Correctly Calculating min, max, and More...

What Can Go Wrong?

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

WALTER E BROWN

Welcome!

A little about me

Emeritus participant in C++ standardization

Introduction

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

— Walter Lippmann

Sound check

[b] [London Fanfare Trumpets: Flourish II]

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

“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 Стемана́)
So What's Wrong?

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

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!

Many types do distinguish equal values

• Bare-bones example:
  ```cpp
  struct student {
    string name; int id;
    inline static int registrar = 0;
    $\{ string n : name[n], id, registrar++ \} \{ \} // c'tor
    friend bool 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!

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.
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.

An important insight

- Given two values a and b, in that order:
  - Unless we find a reason to the contrary, ...
  - \( \text{min} \) should prefer to return a, and ...
  - \( \text{max} \) should prefer to return b.
  - I.e., never should \( \text{max} \) and \( \text{min} \) return the same item:
    - When values a and b are in order, \( \text{min} \) should return a over \( \text{max} \) should return b; ...
    - When values a and b are out of order, \( \text{min} \) should return b over \( \text{max} \) should return a.

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.

Therefore, I recommend ...

- For \( \text{min} \):
  - \( \cdots \{ \text{return out of order}(a, b) ? b : a; \} \ // \) in order ? a : b
- For \( \text{max} \):
  - \( \cdots \{ \text{return out of order}(a, b) ? a : b; \} \ // \) in order ? b : a
- Where:
  - \( \text{inline bool} \)
    - \( \text{out of order}(\cdots x, \cdots y) \{ \text{return y < x; } \} // !!! \)
  - \( \text{inline bool} \)
    - \( \text{in order}(\cdots x, \cdots y) \{ \text{return not out of order}(x, y); \} \)

Analogous logic also applies elsewhere:

- 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++;
    }

These Ideas Are Broadly Applicable

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

— Charles Darwin
Analogous logic also applies elsewhere

- template< class T >
  void sort2(T & a, T & b) {  // postcondition: in order(a, b)
    if( out of order(a, b) ) return;  
    swap(a, b);  
  }

- template< class T >
  // C++20
  void sort3(T & a, T & b, T & c) {
    if( sort2(a, b);  // postcondition: in order(b, c) ) return;
    if( swap(b, c);  // postcondition: in order(a, b) ) return;
    swap(a, b);  
  }

(Did you recognize bubble sort?)

Algorithm logic from stackoverflow — is this correct?

- template< class T >
  void sort2(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 doesn’t use operator < per se

- Standard library algorithms usually specify an overload with an extra parameter, comp, such that:
  - comp(x, y) is called to decide ordering in lieu of x < y.

- Example:
  - template< class Fwd >
    constexpr Fwd is_sorted_until(Fwd first, Fwd last);  // uses operator <
  - template< class Fwd, class Compare >
    constexpr Fwd is_sorted_until(Fwd first, Fwd last, Compare comp);  // calls comp in place of operator <

About the is sorted until algorithm

- “Returns: The last iterator i in [first, last] for which the range [first, i) is sorted... Complexity: Linear.”
- i.e., i induces adj. partitions [first, i) and [i, last) where...
- The former is known to be sorted and of maximal length.

- Equivalently (but better for algorithmic thinkers), without i:
  - Treat [prev, first] as a partition that’s known to be sorted, with an adjoining partition [first, last] in unknown order.
  - Iteratively advance first so long as first is in sorted order with respect to its immediate predecessor (say, prev).
  - By construction, sorted partition [prev, first) has maximal length, so we simply return first (for even empty cases).
Correctly Calculating min, max, and More...

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;
  }
  ```

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;
  }
  ```
  (IMO, the names comp and Compare are too general. E.g., I'd prefer std:ranges::less or std::less to use a named 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(x, y); }
    if( first != last )
      for( Fwd prev = first; ++first != last; prev = first )
        if( out_of_order(prev, first) )
          break;
    return first;
  }
  ```

[alg.sorting.general]/2-3 [ranged]
• "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 std:ranges::less or std::less)

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(x, y); }
    if( first != last )
      for( Fwd prev = first; ++first != last; prev = first )
        if( out_of_order(prev, first) )
          break;
    return first;
  }
  ```

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’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.)
Q2: Do algorithms need both default argument kinds?
- Review the algorithm declaration, then consider a call:
  ```cpp
  template< class Fwd, class Compare = std::ranges::less >
  constexpr Fwd is_sorted_until( Fwd first, Fwd last, Compare lt = { } );
  ```
- ```cpp
  int a[N] = { ... };
  ```
- ```cpp
  is_sorted_until( a+0, a+N )
  ```
  // what type is Fwd?
- It’s `std::ranges::less`, per the default template arg:
  ```cpp
  (A type is never deduced from any default function arg.)
  ```
- Enables calling code to default-construct a 3rd argument, namely `std::ranges::less{ }`.

Q3: Why doesn’t my std library use such default arg’s?
- Short answer: because it’s not allowed to:
  ```cpp
  "An implementation shall not declare a non-member function signature with additional default arguments.”
  ```
  (See [global.functions]/3.)
- Long answer: because doing so is problematic:
  ```cpp
  "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:
  ```cpp
  template< class T = int > void g( T x = { } ) {
  ...
  }
  ```
  ```cpp
  g<MyType>();
  ```
  // what if `MyType` isn’t default-constructible?

### std Disguises for operator <

Everybody’s wearing a disguise….

— Bob Dylan

---

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Where found</th>
<th>Since</th>
<th>Taking</th>
</tr>
</thead>
<tbody>
<tr>
<td>class template less</td>
<td>&lt;functional&gt;</td>
<td>C++98</td>
<td>T, T</td>
</tr>
<tr>
<td>specialization lessvoid</td>
<td>&lt;functional&gt;</td>
<td>C++14</td>
<td>T, U</td>
</tr>
<tr>
<td>class ranges::less</td>
<td>&lt;functional&gt;</td>
<td>C++20</td>
<td>T, U</td>
</tr>
<tr>
<td>function template cmp_less</td>
<td>&lt;utility&gt;</td>
<td>C++20</td>
<td>integer I, J</td>
</tr>
<tr>
<td>overload set isless</td>
<td>&lt;cmath&gt;</td>
<td>C++11</td>
<td>arith A, B</td>
</tr>
<tr>
<td>specification totalOrder</td>
<td>IEEE 754; in spec of <code>&lt;compare&gt;</code>’s strong_order</td>
<td>2008; C++20</td>
<td>flt-pt F, F</td>
</tr>
</tbody>
</table>

---

**My version of std::ranges::less**

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

**My version of std::cmp::less**

```cpp
template< std::integer_type L, std::integer_type R >
  constexpr bool cmp_less( L left, R right ) noexcept
  {...
  }
```
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 flt = common floating point t<L, R>;
    flt x = left, y = right;
    return isunordered(x, y) ? false // avoid FE_INVALID : x < y;
  }

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) >;
      static assert( sizeof(F) == sizeof(int t) ); // assumption
      int x = bit cast< int t >( left ), y = bit cast< int t >( right );
      return signbit(x) ? y <= x : x <= y; // both have sign bit set
      // neither has sign bit set
    }
  }

Bonus Algorithm

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

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 };

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>.)

Correctly Calculating min, max, and More

FIN

Walter E. Brown, Ph.D.
< webrown.cpp @ gmail.com >