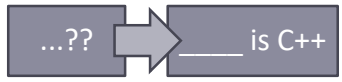
Thoughts on a more powerful *and* simpler C++ (5 of N)

Herb Sutter

Roadmap: 3 "talks"

- Context: More powerful *and* simpler



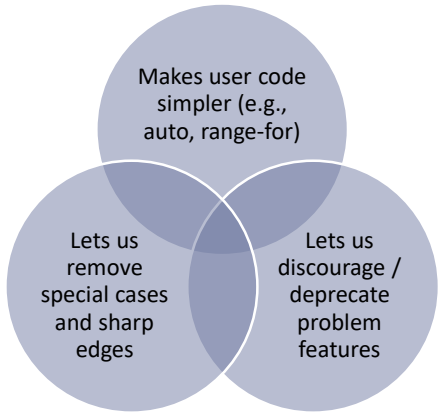
- Update: **Lifetime** (simplifying dangling problems, from CppCon 2015)

- Update: **Metaclasses** (simplifying class authoring, from CppCon 2017)

2

Simplify by adding stuff… wait, what?

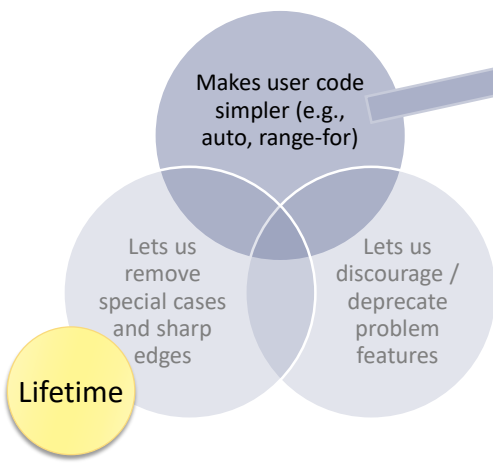
Yes, if the added thing…



3

Simplify by adding stuff… wait, what?

Yes, if the added thing…



Which we achieve via abstraction…

Gain	Elevate an idiom or pattern to…	Examples
K	Language feature (a "**fixed name**" in the form of key"word" or specific syntax)	range-for lambda …
N	User-defined abstraction (**named meaning**)	concept …
K^N	User-defined encapsulated abstraction (**named behavior**)	variable, function, class module?

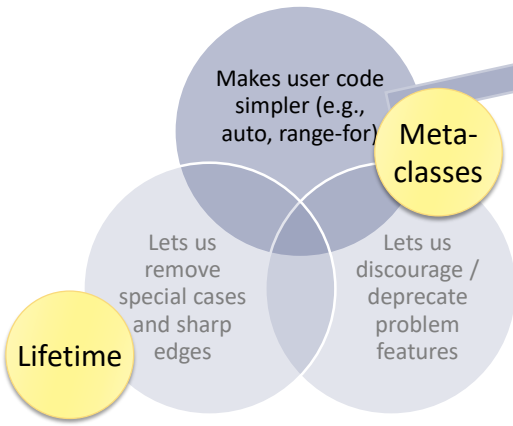
Meta-classes

name ⇒ *Word of Power*
encapsulated ⇒ *composable, combinatorial*

4

Simplify by adding stuff… wait, what?

Yes, if the added thing…



Which we achieve via abstraction…

Gain	Elevate an idiom or pattern to…	Examples
K	Language feature (a "**fixed name**" in the form of key"word" or specific syntax)	range-for lambda …
N	User-defined abstraction (**named meaning**)	concept …
K^N	User-defined encapsulated abstraction (**named behavior**)	variable, function, class module?

name ⇒ *Word of Power*
encapsulated ⇒ *composable, combinatorial*

5

Pop quiz: Which of these "is C++"?

Then: C++98 code

```
circle* p = new circle( 42 );
vector<shape*> v = load_shapes();
for( vector<shape*>::iterator i = v.begin(); i != v.end(); ++i ) {
  if( *i && **i == *p )
    cout << **i << " is a match\n";
}
// … later, possibly elsewhere …
for( vector<shape*>::iterator i = v.begin();
     i != v.end(); ++i ) {
  delete *i;
}
delete p;
```

Now: Modern C++

```
auto p = make_shared<circle>( 42 );
auto v = load_shapes();
for( auto& s : v ) {
  if( s && *s == *p )
    cout << *s << " is a match\n";
}
```

"C++11 feels like a new language"
— Bjarne Stroustrup

6

Pop quiz: Which of these "is C++"?

Q: How many of you found this jarring or unfamiliar at first?

Then: C++98 code

```
int add(int i, int j);
int sub(int i, int j) {  return i-j;  }
template<class T>
T min(T a, T b) {  return b<a ? b : a;  }
```

Now: Modern C++

```
auto add(int i, int j) -> int;
auto sub(int i, int j) {  return i-j;  }
auto min = [](auto a, auto b) {  return b<a ? b : a;  };
```

Then: C++11 code

```
template<class T, class U>
auto add(T a, U b) -> decltype(a+b) {
   static_assert(std::is_arithmetic<T>::value &&
                 std::is_arithmetic<U>::value);
   return a+b;
}
```

Soon?

```
auto add(Number auto a, Number auto b)
  {  return a+b;  }
```

Serious Q: Which of the "new" examples so far are **"not C++ic"** — i.e, not in the spirit of C++?

7

So what *is* C++ / "C++ic"?

Historical strength: **static**
typing, compilation, linking

Major current trend (IME): ↓dynamic, ↑**static**
concepts, contracts, constexpr, static error types (fs_error), virtual → templates, RTTI's typeid → reflection, …

- Core principles:
  - Zero-overhead abstraction — don't pay for what you don't use, what you use is as efficient as you can reasonably write by hand
  - Determinism & control — over time & space, close to hardware, leave no room for a lower language, trust the programmer
  - Link compatibility — with C95 and C++prev
- Useful pragmatics for adoption and library consumption:
  - Backward source compat with C — mostly C95, read-only, consume headers & seamlessly call
  - Backward source compat with C++prev — C++98 and later, read-mostly, to use & to maintain
- What is *not* core C++:
  - Specific syntax — we've been adding & recommending C-incompatible syntax since 1979
  - Tedium — most modern abstractions (e.g., range-*for*, *override*) compatible with zero-overhead
  - Lack of good defaults — good defaults are fine, as long as the programmer can override them
  - Sharp edges — e.g., brittle dangling pointers are not necessary for efficiency and control

8

What's "good" for C++-ic evolution?

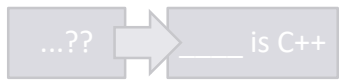
- When proposing a new C++ feature, ask:
  - How does this let the programmer **express intent** more directly?
  - How does this **simplify** C++ code (read, write, debug, maintain)?
  - How does this **remove** gotchas / special cases / classes of errors?
  - How does this **remove** a place where C++ doesn't currently meet one of its key principles, such as zero-overhead or deterministic control?
  - How does this help potentially **deprecate** something from the language? (*cough* macros)
- Example: "using" alias replaces "typedef."
  - Can teach "don't use typedef in new code." But: typedef still works for compatibility.
  - Generally: We've been adding new syntax and saying "don't use the old one" for years.

Suggestion: Let's start viewing backward source compatibility as a "zero-overhead" feature, pay for it only if you use it (stick to "using" in new code ⇒ simpler, less to teach and learn).

9

Roadmap: 3 "talks"

- Context: More powerful *and* simpler



- Update: **Lifetime** (simplifying dangling problems, from CppCon 2015)

- Update: **Metaclasses** (simplifying class authoring, from CppCon 2017)

10

Lifetime update

Kyle Reed (Microsoft) | Neil MacIntosh (Facebook) | Matthias Gehre (Silexica) | Gábor Horváth (Microsoft, Eötvös Loránd University)

- Goal: Validate approach
  - Same as presented in 2015
  - Progress: Shift emphasis from whole-program guarantees to diagnosing common cases
- Goal: Portable results
  - Progress: **Clang** and **VC++** implementations (both partial, in progress)
- Goal: Efficient enough to run during normal compilation
  - Progress: VC++ implementation in IDE static analysis, fast enough to run during editing
  - Progress: Clang implementation in compiler, ~5% compile-time overhead on large LLVM files (no optimization yet)

11

Overview of approach

Core model

- **Local CFG path** rules, **statically** enforced
  - 100% compile-time, 0 run-time overhead
- Identify Owners, **track** Pointers
  - Every local Pointer tracks its "points to set" (pset) of current pointees
- For function calls:
  - **Inputs validated in caller** and assumed in callee: Prevent passing an invalid pointer, or a Pointer the callee could invalidate
  - **Outputs validated in callee** and assumed in caller: Prevent returning in invalid pointer

Defaults (can use heuristics)

- **Goal: Few annotations**
- How: Make common cases the default
  - **Default**-detect types that are Owners (e.g., container, smart pointer) or Pointers (e.g., iterator, range)
  - **Default** function calls rules assume Pointer outputs are derived from Owner and Pointer parameters – correct for the majority of cases
  - When the defaults aren't what you want, use *[[lifetime…]]* annotation to opt out

12

Starter example: Dangling raw pointer

```
// godbolt.org/z/szJjnH

int* p = nullptr;          // pset(p) = {null} – records that now p is null
{
   int x = 0;
   p = &x;                 // A: set pset(p) = {x} – records that now p points to x
   cout << *p;             // B: ok – *p is ok because {x} is alive
}                          // C: x destroyed ⇒ replace "x" with "invalid" in all psets
                           //                ⇒ set pset(p) = {invalid}
cout << *p;                // D: error – because pset(p) contains {invalid}
```

13

Generalizing to user-defined Pointer types

```
// godbolt.org/z/ytRTKt

string_view s;             // pset(s) = {null} – records that now p is null
{
   char a[100];
   s = a;                  // A: set pset(s) = {a} – records that now s points to a
   cout << s[0];           // B: ok – s[0] is ok because {a} is alive
}                          // C: a destroyed ⇒ replace "a" with "invalid" in all psets
                           //                ⇒ set pset(s) = {invalid}
cout << s[0];              // D: error – because pset(s) contains {invalid}
```

14

Generalizing to user-defined Owner types

```
// godbolt.org/z/1Sc_cE

string_view s;              // pset(s) = {null} – records that now p is null

string name = “abcdefghijklmnop”;
s = name;                   // A: set pset(s) = {name'} – “data owned by ‘name’”
cout << s[0];               // B: ok – s[0] is ok because {name} is alive
name = “frobozz”;           // C: name modified ⇒ change “name'” to “invalid” in psets
                            //                  ⇒ set pset(s) = {invalid}
cout << s[0];               // D: error – because pset(s) contains {invalid}
```

15

More user-defined Pointer and Owner types

```
// godbolt.org/z/iq5Qja

vector<bool> vb{false,true,false,true};

auto proxy = vb[0];         // A: set pset(proxy) = {vb'} – “data owned by ‘vb’”

cout << proxy;              // B: ok – proxy is ok because {vb} is alive
vb.reserve(100);            // C: vb modified ⇒ change “vb'” to “invalid” in psets
                            //                ⇒ set pset(proxy) = {invalid}
cout << proxy;              // D: error – because pset(proxy) contains {invalid}
```

16

Function calls: Default return/out lifetimes

- Default: A returned Pointer comes from Owner/Pointer inputs.
  - Vast majority of cases: Derived from Owner and Pointer arguments.

```
auto f( int* p, int* q )  -> int*;          // -> copy of p or q
auto g( string& s )  -> char*;              // -> s' (s-owned)
```

  - Params that are Owner rvalue weak magnets: ***owner const&*** parameters
    - Ignored by default, because *owner const&* can bind to temporary owners.

```
auto find_match( string& s, const string& sub )  -> char*;     // -> s'
```

    - Only if there are no other candidates, consider owner weak rvalue magnets.

```
auto point_into( const string& s, const string& sub )  -> char*;  // -> s',sub'
```

17

Demos

18

Demos


Matthias Gehre (Silexica)

Gábor Horváth (Microsoft, Eötvös Loránd University)

Thursday, September 27 • 14:00 - 15:00

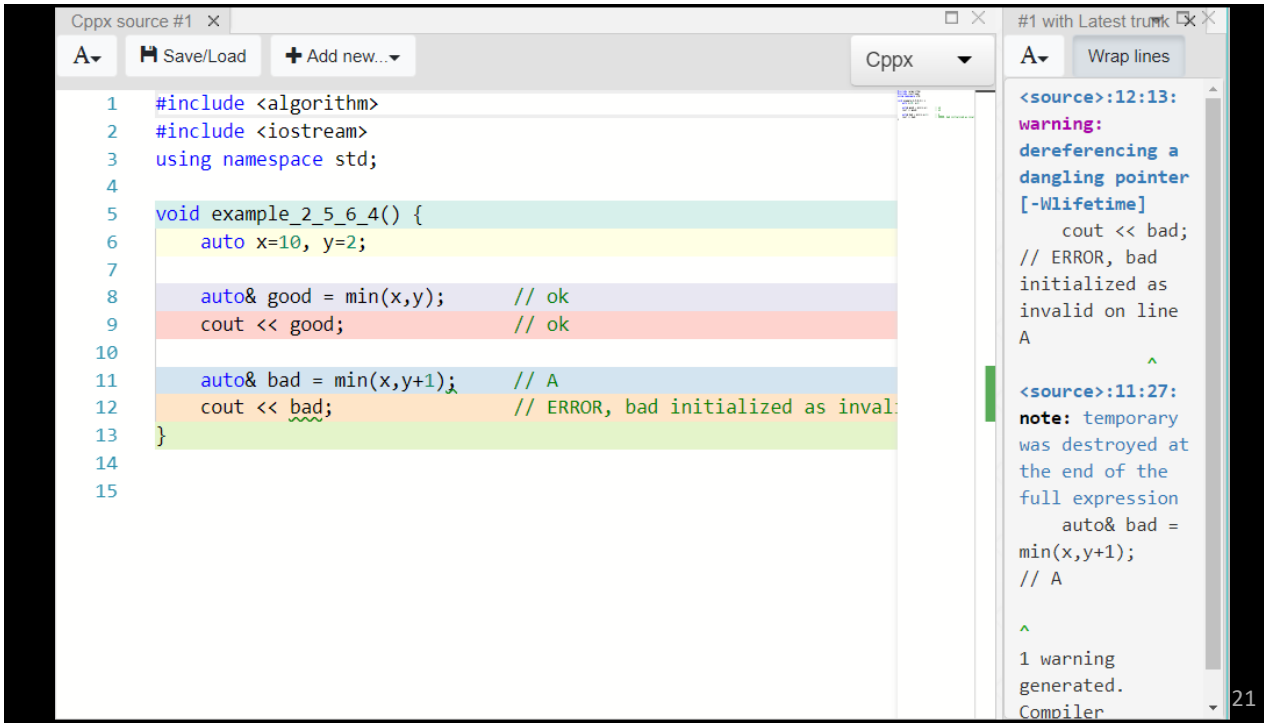
Implementing the C++ Core Guidelines' Lifetime Safety Profile in Clang

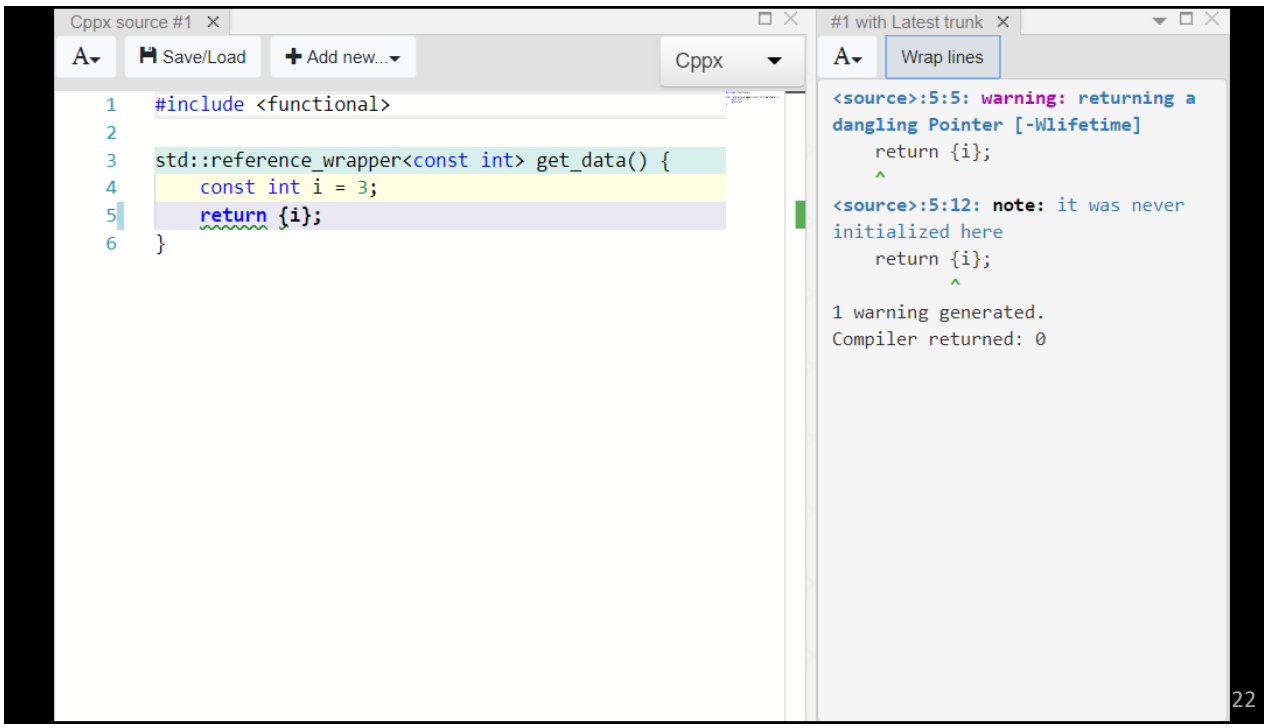
19

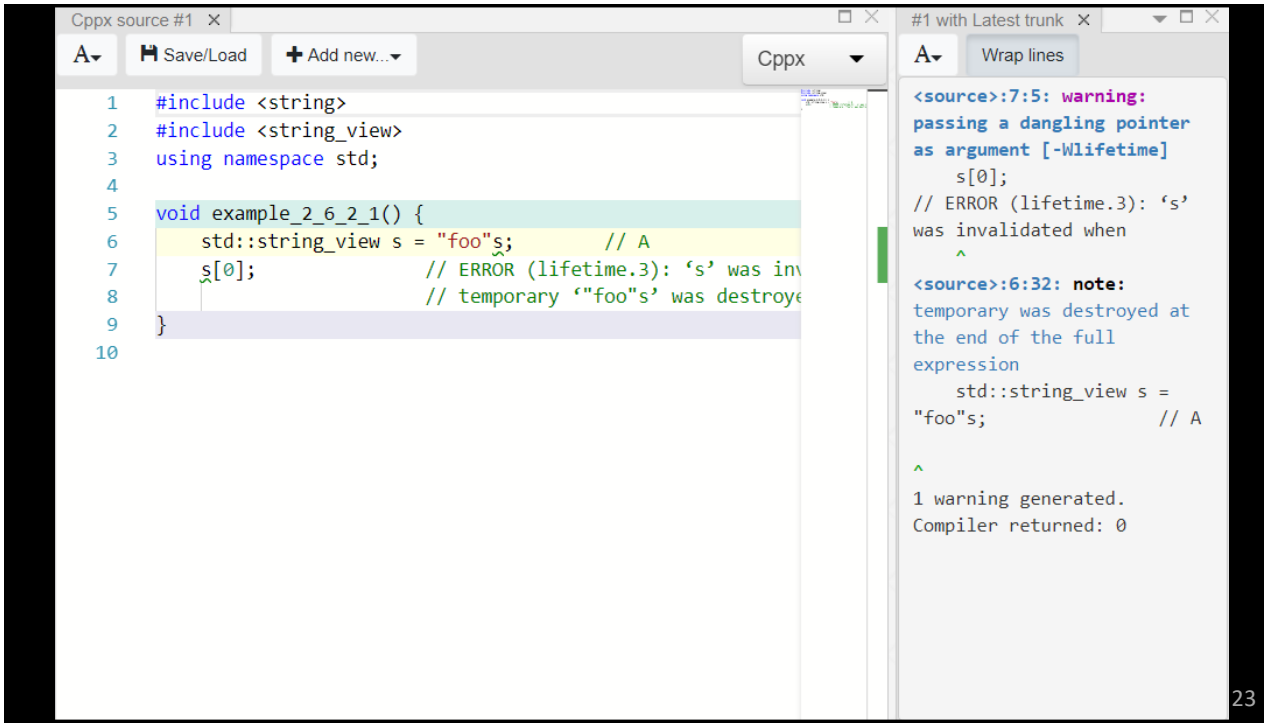
```
3
4   #include <algorithm>
5   #include <iostream>
6   using namespace std;
7
8   void example_2_5_6_4() {
9       auto x = 10, y = 2;
10
11      auto& good = min(x, y);      // ok
12      cout << good;                // ok
13
14      auto& bad = min(x, y + 1);   // A
15      cout << bad;                 // ERROR, bad initialized as invalid on line A
```

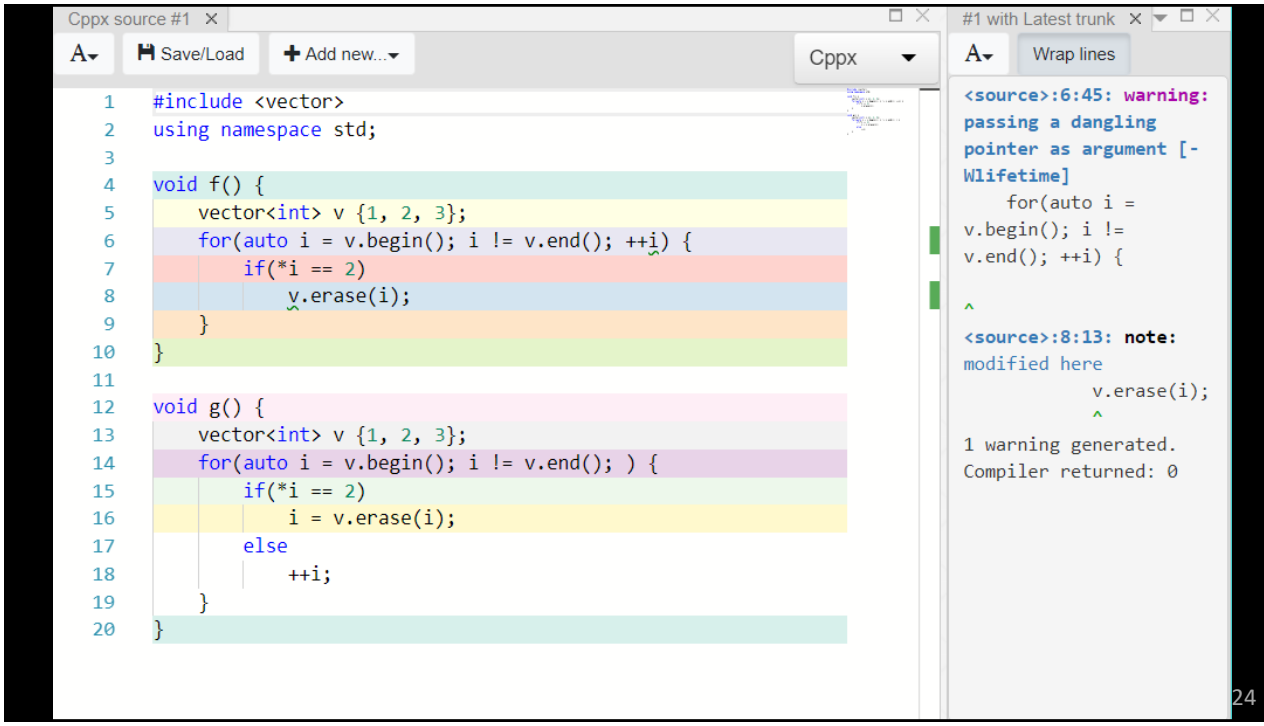
```
const int &bad
A
26489: Don't dereference a pointer that may be invalid: 'bad'. 'bad' may have been invalidated at line 14 (lifetime.1).
```

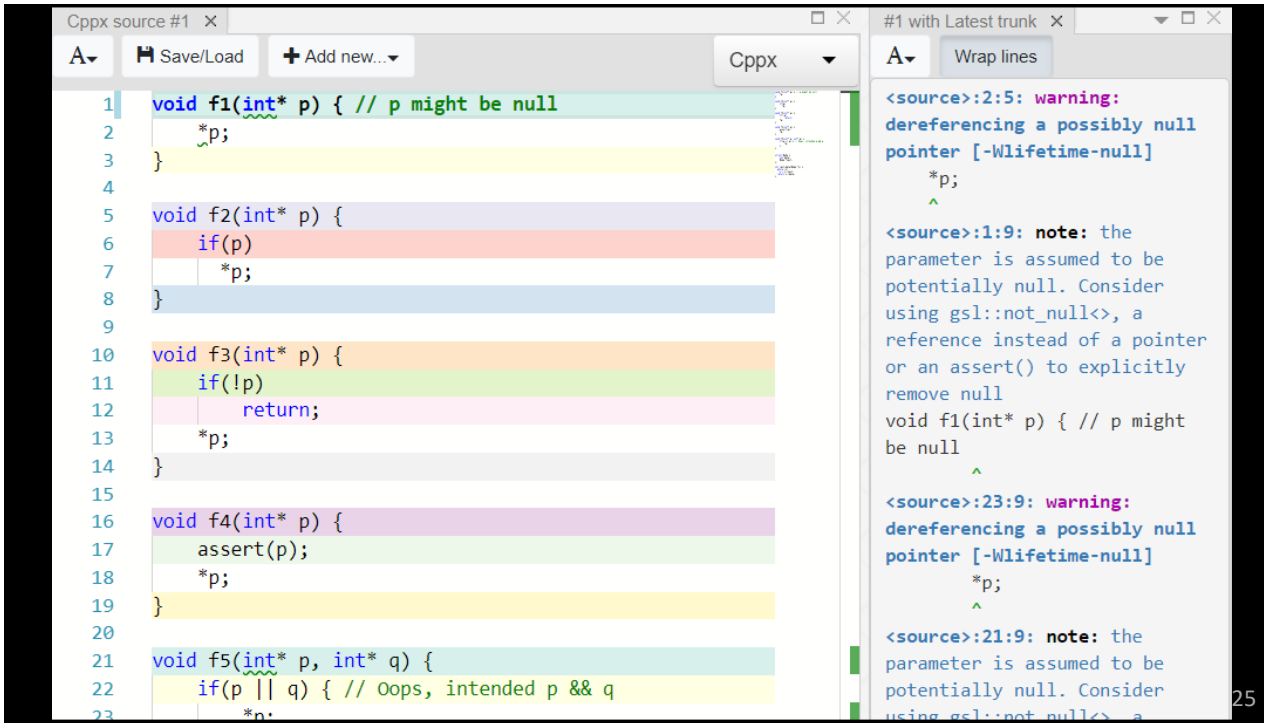
```
20
21
22
23
24
```

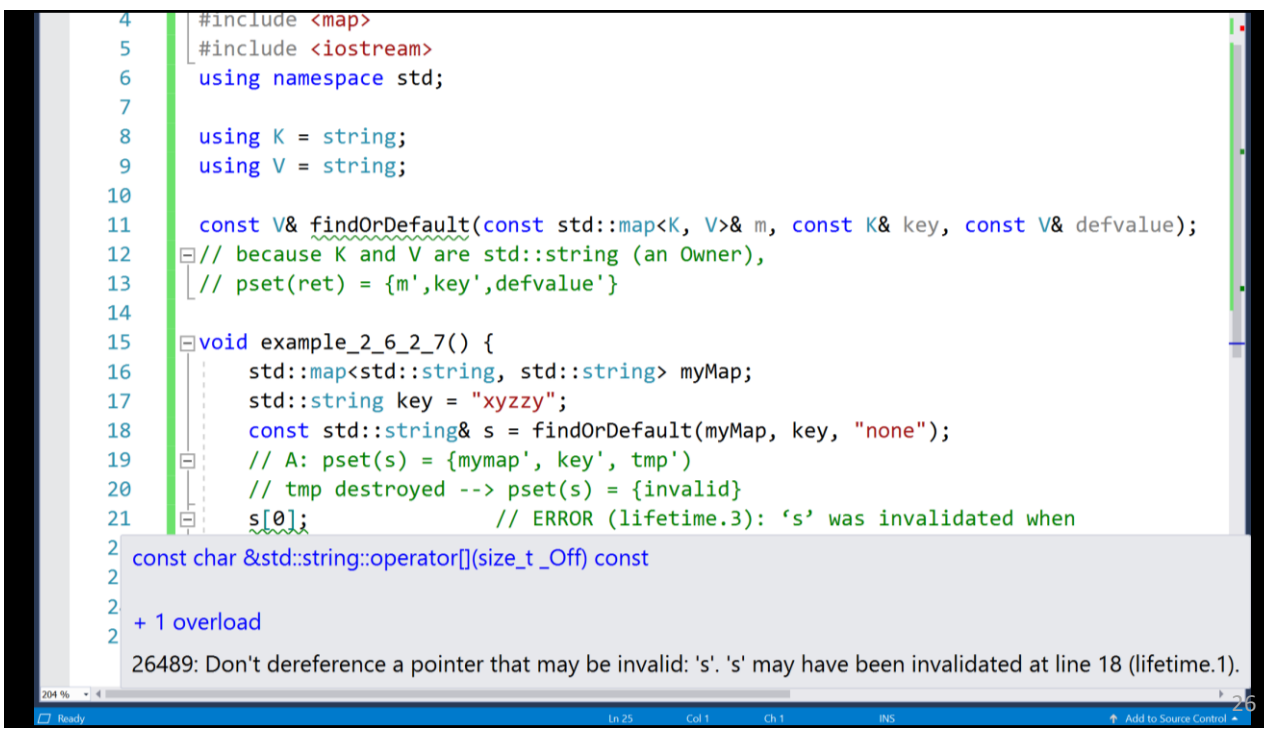

Cppx source #1
Save/Load
Add new...
Cppx
#include <algorithm>
#include <iostream>
using namespace std;
void example_2_5_6_4() {
auto x=10, y=2;
auto& good = min(x,y); // ok
cout << good; // ok
auto& bad = min(x,y+1); // A
cout << bad; // ERROR, bad initialized as inval
}
#1 with Latest trunk
Wrap lines
<source>:12:13: warning: dereferencing a dangling pointer [-Wlifetime]
cout << bad;
// ERROR, bad initialized as invalid on line A
^
<source>:11:27: note: temporary was destroyed at the end of the full expression
auto& bad = min(x,y+1);
// A
^
1 warning generated.
Compiler
21


Cppx source #1
Save/Load
Add new...
Cppx
#include <functional>
std::reference_wrapper<const int> get_data() {
const int i = 3;
return {i};
}
#1 with Latest trunk
Wrap lines
<source>:5:5: warning: returning a dangling Pointer [-Wlifetime]
return {i};
^
<source>:5:12: note: it was never initialized here
return {i};
^
1 warning generated.
Compiler returned: 0
22


Cppx source #1
Save/Load
Add new...
Cppx
#include <string>
#include <string_view>
using namespace std;
void example_2_6_2_1() {
std::string_view s = "foo"s; // A
s[0]; // ERROR (lifetime.3): 's' was inv
// temporary '"foo"s' was destroye
}
#1 with Latest trunk
Wrap lines
<source>:7:5: warning: passing a dangling pointer as argument [-Wlifetime]
s[0];
// ERROR (lifetime.3): 's' was invalidated when
^
<source>:6:32: note: temporary was destroyed at the end of the full expression
std::string_view s = "foo"s; // A
^
1 warning generated.
Compiler returned: 0
23


Cppx source #1
Save/Load
Add new...
Cppx
#include <vector>
using namespace std;
void f() {
vector<int> v {1, 2, 3};
for(auto i = v.begin(); i != v.end(); ++i) {
if(*i == 2)
v.erase(i);
}
}
void g() {
vector<int> v {1, 2, 3};
for(auto i = v.begin(); i != v.end();) {
if(*i == 2)
i = v.erase(i);
else
++i;
}
}
#1 with Latest trunk
Wrap lines
<source>:6:45: warning: passing a dangling pointer as argument [-Wlifetime]
for(auto i = v.begin(); i != v.end(); ++i) {
^
<source>:8:13: note: modified here
v.erase(i);
^
1 warning generated.
Compiler returned: 0
24


Cppx source #1
Save/Load
Add new...
Cppx
#1 with Latest trunk
Wrap lines
void f1(int* p) { // p might be null
*p;
}
void f2(int* p) {
if(p)
*p;
}
void f3(int* p) {
if(!p)
return;
*p;
}
void f4(int* p) {
assert(p);
*p;
}
void f5(int* p, int* q) {
if(p || q) { // Oops, intended p && q
<source>:2:5: warning: dereferencing a possibly null pointer [-Wlifetime-null]
*p;
^
<source>:1:9: note: the parameter is assumed to be potentially null. Consider using gsl::not_null<>, a reference instead of a pointer or an assert() to explicitly remove null
void f1(int* p) { // p might be null
^
<source>:23:9: warning: dereferencing a possibly null pointer [-Wlifetime-null]
*p;
^
<source>:21:9: note: the parameter is assumed to be potentially null. Consider
25


#include <map>
#include <iostream>
using namespace std;
using K = string;
using V = string;
const V& findOrDefault(const std::map<K, V>& m, const K& key, const V& defvalue);
// because K and V are std::string (an Owner),
// pset(ret) = {m',key',defvalue'}
void example_2_6_2_7() {
std::map<std::string, std::string> myMap;
std::string key = "xyzzy";
const std::string& s = findOrDefault(myMap, key, "none");
// A: pset(s) = {mymap', key', tmp')
// tmp destroyed --> pset(s) = {invalid}
s[0]; // ERROR (lifetime.3): 's' was invalidated when
const char &std::string::operator[](size_t _Off) const
+ 1 overload
26489: Don't dereference a pointer that may be invalid: 's'. 's' may have been invalidated at line 18 (lifetime.1).
Ready
Ln 25
Col 1
Ch 1
INS
Add to Source Control
26

```cpp
#include <map>
#include <iostream>
using namespace std;

using K = string;
using V = string;

const V& findOrDefault(const std::map<K,V>& m, const K& key, const V& de
                       // because K and V are std::string (an Owner),
                       // pset(ret) = {m',key',defvalue'}

void example_2_6_2_7() {
    std::map<std::string, std::string> myMap;
    std::string key = "xyzzy";
    const std::string& s = findOrDefault(myMap, key, "none");
                       // A: pset(s) = {mymap', key', tmp')
                       // tmp destroyed --> pset(s) = {invalid}
    s[0];              // ERROR (lifetime.3): 's' was invalidated when
                       // temporary '"none"' was destroyed (line A)
}
```

Wrap lines

```
<source>:18:5: warning: dereferencing a dangling pointer [-Wlifetime]
    s[0]; // ERROR (lifetime.3): 's' was invalidated when
    ^
<source>:15:61: note: temporary
```

27

```cpp
std::string operator+(std::string_view sv1, std::string_view
  return std::string(sv1) + std::string(sv2);
}

void f() {
    std::string_view hello = "hello ";
    std::string_view world = "ho";
    auto s = hello + world;
}

template <typename T>
T concat(const T &x, const T &y) {
  return x + y;
}

void sj4() {
  std::string s = "hi";
  auto c = concat(s, s);

  std::string_view hi = "ho";
  auto xy = concat(hi, hi);
}
```

```
<source>:16:3: warning: returning a Pointer with points-to set ((temporary)') where points-to set ((*x), (*y)) is expected [-Wlifetime]
  return x + y;
  ^
<source>:24:13: note: in instantiation of function template specialization 'concat<std::__1::basic_string_view<char, std::__1::char_traits<char> > >' requested here
  auto xy = concat(hi, hi);
            ^
1 warning generated.
Compiler returned: 0
```

28

```cpp
#include <vector>
#include <optional>

std::vector<int> getVec();
std::optional<std::vector<int>> getOptVec();

void sj3() {
  int sum = 0;
  for (int value : getVec()) {
      sum += value;
  }

  for (int value : *getOptVec()) {
      sum += value;
  }
}
```

```
<source>:13:18: warning: passing a dangling pointer as argument [-Wlifetime]
  for (int value : *getOptVec()) {
                   ^
<source>:13:20: note: temporary was destroyed at the end of the full expression
  for (int value : *getOptVec()) {
                   ^
1 warning generated.
Compiler returned: 0
```

29

Paging…

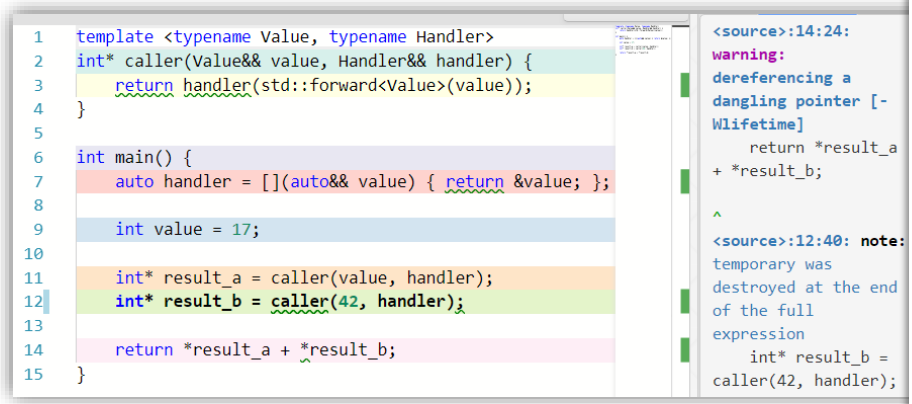
Hayun Ezra Chung

Stephane Guy

Isabella Muerte

30

Hayun Ezra Chung



```
1  template <typename Value, typename Handler>
2  int* caller(Value&& value, Handler&& handler) {
3      return handler(std::forward<Value>(value));
4  }
5
6  int main() {
7      auto handler = [](auto&& value) { return &value; };
8
9      int value = 17;
10
11     int* result_a = caller(value, handler);
12     int* result_b = caller(42, handler);
13
14     return *result_a + *result_b;
15 }
```

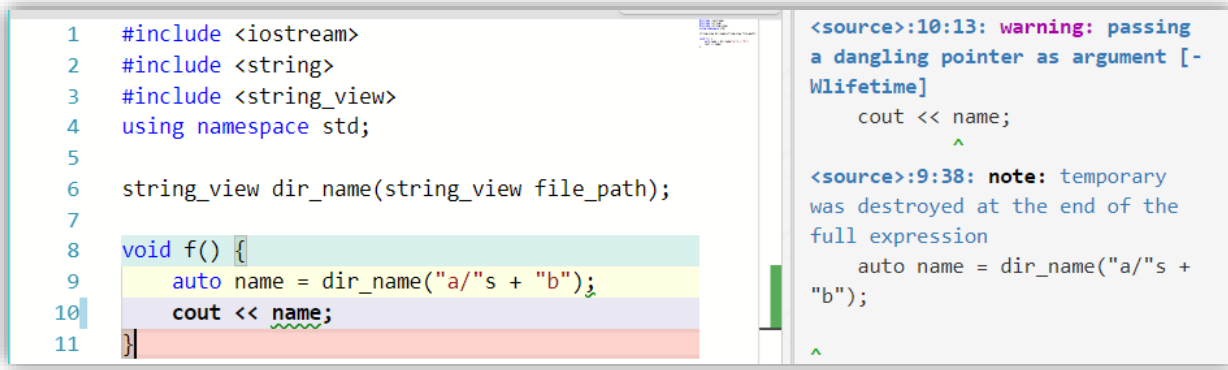
```
<source>:14:24: warning: dereferencing a dangling pointer [-Wlifetime]
    return *result_a + *result_b;
                       ^
<source>:12:40: note: temporary was destroyed at the end of the full expression
    int* result_b = caller(42, handler);
```

"Despite going through a template callback interface, with forwarding references and generic lambda expressions, the lifetime checker correctly deduces the dereference of 'result_b' as the problem point, and even notes object that left it dangling (the value '42'). Both the result and the cause are easily to spot."

31

Stephane Guy

"The string_view based dir_name method from Titus Winters' talk"



```
1  #include <iostream>
2  #include <string>
3  #include <string_view>
4  using namespace std;
5
6  string_view dir_name(string_view file_path);
7
8  void f() {
9      auto name = dir_name("a/"s + "b");
10     cout << name;
11 }
```

```
<source>:10:13: warning: passing a dangling pointer as argument [-Wlifetime]
    cout << name;
            ^
<source>:9:38: note: temporary was destroyed at the end of the full expression
    auto name = dir_name("a/"s + "b");
                                 ^
```

32

Isabella Muerte

```
1  #include <cstring>
2  #include <cstdio>
3
4  int main () {
5      char a[10], b[10], *p, *combine(char*, char*);
6      strcpy(a, "horse");
7      strcpy(b, "fly");
8      p = combine(a, b);
9      std::puts(p);
10 }
11
12 char* combine (char* s, char* t) {
13     char r[100];
14     strcpy(r, s);
15     return r;
16 }
```

```
<source>:15:5: warning:
returning a Pointer with
points-to set (r) where
points-to set ((*s), (*t),
(null)) is expected [-
Wlifetime]
    return r;
    ^
1 warning generated.
Compiler returned: 0
```

"This is taken (and modified) from the very famous "Mastering C Pointers" code that was shown in the following tweet from John Regehr, but had been discussed by (IIRC) Brian Kernighan somewhere around 2012, but recently received some sleuthing to discover its origins and who wrote it. https://twitter.com/johnregehr/status/1006372519769616386 "

33

Demos

34

Demos


Matthias Gehre (Silexica)

Gábor Horváth (Microsoft, Eötvös Loránd University)

Thursday, September 27 • 14:00 - 15:00

Implementing the C++ Core Guidelines' Lifetime Safety Profile in Clang

35

Checklist:

- When proposing a new C++ feature, ask:
  - How does this let the programmer **express intent** more directly?
  - How does this **simplify** C++ code (read, write, debug, maintain)? ✓
  - How does this **remove** gotchas / special cases / classes of errors? ✓
  - How does this **remove** a place where C++ doesn't currently meet one of its key principles, such as zero-overhead or deterministic control?
  - How does this help potentially **deprecate** something from the language? (*cough* macros)

36

Roadmap: 3 "talks"

- Context: More powerful *and* simpler

- Update: **Lifetime** (simplifying dangling problems, from CppCon 2015)

- Update: **Metaclasses** (simplifying class authoring, from CppCon 2017)

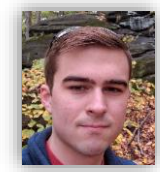
37

Metaclasses update

- Goal: Validate approach
  - Structurally the same as CppCon 2017
  - Progress: SG7 feedback to change "in-place" style to separate "read-only + generate-target"
  - Progress: SG7 feedback on syntactic changes for applying metaclasses
  - Progress: Refined injection model (switch to value-based reflection in progress)


Andrew Sutton (U. Akron, Lock3)


Jennifer Yao (U. Akron, Microsoft)


Wyatt Childers (Lock3)

- Goal: Extend prototype to handle more examples
  - Progress: More of the examples in the paper can now be written
  - Progress: Can now handle parameter lists ⇒ writing wrapper functions

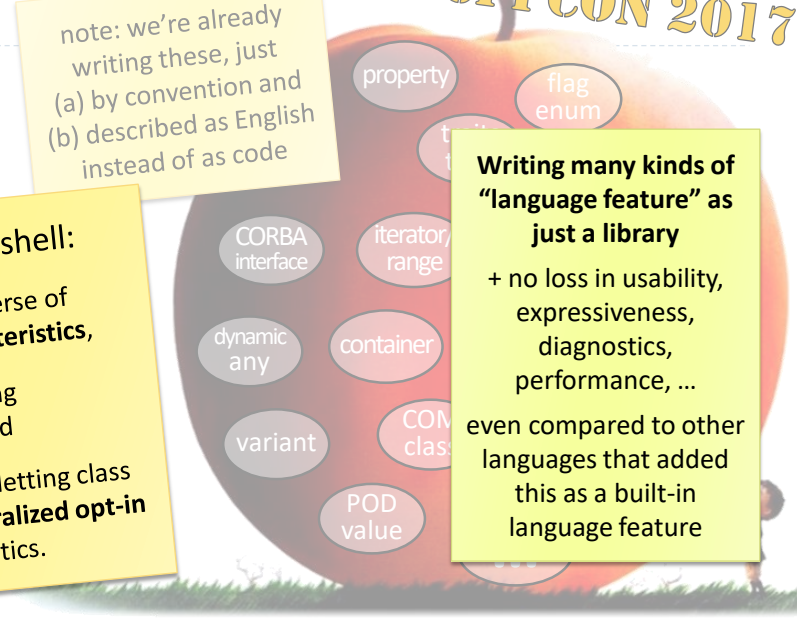
38

Now Playing

- The C++ type system is unified!

note: we're already writing these, just (a) by convention and (b) described as English instead of as code

FROM CPPCON 2017



Writing many kinds of "language feature" as just a library

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

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

Metaclasses goal in a nutshell:

to **name a subset** of the universe of classes having **common characteristics**,

express that subset using **compile-time code**, and

make **classes easier to write** by letting class authors **use the name as a generalized opt-in** to get those characteristics.

New since last year: Defining a metaclass

P0707 R0

```
$class interface {
  ~interface() noexcept { }
  constexpr {
    compiler.require($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();
    }
  }
};
```

P0707 R4

```
constexpr void interface(meta::type source) {
  __generate __fragment struct X
    { virtual ~X() noexcept {} };
  compiler.require(source.variables().empty(),
    "interfaces may not contain data members");
  for (auto f : source.functions()) {
    compiler.require(!f.is_copy() && !f.is_move(),
      "interfaces may not copy or move; "
      "consider a virtual clone()");
    if (!f.has_default_access()) f.make_public();
    compiler.require(f.is_public(),
      "interface functions must be public");
    f.make_pure_virtual();
    __generate f;
  }
}
```

40

New since last year: Applying a metaclass

P0707 R0

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

P0707 R4

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

41

Example

Associating data with locks: **guarded**

But first, how we teach it today…

42

Associate data with locks, take 1 (good)

- Observation: If we knew which mutex covered which data, we could diagnose "oops, forgot to lock" and "oops, took the wrong lock."
- A manual discipline to group data with its mutex:

```
struct MyData {
   vector<int>& v ()       { assert( m_.is_held() ); return v_; }
   Widget*&     w()        { assert( m_.is_held() ); return w_; }
   void         lock()     { m_.lock(); }
   bool         try_lock() { return m_.try_lock(); }
   void         unlock()   { m_.unlock(); }
private:
   vector<int>  v_;
   Widget*      w_;
   mutex_type   m_;
};
```

  - Reasonable migration from existing source (perhaps just add "()").
  - Repetitive: It would be nice to automate the boring parts…

43

Step 1: A testable mutex

- Many mutex types (including *std::mutex*) don't provide a way to ask "have I acquired this?"
- A simple wrapper to the rescue:

```
template<typename Mutex>
class TestableMutex {
public:
   void lock()       { m.lock(); id = this_thread::get_id(); }
   void unlock()     { id = thread::id(); m.unlock(); }
   bool try_lock() { bool b = m.try_lock();  if (b) id = this_thread::get_id();  return b; }
   bool is_held()  { return id == this_thread::get_id(); }
private:
   Mutex m;
   atomic<thread::id> id;
}; // for recursive mutexes, can also add a count (better yet, don't use recursive mutexes)
```

44

Step 2: Innocence by association

- Boilerplate (sorry for the macros, but…):

```
#define GUARDED_WITH(MutType) \
   public: void    lock()       { mut_.lock(); } \
   public: bool    try_lock() { return mut_.try_lock(); } \
   public: void    unlock()    { mut_.unlock(); } \
   private:TestableMutex<MutType> mut_;

#define GUARDED_MEMBER(Type,name) \
   public: Type& name()     { assert(mut_.is_held()); return name##_; } \
   private:Type    name##_;
```

- Then we associate data with a mutex more easily:

```
struct MyData {
   GUARDED_WITH( mutex_type );
   GUARDED_MEMBER( vector<int>, v );
   GUARDED_MEMBER( Widget*, w );
};
```

  - **Note: ISO C++ will never add *guarded<M>* as a language extension (too narrow).**

45

Error detection: The sooner, the better

- Now we can find many latent race conditions automatically and deterministically at test time (vast improvement over intermittent timing-dependent customer bugs):

```
MyData data1 = …, data2 = …;
vector<int>* sneaky = nullptr;
data1.v().push_back( 10 );              // error: will assert
data2.w()->ProcessYearEnd();            // error: will assert

{                                       // enter critical section
   lock_guard<MyData> hold( data1 );    // can treat it as a lockable object
   data1.v().push_back( 10 );           // ok, locked
   data2.w()->ProcessYearEnd();         // error: will assert
   sneaky = &data1.v();                 // ok, but avoid doing this
}

sneaky->push_back( 10 );                // error, but won't assert
```

  - Catches both "oops, forgot to lock" and "oops, took the wrong lock."

46

Using a *guarded<M>* metaclass

Today, by hand

```
struct MyData {
  vector<int>& v()  { assert( m_.is_held() );  return v_;  }
  Widget*&     w() { assert( m_.is_held() );  return w_; }
  void         lock()     {  m_.lock();  }
  bool         try_lock()  {  return m_.try_lock();  }
  void         unlock()    {  m_.unlock();  }
private:
  vector<int>  v_;
  Widget*      w_;
  mutex_type   m_;
};
```

Today, with macros

```
struct MyData {
  GUARDED_WITH( mutex_type );
  GUARDED_MEMBER( vector<int>, v );
  GUARDED_MEMBER( Widget*, w );
};
```

P0707R4

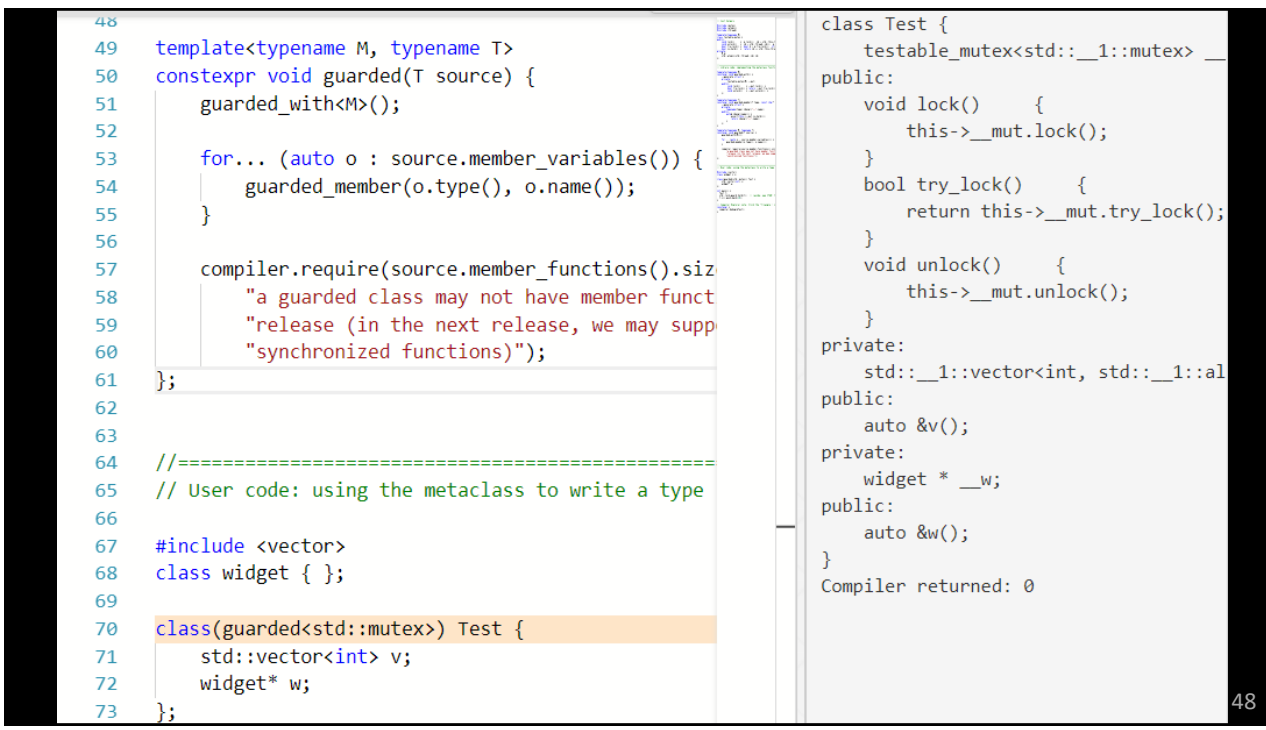
```
class(guarded<mutex_type>) MyData {
   vector<int> v;
   Widget* w;
};
```

Goal: Direct expression of intent

Q: How can we enable the programmer to express their intent in the clearest and most direct way possible?

A: Enable them to write (and utter) a Word of Power

47



48

Using a *guarded<M>* metaclass

Today, by hand

```
struct MyData {
  vector<int>& v()  { assert( m_.is_held() );  return v_;  }
  Widget*&     w() { assert( m_.is_held() );  return w_; }
  void         lock()       { m_.lock();  }
  bool         try_lock()   { return m_.try_lock();  }
  void         unlock()     { m_.unlock();  }
private:
  vector<int>  v_;
  Widget*      w_;
  mutex_type   m_;
};
```

Today, with

```
struct MyData {
  GUARDED_WITH(
  GUARDED_MEMBE
  GUARDED_MEMBE
};
```

P0707R4

```
class(guarded<mutex_type>) MyData {
   vector<int> v;
   Widget* w;
};
```

macros? metaclasses!

If we are serious about getting rid of the remaining reasonable uses of macros, we will love:

modules
constexpr
reflection+generation (metaclasses)

Yes, this is about letting us write more **patterns** as library code

49

Example

Concurrency: **`active`**

But first, how we teach it today…

50

Learn to love threads — on your own terms

- Raw threads are what we have, but are too low-level:
  - Thread mainline has no guard rails: Can be any old undisciplined spaghetti code.
  - Communication is via shared state by default: Fun with mutexes, lock orders, etc.
- What we teach to do by hand (covers common cases, not all uses):
  - **Make the thread mainline a message pump: sequential ⇒ no races among "callee" msgs.**
    - Options: "Thread" can be a coroutine, a series of tasks on a pool, etc. — as-if sequential.
  - **Communicate by sending messages: queued requests, well-formatted (hint: well-typed!), typically with copies of state ⇒ no races between caller and asynchronous callee.**
    - Options: "Queue" can be a priority queue, use multiple channels, etc.
- But "CSP Pattern" is manual: How can we automate these best practices?
  - **Reality: ISO C++ will never add *active* as a language extension (unlike actor languages).**
  - Not directly expressing what we mean makes code harder to write, debug, and maintain.

51

Active objects: Overview

- An active object encapsulates a thread + message_queue.
  - Each object is an asynchronous worker whose mainline is a message pump.
  - Member function calls become async messages (⇒ strongly and statically typed).
- Example of our goal:

```
class worker {                  // (coded specially)
public:
   future<int> func() { … }
};

// in calling code, using an active object
worker w;
auto result = w.func();         // nonblocking
… more work …                   // … concurrent…
use(result.get());              // may block
```

Goal: Direct expression of intent

Covers many classes of concurrency:

Long-running workers
(physics thread, GUI thread, …)

Decoupled independent work
(background save, pipeline stages, …)

Encapsulated resources
(async I/O streams, …)

52

Attach thread lifetime to object lifetime…

- … By attaching the constructor and destructor: ← *Yes, RAII for threads*
  - Constructor: Starts thread and message pump.
  - Destructor: Sends "done" signal, then **blocks** and waits for queue to drain.
- Lets us exploit existing rich language semantics to **control thread lifetimes**:

```
class active1 {
   active2 inner;        // by-value member, by-value lifetime:
   …                     // nested ~~object~~thread bound to enclosing ~~object~~thread
};

void some_function() {
   active1 a;            // stack-based object, stack-based lifetime:
   …                     // ~~object~~thread bound to local scope
} // waits for a and a.inner to complete
```

53

An "Active" helper

This sample shows the basic case: thread + FIFO queue

Variations (pools, channels, …) can be done similarly

- Encapsulates the core mechanics.

```
class Active {
public:
   using Message = function<void()>;
   Active() : thd([=]{ while (!done) mq.receive()(); }) { } // mainline: dispatch loop
   ~Active() { Send([=]{ done = true; }); thd.join(); }     // wait for queue to drain
   void Send( Message m ) { mq.send(m); }                   // enqueue a message
private:
   bool done = false;
   message_queue< Message > mq;
   thread thd;
};
```

54

Pop Quiz: What's the difference?

Mutex locks

```
class log {
   fstream f;
   mutex m;
public:
   void println( /*…*/ ) {
      lock_guard<mutex> hold(m);
      f << /*…*/ << endl;
    }
};
```

Active objects

```
class log {
   fstream f;
   Active a;
public:
   void println( /*…*/ ) { a.Send( [=] {
      f << /*…*/ << endl;
   } ); }

};
```

classic "space vs. time" (concurrency)

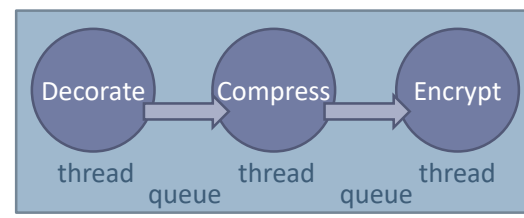
Same calling code either way: `mylog.println( "Hello %1%", name );`

55

Group exercise (1 of 2)

- Implement a concurrent version of the following sequential code:

```
void SendPackets( Buffers& bufs ) {
   for( auto& b : bufs ) {
      Decorate( b );
      Compress( b );
      Encrypt( b );
   }
}
```



  - Assume that:
    - Decorate(x) must end before Decorate(x+1) begins, same for Compress and Encrypt.
    - Decorate(x), Compress(x), and Encrypt(x) must execute in that order.
    - Decorate(x), Compress(y), and Encrypt(z) have no side effects w.r.t. each other and can run concurrently.

56

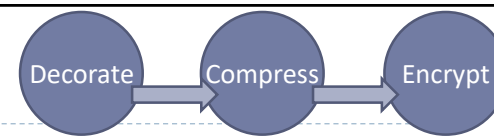
Pipeline stage

- Each stage does just one part.

```
class Stage {
public:
   Stage( function<void(Buffer*)> w ) : work{w} { }
   void Process( Buffer* b ) { a.Send( [=] {
       work( b );
   } ); }
private:
   function<void(Buffer*)> work;
   Active a;                          // NB: remember to put this member last!
};
```

57

Setting up the pipeline



- Three concurrent stages ("communicating sequential processes," anyone?):

```
void SendPackets( Buffers& bufs ) {
   Stage encryptor   ( []   (Buffer* b) { Encrypt(b); } );
   Stage compressor( [&](Buffer* b) { Compress(b);
                                      encryptor.Process(b); } );
   Stage decorator   ( [&](Buffer* b) { Decorate(b);
                                      compressor.Process(b); } );
   for( auto& b : bufs ) {
      decorator.Process( &b );
   }
} // automatically blocks waiting for pipeline to finish
```

Q: How is the pipeline destroyed?

Be specific.

58

Setting up the pipeline (Java 1..6)

- Pre-2014 Java style: **Do you see the problem?**

```
public void SendPackets( Buffers bufs ) {
   Stage encryptor = null;
   Stage compressor = null;
   Stage decorator = null;
   try {
      encryptor = new Stage( new EncryptRunnable() );
      compressor = new Stage( new CompressRunnable( encryptor ) );
      decorator = new Stage( new DecorateRunnable( compressor ) );
      for( b : bufs ) {
         decorator.Process( b );
      }
   } finally {
      if( encryptor != null )    encryptor.dispose();    // automatically block
      if( compressor != null )   compressor.dispose();   // waiting for the
      if( decorator != null )    decorator.dispose();    // pipeline to finish
   }
}
```

59

Setting up the pipeline (Java 1..6)

- Pre-2014 Java style: **Do you see the problem?**

```
public void SendPackets( Buffers bufs ) {
   Stage encryptor = null;
   Stage compressor = null;
   Stage decorator = null;
   try {
      encryptor = new Stage( new EncryptRunnable() );
      compressor = new Stage( new CompressRunnable( encryptor ) );
      decorator = new Stage( new DecorateRunnable( compressor ) );
      for( b : bufs ) {
         decorator.Process( b );
      }
   } finally {
      if( encryptor != null )    encryptor.dispose();    // automatically block
      if( compressor != null )   compressor.dispose();   // waiting for the
      if( decorator != null )    decorator.dispose();    // pipeline to finish
   }
}
```

60

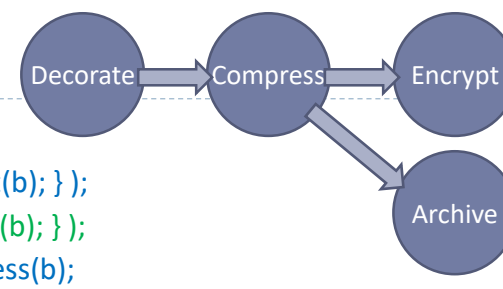
Setting up the pipeline (Java 1..6)

- Pre-2014 Java style: Corrected (but still manual)

```
public void SendPackets( Buffers bufs ) {
   Stage encryptor = null;
   Stage compressor = null;
   Stage decorator = null;
   try {
      encryptor = new Stage( new EncryptRunnable() );
      compressor = new Stage( new CompressRunnable( encryptor ) );
      decorator = new Stage( new DecorateRunnable( compressor ) );
      for( b : bufs ) {
         decorator.Process( b );
      }
   } finally {
      if( decorator != null )    decorator.dispose();     // automatically block
      if( compressor != null )   compressor.dispose();    // waiting for the
      if( encryptor != null )    encryptor.dispose();     // pipeline to finish
   }
}
```

61

And more flexibility…




```
void SendPackets( Buffers& bufs ) {
   Stage encryptor    ( [] (Buffer* b) { Encrypt(b); } );
   Stage archiver     ( [] (Buffer* b) { Archive(b); } );
   Stage compressor ( [&](Buffer* b) { Compress(b);
                                       if (b->something()) encryptor.Process(b);
                                       else archiver.Process(b);
                                     } );
   Stage decorator    ( [&](Buffer* b) { Decorate(b);
                                       compressor.Process(b); );
                                     }
   for( auto b : bufs ) {
      decorator.Process( &b );
   }
} // automatically blocks waiting for pipeline to finish
```

62

Using an *active* metaclass

Today, by hand

```
class A {
public:
   Stage( function<void(Buffer*)> w )
      : work{w} { }
   void Process( Buffer* b ) { a.Send( [=] {
      work( b );
   } ); }
private:
   function<void(Buffer*)> work;
   Active a;        // remember to put this last!
};
```

P0707R4

```
class(active) Stage {
public:
   Stage( function<void(Buffer*)> w )
      : work{w} { }
   void Process( Buffer* b ) {
      work( b );
   }
private:
   function<void(Buffer*)> work;
};
```

63

Using an *active* metaclass (2)

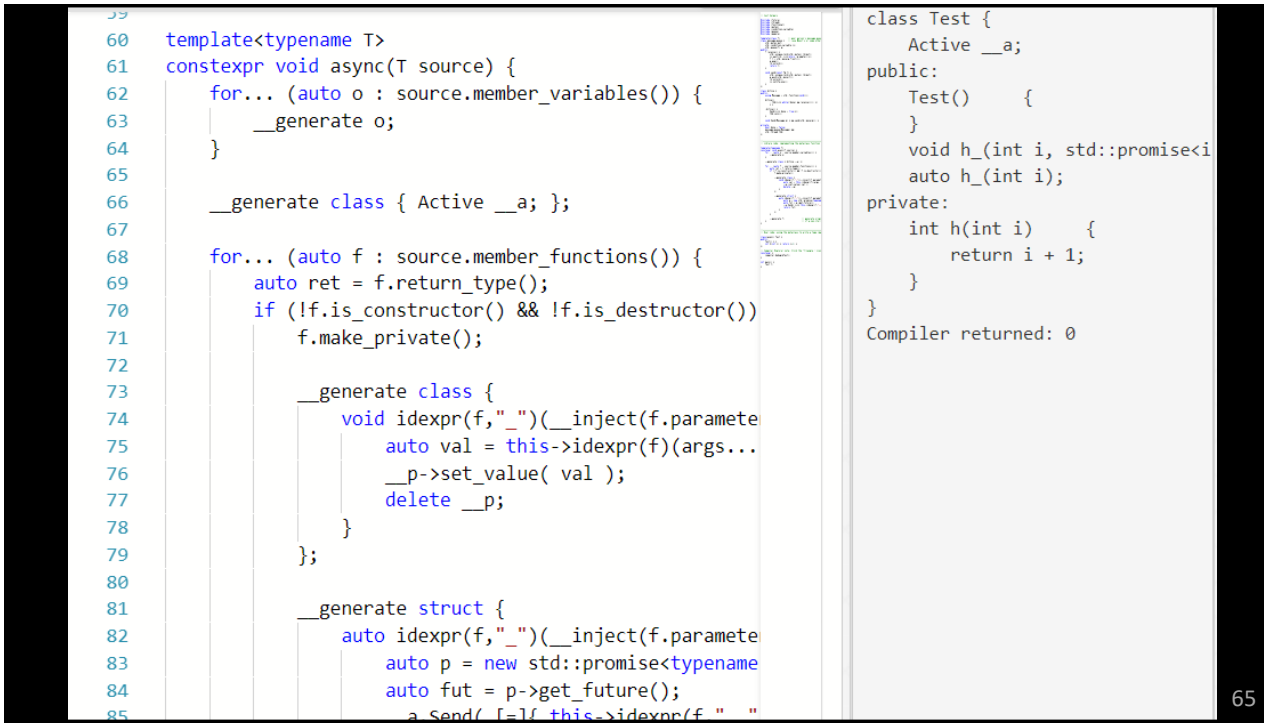
Today, by hand

```
class log {
   fstream f;
   Active a;        // remember to put this last!
public:
   void println( /*...*/ ) { a.Send( [=] {
      f << /*...*/ << endl;
   } ); }
};
```

P0707R4

```
class(active) log {
   fstream f;
public:
   void println( /*...*/ )
      f << /*...*/ << endl;
   }
};
```

64



Using an *active* metaclass (3)

Today, by hand

```
class some_service {
  data stuff;
  Active a;          // remember to put this last!
public:
  future<double> GetResult() {
    promise<double> p;
    future<double> ret = p.get_future();
    a.Send( [p = move(p)]{
      this->DoGetResult( move(p) ); } );
    return ret;
  }
private:
  void DoGetResult( promise<double>&& p ) {
    p.set_value( result );
  }
};
```

P0707R4

```
class(active) some_service {
  data stuff;
public:
  double GetResult() {
    return result;
  }
};
```

Pop quiz:
What's this?

A: "Thread local storage"
But by construction &
with ordinary allocation

66

Example

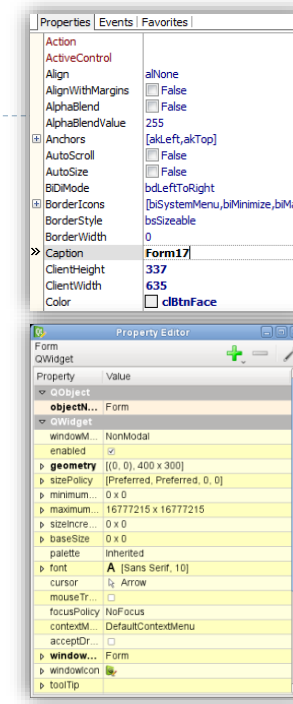
A multiply-rejected C++ language proposal: `property<T>`

But first, how we do it today (nonstandard)…

67

Some history

- Invented many times
  - As nonstandard C++ extensions: Qt, MSVC, Clang, …
  - In many languages: ActionScript 3, C#, D, Delphi, F#, JavaScript, Lua, Objective-C, PHP, Python, Scala, Swift, Ruby, Visual Basic, …
- Useful in practice, and generally liked
  - Convenience of accessing a data member, including from an IDE for design-time defaults
  - Safety of encapsulation: Access control via functions
- Proposed for standardization many times… but failed
  - Benefit debated: "Is it really desirable?" *[Quick A: See lists above]*
  - Cost disputed: "Is it worth the added language complexity?"
  - **Reality: ISO C++ will never add *property T* as a language extension**



Using a *property<T>* metaclass: Defaulted

Today, C#

```
class Item {
   public string Name { get; set; }
}
```

P0707R4

```
class Item {
public:
   class(property<string>) { } Name;
};
```

69

Using a *property<T>* metaclass: Customized

Today, C#

```
class Item {
   private int month;
   public int Month {
      get { return month; }
      set { if ((value>0) && (value<13)) month = value; }
   }
}
```

P0707R4

```
class Item {
   class(property<int>) {
      int month;
      auto get() { return month; }
      void set(int m) { if ((m>0) && (m<13)) month = m; }
   } Month;
};
```

Today, C++ & Qt

```
class Item /*…*/ { /*…*/
public:
   Q_PROPERTY(int Month MEMBER m_month READ getMonth WRITE setMonth)
private:
   int m_month;
   int getMonth() { return m_month; }
   void setMonth(int m) { if ((m>0) && (m<13)) m_month = m; }
};
```

70

```
template<typename Value, typename T>
constexpr void property(T source) {
    // Retain all the declared data members and functions
    bool hasValue = false;
    for... (auto o : source.variables()) {
        if (std::strcmp(o.name(),"value")==0) hasValue = true;
        __generate o;
    }

    bool hasGet = false, hasSet = false;
    for... (auto f : source.member_functions()) {
        if (std::strcmp(f.name(),"get")==0) hasGet = true;
        if (std::strcmp(f.name(),"set")==0) hasSet = true;
        __generate f;
    }

    // If there are no data members or get() function,
    // generate a 'value' data member (this formulation is to
    // avoid injecting an unwanted 'value' member for a
    // property like "area { auto get() { return width*height; }
    // that needs no data members of its own)
    if (!hasGet && source.member_variables().size() == 0) {
        __generate __fragment class {
            Value value;
        };
        hasValue = true;
    }
```

```
public:
class W {
    int value;
    int get() const     {
        return this->value;
    }
    void set(const int &v)     {
        this->value = v;
    }
    void set(int &&v)     {
        this->value = std::move(v);
    }
public:
    W();
    operator int() const     {
        return this->get();
    }
    W(const int &v)     {
        this->set(v);
    }
    W(int &&v)     {
        this->set(std::move(v));
    }
    X::W &operator=(const int &v)
        this->set(v);
        return *this;
    }
    X::W &operator=(int &&v)     {
```

```
    // If there is no get function, generate one that accesses '
    if (!hasGet) {
        if (hasValue)
            __generate __fragment class {
                Value get() const { return this->value; }
            };
        else compiler.error(
            "a data member named 'value' is required to generate
            "function; either provide a data member named 'value
            "write a custom get() function");
    }

    // If T is not const and there is no set function,
    // generate one that accesses 'value'
    if (!std::is_const_v<T> && !hasSet && hasValue) {
        __generate __fragment class {
            void set(const Value& v) { this->value = v; }
            void set(Value&& v) { this->value = std::move(v); }
        };
        hasSet = true;
    }

    // Provide a conversion operator that uses the get() functic
    // whether it was user-supplied or default-generated above
    __generate __fragment struct X {
        X() = default;
        operator Value() const { return this->get(); }
```

```
        return *this;
    }
} width;

[Truncated]
Compiler returned: -1
```

```
60      __generate __fragment struct X {
61          X() = default;
62          operator Value() const { return this->get(); }
63      };
64
65      // If there is a set() function, user-supplied or default-ge
66      // provide a constructor and assignment operator that uses s
67      if (hasSet)
68          __generate __fragment struct X {
69              X(const Value& v) { this->set(v); }
70              X(Value&& v) { this->set(std::move(v)); }
71              X& operator=(const Value& v) { this->set(v); return
72              X& operator=(Value&& v) { this->set(std::move(v)); r
73          };
74  };
75
76
77  //========================================================
78  // User code: using the metaclass to write a type (many times)
79
80  class X {
81  public:
82      class(property<int>) W { } width;
83
84      // Aiming for this:
85      //      property<int> { } width;
86  };
```

```
Compiler returned: -1
```

Goals

FROM CPPCON 2017

- 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

74

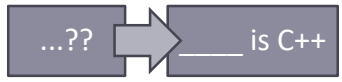
Checklist:

- When proposing a new C++ feature, ask:
  - How does this let the programmer **express intent** more directly? ✓
  - How does this **simplify** C++ code (read, write, debug, maintain)? ✓
  - How does this **remove** gotchas / special cases / classes of errors? ✓
  - How does this **remove** a place where C++ doesn't currently meet one of its key principles, such as zero-overhead or deterministic control?
  - How does this help potentially **deprecate** something from the language? (*cough* macros) ✓

75

Roadmap: 3 "talks"

- Context: More powerful *and* simpler

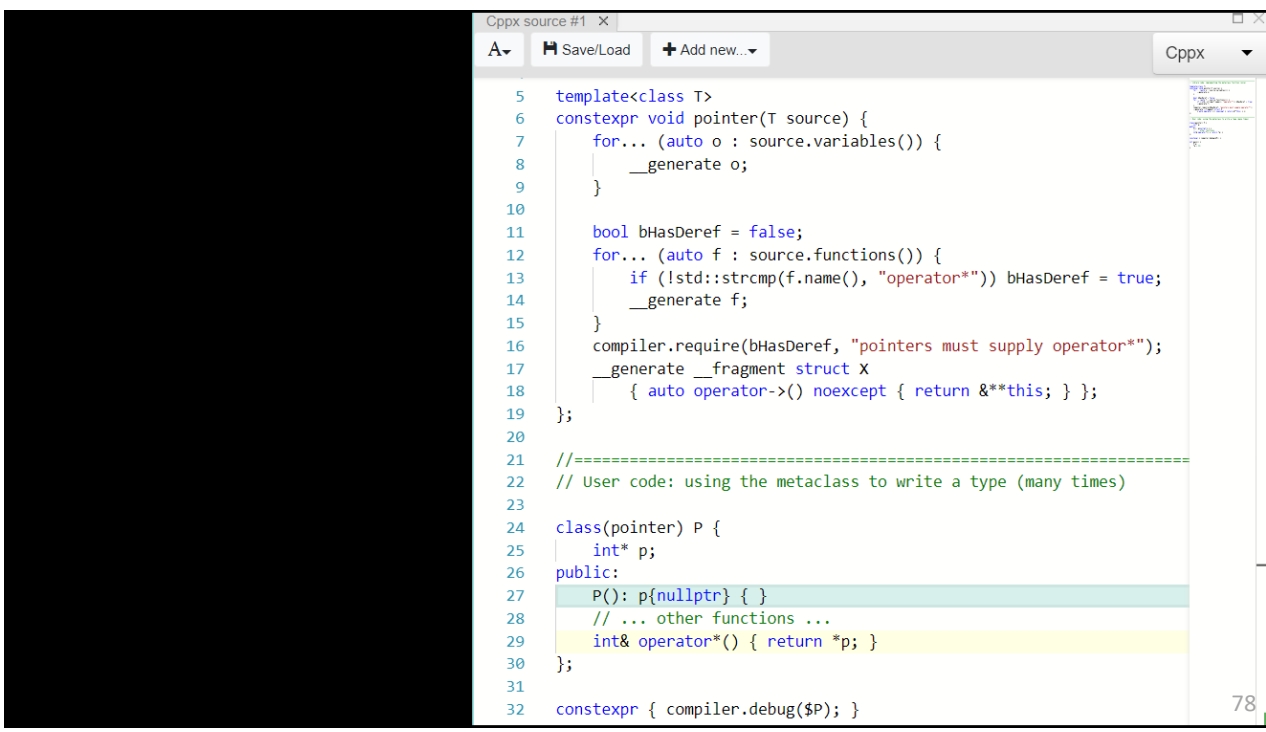


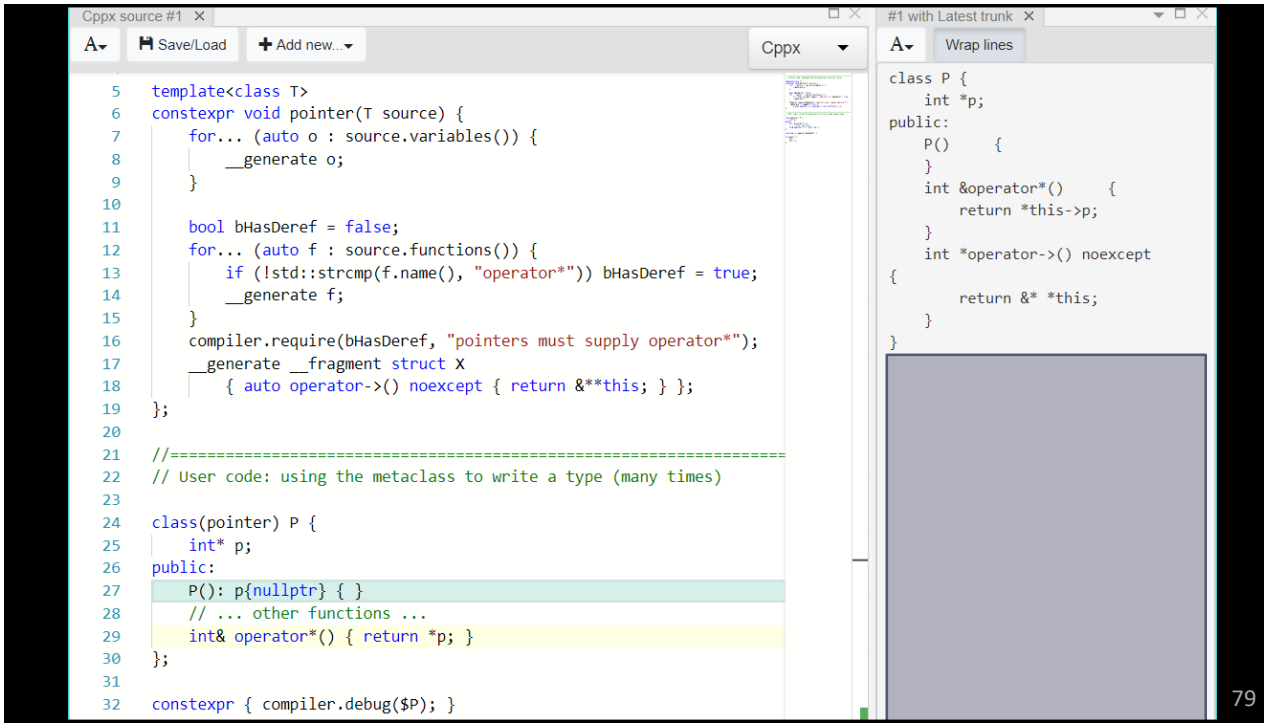
- Update: **Lifetime** (simplifying dangling problems, from CppCon 2015)

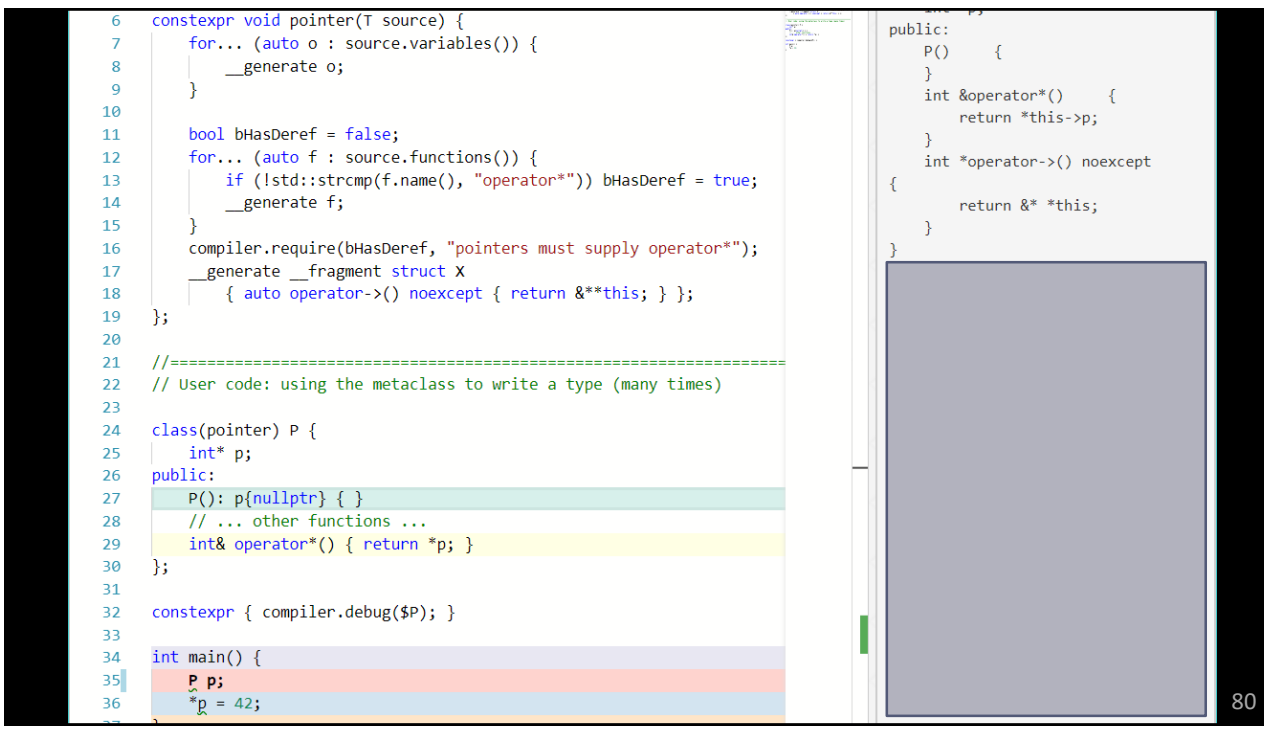
- Update: **Metaclasses** (simplifying class authoring, from CppCon 2017)

76


MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
I HATE YOU.
0x3A28213A
0x6339392C,
0x7363682E.
77


Cppx source #1
Save/Load
Add new...
Cppx
template<class T>
constexpr void pointer(T source) {
for... (auto o : source.variables()) {
__generate o;
}
bool bHasDeref = false;
for... (auto f : source.functions()) {
if (!std::strcmp(f.name(), "operator*")) bHasDeref = true;
__generate f;
}
compiler.require(bHasDeref, "pointers must supply operator*");
__generate __fragment struct X
{ auto operator->() noexcept { return &**this; } };
};
//=========
// User code: using the metaclass to write a type (many times)
class(pointer) P {
int* p;
public:
P(): p{nullptr} { }
// ... other functions ...
int& operator*() { return *p; }
};
constexpr { compiler.debug($P); }
78


Cppx source #1
Save/Load
Add new...
Cppx
#1 with Latest trunk
Wrap lines
template<class T>
constexpr void pointer(T source) {
for... (auto o : source.variables()) {
__generate o;
}
bool bHasDeref = false;
for... (auto f : source.functions()) {
if (!std::strcmp(f.name(), "operator*")) bHasDeref = true;
__generate f;
}
compiler.require(bHasDeref, "pointers must supply operator*");
__generate __fragment struct X
{ auto operator->() noexcept { return &**this; } };
};
//===========
// User code: using the metaclass to write a type (many times)
class(pointer) P {
int* p;
public:
P(): p{nullptr} { }
// ... other functions ...
int& operator*() { return *p; }
};
constexpr { compiler.debug($P); }
class P {
int *p;
public:
P() {
}
int &operator*() {
return *this->p;
}
int *operator->() noexcept
{
return &* *this;
}
}
79

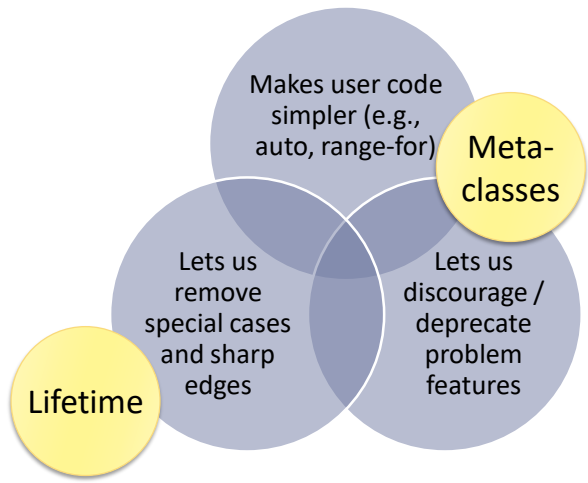

constexpr void pointer(T source) {
for... (auto o : source.variables()) {
__generate o;
}
bool bHasDeref = false;
for... (auto f : source.functions()) {
if (!std::strcmp(f.name(), "operator*")) bHasDeref = true;
__generate f;
}
compiler.require(bHasDeref, "pointers must supply operator*");
__generate __fragment struct X
{ auto operator->() noexcept { return &**this; } };
};
//===========
// User code: using the metaclass to write a type (many times)
class(pointer) P {
int* p;
public:
P(): p{nullptr} { }
// ... other functions ...
int& operator*() { return *p; }
};
constexpr { compiler.debug($P); }
int main() {
P p;
*p = 42;
public:
P() {
}
int &operator*() {
return *this->p;
}
int *operator->() noexcept
{
return &* *this;
}
}
80

```
6   constexpr void pointer(T source) {
7       for... (auto o : source.variables()) {
8           __generate o;
9       }
10
11      bool bHasDeref = false;
12      for... (auto f : source.functions()) {
13          if (!std::strcmp(f.name(), "operator*")) bHasDeref = true;
14          __generate f;
15      }
16      compiler.require(bHasDeref, "pointers must supply operator*");
17      __generate __fragment struct X
18          { auto operator->() noexcept { return &**this; } };
19  };
20
21  //=============================================================
22  // User code: using the metaclass to write a type (many times)
23
24  class(pointer) P {
25      int* p;
26  public:
27      P(): p{nullptr} { }
28      // ... other functions ...
29      int& operator*() { return *p; }
30  };
31
32  constexpr { compiler.debug($P); }
33
34  int main() {
35      P p;
36      *p = 42;
```

```
public:
    P()     {
    }
    int &operator*()     {
        return *this->p;
    }
    int *operator->() noexcept
{
        return &* *this;
    }
}
<source>:36:6: warning: passing a
null pointer as argument to a non-
null parameter [-Wlifetime-null]
    *p = 42;
     ^
<source>:35:5: note: default-
constructed Pointers are assumed to
be null
    P p;
    ^
1 warning generated.
Compiler returned: 0
```

81

Let's try pursue evolution that…



Questions?