

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.

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.

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.

“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.

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.

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:

- X The **by-value parameter passage** is potentially expensive (e.g., for large **string** arg's).
- X 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!])
- X Major concern: are the algorithms **correct for all values**?

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

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
 - X auto &r = min (x.calc(), y.calc()); // dangling reference!

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

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.

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:
 - That equal values are **indistinguishable**, so ...
 - It ought not matter which is returned, so ...
 - 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]



So it was popular. But it is being shame will be And then I'll say, "Why should it matter which value is returned?"

Oh, no. People will remember for centuries and then he writes `max` and 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

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. 23

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. 24

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. 26

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)
- 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. 27

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. 28

Our main takeaways so far

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. 29

operator < Is Spelled Other Ways, Too

Sameness is tiresome; variety is pleasing.
— Mark Twain

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

Many std algorithms don't use operator < per se

- Standard library algorithms often 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 <`

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 31

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 `[..., 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 `[..., first]` has maximal length**, so we simply return `first` (for even empty cases).

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 32

My earliest `operator <` implementation [edited for exposition]

```

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;
}
    
```

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 33

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** (which are typically cheap to copy) rather than the **dereferenced values** (which may be not even copyable)!

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 34

[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 3<sup>rd</sup> 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

**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 <b>less</b>             | <functional>                                         | C++98       | T, T         |
| specialization <b>less&lt;void&gt;</b> | <functional>                                         | C++14       | T, U         |
| class <b>ranges::less</b>              | <functional>                                         | C++20       | T, U         |
| function template <b>cmp_less</b>      | <utility> (why?)                                     | C++20       | integer I, J |
| overload set <b>isless</b>             | <cmath>                                              | C++11       | arith A, B   |
| specification <b>totalOrder</b>        | IEEE 754; in spec of <compare>'s <b>strong_order</b> | 2008; C++20 | flt-pt F, F  |

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

**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.

**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.

**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.

**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.

Now consider IEEE's floating-point layout in that light

- Relative to trad. scientific notation  $\pm d.d\dots \times 10^{\pm e\dots}$ , IEEE decomposes/rebases/reorders/adjusts its parts:

← 32 Bits →

|           |            |             |
|-----------|------------|-------------|
| Sign      | Exponent   | Mantissa    |
| ← 1 Bit → | ← 8 Bits → | ← 23 Bits → |

What if we treated these bits as a 32/64/128-bit int?

← 64 Bits →

|           |             |             |
|-----------|-------------|-------------|
| Sign      | Exponent    | Mantissa    |
| ← 1 Bit → | ← 11 Bits → | ← 52 Bits → |

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 47

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

- template< floating\_point type F > // assumes IEEE
- constexpr bool total\_order( F left, F right ) noexcept

```

{
 if(signbit(left) != signbit(right)) // exactly one is negative
 return signbit(left);
 else {
 using int_t = big_enough_type< sizeof(F)
 , int32_t, int64_t, int128_t >;
 int_t x = bit_cast< int_t >(left)
 , y = bit_cast< int_t >(right);
 return signbit(x) ? in_order(y, x) // both are negative
 : in_order(x, y); // neither is negative
 }
}

```

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 48

### A Bonus Algorithm

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

— Steven Wright

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 49

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 };

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 50

Finally, let's consider minmax over a sequence

- Found in the <algorithm> header:
  - template< forward\_iterator Fwd >
  - pair<Fwd, Fwd >
  - minmax\_element( Fwd first, Fwd last );
- It returns a pair {m, M}, iterators in [first, last), such that:
  - m is the first iterator whose \*m is smallest, while ...
  - M is the last iterator whose \*M is largest.
- Let N = distance(first, last):
  - Separate calls to min then max functions would lead to O(N + N = 2N) calls to out\_of\_order.
  - But Pohl's 1972 algorithm needs only O(3N/2) calls!

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 51

Infrastructure for Ira Pohl's algorithm

- Given forward iterators f1, f2, we'll use:
  - precedes(f1, f2) that returns \*f1 < \*f2 (or lt(\*f1, \*f2) when there's a Compare lt).
  - out\_of\_order(f1, f2) that returns precedes(f2, f1).
  - max(f1, f2) and min(f1, f2) that call out\_of\_order(f1, f2).
- Let mM denote an ordered std::pair of iterators:
  - minMax(f1, f2) that makes an mM pair by returning out\_of\_order(f1, f2) ? mM{ f2, f1 } : mM{ f1, f2 }.
  - meld(a, b) that combines two mM pairs into one via mM{ min(a.first, b.first), max(a.second, b.second) }.

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 52

The logic of Pohl's algorithm [C++20]

- Most of the code is for special cases at begin/end:
  - using mM = std::pair<Fwd, Fwd>;
  - Fwd prev = first;
  - if( prev == last or ++first == last ) // empty? singleton?
    - return mM{prev, prev};
  - for( mM so\_far = minMax(prev, first); ; ) // initial pair
    - if( ++first == last ) // nothing more to process?
      - return so\_far;
    - else if( prev = first; ++first == last ) // final singleton?
      - return meld( so\_far, mM{prev, prev} );
    - else // general case: meld result so far w/ latest pair
      - so\_far = meld( so\_far, minMax(prev, first) );

Copyright © 2020-2021 by Walter E. Brown. All rights reserved. 53

Correctly Calculating  
min, max, and More

←-----→

FIN

Walter E. Brown, Ph.D.

< webrown.cpp @ gmail.com >



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