Lifetime safety: Preventing common dangling

Document Number: **P1179 R1 – version 1.1**
Date: 2019-11-22
Reply-to: Herb Sutter (hsutter@microsoft.com)
Audience: Informational

Major changes in R1: Added SharedOwner concept. Added "Diagnostic sensitivity options" section to define a way to start with zero-false-positive warnings and increasingly opt into additional diagnostics. Changed function annotation syntax to contract-like `pre(cond)` and `post(cond)` syntax, and added that they are implicitly inherited by virtual overrides. Renamed `static` to `global` to reduce confusion with the keyword. Refined default lifetimes semantics of Pointer-to-Pointer parameters.

Abstract

This paper defines the Lifetime profile of the [C++ Core Guidelines]. It shows how to efficiently diagnose many common cases of dangling (use-after-free) in C++ code, using only local analysis to report them as deterministic readable errors at compile time. The approach is to identify variables that are of generalized "Owner" types (e.g., smart pointers, containers, `string`) and "Pointer" types (e.g., `int*`, `string_view`, `span`, iterators, and ranges), and then use a local simple acyclic control flow graph (ACFG) analysis to track what each Pointer points to and identify when modifying an Owner invalidates a Pointer. The analysis leverages C++'s existing strong notions of scopes, object lifetimes, and `const` that carry rich information already available in reasonably modern C++ source code. Interestingly, it appears that with minor extension this analysis can also detect uses of local moved-from variables (use-after-move), which are a form of dangling.

The warnings are deterministic and portable (give consistent results across implementations, validated by implementations in MSVC and Clang) and efficient enough to run during regular compilation (the current Clang-based implementation is unoptimized and incurs only about 5% overhead in compiling large LLVM translation units).

Note: De facto vs. de jure

The primary positive impact of this work comes from being *de facto* available in C++ compilers, even if it is never *de jure* standardized. Having major compilers routinely diagnose these problems will improve or eliminate C++'s reputation for being a breeding ground for dangling pointers/iterators/`string_view`s/etc.

However, since this work was started, there have been parallel EWG/LEWG proposals from other authors, notably [P0936R0], that propose *de jure* standardization of widespread annotations in the language and the standard library for cases that are covered by this work (and, in this proposal, typically without annotation; §2.6.2 describes how all examples from [P0936R0] are diagnosed without any annotation). So I submitted this work as a WG21 P paper, and one result appears to have been that it has been instrumental in removing the safety objections to standardizing `string_view` in C++20 and avoiding pressure to have other annotation-heavier lifetime proposals. If there is strong committee interest in standardizing these diagnostics so they are guaranteed to be available in the box in all C++ compilers, I could do the additional work to turn this into an actual EWG/LEWG proposal (but even then I would prefer to initially target a TS).

1 Overview

1.1 Summary: Approach and illustrative examples

This paper aims to efficiently diagnose dangling in common cases at build time, by leveraging C++'s existing strong notions of *scopes*, *object lifetimes*, and `const` to identify invaliding operations. The approach works on generalized "Owner" and "Pointer" types, and applies the rules equally to all such types:

- An Owner owns the objects it refers to. Examples: `shared_ptr`, `string`, `vector`.
- A Pointer does not own what it refers to. Examples: `int*`, `string_view`, `span`, `vector<T>::iterator`.

1.1.1 Track "points-to" information

In this code, a human programmer can see that dereferencing `p` is fine on line B, but dangling on line D:

```
int* p = nullptr;
{
    int x = 0;
    p = &x;                 // A
    *p = 42;                // B: human says "ok"
}                           // C
*p = 42;                    // D: human says "error"
```

How does a human know D is a bug? By following the code and seeing that `p` points to `x`, which went out of scope.

This paper teaches the compiler to "see" that too, by doing the same thing we did in our heads: **For each local pointer variable, track the set of things it could point to** at any given point in the function's source code, by following the function's control flow paths. In this paper, that is called the `pset`, or "points-to" set, which typically contains the name of local variables, but can also contain the special values `null`, `global`, and `invalid`.

Here is the same example, with the compiler tracking `p`'s points-to set through the function. We update `p`'s points-to set each time we modify `p`, and we also update it when something it points to goes out of scope: Here, when `x` goes out of scope, we replace "`x`" with "`invalid`" in every points-to set that contains `x`, which means for any pointer that at that point in the function could have been pointing to `x`:

```
// 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}
```

We can give useful error messages because we can say what series of `pset` changes led to `invalid`. How did `invalid` enter `p`'s `pset`? Because the set contained `x` when `x` was invalidated on line C. How did the set get to have `x` on line C? Because of the assignment on line A. So we can report something like this:

```
                            // error: '*p' is illegal, p was invalidated when 'x'
                            //        went out of scope on line C (path: A→C→D)
```

1.1.2 Generalizing: Pointers

There's nothing special about built-in * pointers. **All rules apply equally to any generalized "Pointer" type that refers indirectly to another object it doesn't own**, including iterators, ranges, views, spans, proxies, and more. For example, the identical rules diagnose the identical bug when using a `string_view`:

```
// godbolt.org/z/ytRTKt
string_view s;                  // pset(s) = {null}
{
    char a[100];
    s = a;                      // A: pset(s) = {a}
    cout << s[0];               // B: ok
}                               // C: x destroyed ⇒ set pset(s) = {invalid}
cout << s[0];                   // D: error: 's[0]' is illegal, s was invalidated when 'a'
                                //    went out of scope on line C (path: A→C→D)
```

1.1.3 Generalizing: Owners

Objects are often owned by other objects. **A generalized "Owner" type is a type that owns another object**, including containers, owning smart pointers, and more. When a Pointer `p` is made to point to data owned by an Owner `o` we set `pset(p) = {o'}`, or "something owned by `o`." The main difference is that, in addition to changing `o` to `invalid` in all psets when `o` goes out of scope, we also set `o'` to invalid when `o` is modified, because modifying `o` can release the data it points to. For example:

```
// 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 {a} is alive
name = "frobozz";               // C: name modified ⇒ set pset(s) = {invalid}
cout << s[0];                   // D: error: 's[0]' is illegal, s was invalidated when
                                //    'name' was modified on line C (path: A→C→D)
```

Note that "modifying `o`" by default simply means any non-`const` use of `o`, such as calling a non-`const` member function of `o`. That's a useful default that works most of the time; where appropriate, it can be overridden to state that functions like `map::insert` that are non-`const` but don't actually invalidate (see §2.5.7.10).

This paper's rules ignore the specific type of an Owner or Pointer; for example, there is no special rule to treat a raw `int*` pointer any differently from (say) a `string_view` or a `span`. Here are some variations of the same example showing different types diagnosed with the same rules, and the expected error quality:

View variation: `string_view` `godbolt.org/z/qTle8G`	**Proxy variation: `vector<bool>::reference`** `godbolt.org/z/iq5Qja`

`string s = "abcdefghijklmnopqrstuvwxyz";`
`string_view sv = s;`
`if (sv[0] == 'a') { } // ok`
`s = "xyzzy";  // A: invalidates sv`
`if (sv[0] == 'a') { } // ERROR, sv was invalidated`
`// by 's =' (line A)`
	`vector<bool> vb{false,true,false,true};`
	`auto proxy = vb[0];`
	`if (proxy) {/*...*/}  // ok`
	`vb.reserve(100);  // A: invalidates proxy`
	`if (proxy) {/*...*/}  // ERROR, proxy was invalidated`
	`// by 'vb.reserve' (line A)`

Range variation: `filter_view` (Ranges TS) `godbolt.org/z/eqCRLx`	**Iterator variation: `regex_iterator`**

`vector<int> ints{0,1,2,3,4,5};`
`auto even = [](int i){ return !(i%2); };`
`auto view = ints | view::filter(even);`
`cout << *begin(view); // ok`
`ints.push_back(6);  // A: invalidates view`
`cout << *begin(view); // ERROR, view was invalidated`
`// by 'ints.push_back' (line A)`
	`string s = "The quick brown fox jumps over";`
	`regex r("[^\\s]+");`
	`auto iter = sregex_iterator(s.begin(), s.end(), r);`
	`cout << iter->str();  // ok (points to both r and s)`
	`s = "the lazy dog";  // A: invalidates iter`
	`cout << iter->str();  // ERROR, iter was invalidated`
	`// by 's =' (line A)`

1.1.4 Function calls

Finally, since every function is analyzed in isolation, we have to have some way of reasoning about function calls when a function call returns a Pointer. If the user doesn't annotate otherwise, **by default we assume that a function returns values that are derived from its arguments**. For example, given:

```
template<typename T>
const T& min(const T& a, const T& b);
```

by default, `pset(min)` = `pset(a)` ∪ `pset(b)` — that is, the Pointer returned from `min` (here, a `T&`) refers to one of the two input `T&`s.

Or, given:

```
string_view foo(const string& s1, const string& s2);
```

by default, unless otherwise annotated, `pset(foo)` = `{s1', s2'}` — that is, the Pointer returned from `foo` (here, a `string_view`) is derived from the inputs (here, the two `string` Owner parameters).

When compiling the callee, the callee assumes inputs are valid and enforces the assumptions on the outputs' lifetimes. In this example, if the body of `foo` tries to return something different on some path, the programmer gets a compile-time error when `foo` is compiled. For example, this would be an error:

```
string_view foo(const string& s1, const string& s2) {
    static string local = get_a_value();
    if(sometimes()) local = get_another_value();
    return local;            // error, pset {local'} is not a subset of {s1',s2'}
}
```

because it would allow code like this:

```
string_view s = foo("x","y");
string_view s2 = foo("x","y");      // could silently invalidate s
print(s);                           // s could now be dangling
```

When compiling each caller, the caller ensures the inputs are valid and assumes the resulting outputs' lifetimes. Each caller knows the psets of the inputs, and can assume the pset of the output based on that. In this example call site, if we pass two temporary `string`s, the caller knows that the returned `string_view` has a lifetime derived from the arguments, whose lifetimes it knows:

```
// godbolt.org/z/1iOvAO

string_view sv;
```

```
sv = foo("tmp1", "tmp2");     // C: in: pset(arg1) = {__tmp1'}, pset(arg2) = {__tmp2'}
                              // ... so assume: pset(sv) = {__tmp1', __tmp2'}  [the union]
                              // ... and then at the end of this statement, __tmp1 and
                              //     __tmp2 are destroyed, so pset(sv) = {invalid}
cout << sv[0];                // D: error: 'sv[0]' is illegal, s was invalidated when
                              //   '__tmp1' went out of scope on line C (path: C→D)
```

1.2 Guidance: What we teach users

This approach aims for low false positives in "good" code that follows modern C++ conventions, including:

- Prefer to use RAII Owner types (e.g., `shared_ptr`, `vector`) to own other objects. Whenever allocating a new object or raw resource, pass it to an object of such a type that will then own it. (See Guidelines R.1 and R.12.)
- Avoid using unencapsulated *owning* raw `*` pointers that require explicit `delete`, and similar patterns such as raw handles that require explicit `free` calls. (See Guidelines R.3 and R.11.)
- Prefer to use non-owning Pointer types (e.g., `int*`, `string_view`, `span`) locally, as parameters or local variables. Then this analysis will be able to diagnose common cases of dangling for you at compile time. (In cases where you legitimately need to have a non-local Pointer, such as a heap-allocated `vector<string_view>`, that's still okay but will not get much compile-time support to diagnose use-after-free dangling bugs.)
- If you write a custom Owner or Pointer class, it will usually be automatically recognized as an Owner or Pointer. If it is not, write a single annotation on its class declaration to tag it as a `[[gsl::Owner]]` or `[[gsl::Pointer]]` type, respectively.

Note that most of this is what we already teach as well-written modern C++.

1.3 Goals and scope

A primary goal is to define a simple analysis that is easy for compilers to implement efficiently and programmers to reason about clearly. The analysis is local each to function (no whole program analysis), linear and single-pass (suitable for implementation in a C++ front-end compiler rather than a separate expensive analysis step), and directly corresponds to the visible structure of the source code (linearly follows the function's nested blocks).

This paper focuses on the following as target "common cases":

- Local variables (stack variables, parameters, and return values) and local analysis of each individual function's body, composed across opaque function calls by using defaulted (mostly) or annotated (occasionally) lifetime semantics for parameter/return values that are Owners and Pointers.
- Single-level Pointers (e.g., `*`, `&`, `span<widget>`, `string_view`, `vector<int>::iterator`). Multi-level Pointers (e.g., `**`, `span<widget>*`, `vector<span<int>>::iterator`) are also supported but get more conservative invalidation results.

This excludes the following:

- We do not attempt to check after the end of `main`, during static destruction.
- We do not attempt to address all aliasing cases or make concurrency safety guarantees. Programmers are still responsible for eliminating race conditions.

- We do not currently attempt to check of the internals of Owner types, including that we do not attempt to validate the correctness of pointer-based data structures. Each Owner type is assumed to be correctly implemented, and we check the common uses which greatly outnumber the Owner's own functions. In this version, implementations are expected to skip analysis member functions of Owner types.

1.4 Design principles

Note These principles apply to all design efforts and aren't specific to this paper. Please steal and reuse.

The primary design goal is conceptual integrity [Brooks 1975], which means that the design is coherent and reliably does what the user expects it to do. Conceptual integrity's major supporting principles are:

- **Be consistent:** Don't make similar things different, including in spelling, behavior, or capability. Don't make different things appear similar when they have different behavior or capability. — This proposal does not invent special rules that are specific to only raw pointers or specific smart pointers, but uses the same rules for all Pointer types so that the same errors diagnosed for a dangling `T*` are also diagnosed consistently for a dangling iterator, `string_view`, `span`, or other user-defined Pointer type.
- **Be orthogonal:** Avoid arbitrary coupling. Let features be used freely in combination. — This proposal allows arbitrary Owner and Pointer types to be used in combination. This proposal also follows the [C++ Core Guidelines] guidance to permit the same `[[gsl::suppress(`*tag*`)]]` portable warning suppression syntax that works for all Guidelines warnings.
- **Be general:** Don't restrict what is inherent. Don't arbitrarily restrict a complete set of uses. Avoid special cases and partial features. — This proposal treats all non-`const` operations on Owner types as potentially invalidating by default, and does not rely on recognizing a specific preset list of "known" operators.

These also help satisfy the principles of least surprise and of including only what is essential, and result in features that are additive and so directly minimize concept count (and therefore also redundancy and clutter).

Additional design principles include: Make important things and differences visible. Make unimportant things and differences less visible. —This proposal makes deliberate violations of Lifetime checks visible in source code, so that if the code actually fails, then during debugging the programmer can grep and audit the places where `[[gsl::suppress(lifetime` occurs, unlike today where common lifetime-unsafe operations are invisible and compile silently. However, this proposal also intentionally selects defaults that fit the most common cases so that visible annotation is unnecessary most of the time.

1.5 Acknowledgments and implementation notes

Thank you especially to Neil MacIntosh, Kyle Reed, and Gábor Horváth for the Visual C++ extension implementation, and to Matthias Gehre and Gábor Horváth for the Clang-based compiler implementation (linked to throughout this paper), and their comments and feedback including results of running this analysis against large commercial projects.

Note Both are partial in-progress implementations that implement many of the core examples, but parts of this paper are not yet implemented in both or either. For example, null checking is in Clang but not VC++ as of this writing.

Thank you also to all of the following for their comments and feedback on these ideas and/or drafts of this paper: Andrei Alexandrescu, Steve Carroll, Pavel Curtis, Gabriel Dos Reis, Joe Duffy, Daniel Frampton, Anna Gringauze, Chris Hawblitzel, Nicolai Josuttis, Leif Kornstaedt, Aaron Lahman, Ryan McDougall, Nathan Myers, Gor Nishanov, Andrew Pardoe, Jared Parsons, Dave Sielaff, Richard Smith, Jim Springfield, and Bjarne Stroustrup.

1.6 Revision history

R1: Version 1.1 (November 2010). Major changes: Added SharedOwner concept. Added "Diagnostic sensitivity options" section to define a way to start with zero-false-positive warnings and increasingly opt into additional diagnostics. Changed function annotation syntax to contract-like `pre(cond)` and `post(cond)` syntax, and added that they are implicitly inherited by virtual overrides. Renamed `static` to `global` to reduce confusion with the keyword. Refined default lifetimes semantics of Pointer-to-Pointer parameters.

R0: Version 1.0 (September 2018). Major changes: Made the analysis rules rigorous, generalized Owner and Pointer type deduction to use requirements/concepts from the standard and Ranges TS, generalized move support to include diagnosing common use-after-move cases.

Initial incomplete draft: "Lifetimes I and II – v0.9.1": Presented with live prototype demos at CppCon 2015.

2 Proposal

2.1 Type categories: SharedOwner, Owner, Pointer, Indirection, Value, Aggregate

A **SharedOwner** shares ownership of another object (cannot dangle). A SharedOwner type is expressed using the annotation `[[gsl::SharedOwner(DerefType)]]` where `DerefType` is the owned type (and `(DerefType)` may be omitted and deduced as below). For example:

```
template<class T> class [[gsl::SharedOwner(T)]] my_refcounted_smart_pointer;
```

The following standard or other types are treated as-if annotated as Owners, if not otherwise annotated:

- Every type that provides unary `*` and has a user-provided destructor and has a copy constructor and copy assignment operator. (Example: `shared_ptr`.) DerefType is the ref-unqualified return type of `operator*`.
- Every type that has a data member or public base class of a SharedOwner type.

Additionally, for convenient adoption without modifying existing standard library headers, the following well-known standard types are treated as-if annotated as SharedOwners: `shared_future<T>` with DerefType `T`.

An **Owner** uniquely owns another object (cannot dangle). An Owner type is expressed using the annotation `[[gsl::Owner(DerefType)]]` where `DerefType` is the owned type (and `(DerefType)` may be omitted and deduced as below). For example:

```
template<class T> class [[gsl::Owner(T)]] my_unique_smart_pointer;
```

The following standard or other types are treated as-if annotated as Owners, if not otherwise annotated and if not SharedOwners:

- Every type that satisfies the standard Container requirements and has a user-provided destructor. (Example: `vector`.) DerefType is `::value_type`.
- Every type that provides unary `*` and has a user-provided destructor. (Example: `unique_ptr`.) DerefType is the ref-unqualified return type of `operator*`.
- Every type that has a data member or public base class of an Owner type.

Additionally, for convenient adoption without modifying existing standard library headers, the following well-known standard types are treated as-if annotated as Owners: `stack`, `queue`, `priority_queue`, `optional`, `variant`, `any`, and `regex`.

A **Pointer** is not an Owner and provides indirect access to an object it does not own (can dangle). A Pointer type is expressed using the annotation `[[gsl::Pointer(DerefType)]]` where `DerefType` is the pointed-to type (and `(Dereftype)` may be omitted and deduced as below). For example:

```
template<class T> class [[gsl::Pointer(T)]] my_span;
```

The following standard or other types are treated as-if annotated as Pointer, if not otherwise annotated and if not Owners:

- Every type that satisfies the standard Iterator requirements. (Example: `regex_iterator`.) DerefType is the ref-unqualified return type of `operator*`.

- Every type that satisfies the Ranges TS Range concept. (Example: `basic_string_view`.) DerefType is the ref-unqualified type of `*begin()`.
- Every type that satisfies the following concept. DerefType is the ref-unqualified return type of `operator*`.

```
template<typename T>
concept TriviallyCopyableAndNonOwningAndDereferenceable =
    std::is_trivially_copyable_v<T> &&
    std::is_copy_constructible_v<T> &&
    std::is_copy_assignable_v<T> &&
    requires(T t) { *t; };
```

- Every closure type of a lambda that captures by reference or captures a Pointer by value. DerefType is `void`.
- Every type that has a data member or public base class of a Pointer type.

Additionally, for convenient adoption without modifying existing standard library headers, the following well-known standard types are treated as-if annotated as Pointers, in addition to raw pointers and references: `reference_wrapper`, and `vector<bool>::reference`.

> **Note** Pointers follow copy semantics, and if there is a move from a Pointer we assume it does not change the source. As a QoI issue, we could warn if the user opts-in to being a Pointer a type that has user-declared move operations.

A SharedOwner is-an Owner and is-a Pointer.

An **Indirection** is a type that is a Pointer, or an Owner. For an `Indirection` type `I`, `DerefType(I)` is:

- If `I` has a DerefType `R` that is user-specified or is part of automatic deduction as an Owner or Pointer as specified above, then `R`.
- Otherwise, if `I` has unary `operator*` with return type `R&`, then `R`. (Note: This includes raw pointer of type `R*`, and raw reference of type `R&` which is treated as `R*` in this analysis.)
- Otherwise, if `I` has `operator->` with return type `R*`, then `R`.
- Otherwise, if `I` has `operator[]` with return type `R&`, then `R`.
- Otherwise, if `I` has `begin()` with return type `R` where `R` is a Pointer, then `DerefType(R)`.
- Otherwise, if `I` is declared as a template with template parameter list `<typename R /*, ...*/>`, then `R`.
- Otherwise, an unspecified “unknown” type.

For example, `DerefType(unique_ptr<int>)` is `int`, and `DerefType(variant<int,string>)` is unknown.

An **Aggregate** is a type that is not an Indirection and is a class type with public data members none of which are references (`&` or `&&`) and no user-provided copy or move operations, and no base class that is not also an Aggregate. The **elements** of an Aggregate are its public data members.

> **Notes** Aggregates are types we will “explode” (consider memberwise) at local scopes, because the function can operate on the members directly. They are similar to the standard “aggregate” concept but simpler; for this analysis we just need a name for types that have public data and default copy/move.

In the future we could consider fixed-length arrays as well, but the difference for arrays is that whereas a K-member struct will always have its members accessed with names that are known statically (e.g., `s.m1`, `s.m2`), a K-element array can have its elements accessed not only with statically known names (e.g., `a[0]`, `a[1]`) but also with statically unknown names (e.g., `a[i]`).

A **Value** is a type that is neither an Indirection nor an Aggregate.

2.2 Local variables and control flow

"**Function entry**" means the beginning of the function, including the mem-init-list for constructors. "**Function return**" means a `return` statement or the implicit `return` at the end of the function body. A "**local variable**" means any parameter, local variable, member variable (if in a member function), or temporary object in the scope of a function definition; for a local variable of Aggregate type each element is treated as a distinct variable; and for each lambda function defined in the function body each data member (capture) is treated as a distinct variable. A "**local Pointer**" is a local variable of Pointer type. A "**local Owner**" is a local variable of Owner type.

Notes Local struct members are treated as individual variables because that's what they are to the function body, which accesses them directly and individually. For example, this code

```
void test() {
    struct { int* p1; int* p2; int i; } s;    // will be exploded
```

is treated as if it had been written using five distinct local variables

```
void test() {
    int* s__p1;                               // exploded out
    int* s__p2;
    int  s__i;
```

I don't currently include uncaught exceptional paths in "function exit" because they do not result in any communication of valid Pointer results to the caller, so there's no postcondition to enforce.

A function's "**acyclic control flow graph (ACFG)**" means the function's normal local CFG minus backward edges, which are removed by simplifying all `for`/`while`/`do` loops as described in §2.4.9, simplifying all try/catch blocks as described in §2.4.10, and ignoring `continue` and backward `goto` statements. A "**node**" has its usual meaning in a CFG except that each subexpression that mentions a local variable (including its declaration or destruction) or any global variable of Pointer or Owner type is treated as a distinct node (rather than combined in larger basic blocks); every initialized local variable has at least three nodes, for its declaration, initialization, and destruction, respectively. A "**`noexcept` node**" is a node corresponding to a subexpression that is unconditionally `noexcept` by language rule (e.g., built-in integer assignment) or by function declaration. A "**not-`noexcept` node**" is a node that is not a `noexcept` node (this includes an explicit `throw` statement). A "**path**" is a connected directed path in a function's ACFG.

"**Use**" of a local variable `var` means a node corresponding to a mention of `var`'s name that accesses `var`'s value (not its declaration, not just taking its address). A "**`const` use**" means a use that only reads `var` (e.g., a read of a fundamental type, calling a function that treats its `var` argument as `const`) or using `var` as an argument to a function parameter explicitly annotated as `[[gsl::lifetime_const]]` (to annotate the `this` parameter of a function the attribute is in the same position as `override`). A "**non-`const` use**" means a use that is not a `const`

use (e.g., a write or read-modify-write of a fundamental type, or passing `var` to a function that takes it via reference to non-`const` as a function output). "Copy from" and "copy to" have their usual meanings. "**Move from**" `var` means to use `var` as an argument to a `&&` (rvalue reference or forwarding reference) parameter (this includes but is not limited to the parameter of a move constructor or move assignment operator). "**Move to**" `var` means to use `var` as the argument to the `this` parameter of a move constructor or move assignment operator.

2.3 Points-to map (`pmap`) and Points-to set (`pset`)

Compilation of a function definition maintains a **`pmap`** ("points-to map") for each ACFG path. At each node (subexpression) in the function body source code, the path's `pmap` contains:

- one tuple `(x, pset(x))` for every local variable `x` currently in scope, where the set `pset(x)` denotes what `x` could refer to at that point in the source code (e.g., for a Pointer, what it can point to); and
- special entries, notably `(global, {global})`, `(invalid, {invalid})`, `(null, {null})` and `(nullptr, {null})`.

Each element in a `pset(x)` set is one of the following:

	Meaning	A pset entry is invalidated
`var`	p currently refers to local variable `var`	when `var` is destroyed
`global`	p currently refers to a static variable, or an object owned directly by a `const` static Owner object	never (until the end of `main`; may inject additional checking after `main` ends)
`o'`	p currently refers to an object owned directly by Owner `o`	when `o` is destroyed or modified by non-`const` use
`o''` *(etc.)*	p currently refers to an object owned by an object that is owned by `o`	when `o'` is modified by non-`const` use
`null`	p is invalid for any use until tested to be not equal to the null pointer constant	when used without removing `null` from the list via a not-null branch
`invalid`	p is already invalid	always

The set entries are interpreted as "or'd." For example, for a Pointer `p`, `pset(p) == {a,null}` denotes that `p` either points to the object `a` or is null. A variable `x` is said to be:

- "**possibly null**" if `pset(x)` contains `null`; and
- "**invalid**" if `pset(x)` contains `invalid` (and putting `invalid` into `pset(x)` is said to **invalidate** `v`).

QoI As a quality of implementation (QoI) matter, implementations are encouraged, but not required, to track sources of `null` and `invalid` by extending the `pmap` tuples to contain two additional sets for each `p`: `null_history(p)` and `invalid_history(p)`. This enables higher quality diagnostics with source line attributions identifying paths that lead to possibly null and invalid Pointers. Later "QoI" notes refer to these additional sets. The "(line #)" sample diagnostic messages refer to these histories.

`pset(p) = rhs` means to change `pmap` by removing any existing tuple `(p, *)` and inserting the tuple `(p, rhs)`.

`pset(`*list*`)` means `⋃ pset(e) | e ∈` *list*, the union of the psets of each element in *list*.

`pset(p)'` means `{ e' | e ∈ pset(p)` where e is an `Owner}`, to add `'` to each Owner mentioned in the list.

`pset(o')` is `{o''}`, `pset(o'')` is `{o'''}`, etc.

For a non-local Pointer g, there is no pmap entry. Whenever pset(g) is mentioned, it means {global}.

KILL(x) means to replace all occurrences of x and x' and x'' (etc.) in the pmap with invalid. For example, if pmap is {(p1,{a}), (p2,{a'})}, KILL(a') would invalidate only p2, and KILL(a) would invalidate both p1 and p2.

> **QoI** For each p for which pset(p) that was modified by the KILL, additionally append the current node to invalid_history(p) to record the source location that caused the invalidation.

JOIN({pmap$_1$,...,pmap$_n$}) means { (p, pset$_1$(p) ∪ ... ∪ pset$_n$(p) | (p,*) ∈ pmap$_1$ ∪ ... ∪ pmap$_n$ }.

For two psets from and to, from is "**substitutable for**" to if from is no more invalid than to, no more null than to, and no more invalidatable than to. If from is {global}, it is substitutable for any to. Otherwise, from is substitutable for to if all of the following must be true, where o is an Owner:

- If from includes invalid, then to must include invalid.
- If from includes null, then to must include null.
- If from includes o', then to must include o' or o.
- If from includes o'' (etc.), then to must include o'' or o' or o (etc.).
- If to is {global}, then from must be {global}.

For example:

- {a} (points to a, invalidated by the end of lifetime of a) is substitutable for {a,b} (points to either a or b, invalidated by the end of lifetime of either a or b).
- {global} is substitutable for {global,null}.
- {o,o2'} is substitutable for {o',o2'}.

In the following examples:

- green highlights legal uses of valid Pointers;
- x highlights the point at which an invalidation occurs; and
- red highlights subsequent illegal uses of invalidated Pointers (when those are allowed to be formed).

2.4 Analysis rules by node type

2.4.1 Function entry and exit

On function entry, pmap contains the special entries (global, {global}), (invalid, {invalid}), (null, {null}) and (nullptr, {null}) and one entry for each parameter as described in §2.5.

On function exit, pmap is discarded after performing the checks described in §2.5.

2.4.2 Local variable declaration, use, and scope

"Declaration" means the explicit declaration of a named variable, the implicit declaration of an rvalue, the explicit or implicit appearance of a member variable in the mem-init-list (if in a constructor), or a Pointer value created by a type-unsafe cast (in which case the source of the cast is ignored and not treated as an initializer).

> **Note** When the type safety profile is also enforced, then the last case (type-unsafe casting) cannot happen unless the type safety rule is explicitly [[suppress]]-ed. As a possible extension (which as of this writing is partly implemented in the MSVC implementation), we could additionally consider giv-

ing useful Pointer lifetime guarantees for even such type-unsafe code that performs pointer/reference casting; for example, when casting from one Pointer type to another, we could optimistically copy the pset even though the types do not match, as if it were a Pointer copy. This gives useful tracking in the case of casting `byte*` to `SomeLayout*` which actually are equivalent for Lifetime purposes (just not for Type profile purposes).

When a non-parameter non-member Pointer `p` is declared, add `(p, {invalid})` to `pmap`. If the declaration includes an initialization, the initialization is treated as a separate operation; for example, `int* p = expr;` is treated as `int* p` /*= uninitialized*/`;` (governed by this section) followed by `p = expr;` (copying governed by §2.4.4). If the initialization is default initialization or zero initialization, set `pset(p) = {null}`; for example, `int* p;` results in `pset(p) == {invalid}`, whereas `int* p{};` or `string_view p;` results in `pset(p) == {null}`.

When a local Value `x` is declared, add `(x, {x})` to `pmap`.

When a local Owner `x` is declared, add `(x, {x__1'})` to `pmap`.

Note In a member function of an Owner type, this includes the generated `deref__this` (see §2.5).

On `co_await` or `co_yield`, for every Pointer parameter `p`, KILL(`p`). This makes `p` invalid if it still contains its initial value. For coroutines, the assumption that an argument outlives the call holds until a `co_await` or `co_yield`.

On every non-`const` use of a local Owner `o`:

- For each entry `e` in `pset(s)`: Remove `e` from `pset(s)`, and if no other Owner's pset contains only `e`, then KILL(`e`).
- Set `pset(o) = {o__N'}`, where `N` is one higher than the highest previously used suffix. For example, initially `pset(o)` is `{o__1'}`, on o's first non-const use `pset(o)` becomes `{o__2'}`, on o's second non-const use `pset(o)` becomes `{o__3'}`, and so on.

Note The suffix `__N` designates to the `N`th object owned by o. This is primarily useful for SharedOwners, but doesn't hurt (and may be useful) to track also for unique Owners.

On every dereference of a Pointer `p`, enforce that `p` is not invalid:

Rule **(lifetime.1)** It is an error to dereference an invalid Pointer.

On every non-`const` use of a non-Pointer `x` that has a precondition (e.g., passing it to a function with a `[[gsl::pre()]]` contract that includes a non-`const` use of `x`), enforce that `x` is not invalid:

Rule **(lifetime.1a)** It is an error to use a moved-from object.

When a local variable `x` goes out of scope, KILL(`x`) and remove `(x, *)` from `pmap`.

Note For a lifetime-extended temporary, as usual it goes out of scope at the end of the lifetime of the reference rather than at the end of the full-expression.

2.4.2.1 Example

For example:

```
// godbolt.org/z/g9C8G6

void f(string param) {
    int* p;                         // A: pset(p) = {invalid}
    *p = 1;                         // ERROR (lifetime.1), p is invalid (line A)
```

```
    int** pp;                       // B: pset(pp) = {invalid}
    *pp = &p;                       // ERROR (lifetime.1), pp is invalid (line B)

    if (cond) {
        // ...
        int x;
        // ...
    }                               // KILL(x) – invalidates any Pointer to x

    while (cond) {
        // ...
        string s;
        // ...
        s = param;                  // KILL(s') – invalidates any Pointer to s's data
                                    //   (the non-const operation could move s's buffer)
        // ...
    }                               // KILL(s) – invalidates any Pointer to s
    // ...
}                                   // KILL(param) – invalidates any Pointer to param
```

2.4.3 Unary & (address-of)

Taking the address of a global variable, or public data member or array element thereof, creates a temporary `tmp` that points to a variable whose duration lasts longer than this function, and so sets `pset(tmp) = {global}`.

Taking the address of a local variable `x`, or public data member or array element thereof, creates a temporary `tmp` that points to `x`, and so sets `pset(tmp) = {x}`.

2.4.3.1 Example

For example:

```
static int i = 0;

void f() {
    &i;                             // pset(tmp1) = {global}
    int j = 1;
    &j;                             // pset(tmp2) = {j}
}
```

2.4.4 Pointer arithmetic

The Lifetime safety profile requires the following rule from the Bounds safety profile:

> **Rule** **(bounds.1)** Raw pointer arithmetic is not allowed. For a raw pointer `p` and some other variable `x`, the following are errors: `p+x`, `x+p`, `p[x]`, `p+=x`, `++p`, `p++`, `p-x`, `x-p`, `p-=x`, `--p`, `p--`.

Any of these expressions that results in a pointer sets that pointer's pset to `{invalid}`. For example:

```
int x;
int* p = &x;
*p = 1;                         // ok

int* q = p+1;                   // ERROR (bounds.1), pointer arithmetic is not allowed
                                //   pset(q) = pset(p+1) which is {invalid}

*q = 1;                         // ERROR (lifetime.1), q is invalid (line A)
```

2.4.5 Copying and moving

When an Aggregate `a` is copied or moved from `rhs`, the semantics are to copy or move `a.m` from `rhs.m` for each element `m` of `a`.

When a Pointer `p` is copied from `rhs`, set `pset(p) = pset(rhs)`. The Pointer copy is an error if `p` has type `not_null<>` and `pset(rhs)` contains `null`. The Pointer copy is an error if `p` is non-local and `pset(rhs)` contains values other than `null` and/or `global`. If `rhs` has no known pset (e.g., is derived from a parameter, in examples like `p = p->next;`) assume it is `{global}` (that is, assume `rhs` is valid).

Rule	**(lifetime.1b)** It is an error to copy a possibly-null Pointer to a Pointer `p` that is declared `not_null`.
Rule	**(lifetime.1c)** It is an error to copy a Pointer that could refer to something other than `null` or `global` to a non-local Pointer.
QoI	When copying the `pset`, also copy both histories and add the current node to the history if propagating `null` or `invalid` respectively: Copy `null_history(p) = null_history(rhs)`, then if `pset(rhs)` contains `null` additionally append the current node to `null_history(p)`. Copy `invalid_history(p) = invalid_history(rhs)`, then if `pset(rhs)` contains `invalid` additionally append the current node to `invalid_history(p)`.
Note	Binding a reference `T& r = rhs;` is treated as `T* r = &rhs;` — i.e., declaring `r`, taking the address of `rhs`, and copying `pset(tmp)` to `pset(r)`.

When a Pointer `p` is copied from `rhs`, and `p` is `const` (such as a reference) or a non-local variable, `rhs` must not be invalid.

Rule	**(lifetime.2)** It is an error to copy an invalid Pointer to a Pointer `p` that is const (such as a reference) or a non-local variable.
Note	Normally these rules diagnose *dereferencing* an invalid Pointer, not *forming* one, because this allows local Pointer variable reuse. For a `const` Pointer, such as a reference, we should make even creating the unusable Pointer be a build-time error, because the Pointer is unusable and cannot be rebound to make it usable, so there is no reason to allow this.

When an Owner `o` is moved from another Owner `rhs`, the ownership moves from `rhs` to `o`, and so in all psets replace `rhs` with `o`.

Note	This is generally useful for STL containers and smart pointers, but will give false negatives (fail to warn on invalidations) for an Owner type whose move does not fully transfer ownership of all internally owned state, such as `std::string` when it performs the small string optimization (for small strings, the buffer does not move to the new object). In the future we can add an annotation for such an Owner to opt out, and state that moving from such an object invalidates all Pointers into the moved-from owner object, instead of tracking them as now pointing into the moved-to owner object.

When an Owner `&&` parameter is invoked so that the `&&` has a pset of `{x'}`, the `&&` is bound to `x` and `x`'s data will be moved from, so as a postcondition (after the function call) KILL(`x'`).

When a Pointer p that is not a SharedOwner is moved from, the operation is always treated as a copy.

> **Note** Non-owning Pointers follow copy semantics. If there is a move from a Pointer, it is assumed not to change the source.

When a non-Pointer x is moved from, set pset(x) = {invalid} as a postcondition of the function call.

When an Owner x is copied to or moved to or passed by lvalue reference to non-const to a function that has no precondition that mentions x, set pset(x) = {x'}. When a Value x is copied to or moved to or passed by lvalue reference to non-const to a function that has no precondition that mentions x, set pset(x) = {x}. These pset resets are done before processing the function call (§2.5) including preconditions.

2.4.5.1 Example

For example:

```
// godbolt.org/z/G3BCOR

void f() {
    int* p = nullptr;               // A: pset(p) = pset(nullptr) which is {null}
    *p = 1;                         // ERROR (lifetime.1), p is null (line A)

    int* p2;                        // pset(p2) = {invalid}

    {
        struct { int i; } s = {0};
        p = &s.i;                   // pset(p) = pset(temp) = {s.i}, p now points to s.i
        p2 = p;                     // pset(p2) = {s.i}, p2 now also points to s.i

        *p = 1;                     // ok
        *p2 = 1;                    // ok
    }                               // B: KILL(i)
                                    //   → pset(p) = {invalid} and pset(p2) = {invalid}

    *p = 1;                         // ERROR (lifetime.1), p was invalidated when s
                                    //   went out of scope (line B)
    *p2 = 1;                        // ERROR (lifetime.1), p2 was invalidated when s
                                    //   went out of scope (line B)

    p = nullptr;                    // C: pset(p) = pset(nullptr) which is {null}

    int x[100];
    p2 = &x[10];                    // pset(p2) = {x} – p2 now points to x
    *p2 = 1;                        // ok

    p2 = p;                         // D: pset(p2) = pset(p) which is {null}
    *p2 = 1;                        // ERROR (lifetime.1), p2 is null (lines C→D)

    p2 = &x[10];                    // pset(p2) = {x} – p2 now points to x again
    *p2 = 1;                        // ok

    int** pp = &p2;                 // pset(pp) = {p}
    *pp = p;                        // ok
}
```

2.4.5.2 Example: Owner move

For example:

```
vector<int> v1(100);
int* pi = &v1[0];                   // pset(pi) = {v1'}
auto v2 = std::move(v1);            // pset(pi) = {v2'} – note, no KILLs here
```

2.4.5.3 Example: && parameter

For example:

```
// godbolt.org/z/x26Fsg

void consume(vector<int>&&);

void test() {
    vector<int> v(1000);
    auto iter = v.begin();          // pset(iter) = {v'}
    consume(std::move(v));          // A: pset(iter) = {invalid}, pset(v) = {invalid}
    *iter;                          // error, iter was invalidated on line A
    v[100];                         // error, v is moved-from and [] has a precondition
}
```

2.4.6 Local Pointer dereferencing

When a Pointer `p` is dereferenced using unary `*` or `->`, if `pset(p)` contains more than one location, processing of the rest of the expression continues distributively with respect to the locations. If a multi-location Pointer value is copied to, the effect is of appending (rather than replacing) the psets. For example:

```
int *p1 = ..., *p2 = ...;
int *pp = cond ? &p1 : &p2;         // pset(pp) = {p1,p2}
int x;
*pp = &x;                           // appends {x} to pset(p1) and pset(p2)
```

When a local Pointer `p` is dereferenced using unary `*` or `->` to create a temporary `tmp`, then if `pset(pset(p))` is nonempty, set `pset(tmp) = pset(pset(p))` (i.e., `tmp` is itself a Pointer). Otherwise, set `pset(tmp) = {tmp}`.

QoI If `pset(tmp)` is nonempty, then also copy the `null_history` and `invalid_history` and add the current node to the respective history if `pset(tmp)` contains `null` or `invalid`.

This tracks the second-level points-to set of `p` when `p` is a Pointer to Pointer. For example, given this `pmap`:

```
{
    ( p,   {x, y}    ),             // p points to x or y
    ( x,   {a, b, c} ),             // x points to a, b, or c
    ( y,   {null}    )              // y points to null (nothing)
}
```

and that we are processing an ACFG node for the expression `*p`, we insert a new `pmap` tuple:

```
    ( tmp, {a, b, c, null} )        // tmp points to a, b, c, or null
```

2.4.6.1 Example

For example:

```
// godbolt.org/z/KzfY4B
```

```
int* p1 = nullptr;
int* p2 = nullptr;
int* p3 = nullptr;
int* p4 = nullptr;

{
    int   i = 0;

    int&  ri = i;                   // pset(ri) = {i}
    p1 = &ri;                       // pset(p1) = {i}
    *p1 = 1;                        // ok

    int*  pi = &i;                  // pset(pi) = {i}
    p2 = pi;                        // pset(p2) = {i}
    *p2 = 1;                        // ok

    int** ppi = &pi;                // pset(ppi) = {pi}
    **ppi = 1;                      // ok, modifies i

    int* pi2 = *ppi;                // pset(pi2) = pset(pset(ppi)) which is {i}

    p3 = pi2;                       // pset(p3) = {i}
    *p3 = 1;                        // ok

    {
        int   j = 0;
        pi = &j;                    // pset(pi) = {j}  (note: so *ppi now points to j)
        pi2 = *ppi;                 // pset(pi2) = pset(pset(ppi)) but which is now {j}

        **ppi = 1;                  // ok, modifies j
        *pi2 = 1;                   // ok
        p4 = pi2;                   // pset(p4) = {j}
        *p4 = 1;                    // ok
    }                               // A: KILL(j) → pset(pi) = {invalid},
                                    //    pset(pi2) = {invalid}, and pset(p4) = {invalid}

    **ppi = 1;                      // ERROR (lifetime.1), *ppi was invalidated when j
                                    //    went out of scope (line A)
    *pi2 = 1;                       // ERROR (lifetime.1), pi2 was invalidated when j
                                    //    went out of scope (line A)
    *p4 = 1;                        // ERROR (lifetime.1), p4 was invalidated when j
                                    //    went out of scope (line A)
    *p3 = 1;                        // ok
    *p2 = 1;                        // ok
    *p1 = 1;                        // ok

}                                   // B: KILL(i) → pset(p3) = {invalid},
                                    //    pset(p2) = {invalid}, and pset(p1) = {invalid}

*p3 = 1;                            // ERROR (lifetime.1), p3 was invalidated when i
                                    //    went out of scope (line B)
*p2 = 1;                            // ERROR (lifetime.1), p2 was invalidated when i
                                    //    went out of scope (line B)
*p1 = 1;                            // ERROR (lifetime.1), p1 was invalidated when i
                                    //    went out of scope (line B)
```

Note that `ppi` is never modified after initialization, and so its pset records that it always points only to `pi`. However, we track that `*ppi` points first to `i` and then to `j` through the changes to `pset(pi)`.

2.4.6.2 Example: Invalidation by modifying Owners

For example:

```
// godbolt.org/z/UE-Mb0

auto s = make_shared<int>(0);
int* pi3 = s.get();                 // pset(pi3) = {s'} [more on this later]
s = make_shared<int>(1);            // A: KILL(s') → pset(pi3) = {invalid}
*pi3 = 42;                          // ERROR, pi3 was invalidated by
                                    //        assignment to s on line A

// Chris Hawblitzel's example
auto  sv  = make_shared<vector<int>>(100);
shared_ptr<vector<int>>* sv2 = &sv; // pset(sv2) = {sv}
vector<int>* vec = &*sv;            // pset(vec) = {sv'}
int* ptr = &(*sv)[5];               // pset(ptr) = {sv''}

*ptr = 1;                           // ok

                                    // track pset of:    sv2     vec     ptr
                                    //                   -----   -----   -----
                                    //             IN:   sv      sv'     sv''
vec->                               // same as "(*vec)." → pset(*vec) == {sv''}
     push_back(1);                  // KILL(sv'') because non-const operation
                                    //            OUT:   sv      sv'     invalid

*ptr = 3;                           // ERROR, invalidated by push_back

ptr = &(*sv)[5]; // back to previous state to demonstrate an alternative...

*ptr = 4;                           // ok

                                    //             IN:   sv      sv'     sv''
(*sv2).                             // pset(*sv2) == {sv'}
       reset();                     // KILL(sv') because non-const operation
                                    //            OUT:   sv      invalid invalid

vec->push_back(1);                  // ERROR, invalidated by reset
*ptr = 3;                           // ERROR, invalidated by reset
```

Note how the modification of `*sv2` correctly invalidates `ptr` which was obtained via an unrelated path (`sv`).

2.4.6.3 Example: Container of containers

Here is a variation on Chris Hawblitzel's example showing a container of containers.

```
// godbolt.org/z/cAEqzJ

vector<vector<int>> vv;
vector<vector<int>>* vv2 = &vv;     // pset(vv2) = vv
vector<int>* vec = &vv[0];          // pset(vec) = vv'
int* ptr = &(*vec)[5];              // pset(ptr) = vv''

*ptr = 0;                           // ok
```

```
                                    // track pset of:    vv2     vec     ptr
                                    //                   -----   -----   -----
                                    //             IN:   vv      vv'     vv''
vec->                               // same as "(*vec)." → pset(*vec) == {vv''}
     push_back(1);                  // KILL(vv'') because non-const operation
                                    //            OUT:   vv      vv'     invalid

*ptr = 1;                           // ERROR, invalidated by push_back

ptr = &(vv[0])[5]; // back to previous state to demonstrate an alternative...

*ptr = 0;                           // ok

                                    //             IN:   vv      vv'     vv''
vv2->                               // same as "(*vv2)." → pset(*vv2) == {vv'}
     clear();                       // KILL(vv') because non-const operation
                                    //            OUT:   vv      invalid invalid

*ptr = 2;                           // ERROR, invalidated by clear
```

2.4.7 Branches (fork and join nodes)

Every `if` scope begins with a "fork node" that has two outbound ACFG edges (`true` path and `false` path), and ends with a "join node" where the `true` and `false` inbound paths converge (if reachable; that is, unless either or both path unconditionally leads to a function exit before the join).

At each fork node, make a copy of the current `pmap` for each outbound path.

At each join node, `pmap` = JOIN($pmap_{true}$, $pmap_{false}$).

QoI And similarly for `null_history` and `invalid_history`.

An `if` that has no `else` is treated as if it has an empty `else{}` block.

A `switch(`*cond*`)` is treated as if it were an equivalent series of non-nested `if` statements with single evaluation of *cond*; for example, `switch(a){ case 1:/*1*/ case 2:/*2*/ break; default:/*3*/}` is treated as `if(auto& a=a; a==1){/*1*/} else if(a==1||a==2){/*2*/} else{/*3*/}`.

A ternary expression `a?b:c` is treated as if it were an equivalent `if(a){/*b*/}else{/*c*/}` for path purposes. The language-semantic result of the expression is unchanged (it is still an expression that evaluates to either `b` or `c`). For example, `p = cond?p2:p3;` is treated as `if(cond) p=p2; else p=p3;`. For example, `cond?p2:p3 = p;` is treated as `if(cond) p2=p; else p3=p;`. For example, `auto p = cond?p2:p3;` is treated as `auto p = /*uninitialized*/; if(cond) /*construct p from p2*/; else /*construct p from p3*/;`.

2.4.7.1 Example: Invalidation in both branches

Both branches could invalidate. For example:

```
// godbolt.org/z/RjQogH

int* p = nullptr;               // pset(p) = {null}

if(cond) {
    int i = 0;
    p = &i;                     // pset(p) = {i}
    *p = 42;                    // ok
```

```
}                               // A: KILL(i) → pset(p) = {invalid}
else {
    int j = 1;
    p = &j;                     // pset(p) = j
    *p = 42;                    // ok
}                               // B: KILL(j) → pset(p) = {invalid}
// merge → pset(p) = {invalid}

*p = 42;                        // ERROR, p was invalidated when i went out of scope
                                // at line A or j went out of scope at line B.
                                // Solution: increase i's and j's lifetimes, or
                                // reduce p's lifetime
```

2.4.7.2 Example: Invalidation in one branch

Invalidation on only one branch allows the possibility of "*could be* invalidated." For example:

```
// godbolt.org/z/WqK1wT

int* p = nullptr;               // pset(p) = {null}
int i = 0;

if(cond) {
    p = &i;                     // pset(p) = {i}
    *p = 42;                    // ok
}                               // no invalidation
else {
    int j = 1;
    p = &j;                     // pset(p) = {j}
    *p = 42;                    // ok
}                               // A: KILL(j) → pset(p) = {invalid}
// merge → pset(p) = {invalid}

*p = 1;                         // ERROR, p was invalidated when j went out of scope
                                // at line A. Solution: increase j's lifetime, or
                                // reduce p's lifetime

if(cond) *p = 2;        // ERROR, (same diagnostic) even if cond is unchanged
```

Note The case `if(cond) *p;` is still an error because the rules must be portable (they must give the same answer for the same code across implementations without requiring implementations to perform data-dependent reasoning or be omniscient) and the fix is simple in most cases (increase or decrease the lifetime of a specifically named local variable).

2.4.7.3 Example: Invalidation in neither branch

A pointer can be assigned differently on different branches and still be valid after the branches merge. For example:

```
// godbolt.org/z/uXExWo

int* p = nullptr;               // pset(p) = {null}
int i = 0;

{
```

```
    int j = 1;

    if(cond) {
        p = &i;              // pset(p) = {i}
        *p = 42;             // ok
    }                        // no invalidation
    else {
        p = &j;              // pset(p) = {j}
        *p = 42;             // ok
    }                        // no invalidation
    // merge → pset(p) = {i,j}

    *p = 42;                 // ok
}
```

2.4.8 Null/not-null tests and assertions/branches

For a Pointer `p`:

- a `p`-is-not-null-test means an explicit test that `p` is not null by comparing it `!=` to a Pointer `p2` for which `pset(p2) == {null}` only (such as a null pointer constant, `nullptr`) or by using `p`'s conversion to `bool` returning `true`, possibly in combination with other tests using `&&`.
- a `p`-is-null-test means an explicit test that `p` is null by comparing it `==` to a Pointer `p2` for which `pset(p2) == {null}` only (such as a null pointer constant, `nullptr`) or by using `p`'s conversion to `bool` returning `false`, possibly in combination with other tests using `&&`.

When a branch can be entered only on success of a Pointer `p`-is-not-null-test, remove `null` (if present) from `pset(p)` in that branch path; if testing `p`'s not-nullness was the entire condition, additionally set `pset(p) = {null}` in the alternate branch path. For example, `if(p)` and `if(p!=nullptr)` and `if(p!=0 && cond)` remove `null` from `pset(p)` in the `true` branch (and the first two additionally set `pset(p) = {null}` in the `false` branch), but `if(p!=0 || cond)` does not.

When a branch can be entered only on success of a Pointer `p`-is-null-test, set `pset(p) = {null}` in that branch path; if testing `p`'s nullness was the entire condition, additionally remove `null` (if present) from `pset(p)` in the alternate branch path. For example, `if(!p)` and `if(p==nullptr)` and `if(p==0 && cond)` set `pset(p) = {null}` in the `true` branch (and the first two additionally remove `null` from `pset(p)` in the `false` branch), but `if(p==0 || cond)` does not. Additionally, if `p` is a Pointer-to-Pointer parameter, set `pset(deref_p) = { }` in that branch path.

When a Pointer `p`-is-not-null test or a `p`-is-null-test "test" is part of a compound `if` statement condition where the rest will be short-circuited if the test fails (i.e., of the form `if(`*test* `&&` *rest*`)`), the condition is treated as if written `if(`*test*`) if(`*rest*`)`. This implies that `pset(p)` will be adjusted as above for the *rest* expression. For example, given initially `pset(p) == {null,x}`, then performing `if(p && p->x)` is as if written `if(p) if(p->x)`, which is valid because at the point of the second condition `pset(p) == {x}`.

When an `assert` or `[[assert: ]]` can succeed only on success of a Pointer `p`-is-not-null-test, remove `null` (if present) from `pset(p)`. For example, `assert(p)` and `[[assert: p!=nullptr]]` and `assert(p!=0 && cond)` remove `null` from `pset(p)`, but `[[assert: p!=0 || cond]]` does not.

When an `assert` or `[[assert: ]]` can succeed only on success of a Pointer `p`-is-null-test, set `pset(p) = {null}`. For example, `[[assert: !p]]` and `assert(p==nullptr)` and `[[assert: p==0 && cond]]` set `pset(p) = {null}`, but `assert(p==0 || cond)` does not.

2.4.8.1 *Example: Removing `null` from a pset to dereference successfully*

For example, if a Pointer might be null, code can test for non-null and then use the pointer:

```
// godbolt.org/z/4dTX9n
int* p = nullptr;                   // A: pset(p) = {null}
int i = 0;
if(cond) {
    p = &i;                         // pset(p) = {i}
}
// merge: pset(p) = {null,i}
*p = 42;                            // ERROR, p could be null from line A
if(p) {                             // remove null in this branch → pset(p) = {i}
    *p = 42;                        // ok, pset(p) == {i}
}
// here, outside the null testing branch, pset(p) is still {null,i}
```

2.4.8.2 *Example: Replacing `null` in a pset with a valid object*

For example, if a Pointer might be null, code can test for null and replace it with non-null.

```
// godbolt.org/z/Q-4zwP
int i = 0, j = 0;
int* p = cond ? &i : nullptr;       // A: pset(p) = {i, null}
*p = 42;                            // ERROR, p could be null from line A
if(!p) {                            // in this branch, pset(p) = {null}
    p = &j;                         // pset(p) = {j}
}
// NOTE: in implicit “else”, pset(p) = {i}
// merge pset(p) = {j} ∪ {i}
*p = 1;                             // ok, pset(p) = {i,j}, does not contain null
```

2.4.9 Loops

A loop is treated as if it were the first two loop iterations unrolled using an `if`. For example, `for(/*init*/;/*cond*/;/*incr*/){/*body*/}` is treated as `if(/*init*/;/*cond*/){/*body*/;/*incr*/}` `if(/*cond*/){/*body*/}`.

A range-based `for` loop is treated as if it were textually expanded to its semantic definition in ISO C++ [stmt.ranged] in terms of a non-range-based `for` loop, then treated as an ordinary `for` loop as above.

Note There are only three paths to analyze: never taken (the loop body was not entered), first taken (the first pass through the loop body, which begins with the loop entry `pmap`), and subsequent taken (second or later iteration, which begins with the loop body exit `pmap` and so takes into account any invalidations performed in the loop body on any path that could affect the next loop).

This ensures that a subsequent loop iteration does not use a Pointer that was invalidated during a previous loop iteration.

Because this analysis gives the same answer for each block of code (always converges), all loop iterations after the first get the same answer and so we only need to consider the second iteration, and so the analysis algorithm remains linear, single-pass.

As an optimization, if the loop entry `pmap` is the same as the first loop body exit `pmap`, there is no need to perform the analysis on the second loop iteration; the answer will be the same.

2.4.9.1 Example: Loops that do not change psets

Some loops do not change psets used in the loop. For example:

```
p = &a[0];                          // pset(p) = {a}

for( /*...whatever...*/ ) {
    // ...
    if( /*...whatever...*/ ) {
        // ...
        p = &a[i];                  // pset(p) = {a}
        // ...
    }
    // ...
}
merge: pset(p) = {a} /*before loop*/ ∪ {a} /*after loop body*) = {a}

*p;                                 // ok
```

In this case, the set of dependencies on input and output did not change and no further action is needed.

2.4.9.2 Example: Loops that do change psets

If instead `p` could be pointed to another object during the loop, we would take the exit pset and then process the loop exactly one more time treating them as entry dependencies to ensure the loop body did not rely on an invalidatable dependency. For example:

```
p = &a[0];                          // pset(p) = {a}

for( /*...whatever...*/ ) {
    // ...
    if( /*...whatever...*/ ) {
        // ...
        p = &b[i];                  // pset(p) = {b}
        // ...
    }
    // merge → pset(p) = {a,b}
    // ...
}
merge: pset(p) = {a} /*before loop*/ ∪ {b} /*in loop body*) = {a,b}
```

```
// that's different from entry, so process loop one more time with {a,b}:
for( /*...whatever...*/ ) {
    // ...
    if( /*...whatever...*/ ) {
        // ...
        p = &b[i];                  // pset(p) = {b}
        // ...
    }
    // merge → pset(p) = {a,b} again/still
    // ...
}
// pset(p) = {a,b} still
```

2.4.9.3 Example: Loops that invalidate

If the loop body could invalidate, we get a possibly invalid exit dependency:

```
// godbolt.org/z/_midIP

p = &a[0];                          // pset(p) = {a}

for( /*...whatever...*/ ) {
    *p = 42;
    p = nullptr;                    // A: pset(p) = {null}
    // ...
    if( /*...whatever...*/ ) {
        // ...
        p = &b[i];                  // pset(p) = {b}
        // ...
    }
    // merge → pset(p) = {null,b}
    // ...
}
merge: pset(p) = {a} /*before loop*/ ∪ {null,b} /*loop body*/ ={null,a,b}

// that's different from entry, so process loop one more time with {null,a,b}:
for( /*...whatever...*/ ) {
    *p = 42;                        // ERROR, could be null from assignment to p at
                                    // line A in a previous iteration
    p = nullptr;                    // A: pset(p) = {null}
    // ...
    if( /*...whatever...*/ ) {
        // ...
        p = &b[i];                  // pset(p) = {b}
        // ...
    }
    // merge → pset(p) = {null,b}
    // ...
}
// pset(p) = {null,a,b} still
```

2.4.9.4 Example: Loops that allocate (and owner<> vs. unique_ptr)

Some loop bodies allocate:

```
p = &a[0];                          // pset(p) = {a}
bool must_delete = false;

for( /*...whatever...*/ ) {
    // ...
    if( /*...whatever...*/ ) {
        // ...
        p = new A                   // A: pset(p) = {temp'}
                ;                   // KILL(temp) → pset(p) = {invalid}
                                    // ERROR: no delete of owner<> returned from new
        must_delete = true;
        // ...
    }
    // ...
}

if(must_delete)
    delete p;                       // ERROR, delete of non-owner<> is not lifetime-safe
```

Solution: Have more than one pointer. In this case a unique_ptr is appropriate and replaces the explicit flag, so we net out to zero additional variables (and less code since we can omit the explicit fragile delete check).

```
p = &a[0];                          // pset(p) = {a}
unique_ptr<A> up;                   // initially null
for( /*...whatever...*/ ) {
    // ...
    if( /*...whatever...*/ ) {
        // ...
        up = make_unique<int>();
        p = up.get();               // ok, pset(p) = {up'}
        // ...
    }
    // merge: pset(p) = {a, up'}
    // ...
}
merge: pset(p) = {a} /*before loop*/ ∪ {a,up'} /*loop body*/ = {a,up'}

// that's different from entry, so process loop one more time with {a,up'}:
for( /*...whatever...*/ ) {
    // ...
    if( /*...whatever...*/ ) {
        // ...
        p = (up = new A).get();     // ok, pset(p) = {up'}
        // ...
    }
    // merge: pset(p) = {a,up'}
    // ...
```

```
}
// pset(p) = {a,up'} still

if (p)
    *p = 42;                        // ok
```

2.4.10 `try` and `catch`

A `try{/*1*/} catch(/*2*/){/*3*/}` block is treated as if it were a branch of the form `/*1*/ if(`*invented bool condition*`){/*3*/}`. In particular, if control flow could proceed normally out of block 3 because an exception has been handled, the `pmap`s at the end of blocks 1 and 3 are JOINed.

Additionally, within a `try` block, at every non-`noexcept` node *n* take a copy $pmap_n$ of the current `pmap`. Apply any KILL operation within or at the end of the `try` block to all `pmap`s. On entry of the corresponding `catch` block, `pmap` = JOIN($pmap_1$, ..., $pmap_n$).

> **Note** The `catch` block as if it could have been entered from every point in the `try` block where an exception could have been raised. As an optimization, instead of taking full copies of the `pmap`, an implementation could store deltas.

This case is a clear win and we expect this to catch many mistakes.

For an exception caught by reference via `catch(`*cv-qualified* `E& e)`, the reference is treated as a raw pointer and has pset `{global}`.

2.4.10.1 Example: `catch`

For example:

```
int i = 0;
int *p1 = &i, *p2 = p1;

try {
    int j = 0;
    p1 = &j;                        // A: pset(p1) = {j}
    could_throw();
    p1 = &i;                        // pset(p1) = {i}
    could_throw();
    p2 = &j;                        // B: pset(p2) = {j}
    could_throw();
}                                   // KILL(j) → kills j in all pmap copies

catch(...) {                        // join: pset(p1) == pset(p2) == {i, invalid}
    *p1 = 42;                       // ERROR, invalidated by assignment to p1 on line A
    *p2 = 42;                       // ERROR, invalidated by assignment to p2 on line B
}
```

2.4.10.2 Example: `throw`

Unlike a `return`, the type of an `thrown` object cannot be carried through function signatures. Therefore, do not throw a Pointer with lifetime other than `global`. For example:

```
// godbolt.org/z/p_QjCR

static int gi = 0;
```

```
void f() {
    int i = 0;

    throw &i;                       // ERROR
    throw &gi;                      // OK
}
```

2.5 Calling functions

When the analysis sees a function call, apply the following model to simulate an output behavior that is consistent with the Lifetime rules for the callee.

> **Note** All defaults are only for convenience to minimize annotation in the common case. Defaults are never central to the analysis method, and so they can use heuristics.

The following rules are applied consistently when separately analyzing each side of the function call:

- When analyzing the call site, the rules apply to arguments, and the call site enforces preconditions and assumes postconditions.
- When analyzing the callee definition, the rules apply to parameter, and the callee definition assumes preconditions and enforces postconditions. Postconditions are enforced after every parameter `x`'s end-of-scope KILL(`x`) has been done but before `x`'s tuple is removed from `pmap`.

Definitions and general rules:

- **`argument(x)`** means the name at a given call site for the argument to the function input `x` or a function output `x`. For a generated `deref__param` parameter, argument(deref__param) means `*argument(param)` at the call site.
- Treat a parameter `X& param` annotated as `[[gsl::lifetime_const(param)]]` as if declared `const X&`.
- Treat a parameter of forwarding reference type as if annotated `[[gsl::lifetime_const(param)]]`.
- Treat `this` (if present) as if it were declared with `not_null`.
- For each reference parameter, treat the reference as if it were a `not_null` pointer.
- `&&` intentionally means both rvalue references and forwarding references.

2.5.1 Expand parameters and returns: Aggregates, nonstatic data members, and Pointer dereference locations

For each Aggregate parameter `agg` passed by value, `*`, or `&`, treat it as if it were additionally followed by a series of generated element parameters `agg__m` for each element `m` of `agg`. (Note: This includes `this` when the type of `*this` is an Aggregate.) For example, given `struct Agg { int i; char* p; };`, we expand:

- `f(Agg  a)` to `f(Agg  a, int  a.i, char*  a.p)`
- `f(Agg* a)` to `f(Agg* a, int* a.i, char** a.p)`
- `f(Agg& a)` to `f(Agg& a, int& a.i, char*& a.p)`

If the type of `*this` is not an Aggregate, treat each nonstatic data member `m` of `*this` as a parameter passed by value if in a constructor (and on function entry they are treated as described in §2.4.2) or by reference otherwise (and on function entry they are treated as described in this section).

For each Pointer parameter `p`, treat it as if it were additionally followed by a generated `deref__p` parameter of the same type as `DerefType(p)`. Also, if `p` is nullable, treat it as if it were additionally followed by a generated

`null__p` parameter of unspecified type and that is treated as a null by this analysis. These are treated as distinct parameters.

Apply recursively to nested Aggregates' by-value data members.

For an Aggregate return value returned by value (not by `*` or `&`), treat it as if it were multiple return values, one for each element `m` of `agg` that is a Pointer.

2.5.2 Create the input sets

The set `Input` includes:

- Every parameter of any type `X` that is passed by value, `X&`, `const X&`, `X&&`, or as the `X* this` parameter.
- Every reference (for `X&`, `const X&`, `X&&` parameters), treated as a Pointer.

Note that, for a Pointer type `P` and the function signature `f(const P&)`, `Input` contains the `P` and the `&` treated as a Pointer.

For example:

```
                                                    // Input set
void f(int a, int* b, int& c, int** d, int*& e);    // {a, b, c, deref_c, d, e, deref_e}
void m(T* this);                                    // {this, deref_this}
```

The sets `Pin` and `OIn` (Pointer and Owner Inputs) includes:

- Every parameter in `Input` that has type Pointer or Owner, respectively, except:
  - not if it is annotated `[[gsl::lifetime_out(param)]]`
  - not if it is the Owner or the reference of `Owner&&` or `const Owner&` (i.e., neither Owner, nor its `&` or `&&` treated as a Pointer)

Note Includes `this` so that member functions that return class references work. Includes the deref location of `this` for Pointer or Owner member functions, which allows Owner/Pointer member functions like `data()` to work.

For example:

```
void f(Owner o, Owner& ro, const Owner& cro)      // OIn = {ro, deref_ro}
```

The set `OIn_weak` includes (these are "weak rvalue magnets" because they can bind to rvalue arguments):

- The Owner and reference of every `const Owner&` parameter in `Input` (i.e., the Owner, and the `&` treated as a Pointer).

Notes This allows functions to return a Pointer to the Owner itself or to the data owned by the Owner.

The point of treating an Owner this way in the function inputs is that we only care about Owners that will outlive the call, and a reference to Owner is the idiomatic way to refer to an Owner in the scope of the caller. We strip the reference off because we only care about the Owner's identity at the call site. If in the future we add the concept of a SharedOwner, for example to model `shared_ptr` distinctly from `unique_ptr`, then it would make sense to consider SharedOwner parameters passed by value, since Pointers derived from them can still sensibly outlive the function call.

For example:

```
void f(Owner o, Owner& ro, const Owner& cro)      // OIn_weak = {cro, deref_cro}
```

The set Vin includes:

- All the elements of Input that are Values.

Note These rules exist to recognize C++'s idiomatic parameter passing conventions and reverse engineer the programmer's intent. In particular, part of the special treatment of this is to recognize that this is actually used as a reference; it is a const non-null pointer only for historical reasons, because its invention predated references.

2.5.3 Validate inputs: Enforce in caller, assume in callee

For convenience when writing actual explicit annotations is desired, we predefine these convenience variables:

```
namespace gsl {
    static inline void* const null    = nullptr;  // pset is {null}
    static inline void* const invalid = invalid;  // pset is {invalid}
    static inline int   const global  = 0;        // pset is {global}
}
```

A Pointer function input p ∈ Pin can be **lifetime-annotated** with a precondition [[gsl::pre(lifetime(p,{x}))]] where x ∈ Pin ∪ Oin ∪ {global} ∪ {null} ∪ {invalid} - {p}. This expresses that p must have the same lifetime as x. For example, void f(int* p, int* p2) [[gsl::pre(lifetime(p2, {p, null}))]]) expresses that p2 must point to the same thing as p, or null

On function entry, for every p ∈ Pin:

- If p is explicitly lifetime-annotated with x, then each call site enforces the precondition that argument(p) is a valid Pointer and pset(argument(p)) == pset(argument(x)) and both psets have a single entry, and in the callee on function entry set pset(p) = pset(x).
- Otherwise, if p is a generated element parameter agg__m and agg is a Pointer, then it behaves as if annotated [[gsl::pre(lifetime(agg_m,{agg}))]]. For example, given struct Agg { int* p; int i; }; and applying all the foregoing rules, the signature f(Agg* agg) is treated as if written f(Agg* agg, Agg deref__agg, int** agg__p, int* deref__agg__p, int* agg__i, int deref__agg__i) [[gsl::pre(lifetime(agg_p, {agg}))]] [[gsl::pre(lifetime(agg_i,{agg}))]].
- Otherwise, each call site enforces the precondition that argument(p) is valid, and in the callee on function entry set pset(p) = {deref__p,null__p}.

Then in all cases:

- If p is declared not_null, then each call site enforces the precondition that pset(argument(p)) must not contain null, and in the callee remove null (if present) from pset(p).
- Each call site enforces the precondition that no entry in pset(argument(p)) refers to a non-const global Owner, a local Owner being passed by reference to non-const to the same function call, or a local Owner mentioned in a pset of another Pointer being passed to the same function call; and in the callee on function entry record the additional entry pset(p:in) = pset(p).

On function entry, for every other v ∈ Vin:

- Each call site enforces the precondition that, if this function has a precondition that mentions `v`, `pset(argument(v))` must be valid. (Note: For Owners and Values, this verifies they have not been moved-from.)

Rule	**(lifetime.3)** Except for the `this` parameter of a Pointer constructor, it is an error to pass a Pointer as a function argument if it is invalid, if it violates the parameter's `[[gsl::lifetime` precondition, or if its `pset` refers to a non-`const` global Owner, a local Owner being passed by reference to non-`const` to the same function call, or a local Owner mentioned in a `pset` of another Pointer being passed to the same function call.
Note	Note that if a Pointer parameter `p`'s `pset(p)` refers to Owner inputs (e.g., `[[gsl::lifetime(p,{o})]] f(int* p, string& o)`, its input value can be invalidated as usual by modifications of Owner inputs in the function body (e.g., if the first line in `f`'s body modifies `o`, `p`'s initial value is invalidated).

2.5.4 Create the output sets

The set **`Output`** means the empty set for a constructor, otherwise includes:

- All return values.
- All parameters passed by lvalue reference to non-`const` (and is not considered to include the top-level Pointer, because the output is the pointee), except:
  - not if it is annotated `[[gsl::lifetime_in(param)]]`

Note	Parameters passed by `&&` are excluded because they are assumed to be moved/forwarded on entry or during the body of the function.

For example, `void f(int*&)` and `void f(int**)` are each considered to have a single function output of type `int*`, just the same as `int* f();`. For example, `void f(vector<deque<int>::iterator>::iterator)` is considered to have one function output of type `deque<int>::iterator`.

The set **`Pout`** is the set of function outputs that are Pointers.

The set **`Oout`** is the set of function outputs that are Owners.

2.5.5 Validate outputs: Enforce in callee, assume in caller

For every Owner `o` in `Oout`, at each call site reset `pset(argument(o)) = {o'}`.

Notes	Includes `this` so that calls to an Owner's non-`const` member function resets the Owner.

A Pointer function output `ret` can be **lifetime-annotated** with a precondition `[[gsl::pre(lifetime(ret,{s}))]]` or postcondition `[[gsl::post(lifetime(ret,{s}))]]` where $s \subseteq Pin \cup Oin \cup \{global\} \cup \{null\} \cup \{invalid\}$. To refer to the return value, use the function's name; to refer to a dereferenced Pointer to Pointer parameter `p`, `ret` is `*p`. This expresses that `ret` points to what one of `s` points to. For example, `int* f(int* p, int* p2, string o) [[gsl::post(lifetime(f,{p,o}))]]` expresses that the returned pointer points to either what `p` pointed to on input or data that `o` owned on input. For a set `s`, `s:in` means $\{x:in \mid x \in s\}$. For every $ret \in Pout$:

- In the callee on function exit, enforce that `pset(ret)` is substitutable for `pset(s:in)`.

Rule	**(lifetime.4)** It is an error on function exit for a Pointer function output to be invalid or to violate its postcondition.

Note Because only parameters and `global` can be referred to as postcondition requirements, typically this means `pset(ret)` ⊆ `pset(s:in)` or `pset(ret)` == `{global}`.

- Each call site uses the postcondition by setting `pset(argument(ret))` = `pset(argument(s))`.

If a `ret` ∈ `Pout` is not explicitly lifetime-annotated for a postcondition, then it is treated as if annotated as follows, where $\texttt{Pin}_{\texttt{ret}}$, $\texttt{Oin}_{\texttt{ret}}$, and $\texttt{Oin\_weak}_{\texttt{ret}}$ are the respective subsets of `Pin`, `Oin`, and `Oin_weak` for which `DerefType*` is the same type as, or convertible to, `DerefType(ret)*`:

- If $\texttt{Pin}_{\texttt{ret}}$ and $\texttt{Oin}_{\texttt{ret}}$ are not both empty, then as if declared `[[gsl::post(lifetime(ret,{`$\texttt{Pin}_{\texttt{ret}} \cup \texttt{Oin}_{\texttt{ret}}$`}))]]`. For example, for `int* f(int x, int* p, string_view sv, string& s, const string& s2`, the return value is as if declared `[[gsl::post(gsl::lifetime(ret,{p}))]]`. That is, by default the Pointer function output is assumed to be either a copy of one of its Pointer inputs pointing to the same type or to point to data owned by one of its Owner inputs owning the same type.
- Otherwise, if $\texttt{Oin\_weak}_{\texttt{ret}}$ is not empty, then as if declared `[[gsl::lifetime(ret,{`$\texttt{Oin\_weak}_{\texttt{ret}}$`})]]`. For example, for `char* f(int x, const string& s2)`, the return value is as if declared `[[gsl::post(gsl::lifetime(ret,{s2}))]]`. That is, the next fallback is that the Pointer function output is assumed to point to data owned by one of its weak rvalue magnet inputs of the same type (the actual argument is not necessarily an rvalue, and when it is an rvalue the caller will diagnose that).
- Otherwise, if `Pin` and `Oin` are not both empty, then as if declared `[[gsl::post(lifetime(ret,{`$\texttt{Pin} \cup \texttt{Oin}$`}))]]`. That is, the Pointer function output is assumed to be either a copy of one of its Pointer inputs or to point to data owned by one of its Owner inputs, without matching types.
- Otherwise, if `Oin_weak` is not empty, then as if declared `[[gsl::post(lifetime(ret,{`$\texttt{Oin\_weak}$`}))]]`. That is, the Pointer function output is assumed to point to data owned by one of its weak rvalue magnet inputs, without matching types.
- Otherwise, `[[gsl::post(lifetime(ret,{global}))]]`. That is, since there are no Pointer or Owner whose lifetimes are known at the call site, every Pointer function output is assumed to point to a global variable.

Notes The goal of these defaults is to minimize total annotation by recognizing that functions commonly return values derived from their inputs.

If a parameter is both an input and an output (e.g., a Pointer type's `this` parameter), do not allow it to bind to itself. This would cause `invalid` to flow through Pointer constructor calls.

After normal exit (not exceptional exit), each call site applies the postcondition that for every `ret` ∈ `Pout`, on function exit record `pset(argument(ret))` = `pset(ret)`.

2.5.6 Virtual functions

A virtual override implicitly inherits its `pre` and `post` conditions, may not add additional `pre` conditions, and may add additional `post` conditions and/or (redundantly) repeat base `post` conditions.

2.5.7 Examples

2.5.7.1 Example default output psets

For example:

```
                                    // The default pset for the return value is:
int* f();                           // {global}
```

```
int* f(int* p, int* p2);            // pset(p) ∪ pset(p2)
int* f(shared_ptr<X>& o);           // {o'}
int* f(vector<X>& o, string& o2);   // {o', o2'}
```

For example:

```
template<class T> int* f(const T&);

auto sp = make_shared<vector<int>>(100);

f(sp);                              // pass:   shared_ptr<vector<int>>& with pset=={sp}
                                    // return: pset = {sp'}

f(*sp);                             // pass:   vector<int>& with pset == {sp'}
                                    // return: pset = {sp''}

f(sp->begin());                     // pass:   vector<int>::iterator& with pset=={sp''}
                                    // return: pset = {sp'''}

f((*sp)[5]);                        // pass:   int& with pset == {sp''}
                                    // return: pset = {sp'''}
```

2.5.7.2 *Example: Owners*

For example, consider `shared_ptr<int>::get()`, where the only argument is the `this` pointer to an Owner:

```
// godbolt.org/z/1xAGOH

auto sp = make_shared<int>(0);
int* p = sp.get();                  // pset(p) = pset(sp.get()) which is {sp'}
*p = 42;                            // ok
sp = make_shared<int>(1);           // KILL(sp') → pset(p) = {invalid}
*p = 42;                            // ERROR
```

2.5.7.3 *Example: `std::move` and `std::forward`*

Note The parameters of `std::move` and `std::forward` should be annotated `[[gsl::lifetime_const]]`. See §2.5.7.10 Note.

Consider a function template like `std::move` that returns a parameter. (`std::forward` is handled similarly.)

```
template<class T> constexpr typename // if T is not an Owner,
std::remove_reference<T>::type&&     //   pset(ret) = pset(t) = {t}
move([[lifetime_const]] T&& t)
    { return t; }                   // ok, {t} is substitutable for {t}
```

Calling code can reason on the lifetime that flows through `move`:

```
string_view s1, s2;
s1 = move(s2);                      // ok, pset(s1) = pset(ret) = pset(arg) = pset(s2)
```

2.5.7.4 *Example: `std::min` and `std::max`*

Consider `std::min`, which returns one of its input references. (`std::max` is handled similarly.)

```
template<class T>                   // if T is not an Owner
const T&                            //   pset(ret) = {a,b},
min(const T& a, const T& b) {       //   pset(a) = {a}, pset(b) = {b}
    return b < a
```

```
            ? b                     // ok, {b} is substitutable for {a,b}
            : a;                    // ok, {a} is substitutable for {a,b}
}
```

In calling code, this prevents known lifetime errors, including improving existing C++ code. For example, here is a problem reported by a number of people including Andrei Alexandrescu: Because `std::min` returns a reference, if a call to `min(x,y)` might change under maintenance to `min(x,y+1)` we could get a dangling reference if `min` returns a reference to `y+1`, which would be invalidated when the temporary is destroyed after the end of the call expression in which it appears:

```
// godbolt.org/z/1C4t8m

int main() {
    auto x=10, y=2;
    auto& good = min(x,y);          // ok, pset(good) == {x,y}
    cout << good;                   // ok, 2

    auto& bad = min(x,y+1)          // A: IN: pset(arg1)=={x},
                                    //        pset(arg2)=={temp(y+1)}
                                    //    min() returns temp2
                                    //    OUT: pset(temp2) = {x,temp}
                          ;         //    KILL(temp) → pset(temp2) = {invalid}
                                    // ERROR (lifetime.2), initializing bad as invalid

    cout << bad;                    // ERROR, bad initialized as invalid on line A
}
```

In normal C++, this code compiles but has undefined behavior.

> **Note** In practice, on the three major compilers (gcc, VC++, clang) this code does not crash and appears to work. That's because one manifestation of "undefined behavior" can be "happens to do what you expect." Nevertheless, this is undefined and its appearance of working makes the error more pernicious, not less so; slightly different examples will visibly break.

If this code had instead used `max` instead of `min`, therefore returning a reference to the first argument, there would have been no undefined behavior in normal C++ but these rules (I think rightly) would still reject it as statically unsound, having data-dependent safety.

2.5.7.5 Example: Meyers Singleton

Consider the classic Meyers Singleton, where the default returned `pset` is `{global}` because there are no Pointer or Owner parameters:

```
// godbolt.org/z/sKIltx

widget& get_widget() {              // pset(ret) = {global}
    static widget w;
    return w;                       // ok, {global} is substitutable for {global}
}
```

2.5.7.6 Example: Return Pointer that must be invalid (e.g., to local)

In a function body, it is a `lifetime` error to return a pointer that must be invalid, either as a return value or through an inout/out parameter.

```
// godbolt.org/z/vC5g6G

int* f() {
    int i = 0;
    return &i;                  // pset(&i) = {i}, then KILL(i) → pset(ret) = {invalid}
                                // ERROR, {invalid} is not substitutable for {global}
}

void g(int*& pi) {
    int i = 0;
    pi = &i;                    // pset(pi) = {i}
}                               // KILL(i) → pset(pi) = {invalid}
                                // ERROR, pi is non-const& so pset(pi) checked on exit,
                                // and {invalid} is not substitutable for {global}
```

2.5.7.7 Example: Return indirection that may be invalid (e.g., to local)

In a function body, it is a `lifetime` error to return a pointer that could be invalid, either as a return value or through an inout/out parameter.

```
// godbolt.org/z/53qeAD

int* f(int* pi) {
    int i = 0;
    return cond ? pi : &i;      // pset(expr)={pi,i}, KILL(i) → pset(expr)={invalid}
                                // ERROR, {invalid} is not subtitutable for {pi}
}

void g(int*& pi, int* pi2) {
    int i = 0;
    pi = cond ? pi2 : &i;
}                               // KILL(i) → pset(pi) = {invalid}
                                // ERROR, pi is non-const& so pset(pi) checked on exit
                                // and {invalid} is not substitutable for {pi2}
```

2.5.7.8 Example: Calling a function that returns an indirection

For example:

```
// godbolt.org/z/-exrN7

int* f();                       // pset(ret) = {global}

int main() {
    int* p = f();               // pset(p) = {global}
    *p = 42;                    // ok
}
```

2.5.7.9 Example: Passing indirections

Non-`const` Pointer parameters can be reused as usual in the function body:

```
// In function bodies
//
void f(int* p) {                    // pset(p) = {p}
    p = something_else;             // ok, now points to something else
    // ...
}
void g(shared_ptr<int>& s, int* p) { // pset(s) = {s'}, pset(p) = {p}
    s = something_else;             // KILL(s') → no local effect, does not kill p
    // ...                          // pset(p) == {p}, unchanged
}

// At call sites
//
int gi = 0;
shared_ptr<int> gsp = make_shared<int>();
int main() {
    // passing global and local objects
    f(&gi);                 // ok, pset(arg) == {global}
    int i = 0;
    f(&i);                  // ok, pset(arg) == {i}
    f(gsp.get());           // ERROR (lifetime.3), pset(arg) == {gsp'}, gsp is global
    auto sp = gsp;
    f(sp.get());            // ok, pset(arg) == {sp'}, sp's address has not been taken
    g(sp, sp.get());        // ERROR (lifetime.3), pset(arg2) == {sp'}, sp being passed
    g(gsp, sp.get());       // ok, pset(arg2) == {sp'}, sp not being passed
}
```

Note This diagnoses the #1 *correctness* error using smart pointers, and with a clear message highlighting the key variable names. (The #1 *performance* error using smart pointers is covered under the `foundation` coding guidelines profile, which diagnoses needlessly passing smart pointer copies.)

2.5.7.10 Example: Explicitly overriding defaults

Sometimes you want to override the defaults. For example, consider two standard container member functions:

- The insert-with-hint `insert(iter,t)` assumes that the iterator is into `this` container, which is not the default (and would not be allowed by the earlier rule that `iter` could be invalidated by this `insert`). We can express this using `[[gsl::pre(lifetime(iter,{this}))]]`.
- The range-based insert `insert(iter1,iter2)` assumes that the passed iterators are not into `this` container, which is the default. It also assumes that `iter1` and `iter2` have the same lifetime, which is not the default; we can express this using `[[gsl::pre(lifetime(iter2,{iter1}))]]`.

Result:

```
template<class Key, class T, /*...etc...*/>
class map {
public:
    iterator insert(const_iterator pos,
                    const value_type&)  [[gsl::pre(pos, lifetime({this}))]];

    template <class InputIterator>
    void insert(InputIterator first,
                InputIterator last)  [[gsl::pre(lifetime(last, {first}))]];

    // ... more insert overloads and other functions ...
};

map<int,string> m = {{1,"one"}, {2,"two"}}, m2;

m.insert(m2.begin(), {3,"three"});   // ERROR, pset(m2.begin()) != {m'}

m.insert(m.begin(), {3,"three"});    // ok, pset(m.begin()) == {m'}

m.insert(m.begin(), m.end());        // ERROR, psets=={m'}, and m is passed as non-const&

m.insert(m2.begin(), m.end());       // ERROR, psets are not equal

m.insert(m2.begin(), m2.end());      // ok, pset == {m2'}, and m2 is not passed to insert
```

Notes This statically diagnoses several common classes of STL iterator bugs.

There are patterns, notably iterator pairs. For example, every standard library function that has Pointer (iterator) parameters named `first` and `last` should annotate last `[[gsl::pre(lifetime(last,{first}))]]`.

Non-`const` `begin()` and `end()`, non-member `std::begin()` and `std::end()`, and the parameters of `std::move` and `std::forward`, should be annotated `[[gsl::lifetime_const]]`.

For containers where insertion (if available) does not invalidate iterators, likely all non-`const` member functions except `erase` should be additionally annotated `[[gsl::lifetime_const]]`. This includes `array`, `forward_list`, `list`, `set`, `multiset`, `map`, `multimap`, `unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap`.

The `this` parameter of a constructor or assignment operator is always treated as if annotated `[[gsl::lifetime_out]]`.

We should make a list of all the annotations we want for standard library functions, and then for convenient adoption without modifying existing standard library headers, treat those standard library functions as if they were so annotated (as in §2.1 for Owner and Pointer types).

2.5.7.11 Example: Pointer swap

Another example of overriding the defaults is for `std::swap` for Pointers.

```
template<Pointer T>                 // assumes a Pointer concept
void swap(T& a, T& b)
    [[gsl::post(lifetime(deref_a,{deref_b}))]] [[gsl::post(lifetime(deref_b,{deref_a}))]]
{ T tmp = a; a = b; b = tmp; }      // no need for std::move, Pointers are copyable
```

2.5.7.12 Example: Lifetime-`const`

The lifetime-annotation `[[gsl::lifetime_const]]` on a member function of an Owner type means to treat `this` Owner object argument as if declared `const` for the purpose of lifetime invalidation, and on a reference parameter of Owner type means to treat the argument as if declared `const` for the purpose of lifetime invalidation.

> **Note** We don't have a use case yet for going the other way, but we could also provide the opposite annotation `[[gsl::lifetime_mutable]]` if we ever find an example where a `const` Owner parameter `o` may actually invalidate data owned by `o` (which seems weird).

For example, given a `vector<T>` consider two non-`const` member functions, one of which invalidates pointers/iterators into the vector and one of which doesn't:

```
void push_back(const T& t) {                    // can move storage
    if (/*need to grow*/) {
        // ...
        data = /* some new buffer, and copy old data */;
        // ...
    }
    // ...
}

T& operator[](size_t n) [[gsl::lifetime_const]] { // won't move storage
    return data[n];
}
```

We benefit by annotating `operator[]` to treat it as though it were `const`, because even though it is a non-`const` operation, `operator[]` does not perform non-`const` operations on its structure—and therefore does not invalidate references previously obtained from `operator[]` (or equivalently `front()`, etc.). If this is communicated to the caller, then a caller that has a Pointer `int* pi` referring to an `int` inside a `vector<int> v` can know that calling `v[0]` does not invalidate `pi`, whereas calling `v.push_back(42);` does invalidate `pi`.

> **Note** It is debatable whether STL made the right design decision in not distinguishing structure from contents—that is, failing to treat the container's own structure distinctly from the contained elements. But STL isn't alone here, and many C++ libraries have followed such a convention; the lifetime annotation provides a way to tactically add the arguably "missing" `const`. The STL might be a better library if it treated `vector<int>` and `vector<const int>` distinctly; that is, the constness of the elements is distinct from the constness of the container. Then `vector` would mark `operator[]` as a `const` function; and a `vector<int>::iterator` could be allowed to convert to a `vector<const int>::iterator`, avoiding the need for the `const_iterator` oddity. Something to think about for STLv2.
>
> Granted, this complaint does not apply equally to `map`, which we consider next.

Similarly, given a node-based container `map<T>`, consider two non-`const` member functions, one of which invalidates pointers/iterators into the map and one of which doesn't:

```
/*…*/ erase(const T& t) {                              // can invalidate
    // ...
}
```

```
/*…*/ insert(const T& t) [[gsl::lifetime_const]] {      // won't invalidate
    return data[n];
}
```

Note This is not the same structure-vs-contents situation as `vector`, but rather a node-based lifetime semantics situation. However, the approach works the same way for lifetime invalidation purposes; by saying "consider `insert` as a const operation for lifetime invalidation purposes" we express the correct semantics, that `erase` is a function that should be assumed to invalidate pointers and iterators into the container, but `insert` is not.

In both cases, we will flag potential false positives: For `vector`, when `push_back` does not really invalidate because of a careful earlier `reserve` we diagnose invalidation anyway, but such code is arguably has data-dependent correctness so we feel correct in diagnosing it. For `map`, when `erase` removes only one node (or a few) we diagnose invalidation of all pointers and iterators into the map, not just ones to those nodes; this is a stronger conservatism and source of false positives.

2.6 Additional examples

2.6.1 From StackOverflow

Back in 2015, an hour before I was about to add a usage example here of how the above rules and implementation of `unique_ptr` detect lifetime errors, the following was posted on StackOverflow, so let's use this example.

```
unique_ptr<A> myFun()
{
    unique_ptr<A> pa(new A());
    return pa;
}

const A& rA = *myFun();
```

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

Under this `lifetime` profile, the rules mechanically diagnose the problem and give the answer. The convention in this paper is to diagnose the problem at the point the code attempts to use the invalidated local pointer or reference, and so we have the ability to diagnose:

```
// godbolt.org/z/4G-8H-
const A& rA = *myFun();             // A: ERROR, rA is unusable, initialized invalid
                                    // reference (invalidated by destruction of the
                                    // temporary unique_ptr returned from myFun)
use(rA);                            // ERROR, rA initialized as invalid on line A
```

In the first line, `myFun` returns a temporary `unique_ptr` (call it *`temp_up`*), then unary `*` returns a temporary reference *`temp_ref`* with `pset(`*`temp_ref`*`) = `*`temp_up'`, then `temp_up` is destroyed which implies `KILL(`*`temp_up`*`)` → `pset(`*`temp_ref`*`) = invalid`, and finally that is copied to initialize `pset(rA) = invalid`.

As noted earlier, we diagnose the error at the creation of the unusable reference, since references cannot be reseated and so this initialization is just always nonsense.

The poster added a coda:

Note: if I assign the return of `myFun` to a named `unique_ptr` variable before dereferencing it, it works fine.

Indeed it does:

```
auto local = myFun();               // ok, local assumes ownership
const A& rA = *local;               // ok, pset(rA) = {local}
use(rA);                            // ok, we know that local is keeping rA alive
```

2.6.2 From P0936R0 (Richard Smith, Nicolai Josuttis)

The examples in [P0936R0] are diagnosed by this proposal without annotation. Note that we deliberately diagnose only when attempting to actually use an invalid Pointer, because it is common for a Pointer to be temporarily invalidated and then set to a valid value before being used again.

2.6.2.1 Example: `string_view` initialized from `string` literal

This example is correctly diagnosed by this proposal without annotation. (`basic_string` meets the Container requirements and so is automatically recognized as an Owner type, and `basic_string_view` meets the Range requirements and so is automatically recognized as a Pointer type.)

```
// godbolt.org/z/dymV_C

std::string_view s = "foo"s;        // A
s[0];                               // ERROR (lifetime.3): ‘s’ was invalidated when
                                    // temporary ‘"foo"s’ was destroyed (line A)
```

2.6.2.2 Example: `string_view` initialized from temporary `string`

This example is correctly diagnosed by this proposal without annotation. (`basic_string` meets the Container requirements and so is automatically recognized as an Owner type, and `basic_string_view` meets the Range requirements and so is automatically recognized as a Pointer type.)

```
// godbolt.org/z/0aeHPp

std::string operator+ (std::string_view s1, std::string_view s2) {
    return std::string{s1} + std::string{s2};
}

std::string_view sv = “hi”;
sv = sv + sv;                       // A
sv[0];                              // ERROR (lifetime.3): ‘sv’ was invalidated when
                                    // temporary ‘"foo"s’ was destroyed (line A)
```

2.6.2.3 Example: `string_view` initialized from temporary `string` via template

This example is correctly diagnosed by this proposal without annotation. (`basic_string` meets the Container requirements and so is automatically recognized as an Owner type, and `basic_string_view` meets the Range requirements and so is automatically recognized as a Pointer type.)

Note that the diagnosis happens even earlier than P0936 asks for, because we notice the dangling pointer during instantiation of template `add<std::string_view>`.

```
// godbolt.org/z/IsPG8P

std::string operator+ (std::string_view s1, std::string_view s2) {
    return std::string{s1} + std::string{s2};
}
```

```
template<typename T>                    // for T == string_view, pset(ret) = {x1,x2},
T add(T x1, T x2) {                     //   pset(x1) = {x1}, pset(x2) = {x2}
    return x1 + x2 ;                    // A: ERROR (lifetime.4): {tmp'} was invalidated
}                                       //   when temporary 'x1 + x2' was destroyed (line A)

sv = add(sv, sv);
```

For a variation of this example, see also `std::min` in §2.5.7.4.

2.6.2.4 Example: Returned reference to parameter

This example is correctly diagnosed by this proposal without annotation (and without error, the code is fine).

```
// godbolt.org/z/wncC9a

struct X { int a, b; };

int& f(X& x) { return x.a; }            // ok, pset(ret) == pset(x)
```

For a variation of this example, see also `std::move` in §2.5.7.3

2.6.2.5 Example: Returned reference to temporary argument

This example is correctly diagnosed by this proposal without annotation. (`basic_string` meets the Container requirements and so is automatically recognized as an Owner type.)

```
// godbolt.org/z/AqXDYp

char& c = std::string{"hello my pretty long string"}[0];
cout << c;                              // A: ERROR (lifetime.2): illegal to initialize a
                                        // reference 'c' with an invalid pointer, pointer
                                        // was invalidated with temporary string was
                                        // destroyed (line A)
```

2.6.2.6 Example: Reversed ranges

This example is correctly diagnosed by this proposal without annotation. (This assumes `reversed_range` meets the Range requirements and so is automatically recognized as a Pointer type.)

```
// Use either of the following definitions for reversed() – both definitions are valid

template<Range R>                       // assuming R satisfies Range and so is a Pointer,
reversed_range reversed(R&& r) {        //   pset(ret) = pset(r) = {r}
    return reversed_range{r};           // ok, pset(ret) == pset(r)
}

template<Range R>                       // assuming R satisfies Range and so is a Pointer,
reversed_range reversed(R r) {          //   pset(ret) = pset(r) = {r}
    return reversed_range{r};           // ok, pset(ret) == pset(r)
}

// Now do a range-for, which expands per [stmt.ranged] (see §2.4.9)

for (auto x : reversed(make_vector()))
                                        // A: ERROR (lifetime.1): __begin was invalidated
                                        // when temporary 'make_vector()' was destroyed
                                        // (line A)
```

2.6.2.7 Example: "A classical bug"

This example is correctly diagnosed by this proposal without annotation.

> **Note** Because the search values happen to be also Owners, the default lifetime annotation `{m',key',defvalue'}` is stricter than the one the user would likely have written by hand, namely `{m',key,defvalue}` or `{key,defvalue}`. That's okay in the function body because `{key,defvalue}` and `{m',key,defvalue}` are substitutable for `{m',key',defvalue'}`, and it's okay to diagnose this call site because the destruction of the temporary `defvalue` also invalidates `defvalue'`.)

```
// godbolt.org/z/BvP4w6

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'}

std::map<std::string, std::string> myMap;
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)
```

For a variation of this example, see also `std::min` in §2.5.7.4.

2.6.3 From Ryan McDougall

2.6.3.1 Example: Eigen expression templates (reported by Chris Patton)

This example is correctly diagnosed by this proposal, with light annotation: The Eigen library should annotate its expression template `Cwise*Op` types as a `[[gsl::Owner]]`. (The specific type being used in this example is `CwiseBinaryOp`.)

> **Note** If we can identify a reliable heuristic for automatically identifying expression template 'reference' types, we can add that to the list of automatically recognized Pointer types in §2.1 and then this type would not need to be annotated.

```
class Type {
  Eigen::Vector3d calculateVector(); // Matrix<double,3,1>
};

auto point =                        // A: captures an expression template object
      (src.calculateVector() + dst.calculateVector()) * 0.5;

Eigen::Vector3d point2 = point;     // ERROR (lifetime.3): 'point' was invalidated when
                                    // temporary '...' was destroyed (line A)

const Eigen::Vector3d point3 =      // no dangling references
      (src.calculateVector() + dst.calculateVector()) * 0.5;
```

2.6.3.2 Example: Parameter use-after-move

This example is correctly diagnosed by this proposal without annotation.

```
class Foo {
  Foo(Bar&& bar)
```

```
      : bar_(std::move(bar)) {        // A
        bar.set_safety_critical(true); // oops, typo: meant bar_
                                       // ERROR (lifetime.1): 'bar' was invalidated when
                                       // moved from (line A)
    }
    // ...
};
```

2.6.3.3 *Example: `Expected` use-after-move*

This example is correctly diagnosed by this proposal the annotation of the `Expected` type as a `[[gsl::Owner]]`.

The API for this `Excepted` type doesn't make clear whether the user should use the value in-place (simpler), or move it out before use (cleaner), so the result is you get users doing both:

```
template <typename T>
class Expected {
    T& get() { return value_; }
    T value_;
    // ...
};

void take_it(Baz&&);

Expected<Baz> result = baz();

do_some_work(result.get());          // pset(tmp) = {result'}

take_it(std::move(result.get()));    // A: added recently in a rush
                                     // pset(tmp) = {result'}, passed to rref param,
                                     // → KILL(result') and pset(result) = {invalid}

do_other_work(result.get());         // ERROR (lifetime.1): 'result' was invalidated when
                                     // its data was moved from (line A)
```

2.7 Diagnostic sensitivity options

To allow the user to opt into stricter and noisier diagnostic cases from a legacy project to a Lifetime-checked project, provide a set of diagnostic options that includes the ability to opt into at least the following groups of possibly-noisy diagnostics:

1. Diagnosing return of "the wrong null" in an example like the following, where the analysis cannot always easily tell that a general "null" is being returned only if a parameter is null:

```
int* f(int* p) {                     // pset(p) initially {deref__p, null__p}
    if(!p) return p;                 // ok, returns {null__p}
    if(!p) return nullptr;           // potentially noisy warning: returns {null}
}
```

2. Diagnostics from assuming Pointer parameters can be null by default. (If not selected, Pointer parameters are not assumed to be possibly null in the function body, but are also not enforced to be not-null in the function caller.)

3. Diagnostics from inferred pre/post conditions, not just explicitly written pre/post conditions.

4. Diagnostics treating Pointer-to-Pointer parameters as outputs, not just Pointer& parameters.

5. Diagnostics from inferred Pointer/Owner annotations, not just explicitly written Pointer/Owner annotations.

3 Bibliography

[Brooks 1975] F. Brooks. *The Mythical Man-Month* (Addison-Wesley, 1975).

[C++ Core Guidelines] B. Stroustrup and H. Sutter, eds. *The C++ Core Guidelines* (GitHub, 2018).

[P0936R0] R. Smith and N. Josuttis. "Bind returned/initialized objects to the lifetime of parameters, Rev. 0" (WG21 paper, 2018-02-12).

[Stroustrup 1994] B. Stroustrup. *The Design and Evolution of C++* (Addison-Wesley, 1994).

4 Future notes

4.1 Nullable owners

Should we add a NullableOwner concept? For example, containers like `vector` are always `{o'}` even if default-constructed, whereas `unique_ptr` wants to be nullable if default-constructed. Perhaps a default detection heuristic would be for owners that have `*` and `->` (i.e., offer explicit access to contained data) be assumed to be nullable by default, which would include smart pointers but exclude containers.

4.2 Namespace-scope static variables (aka "global")

The primary thing we teach people about such global variables is "avoid them." This analysis focuses on locals, and only needs enough support for globals to reason about the locals, such as when copying from a global Pointer to a local one.

Trying to reason about global variables is inherently in tension with local analysis. For example, any function call could modify a global, and we don't want to have to compute and communicate the set of global variables that a caller and callee might have in common in order to treat them as additional parameters.

But then there are…

4.3 Function local static variables

… which have "global" lifetime but only local visibility. These, we could reason about. If we wanted to, here is an extension we could contemplate for function local statics.

Process the function twice, "first time through function" and "not-first time through the function" separately, similarly to what we do for loops. For a function local static Pointer variable `p`:

- In the "first time" analysis, `p` is treated exactly like a local. In particular, we set the initial `pset(p)` from its initializer.
- In the "non-first time" analysis, we ignore the initializer and instead set the initial `pset(p) = {global}` (valid for lifetime of function) if `p`'s type is `not_null`, otherwise set `pset(p) = {null,global}`. (Allowing null will cause warnings in some existing code, to silence the warning just make the type `not_null` or `assert(p)`.)

4.4 Pointers to member pointers

Pointers to members whose pointed-at type is itself a pointer type are just a variation on Pointers to Pointers. However, they are rare enough (and discouraged enough) that I would like to put them in the "future" section and see whether we can avoid having to acknowledge and support them. In case we add it, this section documents the design.

A pointer to member can treated the same as a Pointer to Pointer. For a Pointer to Pointer `pp` that can point to multiple things (e.g., `int** pp = rand()%2 ? &p1 : &p2;` so that `pset(pp)` is `{p1,p2}`), the dereferenced Pointer has the union of the pointees' psets (e.g., `pset(*p) == pset(p1) ∪ pset(p2)`). Similarly, when following a pointer to member whose type is a pointer, and if there is more than one matching data member, take the union of the psets of the members. For example:

```
struct Agg { int* p1; int* p2; };

Agg a;                              // pset(p1) = pset(p2) = {invalid}
int i;
a.p1 = &i;                          // pset(p1) = {i}

int* Agg::*memptr = something();    // could point to .p1 or .p2

cout << *(a.*memptr);               // error, pset is {i,invalid}
```