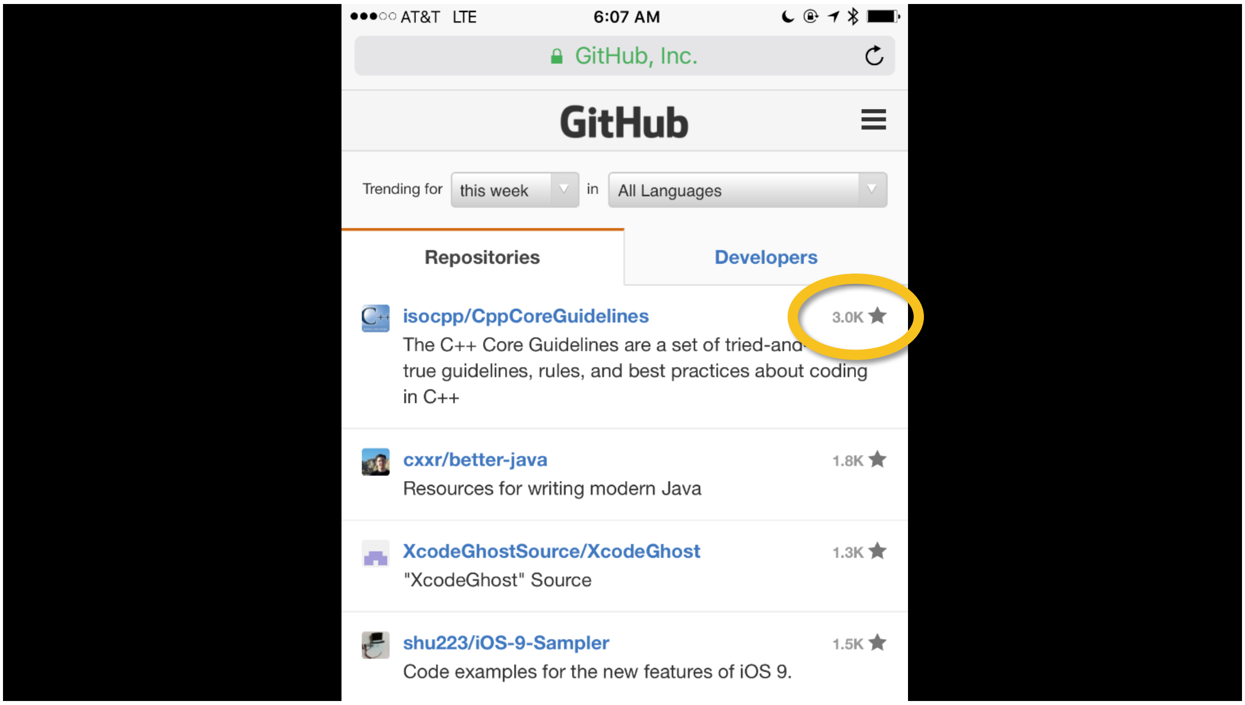
Writing Good C++14… *By Default*

Herb Sutter

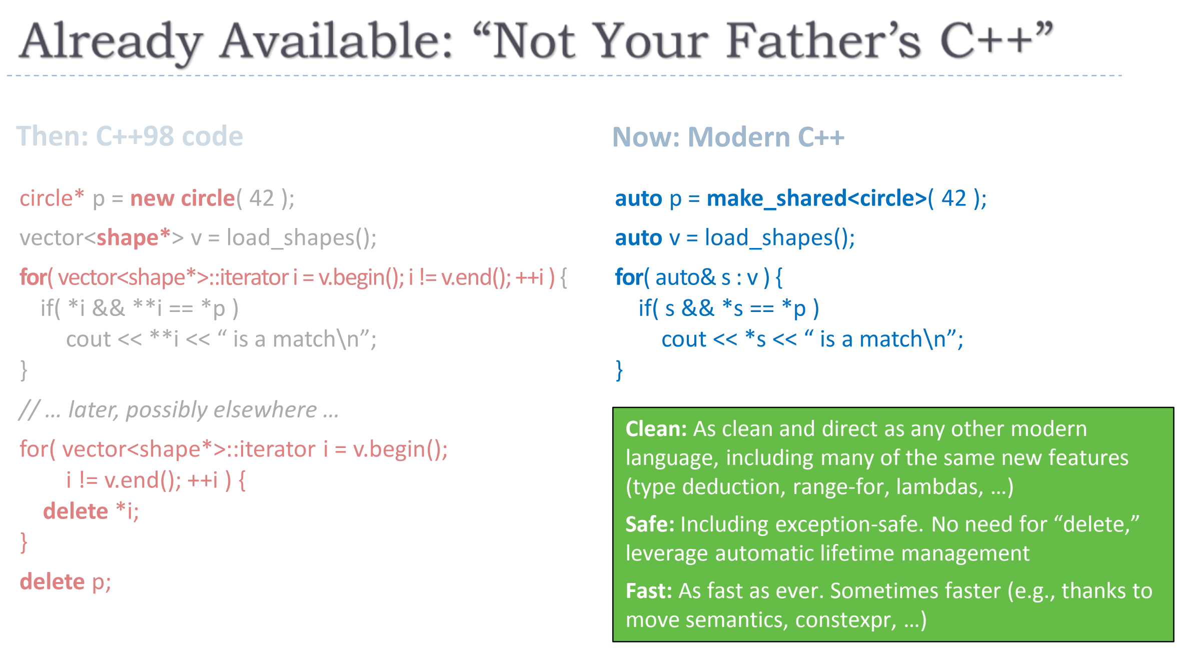


isocpp/CppCoreGuidelines 3.0K ★
The C++ Core Guidelines are a set of tried-and-true guidelines, rules, and best practices about coding in C++

cxxr/better-java 1.8K ★
Resources for writing modern Java

XcodeGhostSource/XcodeGhost 1.3K ★
"XcodeGhost" Source

shu223/iOS-9-Sampler 1.5K ★
Code examples for the new features of iOS 9.



Already Available: “Not Your Father’s 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;
```

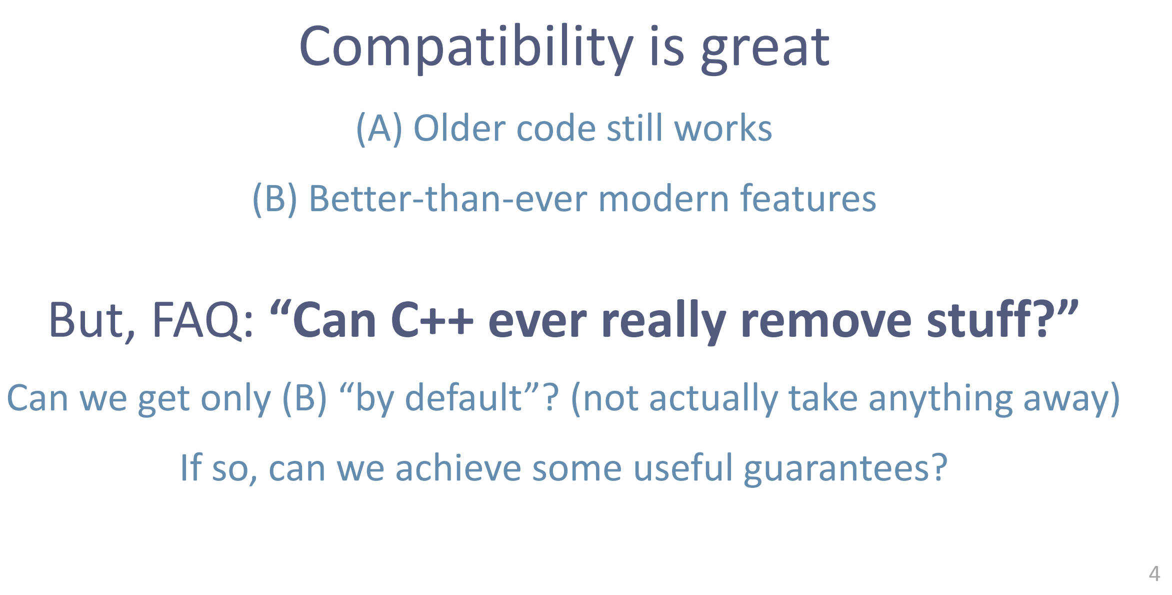
Now: Modern C++

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

Clean: As clean and direct as any other modern language, including many of the same new features (type deduction, range-for, lambdas, …)

Safe: Including exception-safe. No need for “delete,” leverage automatic lifetime management

Fast: As fast as ever. Sometimes faster (e.g., thanks to move semantics, constexpr, …)



Compatibility is great

(A) Older code still works

(B) Better-than-ever modern features

But, FAQ: **“Can C++ ever really remove stuff?”**

Can we get only (B) “by default”? (not actually take anything away)

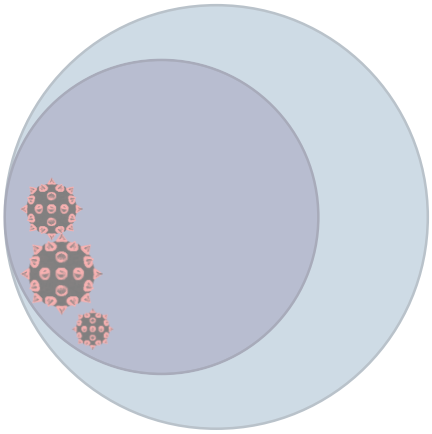
If so, can we achieve some useful guarantees?

4

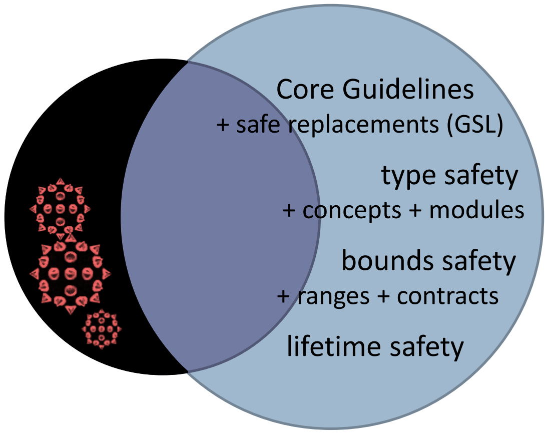
Acknowledgments

- This is the beginning of open source project(s). We need your help.
  - **C++ Core Guidelines** – all about "getting the better parts by default" (*github.com/isocpp*)
  - **Guideline Support Library (GSL)** – first implementation available (*github.com/microsoft/gsl*)
    – portable C++, tested on Clang / GCC / Xcode / MSVC, for (variously) Linux / OS X / Windows
  - **Checker tools** – first implementation next month (MSVC 2015 Upd.1 CTP timeframe)
    – "type" and "bounds" safety profiles (initially Windows binary, intention is to open source)
- Just getting to this starting point is thanks to collaboration and feedback from:
  - Bjarne Stroustrup, myself, Gabriel Dos Reis, Neil MacIntosh, Axel Naumann, Andrew Pardoe, Andrew Sutton, Sergey Zubkov
  - Andrei Alexandrescu, Jonathan Caves, Pavel Curtis, Joe Duffy, Daniel Frampton, Chris Hawblitzel, Shayne Hiet-Block, Peter Juhl, Leif Kornstaedt, Aaron Lahman, Eric Niebler, Gor Nishanov, Jared Parsons, Jim Radigan, Dave Sielaff, Jim Springfield, Jiangang (Jeff) Zhuang, & more…
  - CERN, Microsoft, Morgan Stanley
    - **GSL is derived from production code:** network protocol handlers; kernel Unicode string handlers; graphics routines; OS shell enumerator patterns; cryptographic routines; …

Superset   Superset + Subset



ISO C++98 → C++11 → C++14 → …



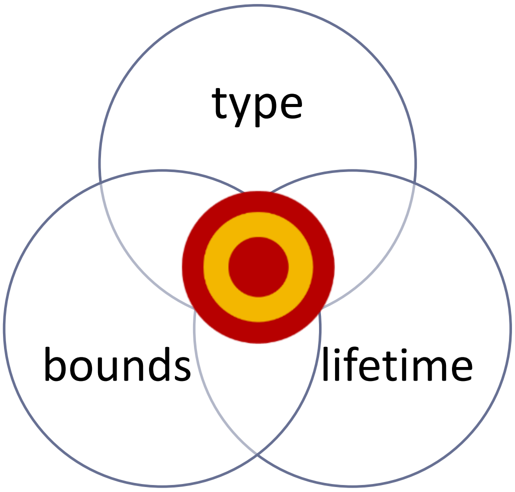
ISO C++ and C++ Core Guidelines

Initial target: Type & memory safety

- Traditional definition
  = type-safe
  + bounds-safe
  + lifetime-safe

- Examples:
  - Type: Avoid unions, use *variant*
  - Bounds: Avoid pointer arithmetic, use *array_view*
  - Lifetime: Don't leak (forget to delete), don't corrupt (double-delete), don't dangle (e.g., return &local)

- Future: Concurrency, security, …



Target guarantee

applies to **recompiled** code

C++ code **compiled in the safe subset**
is **never the root cause** of type/memory safety errors,
except where explicitly annotated as unsafe.

never needs to be **audited** for safety

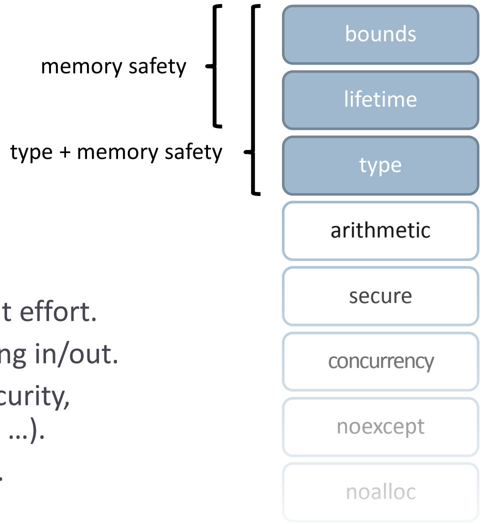
always have **opt-out** to full control

Goal is not to provide verified, whole-program guarantees of safety.

Goal is to enable type and memory **safety by construction**, for as much of **your program code** as possible. This type and memory safety can be **enforced at compile time** via static language subset restrictions **+ at run time** by validation/enforcement (fail-fast, or configurable).

Safety profiles

- A *profile* is:
  - a cohesive set of **deterministic and portable subset rules**
  - designed to achieve a **specific guarantee**



- Benefits of decomposed profiles:
  - Articulates what guarantee you get for what effort.
  - Avoids monolithic “safe/unsafe” when opting in/out.
  - Extensible to future safety profiles (e.g., security, concurrency, arithmetic, noexcept, noalloc, …).
  - Enables incremental development/delivery.

Safety profiles

	type	bounds	lifetime
Goal: Target guarantee	No use of a location as a T that contains an unrelated U		
Superset: New libraries	*byte* *variant<Ts…>*		
Subset: Restrictions	Examples: • No use of uninit variables • No reinterpret_cast • No static_cast downcasts • No access to union mbrs		
Open questions			

Type safety overview

- GSL types
  - *byte*: Raw memory, not char
  - *variant<…Ts>*: Contains one object at a time (“tagged union”)
- Rules

1. Don’t use *reinterpret_cast*.
2. Don’t use *static_cast* downcasts. Use *dynamic_cast* instead.
3. Don’t use *const_cast* to cast away *const* (i.e., at all).
4. Don’t use C-style *(T)expression* casts that would perform a *reinterpret_cast*, *static_cast* downcast, or *const_cast*.
5. Don’t use a local variable before it has been initialized.
6. Always initialize a member variable.
7. Avoid accessing members of raw unions. Prefer *variant* instead.
8. Avoid reading from varargs or passing vararg arguments. Prefer variadic template parameters instead.

(Also: safe math → separate profile)

Safety profiles

	type	**bounds**	**lifetime**
Goal: Target guarantee	No use of a location as a T that contains an unrelated U		
Superset: New libraries	*byte* *variant<Ts…>*		
Subset: Restrictions	Examples: • No use of uninit variables • No reinterpret_cast • No static_cast downcasts • No access to union mbrs		
Open questions	Completing GSL types: • Standardizing *variant<>* • Leave no valid reason to use raw unions + manual discriminant		

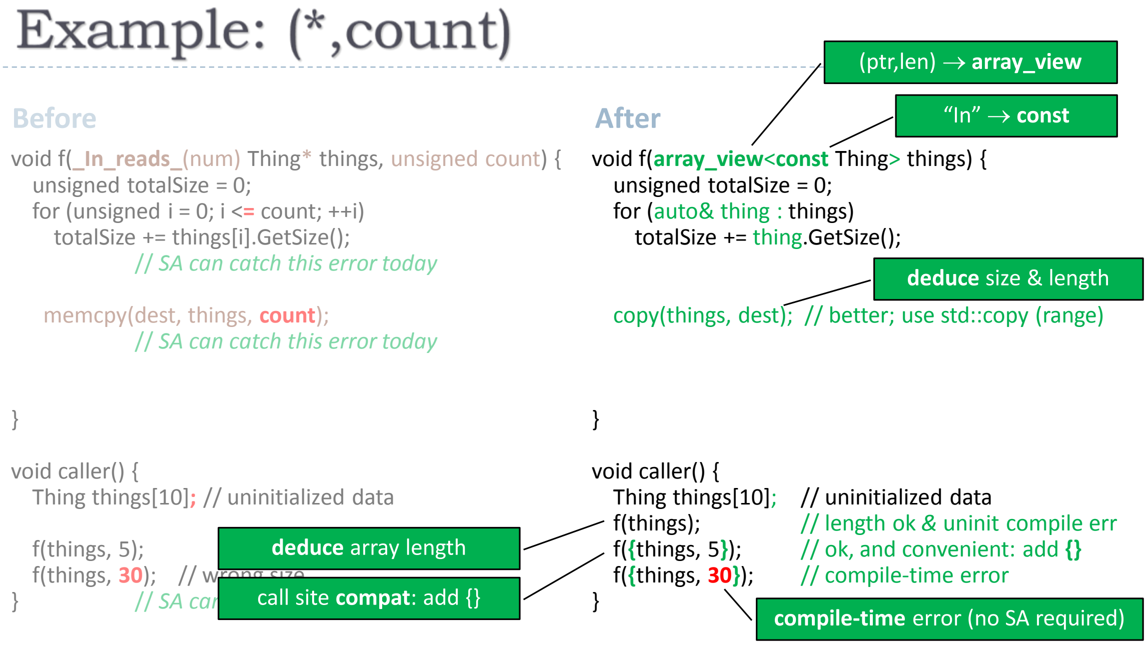
Safety profiles

	type	bounds	lifetime
Goal: Target guarantee	No use of a location as a T that contains an unrelated U	No accesses beyond the bounds of an allocation	
Superset: New libraries	*byte* *variant<Ts…>*	*array_view<>* *string_view<>* ranges	
Subset: Restrictions	Examples: • No use of uninit variables • No reinterpret_cast • No static_cast downcasts • No access to union mbrs	Examples: • No pointer arithmetic • Bounds-safe array access	
Open questions	Completing GSL types: • Standardizing *variant<>* • Leave no valid reason to use raw unions + manual discriminant		

Bounds safety overview

- GSL types
  - *array_view<T,Extents>*: A view of contiguous T objects, **replaces (*,len)**
  - *string_view<CharT,Extent>*: Convenience alias for a 1-D *array_view*
    - Note: *array_view* and *not_null* are the only GSL types with any run-time work

- Rules
  1. Don't use pointer arithmetic. Use *array_view* instead.
  2. Only index into arrays using constant expressions.
  3. Don't use array-to-pointer decay.
  4. Don't use *std::* functions and types that are not bounds-checked.



Example: (*,count)

Before

```
void f(_In_reads_(num) Thing* things, unsigned count) {
  unsigned totalSize = 0;
  for (unsigned i = 0; i <= count; ++i)
    totalSize += things[i].GetSize();
            // SA can catch this error today

   memcpy(dest, things, count);
            // SA can catch this error today


}

void caller() {
  Thing things[10]; // uninitialized data

  f(things, 5);
  f(things, 30);    // wrong size
}           // SA ca
```

After

```
void f(array_view<const Thing> things) {
  unsigned totalSize = 0;
  for (auto& thing : things)
    totalSize += thing.GetSize();


  copy(things, dest);  // better; use std::copy (range)



}

void caller() {
  Thing things[10];    // uninitialized data
  f(things);           // length ok & uninit compile err
  f({things, 5});      // ok, and convenient: add {}
  f({things, 30});     // compile-time error
}
```

(ptr,len) → **array_view**

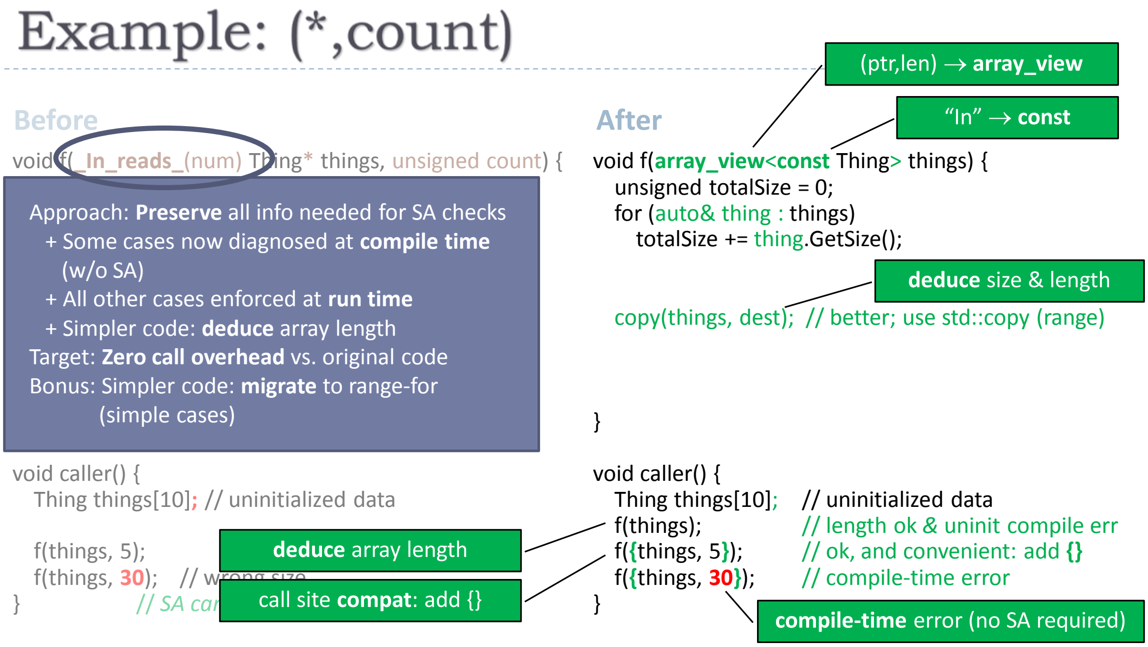
“In” → **const**

deduce size & length

deduce array length

call site **compat**: add {}

compile-time error (no SA required)



Example: (*,count)

Before

```
void f(_In_reads_(num) Thing* things, unsigned count) {
```

Approach: **Preserve** all info needed for SA checks
- + Some cases now diagnosed at **compile time** (w/o SA)
- + All other cases enforced at **run time**
- + Simpler code: **deduce** array length

Target: **Zero call overhead** vs. original code
Bonus: Simpler code: **migrate** to range-for (simple cases)

```
void caller() {
  Thing things[10]; // uninitialized data

  f(things, 5);
  f(things, 30);    // wrong size
}           // SA ca
```

After

```
void f(array_view<const Thing> things) {
  unsigned totalSize = 0;
  for (auto& thing : things)
    totalSize += thing.GetSize();


  copy(things, dest);  // better; use std::copy (range)



}

void caller() {
  Thing things[10];    // uninitialized data
  f(things);           // length ok & uninit compile err
  f({things, 5});      // ok, and convenient: add {}
  f({things, 30});     // compile-time error
}
```

(ptr,len) → **array_view**

“In” → **const**

deduce size & length

deduce array length

call site **compat**: add {}

compile-time error (no SA required)

Applying a profile: Explicit opt-out

- Other languages: *unsafe{…}*
  - Monolithic = all-or-nothing adoption, specification, and delivery

```
unsafe {                    // early strawman
   *(ptr + offset) = 42;
   y = (Y&)(my_x);
   memcpy(somewhere, things, count);
}
```

- This design: *[[suppress(profile)]]* and *[[suppress(rule)]]*
  - On blocks or statements
  - Opt out of a profile, or a specific rule
    - Documents what to audit for
    - Portable C++CG warning suppression
  - *[[attributes]]* ⇒ header compatibility
    - Modern compilers are already required to ignore attributes they don't support

```
[[suppress(bounds)]]{
   *(ptr + offset) = 42;
   memcpy(somewhere, things, count);
}

[[suppress(type.casts)]] y = (Y&)(my_x);
```

17

Example: (*,count)

(ptr,len) → **array_view**

"In" → **const**

Before

```
void f(_In_reads_(num) Thing* things, unsigned count) {
   unsigned totalSize = 0;
   for (unsigned i = 0; i <= count; ++i)
      totalSize += things[i].GetSize();
            // SA can catch this error today

   memcpy(dest, things, count);
            // SA can catch this error today
```

doesn't require a new compiler; implementable in any build tool (compiler, SA, lint, …)

```
void caller() {
   Thing things[10]; // uninitialized data

   f(things, 5);
   f(things, 30);    // wrong size
}           // SA car
```

deduce array length

call site **compat**: add {}

After

```
void f(array_view<const Thing> things) {
   unsigned totalSize = 0;
   for (auto& thing : things)
      totalSize += thing.GetSize();
```

deduce size & length

```
   copy(things, dest);  // better; use std::copy (range)

   [[suppress(bounds)]]
      memcpy(dest, things.data(), things.bytes());
}
```

still get benefits even when calling unsafe code

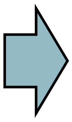
```
                      itialized data
   f(things);           // length ok & uninit compile err
   f({things, 5});      // ok, and convenient: add {}
   f({things, 30});     // compile-time error
}
```

compile-time error (no SA required)

Using types in/with old code

- New types interoperate cleanly with existing code, so you can adopt them incrementally. They also address container diversity.
- All these callers, and all their types… … work with one call target

```
std::vector<int>& vec;   f(vec);

int* p; size_t len;      f({p,len});

std::array<int>& arr;    f(arr);
```

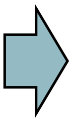


```
void f(array_view<int> av);
```

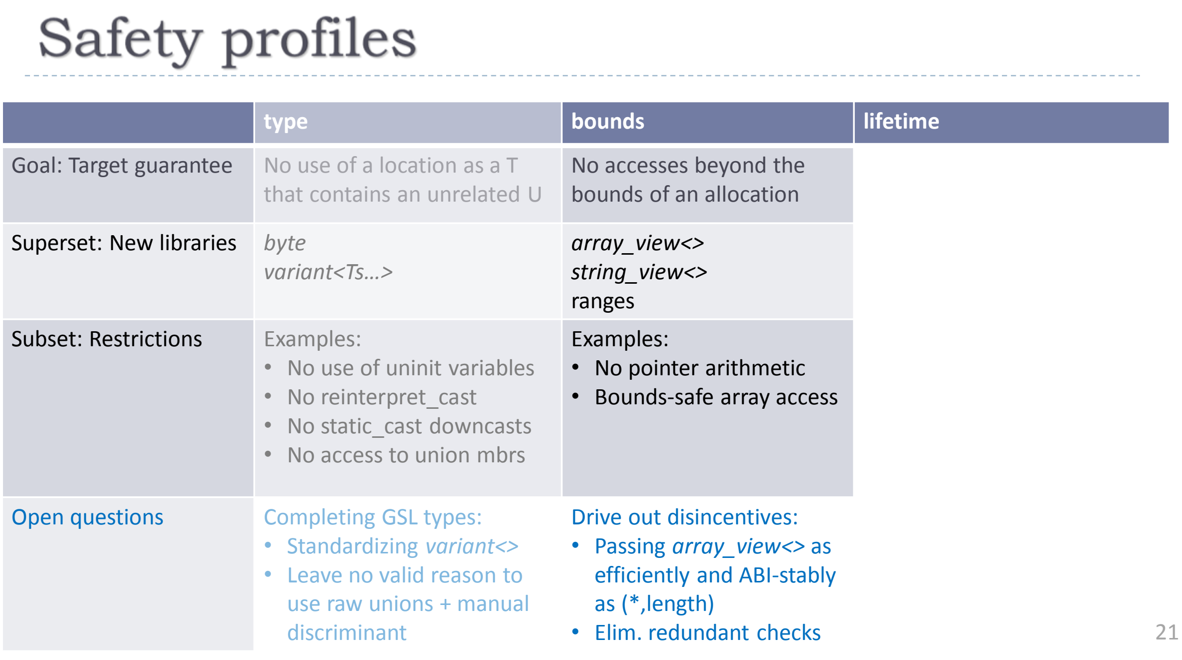
Using types in/with old code

- New types interoperate cleanly with existing code, so you can adopt them incrementally. They also address string diversity.
- All these callers, and all their types… … work with one call target

```
std::wstring& s;         f(s);
wchar_t* s, size_t len;  f({s,len});
QString s;               f(s);
CStringA s;              f(s);
PCWSTR s;                f(s);
BSTR s;                  f(s);
_bstr_t s;               f(s);
UnicodeString s;         f(s);
CComBSTR s;              f(s);
CAtlStringW& s;          f(s);
/* … known incomplete sample … */
```

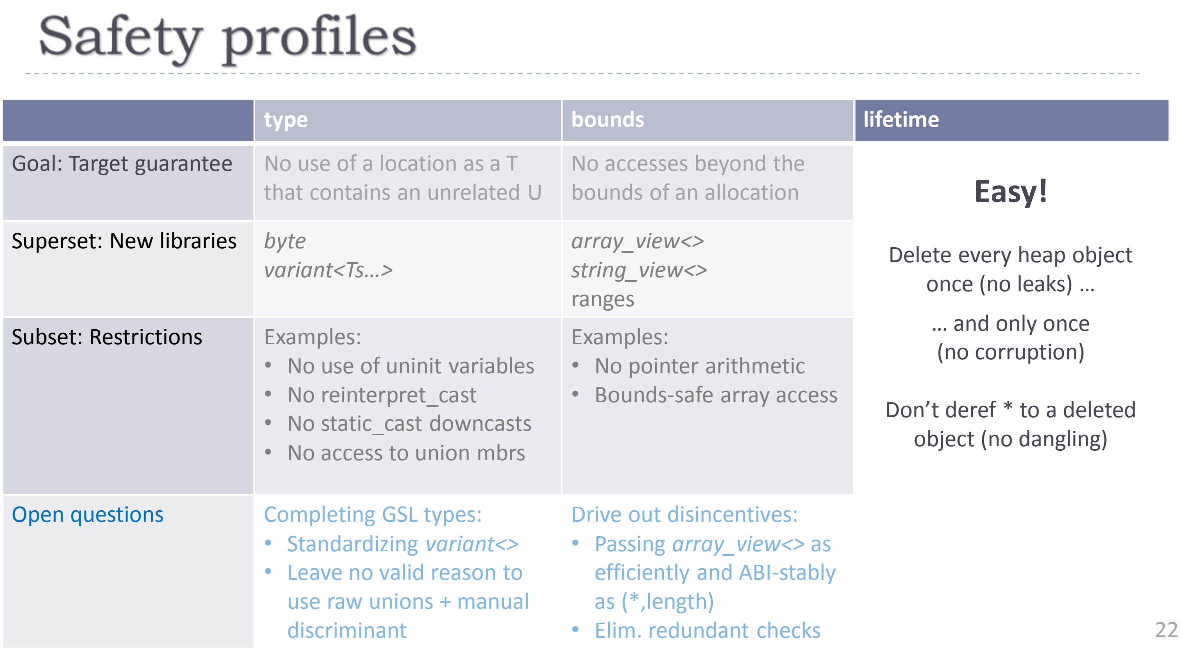


```
void f(wstring_view s);
```



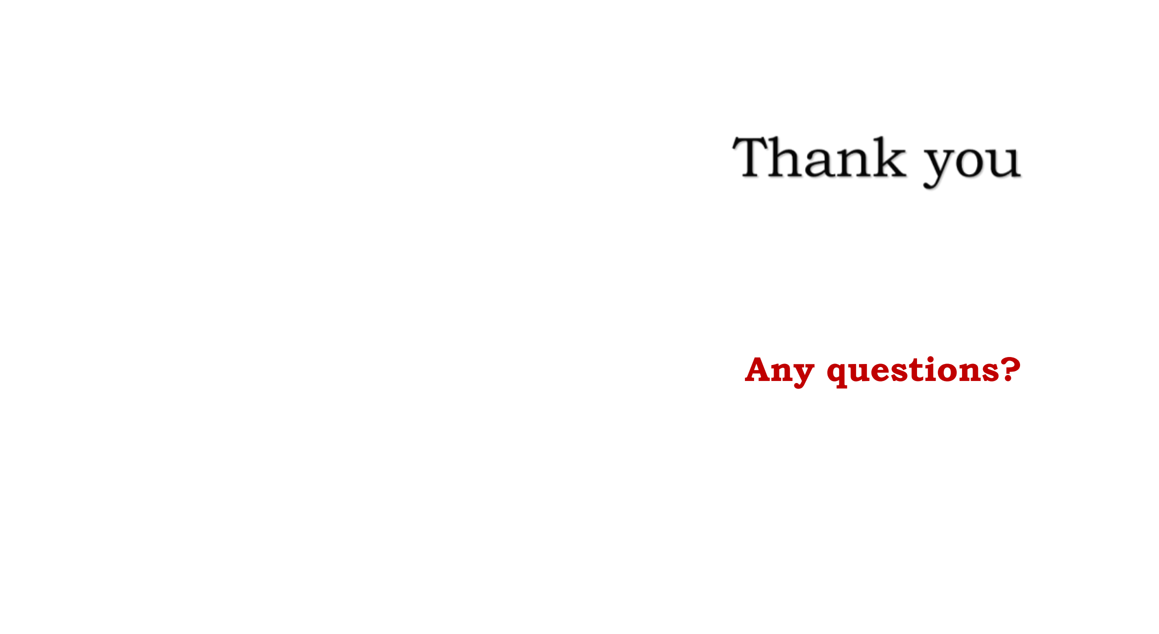
Safety profiles

	type	bounds	lifetime
Goal: Target guarantee	No use of a location as a T that contains an unrelated U	No accesses beyond the bounds of an allocation	
Superset: New libraries	*byte* *variant<Ts…>*	*array_view<>* *string_view<>* ranges	
Subset: Restrictions	Examples: • No use of uninit variables • No reinterpret_cast • No static_cast downcasts • No access to union mbrs	Examples: • No pointer arithmetic • Bounds-safe array access	
Open questions	Completing GSL types: • Standardizing *variant<>* • Leave no valid reason to use raw unions + manual discriminant	Drive out disincentives: • Passing *array_view<>* as efficiently and ABI-stably as (*,length) • Elim. redundant checks	



Safety profiles

	type	bounds	lifetime
Goal: Target guarantee	No use of a location as a T that contains an unrelated U	No accesses beyond the bounds of an allocation	**Easy!** Delete every heap object once (no leaks) … … and only once (no corruption) Don't deref * to a deleted object (no dangling)
Superset: New libraries	*byte* *variant<Ts…>*	*array_view<>* *string_view<>* ranges	
Subset: Restrictions	Examples: • No use of uninit variables • No reinterpret_cast • No static_cast downcasts • No access to union mbrs	Examples: • No pointer arithmetic • Bounds-safe array access	
Open questions	Completing GSL types: • Standardizing *variant<>* • Leave no valid reason to use raw unions + manual discriminant	Drive out disincentives: • Passing *array_view<>* as efficiently and ABI-stably as (*,length) • Elim. redundant checks	



Thank you

Any questions?

Safety profiles

Known hard "40-year" problem

Many wrecks litter this highway

Handle only C because "C is simpler"
or, Incur run-time overheads (e.g., GC)
or, Rely on whole-program analysis
or, Require extensive annotation
or, Invent a new language
or, . . .

We believe we have something conceptually simple

Observation: C++ code is simpler – **C++ source contains more information**
We can leverage C++'s strong **scope** and **ownership** semantics
Special acknowledgments: Bjarne Stroustrup & Neil MacIntosh, + more

lifetime

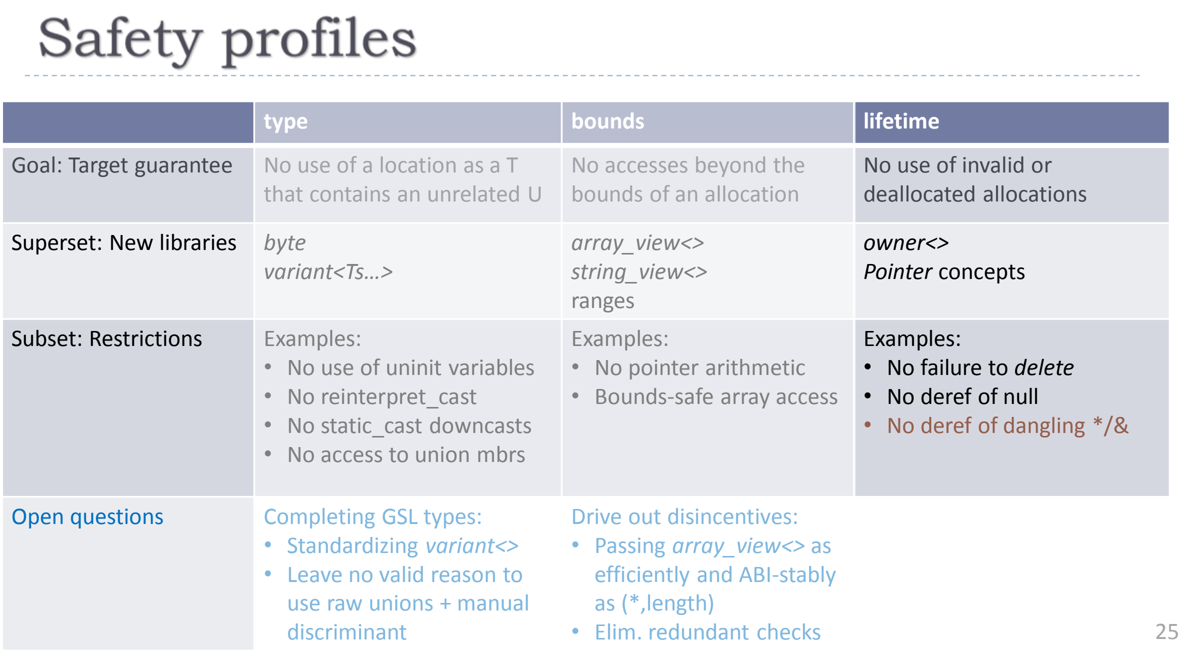
Easy!to state

Delete every heap object once (no leaks) …

… and only once (no corruption)

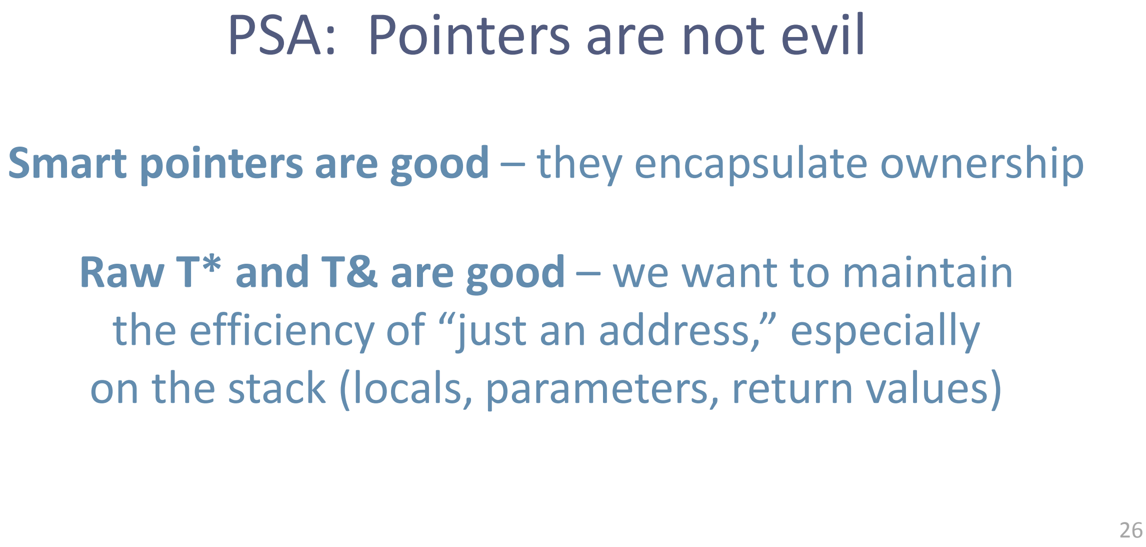
Don't deref * to a deleted object (no dangling)

24



Safety profiles

	type	bounds	lifetime
Goal: Target guarantee	No use of a location as a T that contains an unrelated U	No accesses beyond the bounds of an allocation	No use of invalid or deallocated allocations
Superset: New libraries	*byte* *variant<Ts…>*	*array_view<>* *string_view<>* ranges	*owner<>* *Pointer* concepts
Subset: Restrictions	Examples: • No use of uninit variables • No reinterpret_cast • No static_cast downcasts • No access to union mbrs	Examples: • No pointer arithmetic • Bounds-safe array access	Examples: • No failure to *delete* • No deref of null • No deref of dangling */&
Open questions	Completing GSL types: • Standardizing *variant<>* • Leave no valid reason to use raw unions + manual discriminant	Drive out disincentives: • Passing *array_view<>* as efficiently and ABI-stably as (*,length) • Elim. redundant checks	



PSA: Pointers are not evil

Smart pointers are good – they encapsulate ownership

Raw T* and T& are good – we want to maintain the efficiency of "just an address," especially on the stack (locals, parameters, return values)

Lifetime safety overview

- GSL types, aliases, concepts
  - *Indirection* concept:
    - **Owner (can't dangle):** *owner*<>, containers, smart pointers, …
    - **Pointer (could dangle):** *, &, iterators, *array_view*/*string_view*, ranges, …
    - *not_null<T>*: Wraps any Indirection and enforces non-null
  - *owner*<>: Alias, ABI-compatible, building block for smart ptrs, containers, …
    - Mainly *owner<T*>*
- Rules
  1. Prefer to allocate heap objects using *make_unique*/*make_shared* or containers.
  2. Otherwise, use *owner*<> for source/layout compatibility with old code.
     Each non-null *owner*<> must be deleted exactly once, or moved.
  3. Never dereference a null or invalid Pointer.
  4. Never allow an invalid Pointer to escape a function.

27

Approach

- **Local** rules, **statically** enforced
  - No run-time overhead
  - Whole-program guarantees if we build the whole program
- **Identify** Owners, **track** Pointers
  - Enforce **leak-freedom** for Owners
  - Track **"points to"** for Pointers
- **Few annotations**
  - **Infer** Owner and Pointer types:
    Contains an Owner ⇒ Owner
    Else, contains Pointer ⇒ Pointer
  - **Default** lifetime is correct for the vast majority of param/return Pointers

Principles

- A Pointer tracks its pointee(s) and must not outlive them
- Track the outermost object
  - Class member: track enclosing object
  - Array element: track enclosing array
  - Heap object: track its Owner
- Pointer parameters are valid for the function call & independent by default
  - Enforced in the caller: Prevent passing a Pointer the callee could invalidate
- A Pointer returned from a function is derived from its inputs by default
  - Enforced in the callee

28

Lifetime in three acts

Act I: Local analysis – function bodies

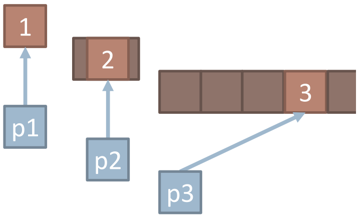
Act II: Calling functions – function parameters

Act III: Calling functions – function return/out values

Example: Pointer to local

- Here's a warmup:

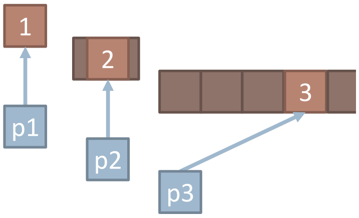
```
int *p1 = nullptr, *p2 = nullptr, *p3 = nullptr;   // p1, p2, p3 point to null
{
   int i = 1;
   struct mystruct { char c; int i; char c2; } s = {'a', 2, 'b'};
   array<int> a = {0,1,2,3,4,5,6,7,8,9};
   p1 = &i;                     // p1 points to i
   p2 = &s.i;                   // p2 points to s
   p3 = &a[3];                  // p3 points to a
   *p1 = *p2 = *p3 = 42;        // ok, all valid
} // A
```



Example: Pointer to local

- Here’s a warmup:

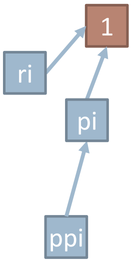
```
int *p1 = nullptr, *p2 = nullptr, *p3 = nullptr;   // p1, p2, p3 point to null
{
   int i = 1;
   struct mystruct { char c; int i; char c2; } s = {‘a’, 2, ’b’};
   array<int> a = {0,1,2,3,4,5,6,7,8,9};
   p1 = &i;                    // p1 points to i
   p2 = &s.i;                  // p2 points to s
   p3 = &a[3];                 // p3 points to a
   *p1 = *p2 = *p3 = 42;       // ok, all valid
} // A: destroy a, s, i → invalidate p3, p2, p1
*p1 = 1;           // ERROR, p was invalidated when i went out of scope at line A.
                   // Solution: increase i’s lifetime, or reduce p’s lifetime.
*p2 = *p3 = 1;     // (ditto for p2 and p3, except “s” and “a” instead of “i”)
```

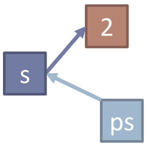


Example: Address-of, and Pointer to Pointer

- Warmup #2: Taking the address (of any object, incl. an Owner or Pointer)

```
int i  = 1;                     // non-Pointer
int& ri = i;                    // ri points to i
int* pi = &ri;                  // pi points to i


int** ppi = &pi;                // ppi points to Pointer pi


auto  s  = make_shared<int>(2);
auto* ps = &s;                  // ps points to Owner s
```

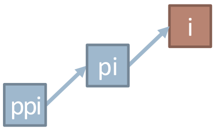




Example: Dereferencing

- Warmup #3: Dereferencing. From the previous example…

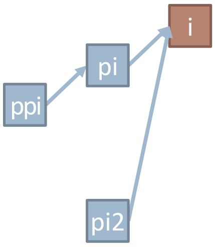
```
int i  = 0;
int* pi = &i;              // pi points to i
int** ppi = &pi;           // ppi points to pi
```



Example: Dereferencing

- Warmup #3: Dereferencing. From the previous example…

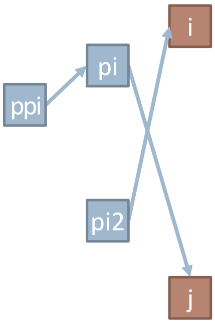
```
int i  = 0;
int* pi = &i;              // pi points to i
int** ppi = &pi;           // ppi points to pi

                           // IN:  ppi points to pi, pi points to i
int* pi2 = *ppi;           // *ppi points to i
                           // OUT: pi2 points to i
```



Example: Dereferencing

- Warmup #3: Dereferencing. From the previous example…

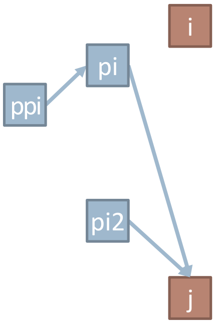
```
int i  = 0;
int* pi = &i;              // pi points to i
int** ppi = &pi;           // ppi points to pi

                           // IN:  ppi points to pi, pi points to i
int* pi2 = *ppi;           // *ppi points to i
                           // OUT: pi2 points to i
int   j = 0;
pi = &j;                   // pi points to j – **ppi points to j
```



Example: Dereferencing

- Warmup #3: Dereferencing. From the previous example…

```
int i  = 0;
int* pi = &i;              // pi points to i
int** ppi = &pi;           // ppi points to pi

                           // IN:  ppi points to pi, pi points to i
int* pi2 = *ppi;           // *ppi points to i
                           // OUT: pi2 points to i
int   j = 0;
pi = &j;                   // pi points to j – **ppi points to j
                           // IN:  ppi points to pi, pi points to j
pi2 = *ppi;                // *ppi points to j
                           // OUT: pi2 points to j
```



EOW

end of warmups

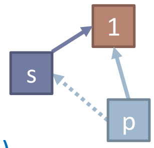
BOF

beginning of fun

Example: Pointer from Owner

- Getting a Pointer from an Owner:

```
auto s = make_shared<int>(1);
int* p = s.get();               // p points to s' = an object
                                // owned by s (current value)
*p = 42;                        // ok, p is valid
```

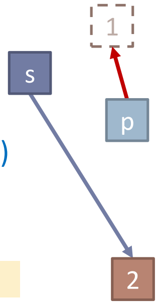


Example: Pointer from Owner

- Getting a Pointer from an Owner:

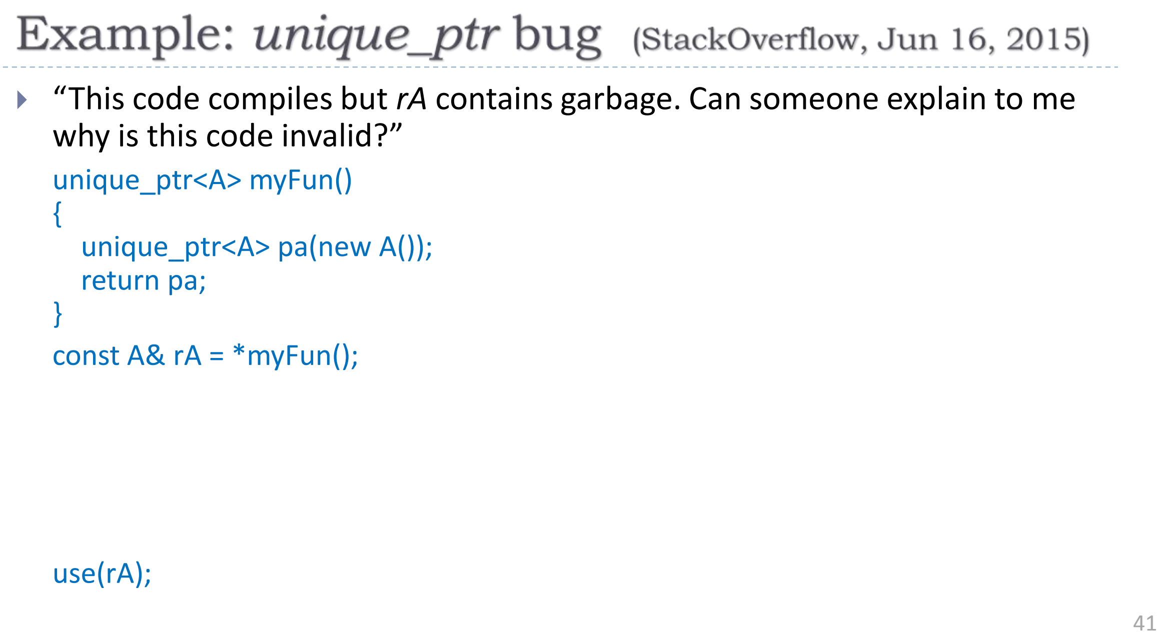
not specific to std:: smart pointers – intended to work for custom smart pointers

```
auto s = make_shared<int>(1);
int* p = s.get();               // p points to s' = an object
                                // owned by s (current value)
*p = 42;                        // ok, p is valid

s = make_shared<int>(2);  // A: modify s → invalidate p

*p = 43;                        // ERROR, p was invalidated by assignment to s at line A
```



not specific to smart pointers at all – general rule detects modifying an Owner

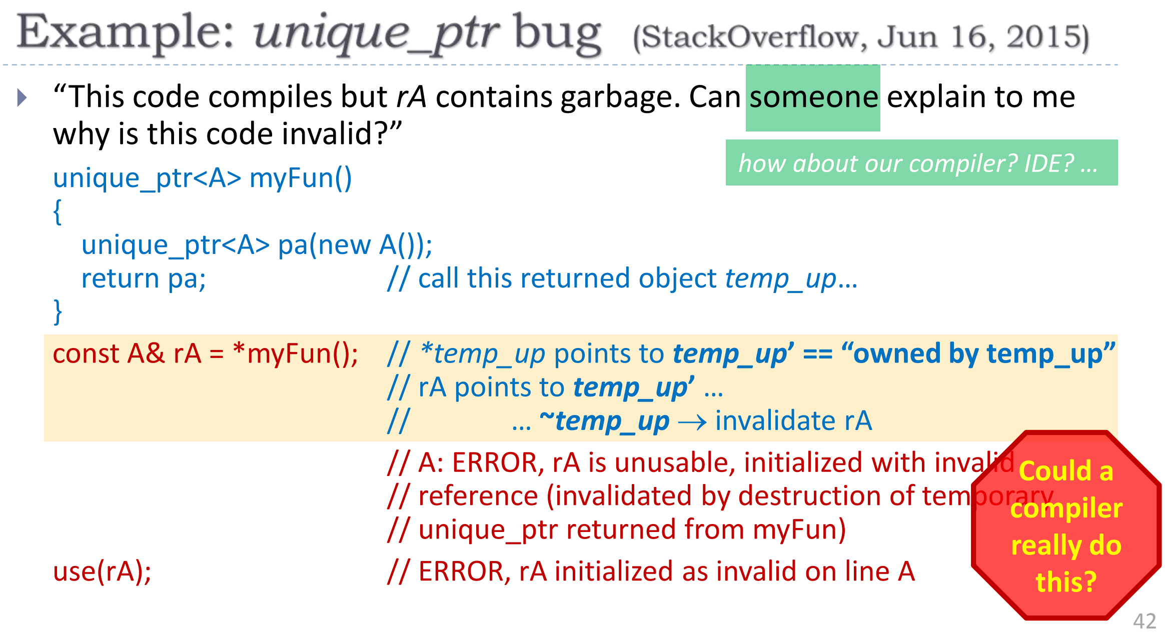
Could a compiler really do this?



Example: *unique_ptr* bug (StackOverflow, Jun 16, 2015)

- “This code compiles but *rA* contains garbage. Can someone explain to me why is this code invalid?”

```
unique_ptr<A> myFun()
{
   unique_ptr<A> pa(new A());
   return pa;
}
const A& rA = *myFun();




use(rA);
```



Example: *unique_ptr* bug (StackOverflow, Jun 16, 2015)

- “This code compiles but *rA* contains garbage. Can someone explain to me why is this code invalid?”

how about our compiler? IDE? …

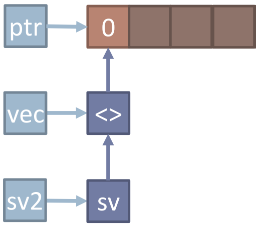
```
unique_ptr<A> myFun()
{
   unique_ptr<A> pa(new A());
   return pa;                   // call this returned object temp_up…
}
const A& rA = *myFun();         // *temp_up points to temp_up’ == “owned by temp_up”
                                // rA points to temp_up’ …
                                //            … ~temp_up → invalidate rA
                                // A: ERROR, rA is unusable, initialized with invalid
                                // reference (invalidated by destruction of temporary
                                // unique_ptr returned from myFun)
use(rA);                        // ERROR, rA initialized as invalid on line A
```

Could a compiler really do this?

Example: *shared_ptr<vector<int>>*

```
auto  sv  = make_shared<vector<int>>(100);
shared_ptr<vector<int>>* sv2 = &sv;     // sv2 points to sv
vector<int>* vec = &*sv;                // vec points to sv’
int* ptr = &(*sv)[0];                   // ptr points to sv’’
```

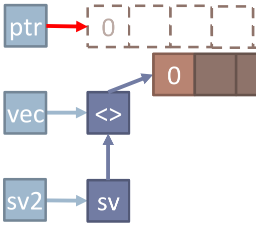
```
*ptr = 1;                     // ok
```



Example: *shared_ptr<vector<int>>*

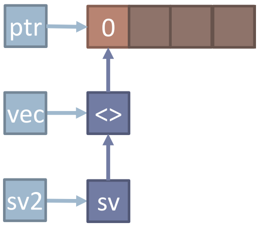
```
auto  sv  = make_shared<vector<int>>(100);
shared_ptr<vector<int>>* sv2 = &sv;     // sv2 points to sv
vector<int>* vec = &*sv;                // vec points to sv’
int* ptr = &(*sv)[0];                   // ptr points to sv’’
```

```
*ptr = 1;                     // ok

                              // points-to:       sv2       vec       ptr
                              //           IN:    sv        sv’       sv’’
vec->                         // same as “(*vec).”, and *vec is sv’
    push_back(1);             // A: modifying sv’ invalidates sv’’
                              //          OUT:    sv        sv’       invalid

*ptr = 2;                     // ERROR, ptr was invalidated by “push_back” on line A
```



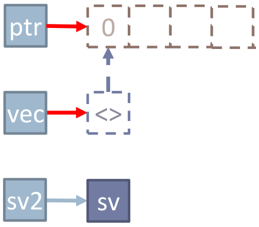
Example:
shared_ptr<vector<int>>

```
auto  sv  = make_shared<vector<int>>(100);
shared_ptr<vector<int>>* sv2 = &sv;    // sv2 points to sv
vector<int>* vec = &*sv;               // vec points to sv’
int* ptr = &(*sv)[0];                  // ptr points to sv’’

*ptr = 1;                  // ok

                           // points-to:      sv2     vec     ptr
                           //           IN:   sv      sv’     sv’’
vec->                      // same as “(*vec).”, and *vec is sv’
     push_back(1);         // A: modifying sv’ invalidates sv’’
                           //          OUT:   sv      sv’     invalid

*ptr = 2;                  // ERROR, ptr was invalidated by “push_back” on line A


ptr = &(*sv)[0];           // back to previous state to demonstrate an alternative...
```



Example:
shared_ptr<vector<int>>

```
auto  sv  = make_shared<vector<int>>(100);
shared_ptr<vector<int>>* sv2 = &sv;    // sv2 points to sv
vector<int>* vec = &*sv;               // vec points to sv’
int* ptr = &(*sv)[0];                  // ptr points to sv’’

*ptr = 1;                  // ok

                           // points-to:      sv2     vec     ptr
                           //           IN:   sv      sv’     sv’’
vec->                      // same as “(*vec).”, and *vec is sv’
     push_back(1);         // A: modifying sv’ invalidates sv’’
                           //          OUT:   sv      sv’     invalid

*ptr = 2;                  // ERROR, ptr was invalidated by “push_back” on line A


ptr = &(*sv)[0];           // back to previous state to demonstrate an alternative...

                           //           IN:   sv      sv’     sv’’
(*sv2).                    // *sv2 is sv
      reset();             // B: modifying sv invalidates sv’
                           //          OUT:   sv      invalid invalid

vec->push_back(1);         // ERROR, vec was invalidated by “reset” on line B
*ptr = 3;                  // ERROR, ptr was invalidated by “reset” on line B
```



Could a compiler really do this?

Branches, Loops, *nullptr*, *throw*, *catch*

- Branches add the possibility of "or": *p* can point to ***x*** or ***y***
- Loops are like branches: If exit set != entry set, process loop body once more
- "Points to null" removed in a branch that tests against null pointer constant

```
p = cond ? x : nullptr;     // A: p points to x or null
*p = 42;                    // ERROR, p could have been set to null on line A
if (p != nullptr)           // or != 0, or != NULL, …
   *p = 42;                 // ok, p points to x
```

- *try*/*catch*: treat a *catch* block as if it could have been entered from every point in the *try* block where an exception could have been raised
  - Record all potential invalidations in the *try* block (any may have executed)
  - Remove any revalidations in the *try* block (potentially none were executed)
  - Note: This is an example of how the model is intentionally **conservative**. Finalizing the rules against RWC includes ensuring reasonably low **false positives**.

47

Lifetime in three acts

Act I: Local analysis – function bodies

Act II: Calling functions – function parameters

Act III: Calling functions – function return/out values

48

```
T* p = …;
f( p );
```

Here, I have a pointer for you.

It’s good. Trust me.

Calling functions: Parameter lifetimes

- In callee, **assume** Pointer params are valid for the call, and independent.

```
void f( int* p ) { … }          // in f, assume p is valid for its lifetime (≈“p points to p”)
```

- In caller, **enforce** no arguments that we know the callee can invalidate.

```
void f(int*);
void g(shared_ptr<int>&, int*);
shared_ptr<int> gsp = make_shared<int>();
int main() {
   f(gsp.get());                // ERROR, arg points to gsp’, and gsp is modifiable by f
   auto sp = gsp;
   f(sp.get());                 // ok, arg points to sp’, and sp is not modifiable by f
   g(sp, sp.get());             // ERROR, arg2 points to sp’, and sp is modifiable by f
   g(gsp, sp.get());            // ok, arg2 points to sp’, and sp is not modifiable by f
}
```

#1 correctness issue using smart pointers

Could a compiler really do this?

Aside: Smart pointers are great
… but commonly misused

#1 correctness issue with smart pointers:

Accidental silent invalidation in the case just shown (incl. reentrancy)
→ can fully address with Lifetime rules

#1 performance issue with smart pointers:

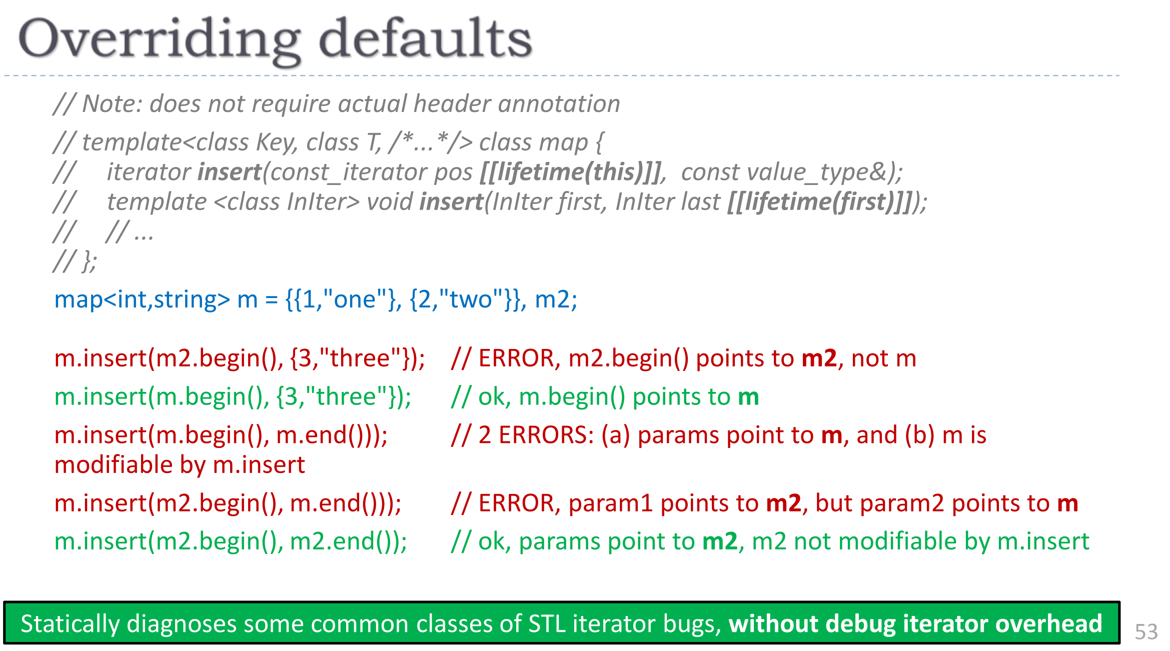
Passing as parameters inappropriately
→ can fully address with Guideline rules (see Bjarne’s talk)

Overriding defaults

- Sometimes you want to override the defaults. For example, in STL:
  - Insert-with-hint *insert(iter,t)* assumes *iter* is into *this (not allowed by default because *iter* could be (is!) invalidated by *insert*). We can express this using [[lifetime(this)]].
  - Range-based insert *insert(iter1,iter2)* assumes *iter1*, *iter2* are not into *this (the default). It also assumes that *iter1* and *iter2* have the same lifetime (not the default). We can express this using [[lifetime(iter1)]].

```
template<class Key, class T, /*...*/> class map {
   iterator insert(const_iterator pos [[lifetime(this)]],  const value_type&);
   template <class InIter> void insert(InIter first, InIter last [[lifetime(first)]]);
   // ...
};
```

Statically diagnoses some common classes of STL iterator bugs, **without debug iterator overhead**

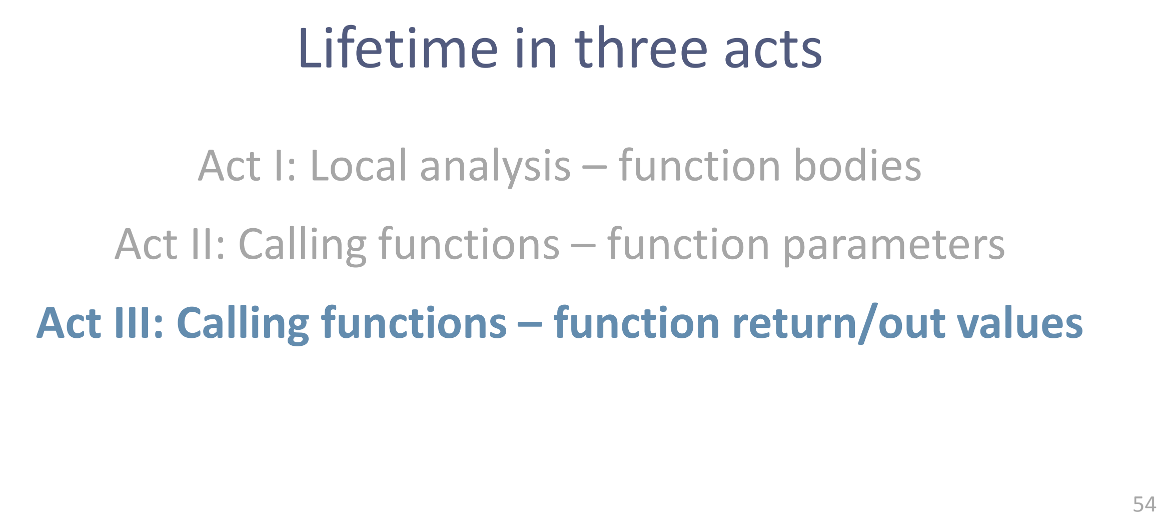


Overriding defaults

```
// Note: does not require actual header annotation
// template<class Key, class T, /*...*/> class map {
//     iterator insert(const_iterator pos [[lifetime(this)]],  const value_type&);
//     template <class InIter> void insert(InIter first, InIter last [[lifetime(first)]]);
//     // ...
// };
map<int,string> m = {{1,"one"}, {2,"two"}}, m2;

m.insert(m2.begin(), {3,"three"});   // ERROR, m2.begin() points to m2, not m
m.insert(m.begin(), {3,"three"});    // ok, m.begin() points to m
m.insert(m.begin(), m.end()));       // 2 ERRORS: (a) params point to m, and (b) m is
modifiable by m.insert
m.insert(m2.begin(), m.end()));      // ERROR, param1 points to m2, but param2 points to m
m.insert(m2.begin(), m2.end());      // ok, params point to m2, m2 not modifiable by m.insert
```

Statically diagnoses some common classes of STL iterator bugs, **without debug iterator overhead**

53



Lifetime in three acts

Act I: Local analysis – function bodies

Act II: Calling functions – function parameters

Act III: Calling functions – function return/out values

54

```
int*  f( /*…*/ );
```

I see you have a pointer for me.

I wonder where you got it from?

Sound and conservative

- In principle, you have to "state" the lifetime of a returned Pointer.
  - Caller **assumes** that lifetime.
  - Callee **enforces** that lifetime when separately compiling callee body.
- Defaults are to minimize the frequency that you have to "state" it explicitly, so that most of the time you "state" it the convenient way: as whitespace.
  - **Vast majority of returned Pointers are derived from Owner and Pointer inputs.** No annotation needed.
  - If there are no inputs (e.g., Singletons), we assume you're returning a pointer to something *static*. This handles Singleton *instance* functions, etc. No annotation needed.
  - **Only if it's "something else": Clear error when separately compiling the callee.** Then annotate the declaration (to fix the compile error).

Calling functions: Return/out lifetimes

- A returned Pointer is assumed to come from Owner/Pointer inputs.
  - Vast majority of cases: Derived from Owner and Pointer arguments.

```
int*    f( int* p, int* q );                        // ret points to *p or *q
char*   g( string& s );                             // ret points to s' (s-owned)
```

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

```
char*   find_match( string& s, const string& sub );  // ret points to s'
```

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

```
const char* point_into( const string& sub );         // ret points to sub'
```

  - Params that are Owner rvalue strong magnets: ***owner&&*** parameters
    - Always ignored, because *owner&&* strongly attracts temporary owners.

```
int*    find_match( unique_ptr<X>&& );               // ret points to static
```

Example: *find_match*

Declaration, and caller code

```
char*                           // default: points to s'
find_match(string& s, const string& sub);

// --- sample call sites ---------------------------------------

string str = "xyzzy", z = "zzz";

p = find_match(str, z);                // p points to str'

p = find_match(str, "literal");        // p points to str'

p = find_match(str, z+"temp");         // p points to str'

p = find_match(str, "UDL"s);           // p points to str'

// all p's are valid until str is modified or destroyed
```

Callee

```
char*                           // default: points to s'
find_match(string& s, const string& sub)
{
   if(...) return &s[i];        // ok, {s'} ⊇ {s'}

   if(...) return &sub[j];      // ERROR, {s'} ⊉ {sub'}


   char* ret = nullptr;         // ret points to null

   if(...) ret = &s[i];         // ok, ret points to s'
   else ret = &sub[i];          // ok, ret points to sub'
   // merge branches: here ret points to s' or sub'

   return ret;                  // ERROR, {s'} ⊉ {s',sub'}
}
```

Examples: *vector<T>::operator[] & begin*

operator[]

```
T&           // default: points to (*this)'
vector<T>::operator[](size_t);


// --- sample call site ---

vector<int> v = {1,2,3,4};

auto p = &vec[0];    // p points to v'

// p is valid until v is modified or destroyed
```

begin

```
iterator     // default: points to (*this)'
vector<T>::begin();


// --- sample call site ---

vector<int> v = {1,2,3,4};

auto it = begin(vec);    // it points to v'

// it is valid until v is modified or destroyed
```

Example: *std::min, std::max* (AA, since 20th century)

- Since C++98:

```
template<class T>
const T& min(const T& a, const T& b)   { return b<a ? b : a; }
```

  - *"Youbetcha, that's efficient. I can foresee no problems with that…"*

```
int x=10, y = 2;
int& ref = min(x,y);          // ok
cout << ref;                  // ok, prints 2
int& bad = min(x,y+1);

cout << bad;
```

Example: *std::min, std::max* (AA, since 20th century)

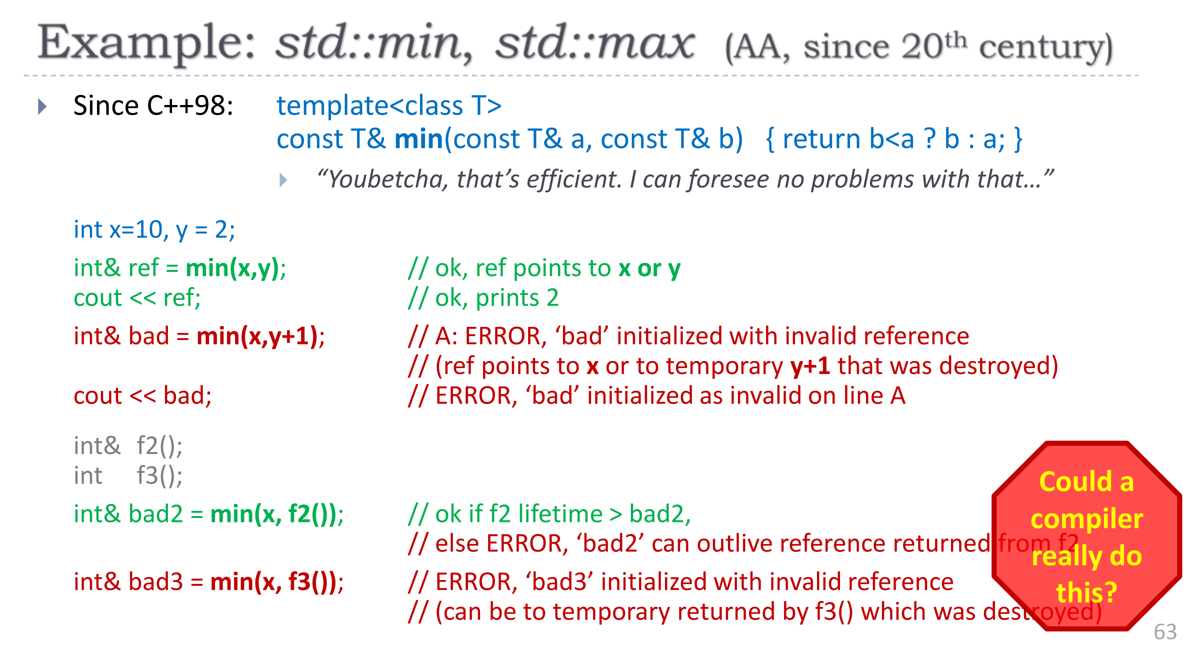
- Since C++98: template<class T>
  const T& **min**(const T& a, const T& b) { return b<a ? b : a; }
  - *"Youbetcha, that's efficient. I can foresee no problems with that…"*

```
int x=10, y = 2;
int& ref = min(x,y);            // ok
cout << ref;                    // ok, prints 2
int& bad = min(x,y+1);          // trap for the unwary programmer – and data-dependent
                                //                  (std::max would not fail in this case!)
cout << bad;                    // boom, probably

int&  f2();
int   f3();
int& bad2 = min(x, f2());

int& bad3 = min(x, f3());
```

Example: *std::min, std::max* (AA, since 20th century)

- Since C++98: template<class T>
  const T& **min**(const T& a, const T& b) { return b<a ? b : a; }
  - *"Youbetcha, that's efficient. I can foresee no problems with that…"*

```
int x=10, y = 2;
int& ref = min(x,y);            // ok
cout << ref;                    // ok, prints 2
int& bad = min(x,y+1);          // trap for the unwary programmer – and data-dependent
                                //                  (std::max would not fail in this case!)
cout << bad;                    // boom, probably

int&  f2();
int   f3();
int& bad2 = min(x, f2());       // ok… if f2 returns a reference with suitable lifetime
                                // otherwise, trap for the unwary programmer
int& bad3 = min(x, f3());       // trap for the unwary programmer
```



Example: *std::min, std::max* (AA, since 20th century)

- Since C++98:

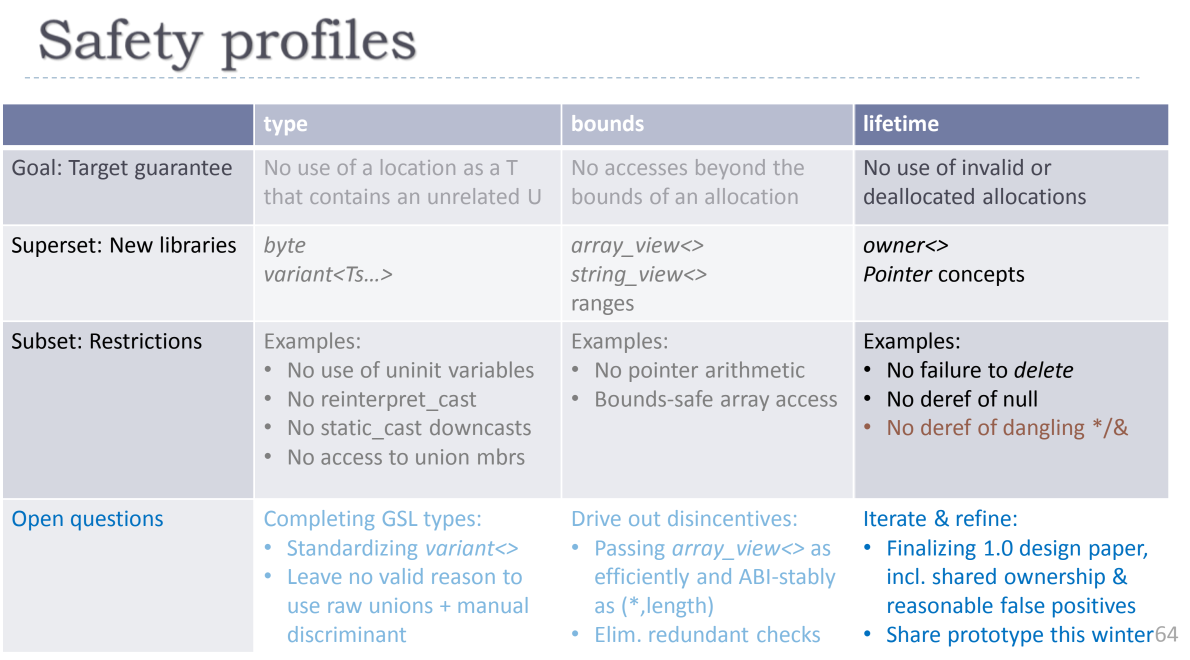
```
template<class T>
const T& min(const T& a, const T& b)   { return b<a ? b : a; }
```

  - *"Youbetcha, that's efficient. I can foresee no problems with that…"*

```
int x=10, y = 2;
int& ref = min(x,y);           // ok, ref points to x or y
cout << ref;                   // ok, prints 2
int& bad = min(x,y+1);         // A: ERROR, 'bad' initialized with invalid reference
                               // (ref points to x or to temporary y+1 that was destroyed)
cout << bad;                   // ERROR, 'bad' initialized as invalid on line A

int&  f2();
int   f3();
int& bad2 = min(x, f2());      // ok if f2 lifetime > bad2,
                               // else ERROR, 'bad2' can outlive reference returned from f2
int& bad3 = min(x, f3());      // ERROR, 'bad3' initialized with invalid reference
                               // (can be to temporary returned by f3() which was destroyed)
```

Could a compiler really do this?

63



Safety profiles

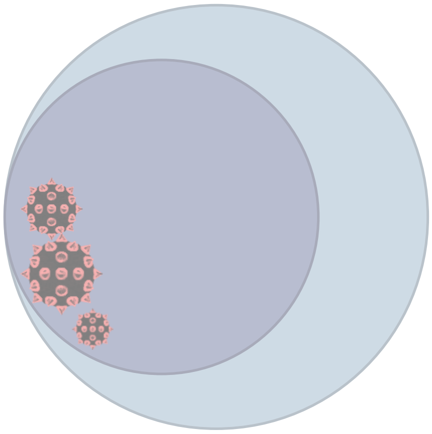
	type	bounds	lifetime
Goal: Target guarantee	No use of a location as a T that contains an unrelated U	No accesses beyond the bounds of an allocation	No use of invalid or deallocated allocations
Superset: New libraries	*byte* *variant<Ts…>*	*array_view<>* *string_view<>* ranges	*owner<>* *Pointer* concepts
Subset: Restrictions	Examples: • No use of uninit variables • No reinterpret_cast • No static_cast downcasts • No access to union mbrs	Examples: • No pointer arithmetic • Bounds-safe array access	Examples: • No failure to *delete* • No deref of null • No deref of dangling */&
Open questions	Completing GSL types: • Standardizing *variant<>* • Leave no valid reason to use raw unions + manual discriminant	Drive out disincentives: • Passing *array_view<>* as efficiently and ABI-stably as (*,length) • Elim. redundant checks	Iterate & refine: • Finalizing 1.0 design paper, incl. shared ownership & reasonable false positives • Share prototype this winter

64

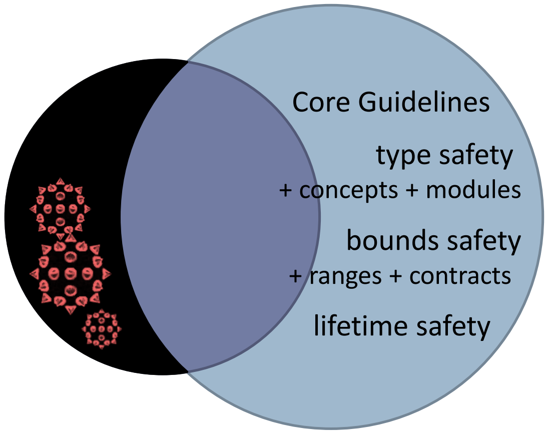
Can safety make C++ simpler?

- Yes, directly (obviously): Statically eliminate classes of errors.
- But also indirectly: We already saw *std::min* & *std::max*. Now…
  - **Q:** Why do C++ smart pointers like *shared_ptr<T>* have "*.get()*" instead of a (convenient!) implicit conversion to *T**?
  - **A:** Accidental conversion to *T** allows code to accidentally compile:
    - and make wild pointers (oops, *sp+42* compiled, but I meant **sp+42*)
    - and dangle pointers (oops, didn't know I got a raw pointer, wasn't careful)
- Safety affects library design:
  - Conjecture: If we can prevent **bounds** (pointer arithmetic) and **lifetime** (dangling) errors, then smart pointers could safely implicitly convert to raw pointers.

Superset | Superset + Subset



ISO C++98 → C++11 → C++14 → …



Core Guidelines

type safety
+ concepts + modules

bounds safety
+ ranges + contracts

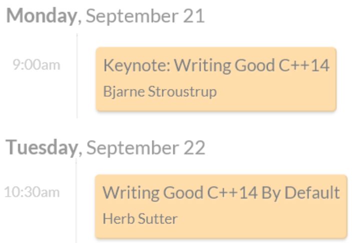
lifetime safety

ISO C++ and C++ Core Guidelines

Acknowledgments (reprise)

- This is the beginning of open source project(s). We need your help.
  - **C++ Core Guidelines** – all about "getting the better parts by default" (*github.com/isocpp*)
  - **Guideline Support Library (GSL)** – first implementation available (*github.com/microsoft/gsl*) – portable C++, tested on Clang / GCC / Xcode / MSVC, for (variously) Linux / OS X / Windows
  - **Checker tools** – first implementation next month (MSVC 2015 Upd.1 CTP timeframe) – "type" and "bounds" safety profiles (initially Windows binary, intention is to open source)
- Just getting to this starting point is thanks to collaboration and feedback from:
  - Bjarne Stroustrup, myself, Gabriel Dos Reis, Neil MacIntosh, Axel Naumann, Andrew Pardoe, Andrew Sutton, Sergey Zubkov
  - Andrei Alexandrescu, Jonathan Caves, Pavel Curtis, Joe Duffy, Daniel Frampton, Chris Hawblitzel, Shayne Hiet-Block, Peter Juhl, Leif Kornstaedt, Aaron Lahman, Eric Niebler, Gor Nishanov, Jared Parsons, Jim Radigan, Dave Sielaff, Jim Springfield, Jiangang (Jeff) Zhuang, & more…
  - CERN, Microsoft, Morgan Stanley
    - **GSL is derived from production code:** network protocol handlers; kernel Unicode string handlers; graphics routines; OS shell enumerator patterns; cryptographic routines; …

So far: Related talks:



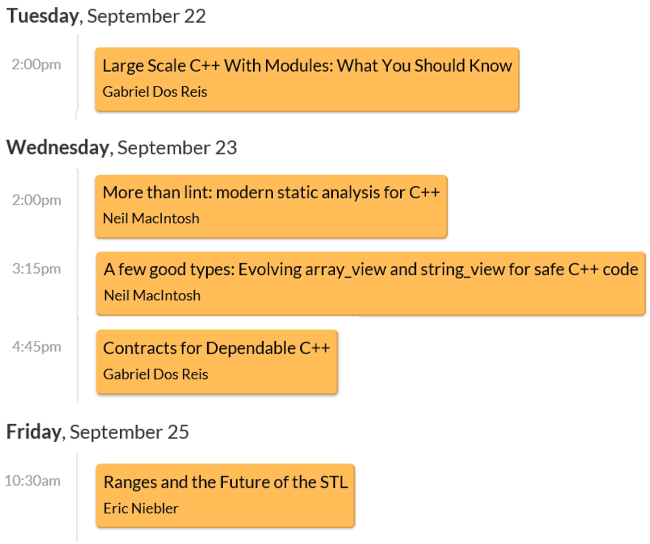
Monday, September 21

9:00am Keynote: Writing Good C++14
Bjarne Stroustrup

Tuesday, September 22

10:30am Writing Good C++14 By Default
Herb Sutter

CppCon 2015



Tuesday, September 22

2:00pm Large Scale C++ With Modules: What You Should Know
Gabriel Dos Reis

Wednesday, September 23

2:00pm More than lint: modern static analysis for C++
Neil MacIntosh

3:15pm A few good types: Evolving array_view and string_view for safe C++ code
Neil MacIntosh

4:45pm Contracts for Dependable C++
Gabriel Dos Reis

Friday, September 25

10:30am Ranges and the Future of the STL
Eric Niebler

Writing Good C++14… *By Default*

Questions? (really)