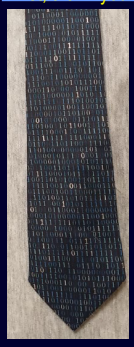
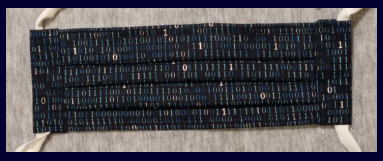
First, thank you to someone special





1

What's in a Name? and What's a Name In?

Walter E. Brown, Ph.D.

< webrown.cpp @ gmail.com >

2

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!)

3

Emeritus participant in C++ standardization

- Written ~170 papers for WG21, proposing such now-standard C++ library features as gcd/lcm, cbegin/cend, common_type, and void_t, as well as all of headers <random> and <ratio>.
- Influenced such core language features as *alias templates*, *contextual conversions*, and *variable templates*; recently worked on *requires-expressions*, operator<=>, and more!
- Conceived and served as Project Editor for *Int'l Standard on Mathematical Special Functions in C++* (ISO/IEC 29124), now incorporated into 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! ☺

4

It's been said ...

The beginning of wisdom
is to call things by their right names.
— **Confucius** (paraphrase)

The power of the name
is the most elementary of all.
— **David Eddings**, 1988

... and, as every wizard knows,
once you have a thing's real name
you have the first step to its taming.
— Sir **Terry Pratchett**, 2008

5

The topic is names, but ...

- We won't today discuss choosing names — others have already done that during past conferences.
- Rather, we'll talk about processing names ...
- As handled by our compilers, linkers, *etc.*
- We'll see why this is such a deep topic for C++ programmers in particular.
- Assuredly we've all been taught scope rules during our early training.
- However, ...

6

Caveats about the "scope" of this talk

- The topic of C++ names is so intricate that no single one-hour talk would be enough to cover it in full.
- To make matters worse:
  - WG21 regularly discovers new edge cases and resolves them via oft-subtle tweaks to the C++ grammar/rules.
  - New language features (*e.g.*, C++20's modules) introduce new rules and update details of existing rules.
  - Proposal N1787, a major overhaul of the C++20 rules, is under review for C++23.
- So this talk will focus on underlying principles, with several examples of their effect on our C++ code.

 7

7

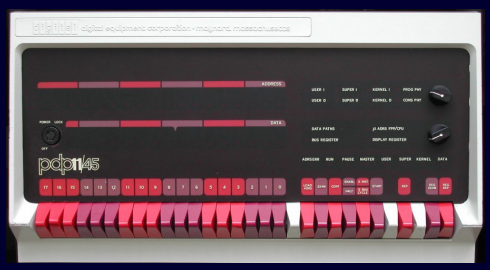
How did we programmers start using names?

- Early assembly languages (vintage 1950..1970) were commonly termed symbolic programming systems ...
  - Largely because they featured the use of names in lieu of memory addresses ...
  - Which proved to be a huge productivity improvement ...
  - As well as an enormous correctness improvement!
- Since then, names have became ubiquitous at all programming levels above bare machine language...
  - Although as late as 1980 it was not uncommon for me to toggle machine code into memory one bit at a time!

 8

8

Front panel, color-coded for octal input



 9

9

If you were programming circa 1951

The following narration is by computing pioneer and Turing Award recipient Sir **Maurice Wilkes** (1913–2010)

 10

10

So how do we program with names?

- Sometimes, we use names (*e.g.*, enums) just as symbols for specific values. (Not limited to programming: π, e, ϕ.)
- Other program names (of functions, variables, labels, ...) are transformed into machine addresses (lvalues):
  - Done by our system software (compiler, linker, loader, ...).
  - Allows us to avoid the tedious details and complexity of directly/manually dealing in machine addresses.
  - (Witness the huge percentage of bugs due to pointers.)
- Few program names survive into executable binaries:
  - Only those needed for dynamic linking/loading, and ...
  - Optionally, those retained for debugging purposes.

 11

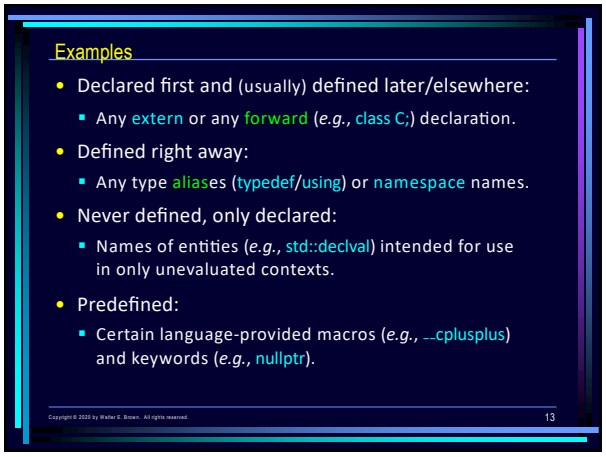
11

C++ names

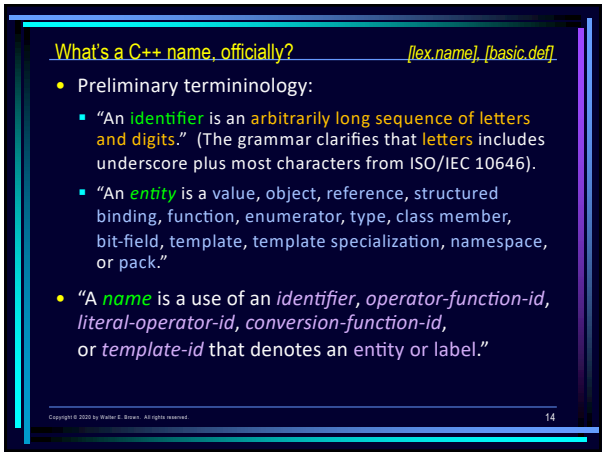
- Like many programming languages, C++ requires that:
  - Before using a name, we must introduce it, providing ...
  - An indication of what that name denotes and connotes.
- In C++:
  - A declaration is an introduction w/ partial information.
  - A definition is a declaration w/ complete information.
- Depending on what's being named and how it's used:
  - Some C++ names can be declared first and defined later.
  - Some C++ names must be defined right away.
  - Some C++ names need never be defined, only declared.
  - Some C++ names are predefined.

 12

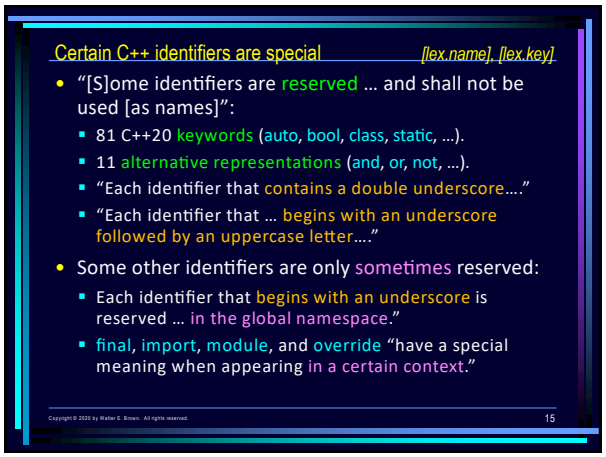
12



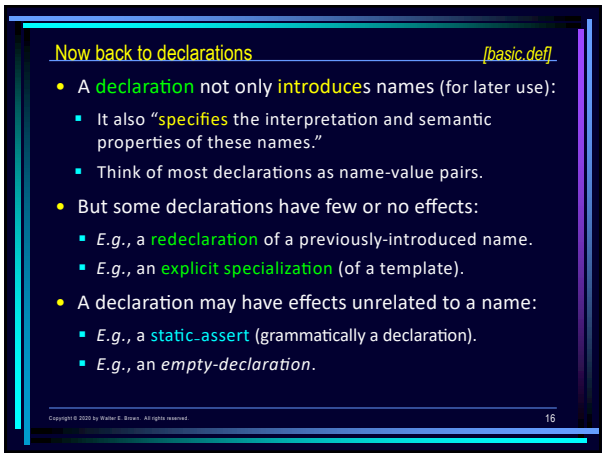
13



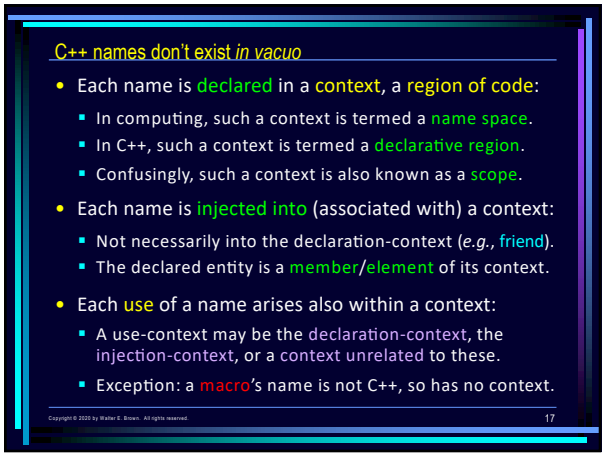
14



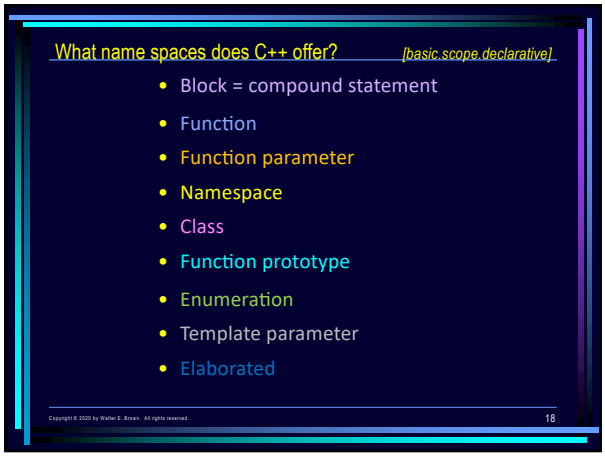
15



16



17



18

Name injection oddities ①

- A newly-introduced name is usually injected into its declaration-context, *i.e.*, its containing name space.
- By its nature, a friend declaration doesn't (and can't) behave that way:
  - Why? Any name being granted friendship isn't a member of the class that's granting the friendship.
  - (If it were a member, friendship would be unnecessary!)
  - So the name is instead injected into the innermost (nearest) enclosing namespace.
  - But unqualified name lookup won't find it there unless there's also a matching declaration in that namespace.

 19

19

Name injection oddities ②

- Like most declarations, a using declaration (*e.g.*, using std::sqrt;) injects its name into its declaration-context.
- However, a *using-directive* (*e.g.*, using namespace std;) follows a different rule:
  - "The names [in the nominated namespace] appear as if they were [injected into] ...
  - "The nearest enclosing namespace [that] contains both the *using-directive* and the nominated namespace."
- An anonymous namespace behaves as if a *using-directive* nominated it in the enclosing namespace.
- And so does an inline namespace.

 20

20

Example *[basic.scope.namespace]*

```
namespace N {
  int k;
  int g( int a ) { return a; }
  int p( );
  void q( );
}
namespace { int b = 1; }
  ⋮
```

```
namespace N {
① int g( char a ) {
②   return b + a; }
③ int k;
④ int p( );
⑤ int p( ) { return g(k); }
⑥ int q( );
}
```

① overloads int N::g(int)
② b vis. in the global namespace
③ no: k is already defined!
④ redeclares p
⑤ redeclares and now defines p
⑥ no: inconsistent redeclaration!

 21

21

What happens whenever we use any declared name?

- The compiler must figure out that name's properties.
- Such information is in the name's declaration, so:
  - The compiler treats the name being used as a key ...
  - To locate that name's corresponding declaration.
- This ~~scavenger hunt~~ mapping is termed name lookup:
  - A family of incredibly intricate interacting algorithms, replete with special cases, deemed by several experts to be "the most complicated part of C++."
  - *E.g.*, "Lookups in which function names are ignored include names appearing in a *nested-name-specifier*, an *elaborated-type-specifier*, or a *base-specifier*."

 22

22

Why are name lookup algorithms so complicated?

- Imagine using a name within the body of a for-loop ...
- That's part of the body of a function ...
- That's a member of a class ...
- That's nested within an outer class ...
- That multiply inherits from 3 base classes ...
- With each base class from a different namespace ...
- Where one namespace is anonymous, ...
- The second namespace is inline, and ...
- The third namespace was named in a using-directive.

 23

23

Some of the other C++ features that complicate name lookup

- Qualified names
- Elaborated names
- Argument-dependent names
- Overloaded names
- try-blocks, catch clauses, and function-try-blocks
- friend declarations
- using declarations
- Template specializations (explicit and partial)
- Templates' dependent names
- virtual inheritance
- virtual functions
- enum class enumerators
- Operator overloading
- Lambda captures

 24

24

How are uses matched to declarations? [basic.lookup]

- In code, each name use triggers at least one of these name lookup algorithms:
  - Unqualified name lookup
  - Qualified name lookup
  - Argument-dependent lookup (ADL)
  - Elaborated type specifiers
  - Class member access
  - Using-directives and namespace aliases
- And sometimes triggers more than one of these!
- And sometimes with a context-specific variation.

 25

25

And then what? [basic.lookup]

- "Name lookup associates the use of a name with a set of declarations of that name":
  1. "The declarations found by name lookup shall either all declare the same entity or [else shall] form an overload set [of functions and/or function templates]."
  2. "Overload resolution takes place after name lookup has succeeded."
  3. If ① and ② succeed, then "access rules are considered."
- "Only after [①, ② (if applicable), and ③] have succeeded are the semantic properties introduced by the name's declaration ... used further in expression processing."

 26

26

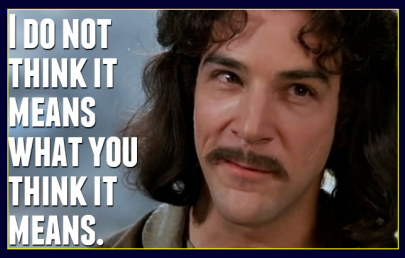
By the way, ...

- Compilers typically compare each newly-introduced name against previously-declared names:
  - To merge redeclarations, ...
  - To diagnose conflicting declarations, and ...
  - To recognize overload sets.
- This process is not triggered by a name's use:
  - Therefore it's technically not considered name lookup.
  - But it has many similarities.

 27

27

The ultimate name lookup diagnostic?



 28

28

Example [basic.lookup.unqual]

```
typedef int f;
namespace N {
  struct A {
    friend void f( A & );
    operator int( );
    void g( A a ) {
      int k = f( a );
    }
  };
}
```

- How to locate f's declaration?

- That f was declared by the typedef, not by the friend.
- The friend decl injects its f into the nearest enclosing namespace, namely N.
- Since there's no matching decl in N, unqualified name lookup won't find the friend and keeps looking outward, finding the typedef.
- (ADL would find the friend, but isn't triggered because f doesn't name a function.)

 29

29

Example [basic.lookup.unqual]

```
namespace A :: N {
  void g( );
}
void A :: N :: g( ) {
  ⋮
  k = 5;
  ⋮
}
```

- What name spaces are searched for k's decl, and in what order?

1. Outermost block scope of A::N::g, before the use of k.
2. Scope of namespace N.
3. Scope of namespace A.
4. Global scope, before the definition of A::N::g.

 30

30

Example [basic.lookup.unqual]

```
namespace M {
  class B { };
}
namespace N {
  class Y : public M::B {
    class X {
      int a[k];
    };
  };
}
```

- How to look up **k**'s decl?

1. Class N::Y::X's scope, before the use of **k**.
2. Class N::Y, before the definition of N::Y::X.
3. N::Y's base class M::B.
4. Namespace N, before the definition of N::Y.
5. Global scope, before the definition of N.

 31

31

Example [basic.lookup.unqual]

```
namespace N {
  int a = 4;
  extern int b;
}
int a = 2;
int N::b = a;
```

Where will UNL find **a**'s declaration: in namespace N, or in the global namespace?

 32

32

An odd/unusual/bizarre form of C++ name lookup

- Valid code?
  - `int S;`
    `struct S { ⋯ };`
  - Yes, S is multiply-declared in a single scope, but it is not overloaded because no functions are declared.
  - Nonetheless, it is valid C++ (because C allows this code).
- When using this (mis?)feature:
  - The variable's name hides the type's name from UNL.
  - But the type's name becomes visible when preceded by struct/class/union (*i.e.*, as an elaborated type specifier).
  - Please avoid such code whenever possible.

 33

33

Consider this example

```
class S { ⋯ };
S S;        // ok, but IMO perverse
S T;        // ill-formed decl of T: this S is not a type
class S U;  // ok, now S is a type, so U is a variable
```

- "[Elaborated type specifiers were] done for compatibility with C because of C's weird separate name space for tag names."

— G. Dos Reis, 2020-08-13

 34

34

Argument-dependent lookup (ADL) [basic.lookup.argdep]

- "When [a function is called via] an *unqualified-id*, other namespaces not considered [by UNL] may be searched, and … declarations not otherwise visible may be found."

```
namespace N {
  struct S { };
  void g( S );
}
⋮
void h( ) {
  N::S s;
  g( s );     ①
  ( g )( s ); ②
}
```

① g is an *unqualified-id*, so we also search the namespace, N, of argument s's type, S. There we find N::g, so we call it.

② (g) is an expression, not an *unqualified-id*, due to the parens, so no other namespaces are searched. UNL hadn't found a suitable function decl, so the call is ill-formed.

 35

35

Qualified name lookup and member access are similar

- Two lookups are needed, although:
  - QNL uses a :: (scope-resolution operator), whereas …
  - A class member access uses a . or –> operator.
- But a few commonalities:
  - Post-lookup, the left operand must denote or connote …
  - A context in which the right operand is then looked up.
  - (The global namespace is the implicit left operand for the unary scope-resolution operator.)
  - Neither of these can be followed by any ADL, because ADL depends on the result(s) of only UNL.

 36

36

A concluding recently-posted example

```
struct A { };
namespace N {
    struct A { ~A( ); };
};
struct C : N::A { };
struct B : A, C {
    ~B( );
    using  A = C;
};
B *bp;
int main( ) {
    bp –> A::~A( );
}
```

 37

37

Posted tentative analysis for further discussion

- The A in bp->A:: is looked up first in the scope of B; we only take an unqualified lookup result if there is no A in B. (Even in a dependent context.) So the 'A::' names C. (The fact that there's a ~ afterwards doesn't affect this lookup.)
- The A in ~A is looked up in the same fashion as the name prior to the ::~. So it's first looked up in the scope of B, and we only take an unqualified lookup result if there is no A in B. (Even in a dependent context.) So the '~A' also names C.
- So both names in the destructor-name do name the same type, as required by *[expr.prim.id.qual]p2*'s "Where *type-name* ::~ *type-name* is used, the two *type-name*s shall refer to the same type", and we have a valid destructor name for ::C. Calling that base class destructor also appears to be valid.

 38

38

"Through the Looking-Glass"

"Must a name mean something?"
Alice asked doubtfully.

"Of course it must,"
Humpty Dumpty said ... .

— **Lewis Carroll**, 1871

 39

39

What's in a Name? and What's a Name In?

FIN

WALTER E. BROWN, PH.D.

< webrown.cpp @ gmail.com >

40