Correctly Calculating min, max, and More

What Can Go Wrong?

Walter E. Brown, Ph.D.

< webrown.cpp @ gmail.com >

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!)

 4

Emeritus participant in C++ standardization

- Written ~175 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! ☺

 5

The Big Picture

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

— Walter Lippmann

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 < 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.

 7

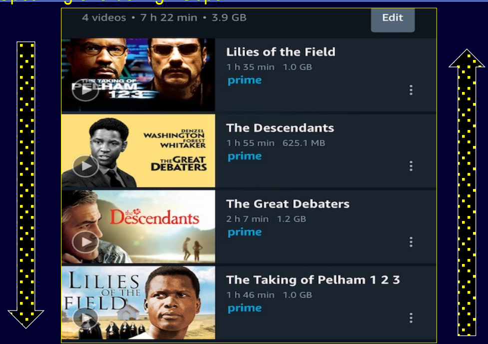
"One of the amazing things which we … discover is that ordering is very important.

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

— Alexander Stepanov
(né *Алекса́ндр Степа́нов*)

 8

Speaking of ordering: Oops!



 9

Early Adventures with `min` and `max`

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

An intuitive approach ①

- As function-like macros in the C style:
  - `#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; }`

 11

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?

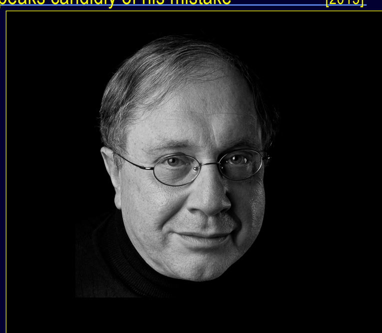
 12

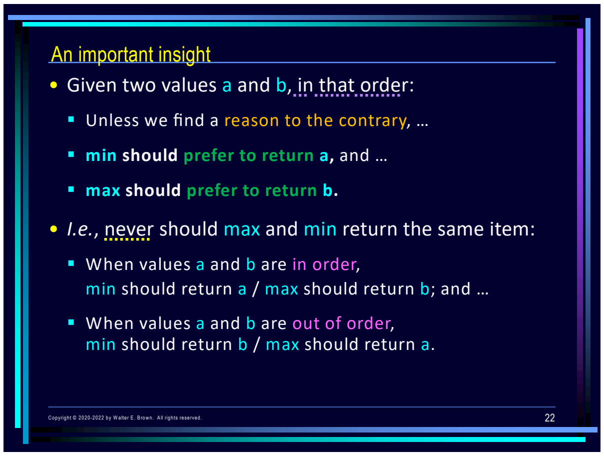
The cures are mostly straightforward

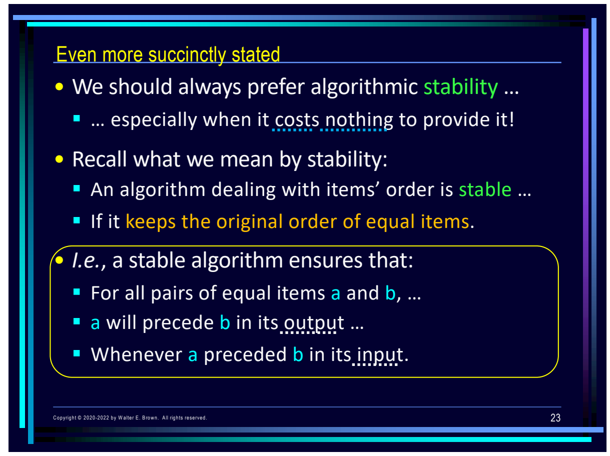
- Per the std library's specification:
  - ✓ Enforce consistent types via a named type parameter.
  - ✓ Avoid expensive copies via call/return by ref-to-const.
- After these adjustments we have:
  - ```
    template< class T >
    T const &
      min( T const & a, T const & b ) { return a < b ? a : b; }
    ```
  - And analogously for max.
- (But pls 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!`

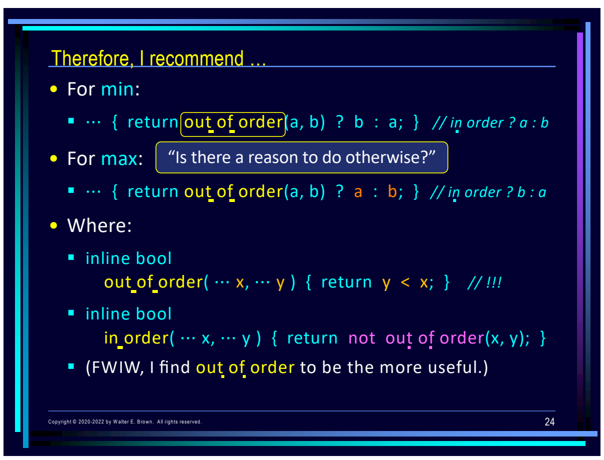
 13

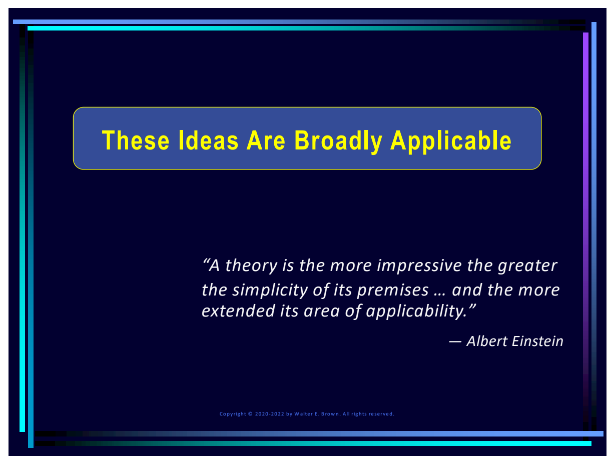
So What's Wrong?

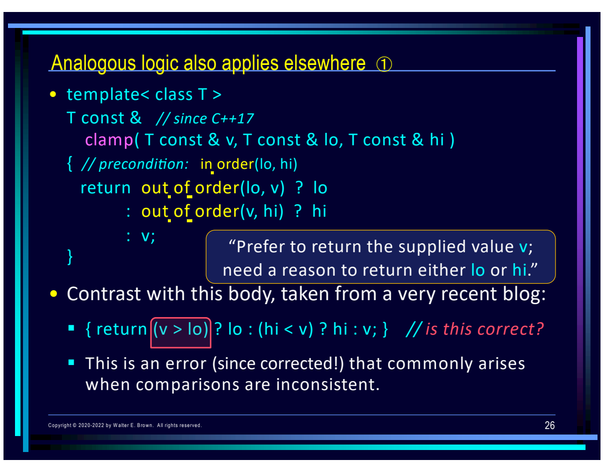
"[N]ever feel badly about making mistakes … as long as you … learn from them."
— Norton Juster
    ```

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

- Can you identify the misbehavior(s)?
  - ```
    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?
  - Yet the C++ standard library does this.
    (“It took Stepanov 15 years to get min and max right.”)

 16

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!

 17

Alex. Stepanov speaks candidly of his mistake [2013]



 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!

 19

How Do We Address This?

“[O]nly wise men learn from their mistakes.”
— Winston Churchill

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 seq. is *sorted* if, for every iterator i and non-negative integer n, ∗(i + n) < ∗i is false [*i.e.*, not out of order].

 21
    ```

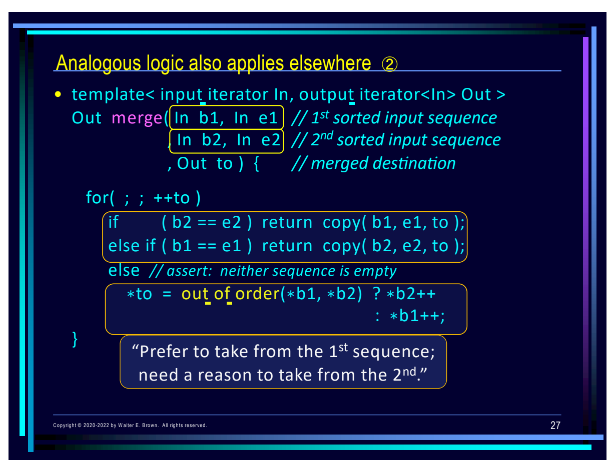

An important insight
Given two values a and b, in that order:
Unless we find a reason to the contrary, …
min should prefer to return a, and …
max should prefer to return b.
I.e., never should max and min return the same item:
When values a and b are in order,
min should return a / max should return b; and …
When values a and b are out of order,
min should return b / max should return a.
Copyright © 2020-2022 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 © 2020-2022 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, I find out_of_order to be the more useful.)
Copyright © 2020-2022 by Walter E. Brown. All rights reserved.
24


These Ideas Are Broadly Applicable
"A theory is the more impressive the greater the simplicity of its premises … and the more extended its area of applicability."
— Albert Einstein


Analogous logic also applies elsewhere ①
template< class T >
T const & // since C++17
clamp(T const & v, T const & lo, T const & hi)
{ // precondition: in_order(lo, hi)
return out_of_order(lo, v) ? lo
: out_of_order(v, hi) ? hi
: v;
}
"Prefer to return the supplied value v;
need a reason to return either lo or hi."
Contrast with this body, taken from a very recent blog:
{ return (v > lo) ? lo : (hi < v) ? hi : v; } // is this correct?
This is an error (since corrected!) that commonly arises when comparisons are inconsistent.
Copyright © 2020-2022 by Walter E. Brown. All rights reserved.
26


Analogous logic also applies elsewhere ②
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
for(; ; ++to)
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 © 2020-2022 by Walter E. Brown. All rights reserved.
27

Analogous logic also applies elsewhere ③

```
template< swappable 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< swappable 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);
}
```

- (BTW, did you recognize bubble sort?)

 28

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!

 29

Our main takeaways so far

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

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

 31

operator< Is Spelled Other Ways, Too

"Sameness is tiresome; variety is pleasing."

— Mark Twain

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 <
    ```

 33

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 thinking), 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 return first (for even empty or singleton ranges).

 34

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;
}
```

Tip: prefer pre-increment;
need a reason to use post-increment.
(Ditto for pre- vs post-decrement.)

35

Nowadays, 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)!

36

[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/ or s/comp/before/.

37

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    // could be named indirect_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;
}
```


38

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?

39

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; }          // note: 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.)

40

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 inferred 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, in this case std::ranges::less{ } .

 41

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 supplying a type but not a value:
    template< class T = int > void g(T x = { }) { ⋯ }
    ⋮
    g<MyType>( ); *// what if MyType isn't default-constructible?*

 42

std Disguises for operator<

"Everybody's wearing a disguise...."

— Bob Dylan

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

 44

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);
  }
};
```

 45

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;  // safely converts lesser to greater rank
  else  // mixed signedness
    if constexpr( signed_type<L> )  // and unsigned_type<R>
      return left < 0 ? true : as_unsigned(left) < right;
    else  // unsigned_type<L> and signed_type<R>
      return right < 0 ? false : left < as_unsigned(right);
}
```

 46

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;
}
```

 47

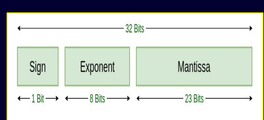
The ordering specified by IEEE's totalOrder predicate

- 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.

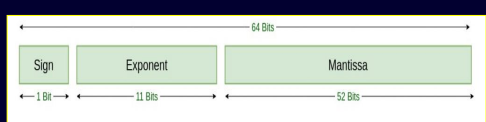
 48

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

- Relative to trad. scientific notation ±d.d... x 10^{±e...}, IEEE decomposes/rebases/reorders/adjusts its parts:



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



 49

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

```
template< floating_point_type F >                  // assumes IEEE rep's
constexpr  bool
   total_order( F left, F right ) noexcept         // 2008–2018 signature
{
   if(  signbit(left)  !=  signbit(right)  )       // one is negative, one is not
      return  signbit(left);
   else  {   // same signs
      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
   }
}
```

 50

And one more: lexicographical_compare! [edited for exposition]

```
template< input_iterator  In1, input_iterator  In2 >
constexpr bool
   lexicographical_compare( In1 first1,  In1 last1
                          , In2 first2,  In2 last2
                          )
{
   for(   ;  first2 != last2;   ++first1, (void)++first2  )
      if(  first1 == last1  )  return true;
      else if(  iter_out_of_order(first1, first2)  )  return false;
      else if(  iter_out_of_order(first2, first1)  )  return true;

   return false;
}
```

 51

There's More to Comparing Than Comparing Correctly

"Comparison is the thief of joy."

— *Theodore Roosevelt*

E.g., std::cmp_less isn't always the answer [courtesy R. Seacord]

- Do you see the potential security risk in this seemingly-innocent code? (Hint: possible buffer overrun!)
-
```
char *
  char_array_copy( size_t n, char const * a )        // lightly edited
{
  char * p = n == 0 ? NULL
                    : (char *) malloc( n );
  if( p != NULL )
    for( auto k = 0; k < n; ++k )
      p[k] = *a++;
  return p;
}
```

Need cmp_less(k, n) here, but there's a deeper issue!

Consider the range of possible values of k, *vis-à-vis* the range of possible values of n.

 53

Case 1: when sizeof(k) == sizeof(n) [lightly edited]

- "The unsigned n may contain a value greater than INT_MAX [an int's maximum value]:
  - "Assuming quiet wraparound on signed overflow [which is a common manifestation of this undefined behavior]…
  - "Once [the int] k is incremented beyond INT_MAX, k takes on negative values starting with (INT_MIN).
  - "Consequently, the memory locations referenced by p[k] precede the memory referenced by p, and a write outside array bounds occurs."

 54

Case 2: when sizeof(k) < sizeof(n) [lightly edited]

- "For values of n where
  (size_t)INT_MIN < n ≤ SIZE_MAX ,
  - "k wraps and takes the values
    INT_MIN
    to
    INT_MIN + (n – (size_t)INT_MIN – 1) .
  - "Execution of the loop overwrites memory from
    p[INT_MIN]
    through
    p[INT_MIN + (n – (size_t)INT_MIN – 1)] ."

 55

Advice re comparisons

- Do you have a 100%, iron-clad guarantee that the code will never be recompiled:
  - With any other compiler/library? Or …
  - With any other version of your compiler/library? Or …
  - For any other hardware/software platform?
- If not, I respectfully but strongly recommend that we improve our code's portability by:
  - Avoiding mixed-signedness comparisons.
  - Preferring same-type comparisons (and arithmetic, too).
  - Planning for same types even before starting to code.

 56

E.g., in the function we last examined …

- … I recommend the code use same-type comparisons:
  -
```
char *
  char_array_copy( size_t n, char const * a )
{
  char * p = n == 0uz ? nullptr
                      : (char *) malloc( n );
  if( p != nullptr )
    for( size_t k = 0uz; k != n; ++k )
      p[k] = *a++;
  return p;
}
```

Why? Same-type!

Why? Same-type! (C++20)

Why? Correct post-condition!

 57

When we don't use same types …

 58

Not an allegory: using same types is important

- "On June 4, 1996 an unmanned Ariane 5 rocket … exploded just forty seconds after its lift-off….
  - "The destroyed rocket and its cargo were valued at $500 million….
  - "It turned out that the cause of the failure was … a 64 bit floating point number [that] was converted to a 16 bit signed integer.
  - "The number was larger than 32,767, the largest integer storeable [*sic*] in a 16 bit signed integer, and thus the conversion failed."

— Douglas N. Arnold

A Bonus Algorithm

"The true delight is in the finding out, rather than in the knowing."

— Isaac Asimov

Sometimes we 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, let's consider minmax of a sequence

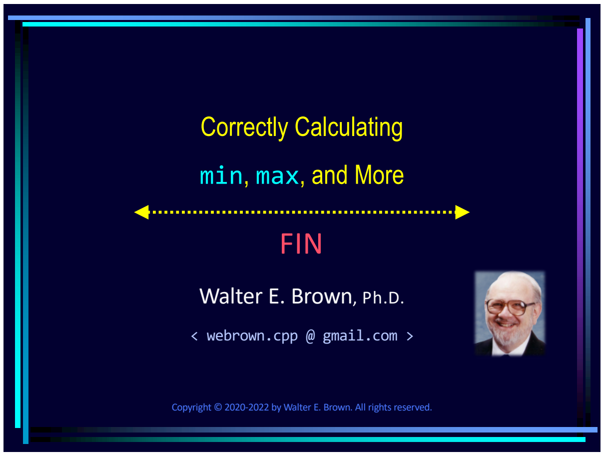
- Found in the <algorithm> header:
  - ```
    template< forward_iterator Fwd >
    pair<Fwd, Fwd>
      minmax_element( Fwd first, Fwd last );
    ```
  - We want a pair {m, M}, iterators in [first, last), such that:
    - m is the first iterator whose ∗m is smallest, while …
    - M is the last iterator whose ∗M is largest.
- Let *N* denote distance(first, last):
  - Separate calls to min then max functions would lead to $O(N + N = 2N)$ calls to out_of_order.
  - But Ira Pohl's 1972 algorithm needs only $O(\frac{3}{2}N)$ calls!

Infrastructure for Ira Pohl's algorithm

- Given fwd iterators f1, f2 , we'll use iter_ versions of:
  - precedes(f1, f2), returning ∗f1 < ∗f2 (or returning lt(∗f1 , ∗f2) when there's a Compare lt).
  - out_of_order(f1, f2), returning precedes(f2, f1) .
  - min(f1, f2) / max(f1, f2), each calling out_of_order(f1, f2) .
- Let mM denote an in_order std::pair of iterators, then:
  - pair_up(f1, f2) makes such an in_order mM pair, returning out_of_order(f1, f2) ? mM{ f2, f1 } : mM{ f1, f2 } .
  - meld(p, q) combines two mM pairs into one, returning mM{ min(p.first , q.first) , max(p.second, q.second) } .

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

- Of 11 lines, 5 initialize and 4 terminate the algorithm:
  - ```
    using mM = std::pair<Fwd, Fwd>;  // in_order(first, second)
    ```
  - ```
    Fwd prev = first;
    if( prev == last or ++first == last )   // empty? singleton?
      return mM{prev, prev};
    ```
  - ```
    for( mM so_far = pair_up(prev, first); ; )   // form 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/ current new pair
        so_far = meld( so_far, pair_up(prev, first) );
    ```



Correctly Calculating
min, max, and More
FIN
Walter E. Brown, Ph.D.
< webrown.cpp @ gmail.com >
Copyright © 2020-2022 by Walter E. Brown. All rights reserved.