


Extrema:
Correctly Calculating min and max

◀.....▶

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




Edition: 2021-06-19. Copyright © 2020-2021 by Walter E. Brown. All rights reserved.

1

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.

2

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.

3

Description of this talk

- The C++ standard library long ago selected `operator<` as its ordering primitive.
- This brief talk will explain why `operator<` must be used with care, in even such seemingly simple algorithms 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.

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

4

The intuitive approach ①

- As C-style macros:
 - `#define MIN(a,b) ((a)<(b)?(a):(b))`
 - `#define MAX(a,b) ((b)<(a)?(a):(b))`
- 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 (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.

5

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 several 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 non-obvious how to compare them generically — e.g., consider **signed** vs. **unsigned**!)
- ✗ Major concern: are the algorithms **correct for all values**?

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

6

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; }`

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

7

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.

8

In other words, ...

- ... 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:
 - That equal values are indistinguishable, so ...
 - It ought not matter which is returned, so ...
 - This case is uninteresting and not worth even discussing.
- Alas, for `min` and `max` algorithms, such opinions are superficial and, in general, are incorrect!

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

9

Many types do distinguish equal values

- 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 not salient`
`};`
- Since each `student` variable has a unique `id` number, it matters greatly which one is returned by `min/max`!

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

10

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.
- Never should `max` and `min` return the same value:
 - 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.

11

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 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.

12

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:

```
bool out_of_order( ... x, ... y ) { return y < x; }  
bool in_order( ... x, ... y ) { return not out_of_order(x, y); }
```

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 13

13

Analogous logic also applies elsewhere ①

- template< input_iterator In, output_iterator<In> Out >
Out merge(In b1, In e1 // 1st range
 , In b2, In e2 // 2nd range
 , Out to) {
 while(b1 != e1 and b2 != e2)
 if(out_of_order(*b1, *b2)) *to++ = *b1++;
 else *to++ = *b2++;
 while(b1 != e1) *to++ = *b1++;
 while(b2 != e2) *to++ = *b2++;
 return result;
}

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

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 14

14

Analogous logic also applies elsewhere ②

- template< class T >
void sort2(T & a, T & b) {
 if(out_of_order(a, b)) if(in_order(a, b)) return;
 swap(a, b);
} // postcondition: in_order(a, b)
- template< class T >
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. 15

15

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. 16

16

Our main takeaways

By itself, operator < is **not** sufficient to tell us whether its operands are **in order**.

By itself, operator < is sufficient to tell us only whether its **reversed** operands are **out of order**.

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

17

Bonus algorithm: minmax

- Suppose you need both extrema:
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 out_of_order than the two made via separate calls to min and to max:
return out_of_order(a, b) ? { b, a } : { a, b };

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

18

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>`.)

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

19


Extrema:
Correctly Calculating min and max

←.....→

FIN

Walter E. Brown, Ph.D.

< webrown.cpp @ gmail.com >



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

20