Welcome!

21

Correctly Calculating min, max, and More:
What Can Go Wrong?

WALTER E BROWN

Cppcon
The C++ Conference

2021 October 24-29

1

Sound check [London Fanfare Trumpets: *Flourish 3*]

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

2

Correctly Calculating min, max, and More

What Can Go Wrong?

Must Go Wrong!

Walter E. Brown, Ph.D.

< webrown.cpp @ gmail.com >

Edition: 2021-10-29. Copyright © 2020-2021 by Walter E. Brown. All rights reserved.

3

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. 5

5

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. 6

6

Introduction

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.

7

Today's 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).
- This talk will explain why operator < (and its aliases) must be used with care, in even seemingly simple algorithms such as max and min.
- We will also 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.
- (Of course, we will also present a straightforward technique to avoid such mistakes.)

 8

8

"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)

 9

9

First Attempts

Life is trying things to see if they work.

— Ray Bradbury

10

The intuitive approach ①

- As C-style macros:
  - `#define MIN (a, b) ( ((a) < (b)) ? (a) : (b) )`
  - `#define MAX (a, b) ( ((b) < (a)) ? (a) : (b) )`
- Repackaged, now as simple functions:
  - `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; }`

 11

11

The 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 can be expensive (*e.g.*, for large string arg's).
- ✗ When the arguments have distinct types, it's unclear what the return type should be. (It's even nonobvious how to compare them generically — *e.g.*, consider signed vs. unsigned!)
- ✗ Major concern: are the algorithms correct for all values?

 12

12

The cures are mostly straightforward

- ✓ Enforce consistent types via a named parameter type.
- ✓ 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; }
    ```
  - ```
    template< class T >
    T const &
      max( T const & a, T const & b ) { return b < a ? a : b; }
    ```

 13

13

So What's Wrong?

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

— Norton Juster

14

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.")

 15

15

To be specific, …

- … these algorithms mishandle the case of a == b!
  - "[At] CppCon 2014, Committee member Walter Brown mentioned that 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!

 16

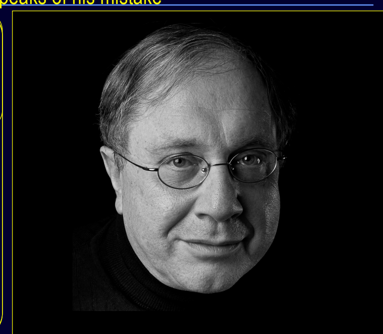
16

Alex Stepanov speaks of his mistake

So for as long as C++ stands, my shame will be publicly visible.

And how stupid could one be? I mean, one 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!

And you will say, 'Well, but nobody will remember that.' Oh, no. People will remember for centuries because that's the max in the standard library!



 17

17

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
      friend bool                                        // "hidden friend"
        operator < ( student s1, student s2 )
      { return s1.name < s2.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!

 18

18

How Do We Address This?

[O]nly wise men learn from their mistakes.

— Winston Churchill

19
    ```

The 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.
  - *I.e.*, not all sorted sequences are 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.

 20

20

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.

 21

21

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.

 22

22

Therefore, I recommend …

- For min:
  - `… { return out_of_order(a, b) ? b : a; }` *// in order ? a : b*
- For max:
  - `… { 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); }
```

 23

23

These Ideas Are Broadly Applicable

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

— Charles Darwin

24

Analogous logic also applies elsewhere ①

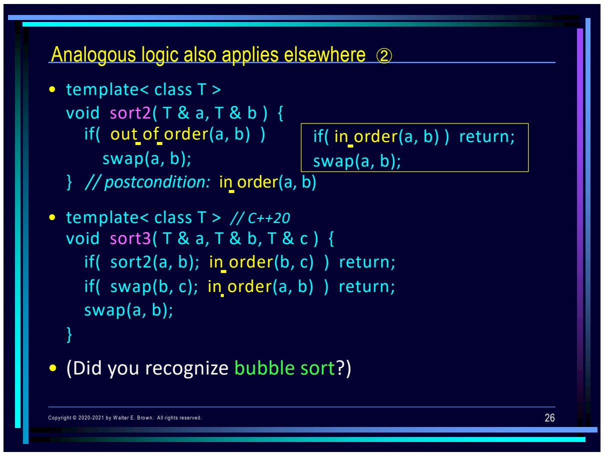
```
template< input_iterator In, output_iterator<In> Out >
Out merge( In b1, In e1   // 1st sorted input range
         , In b2, In e2   // 2nd sorted input range
         , Out to ) {     // merged destination

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

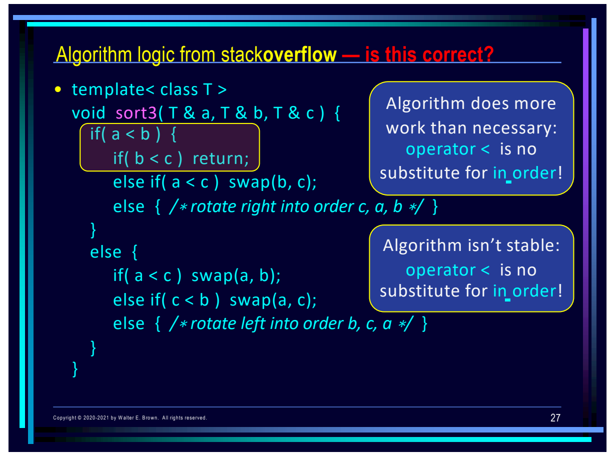
"Prefer the 1st range. Must have a reason to take from the 2nd."

 25

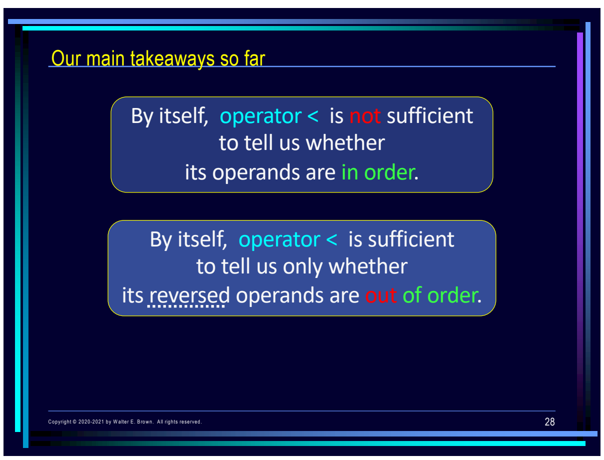
25



26

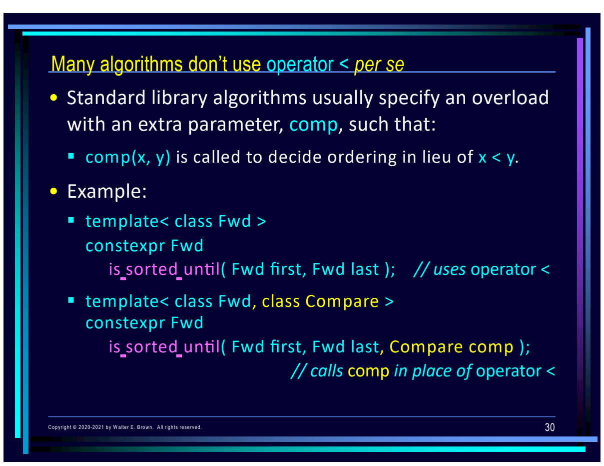


27

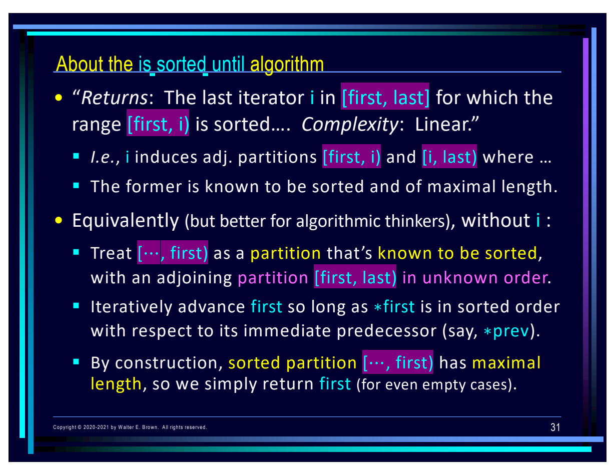


28

29



30



31

My earliest implementation

- Using operator < :
  - ```
    template< class Fwd >              // forward iterator
    constexpr Fwd
      is_sorted_until( Fwd first, Fwd last )
    {
      if( first != last )       // init/reinit loop as if by prev = first++
        for( Fwd prev = first; ++first != last; prev = first )
          if( *first < *prev )   // in order? out of order?
            break;
      return first;
    }
    ```

 32

32

But, as before, I prefer and recommend …

- … to use a named order predicate:
  - ```
    template< class Fwd >
    constexpr Fwd
      is_sorted_until( Fwd first, Fwd last )
    {
      #define out_of_order( x, y ) (*(y) < *(x) )
      if( first != last )
        for( Fwd prev = first; ++first != last; prev = first )
          if( out_of_order(prev, first) )
            break;
      return first;
    }
    ```

Tip: Pass the iterators (typically cheap to copy) rather than the dereferenced values (which may be not even copyable)!

 33

33

[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. *E.g.*, I'd prefer s/comp/less_than/ or s/comp/lt/ or s/comp/precedes/.)

 34

34

Even when an explicit less-than predicate is supplied …

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

 35

35

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?

 36

36

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.)

 37

37
    ```

Q2: Do algorithms need both default argument kinds?

- Review the algorithm declaration, 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 deduced from any default function arg.)
  - Enables calling code to default-construct a 3rd argument, namely std::ranges::less{ } .

 38

38

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

- Short answer: because it's not allowed to:
  - "An implementation shall not declare a non-member function signature with additional default arguments." (See [global.functions]/3.)
- Longer answer: 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.)
  - Also, suppose the caller provides a type but not a value:
    ```
    template< class T = int > void g( T x = { } ) { ⋯ }
      ⋮
    g<MyType>( );   // what if MyType isn't default-constructible?
    ```

 39

39

std Disguises for operator <

Everybody's wearing a disguise....

— Bob Dylan

40

How many ways can std design and ~~disguise~~ spell operator < ?

Name	Where found	Since	Taking
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

 41

41

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

```
struct less {
  template< class L, class R >
  constexpr bool operator( ) ( L && left, R && right ) const
  {
    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);
  }
};
```

 42

42

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( signed_type<L> == signed_type<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);
}
```

 43

43

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
{
   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;
}
```

 44

44

My version of IEEE's totalOrder [restricted for exposition]

```
template< floating_point_type F >                     // assumes IEEE
constexpr  bool  totalOrder( F left, F right )  {
   if(  signbit(left)  !=  signbit(right)  )          // opposite sign bits
      return  signbit(left);
   else  {
      using  int_t  =  big_enough_type< sizeof(F)
                                      ,  int, long, long long >;
      static_assert( sizeof(F) == sizeof(int_t)  );   // assumption
      int_t  x  =  bit_cast< int_t >( left )
         ,  y  =  bit_cast< int_t >( right );
      return  signbit(x)  ?  y  <=  x          // both have sign bit set
                          :  x  <=  y;         // neither has sign bit set
   }
}
```

 45

45

Bonus Algorithm

"I Xeroxed a mirror.
Now I have an extra Xerox machine."

— *Steven Wright*

46

Suppose you need both extrema

- We could reuse min and max:
  - ```
    template< class T >
    pair<T const &, T const & >
       minmax( T const & a, T const & b )
    {
       return  { min(a, b), max(a, b) };
    }
    ```
- But it's cheaper to make one call to operator < than the two made within separate calls to min and to max:
  - ```
    if(  out_of_order(a, b)  )  return  { b, a };
    else                        return  { a, b };
    ```

 47

47

Finally, a modest programming challenge

- If you've never considered the generalized minmax:
  - ```
    template< forward_iterator F >
    pair<F, F>
       minmax( F from, F upto );   // let N = distance(from, upto)
    ```
  - It returns m and M, iterators in [from, upto), such that m is the first iterator whose *m is smallest, and M is the last iterator whose *M is largest.
- Separate calls to min then max functions would lead to $\mathcal{O}(N + N = 2N)$ calls to out_of_order:
  - But Pohl's minmax needs only $3N/2$ calls to out_of_order.
  - (This is std::minmax_element in <algorithm>.)

 48

48

Correctly Calculating

min, max, and More

FIN

Walter E. Brown, Ph.D.

< webrown.cpp @ gmail.com >

49
    ```