~~Accelerated~~ ~~Effective~~ ~~Ineffective~~

Irksome C++

Walter E. Brown, Ph.D.

< webrown.cpp @ gmail.com >

A little about me

- B.A. (mathematics); M.S., Ph.D. (computer science).
- Professional programmer for over 60 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.
  - Trained and lectured internationally.
  - Retired from the Scientific Computing Division at Fermilab, with specialty in C++ programming and in-house consulting.
- **Not dead!** ☺ — still doing training & consulting. (Email me!)

 2

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*, *variable templates*, *requires-expressions*, `operator<=>`, and more!
- Conceived and served as Project Editor for the *Int'l Standard on Mathematical Special Functions in C++* (ISO/IEC 29124), incorporated into `<cmath>` since C++17.
- Be forewarned: Based on my training and experience, I hold some fairly strong opinions about computer software and programming methodology — I do know that these opinions are not shared by all programmers, but they should be! ☺

 3

What's this talk about?

- As much as I like C++, it is certainly not free of — let's say — quirks.
- Both the core language and the standard library exhibit idiosyncrasies, sometimes even in conflict with one another.
  - Most of these "oops" are due to the spectrum of viewpoints held by the hundreds of contributors who have participated in evolving and standardizing C++ over the past 40+ years.
- This talk points out many of these inconsistencies and numerous other infelicities in naming, behavior, or both.
  - ✗ While they're far from fatal flaws, I do find them irksome to learn and teach.

 4

Irk?

- aggravate
- annoy
- bother
- bug
- burn (up)
- chafe
- eat
- exasperate
- frost
- gall
- get
- grate
- gripe
- hack (off)
- irritate
- itch
- nark [*British*]
- nettle
- peeve
- persecute
- pique
- put out
- rasp
- rile
- ruffle
- spite
- vex

Irksome?

- abrasive
- aggravating
- angering
- annoying
- biting
- bothersome
- brattish
- bratty
- burdensome
- carking
- chafing
- discomforting
- displeasing
- disquieting
- distractive
- distressing
- disturbing
- enraging
- exasperating
- frustrating
- galling
- grating
- importunate
- importune
- inconveniencing
- infuriating
- irritating
- jangling
- jarring
- maddening
- mischievous
- nettlesome
- nettling
- offensive
- painful
- peeving
- pesky
- pestiferous
- pestilent
- pestilential
- pesty
- plaguey
- plaguy
- rankling
- rebarbative
- riling
- spiny
- stressful
- thorny
- tiresome
- troublesome
- troubling
- trying
- upsetting
- vexatious
- vexing
- worrisome

As we proceed, please keep this in mind

- "A computer
  is a stupid machine
  with the ability
  to do incredibly smart things, …
- "… while computer programmers
  are smart people
  with the ability
  to do incredibly stupid things."

— Bill Bryson, 1998

Let's begin with vacuity (nothingness)

"Sometimes, in programming,
it's important to do nothing.
It's equally important to do it well."

- So let's envision a type that can be named and manipulated (*e.g.*, by cv/ref-qualifying it, *etc.*), but that is otherwise impotent:
  - ✗ No, not `void`; that's an incomplete type that can never be completed.
  - ✗ Also, please recall that "Forming the type 'reference to *cv* `void`' is ill-formed."
    (N4988 [dcl.ref]/1 — This last sentence of the paragraph almost seems to be hiding: it follows an Example, then a Note, and then a page break!)

I did consider …

- … using one of the numerous tag types afforded by the std library:
  - Even C++98 already had `input_iterator_tag`, `forward_iterator_tag`, ….
  - Since then, we've added many more: `adopt_lock_t`, `allocator_arg_t`, `destroying_delete_t`, `in_place_t`, `monostate`, `nontype_t`, `nullopt_t`, `unreachable_sentinel_t`, `unexpect_t`, `row_major_t`, ….
  - But tag types are typically both default-constructible and copyable (*e.g.*, to be suitable for parameter passage), neither of which behaviors I wanted.
- What I did want, for my own C++ library, was a type:
  - ✗ That can't be constructed, initialized, copied, moved, destroyed, *etc.*, …
  - ✓ Yet that can be treated as a complete, empty, *cvref*-able, inheritable type.
  - ✓ For use when no other type will do. (BTW, this makes a great interview question.)

How hard can it be to design a type that by itself can do nothing?

- Well, once upon a time (>25 years ago, in the pre-C++98 era), I devised such an intentionally mostly-unusable type, initially defined thusly:
  - ```
    struct nonesuch
    {
      nonesuch( );
      ~nonesuch( );

      nonesuch( nonesuch const & );
      nonesuch operator = ( nonesuch const & );
    };
    ```
    none of these implemented
- I much later discovered other code bases with a similar type:
  - *E.g.*, clang's `libc++` has types named `__nat` (acronym for *not a type*).
  - Several `github` projects had the equivalent, too.

By the way, I later also discovered …

- … that I was not the first to adopt the name nonesuch.
- Among others, I found a record label, a publisher, a brewery, a distillery, and mincemeat by that name:

Moreover, I even found …

- … a folk tune by that name:


    ```

Once compilers started to support C++0x [which became C++11], …

- … I updated my type to use a then-new feature:
  - ```
    struct nonesuch
    {
      nonesuch( )                                   = delete;
      ~nonesuch( )                                  = delete;

      nonesuch( nonesuch const & )                  = delete;
      nonesuch operator = ( nonesuch const & )      = delete;
    };
    ```
- These now-deleted definitions improved code clarity, IMO, and also enabled better diagnostics:
  - All Was Good™.

This nonesuch type served me very well for many, many years

- Among several other uses:
  - This `nonesuch` became an integral part …
  - Of my *detection idiom* code [2014].
- In due course, that idiom was proposed, adopted, and published in the ISO *Library Fundamentals* TS [Technical Specification]:
  - *E.g.*, GCC and clang each provide `std::experimental::nonesuch` …
  - Via header `<experimental/type_traits>`.

Alas, … ☹

- As it turned out, I overlooked something …
  - As C++ continued to evolve from ’98 ⇢ ’11 ⇢ ’14 ⇢ ’17 ⇢ ’20 ⇢ ’23 ⇢ ….
  - Namely, that somewhere along the way, …
  - My intentionally useless type now abruptly met the changed definition of the type classification we know as an *aggregate*!
- This meant …
  - That, contrary to its design intent, …
  - A `nonesuch` object suddenly, unintentionally (and unfortunately) …
  - ✗ “… can be [instantiated and] initialized from `{}` in [certain contexts].”

It turned out that … [dcl.init.aggr]

- … Over time, the definition of an aggregate has been “incrementally expanded to cover more user-defined types.” — N. Ranns, 2022
- An *aggregate* is either an array, or else a class with none of these:
  - ✗ C’tors that are [98] user-declared; [11/14] user-provided or explicit; [17] … or inherited; [20] user-declared, explicit, or inherited; [23] user-declared or explicit.
  - ✗ [98/11/14/17/20/23] Private or protected direct non-static data members.
  - ✗ [11 only] Default member initializers for non-static data members.
  - ✗ [98/11/14/17/20] Virtual member functions; [23] … or virtual base classes.
  - ✗ [98/11/14] Base classes; [17/20] virtual, private, or protected base classes; [23] private or protected direct base classes.


    ```

## Well, …





## Fortunately, …

- My C++ standardization colleague Tim Song noticed this evolution, and …
- In 2017, Tim submitted LWG Issue 2960, pointing out that `nonesuch`:
  - ✗ "should have no default constructor" (rather than a deleted one), and …
  - ✗ "shouldn't be an aggregate" (to avoid `{}`-initialization).
- So I quickly fixed my `nonesuch` implementation:
  - ✓ I removed its default c'tor, and …
  - ✓ I provided a previously-unneeded private base class …
  - ✓ So that my `nonesuch` would no longer qualify as an aggregate.




## I greatly appreciated the issue's insight …

- … and the resulting improvements in the specification of my `nonesuch` type.
- In addition, I now humbly nominate this issue as my candidate for a "Best Issue Title" award:
  - **LWG Issue 2960: "`nonesuch` is insufficiently useless."**
  - Thanks again, Tim!

✗ (But I'm still irked about needing to adjust my code for such a reason!)




## Incidentally, …

- Did we all notice the earlier distinctions among the (very similar) terms *user-declared* vs. *user-defined* vs. *user-provided*?
  - "A function is *user-provided* if it is user-declared and not explicitly `default`ed or `delete`d on its first declaration" [dcl.fct.def.default]/5.
  - This affects triviality, for example: "A default constructor is *trivial* if it is not user-provided …." [class.default.ctor]/3.

✗ Thus, the following similar types don't have equivalent semantics:

- `struct A { A() = default; … };` *// A's c'tor not user-provided, so A may be trivial*
- `struct B { B(); … };`
  `B::B() = default;` *// B's c'tor is user-provided, so B can't be trivial*

✗ I acknowledge the utility of such subtleties, yet find them irksome to recall.

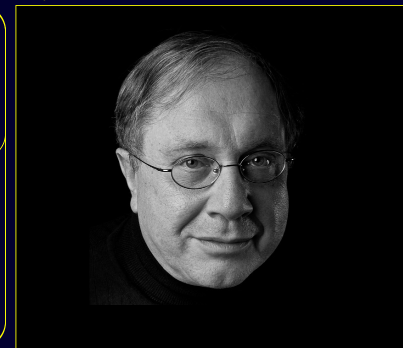
## I have a dream

- I wish we (WG21) could just fix all our obvious mistakes:
  - We can't, because "Every change breaks someone somewhere."
  - But it still seems unconscionable to me to permit incorrect code to linger in any library, let alone in the C++ standard library.
- Consider this question: why should `max` and `min` algorithms ever return the same result for the same arguments?
  - ✓ I would expect, when `min` returns (by reference) one of a given pair of values, that `max` would return the other value — wouldn't you?
  - ✗ But that's not how the standard library defines its `max`!
  - ✗ (I'm irked that the even the newer `std::ranges::max` just "Returns the first [*sic!*] argument [as `min` does] when the arguments are equivalent" [alg.min.max]/10).

## Alex. Stepanov speaks candidly of his mistake [2013]

So for as long as
C++ stands, my
shame will be
publicly visible.”
these orderings,
and writing min
Oh, no. People
in the most
will remember
generic way,
for centuries
and then he
because that's
writes max and
the max in the
he screws it up!
standard library!



## If we could turn back the clock …

- Here's the corrected algorithm:
  - ```
    template< class T >
    T max( T a, T b ) { return b < a ? a : b; }
    ```
- Except that the std library specifies passing by-reference, not by-value:
  - ```
    template< class T >
    T const & max( T const & a, T const & b )
    { return b < a ? a : b; }
    ```
  - It's true that call by-reference does prevent potentially expensive copying, *e.g.*, if the parameter types were `std::string`s.
  - But `return` by-reference leads to other potential issues, not always obvious.

What other issues?

- For example, consider the effects of these two very similar decl's:
  - `auto   r1 = max( calc1(), calc2() );`
  - `auto & r2 = max( calc1(), calc2() );`
- `r1` is clearly a copy, but `r2` is a reference … to what?
  - Note that, in each call, `max`'s arguments are temporaries.
  - Those temporaries' lifetimes will expire at the end of the statement that materialized them, so …
  - ✗ `r2` will quickly become a dangling reference!
- It would be really nice to avoid such a significant problem, right?

More issues

- Since `max` returns by-reference-to-`const`, its callers can't write such otherwise reasonable-looking code as:
  - `max(x,y) = 0;` *// x and y of a numeric type*
  - `max(x,y).mutate_me( );` *// x and y of a class type*
- To make those work, some clients opt to cast away the `const`ness:
  - `const_cast<MyType &>( max(x,y) ) = 0;`
  - `const_cast<MyType &>( max(x,y) ).mutate_me( );`

✗ But even that ugliness doesn't work when `max` is given a temporary (or two) as an argument!

✗ (Updating a value that imminently expires is generally pointless, right?)

 25

What are our options?

- If either argument is a temporary:
  - ✗ We dare not return any reference, lest it quickly dangle once the temporary's lifetime ends.
  - ✓ Thus, our only safe choice is to return a copy, *i.e.*, by-value.
- When neither argument is a temporary:
  - We can safely return a reference … but what kind of reference?
  - ✓ Return by-mutable-lref iff both arguments are mutable.
  - ✓ Otherwise, return by-immutable-lref.

 26

We can do all this with just 2 overloads!

[L. D'Allessandro]

1) When we're given two lvalue arguments:

```
template< class A, class B >
concept same_unqual_types = same_as< remove_cvref_t<A>
                                   , remove_cvref_t<B> >;

constexpr auto max( auto & a, auto & b ) -> auto &
  requires same_unqual_types< decltype(a), decltype(b) >
{ return b < a ? a : b; } // want args' common type here, so use ternary, not if-else
```

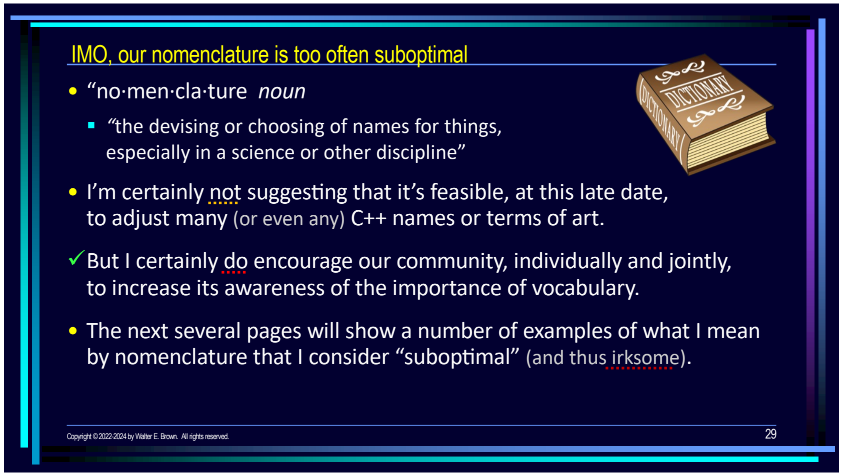
2) When we're given either one or two prvalue arguments:

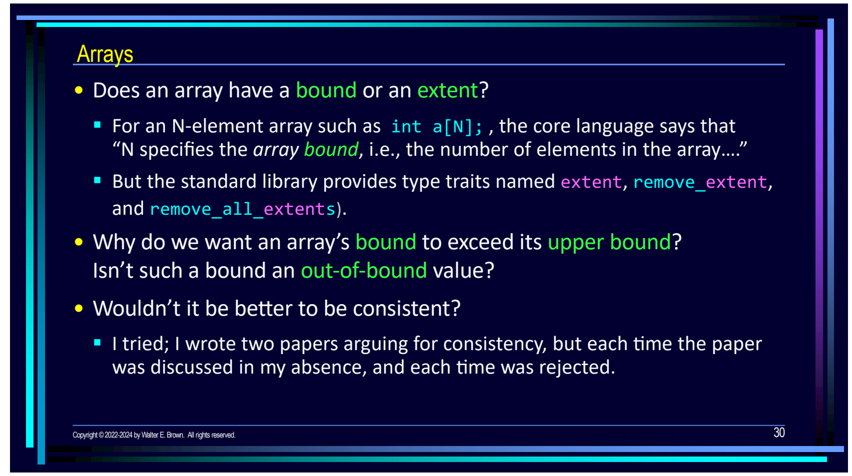
```
constexpr auto max( auto && a, auto && b )
  requires same_unqual_types< decltype(a), decltype(b) > // as above
{ return b < a ? forward<decltype(a)>( a )
               : forward<decltype(b)>( b ); }
```

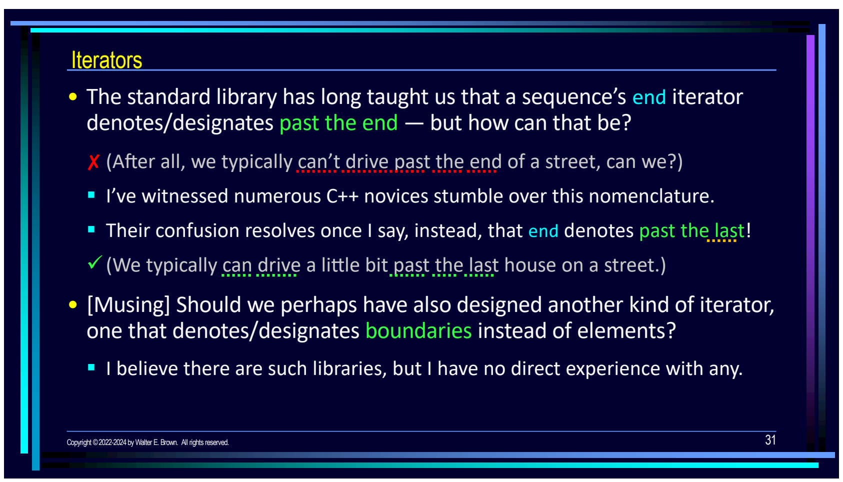
 27

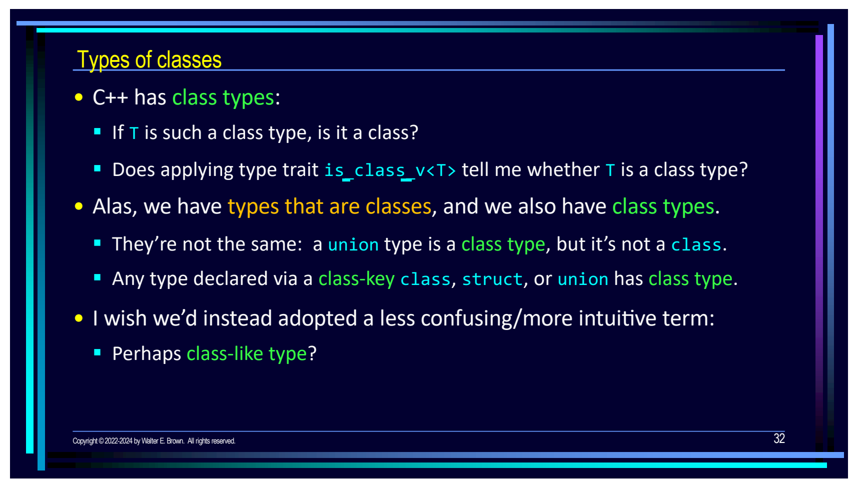
But that wouldn't be backward compatible, so …

 28


IMO, our nomenclature is too often suboptimal
• "no·men·cla·ture noun
▪ "the devising or choosing of names for things, especially in a science or other discipline"
• I'm certainly not suggesting that it's feasible, at this late date, to adjust many (or even any) C++ names or terms of art.
✓ But I certainly do encourage our community, individually and jointly, to increase its awareness of the importance of vocabulary.
• The next several pages will show a number of examples of what I mean by nomenclature that I consider "suboptimal" (and thus irksome).
DICTIONARY
Copyright © 2022-2024 by Walter E. Brown. All rights reserved.
29


Arrays
• Does an array have a bound or an extent?
▪ For an N-element array such as int a[N]; , the core language says that "N specifies the array bound, i.e., the number of elements in the array...."
▪ But the standard library provides type traits named extent, remove_extent, and remove_all_extents).
• Why do we want an array's bound to exceed its upper bound? Isn't such a bound an out-of-bound value?
• Wouldn't it be better to be consistent?
▪ I tried; I wrote two papers arguing for consistency, but each time the paper was discussed in my absence, and each time was rejected.
Copyright © 2022-2024 by Walter E. Brown. All rights reserved.
30


Iterators
• The standard library has long taught us that a sequence's end iterator denotes/designates past the end — but how can that be?
✗ (After all, we typically can't drive past the end of a street, can we?)
▪ I've witnessed numerous C++ novices stumble over this nomenclature.
▪ Their confusion resolves once I say, instead, that end denotes past the last!
✓ (We typically can drive a little bit past the last house on a street.)
• [Musing] Should we perhaps have also designed another kind of iterator, one that denotes/designates boundaries instead of elements?
▪ I believe there are such libraries, but I have no direct experience with any.
Copyright © 2022-2024 by Walter E. Brown. All rights reserved.
31


Types of classes
• C++ has class types:
▪ If T is such a class type, is it a class?
▪ Does applying type trait is_class_v<T> tell me whether T is a class type?
• Alas, we have types that are classes, and we also have class types.
▪ They're not the same: a union type is a class type, but it's not a class.
▪ Any type declared via a class-key class, struct, or union has class type.
• I wish we'd instead adopted a less confusing/more intuitive term:
▪ Perhaps class-like type?
Copyright © 2022-2024 by Walter E. Brown. All rights reserved.
32

An operator

- Consider the unary `operator *` :
  - Is it termed the indirection operator?
  - Or is it the dereference (or the dereferencing) operator?
- At least we're consistent, right? Well, …
  - In the core language, "The unary `*` operator performs *indirection*…."
  - In the standard library (many places), "Each iterator … is dereferenced …."
- While we're at it, shouldn't we consistently name an entity that accepts and internally dereferences iterators? Today, namespace `std` has:
  - `iter_swap`
  - `iter_value_t`
  - `iter_move`
  - `indirect_result_t`
  - `indirectly_readable`
  - `indirect_binary_predicate`

Function declarations

- Do we declare a function as `noexcept` or as `nothrow`?
- We evidently use both terms — within even a single line of code!

```
template< class T >
void g( … ) noexcept( is_nothrow_assignable_v<T> );
```

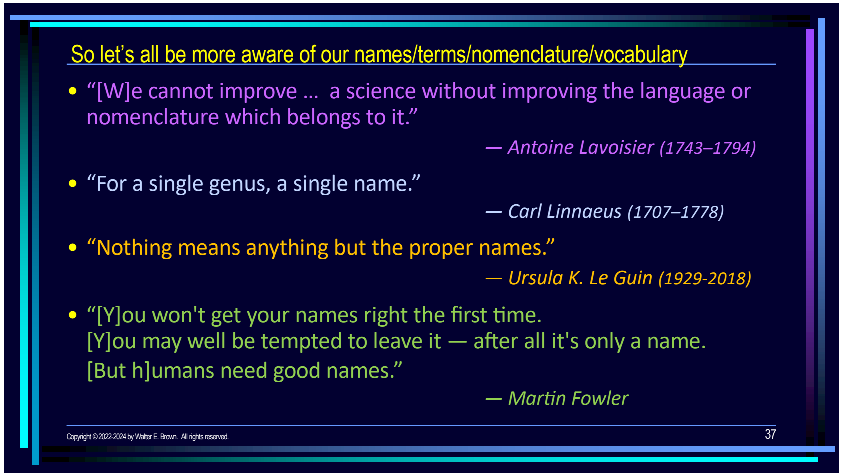
- To my regret, I may be partly responsible for this:
  - In "Toward Improved Optimization Opportunities in C++0X" (2004), I proposed `nothrow` as a new qualifier in function declarations.
  - It was reproposed/adopted years later in the core language as `noexcept`.
  - But `nothrow` was adopted as the library's nomenclature for such behavior.
  - Please accept my apologies; I wasn't in the room when these were decided.

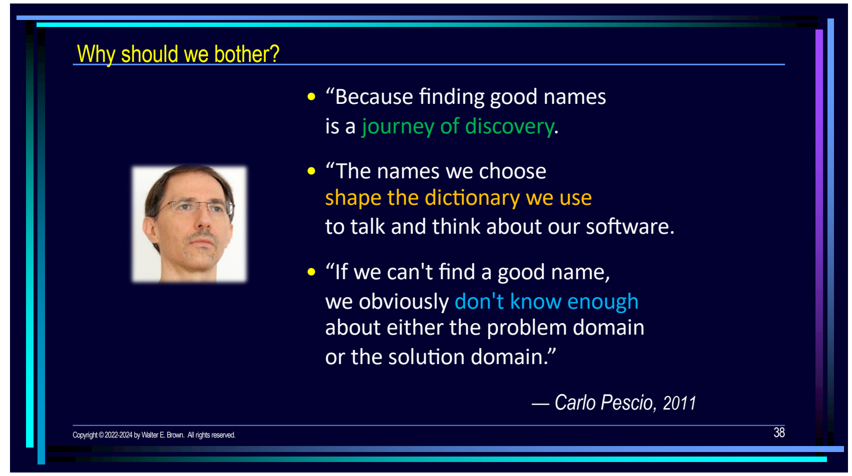
Component names

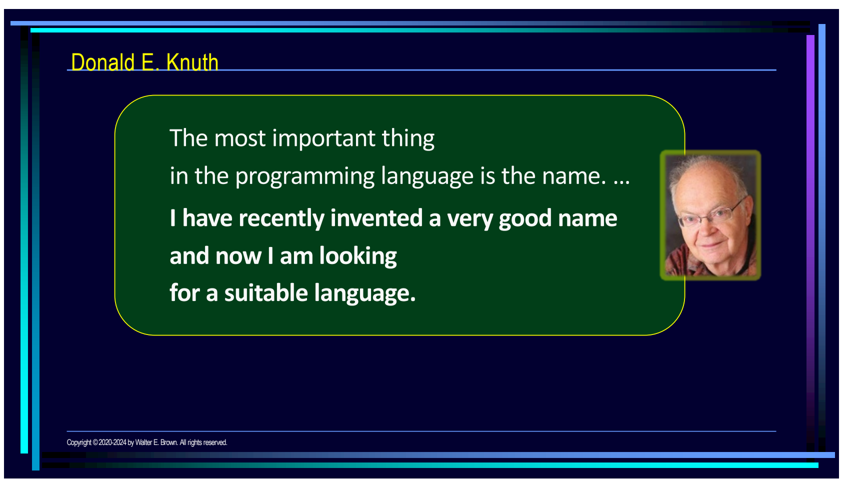
- C++23 adds `remove_prefix` and `remove_suffix` member functions:
  - Yet there are no affixes in their interfaces!
  - Rather, these functions trim the ends of a string, a term of long standing.
  - So why did we not name them `trim_front` and `trim_back`?
- Speaking of `_front`/`_back`:
  - Shouldn't `push_back` have been better named `append`, …
  - And `push_front` been better named `prepend`?

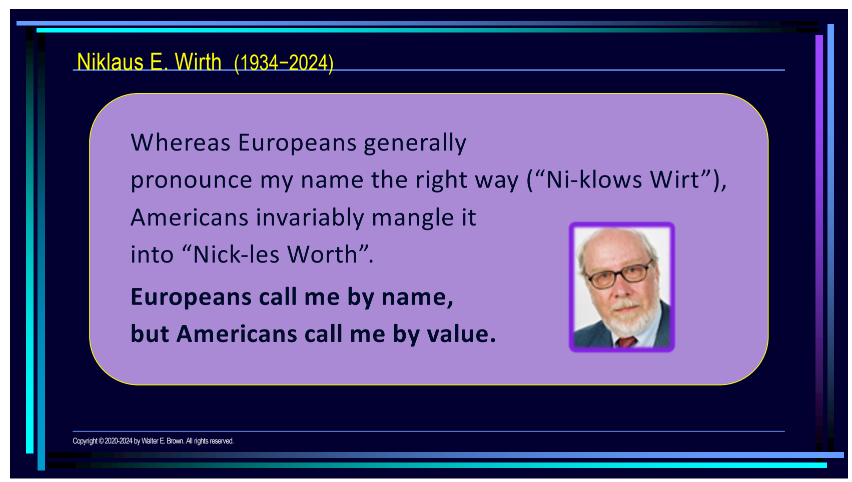
Type relationships

- Among the strongest relationships between two types is inheritance.
- Invented in 1969 (Simula), inheritance is a key feature of C++:
  - `public` inheritance gives us subtyping (the is-a relationship), the basis of Barbara Liskov's substitution principle (1987).
  - `private` inheritance lets us express implementation commonality (the used-as-a relationship), *e.g.*, for mixins.
  - `protected` inheritance permits multi-generational access.
- We commonly say that a derived class inherits from a base class:
  - So why is our library concept for this named `derived_from`?
  - Shouldn't we have named it `inherits_from` (and reversed its arguments)?


So let's all be more aware of our names/terms/nomenclature/vocabulary
"[W]e cannot improve … a science without improving the language or nomenclature which belongs to it."
— Antoine Lavoisier (1743–1794)
"For a single genus, a single name."
— Carl Linnaeus (1707–1778)
"Nothing means anything but the proper names."
— Ursula K. Le Guin (1929-2018)
"[Y]ou won't get your names right the first time.
[Y]ou may well be tempted to leave it — after all it's only a name.
[But h]umans need good names."
— Martin Fowler
Copyright © 2022-2024 by Walter E. Brown. All rights reserved.
37


Why should we bother?
"Because finding good names is a journey of discovery.
"The names we choose shape the dictionary we use to talk and think about our software.
"If we can't find a good name, we obviously don't know enough about either the problem domain or the solution domain."
— Carlo Pescio, 2011
Copyright © 2022-2024 by Walter E. Brown. All rights reserved.
38


Donald E. Knuth
The most important thing
in the programming language is the name. …
I have recently invented a very good name
and now I am looking
for a suitable language.
Copyright © 2020-2024 by Walter E. Brown. All rights reserved.


Niklaus E. Wirth (1934–2024)
Whereas Europeans generally
pronounce my name the right way ("Ni-klows Wirt"),
Americans invariably mangle it
into "Nick-les Worth".
Europeans call me by name,
but Americans call me by value.
Copyright © 2020-2024 by Walter E. Brown. All rights reserved.

Consistency in spelling counts, too

- By convention, the std library uses underscores to set off prepositions that occur within a std library name; *e.g.*:
  - ✓ By: `valueless_by_exception`, `chunk_by_view`.
  - ✓ From: `derived_from`, `constructible_from`, `shared_from_this`.
  - ✓ In/out: `in_out_result`.
  - ✓ Of: `out_of_range`, `alignment_of`, `is_base_of`.
  - ✓ To: `to_underlying`, `to_chars`, `to_string`, `convertible_to`.
  - ✓ With: `common_with`, `swappable_with`, `totally_ordered_with`.
  - ✗ … And then we have `addressof`, `tolower`, `toupper`!

✗ Every such inconsistency is irksome: more to learn/remember/teach.

 41

Speaking of prepositions …

- Header `<utility>` specifies these modest algorithms:
  - ```
    template< class T >
    constexpr add_const_t<T> &      as_const(T & t) noexcept;
    ```
  - ```
    template< class T >
    constexpr underlying_type_t<T>  to_underlying(T value) noexcept;
    ```
- These simple algorithms are really quite similar in structure:
  - Each returns its argument converted to a different type.
  - At first glance, it seems curious, but ultimately reasonable, that the one passes/returns by-ref and the other by-value.
  - But I find it much less reasonable for one to be named `as_⋯` while the other is named `to_⋯`.

✗ Each such inconsistency is irksome: more to learn, remember, and teach.

 42

Those aren't the only ones

- As we saw, `std::as_const` is a function template:
  - So what's `std::as_const_view`?
  - According to header `<ranges>`, it's a type!
    - "`as_const_view` presents a view of an underlying sequence as constant. That is, the elements of an `as_const_view` cannot be modified."
  - I get that it's a nonmodifiable view, *i.e.*, a `const_view`.
  - ✗ I'm just unsure what the `as_` prefix contributes.
- Those examples are just the tip of the `as_⋯` vs. `to_⋯` iceberg:
  - `as_`{`bytes`, `writable_bytes`, `rvalue`, `rvalue_view`}
  - `to_`{`address`, `array`, `bytes`, `chars`, `integer`, `local`, `string`, `wstring`, `utc`, …}

 43

Well, …

 44

IMO, we should strive for technical accuracy in naming

- I wish `operator %` had not been confused with modulus/modulo/mod.
- It's actually known as the remainder operator (per [expr.mul]/4):
  - "The binary `%` operator yields the remainder from the division of the first expression by the second."
- Yet the standard library specifies (in [arithmetic.operations.modulus]/1):
  - "Class template `modulus`" that "*Returns*: `x % y`".
- Generally, `operator %` corresponds to "modulus" only when `x >= 0` and `y > 0` (although math's definitions do vary slightly).
  - In my own library, I renamed this template `remainder` to remove any doubt.

boolishness ☺

- I dearly wish C++ had never classified `bool` as an integer type.
  - Did anyone ever have a good reason to want the likes of either `true + false` or `42 * true` to be valid expressions?
- Here's some code I recently found in the wild (reformatted to fit here):
  - ```
    return dividend / divisor
         + (abs_remainder >= abs_half_divisor
           + (divisor % T{2} || quotient_sign < 0)
           ) * quotient_sign;
    ```
  - Note the arithmetic conflation of `int`s, `bool`s-as-`int`s, and `int`s-as-`bool`s?
  - How long does it take even a seasoned programmer to grok what's going on?

BTW, we really must better teach `bool`

- I was recently shown this code (class names sanitized):
  - ```
    class D : public B
    { ⋮
      bool isEnabled() override
      {
        if( ! B::isEnabled() ) {
          return false;
        }
        return true;
      }
      ⋮
    };
    ```
    ```
    return B::isEnabled();
    ```
    So why is this function overriding the base class at all?
  - Says the submitter: "I think this is a beautiful little smear of bad code, because it's useless on multiple levels."

Excessive abstraction

- `std::ranges::in_out_result` is a class template that allows us to store two iterators as a single entity:
  - ```
    template< class I, class O >
    struct in_out_result { I in; O out;  … };
    ```
- "Each standard library algorithm that uses this family of return types declares a new alias type ….":
  - ```
    template< class I, class O >
    using copy_result = in_out_result<I, O>;
    ```
- So now, each time I call `copy`, I have to look up what a `copy_result` is:
  - Why not just use `in_out_result` as the result type?
  - That name tells me exactly what I need to know!

"It is not the case that I have failed to avoid being not unafraid."

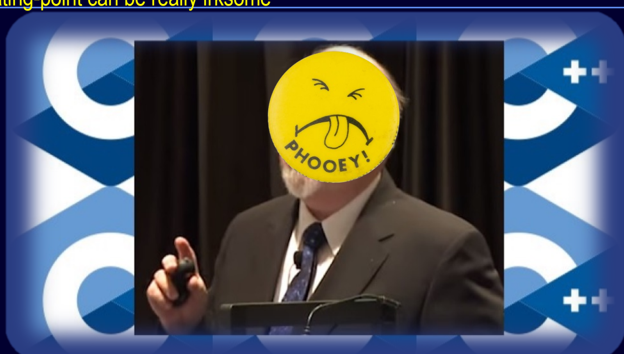
- The std library specifies a few variable templates of type `bool`:
  - `<ranges>`: `enable_view`
  - `<ranges>`: `disable_sized_range`
  - `<iterator>`: `disable_sized_sentinel_for`
- That's more than I can comfortably remember; I needs must look it up every time!
- Negative names and phrases tend to be troublesome for many:
  - *E.g.*, instead of writing: `while (not done) …`
  - We could just as easily instead code: `while (more_to_do) …`

And then there's the tale of floating-point literals

- I expect each of these assertions to pass everywhere; don't you?
  - `static_assert(        3.14F == 3.14F );`
  - `static_assert( (float)3.14F == 3.14F );`
  - ```
    constexpr auto x = 3.14F;
    assert       ( x == 3.14F );
    static_assert( x == 3.14F );
    ```

✗ Yet all but the first may fail!
  - Why? It depends on the setting (in `<cfloat>`) of macro `FLT_EVAL_METHOD`.
  - *E.g.*, when `FLT_EVAL_METHOD == 2` (as it is in GCC for x86 with i387 math), `3.14F` is evaluated as `long double`, not as `float`.
  - Even though the literal's type is still `float`.

Floating-point can be really irksome



A classic

- Among the most well-known std library botches:
  - Specializing `vector` w/out preserving its basic container functionality, …
  - Namely `vector<bool>`.
- Since `vector<bool>` doesn't meet the container requirements …
  - It's not guaranteed to work properly with std library algorithms.
  - Why not? In part because it's a proxy container.
  - Asking for a reference to a `bool` in the vector gives you a proxy whose behavior is reference-like, but whose type isn't a true reference type.
- So why do we have `vector<bool>` in the standard?
  - "Some think it was an experiment which went wrong and somehow stayed in the standard for backwards compatibility."
    ```

And then there's initialization: how many forms do we have? [N. Josuttis]

1) Direct initialization — explicit initial value
2) Copy initialization — uses = syntax; affected by explicit declaration
3) Default initialization — via corresponding c'tor
4) Zero initialization — via (possibly converted) 0
5) Value initialization — via c'tor or 0
6) List initialization — uses { … } syntax
7) Aggregate initialization — special list initialization for an aggregate type

 53

So, there are seven ways to declare/initialize an int, right? [N. Josuttis]

- `int k0;`
- `int k1 = 42;`
- `int k2( 42 );`
- `int k3 = int( );`
- `int k4{ 42 };`
- `int k5{ };`
- `int k6 = { 42 };`
- `int k7 = { };`
- `auto k8 = 42;`
- `auto k9{ 42 };`
- `auto k10 = { 42 };`
- `auto k11 = int{ 42 };`
- `int k12( );` // ✘ *(it's a function!)*
- `int k13( 4, 2 );` // ✘
- `int k14 = ( 4, 2 );` // *(comma op)*
- `int k15 = int( 4, 2 );` // ✘

 54

A decade ago, we deprecated the red squares [H. Hinnant, 2014]

Special Members

compiler implicitly declares

user declares	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

- "I think eventual removal by whatever option is a good direction."
- "Today, the most likely guess is that they will never be removed. Today compilers are even reluctant to turn this deprecated warning on by default."

 55

Well, …

 56


Irksome C++
FIN
Walter E. Brown, Ph.D.
< webrown.cpp @ gmail.com >
#$*@&!
Copyright © 2022–2024 by Walter E. Brown. All rights reserved.