Pursue P1179 as a Lifetime Safety TS

Document Number: **P3465 R0**
Date: 2024-10-15
Reply-to: Herb Sutter (herb.sutter@gmail.com)
Audience: SG23, EWG

Contents

Abstract

Lifetime safety is the hardest of the four major programming language safety areas we most need to improve (the others are bounds, type, and initialization).

We have an implemented approach that requires near-zero annotation of existing source code.

With the committee's blessing as a TS, we can finish and ship it.

I think it is worth pursuing a compatible path first before, or at least concurrently with, trying to graft another foreign language's semantics onto C++ which turns C++ into "something else" and/or build an off-ramp from C++. — I like Rust a lot, and every language should learn from others! But Rust is Rust, and C++ and C++, and any language's first choice should not be to just transliterate features from another language that has its own great but fundamentally different object and lifetime design (e.g., just as we wouldn't copy C#'s or Swift's object and lifetime models for C++, though C# and Swift are great languages too).

1 Motivation

Lifetime safety is the hardest of the four major programming language safety areas we most need to improve (the others are bounds, type, and initialization).

We have an implemented approach that requires near-zero annotation of existing source code: [P1179R1], which is also the *C++ Core Guidelines* Lifetime profile [Pro.Lifetime].

In WG 21, until now P1179 was submitted as an "FYI" informational paper that was not proposed or presented.

However, now we have current proposals to graft foreign languages' lifetime models onto C++ (e.g., Circle) or to invent new untried models (e.g., Hylo). So I think it's time to also consider [P1179R1]: With the committee's blessing as a TS, we can finish and ship it.

2 Proposal

For full details, see [P1179R1]. It includes:

- why this is a general solution that works for all Pointer-like types (not just raw pointers, but also iterators, views, etc.) and Owner-like types (not just smart pointers, but also containers etc.)
- why this is a scalable compile-time solution, because it requires only function-local analysis
- **why zero annotation is required by default, because existing C++ source code already contains sufficient information**
- examples of many common familiar bugs are already caught at compile time (e.g., changing a container while iterating over it and accidentally invalidating the iterator)

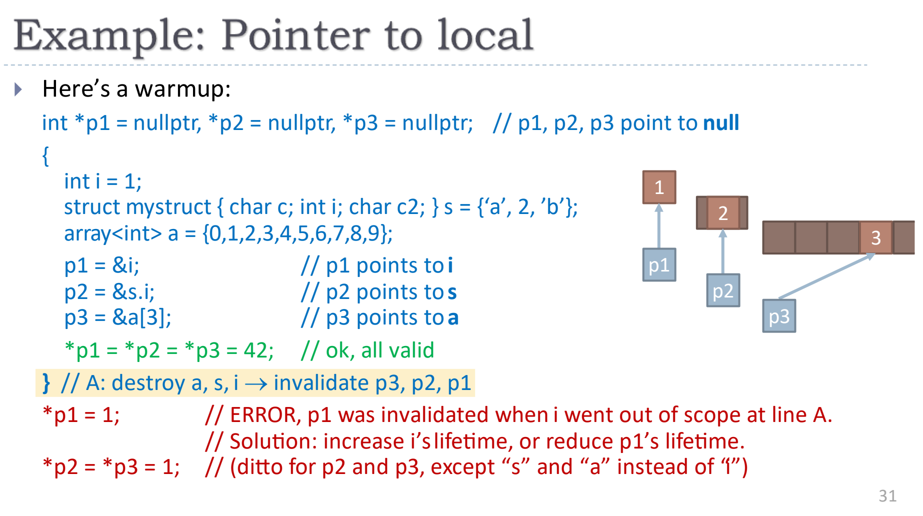
2.1 Examples

[Sutter2015] is a video of the talk I gave at CppCon 2015 where I explained the model and Neil MacIntosh live-demonstrated the following examples on stage, working in the Visual C++ static analysis.

The following are slide examples from that talk showing examples already found by this analysis.

Notes:

- **None of the following examples require any annotation.**
- Each of the red "Stop sign" icons, "Could a compiler really do this?", is a place where the talk video contains a live demonstration using the early Visual C++ implementation.

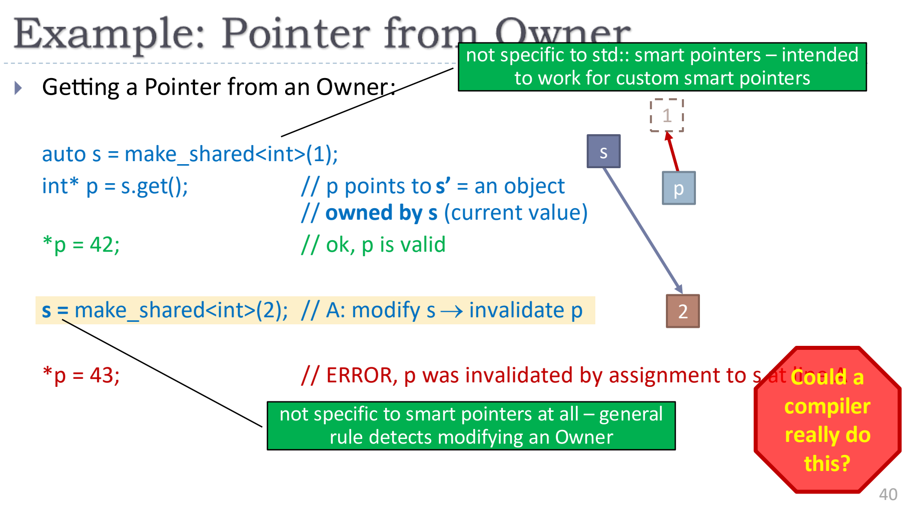


Example: Pointer to local

- 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, p1 was invalidated when i went out of scope at line A.
                    // Solution: increase i's lifetime, or reduce p1's lifetime.
*p2 = *p3 = 1;      // (ditto for p2 and p3, except "s" and "a" instead of "i")
```

31



Example: Pointer from Owner

- Getting a Pointer from an Owner:

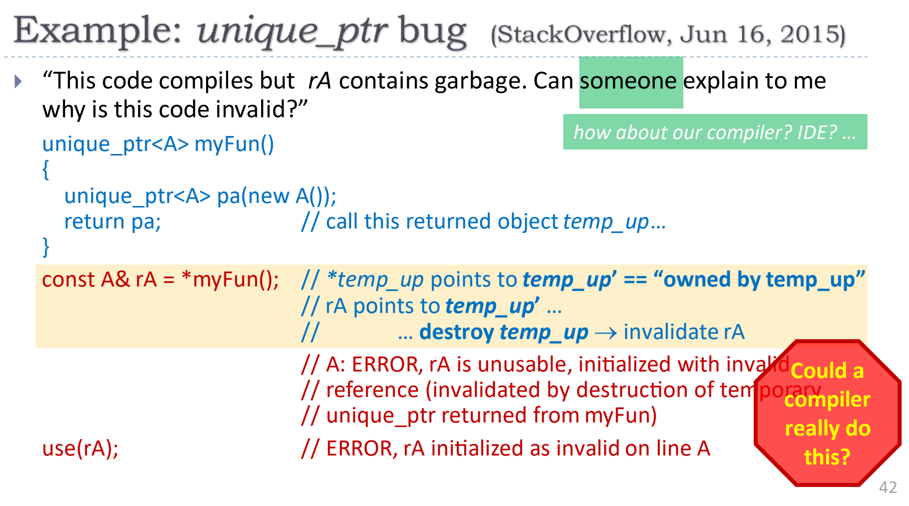
not specific to std:: smart pointers – intended to work for custom smart pointers

```
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?

40



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

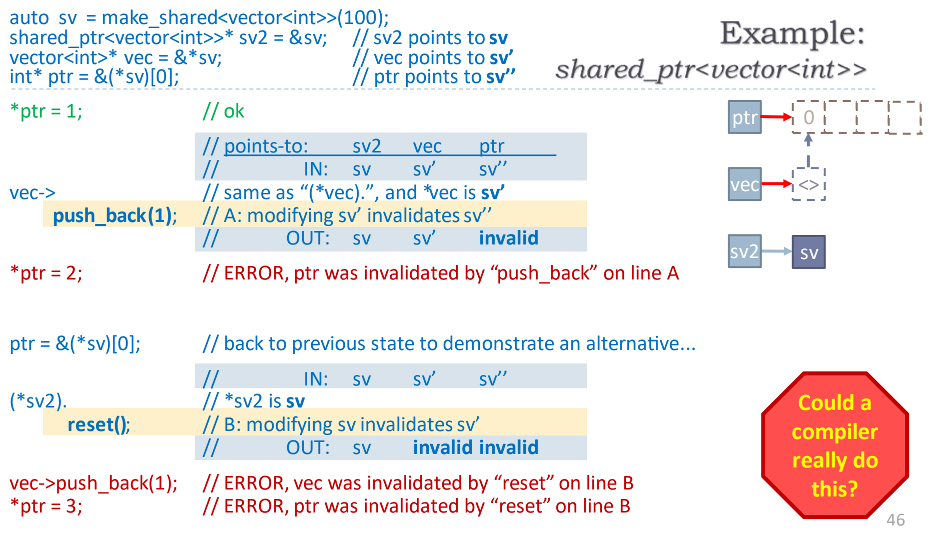
- "This code compiles but *rA* contains garbage. Can someone explain to me why is this code invalid?"

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' …
                               //          … destroy 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
```

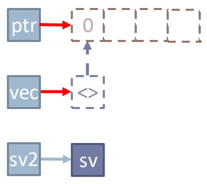
Could a compiler really do this?

42



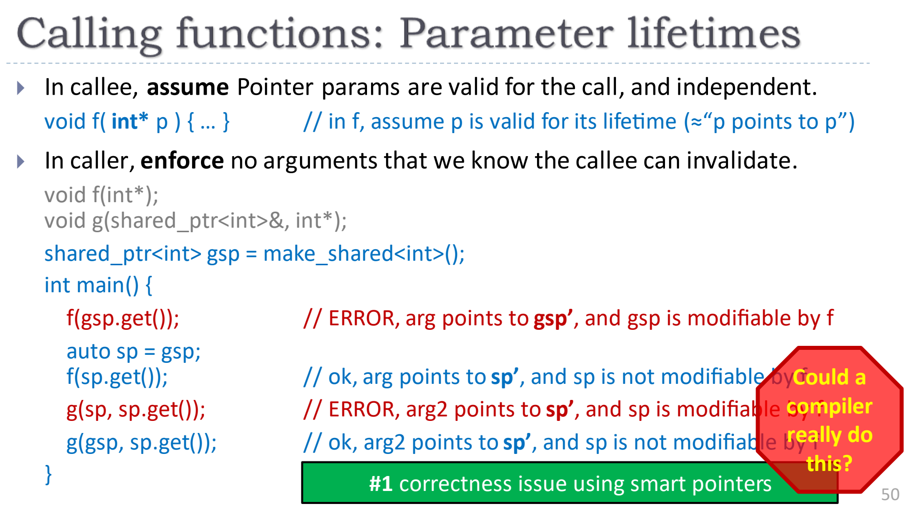
Example: *shared_ptr<vector<int>>*

```
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?

46



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")
```

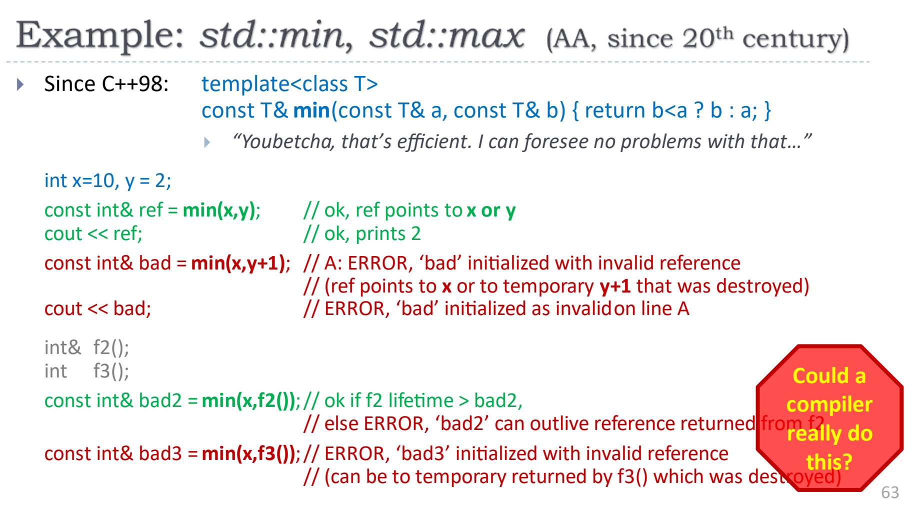
- In caller, **enforce** no arguments that we know the callee can invalidate.

```
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?

50



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;
const int& ref = min(x,y);     // ok, ref points to x or y
cout << ref;                   // ok, prints 2
const 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();
const int& bad2 = min(x,f2()); // ok if f2 lifetime > bad2,
                               // else ERROR, 'bad2' can outlive reference returned from f2
const 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

2.2 Why encourage further investigation of this lifetime solution?

Unlike competing proposals to graft other languages' lifetime models onto C++, [P1179R1]:

- **embraces C++'s current language features, idioms, and guidance**, rather than trying to drastically change C++ into something else
- **already finds many common lifetime bugs in existing code with near-zero annotation** ("just recompile your code with a Lifetime TS compiler and we'll find high quality lifetime errors")
- has been encouraged in the *C++ Core Guidelines*, [Pro.Lifetime]
- has been designed with the assistance of experienced static analysis experts in C++ and other languages
- has been partially implemented by two vendors (Microsoft, JetBrains), and so is the least experimental current proposal

2.3 Why a TS?

I consider this "95% done." It has languished a bit with incomplete implementations because of other commitments and distractions, but some tangible committee encouragement will make it possible to finish implementing the remaining parts of the design (mainly around parameters, which is how the local analysis composes to cover the program) and to gain more usage experience.

Publishing a TS based on this work will achieve that.

I think it is worth pursuing this compatible path first before, or at least at the same time as, trying to graft another foreign language's semantics onto C++ which turns C++ into "something else" and/or build an off-ramp from C++.

3 References

[Sutter2015] H. Sutter. "Writing Good C++14… By Default" (CppCon 2015 talk). This link is to the part of the talk starting at 29:05 that explains and live-demonstrates [P1179R1] already using an early prototype in the Visual C++ 2015 compiler to catch many common bugs including the ones mentioned in this paper.

[P1179R1] H. Sutter. "Lifetime safety: Preventing common dangling" (WG21 paper, November 2019).

[Pro.Lifetime] B. Stroustrup and H. Sutter, editors. *C++ Core Guidelines* Pro.Lifetime Profile for lifetime safety.