Earlier timestamp takes priority,
    // and hence is "greater", i.e., higher in priority
    if (this->time_stamp_ < rhs.time_stamp_) return std::partial_ordering::greater;
    else if (this->time_stamp_ > rhs.time_stamp_) return std::partial_ordering::less;

    // Assume for this example almost surely not to occur . . .
    else return std::partial_ordering::equivalent;      // Also the overall default
}
```

This is another case where a lower value (earlier timestep) implies an object is “greater”, and vice versa

In this simple example, we’re assuming this default case will almost surely not occur, but return equivalent if it is reached

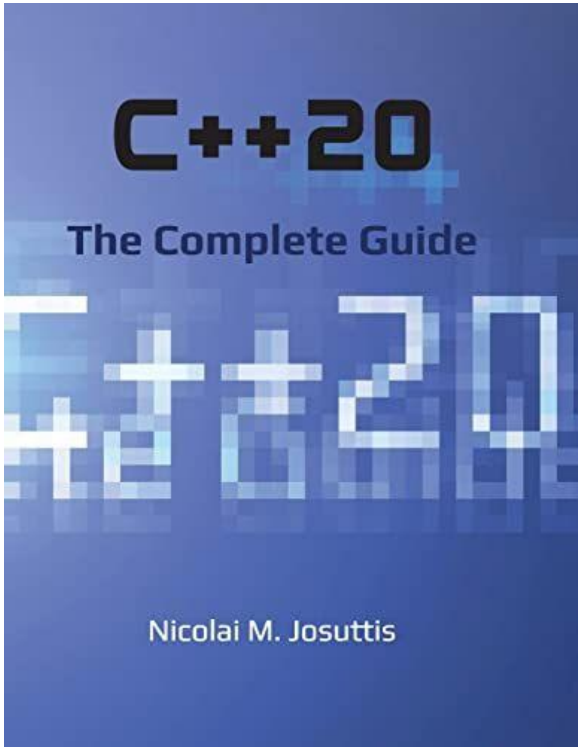
Summary

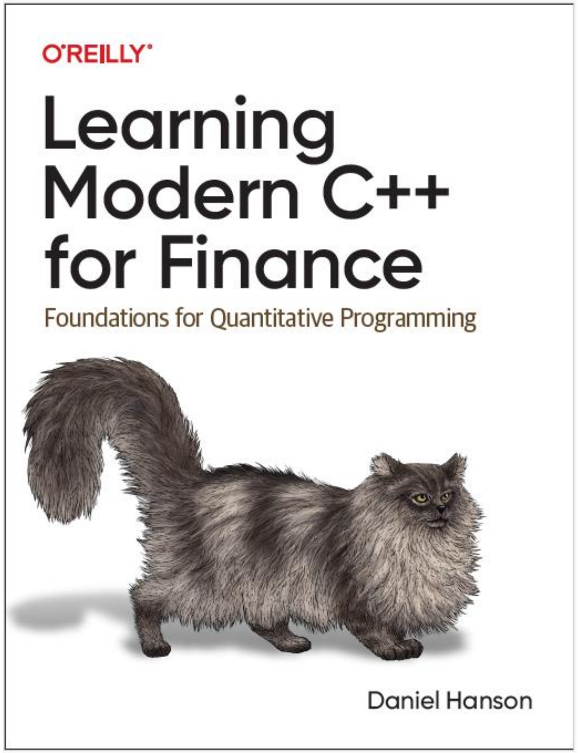
- The **<=>** operator allows us to incorporate all six comparison operators into a single operator implementation

- There can also be cases where a separate **==** operator definition is desired (as we saw in the Fraction class)
  - Can use as the condition for **`std::xxxx_ordering::equivalent`** in **<=>**
  - Note this also gives you **!=** for free (no need to define separately)
  - There is also a default option for the == operator

- A default option is also available for **<=>**
  - Lexicographic comparison of data members on two objects
  - Will not always suffice
  - Coverage in textbooks and references is often limited to this case

- Nicolai Josuttis, *C++20: The Complete Guide* (LeanPub), Ch 1
- Hanson, *Learning Modern C++ for Finance* (O'Reilly, November 2024), Ch 2 (shameless plug)





- cppreference.com

Contact/ Questions

- Contact:
  - `daniel (at) cppcon.org` (Student Program Coordinator, CppCon)
  - https://www.linkedin.com/in/danielhanson/
- Thank You!
- Questions?