A Medley of C++

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 C++17's <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 origin(?) of "Lightning Talks"

- "A *lightning talk* is a very short presentation … given at a conference or similar forum."
- It's claimed that:
  - "The term was first coined … in June 2000 [although] …
  - "The practice of lightning talks was first known to be used in 1997."
- Yet I remember that:
  - At the 1st programmers' conference I ever attended …
  - We had an hour of short talks under the common title, "**What Every FORTRAN Programmer Ought to Know**."

 6

Today: an homage

- That 1971 talk was memorable (at least to me):
  - In part because not every topic needs a full hour, right?
- So I have for you today several mini-talks:
  - In the approximate style that I witnessed ~50 years ago.
  - With C++ (rather than Fortran) as their unifying theme.
- As the common title for this talk, I first thought of:
  - "**What Every F C++ Programmer Ought to Know**."
  - But that seemed too obvious; too much like a clone.
  - Instead, I chose the simpler "**A Medley of C++**"

 7

Today's program

① "The Universe of C++ Types"

② "… C++ Declaration Syntax"

③ "Why Are They Named *Lambdas*?"

④ "What Does C++20 Owe to …"

 8

Today's program

⑤ "A Review of Published Code"

⑥ "An Adventure in Applied Detection"

⑦ "Expression Categories"

⑧ "decltype, auto, and decltype(auto)"

⑨ "It's Not Only Unique"

9

A Medley of C++

FIN

Walter E. Brown, Ph.D.

< webrown.cpp @ gmail.com >

The Universe of C++ Types

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!)

 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*, 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 C++17's <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! ☺

 3

Let's explore/partition this universe

- The C++ *primary* type classifications do not overlap:
  - The core language specifies these in *[basic.types]*.
  - The standard library specifies corresponding type traits in *[meta.unary]*.
  - ☺ Fortunately, they mostly agree.
- A type's *cv*-qual's does not affect its classification.
- But here's a related puzzle for you (answer at the end):

For which primary classification(s) of types ***T*** would the following predicate yield false?

std::is_const_v< ***T*** const >

 4

Fundamental types

- void *types*
  - There are 4 (counting *cv*-qualification).
- std::nullptr_t *types*:
  - Like the voids, these 4 have their own classification.
- *Arithmetic types*:
  - *Floating-point types*: { float, double, long double }*
  - *Integral/integer types*: See next page.

* More coming in C++23 (*e.g.*, bfloat16); details in P1467.

 5

Integral types, a.k.a. integer types

- { bool, char, wchar_t, char8_t, char16_t, char32_t }
- *Signed integer types*:
  - *Standard signed integer types*:
    signed { char, short int, int, long int, long long int }
  - *Extended signed integer types*:
    Implementation-defined (but does anyone?).
- *Unsigned integer types*:
  - *Standard unsigned integer types*:
    unsigned { char, short int, int, long int, long long int }
  - *Extended unsigned integer types*:
    Implementation-defined (again, does anyone?).

 6

Compound types (*i.e.*, types based on *n* ≥ 1 other types)

- *Arrays of* { *known, unknown* } *extent*:
  - Homogeneous; *i.e.*, composed of objects (elements) of a single specified type.
- { *Lvalue, rvalue* } *references*:
  - To an { object, free function, static member function } of some specified type.
- *Pointers*:
  - To void, or …
  - To an { object, free function, static member function } of some specified type.

 7

Compound types (continued)

- *Functions*:
  - Have *m* ≥ 0 parameters of specified types, and …
  - A bool value denoting its noexcept status, and …
  - A return { void, object, reference } type.
- *Classes*:
  - Contain a sequence of *m* ≥ 0 objects of various (*i.e.*, possibly heterogeneous) specified types, and …
  - Contain a set of specified types, enumerations, and functions for manipulating these objects, and …
  - Contain a set of restrictions on these members' access.

 8

Compound types (concluded)

- *Unions*:
  - Classes that may contain objects of different specified types …
  - At different times.
- *Enumerations { scoped, unscoped }*:
  - Comprise a set of named constant values …
  - That share a single underlying specified integral type.
- *Pointers-to-member-of*:
  - Identify non-static { data, function } members of a specified type …
  - Within objects of a specified class type.

 9

Exactly one of these traits will be true for every type *T*

… _v< T > (red = fundamental; blue = compound)	
is_void	is_lvalue_reference
is_null_pointer	is_rvalue_reference
is_signed	is_member_object_pointer
is_unsigned	is_member_function_pointer
is_floating_point	is_enum
is_bounded_array	is_union
is_unbounded_array	is_class
is_pointer	is_function

 10

Some composite type classifications

- *Class types*:
  - All types declared with a *class-key* { class, struct, union }.
- *Function object types* (std library def'n):
  - All ptr-to-function types, and …
  - All class types with an operator( ) member, and …
  - All class types with a conversion function whose target type is a { ptr, ref, ref-to-ptr }-to-function type.
- *Callable types* (std library def'n):
  - All { function object, ptr-to-member } types.

 11

A few more composite type classifications

- *Scalar types*:
  - All { arithmetic, enumeration, pointer, pointer-to-member, std::nullptr_t } types.
- *Object types*:
  - All non-{ function, reference, void } types.
- *Incomplete types*:
  - All void types, and …
  - All declared-but-not-defined class types, and …
  - All enumerations "in certain contexts", and …
  - All array types such that { bound is unknown, element type is incomplete }.

 12

Example: detecting a type's completeness (since C++11)

- Apply (a variant of) the *detection idiom*:
  - `template< class, class = std::size_t >` *// primary template*
    `bool is_complete_v = false;`
  - `template< class T >` *// partial specialization*
    `bool is_complete_v<T, decltype( sizeof(T) ) > = true;`
- Use such a trait carefully! For some types, the answer might be different at different points in your program:
  - *E.g.*, `false` after a class's forward declaration (incomplete) vs. `true` after that class's later definition (now complete).
  - Such inconsistency violates the C++ One Definition Rule (ODR), so query a type's completeness at most once.

 13

Finally: the puzzle's answer

- `std::is_const_v< T const >` yields `false` when …
  - `T` is either a *reference* type or a *function* type.
- Perhaps this is not useful knowledge every day, but:
  - Historically, the `std::is_function` trait was implemented via a primary template + 48 (!) partial specializations:
  - 48 = 4 *cv* qual's × 3 ref qual's × 2 `noexcept` × 2 ellipsis arg.
  - These don't include non-standard calling conventions!
  - ```
    template< class T >
    bool is_function_v
      = not is_const_v< T const >    // true iff ref-or-fctn type
    and not is_reference_v< T >;     // now rule out ref type
    ```

 14

The Universe of C++ Types

FIN

WALTER E. BROWN, PH.D.

< webrown.cpp @ gmail.com >


    ```

The Principle behind C++ Declaration Syntax

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!)

 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*, 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 C++17's <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! ☺

 3

Some claim our declarations are impossible, but …

"Impossible is not a declaration.

It's a dare."

— Muhammad Ali

 4

C declarations' guiding principle (emphasis added)

"The style used in expressions carries through to declarations…."

— Dennis M. Ritchie,
The Development of the C Language
(presented at HOPL-II, 1993)

 5

From expressions to declarations

- That principle applies in C++, too — let's look at some examples.
- For an entity x, what expressions could give a result of type int?
  - If a simple variable: x.
  - If an array: x[⋯].
  - If a pointer: ∗x.
  - If a function: x(⋯).
- So, how would x be declared in each of these cases?
  - int x; *// simple variable*
  - int x[⋯]; *// array*
  - int ∗x; *// pointer*
  - int x(⋯); *// function*

 6

The parts of a typical declaration, left to right

- First we write a type, followed by …
- An expression (declarator) of a form that could give a value of that type, and then ...
- Punctuation (a semicolon if at end, else a comma).
- Examples (redux):
  - `int x;` *// simple variable*
  - `int x[ ⋯ ];` *// array*
  - `int *x;` *// pointer*
  - `int x( ⋯ );` *// function*
- Redundant paren's:
  - `int (x);`
  - `int (x[ ⋯ ]);`
  - `int (*x);`
  - `int (x( ⋯ ));`

 7

Precedence applies, even in declarators

- *E.g.*, recall that operator ( ) has precedence over (binds more tightly than/has seniority over) operator * :
  - So in an expression, operator ( ) claims all its operands …
  - Before operator * begins to claim its operands.
- Similar expressions can therefore differ in meaning:
  - `* p( ⋯ )` *// p is callable, and …*
    // … the call's result is dereferenceable
  - `(*q)( ⋯ )` *// q is dereferenceable, and …*
    // … the dereferenced result is callable
- That difference is reflected in their resp. declarators:
  - `int * p( ⋯ );` *// p has a function type* `int*( ⋯ )`
  - `int (*q)( ⋯ );` *// q has a pointer-to-function type* `int (*)( ⋯ )`

 8

Multiple declarators

- Multiple declarators may share a common type:
  - `int k, a[ 10 ], *p, g( double );`
  - `int k` *// known as the One True Comma layout*
    `, a[ 10 ]`
    `, *p`
    `, g( double )`
    `;`
- Even class members can be declared to share a type:
  - ```
    struct S {
      int zero = 0, one[ 1 ] = { 1 }, two( );
    };
    ```
  - `int S :: two( ) { return 2; }`

 9

According to *[dcl.decl.general]/3*

- "A declaration with several declarators is usually equivalent to the corresponding sequence of declarations each with a single declarator."
- But not quite always:
  - ✗ `auto i = 1, j = 2.0;` *// error: deduced types do not match*
  - ✓ `auto i = 1;` *// okay, i deduced to have type* int
    `auto j = 2.0;` *// okay, j deduced to have type* double
  - ✓ `struct S { ⋯ };`
    `S S, T;` *// declare two variables of type S*
  - ✗ `struct S { ⋯ };`
    `S S;` *// okay (but please don't do this!)*
    `S T;` *// error: variable S hides type S*

 10

More syntax options

- We earlier said: "First we write a type, …":
  - We did **not** say, "First we write a type name, …."!
- Examples:
  - `struct S { ⋯ } s, t;`
    // defines type S, also defines variables s and t
  - `struct { ⋯ } u;`
    // defines variable u with an unutterable type

 11

C++ offers still more syntax options

- Attributes: `[[ ⋯ ]]` .
- Initializers: `= ⋯`, `( ⋯ )`, `{ ⋯ }` .
- Qualifiers: const, volatile, &, && .
- Specifiers: constexpr, consteval, constinit, extern, friend, inline, mutable, static, thread_local, typedef, virtual, ….

 12
    ```

Recap of the main takeaway (please teach this!)

"The style used in expressions carries through to declarations…."

— Dennis M. Ritchie,
The Development of the C Language
(presented at HOPL-II, 1993)

 13

This principle applies even to typedefs

- To write a traditional typedef:
  - Pretend to write a variable's declaration, then …
  - Just write the keyword typedef in front!
- Example:
  - A variable of a ptr-to-fctn type: `int (* ptf)( float );`
  - An alias for such a type: `typedef int (* ptf)( float );`
  - Since C++11, equiv. to: `using ptf = int (*)( float );`

 14

Outside the C subset, C++ had to adapt

- Recall (a) that C has no reference types, and (b) that C++ has no expressions of reference type:
  - "If an expression initially has the type *reference to T*, the type is adjusted to *T* …" (see *[expr.type]/1*).
  - Thus, no C++ operator can produce any ref. type.
- Instead, C++ adapted pointer syntax to declare an entity of reference type:
  - We use & or && in the declarator instead of *.
  - Example: `int * x ⋯ ;` *// x has pointer type*
    compare to: `int & y ⋯ ;` *// y has lvalue-ref type*
    and to: `int && z ⋯ ;` *// z has rvalue-ref type*

 15

One more C++ innovation was needed …

- … since C has no scope-resolution operator (::).
- Example:
  - `struct S { bool b; double d( float ); };`
  - Member `S :: b` has type `bool`, an object type.
  - Member `S :: d` has type `double( float )`, a fctn type.
- Declaring a pointer-to-member both (a) names the containing class and (b) gives the pointee's type:
  - `bool S :: * pb = & S :: b;`
  - `double ( S :: * pd ) ( float ) = & S :: d;`

 16

A final caveat

- This is, of course, not quite the complete story.
- For example, we haven't mentioned the role of type aliases in declaring other entities.
- Nor have we discussed the forbidden ref-to-ref (a.k.a. reference-collapsing) types.
- Nor have we described such recent additions as late-specified return types or structured bindings.
- So, as seems typical in C++:
  - There's always more to learn!

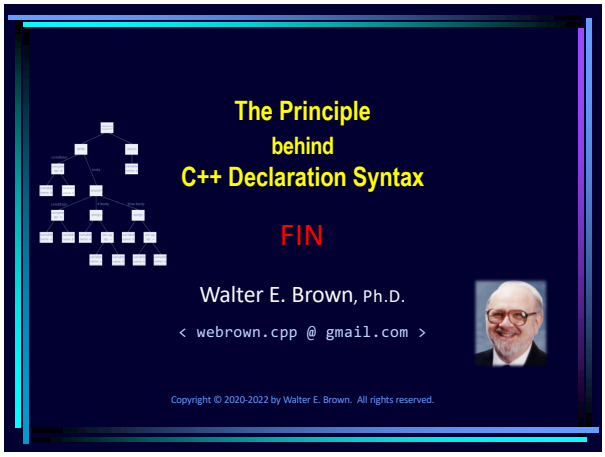
 17

Nonetheless, we have a principle to guide us

"The style used in expressions carries through to declarations…."

— Dennis M. Ritchie,
The Development of the C Language
(presented at HOPL-II, 1993)

 18



α? β? ζ? θ? μ? δ?

Why Are They Named *Lambdas*?

σ? γ? π? ω?

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!)

 2

Emeritus participant in C++ standardization

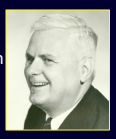
- Written ~175 papers for WG21, proposing such now-standard C++ library features as gcd/lcm, cbegin/cend, common_type, and void_t, as well as all of headers <random> and <ratio>.
- Influenced such core language features as *alias templates*, *contextual conversions*, and *variable templates*; recently 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 C++17's <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! ☺

 3

We owe much to these pioneers of computation …

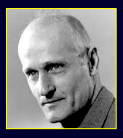
Alan Turing
1912–1954

Emil Post
1897–1954



Alonzo Church
1903–1995

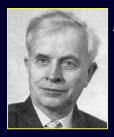
Kurt Gödel
1906–1978



Stephen Kleene
1909–1994

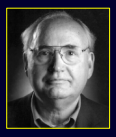
 4

… As well as to these pioneers of computation

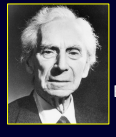


Andrey Markov, Jr.
1903–1979

Axel Thue
1863–1922

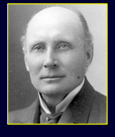


Dana Scott
1932–



Bertrand Russell
1872–1970

Alfred Whitehead
1861–1947



 5

From theory to practice

- They researched the foundations of mathematics:
  - Eventually their respective works largely converged into a branch of math's known as computability theory.
  - But Church's notation became the foundation of the LISP programming language [McCarthy, 1958].
- "Lisp is today's equivalent of Latin. Educated people are supposed to have studied and forgotten it."

— Peter Dimov,
Simple C++11 metaprogramming,
2015

 6

Functional programming lives on

- In such higher-order programming, functions are not only callable, but are copyable values, too:
  - In C++, function objects have always fit this description, …
  - So C++11 introduced "lambda-expr's [as] a concise way to create a simple function object" (see *[expr.prim.lambda]/1*).
  - And, as we know, C++ lambdas have evolved ever since.
- Some consider that "the untyped λ-calculus was the first object-oriented language."

— William R. Cook,
On Understanding Data Abstraction, Revisited,
OOPSLA 2009

 7

Origin of lambda as a term of art

- Today, math functions are named (*e.g.*, *f*(y) = 3y + 1):
  - (It's believed Leonhard Euler was first to do so [1734].)
- But Alonzo Church used [1930s] anonymous fctn def'ns (*e.g.*, ŷ . 3y + 1) in his computing foundations work:
  - He'd adapted the ŷ notation (for function-abstraction) from Whitehead & Russell's notation (for class-abstraction).
  - (BTW, that's the form I first learned in graduate school.)
- Church wrote [1964] that when he published his work:
  - His caret (hat, ^) was typeset separately (*e.g.*, Λy . 3y + 1) …
  - Thus resembling a Greek capital Lambda, that was later changed to lower case (*e.g.*, λy . 3y + 1).

 8

However, …

- Dana Scott (Church's former student and 1976 Turing Award recipient) told a different story in his [2018] talk *Looking Backward; Looking Forward*.
- Scott said that he once appealed to Church's son-in-law (John Addison) to ask Church the origin of lambda:
  - So Addison wrote Church a postcard, asking …
  - "… Why did you choose lambda as your operator?"
- Church allegedly returned the postcard in an envelope, having annotated the postcard with the words, "eenie, meenie, miney, mo"!

 9

In either case …

- Church's notation for functions/function abstraction became known as lambda expressions, …
- And Church's collected work in this area became known as the lambda calculus, …
- Long since recognized as a universal model of computation …
- That's been proven equivalent in computational power to the models set forth by Post, Kleene, and Turing, among others.

 10

Why Are They Named *Lambdas*?

FIN

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

What Does C++20 Owe to Prof. Dr. Emmy Noether?

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!)

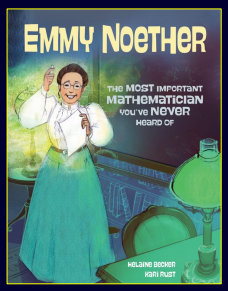
 2

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*, 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 C++17's <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! ☺

 3

"The Most Important Mathematician You've Never Heard Of"



"Scientists are a famously anonymous lot, but few can match [the] perverse and unmerited obscurity [of] the 20th-century mathematical genius Amalie [Emmy] Noether."

— Natalie Angier,
The New York Times, 2012-03-26

 4

Emmy Noether (1882-1935) [adapted from A. Borschel-Dan]

- Although she was a mathematician's daughter, she was not allowed to enroll in Univ. classes:
  - The academic senate warned that coed education would "overthrow all academic order." *E.g.*, a faculty member said, "What will our soldiers think when they … find that they are required to learn at the feet of a woman?"
  - To which mathematician David Hilbert famously replied, "I do not see that the candidate's gender is an argument against her admission…. After all, we are a university, not a bathhouse."

- Noether was at last officially permitted to study, but had to ask permission from each lecturer she attended.

 5

At the start of her career

- After winning the right to matriculate, Noether earned a Ph.D. in 1907, but no faculty jobs were then open to women:
  - So she worked without pay for 7+ years.
  - Then "hired" by Hilbert at the Univ. of Göttingen in 1915, but spent years lecturing under his name, still unpaid.
- After completing a second doctorate (*Habilitation*, 1919), she was finally appointed to a faculty position in 1923 as *Lehrbeauftragte für Algebra*.

 6

"… Emmy Noether, Einstein's Equal"

- "Noether continued to teach as she produced papers and theories at a staggering pace," including many rarely acknowledged contributions to works written by her students and colleagues.
- "She developed a reputation as a fast talker, a stern critic, and a devoted teacher."

— Troy Brownfield,
The Saturday Evening Post, 2018

- A decade later (1933), she fled to the U.S.A. to escape Nazi persecution.

 7

In the USA, she joined Bryn Mawr, with ties to Princeton's IAS

- "Dr. Noether is the most eminent woman in mathematics in Europe and has had more students at Göttingen than anyone else in the department."

— Dr. Marion Edwards Park
President, Bryn Mawr College

- "Miss Noether's methods of thinking and working … were [to] recognize the unessentials [and] brush them aside…. This was … far from a superficial achievement…."

— Dr. Ruth Stauffer,
Noether's last doctoral student

 8

Some of Noether's significant contributions

- "The development of abstract algebra, … one of the most distinctive innovations of 20th century math's, is largely due to her – in her published papers, in lectures, and in … influence on her contemporaries."

— Nathan Jacobson

- *Noether's Theorem* [1915, pub. 1918] is "one of the most important … theorems ever proved in … the development of modern physics, possibly on a par with the Pythagorean theorem."

— Leon Lederman & Christopher Hill

 9

"Emmy Noether saved General Relativity" w/ her theorem

- "[Since] 1915, General Relativity, a new … way of thinking about gravity, had captured the attention of physicists…. But [this new theory had] difficulties which even Einstein could not resolve.
- "We would likely not be celebrating this landmark theory were it not for [Emmy Noether] who, at her prime, couldn't even secure a teaching role in her homeland because of her gender.
- "[Noether's Theorem provided] the mathematical breakthroughs that general relativity needed to win over physicists."

— Robert Lea

 10

But Noether is now recalled by relatively few of us

- Her mathematical originality was "absolute beyond comparison"

— Bartel L. van der Waerden

- She "changed the face of algebra by her work."

— Hermann Weyl

- She "is … the greatest woman mathematician who has ever lived; and the greatest woman scientist of any sort now living, and a scholar at least on the plane of Madame Curie."

— Norbert Wiener

 11

A very special posthumous tribute [excerpted]

- **"Within the past few days a distinguished mathematician, Professor Emmy Noether, … died in her fifty-third year.**
- **"In the judgment of the most competent living mathematicians, [she] was the most significant creative mathematical genius thus far produced … ."**

Bryn Mawr College Grounds

Albert Einstein

— Albert Einstein,
"Professor Einstein Writes in Appreciation of a Fellow-Mathematician,"
The New York Times, 1935-05-05

 12

A biographer summarizes

- "In her 53 years,
  - "many spent bucking a system that impeded her pursuit of mathematics,
  - "Noether had an extraordinary impact on both algebra and physics.
- "There's no telling what else she might have accomplished if society and fate had been more kind."

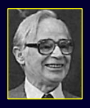
— Steve Nadis,
The Universe According to Emmy Noether

13

What does Noether's work have to do with C++20?

- "It was she who taught us to think in terms of simple and general algebraic concepts…"

— Pavel S. Alexandrov



- "For Emmy Noether, relationships among numbers, functions, and operations became … amenable to generalisation … only after they have been … reduced to general conceptual relationships…."

— Bartel L. van der Waerden

14

"The greatest thing … in 20th century mathematics!"

- "What was [Noether's] idea? [It was that y]ou could separate concepts from implementation. You could just deal with concepts….
- "So, if anybody invented generic programming, she did…. She realized it all. She realized that you could fully separate these two things.
- "Her contribution needs to be remembered."

— Alexander Stepanov, 2003

15

Emmy Noether, "Mathematical genius"

"Even now, the world still learns from Noether, whose abstract principles are fundamental…."

— *Emily Barone, 2020*

16

C++20 Owes Concepts to Prof. Dr. Emmy Noether ע"ה

"Her contribution needs to be remembered."

Walter E. Brown, Ph.D.

< webrown.cpp @ gmail.com >

matching bytes:

A Review of Published Code

Walter E. Brown, Ph.D.

< webrown.cpp @ gmail.com >

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!)

 2

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*, and *variable templates*; recently worked on *requires-expressions*, operator<=>, and more!
- Conceived and served as Project Editor for ISO/IEC 29124: *Int'l Standard on Mathematical Special Functions in C++*, now part of <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! ☺

 3

3

Background

- A while ago, I read a blog post that started with this problem statement:
  - "Suppose that you give me two ASCII strings having the same number of characters.
  - "I wish to compute … the number of matching characters (same position, same character)."
- "The conventional approach in C would look as follow[s]:"
  - [code block, forthcoming]
  - "There is nothing wrong with this code. …"
- While there may indeed be "nothing wrong" with the *algorithm*, I respectfully opine that the published *code* is not of publication quality.

 4

4

Before I show you the code …

- Please, please don't seek out the blog's original author:
  - I truly have no wish to embarrass anyone, …
  - Nor do I wish to start any sort of code war or feud.
- Instead, kindly accept this review as an opinion:
  - Which, as always, you are free to adopt, adapt, or reject.
- Finally, please keep in mind that the blog article's main thrust was the speedup of the initial algorithm, …
  - Which I found to be an interesting analysis, but comparing algorithms is not my topic today.
  - Rather, this short talk is about code quality and about making incremental improvements toward quality code.

 5

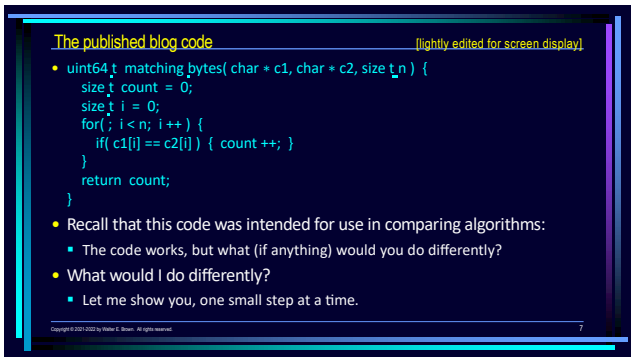
5

My approach to code review

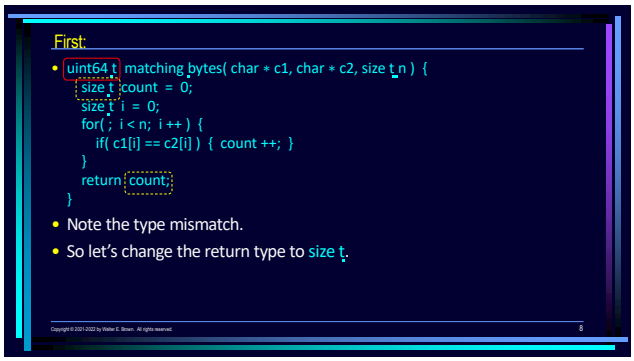
- When assessing code, I seek good things to praise:
  - *E.g.*, when I (once upon a time) graded students' programs, …
  - I looked for reasons to reward with a better grade, …
  - Rather than for reasons to penalize with a poorer grade.
- In my rubric at the time, code that:
  - Fails even to compile merited only an F (a failing grade).
  - Compiles and runs, but crashes or gives incorrect results, merited only a D (a barely-passing grade).
  - Runs to completion and gives correct results merited at least a C (a grade of minimal competence).
  - After that, I sought evidence of further programming lessons learned.

 6

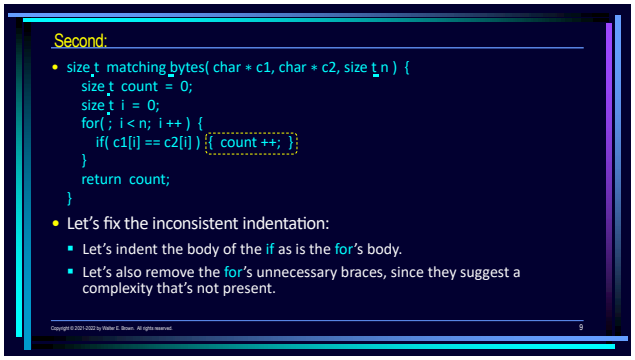
6



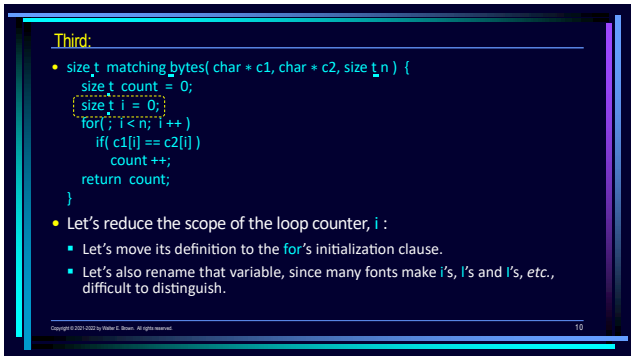
7



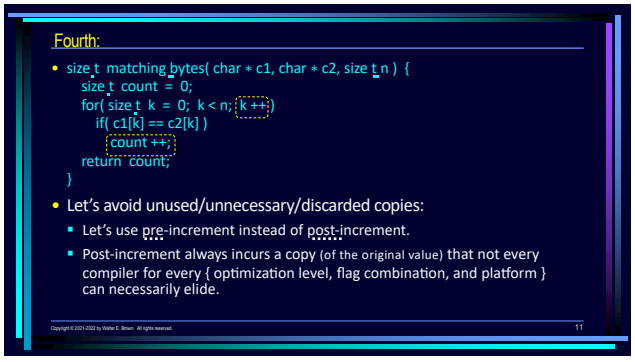
8



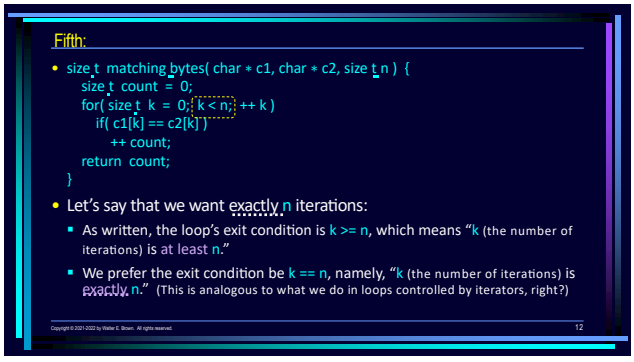
9



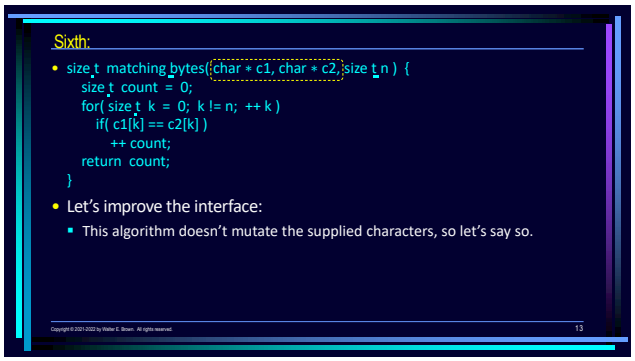
10



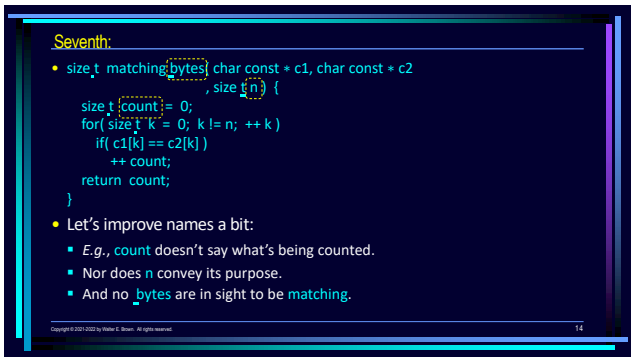
11



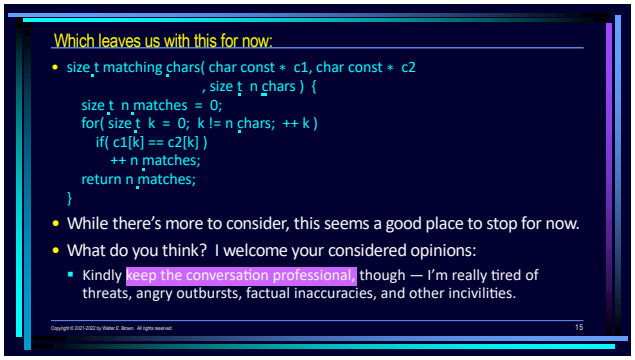
12



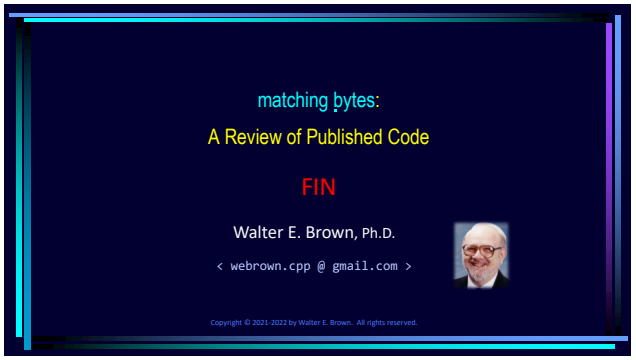
13



14



15



16