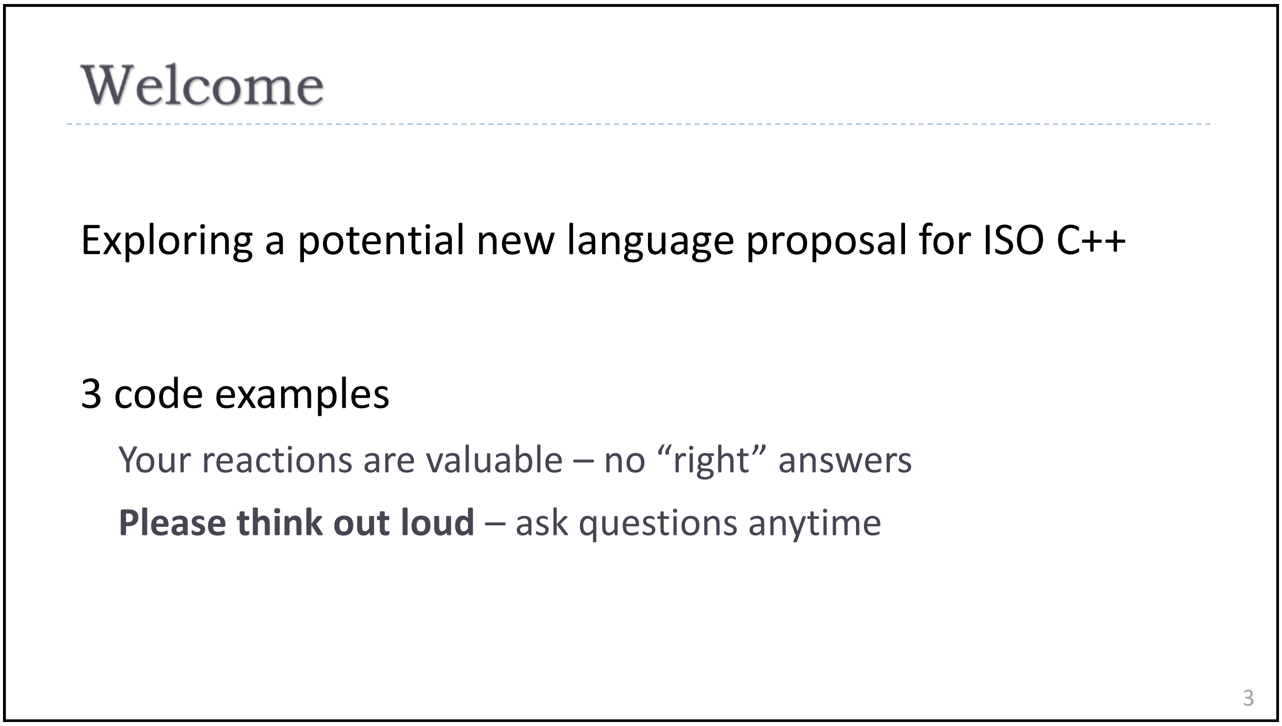
Metaclasses

Goal: Making C++ more powerful, and simpler

Herb Sutter

Prelude: An informal UX study

- Two volunteers please, willing to be on camera
  - < **5** years' experience
  - > **10** years' experience

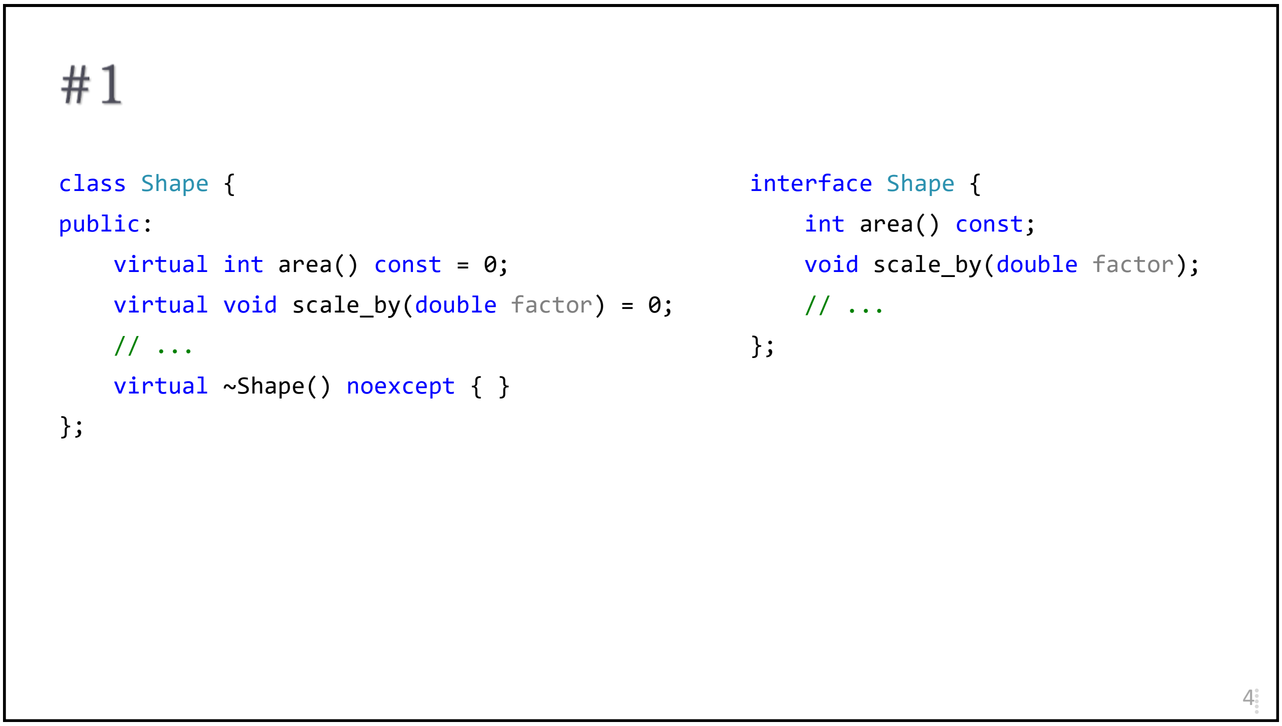


Welcome

Exploring a potential new language proposal for ISO C++

3 code examples

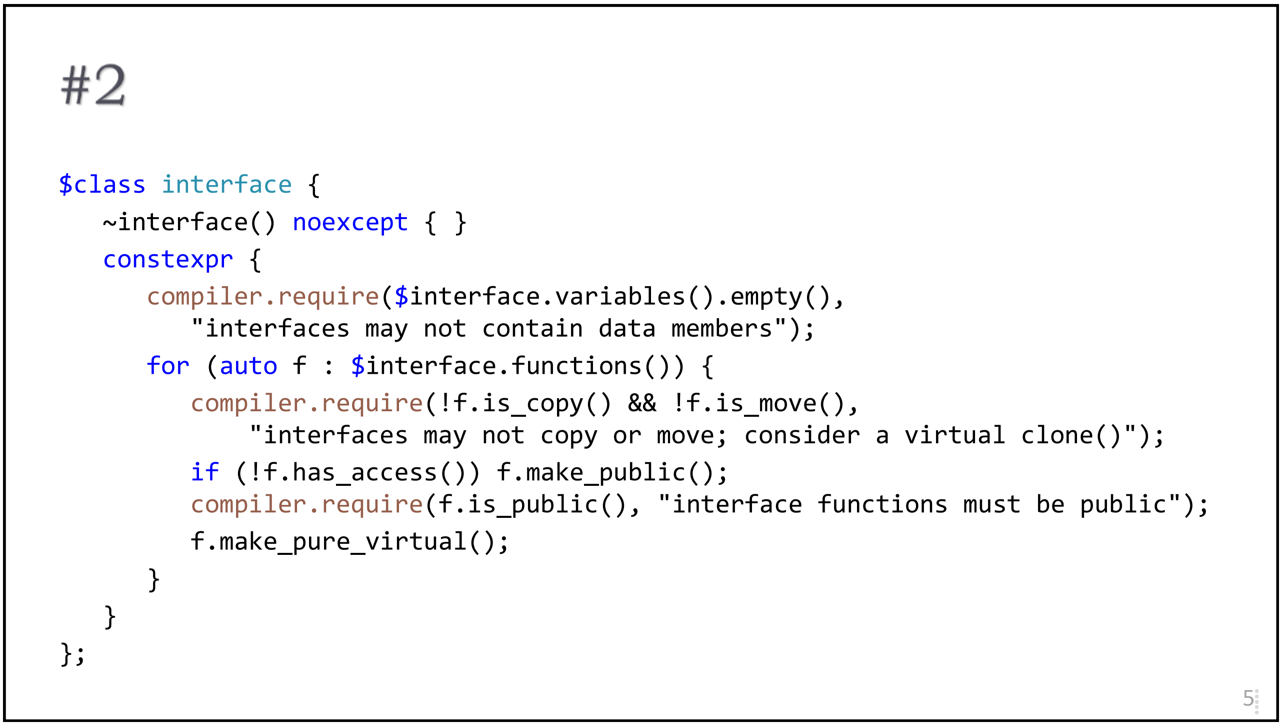
- Your reactions are valuable – no “right” answers
- **Please think out loud** – ask questions anytime



#1

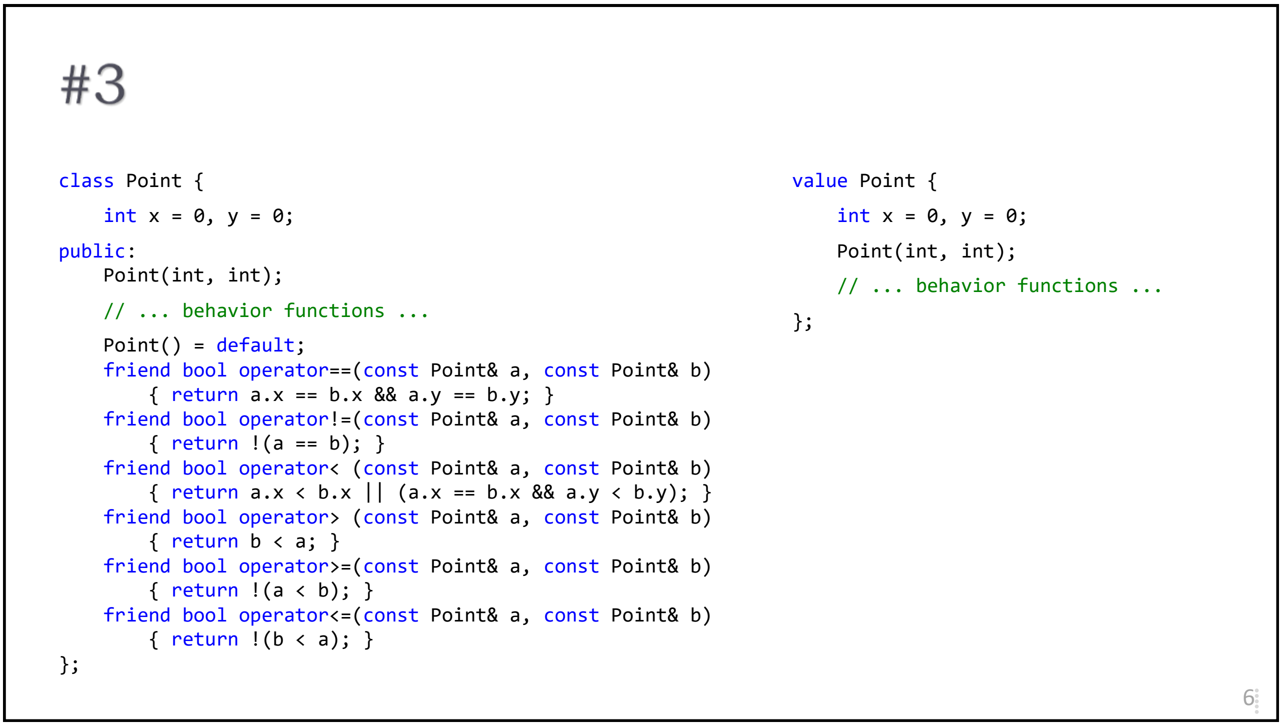
```
class Shape {
public:
    virtual int area() const = 0;
    virtual void scale_by(double factor) = 0;
    // ...
    virtual ~Shape() noexcept { }
};
```

```
interface Shape {
    int area() const;
    void scale_by(double factor);
    // ...
};
```



#2

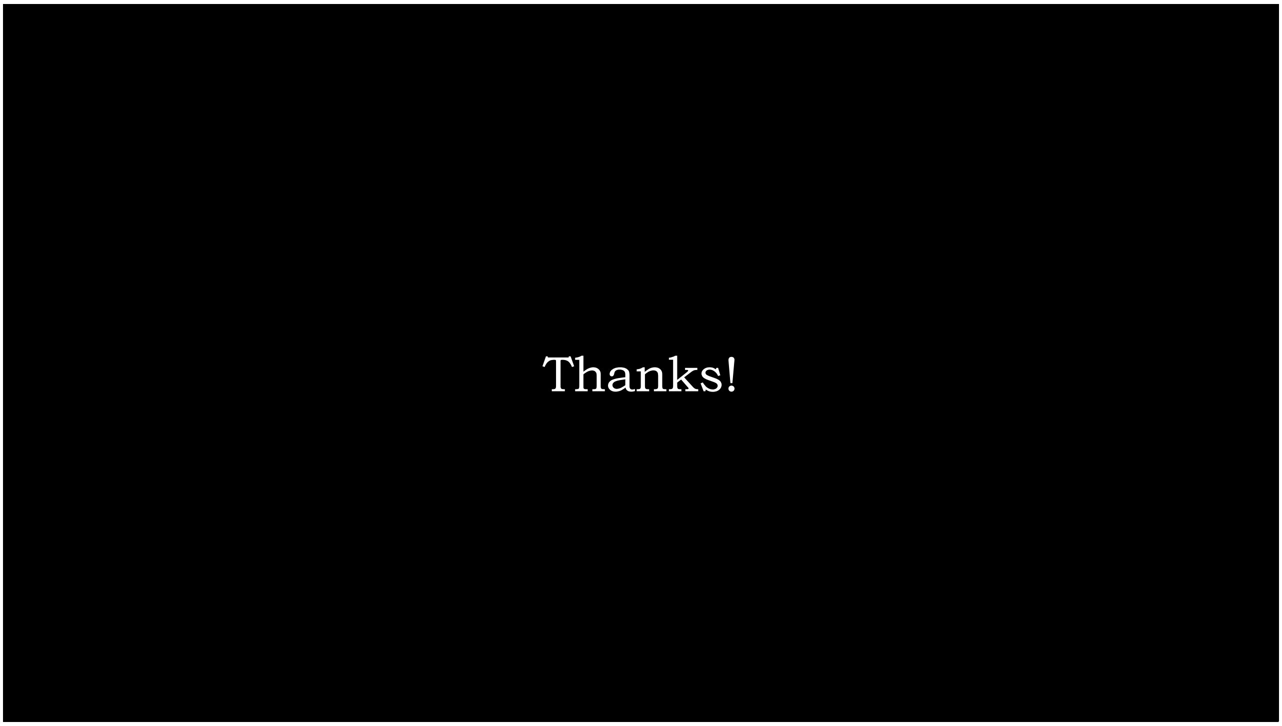
```
$class interface {
   ~interface() noexcept { }
   constexpr {
      compiler.require($interface.variables().empty(),
         "interfaces may not contain data members");
      for (auto f : $interface.functions()) {
         compiler.require(!f.is_copy() && !f.is_move(),
             "interfaces may not copy or move; consider a virtual clone()");
         if (!f.has_access()) f.make_public();
         compiler.require(f.is_public(), "interface functions must be public");
         f.make_pure_virtual();
      }
   }
};
```



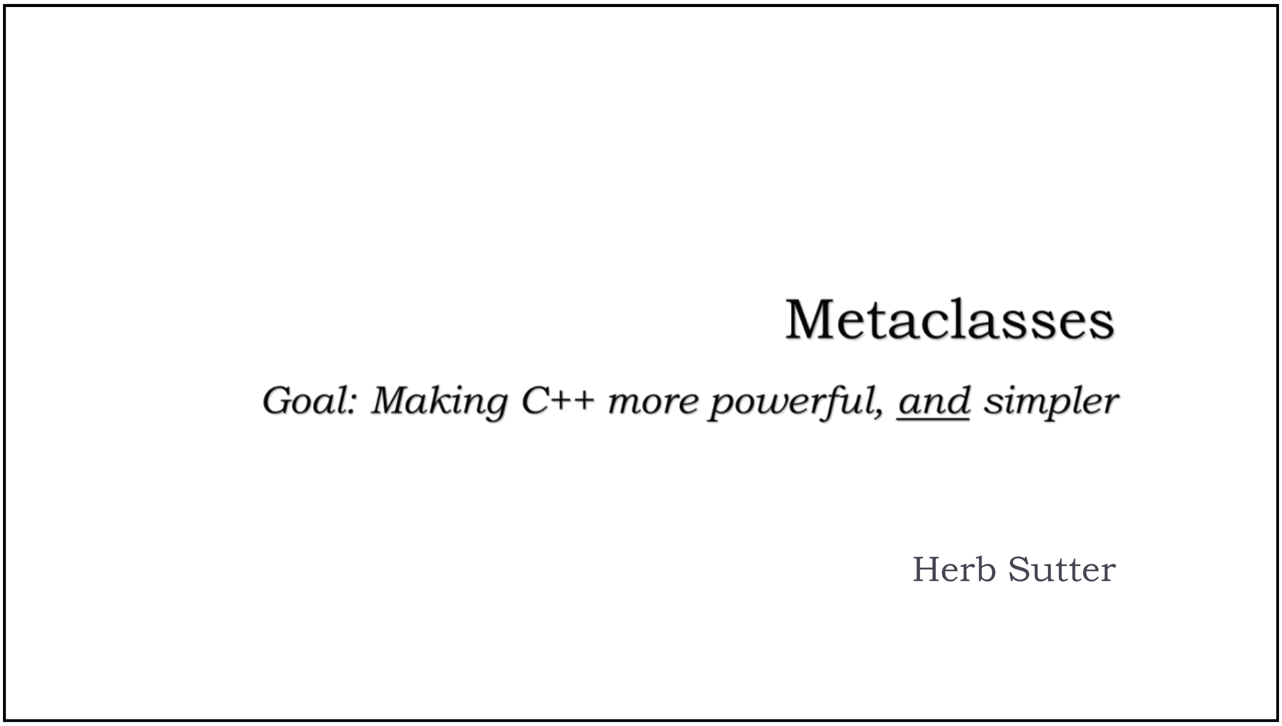
#3

```
class Point {
    int x = 0, y = 0;
public:
    Point(int, int);
    // ... behavior functions ...
    Point() = default;
    friend bool operator==(const Point& a, const Point& b)
        { return a.x == b.x && a.y == b.y; }
    friend bool operator!=(const Point& a, const Point& b)
        { return !(a == b); }
    friend bool operator< (const Point& a, const Point& b)
        { return a.x < b.x || (a.x == b.x && a.y < b.y); }
    friend bool operator> (const Point& a, const Point& b)
        { return b < a; }
    friend bool operator>=(const Point& a, const Point& b)
        { return !(a < b); }
    friend bool operator<=(const Point& a, const Point& b)
        { return !(b < a); }
};
```

```
value Point {
    int x = 0, y = 0;
    Point(int, int);
    // ... behavior functions ...
};
```



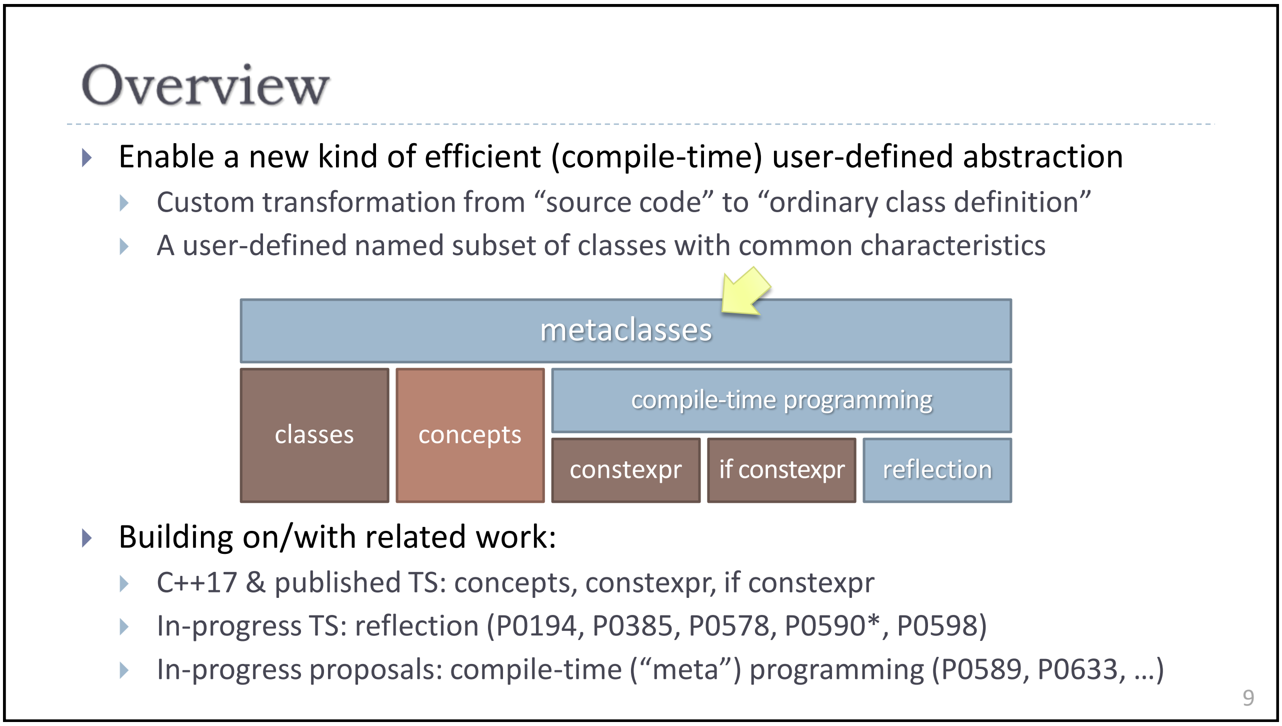
Thanks!



Metaclasses

Goal: Making C++ more powerful, and simpler

Herb Sutter



Overview

- Enable a new kind of efficient (compile-time) user-defined abstraction
  - Custom transformation from “source code” to “ordinary class definition”
  - A user-defined named subset of classes with common characteristics

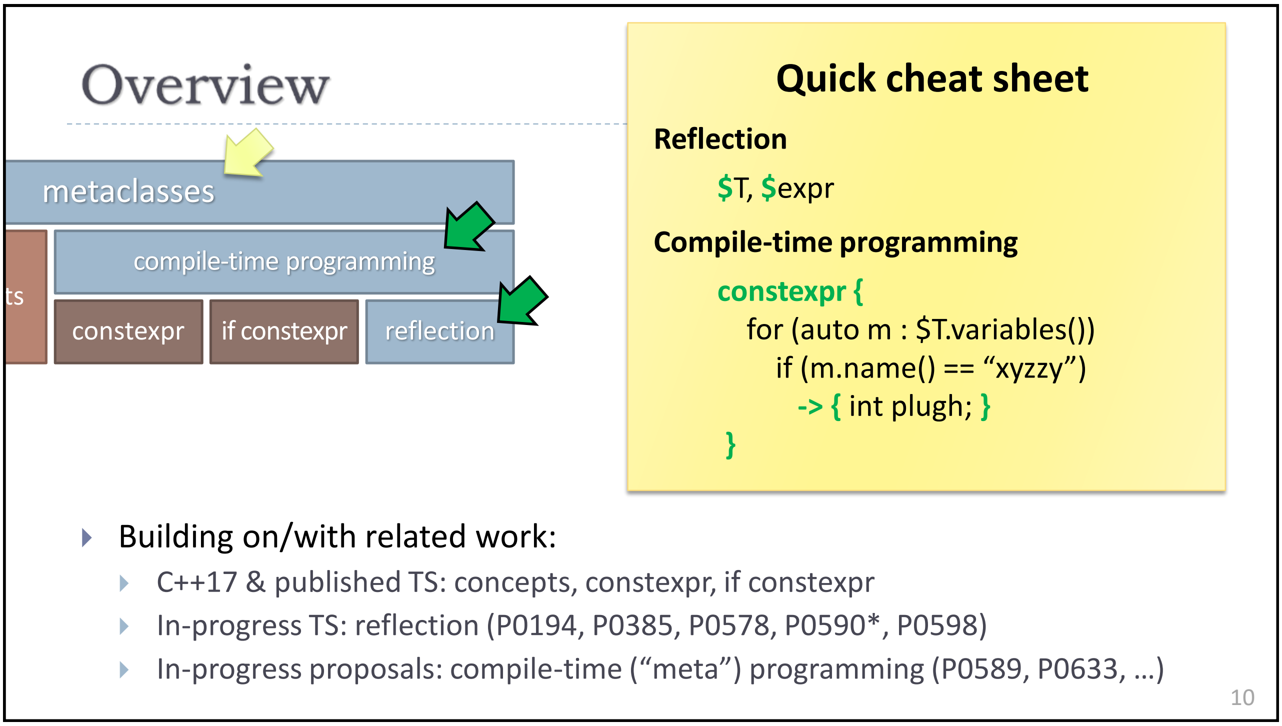
metaclasses

classes | concepts | compile-time programming

constexpr | if constexpr | reflection

- Building on/with related work:
  - C++17 & published TS: concepts, constexpr, if constexpr
  - In-progress TS: reflection (P0194, P0385, P0578, P0590*, P0598)
  - In-progress proposals: compile-time (“meta”) programming (P0589, P0633, …)

9



Overview

metaclasses

ts | compile-time programming

constexpr | if constexpr | reflection

Quick cheat sheet

Reflection

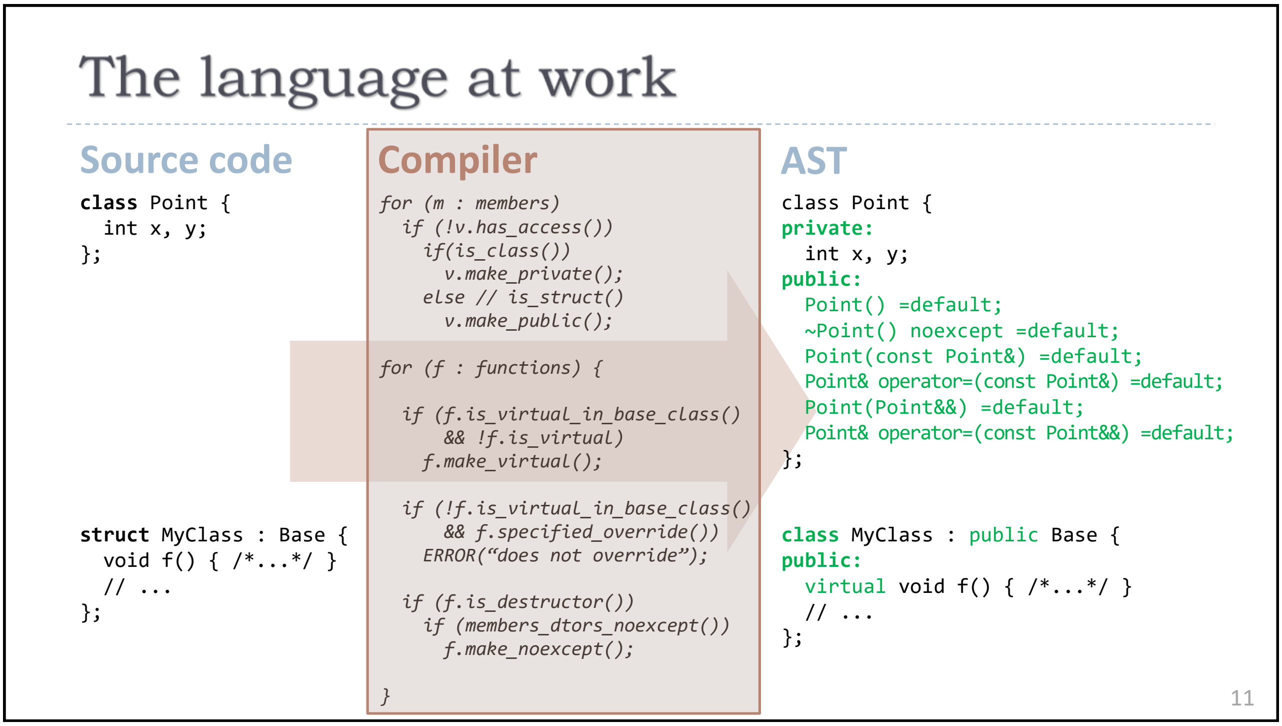
$T, $expr

Compile-time programming

```
constexpr {
    for (auto m : $T.variables())
        if (m.name() == “xyzzy”)
            -> { int plugh; }
}
```

- Building on/with related work:
  - C++17 & published TS: concepts, constexpr, if constexpr
  - In-progress TS: reflection (P0194, P0385, P0578, P0590*, P0598)
  - In-progress proposals: compile-time (“meta”) programming (P0589, P0633, …)

10



The language at work

Source code

```
class Point {
  int x, y;
};
```

```
struct MyClass : Base {
  void f() { /*...*/ }
  // ...
};
```

Compiler

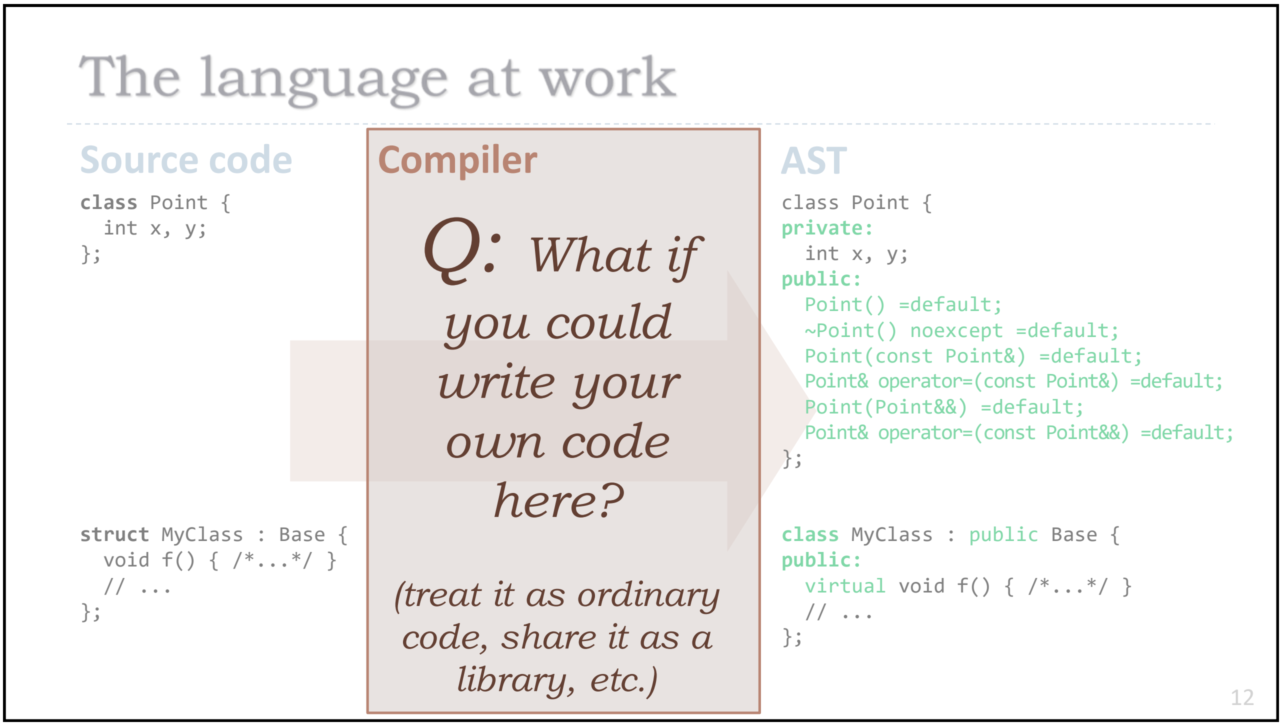
```
for (m : members)
  if (!v.has_access())
    if(is_class())
      v.make_private();
    else // is_struct()
      v.make_public();

for (f : functions) {

  if (f.is_virtual_in_base_class()
      && !f.is_virtual)
    f.make_virtual();

  if (!f.is_virtual_in_base_class()
      && f.specified_override())
    ERROR("does not override");

  if (f.is_destructor())
    if (members_dtors_noexcept())
      f.make_noexcept();

}
```

AST

```
class Point {
private:
  int x, y;
public:
  Point() =default;
  ~Point() noexcept =default;
  Point(const Point&) =default;
  Point& operator=(const Point&) =default;
  Point(Point&&) =default;
  Point& operator=(const Point&&) =default;
};
```

```
class MyClass : public Base {
public:
  virtual void f() { /*...*/ }
  // ...
};
```

11



The language at work

Source code

```
class Point {
  int x, y;
};
```

```
struct MyClass : Base {
  void f() { /*...*/ }
  // ...
};
```

Compiler

Q: What if you could write your own code here?

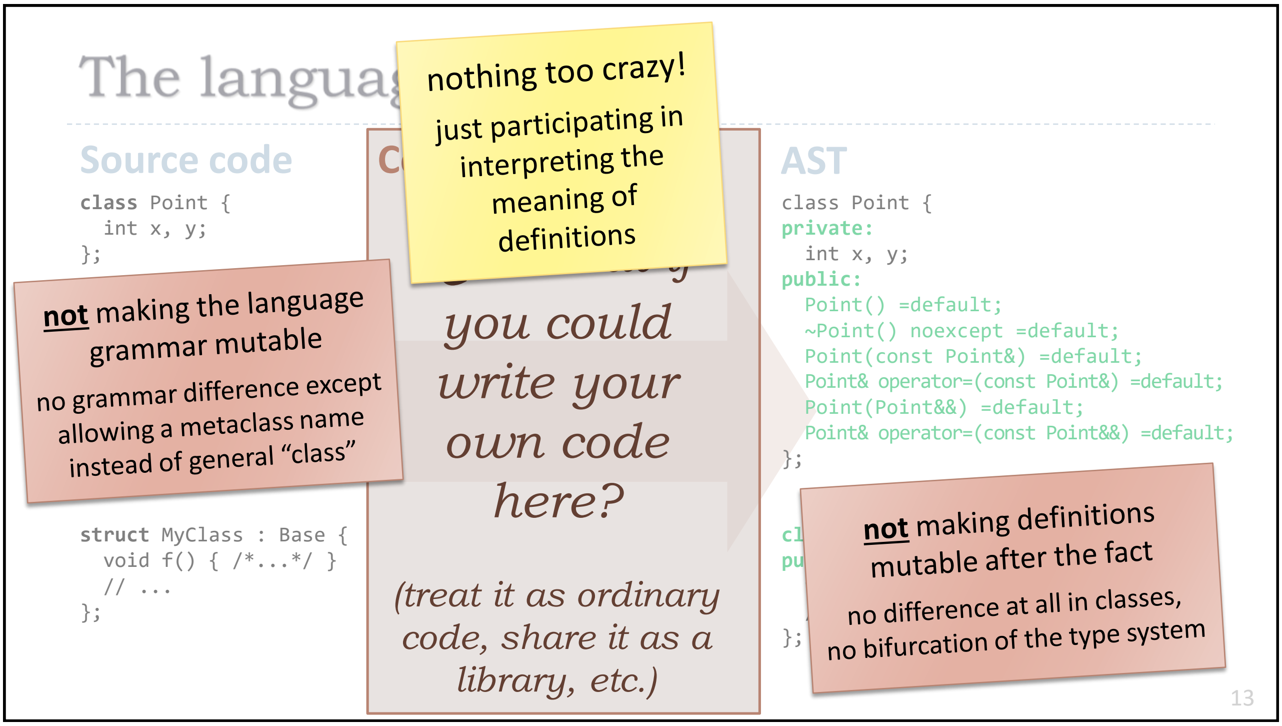
(treat it as ordinary code, share it as a library, etc.)

AST

```
class Point {
private:
  int x, y;
public:
  Point() =default;
  ~Point() noexcept =default;
  Point(const Point&) =default;
  Point& operator=(const Point&) =default;
  Point(Point&&) =default;
  Point& operator=(const Point&&) =default;
};
```

```
class MyClass : public Base {
public:
  virtual void f() { /*...*/ }
  // ...
};
```

12



The languag

nothing too crazy!

just participating in interpreting the meaning of definitions

Source code

```
class Point {
  int x, y;
};
```

not making the language grammar mutable

no grammar difference except allowing a metaclass name instead of general "class"

```
struct MyClass : Base {
  void f() { /*...*/ }
  // ...
};
```

C

you could write your own code here?

(treat it as ordinary code, share it as a library, etc.)

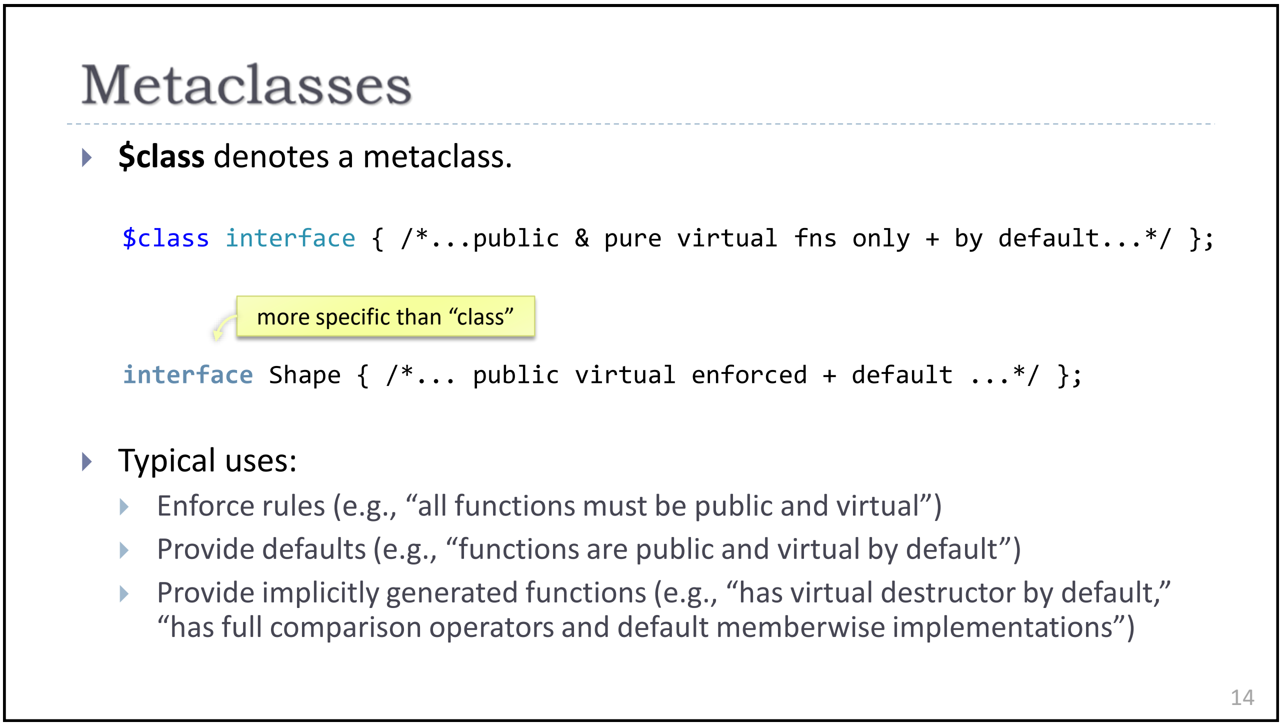
AST

```
class Point {
private:
  int x, y;
public:
  Point() =default;
  ~Point() noexcept =default;
  Point(const Point&) =default;
  Point& operator=(const Point&) =default;
  Point(Point&&) =default;
  Point& operator=(const Point&&) =default;
};

cl
pu

};
```

not making definitions mutable after the fact

no difference at all in classes, no bifurcation of the type system

13



Metaclasses

- **$class** denotes a metaclass.

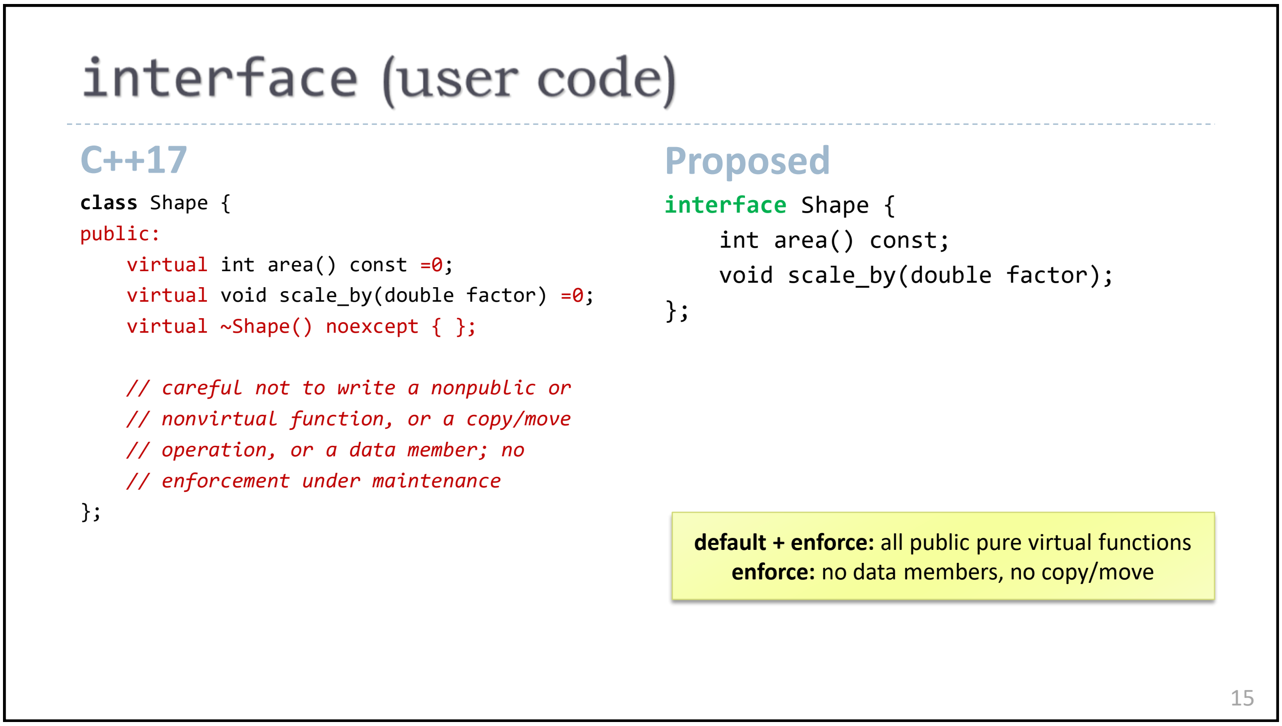
```
$class interface { /*...public & pure virtual fns only + by default...*/ };
```

more specific than "class"

```
interface Shape { /*... public virtual enforced + default ...*/ };
```

- Typical uses:
  - Enforce rules (e.g., "all functions must be public and virtual")
  - Provide defaults (e.g., "functions are public and virtual by default")
  - Provide implicitly generated functions (e.g., "has virtual destructor by default," "has full comparison operators and default memberwise implementations")

14



interface (user code)

C++17

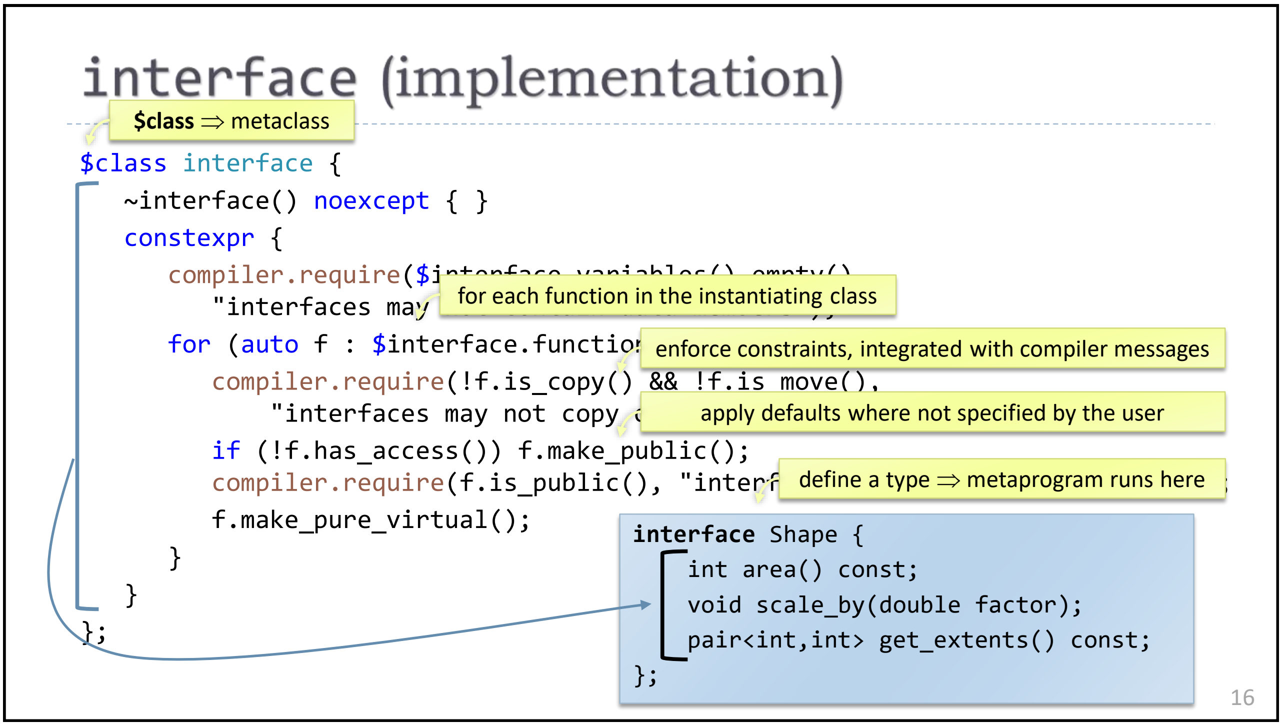
```
class Shape {
public:
    virtual int area() const =0;
    virtual void scale_by(double factor) =0;
    virtual ~Shape() noexcept { };

    // careful not to write a nonpublic or
    // nonvirtual function, or a copy/move
    // operation, or a data member; no
    // enforcement under maintenance
};
```

Proposed

```
interface Shape {
    int area() const;
    void scale_by(double factor);
};
```

default + enforce: all public pure virtual functions
enforce: no data members, no copy/move

15



interface (implementation)

$class ⇒ metaclass

```
$class interface {
    ~interface() noexcept { }
    constexpr {
        compiler.require($interface.variables().empty()
            "interfaces may
        for (auto f : $interface.functio
            compiler.require(!f.is_copy() && !f.is move(),
                "interfaces may not copy
            if (!f.has_access()) f.make_public();
            compiler.require(f.is_public(), "interf
            f.make_pure_virtual();
        }
    }
};
```

for each function in the instantiating class

enforce constraints, integrated with compiler messages

apply defaults where not specified by the user

define a type ⇒ metaprogram runs here

```
interface Shape {
    int area() const;
    void scale_by(double factor);
    pair<int,int> get_extents() const;
};
```

16

interface (implementation)

```
$class interface {
   ~interface() noexcept { }
   constexpr {
      compiler.require($interface.variables().empty(),
         "interfaces may not contain data members");
                                  ctions()) {
                                  py() && !f.is_move(),
                                  opy or move; consider a
                                  ke_public();
                                  lic(), "interface funct
```

Look ma, no standardese!

Define language-like features using the language itself – can read the source code to “language features” like we can read the source code to STL and other libs

Bonus: Does my spec have a bug? Unit-test and debug it as usual… it’s just code

We do not have unit testing and debugging for “standardese”

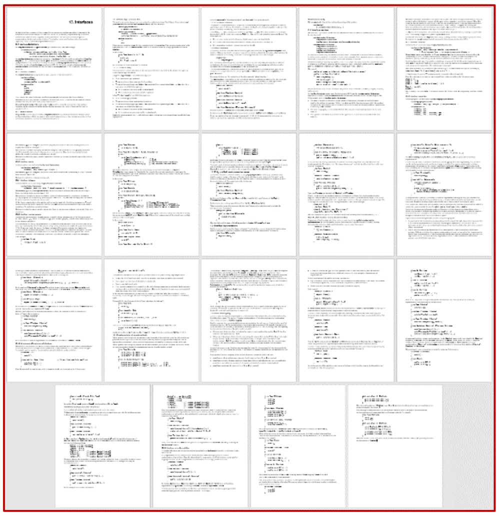
```
interface Shape {
    int area() const;
    void scale_by(double factor);
    pair<int,int> get_extents() const;
};
```

+ no loss in usability, expressiveness, diagnostics, performance, …

even compared to other languages that added this as a built-in language feature

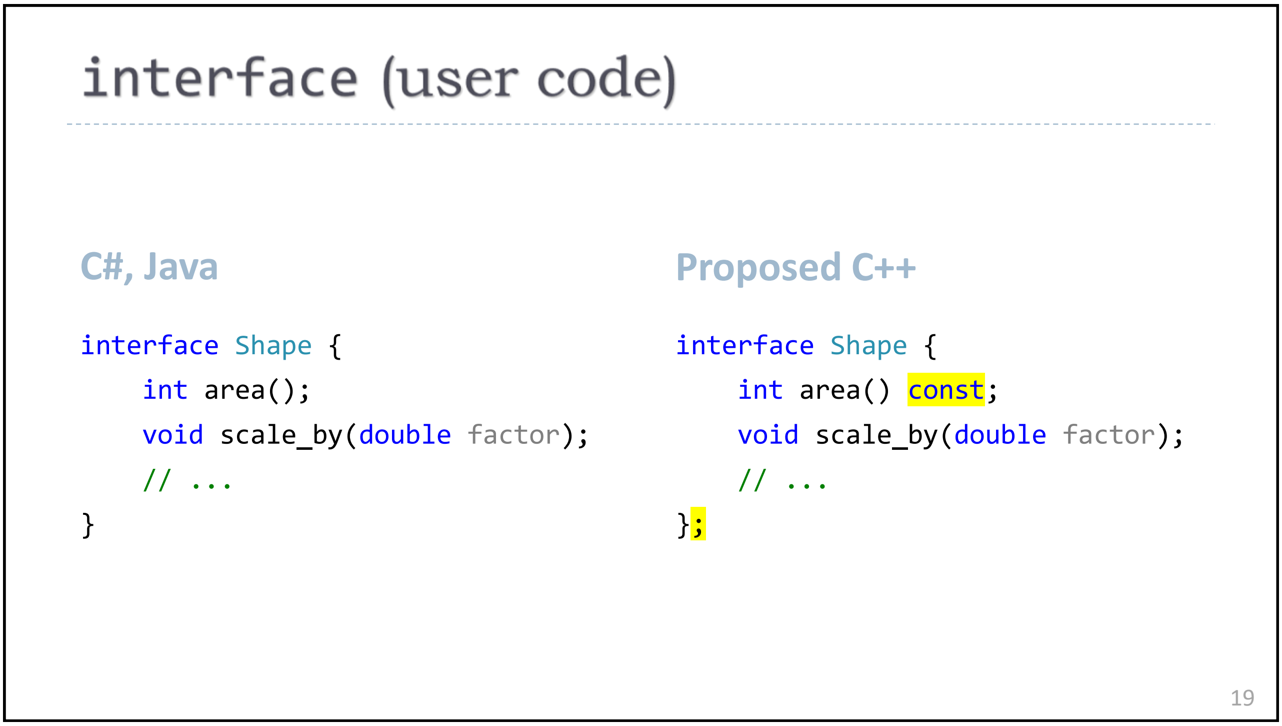
interface (implementation)

C# language: ~18pg, English



Proposed C++: ~10 lines, testable code

```
$class interface {
   ~interface() noexcept { }
   constexpr {
      compiler.require($interface.variables().empty(),
         "interfaces may not contain data members");
      for (auto f : $interface.functions()) {
         compiler.require(!f.is_copy() && !f.is_move(),
             "interfaces may not copy or move; "
             "consider a virtual clone()");
         if (!f.has_access()) f.make_public();
         compiler.require(f.is_public(),
             "interface functions must be public");
         f.make_pure_virtual();
      }
   }
};
```



interface (user code)

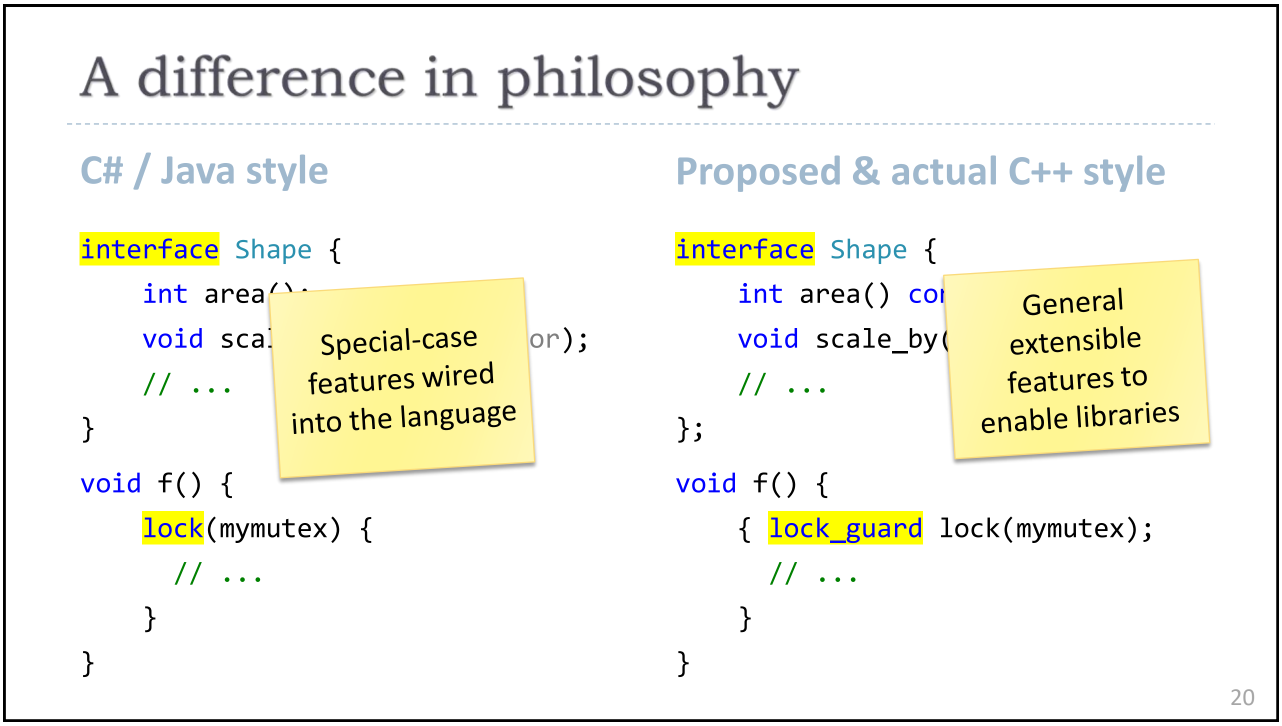
C#, Java

```
interface Shape {
    int area();
    void scale_by(double factor);
    // ...
}
```

Proposed C++

```
interface Shape {
    int area() const;
    void scale_by(double factor);
    // ...
};
```

19



A difference in philosophy

C# / Java style

```
interface Shape {
    int area(
    void sca             or);
    // ...
}
void f() {
    lock(mymutex) {
      // ...
    }
}
```

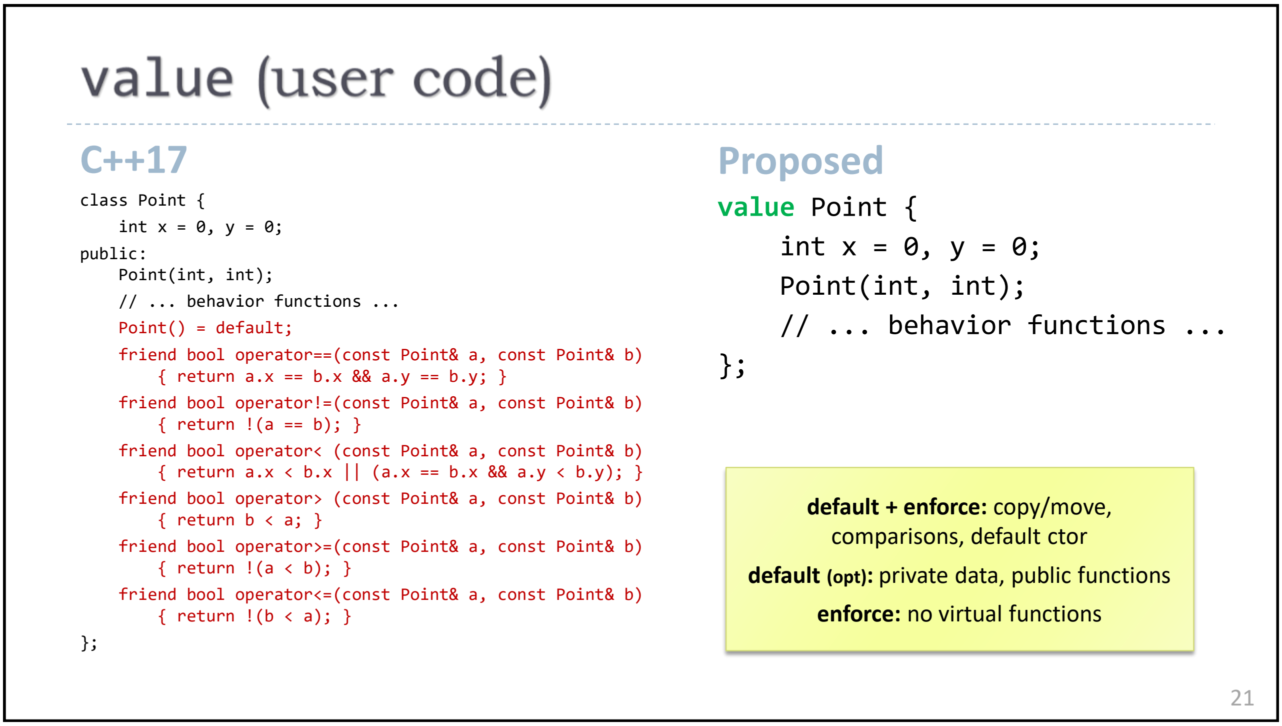
Special-case features wired into the language

Proposed & actual C++ style

```
interface Shape {
    int area() co
    void scale_by(
    // ...
};
void f() {
    { lock_guard lock(mymutex);
      // ...
    }
}
```

General extensible features to enable libraries

20



value (user code)

C++17

```
class Point {
    int x = 0, y = 0;
public:
    Point(int, int);
    // ... behavior functions ...
    Point() = default;
    friend bool operator==(const Point& a, const Point& b)
        { return a.x == b.x && a.y == b.y; }
    friend bool operator!=(const Point& a, const Point& b)
        { return !(a == b); }
    friend bool operator< (const Point& a, const Point& b)
        { return a.x < b.x || (a.x == b.x && a.y < b.y); }
    friend bool operator> (const Point& a, const Point& b)
        { return b < a; }
    friend bool operator>=(const Point& a, const Point& b)
        { return !(a < b); }
    friend bool operator<=(const Point& a, const Point& b)
        { return !(b < a); }
};
```

Proposed

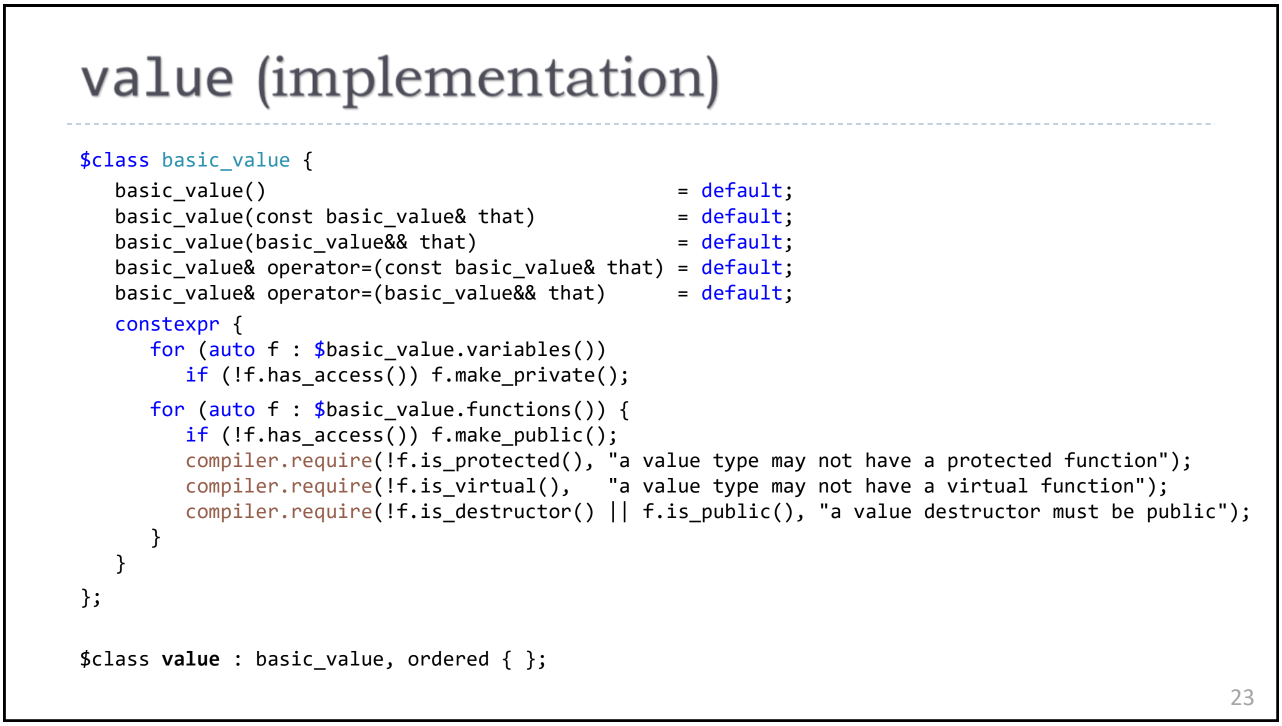
```
value Point {
    int x = 0, y = 0;
    Point(int, int);
    // ... behavior functions ...
};
```

default + enforce: copy/move, comparisons, default ctor

default (opt): private data, public functions

enforce: no virtual functions

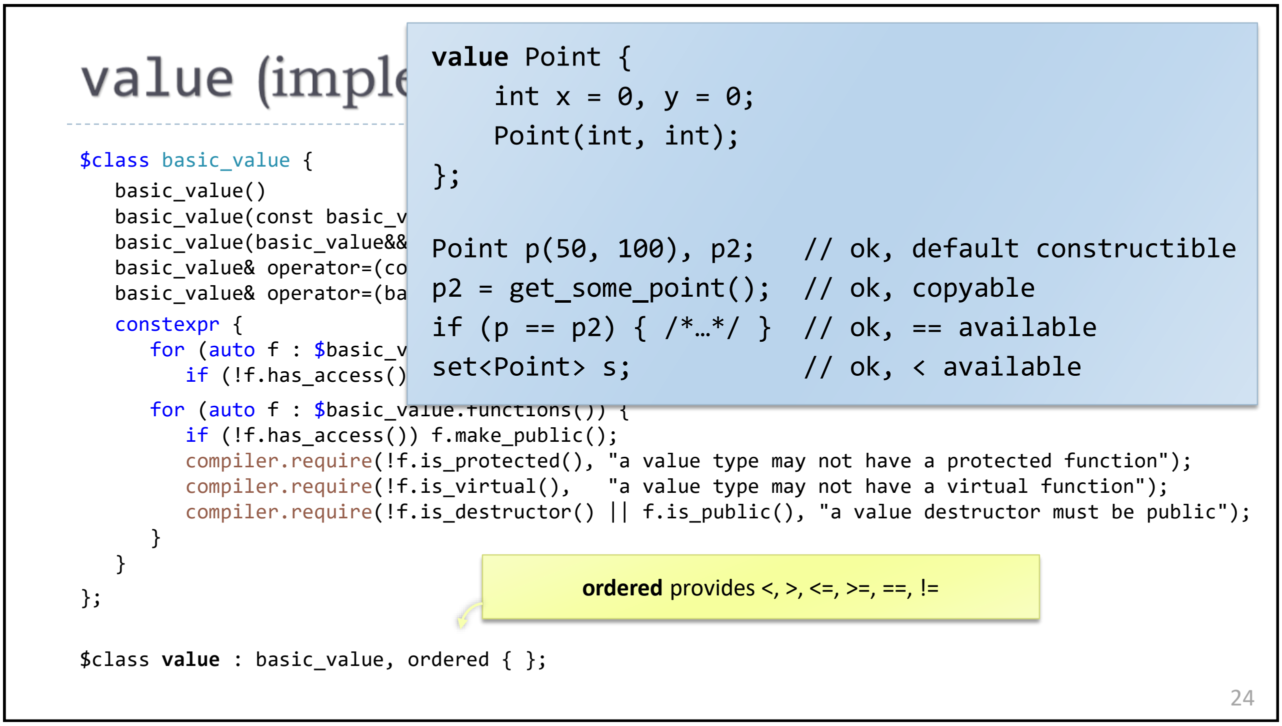
21



value (implementation)

```
$class basic_value {
   basic_value()                                   = default;
   basic_value(const basic_value& that)            = default;
   basic_value(basic_value&& that)                 = default;
   basic_value& operator=(const basic_value& that) = default;
   basic_value& operator=(basic_value&& that)      = default;
   constexpr {
      for (auto f : $basic_value.variables())
         if (!f.has_access()) f.make_private();
      for (auto f : $basic_value.functions()) {
         if (!f.has_access()) f.make_public();
         compiler.require(!f.is_protected(), "a value type may not have a protected function");
         compiler.require(!f.is_virtual(),   "a value type may not have a virtual function");
         compiler.require(!f.is_destructor() || f.is_public(), "a value destructor must be public");
      }
   }
};

$class value : basic_value, ordered { };
```

23



value (imple

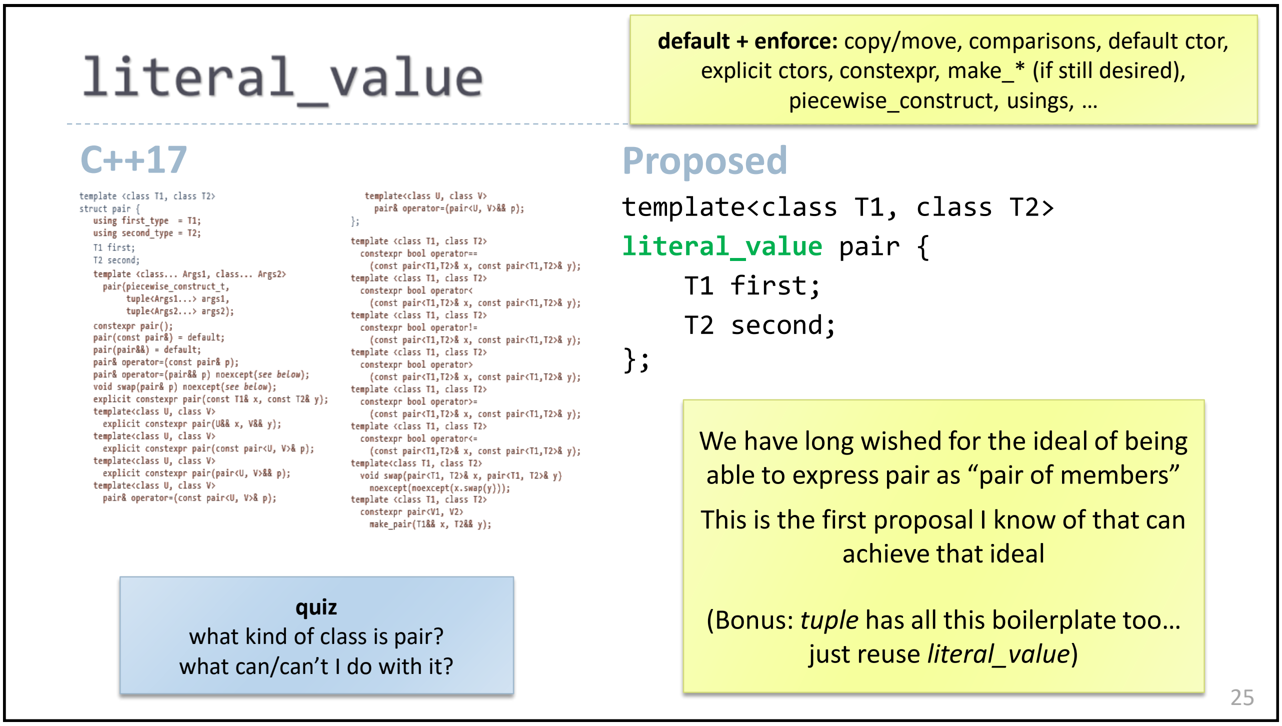
```
$class basic_value {
   basic_value()
   basic_value(const basic_v
   basic_value(basic_value&&
   basic_value& operator=(co
   basic_value& operator=(ba
   constexpr {
      for (auto f : $basic_v
         if (!f.has_access()
      for (auto f : $basic_value.functions()) {
         if (!f.has_access()) f.make_public();
         compiler.require(!f.is_protected(), "a value type may not have a protected function");
         compiler.require(!f.is_virtual(),   "a value type may not have a virtual function");
         compiler.require(!f.is_destructor() || f.is_public(), "a value destructor must be public");
      }
   }
};
```

```
value Point {
    int x = 0, y = 0;
    Point(int, int);
};

Point p(50, 100), p2;   // ok, default constructible
p2 = get_some_point();  // ok, copyable
if (p == p2) { /*…*/ }  // ok, == available
set<Point> s;           // ok, < available
```

ordered provides <, >, <=, >=, ==, !=

```
$class value : basic_value, ordered { };
```

24



literal_value

default + enforce: copy/move, comparisons, default ctor, explicit ctors, constexpr, make_* (if still desired), piecewise_construct, usings, …

C++17

```
template <class T1, class T2>
struct pair {
   using first_type  = T1;
   using second_type = T2;
   T1 first;
   T2 second;
   template <class... Args1, class... Args2>
     pair(piecewise_construct_t,
          tuple<Args1...> args1,
          tuple<Args2...> args2);
   constexpr pair();
   pair(const pair&) = default;
   pair(pair&&) = default;
   pair& operator=(const pair& p);
   pair& operator=(pair&& p) noexcept(see below);
   void swap(pair& p) noexcept(see below);
   explicit constexpr pair(const T1& x, const T2& y);
   template<class U, class V>
     explicit constexpr pair(U&& x, V&& y);
   template<class U, class V>
     explicit constexpr pair(const pair<U, V>& p);
   template<class U, class V>
     explicit constexpr pair(pair<U, V>&& p);
   template<class U, class V>
     pair& operator=(const pair<U, V>& p);
   template<class U, class V>
     pair& operator=(pair<U, V>&& p);
};

template <class T1, class T2>
  constexpr bool operator==
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template <class T1, class T2>
  constexpr bool operator<
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template <class T1, class T2>
  constexpr bool operator!=
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template <class T1, class T2>
  constexpr bool operator>
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template <class T1, class T2>
  constexpr bool operator>=
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template <class T1, class T2>
  constexpr bool operator<=
    (const pair<T1,T2>& x, const pair<T1,T2>& y);
template<class T1, class T2>
  void swap(pair<T1, T2>& x, pair<T1, T2>& y)
    noexcept(noexcept(x.swap(y)));
template <class T1, class T2>
  constexpr pair<V1, V2>
    make_pair(T1&& x, T2&& y);
```

quiz
what kind of class is pair?
what can/can't I do with it?

Proposed

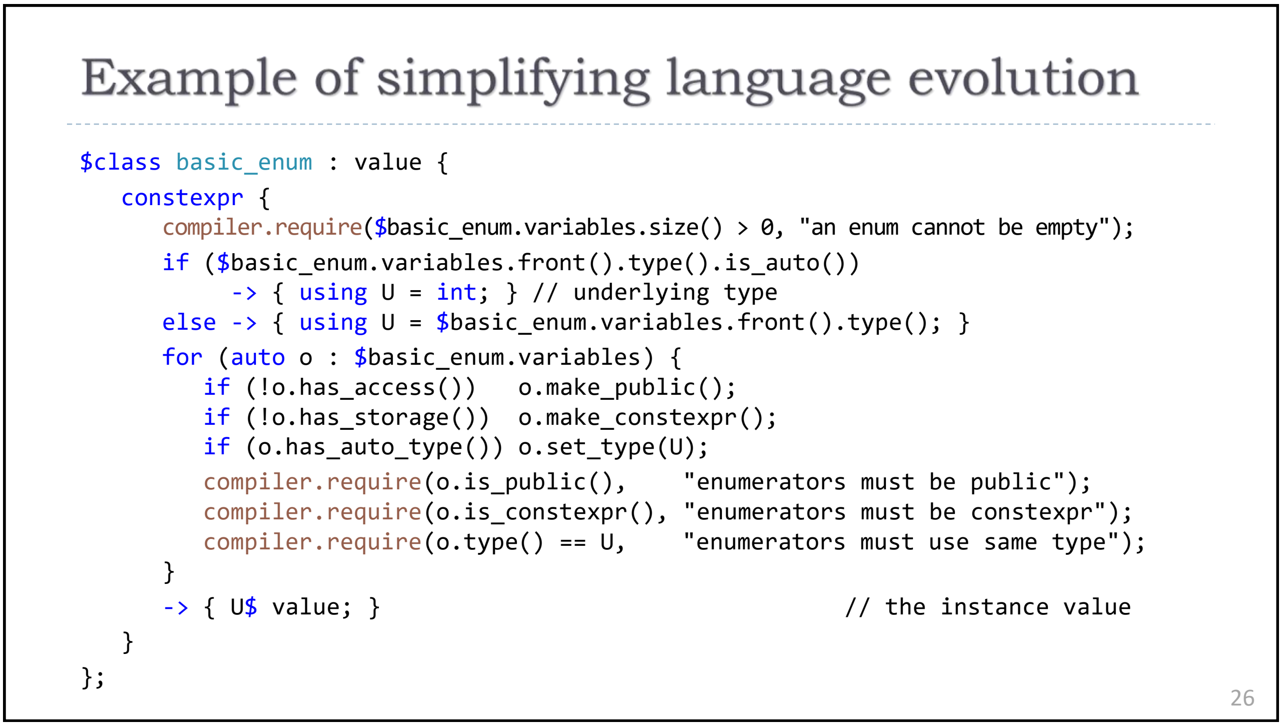
```
template<class T1, class T2>
literal_value pair {
    T1 first;
    T2 second;
};
```

We have long wished for the ideal of being able to express pair as "pair of members"

This is the first proposal I know of that can achieve that ideal

(Bonus: *tuple* has all this boilerplate too… just reuse *literal_value*)

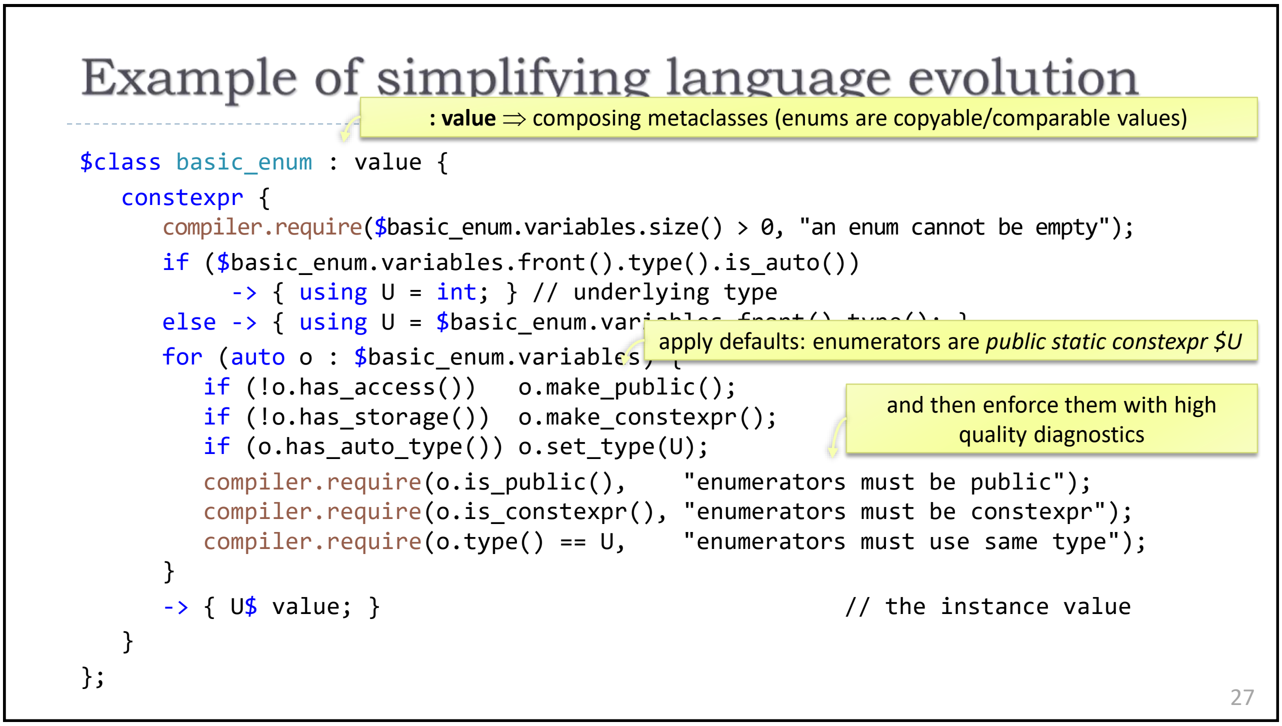
25



Example of simplifying language evolution

```
$class basic_enum : value {
   constexpr {
      compiler.require($basic_enum.variables.size() > 0, "an enum cannot be empty");
      if ($basic_enum.variables.front().type().is_auto())
           -> { using U = int; } // underlying type
      else -> { using U = $basic_enum.variables.front().type(); }
      for (auto o : $basic_enum.variables) {
         if (!o.has_access())   o.make_public();
         if (!o.has_storage())  o.make_constexpr();
         if (o.has_auto_type()) o.set_type(U);
         compiler.require(o.is_public(),    "enumerators must be public");
         compiler.require(o.is_constexpr(), "enumerators must be constexpr");
         compiler.require(o.type() == U,    "enumerators must use same type");
      }
      -> { U$ value; }                                  // the instance value
   }
};
```

26



Example of simplifying language evolution

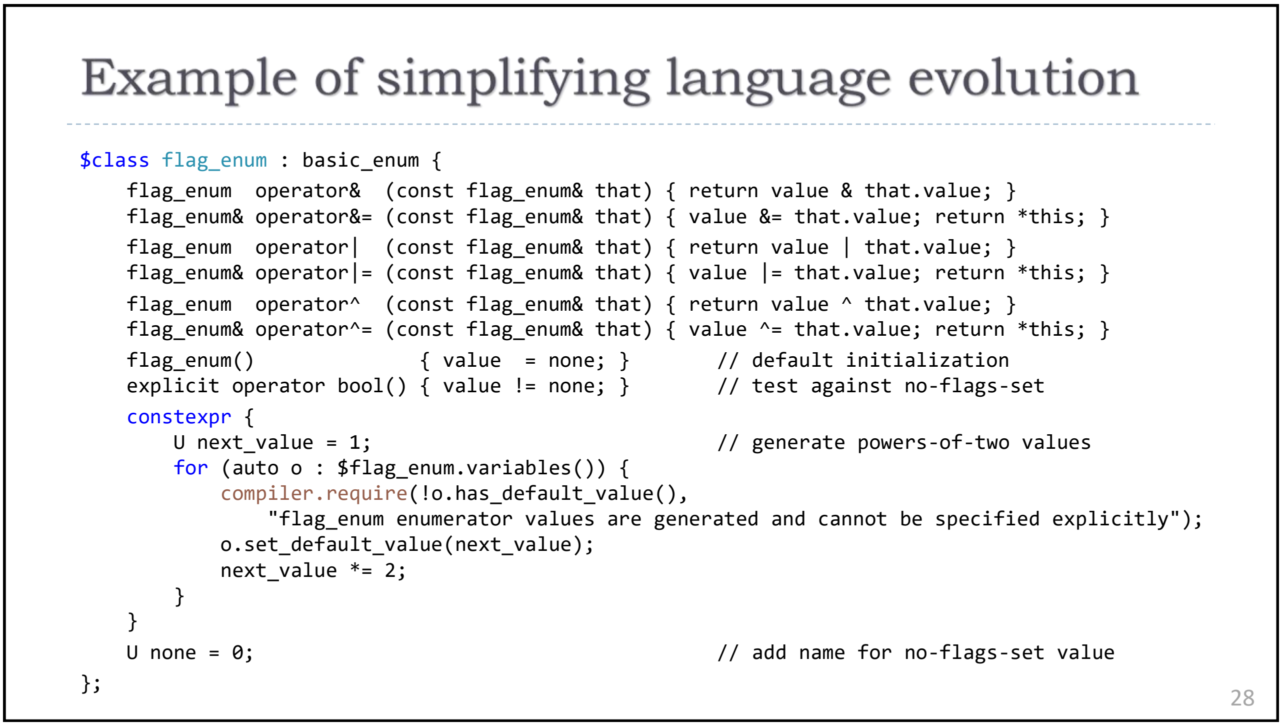
: value ⇒ composing metaclasses (enums are copyable/comparable values)

```
$class basic_enum : value {
   constexpr {
      compiler.require($basic_enum.variables.size() > 0, "an enum cannot be empty");
      if ($basic_enum.variables.front().type().is_auto())
           -> { using U = int; } // underlying type
      else -> { using U = $basic_enum.variables.front().type(); }
      for (auto o : $basic_enum.variables) {
         if (!o.has_access())   o.make_public();
         if (!o.has_storage())  o.make_constexpr();
         if (o.has_auto_type()) o.set_type(U);
         compiler.require(o.is_public(),    "enumerators must be public");
         compiler.require(o.is_constexpr(), "enumerators must be constexpr");
         compiler.require(o.type() == U,    "enumerators must use same type");
      }
      -> { U$ value; }                                  // the instance value
   }
};
```

apply defaults: enumerators are *public static constexpr $U*

and then enforce them with high quality diagnostics

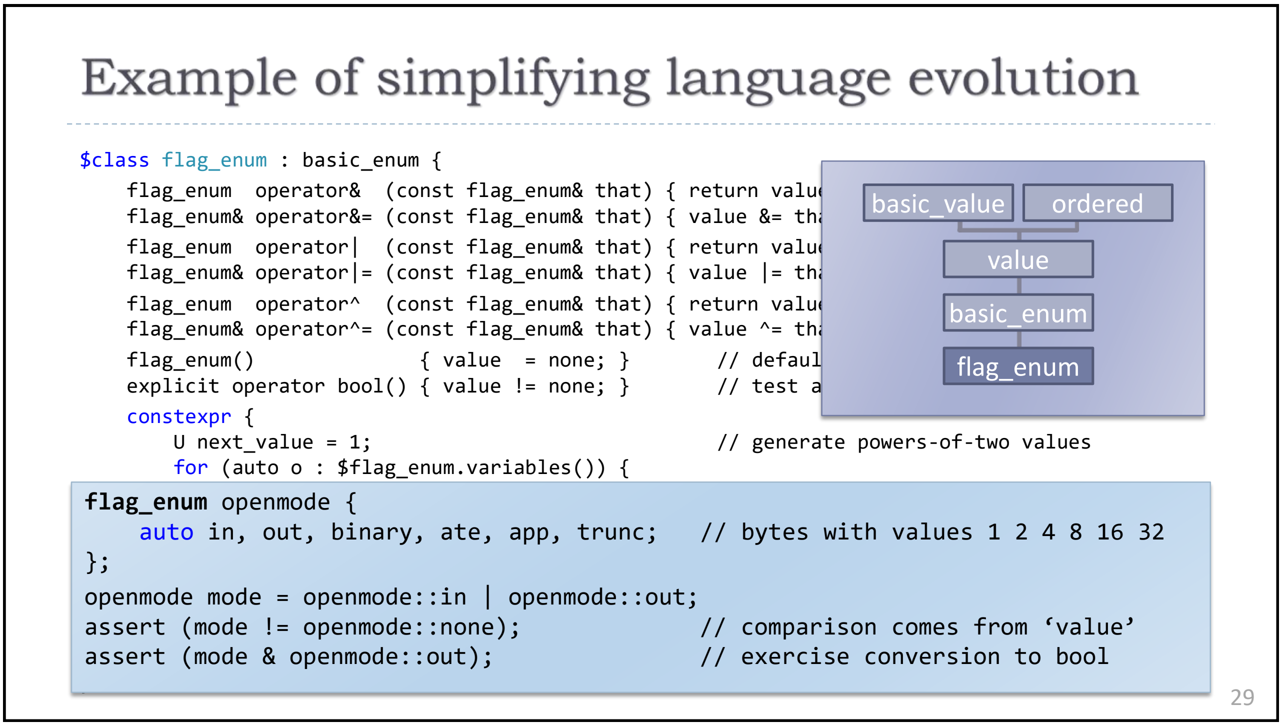
27



Example of simplifying language evolution

```
$class flag_enum : basic_enum {
    flag_enum  operator&  (const flag_enum& that) { return value & that.value; }
    flag_enum& operator&= (const flag_enum& that) { value &= that.value; return *this; }
    flag_enum  operator|  (const flag_enum& that) { return value | that.value; }
    flag_enum& operator|= (const flag_enum& that) { value |= that.value; return *this; }
    flag_enum  operator^  (const flag_enum& that) { return value ^ that.value; }
    flag_enum& operator^= (const flag_enum& that) { value ^= that.value; return *this; }
    flag_enum()              { value  = none; }       // default initialization
    explicit operator bool() { value != none; }       // test against no-flags-set
    constexpr {
        U next_value = 1;                             // generate powers-of-two values
        for (auto o : $flag_enum.variables()) {
            compiler.require(!o.has_default_value(),
                "flag_enum enumerator values are generated and cannot be specified explicitly");
            o.set_default_value(next_value);
            next_value *= 2;
        }
    }
    U none = 0;                                       // add name for no-flags-set value
};
```

28

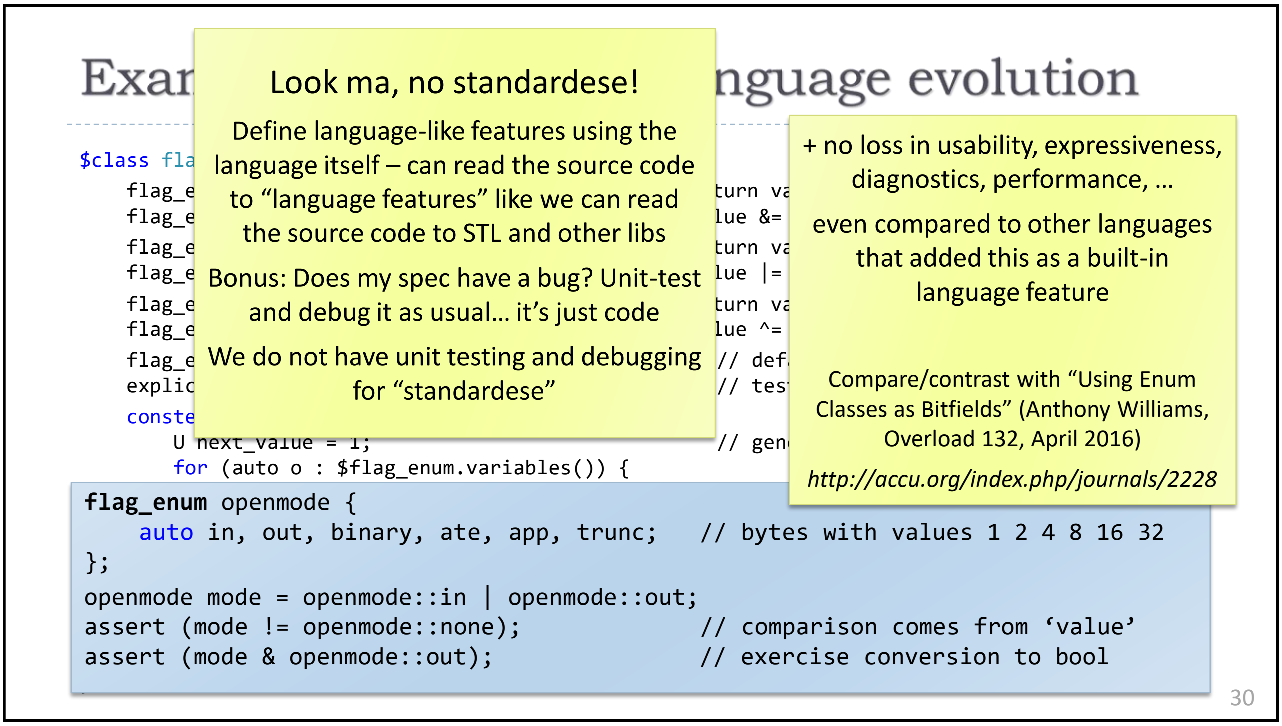


Example of simplifying language evolution

```
$class flag_enum : basic_enum {
    flag_enum  operator&  (const flag_enum& that) { return valu
    flag_enum& operator&= (const flag_enum& that) { value &= tha
    flag_enum  operator|  (const flag_enum& that) { return valu
    flag_enum& operator|= (const flag_enum& that) { value |= tha
    flag_enum  operator^  (const flag_enum& that) { return valu
    flag_enum& operator^= (const flag_enum& that) { value ^= tha
    flag_enum()              { value  = none; }       // defaul
    explicit operator bool() { value != none; }       // test a
    constexpr {
        U next_value = 1;                             // generate powers-of-two values
        for (auto o : $flag_enum.variables()) {
```

```
flag_enum openmode {
    auto in, out, binary, ate, app, trunc;   // bytes with values 1 2 4 8 16 32
};
openmode mode = openmode::in | openmode::out;
assert (mode != openmode::none);             // comparison comes from ‘value’
assert (mode & openmode::out);               // exercise conversion to bool
```

29



Exar nguage evolution

```
$class fla
    flag_e                                              turn va
    flag_e                                              lue &=
    flag_e                                              turn va
    flag_e                                              lue |=
    flag_e                                              turn va
    flag_e                                              lue ^=
    flag_e                                              // def
    explic                                              // tes
    conste
        U next_value = 1;                               // gen
        for (auto o : $flag_enum.variables()) {
```

Look ma, no standardese!

Define language-like features using the language itself – can read the source code to "language features" like we can read the source code to STL and other libs

Bonus: Does my spec have a bug? Unit-test and debug it as usual… it's just code

We do not have unit testing and debugging for "standardese"

+ no loss in usability, expressiveness, diagnostics, performance, …

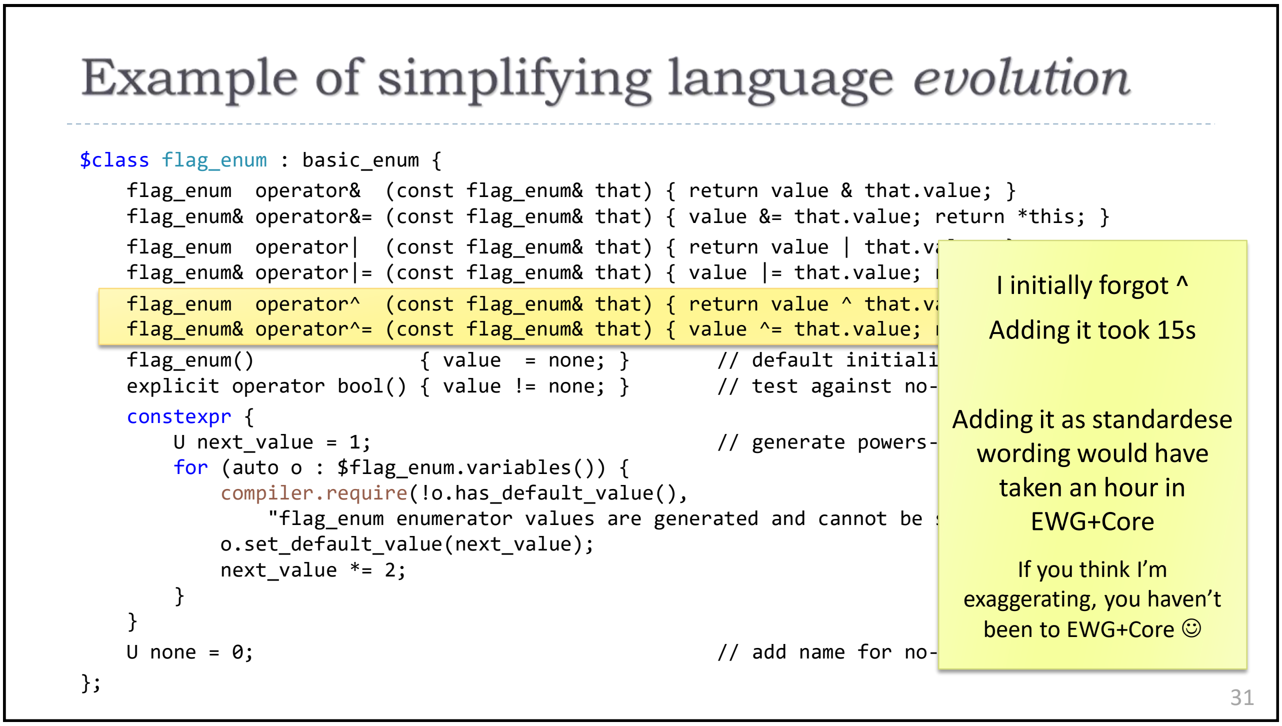
even compared to other languages that added this as a built-in language feature

Compare/contrast with "Using Enum Classes as Bitfields" (Anthony Williams, Overload 132, April 2016)

http://accu.org/index.php/journals/2228

```
flag_enum openmode {
    auto in, out, binary, ate, app, trunc;   // bytes with values 1 2 4 8 16 32
};
openmode mode = openmode::in | openmode::out;
assert (mode != openmode::none);             // comparison comes from 'value'
assert (mode & openmode::out);               // exercise conversion to bool
```

30



Example of simplifying language *evolution*

```
$class flag_enum : basic_enum {
    flag_enum  operator&  (const flag_enum& that) { return value & that.value; }
    flag_enum& operator&= (const flag_enum& that) { value &= that.value; return *this; }
    flag_enum  operator|  (const flag_enum& that) { return value | that.va
    flag_enum& operator|= (const flag_enum& that) { value |= that.value; 
    flag_enum  operator^  (const flag_enum& that) { return value ^ that.va
    flag_enum& operator^= (const flag_enum& that) { value ^= that.value; 
    flag_enum()               { value  = none; }        // default initiali
    explicit operator bool() { value != none; }        // test against no-
    constexpr {
        U next_value = 1;                               // generate powers-
        for (auto o : $flag_enum.variables()) {
            compiler.require(!o.has_default_value(),
                "flag_enum enumerator values are generated and cannot be 
            o.set_default_value(next_value);
            next_value *= 2;
        }
    }
    U none = 0;                                         // add name for no-
};
```

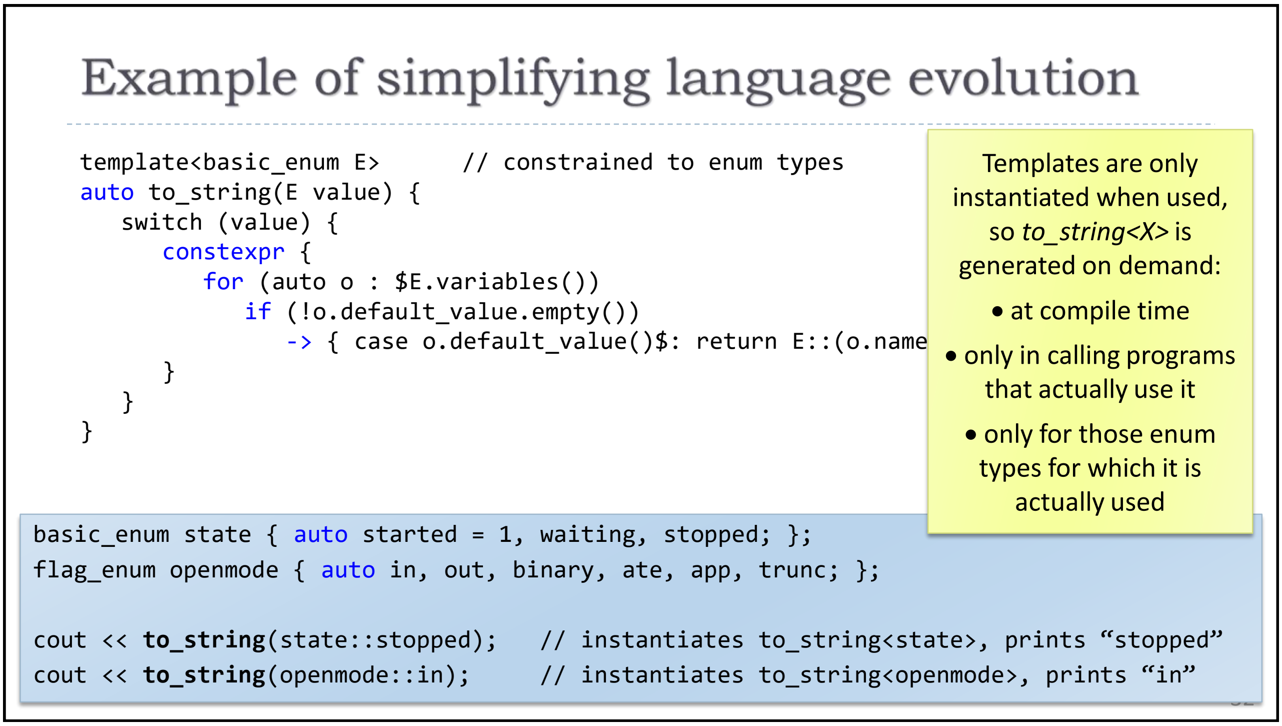
I initially forgot ^

Adding it took 15s

Adding it as standardese wording would have taken an hour in EWG+Core

If you think I'm exaggerating, you haven't been to EWG+Core ☺

31



Example of simplifying language evolution

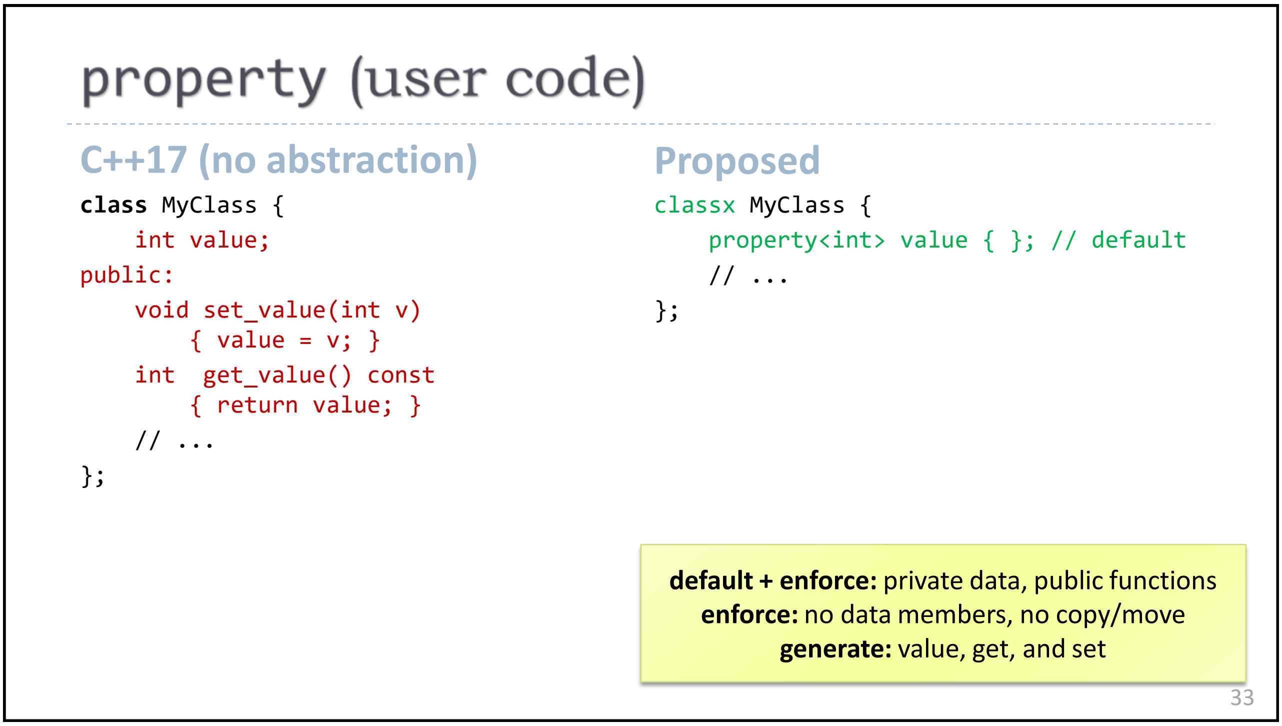
```
template<basic_enum E>      // constrained to enum types
auto to_string(E value) {
   switch (value) {
      constexpr {
         for (auto o : $E.variables())
            if (!o.default_value.empty())
               -> { case o.default_value()$: return E::(o.name
      }
   }
}
```

Templates are only instantiated when used, so *to_string<X>* is generated on demand:

- at compile time
- only in calling programs that actually use it
- only for those enum types for which it is actually used

```
basic_enum state { auto started = 1, waiting, stopped; };
flag_enum openmode { auto in, out, binary, ate, app, trunc; };

cout << to_string(state::stopped);   // instantiates to_string<state>, prints “stopped”
cout << to_string(openmode::in);     // instantiates to_string<openmode>, prints “in”
```

32



property (user code)

C++17 (no abstraction)

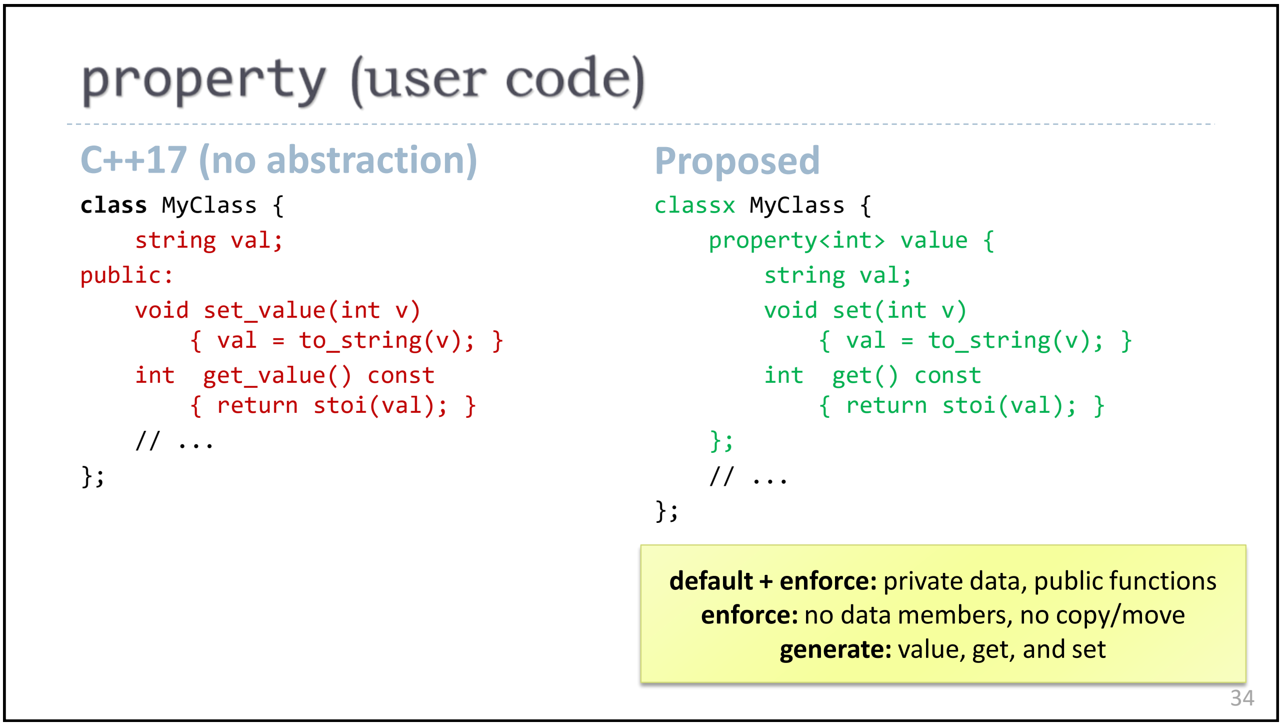
```
class MyClass {
    int value;
public:
    void set_value(int v)
        { value = v; }
    int  get_value() const
        { return value; }
    // ...
};
```

Proposed

```
classx MyClass {
    property<int> value { }; // default
    // ...
};
```

default + enforce: private data, public functions
enforce: no data members, no copy/move
generate: value, get, and set

33



property (user code)

C++17 (no abstraction)

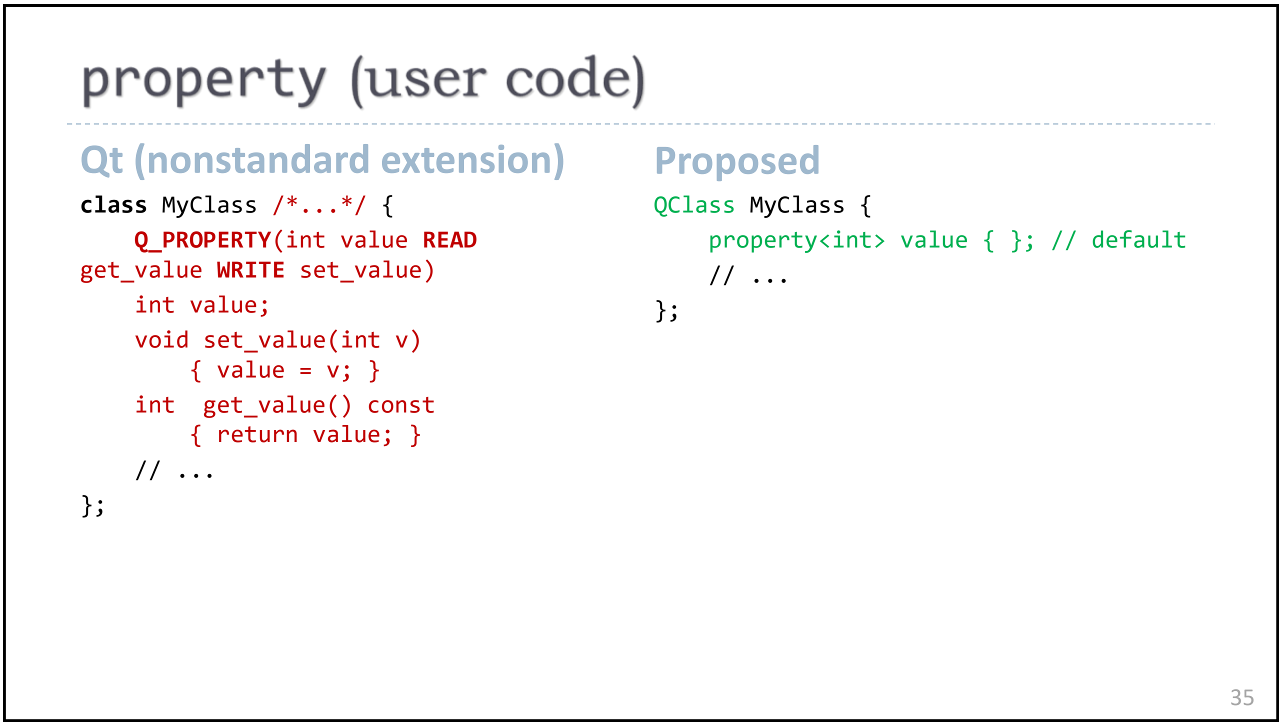
```
class MyClass {
    string val;
public:
    void set_value(int v)
        { val = to_string(v); }
    int  get_value() const
        { return stoi(val); }
    // ...
};
```

Proposed

```
classx MyClass {
    property<int> value {
        string val;
        void set(int v)
            { val = to_string(v); }
        int  get() const
            { return stoi(val); }
    };
    // ...
};
```

default + enforce: private data, public functions
enforce: no data members, no copy/move
generate: value, get, and set

34



property (user code)

Qt (nonstandard extension)

```
class MyClass /*...*/ {
    Q_PROPERTY(int value READ
get_value WRITE set_value)
    int value;
    void set_value(int v)
        { value = v; }
    int  get_value() const
        { return value; }
    // ...
};
```

Proposed

```
QClass MyClass {
    property<int> value { }; // default
    // ...
};
```

35

property (user code)

Qt (nonstandard extension)

```
class MyClass /*...*/ {
    Q_PROPERTY(int value_2x READ
get_value WRITE set_value)
    int value_2x;
    void set_value(int v)
        { value_2x = v*2; }
    int  get_value() const
        { return value_2x/2; }
    // ...
};
```

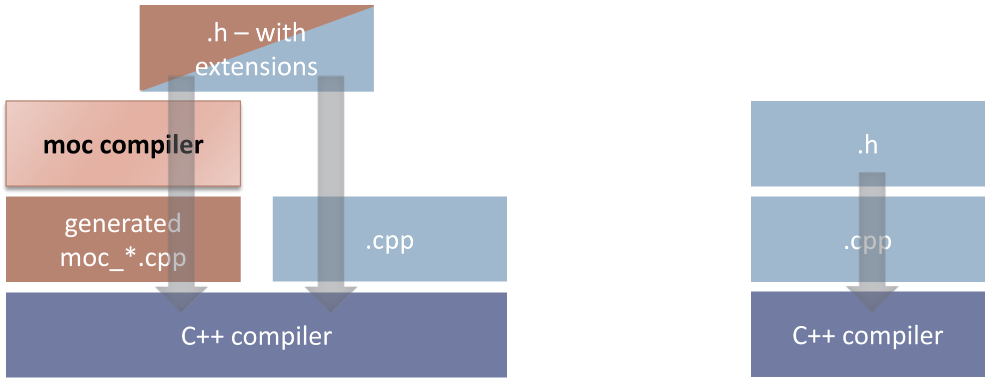
Proposed

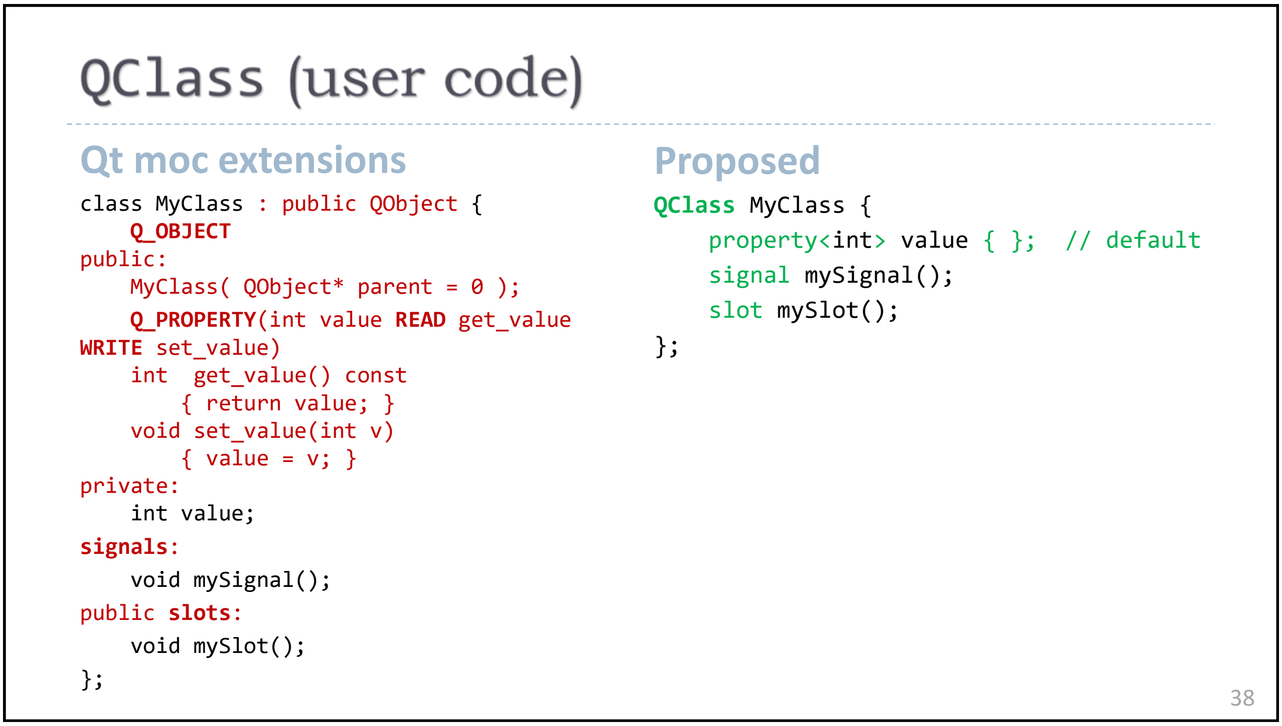
```
QClass MyClass {
    property<int> value {
        int  value_2x;
        void set(int v)
            { value_2x = v*2; }
        int  get() const
            { return value_2x/2; }
    };
    // ...
};
```

When you can’t express it all in C++ code

Qt moc

Proposed





QClass (user code)

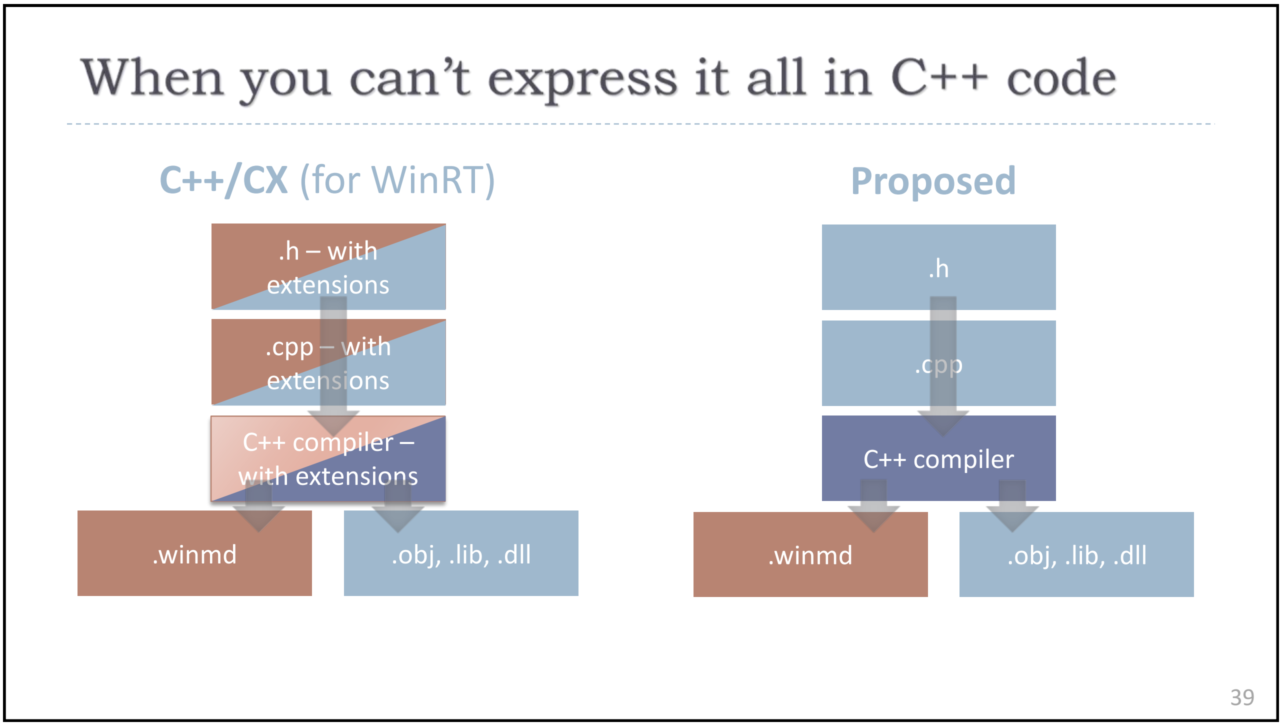
Qt moc extensions

```
class MyClass : public QObject {
    Q_OBJECT
public:
    MyClass( QObject* parent = 0 );
    Q_PROPERTY(int value READ get_value
WRITE set_value)
    int  get_value() const
        { return value; }
    void set_value(int v)
        { value = v; }
private:
    int value;
signals:
    void mySignal();
public slots:
    void mySlot();
};
```

Proposed

```
QClass MyClass {
    property<int> value { };  // default
    signal mySignal();
    slot mySlot();
};
```

38



When you can’t express it all in C++ code

C++/CX (for WinRT)

.h – with extensions

.cpp – with extensions

C++ compiler – with extensions

.winmd

.obj, .lib, .dll

Proposed

.h

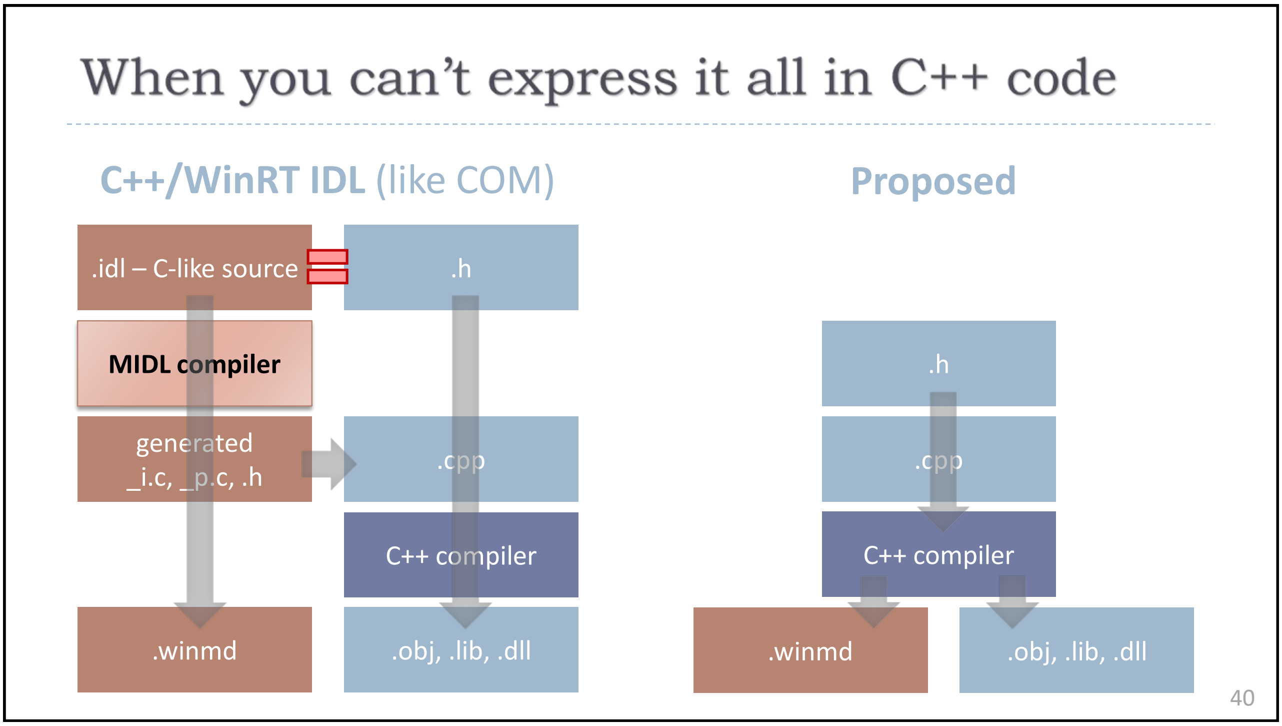
.cpp

C++ compiler

.winmd

.obj, .lib, .dll

39



When you can’t express it all in C++ code

C++/WinRT IDL (like COM)

.idl – C-like source

.h

MIDL compiler

generated
_i.c, _p.c, .h

.cpp

C++ compiler

.winmd

.obj, .lib, .dll

Proposed

.h

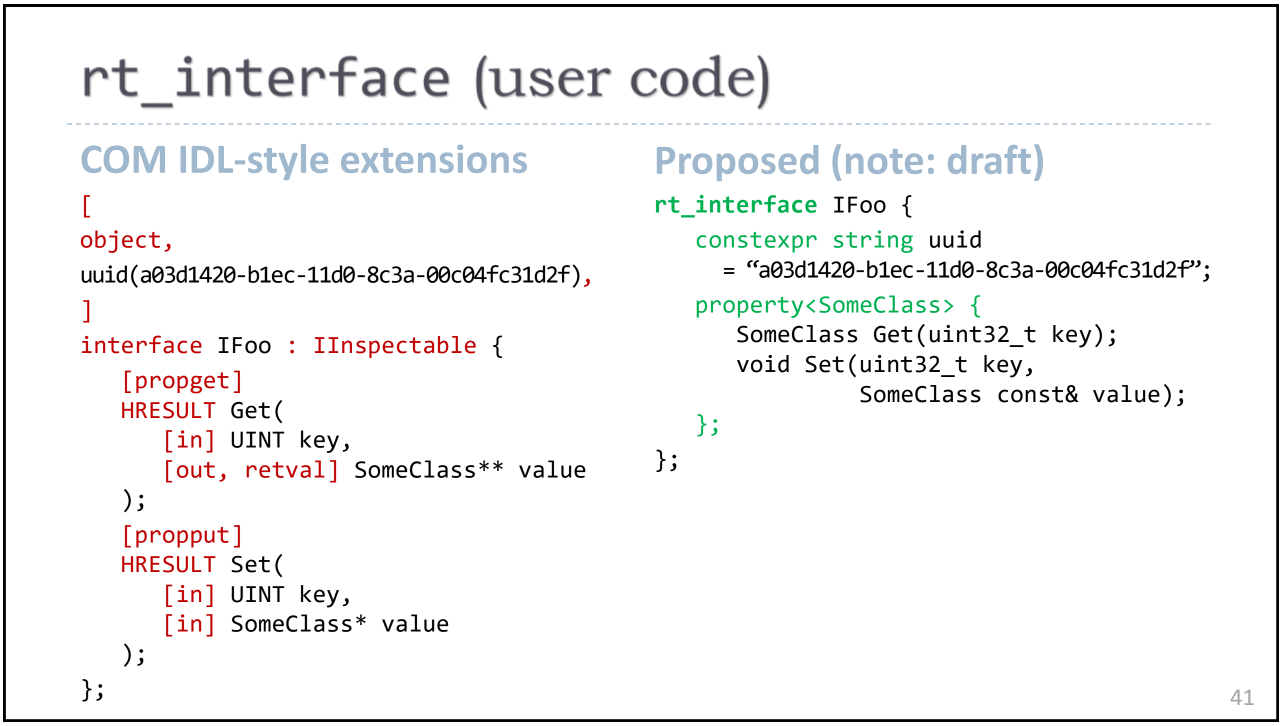
.cpp

C++ compiler

.winmd

.obj, .lib, .dll

40



rt_interface (user code)

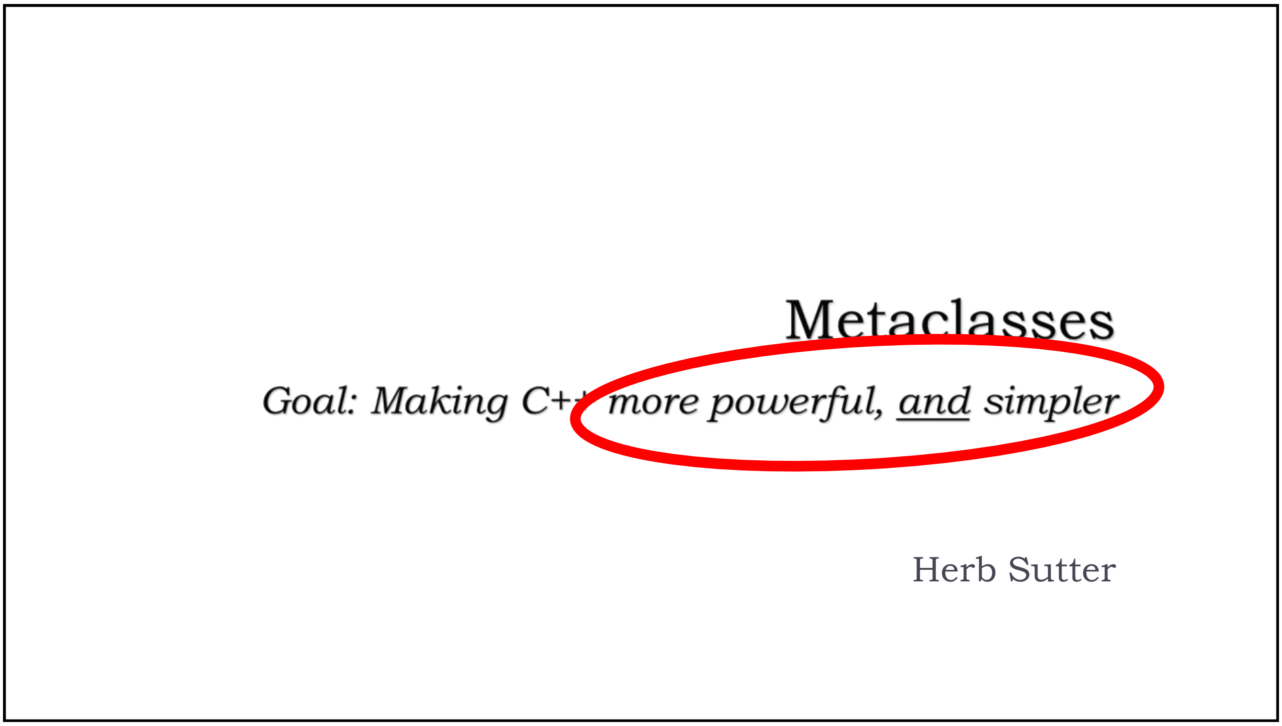
COM IDL-style extensions

```
[
object,
uuid(a03d1420-b1ec-11d0-8c3a-00c04fc31d2f),
]
interface IFoo : IInspectable {
   [propget]
   HRESULT Get(
      [in] UINT key,
      [out, retval] SomeClass** value
   );
   [propput]
   HRESULT Set(
      [in] UINT key,
      [in] SomeClass* value
   );
};
```

Proposed (note: draft)

```
rt_interface IFoo {
   constexpr string uuid
      = “a03d1420-b1ec-11d0-8c3a-00c04fc31d2f”;
   property<SomeClass> {
      SomeClass Get(uint32_t key);
      void Set(uint32_t key,
               SomeClass const& value);
   };
};
```

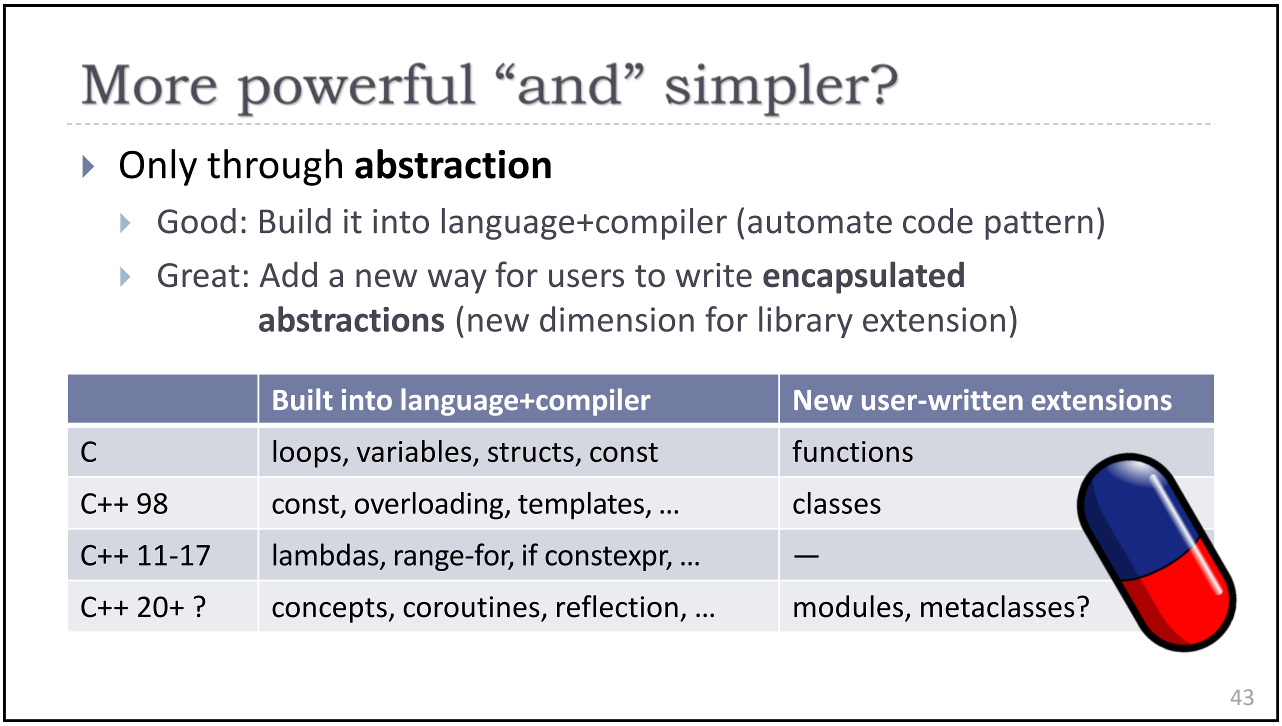
41



Metaclasses

Goal: Making C++ more powerful, and simpler

Herb Sutter

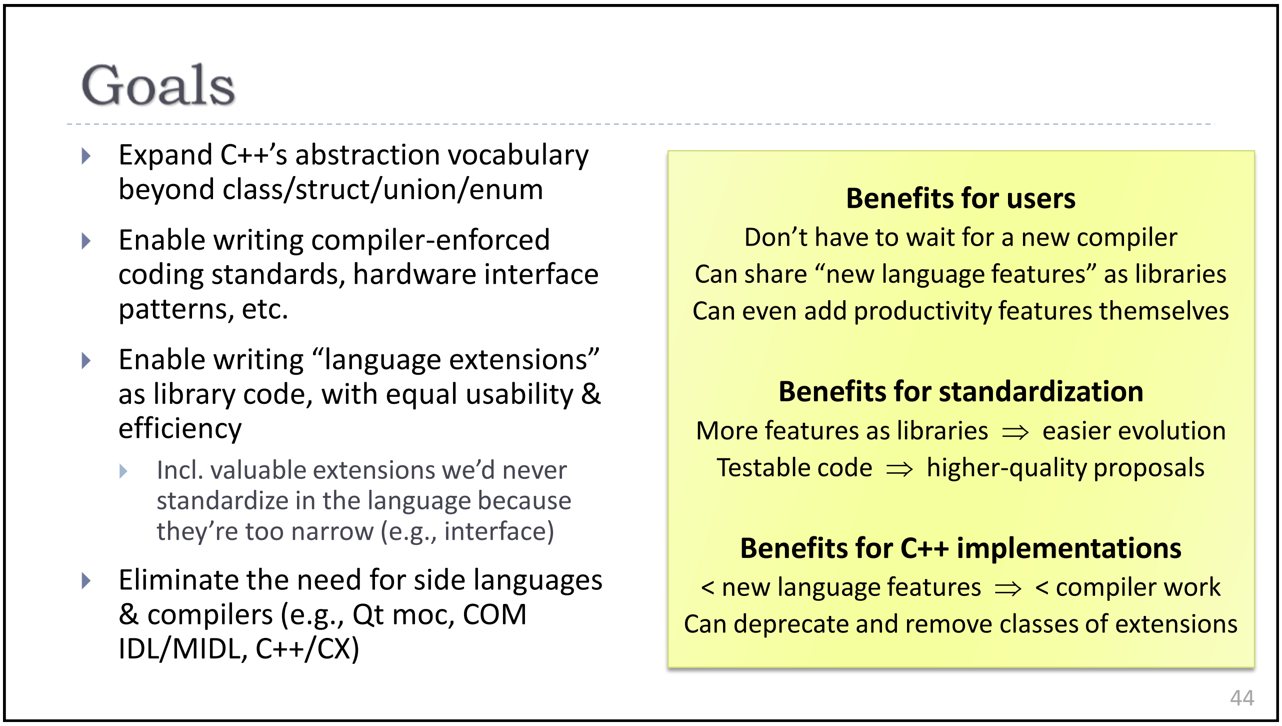


More powerful “and” simpler?

- Only through **abstraction**
  - Good: Build it into language+compiler (automate code pattern)
  - Great: Add a new way for users to write **encapsulated abstractions** (new dimension for library extension)

	Built into language+compiler	New user-written extensions
C	loops, variables, structs, const	functions
C++ 98	const, overloading, templates, …	classes
C++ 11-17	lambdas, range-for, if constexpr, …	—
C++ 20+ ?	concepts, coroutines, reflection, …	modules, metaclasses?

43



Goals

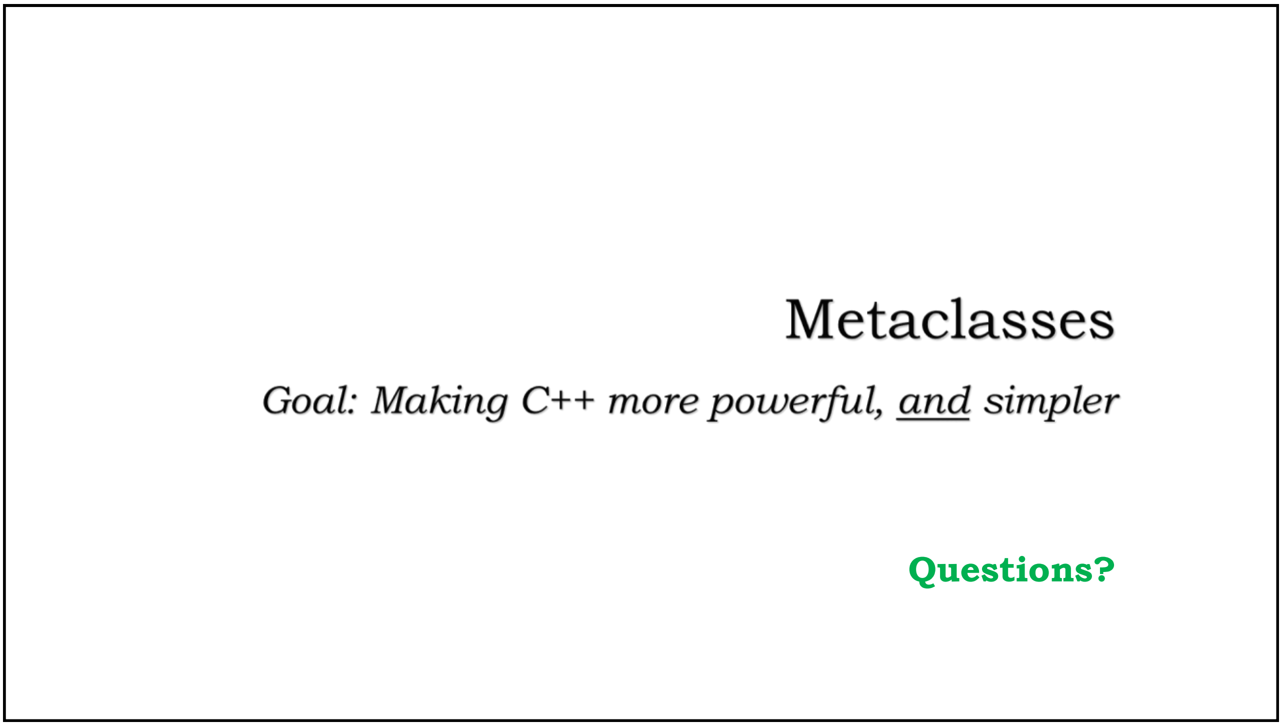
- Expand C++'s abstraction vocabulary beyond class/struct/union/enum
- Enable writing compiler-enforced coding standards, hardware interface patterns, etc.
- Enable writing "language extensions" as library code, with equal usability & efficiency
  - Incl. valuable extensions we'd never standardize in the language because they're too narrow (e.g., interface)
- Eliminate the need for side languages & compilers (e.g., Qt moc, COM IDL/MIDL, C++/CX)

Benefits for users
Don't have to wait for a new compiler
Can share "new language features" as libraries
Can even add productivity features themselves

Benefits for standardization
More features as libraries ⇒ easier evolution
Testable code ⇒ higher-quality proposals

Benefits for C++ implementations
< new language features ⇒ < compiler work
Can deprecate and remove classes of extensions

44



Metaclasses

Goal: Making C++ more powerful, and simpler

Questions?