Correctly Calculating min, max, and More

What Can Go Wrong?

Walter E. Brown, Ph.D.

< webrown.cpp @ gmail.com >

Edition: 2021-12-09. Copyright © 2020-2021 by Walter E. Brown. All rights reserved.

A little about me

- B.A. (math's); M.S., Ph.D. (computer science).
- Professional programmer for over 50 years, programming in C++ since 1982.
- Experienced in industry, academia, consulting, and research:
  - Founded a Computer Science Dept.; served as Professor and Dept. Head; taught and mentored at all levels.
  - Managed and mentored the programming staff for a reseller.
  - Lectured internationally as a software consultant and commercial trainer.
  - Retired from the Scientific Computing Division at Fermilab, specializing in C++ programming and in-house consulting.
- Not dead — still doing training & consulting. (Email me!)

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 4

Emeritus participant in C++ standardization

- Written ~170 papers for WG21, proposing such now-standard C++ library features as gcd/lcm, cbegin/cend, common_type, and void_t, as well as all of headers <random> and <ratio>.
- Influenced such core language features as *alias templates*, *contextual conversions*, and *variable templates*; recently worked on *requires-expressions*, operator<=>, and more!
- Conceived and served as Project Editor for *Int'l Standard on Mathematical Special Functions in C++* (ISO/IEC 29124), now incorporated into <cmath>.
- Be forewarned: Based on my training and experience, I hold some rather strong opinions about computer software and programming methodology — these opinions are not shared by all programmers, but they should be! ☺

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 5

The Big Picture

The study of error … serves as a stimulating introduction to the study of truth.

— *Walter Lippmann*

Copyright © 2020-2021 by Walter E. Brown. All rights reserved.

About this talk

- The C++ standard library long ago selected operator < as its ordering primitive, and even spells it in several different ways (*e.g.*, std::less).
- Today, we will first illustrate why operator < (no matter its spelling) must be used with care, in even seemingly simple algorithms such as max and min.
- Then we will discuss the use of operator < in other order-related algorithms, showing how easy it is to make mistakes when using the operator < primitive directly, no matter how it's spelled.
- Along the way, we will also present a straightforward technique to help us avoid such mistakes.

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 7

"One of the amazing things which we … discover is that ordering is very important.

Things which we could do with ordering cannot be effectively done just with equality."

— *Alexander Stepanov* (né *Алекса́ндр Степа́нов*)

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 8

Copyright © 2020-2021 by Walter E. Brown.
All rights reserved.

Early Attempts

Life is trying things to see if they work.
— Ray Bradbury

Copyright © 2020-2021 by Walter E. Brown. All rights reserved.

An intuitive approach ①

- As C-style function-like macros:
  - `#define MIN(a, b)   ( (a) < (b) ? (a) : (b) )`
  - `#define MAX(a, b)   ( (b) < (a) ? (a) : (b) )`
- Repackaged, now as functions (with one overload/type):
  - `int min ( int a, int b ) { return a < b ? a : b; }`
  - `int max ( int a, int b ) { return b < a ? a : b; }`
- Lifted, now as simple (C++20) function templates:
  - `auto min ( auto a, auto b ) { return a < b ? a : b; }`
  - `auto max ( auto a, auto b ) { return b < a ? a : b; }`

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 10

An intuitive approach ②

- But those C++ templates ...
  - `auto min ( auto a, auto b ) { return a < b ? a : b; }`
  - `auto max ( auto a, auto b ) { return b < a ? a : b; }`

... have a few issues:

✗ The by-value parameter passage is potentially expensive (*e.g.*, for large string arg's).

✗ When the arguments have distinct types, it's unclear what the return type should be. (Can we even compare such arg's generically? *E.g.*, consider signed vs. unsigned [forthcoming!])

✗ Major concern: are the algorithms correct for all values?

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 11

The cures are mostly straightforward

- Per the std library's specification:
  - ✓ Enforce consistent types via a named type parameter.
  - ✓ Avoid expensive copies via call/return by ref-to-const.
- After these adjustments we have:
  - ```
    template< class T >
    T const &
      min( T const & a, T const & b ) { return a < b ? a : b; }
    ```
  - And analogously for max.
- Just recall that lvalue ref's to rvalues can be subtle:
  - ✓ `auto    z = min( x.calc( ), y.calc( ) );  // copies a temporary`
  - ✗ `auto & r = min( x.calc( ), y.calc( ) );  // dangling reference!`

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 12

So What's Wrong?

[N]ever feel badly about making mistakes ...
as long as you ... learn from them.
— Norton Juster

Copyright © 2020-2021 by Walter E. Brown. All rights reserved.

Alas, none of the code I've shown so far is right!

- Can you identify the misbehaviors?
  - ```
    template< class T >
    T const &
      min ( T const & a, T const & b ) { return a < b ? a : b; }
    ```
  - ```
    template< class T >
    T const &
      max( T const & a, T const & b ) { return b < a ? a : b; }
    ```
- Did you notice that each returns b when a == b?
  - Why should max and min of the same two arguments ever give the same result?
  - ("It took Stepanov 15 years to get min and max right.")

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 16

Copyright © 2020-2021 by Walter E. Brown.
All rights reserved.
    ```

To be specific, ...

- ... these algorithms mishandle the case of a == b!
  - "[At] CppCon 2014, Committee member Walter Brown mentioned that [std] max returns the wrong value [when] both arguments have an equal value. ...
  - "*Why should it matter which value is returned?*"
- Many programmers have made similar observations:
  1. That equal values are indistinguishable, so ...
  2. It ought not matter which is returned, so ...
  3. This is an uninteresting case, not worth discussing.
- Alas, for min and max (and related) algorithms, such opinions are superficial and incorrect!

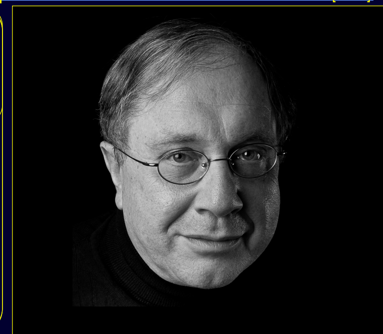
Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 17

Alex Stepanov speaks of his mistake [2013]

shame will be publicly visible.”

spends decades working on all these orderings, and writing min in the most generic way, and then he writes max and he screws it up!

Oh, no. People will remember for centuries because that’s the max in the standard library!



Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 18

Many types do distinguish equal values

- Bare-bones example:
  - ```
    struct student {
      string name; int id;
      inline static int registrar = 0;
      S( string n ) : name{ n }, id{ registrar++ } { }   // c'tor
      bool
        operator < ( student s )
      { return name < s.name; }              // id is not salient
    };
    ```
- Since each student variable has a unique id number:
  - Even equal values are distinguishable, so ...
  - It can matter greatly which one is returned by min/max!

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 19

How Do We Address This?

[O]nly wise men learn from their mistakes.

— Winston Churchill

Copyright © 2020-2021 by Walter E. Brown. All rights reserved.

A mathematics perspective

- A monotonically increasing sequence is sorted:
  - But not conversely!
  - Counterexample: a sequence of identical values is sorted, but is certainly not monotonically increasing.
- Instead, we must say:
  - That a sequence is sorted iff it is non-decreasing.
  - This allows us to have equal items in a sorted sequence.
- C++ embraces this viewpoint (see [alg.sorting.general]/5):
  - A sequence is *sorted* if, for every iterator i and non-negative integer n, ∗(i + n) < ∗i is false.

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 21

An important insight

- Given two values a and b, in that order:
  - Unless we find a reason to the contrary, ...
  - **min should prefer to return a,** and ...
  - **max should prefer to return b.**
- *I.e.*, never should max and min return the same item:
  - When values a and b are in order, min should return a / max should return b; ...
  - When values a and b are out of order, min should return b / max should return a.

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 22

Copyright © 2020-2021 by Walter E. Brown.
All rights reserved.
    ```

Even more succinctly stated

- We should always prefer algorithmic stability …
  - … especially when it costs nothing to provide it!
- Recall what we mean by stability:
  - An algorithm dealing with items' order is stable …
  - If it keeps the original order of equal items.
- *I.e.*, a stable algorithm ensures that:
  - For all pairs of equal items a and b, …
  - a will precede b in its output …
  - Whenever a preceded b in its input.

Copyright © 2020-2021 by Walter E. Brown. All rights reserved.

Therefore, I recommend …

- For min:
  - `… { return out_of_order(a, b) ? b : a; } // in_order ? a : b`
- For max: "Is there a reason to do otherwise?"
  - `… { return out_of_order(a, b) ? a : b; } // in_order ? b : a`
- Where:
  - ```
    inline bool
      out_of_order( … x, … y ) { return y < x; } // !!!
    ```
  - ```
    inline bool
      in_order( … x, … y ) { return not out_of_order(x, y); }
    ```
  - FWIW, in my experience, out_of_order is the more useful.

Copyright © 2020-2021 by Walter E. Brown. All rights reserved.

These Ideas Are Broadly Applicable

[The] principle, by which each slight variation, if useful, is preserved, [I have termed] Natural Selection.

— Charles Darwin

Copyright © 2020-2021 by Walter E. Brown. All rights reserved.

Analogous logic also applies elsewhere ①

```
template< input_iterator In, output_iterator<In> Out >
Out merge( In b1, In e1   // 1st sorted input sequence
         , In b2, In e2   // 2nd sorted input sequence
         , Out to ) {     // merged destination

  while( true )
    if      ( b2 == e2 ) return copy( b1, e1, to );
    else if ( b1 == e1 ) return copy( b2, e2, to );
    else  // assert: neither sequence is empty
      *to++ = out_of_order(*b1, *b2) ? *b2++
                                     : *b1++;
}
```

"Prefer to take from the 1st sequence; need a reason to take from the 2nd."

Copyright © 2020-2021 by Walter E. Brown. All rights reserved.

Analogous logic also applies elsewhere ②

- ```
  template< class T >
  void sort2( T & a, T & b ) {
    if( out_of_order(a, b) )
      swap(a, b);
  } // postcondition: in_order(a, b)
  ```
  ```
  if( in_order(a, b) ) return;
  swap(a, b);
  ```
- ```
  template< class T > // C++20
  void sort3( T & a, T & b, T & c ) {
    if( sort2(a, b); in_order(b, c) ) return;
    if( swap(b, c); in_order(a, b) ) return;
    swap(a, b);
  }
  ```
  - (Did you recognize bubble sort?)

Copyright © 2020-2021 by Walter E. Brown. All rights reserved.

Algorithm logic from stackoverflow — is this correct?

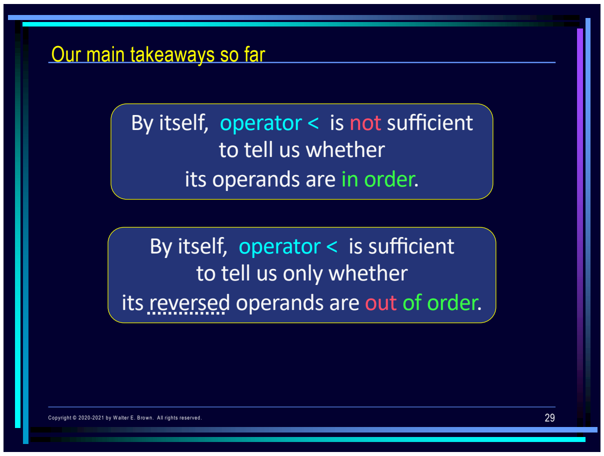
- ```
  template< class T >
  void sort3( T & a, T & b, T & c ) {
    if( a < b ) {
      if( b < c ) return;
      else if( a < c ) swap(b, c);
      else { /* rotate right into order c, a, b */ }
    }
    else {
      if( a < c ) swap(a, b);
      else if( c < b ) swap(a, c);
      else { /* rotate left into order b, c, a */ }
    }
  }
  ```

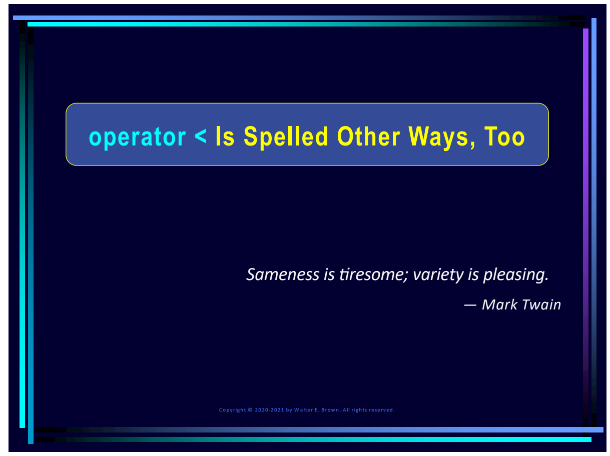
Algorithm does more work than necessary: operator < is no substitute for in_order!

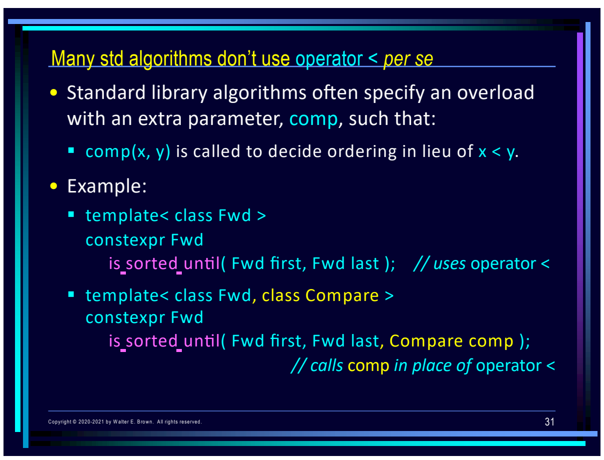
Algorithm isn't stable: operator < is no substitute for in_order!

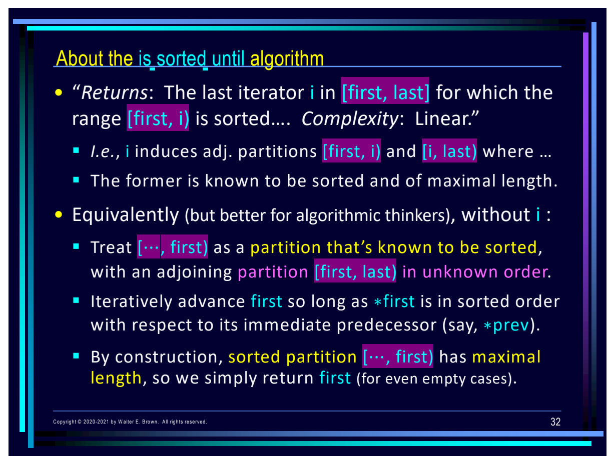
Copyright © 2020-2021 by Walter E. Brown. All rights reserved.

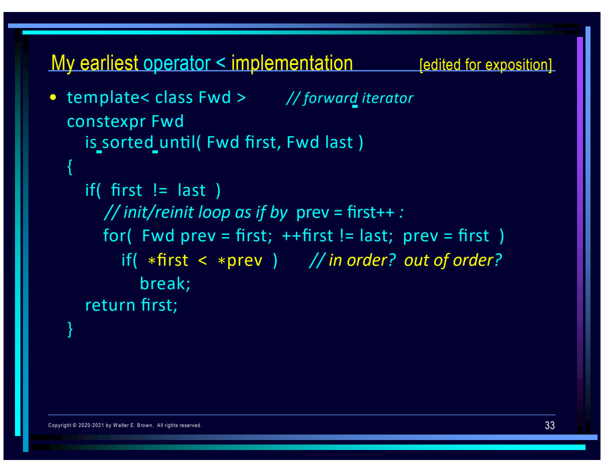
Copyright © 2020-2021 by Walter E. Brown.
All rights reserved.
  ```

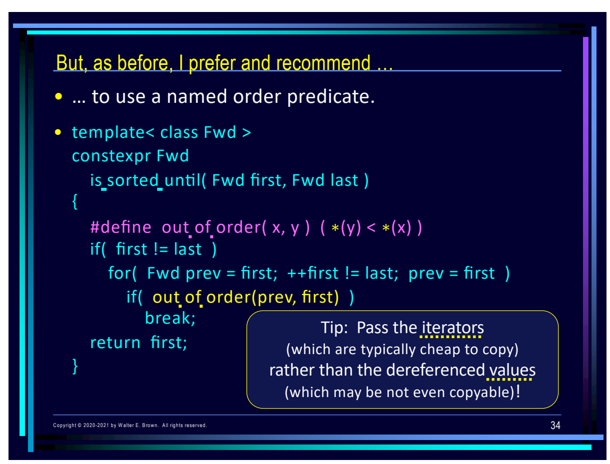












Copyright © 2020-2021 by Walter E. Brown.
All rights reserved.

[alg.sorting.general]/2-3 [rearranged]

- "[The declaration] Compare comp is used throughout [as a parameter that denotes] an ordering relation."
  - "Compare is a function object type [whose] call operation … yields true if the first argument of the call is less than the second, and false otherwise."
  - "… comp [induces] a strict weak ordering on the values."
  - "For all algorithms that take Compare, there is a version that uses operator < instead."
- IMO, the names comp and Compare are too general:
  - I'd prefer, *e.g.*, s/comp/less_than/ or s/comp/lt/ or s/comp/precedes/.

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 35

Even when we have an explicit less-than predicate …

- … I still recommend adapting it via an order predicate.
- ```
  template< class Fwd, class Compare >
  constexpr Fwd
    is_sorted_until( Fwd first, Fwd last, Compare precedes )
  {
    auto iter_out_of_order
      = [=] ( Fwd x, Fwd y ) { return precedes(*y, *x); };
    if( first != last )
      for( Fwd prev = first; ++first != last; prev = first )
        if( iter_out_of_order(prev, first) )
          break;
    return first;
  }
  ```

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 36

Or we can avoid overloading

- … via a single template that has judicious default arg's:
  - ```
    template< class Fwd, class Compare = std::ranges::less >
    constexpr Fwd
      is_sorted_until( Fwd first, Fwd last, Compare lt = { } )
    {
      ⋮  // unchanged
    }
    ```
- Q1: What, exactly, is std::ranges::less?
- Q2: Do we need both a default function argument and a default template argument?

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 37

Q1: What's std::ranges::less?

- It's a class declared in <functional>:
  - ```
    struct less {                        // simplified for exposition
      template< class T, class U >
      constexpr bool
        operator ( ) ( T && t, U && u ) const
      { return t < u; }                  // heterogeneous comparison
    };
    ```
  - A variable of type less is a function object, as it's callable via its operator ( ) member template.
- (There's also std::less, a template whose operator ( ) is strictly homogeneous [more later]. Many/most today seem to prefer the design of std::ranges::less.)

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 38

Q2: Do algorithms need both default argument kinds?

- Let's review the algorithm's decl., then consider a call:
  - ```
    template< class Fwd, class Compare = std::ranges::less >
    constexpr Fwd
      is_sorted_until( Fwd first, Fwd last, Compare lt = { } ) ;
    ```
  - ```
    int a[N] = { ⋯ };
    ⋯ is_sorted_until( a+0, a+N ) ⋯    // what type is Fwd?
    ```
  - Fwd is deduced as int ∗. Now: what type is Compare?
- It's std::ranges::less, per the default template arg:
  - (A type is never inferred from any default function arg.)
  - Enables calling code to default-construct a 3rd argument, namely std::ranges::less{ } .

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 39

Q3: Why doesn't the std library use such default arg's?

- In brief, because it's prohibited (unless thusly specified):
  - "An implementation shall not declare a non-member function signature with additional default arguments." (See [global.functions]/3.)
- Why not consolidate? Because doing so is problematic:
  - "The difference between two overloaded functions and one function with a default argument can be observed by taking a pointer to function." (See N1070, 1997.)
  - Further, consider a call with a type but without a value:
    ```
    template< class T = int > void g( T x = { } ) { ⋯ }
      ⋮
    g<MyType>( );  // what if MyType isn't default-constructible?
    ```

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 40

Copyright © 2020-2021 by Walter E. Brown.
All rights reserved.
    ```

std Disguises for operator <

Everybody's wearing a disguise....

— Bob Dylan

Copyright © 2020-2021 by Walter E. Brown. All rights reserved.

How many ways can std design and spell operator < ?

Name	Where found	Since	Arg. types
class template `less`	`<functional>`	C++98	T, T
specialization `less<void>`	`<functional>`	C++14	T, U
class `ranges::less`	`<functional>`	C++20	T, U
function template `cmp_less`	`<utility>` (why?)	C++20	integer I, J
overload set `isless`	`<cmath>`	C++11	arith A, B
specification `totalOrder`	IEEE 754; in spec of `<compare>`'s `strong_order`	2008; C++20	flt-pt F, F

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 42

My version of std::ranges::less [edited for exposition]

```
struct less
{
   template< class L, class R >
   constexpr bool
      operator( ) ( L && left, R && right ) const noexcept(···)
   {
      if constexpr( are_std_integer_types<L, R> )
         return cmp_less( left, right );          // forthcoming
      else if constexpr( are_std_arithmetic_types<L, R> )
         return isless( left, right );            // forthcoming
      else
         return forward<L>(left) < forward<R>(right);
   }
};
```

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 43

My version of std::cmp_less [edited for exposition]

```
template< std_integer_type L, std_integer_type R >
constexpr bool
   cmp_less( L left, R right ) noexcept
{
   if constexpr( same_signedness_types<L, R> )
      return left < right;
   else if constexpr( signed_type<L> ) // and unsigned_type<R>
      return left < 0 ? true : as_unsigned(left) < right;
   else // signed_type<R> and unsigned_type<L>
      return right < 0 ? false : left < as_unsigned(right);
}
```

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 44

My version of std::isless [edited for exposition]

```
template< std_arithmetic_type L, std_arithmetic_type R >
constexpr bool
   isless( L left, R right ) noexcept          // not an overload set
{
   using fl_t = common_floating_point_t<L, R>;
   fl_t x = left
      , y = right;
   return isunordered(x, y) ? false            // avoid FE_INVALID
                            : x < y;
}
```

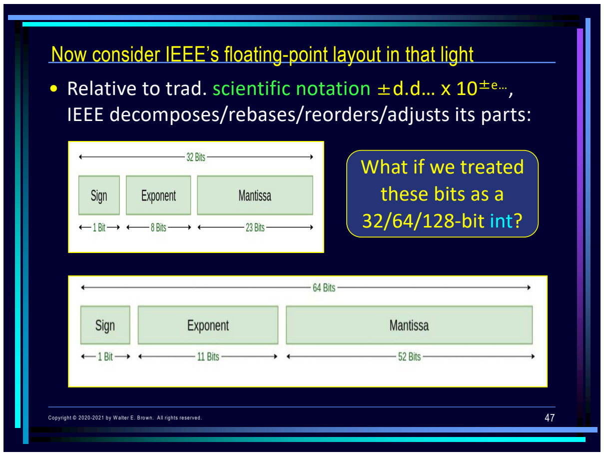
Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 45

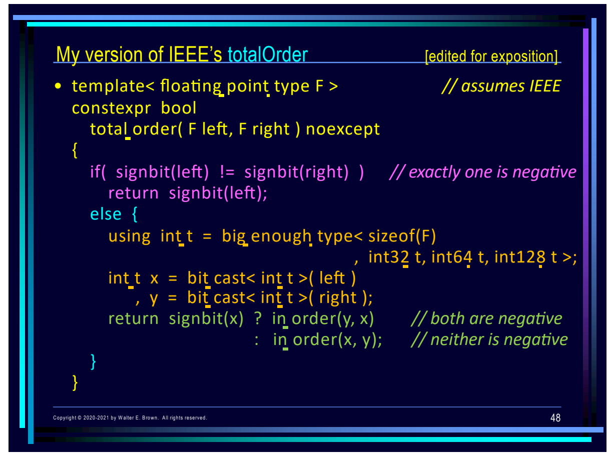
The ordering used by IEEE's totalOrder predicate

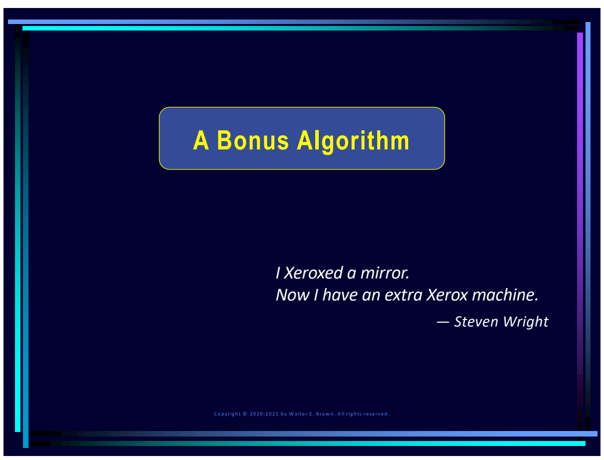
- Most- to least-negative, then least- to most-positive.
- *I.e.*, first all negative values, in the following order:
  - All negative quiet NaNs, then all negative signaling NaNs, each ordered per their payload bits.
  - Then negative infinity, then all negative normalized and denormal numbers in value order, then negative zero.
- Then all positive values, in the opposite order:
  - Positive zero, then all positive denormal and normalized numbers in value order, then positive infinity.
  - All positive signaling NaNs, then all positive quiet NaNs, each ordered per their payload bits.

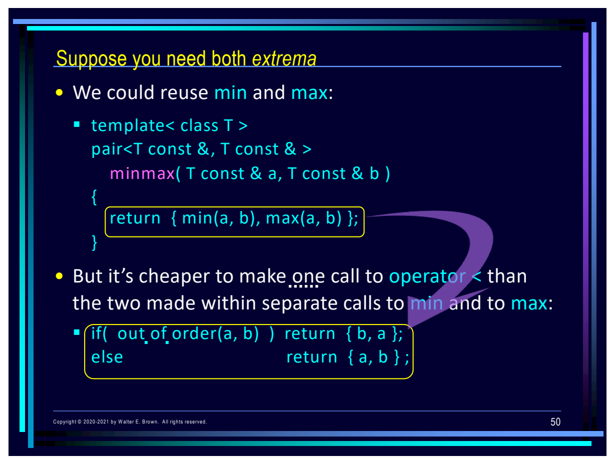
Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 46

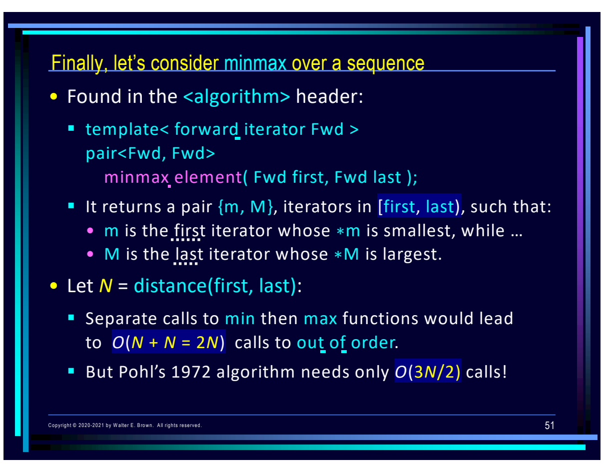
Copyright © 2020-2021 by Walter E. Brown.
All rights reserved.

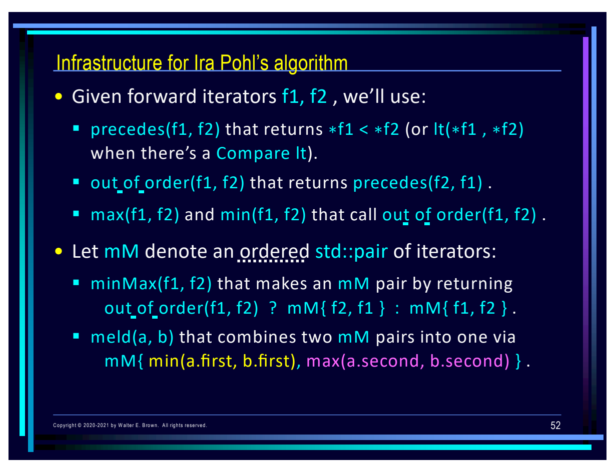




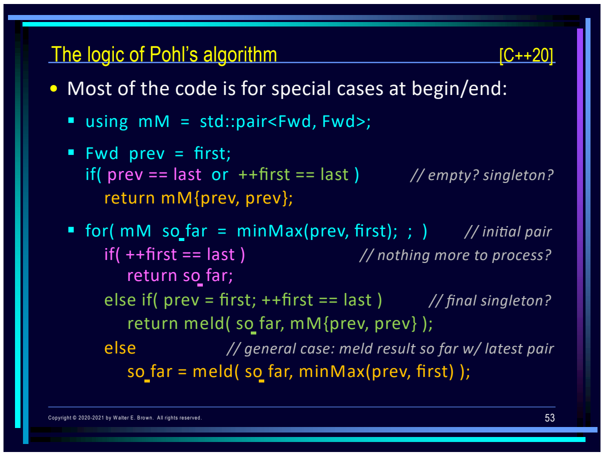


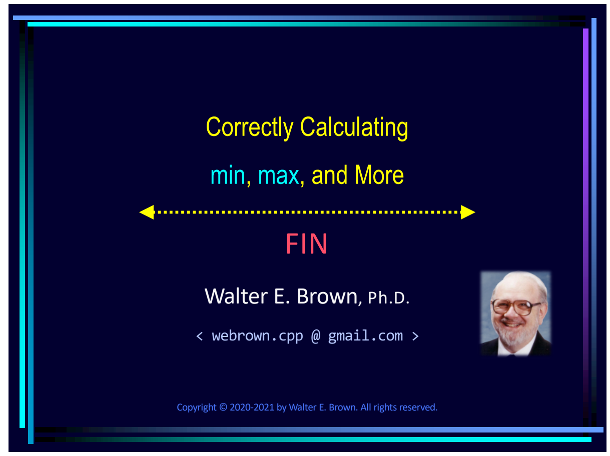






Copyright © 2020-2021 by Walter E. Brown.
All rights reserved.





Copyright © 2020-2021 by Walter E. Brown.
All rights reserved.